# Factored Task and Motion Planning with Combined Optimization, Sampling and Learning

by **Joaquim Ortiz-Haro, M.Sc.**

A thesis submitted for the degree of

Doctor of Natural Sciences (Dr. rer. nat.)

**TU Berlin**

Faculty IV - Electrical Engineering and Computer Science

Learning and Intelligent Systems

Berlin, November 2023

---

**Ph.D. Advisor**: Prof. Dr. Marc Toussaint (TU Berlin)
**Doctoral Committee**:
Chair: Prof. Dr. Marc Alexa (TU Berlin)
Reviewer: Prof. Dr. Marc Toussaint (TU Berlin)
Reviewer: Prof. Dr. Georg Martius (University of Tübingen)
Reviewer: Prof. Dr. Tomás Lozano-Pérez (MIT)

# Contents

# Acknowledgements

I would like to express my deepest gratitude to my supervisor, Marc Toussaint. Thank you for your guidance and encouragement throughout my Ph.D. Your knowledge and passion for robotics research, mastering both theory and practice, have been an inspiration to me.

I am very grateful to the members of the doctoral committee, Prof. Georg Martius and Prof. Tomás Lozano, whose research I have followed closely and admired, for reviewing and evaluating my work. I am confident that your feedback and evaluation will significantly contribute to the refinement of this thesis. Additionally, I would like to express my gratitude to Prof. Marc Alexa for serving as the chair of my doctoral committee.

A heartfelt thanks to Erez Karpas and Michael Katz; collaborating with you has been a great learning experience and a pleasure, and I really enjoyed the short research stay at Technion in Haifa. My sincere appreciation goes to Prof. Georg Martius and Prof. David Remy for their advice on the Thesis Advisory Committee of the IMPRS-IS program.

I have been lucky to work in a group of great colleagues and friends. Danny, Ingmar, Valentin, Svetlana, Pia, Akmaral, Khaled, Wolfgang, Ilaria, Oz, Andreas, and Jung-Su, I have learned a lot from each one of you. Collaborating on research projects has been a pleasure, and you have created a pleasant and stimulating work environment. I look forward to meeting you again at robotics conferences all around the world!

Because research often requires time to step away and enjoy other facets of life, I would like to thank all the wonderful friends I met in Berlin. I had a great time here with all of you, and I look forward to keeping in touch and seeing you again.

And finally, but most importantly, I would like to thank all my family, whose support has been fundamental, especially when I needed it the most. Your love, support, and belief in me have been my pillars of strength. I dedicate this achievement to you, with immense love and gratitude. Thank you for always being there.

# Abstract

Modern robots excel at performing simple and repetitive tasks in controlled environments; however, future applications, such as robotic construction and assistance, will require long-term planning of physical interactions.

These problems can be formulated as Task and Motion Planning (TAMP). The goal is to find how the robot should move to solve complex tasks requiring multiple interactions with objects in the environment, such as building furniture or cleaning and organizing the kitchen. However, TAMP is notoriously difficult to solve because it involves a tight combination of task planning and motion planning, considering geometric and physical constraints.

In this thesis, we aim to improve the performance of TAMP algorithms from three complementary perspectives. First, we investigate the integration of discrete task planning with continuous trajectory optimization. Our main contribution is a conflict-based solver that automatically discovers why a task plan might fail when considering the constraints of the physical world. This information is then fed back into the task planner, resulting in an efficient, bidirectional, and intuitive interface between task and motion, capable of solving TAMP problems with multiple objects, robots, and tight physical constraints.

Traditionally, there have been two competing approaches to solving TAMP problems: sample-based and optimization-based methods. In the second part, we first illustrate that, given the wide range of tasks and environments within TAMP, neither sampling nor optimization is superior in all settings. To combine the strengths of both approaches, we have designed meta-solvers for TAMP, adaptive solvers that automatically select which algorithms and computations to use and how to best decompose each problem to find a solution faster.

A third promising direction to improve TAMP algorithms is to learn from previous solutions to similar problems. In the third part, we combine deep learning architectures with model-based reasoning to accelerate computations within our TAMP solver. Specifically, we target infeasibility detection and nonlinear optimization, focusing on generalization, accuracy, compute time, and data efficiency.

At the core of our contributions is a refined, factored representation of the trajectory optimization problems inside TAMP. This structure not only facilitates more efficient planning, encoding of geometric infeasibility, and meta-reasoning but also provides better generalization in neural architectures.

# Zusammenfassung

Moderne Roboter sind hervorragend darin, einfache und wiederholte Aufgaben in kontrollierten Umgebungen auszuführen. Zukünftige Anwendungen, wie die robotergestützte Roboterassistenz und das robotergestützte Bauen, werden jedoch eine langfristige Planung physischer Interaktionen erfordern.

Diese Probleme können als Aufgaben- und Bewegungsplanung (Task and Motion Planning, TAMP) formuliert werden. Dabei ist das Ziel, lange Abfolgen von Roboteraktionen zu finden, um komplexe Aufgaben zu lösen, die mehrere Interaktionen mit der Umgebung erfordern und dabei geometrische und physische Beschränkungen berücksichtigen. TAMP ist bekannterweise sehr schwer zu lösen, da es eine enge Kombination von Aufgabenplanung und Bewegungsplanung erfordert.

In dieser Arbeit zielen wir darauf ab, die Leistung von TAMP-Algorithmen aus drei komplementären Perspektiven zu verbessern. Zuerst untersuchen wir, wie man Aufgabenplaner mit Trajektorienoptimierung integriert. Unser Hauptbeitrag ist ein neues Framework, das automatisch entdeckt und kodiert, warum ein Aufgabenplan angesichts der Beschränkungen der physischen Welt scheitern könnte. Dies führt zu einer effizienten und intuitiven Integration von Aufgaben- und Bewegungsplanung.

Traditionell gab es zwei konkurrierende Ansätze, um TAMP-Probleme zu lösen: Stichprobenbasierte und optimierungsbasierte Methoden. Im zweiten Teil zeigen wir zuerst, dass angesichts der Vielzahl von Aufgaben und Umgebungen innerhalb der TAMP weder Stichproben noch Optimierung in allen Situationen überlegen sind. Um die Stärken beider Ansätze zu kombinieren, haben wir Meta-Lösungsalgorithmen für TAMP entwickelt: adaptive Solver, die automatisch auswählen, welche Algorithmen und Berechnungen verwendet werden sollen und wie jedes Problem am besten zerlegt werden kann, um eine Lösung schneller zu finden.

Ein dritter vielversprechender Ansatz zur Verbesserung der TAMP-Algorithmen besteht darin, von früheren Lösungen ähnlicher Probleme zu lernen. Im dritten Abschnitt schlagen wir zwei verschiedene neuronale Architekturen vor, um teure Berechnungen in unserem Lösungsalgorithmus, nämlich Unlösbarkeitserkennung und nichtlineare Optimierung, mit Hilfe von Deep-Learning-Methoden zu beschleunigen.

Im Kern unserer Beiträge steht eine verfeinerte, faktorisierte Darstellung der Trajektorienoptimierungsprobleme innerhalb von TAMP. Diese Struktur erleichtert nicht nur eine effizientere Planung und Kodierung geometrische Unlösbarkeiten, sondern ermöglicht auch das Schlussfolgern über potenzielle Rechenoperationen und bietet eine bessere Generalisierung in neuronalen Architekturen.

# Resumen

Los robots modernos sobresalen en la realización de tareas simples y repetitivas en entornos controlados; sin embargo, las aplicaciones futuras, como la construcción y la asistencia robótica, requerirán una planificación autónoma de diversas interacciones físicas.

Estos problemas se pueden formular como Planificación de Tareas y Movimientos (TAMP, por sus siglas en inglés). El objetivo es encontrar cómo debe moverse el robot para resolver tareas complejas que requieren múltiples interacciones con los objetos del entorno, como por ejemplo, montar un mueble o limpiar y recoger la cocina. Sin embargo, la resolución de TAMP es notoriamente difícil porque implica una combinación estrecha de planificación de tareas y planificación de movimientos, considerando restricciones geométricas y físicas.

En esta tesis, nuestro objetivo es mejorar el rendimiento de los algoritmos TAMP desde tres perspectivas complementarias. Primero, investigamos la integración de la planificación de tareas discretas con la optimización continua de trayectorias. Nuestra principal contribución es un algoritmo que descubre automáticamente por qué un plan de tareas podría fallar al considerar las restricciones del mundo físico. Esta información retroalimenta al planificador de tareas, resultando en una interfaz bidireccional e intuitiva entre tareas y movimientos, capaz de resolver problemas con múltiples objetos, robots y restricciones físicas complejas.

Tradicionalmente, ha habido dos enfoques competitivos para resolver problemas TAMP: métodos basados en muestreo y métodos basados en optimización. En la segunda parte, primero ilustramos que, dada la amplia variedad de tareas y entornos dentro de TAMP, ni el muestreo ni la optimización son superiores en todos los escenarios. Para combinar las fortalezas de ambos enfoques, hemos diseñado meta-algoritmos para TAMP, algoritmos adaptativos que seleccionan automáticamente qué algoritmos y cálculos usar y cómo descomponer mejor cada problema para encontrar una solución más rápidamente.

Una tercera dirección prometedora para mejorar los algoritmos TAMP es aprender de soluciones previas a problemas similares. En la tercera parte, combinamos arquitecturas de aprendizaje profundo con razonamiento basado en modelos para acelerar los cálculos dentro de nuestros algoritmos. Específicamente, nos enfocamos en la detección de qué restricciones físicas son inviables y en la optimización no lineal, centrándonos en la generalización, precisión, tiempo de ejecución y la eficiencia de datos.

En el núcleo de nuestras contribuciones se encuentra una representación refinada, desglosada y fragmentada de los problemas de optimización de trayectorias dentro de TAMP. Esta estructura no solo facilita una planificación más eficiente, análisis de restricciones geométricas y meta-algoritmos, sino que también proporciona una mejor generalización en arquitecturas neuronales.

# Resum

Els robots moderns excel·leixen en la realització de tasques simples i repetitives en entorns controlats; tanmateix, les aplicacions futures, com la construcció i l'assistència robòtica, requeriran una planificació autònoma de diverses interaccions físiques.

Aquests problemes es poden formular com a Planificació de Tasques i Moviments (TAMP, per les seves sigles en anglès). L'objectiu és trobar com s'ha de moure el robot per resoldre tasques complexes que requereixen múltiples interaccions amb els objectes de l'entorn, com ara muntar un moble o netejar i recollir la cuina. La resolució de TAMP és notòriament difícil perquè requereix una combinació estreta de planificació de tasques i planificació de moviments, considerant restriccions geomètriques i físiques.

En aquesta tesi, el nostre objectiu és millorar el rendiment dels algoritmes TAMP des de tres perspectives complementàries. Primer, investiguem la integració de la planificació de tasques discretes amb l'optimització contínua de trajectòries. La nostra principal contribució és un algorisme que descobreix automàticament per què un pla de tasques podria fallar quan es consideren les restriccions del món físic. Aquesta informació retroalimenta al planificador de tasques, resultant en una interfície bidireccional entre tasques i moviments, capaç de resoldre problemes amb múltiples objectes, robots i restriccions físiques.

Tradicionalment, hi ha hagut dos enfocaments competitius per resoldre problemes TAMP: mètodes basats en mostreig i mètodes basats en optimització. A la segona part, primer il·lustrem que, donada l'àmplia varietat de tasques i entorns dins de TAMP, ni el mostreig ni l'optimització són superiors en tots els escenaris. Per combinar les fortaleses de tots dos enfocaments, hem dissenyat meta-algorismes per a TAMP, algorismes adaptatius que seleccionen automàticament quins algorismes i càlculs utilitzar i com descompondre millor cada problema per trobar una solució més ràpidament.

Una tercera direcció prometedora per millorar els algoritmes TAMP és aprendre de solucions prèvies a problemes similars. A la tercera part, combinem arquitectures d'aprenentatge profund amb raonament basat en models per accelerar els càlculs dins dels nostres algorismes. Específicament, apliquem aquests models en la detecció de quines restriccions físiques són inviables, i en l'optimització no lineal, centrant-nos en la generalització, precisió, temps d'execució i eficiència de dades.

En el nucli de les nostres contribucions hi ha una representació refinada, desglossada i fragmentada dels problemes d'optimització de trajectòries dins de TAMP. Aquesta estructura no només facilita una planificació més eficient, millor anàlisi de les restriccions geomètriques i meta-algorismes, sinó que també proporciona una millor generalització en arquitectures neuronals.

# Chapter 1

# Introduction

**Autonomy of robotic systems** Nowadays, robots are ubiquitous in industrial settings such as factories and warehouses, where they perform repetitive tasks in controlled environments. However, robotic systems still lack the robustness and autonomy needed to become useful companions in our daily lives, helping humans with construction sites, working in hazardous environments, household chores, and elderly care.

Traditionally, robots have operated in very controlled environments, such as factories for car manufacturing. At the most basic level of autonomy, a human worker would manually provide a reference trajectory for the robot, and the robot's task would be to follow this trajectory in a repetitive manner. At a second level of autonomy, the human operator defines a desired goal configuration for the robot, and the robot has to find a collision-free path to reach the goal.

Roughly, these two levels of autonomy correspond, respectively, to two distinct fields in robotics: optimal control [Siciliano et al., 2008] and motion planning [LaValle, 2006]. They both address fundamental problems in robotics, with open and interesting questions from both research and engineering perspectives. However, optimal control and motion planning require precise task specifications and cannot handle complex and long-term tasks independently.

Thus, the degree of autonomy of these systems is insufficient for deploying robots in foreseeable future applications such as construction sites or household environments. As human users, it is not convenient to specify short-term goals or reference trajectories. Instead, we need robots to operate at a higher degree of autonomy, where we provide only a high-level goal such as "clean the table", "stack all the blocks" or "build a chair" and the robot solve the task.

These types of problems, which require long-term planning of a sequence of physical interactions with the environment, considering geometric and physical constraints, can be

formalized as Task and Motion Planning (TAMP) [Garrett et al., 2021]. Together with other fundamental challenges in robotics, such as perception [Thrun et al., 2005] and dexterous manipulation [Billard and Kragic, 2019, Bicchi and Kumar, 2000], TAMP will play a central role in our quest for the autonomy of robotic systems.

Algorithmic improvements, better formulations, and a deeper understanding of Task and Motion Planning are fundamental for future robotics systems. In this thesis, we aim to improve the performance of TAMP algorithms from three complementary perspectives: more efficient solvers that integrate classical planning with trajectory optimization, meta-solvers that automatically select how to solve and decompose the problems, and learning-based methods to accelerate expensive computations and reuse solutions to similar problems.

**Task and Motion Planning** In Task and Motion Planning (TAMP), robots must plan a sequence of feasible motions and interactions with other objects to achieve a desired state of the environment, considering the geometric and physical constraints of the world (e.g., [Toussaint, 2015, Kaelbling and Lozano-Pérez, 2011, Dantam et al., 2018][1]). It combines aspects of both task planning and motion planning and usually assumes a deterministic transition model and perfect state information.

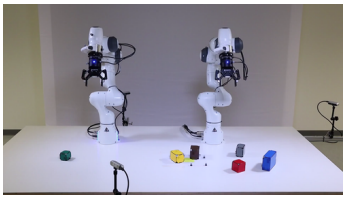
TAMP problems can be solved by planning at two levels of abstraction: a high-level task plan (also known as an action skeleton or sequence of high-level actions) and low-level motion. In the high-level task, the robots decide with which objects to interact, in which order, and what type of interactions, e.g., grasping, pushing, or throwing. Such information can be encoded using discrete variables, resulting in a discrete planning problem [Fikes and Nilsson, 1971, Bonet and Geffner, 2001]. Additionally, the robot must plan low-level motion that fulfills the geometric and physical constraints of the real world, such as collision avoidance, stability, reachability, and friction [Toussaint et al., 2018]. Notably, in real-world applications, there are strong interdependencies between the high-level task and the low-level motion. This prevents a naive decomposition of the problem using directly off-the-shelf task planners and motion planners, making TAMP problems notoriously difficult to solve.

TAMP includes a broad class of problems, including multimodal motion planning, sequential manipulation planning, rearrangement planning, and hybrid task planning. Consider, for example, the problem of building a tower of blocks with two robots, as shown in Fig. 1.1. Fig. 1.2 shows some key configurations of the solution to this TAMP problem. Here, considering reachability and collision avoidance constraints is fundamental because two blocks are obstructing the placement of the tower, and each robot can only reach a subset of the blocks.

Similarly, in the TAMP problem in Fig. 1.3, two robots have to put a ball on top of a tower of blocks. The ball is out of reach, but a stick can be used to push it to the center of the table.

[1] A comprehensive discussion of related work in TAMP is provided later in Chapter 2.

Initial state



Goal

```
Build tower of blocks
blue-gray-red-green
in the center of the table.
```

Figure 1.1.: Task and Motion Planning – Example 1.

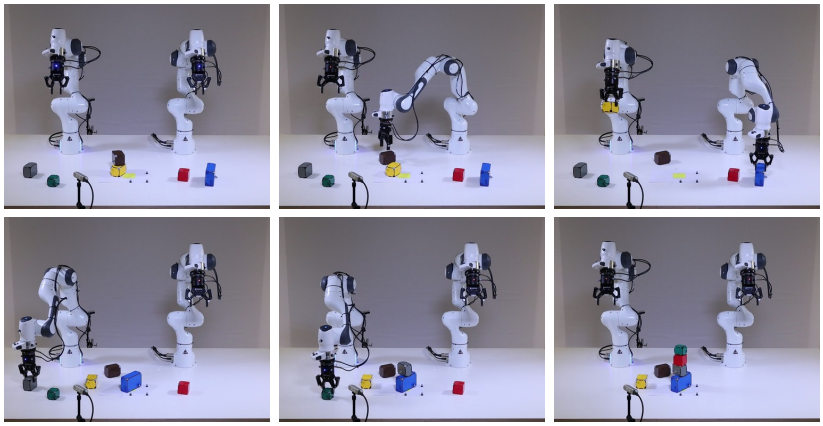


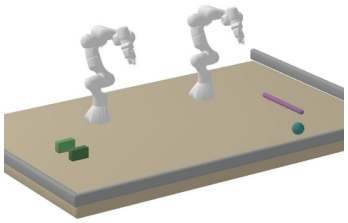
Figure 1.2.: A solution to the Task and Motion Planning problem of Fig. 1.1.

Fig. 1.4 shows some key configurations of the solution to this TAMP problem. Remarkably, in both examples, the human user has only provided the abstract goal, and all the motions and interactions with the environment are computed autonomously by the robots.

*Why are TAMP problems difficult to solve?* First, the complexity of the problem grows exponentially with the number of objects and robots in the environment, often denoted with the term "curse of dimensionality". Second, generating collision-free and feasible robot trajectories is a challenging problem on its own because it requires motion planning and optimization in continuous spaces with geometric and physical nonlinear constraints, such as collision avoidance, grasping, and stability.

Finally, TAMP requires joint reasoning about motion planning and task planning, often with strong dependencies and interactions between the two levels. Geometric constraints and physical constraints have an influence on whether a task plan is valid, and how the robots execute each intermediate step of the plan is relevant for achieving the final high-level goal.

Initial state Goal



```
Build tower of blocks
light green - dark green
with blue ball on top.
```

Figure 1.3.: Task and Motion Planning – Example 2.

Beyond its potential applications, Task and Motion Planning is a very interesting research field because it requires planning at two levels of abstraction while accounting for strong dependencies and without easy decompositions. Such problems are usually easy for humans to solve, as humans excel at reasoning at different levels of abstraction and combining long-term planning with motion planning. However, they are very challenging to formalize mathematically and to solve with computers.

From a robotics perspective, designing TAMP algorithms requires understanding and combining state-of-the-art planning algorithms, such as classical planning, trajectory optimization, and motion planning. Using off-the-shelf solvers is usually not sufficient, as solving TAMP problems requires new interfaces and functionality to achieve a tighter integration.

Throughout this thesis, we assume perfect knowledge of the state of the environment where robots operate, as is typically done in the TAMP literature. Thus, our robots will know the positions and the geometry of the objects they want to manipulate. While this is a general limitation for deploying robots in the real world today–because we cannot assume this perfect information in uncontrolled environments–TAMP with perfect state knowledge is still an open and unsolved research problem.

## 1.1. Sampling and Optimization Methods for Task and Motion Planning

Algorithms for TAMP problems are complex systems that combine and tightly integrate different algorithmic components from task planning and motion planning. Traditionally, there have been two alternative and competing approaches to solving these problems, which

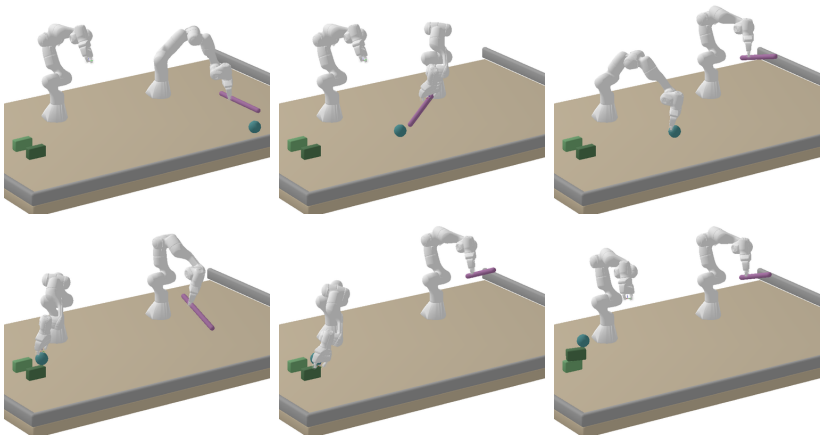


Figure 1.4.: A solution to the Task and Motion Planning problem in Fig. 1.3.

differ in the tools used to compute the motion of the robots and in how the discrete and continuous search are interleaved.

Sample-based approaches to TAMP (e.g., [Garrett et al., 2021, Srivastava et al., 2014]s) incrementally discretize the continuous search space and attempt to compute a solution incrementally, step by step, using constraint sampling methods and sample-based motion planning [Kavraki et al., 1996, LaValle and Kuffner, 2001]. For example, to generate the robot motion that picks up and moves one object, these methods would first generate a valid grasp and a valid placement of the object, then a robot configuration for the pick and for the place using inverse kinematics, and finally a trajectory. Because the motion is computed incrementally in several steps using conditionally constrained sampling, these methods cannot efficiently account for joint constraints in the motion, e.g., when very precise grasps or placements are required to enable subsequent actions.

On the other hand, optimization-based approaches to TAMP first compute candidate task plans [Toussaint, 2015, Toussaint et al., 2018] and then attempt to find a motion plan for the full task plan using trajectory optimization [Bertsekas, 1997, Betts, 1998, Nocedal and Wright, 2006]. This accounts for all the constraints of the motion jointly, instead of sequentially, as in the case of sample-based approaches. There is often a huge number of potential task plans that fail because of the geometric constraints. Thus, a significant challenge in these approaches is to design a good interface between the task and the motion so that candidate plans are informed about geometric infeasibility.

Our work builds on Logic Geometric Programming (LGP) [Toussaint, 2015], a prominent optimization-based formulation of TAMP, which represents the problem as a joint op-

timization over discrete and continuous variables, where different high-level task plans imply different nonlinear constraints for the robot motion.

In the first section of this thesis, we present two new optimization-based solvers for TAMP that combine and refine tools and techniques from trajectory optimization, conflict-based search, and classical planning. Our key contributions are two new bidirectional interfaces between the discrete and continuous levels, which allow informing task planners about geometric infeasibility, resulting in very efficient TAMP solvers.

Our first solver, *Diverse Planning for LGP*, interfaces trajectory optimization with state-of-the-art classical planning by detecting and encoding infeasible prefixes of the task plan (Chapter 4).

However, encoding prefix infeasibility is not enough to solve problems with multiple robots and objects, as the number of candidate high-level plans to evaluate grows exponentially fast.

In our second solver, *Factored-NLP Planner*, we combine classical planning and optimization in a more precise and effective way, leveraging a novel factored representation of LGP problems. Our framework automatically detects which nonlinear constraints fail and encodes this information back into the task planner. This results in a very efficient interface, solving TAMP with several robots, objects, and intricate geometric constraints in just a few seconds (Chapter 5).

**Meta-solvers: An adaptive combination of sampling and optimization** TAMP encompasses a wide range of problem settings, including varying numbers of robots and objects, as well as different goals, geometries, and physical constraints. Given this diversity and complexity, it is unrealistic to expect that a single algorithm can efficiently solve all TAMP problems.

Despite recent advances in both sample-based and optimization-based approaches to TAMP, the performance of each solver is limited by the capabilities of the underlying methods used to generate the motion. When a problem can be easily decomposed into simpler, independent subproblems – where the motion of the robot can be computed independently – sample-based approaches are very efficient. In contrast, when the motion of the robot is highly constrained, and there are long-term dependencies between the actions, optimization-based approaches are more efficient.

Minor variations in the task or the environment can make a TAMP problem more suitable for one type of solver or the other. Therefore, designing a TAMP solver that can efficiently solve all problems while strictly adhering to either the optimization or sample-based paradigm is impossible.

The second part of this thesis investigates how to design meta-solvers for Task and Motion Planning. Intuitively, a meta-solver is a solver that can choose which type of solver to use

based on the problem at hand [Russell and Wefald, 1991]. In our project, the meta-solver will actively reason about how to best decompose the problem and which algorithm and strategy to use to compute the robot motions, two vital questions that are fixed by design in current TAMP solvers.

We first investigate the problem of finding the keyframe configurations for a fixed high-level task plan (Chapter 6). This is a fundamental subproblem of TAMP that already exposes the trade-off between choosing either sampling or joint nonlinear optimization. Our algorithm will learn how to best decompose the problem to maximize the number of solutions found in a fixed computational time. The discovered optimal strategies are adaptive hybrid combinations of optimization and sampling, which outperform user-predefined decompositions as well as full joint optimization.

In a subsequent project, we tackle the comprehensive TAMP problem. To bridge the gap between optimization and sample-based approaches, we first define a novel computational state that extends the traditional notion of discrete-continuous state with free states subject to constraints. Based on this formulation, we introduce a TAMP meta-solver, a hybrid solver that automatically uses a flexible combination of optimization and sampling methods to solve for the high-level task plan and the low-level motion in a TAMP problem (Chapter 7).

## 1.2. Accelerating Model-Based Solvers with Deep Learning

The traditional approach to solving a Task and Motion Planning (TAMP) problem in the real world can be described in three stages: First, we create a model of the world using our knowledge of physics and geometry. Second, we use this model, together with a TAMP solver, to compute the best robot actions. Third, we apply these actions in the real-world system.

Triggered by the success of Deep Learning [Goodfellow et al., 2016] in other fields, such as computer vision [Krizhevsky et al., 2012, Mildenhall et al., 2021] and natural language processing [Hochreiter and Schmidhuber, 1997, Vaswani et al., 2017], there has been an explosion of data-driven approaches that leverage data rather than first-principles physics models to solve robotic problems.

However, in the context of TAMP, where long-term planning is required, abstract and physics models are very valuable as they enable scaling and generalization to different types of problems and scenarios. Instead of replacing world models and planning algorithms, we investigate how to combine data-driven approaches with model-based approaches to improve the efficiency of TAMP solvers. Specifically, we are interested in using a dataset of solutions to similar problems to accelerate planning algorithms, making expensive computations more tractable and enabling real-time solutions to combinatorial and large-scale optimization problems.

The combination of learning and model-based approaches has received a lot of attention in recent years, as it can potentially combine the best of both worlds, but it presents several challenges. A fundamental open research question is how to represent the problem and the solutions to achieve strong generalization and accuracy. Useful neural models should be applicable in a wide range of new, unseen problems, instead of just memorizing the training data as if it were a table of problems and solutions.

In this thesis, we investigate how to use deep learning to accelerate TAMP solvers, focusing on two computationally expensive operations in our TAMP solvers: solving nonlinear optimization programs and determining which constraints are infeasible in an overconstrained optimization problem.

First, we propose deep generative models [Goodfellow et al., 2014, Kingma and Welling, 2013] to provide a good initial warm start for nonlinear optimization in very challenging optimization problems (Chapter 8). In the second project, we train a graph-based classifier [Kipf and Welling, 2016] using the structure of the nonlinear program to directly predict which constraints are infeasible (Chapter 9). In both projects, we present a unique contribution toward understanding how to leverage structural knowledge of the TAMP problem for creating more accurate and generalizable neural models.

## 1.3. The Factored Structure of Task and Motion Planning

A deep study and analysis of the factored structure of Task and Motion Planning problems are at the core of all our contributions.

Using the underlying factored structure of large problems has been key in other robotics fields. For example, in Simultaneous Localization and Mapping (SLAM), the factored structure is used to solve sparse optimization problems faster to estimate the position of a robot while creating a map of the environment [Dellaert et al., 2017]. In classical planning, the factorization of the state space using a set of variables is key to designing domain-independent heuristics that guide heuristic search algorithms [Bonet and Geffner, 2001].

Similarly, a correct and precise formulation of TAMP problems can unveil significant structure that can be exploited to solve the problem more efficiently. First, we have different objects and robots in the scene, which implies a natural factorization of the continuous configuration space and the high-level discrete representation. Second, there is a temporal structure that arises from the temporal dimension of a planning problem. Furthermore, even when the required motions are long and involve multiple robots and objects, each constraint and physical interaction usually involves a few objects at a time.

In this thesis, we use a refined, factored representation of the trajectory optimization problems within TAMP, which exposes the local dependencies between variables and constraints.

For instance, when a robot picks up an object, the new interaction constrains the motion of the object with respect to the gripper but does not create additional constraints between other objects in the environment. Additionally, we show how the nonlinear constraints of the motion are defined by the high-level task plan. For instance, a pushing interaction will imply very different constraints on the robot's and object's motion than a pick, place, or handover action.

A factored representation of TAMP is required to identify which are the building blocks of all the TAMP problems, understand how they are connected, and unveil how these combinations of local structures can compose a very rich set of TAMP problems. For instance, consider again the two TAMP problems of Figs. 1.1 and 1.3, with solutions shown in Figs. 1.2 and 1.4. They both share the same local structure, with multiple objects and robots, pick and place interactions (with additional pushing interactions in the second example). What information can we reuse from one problem to the other? In what sense can we use the solution of one problem to inform the other? How can we decompose the problem? How can we share information about different high-level task plans in the same scenario, or even across problems?

During this thesis, we will incrementally tackle all these questions by using a factored representation of the trajectory optimization problems within TAMP. In our new TAMP solvers, we introduce a refined factored Logic Geometric Program formulation, which, combined with technical algorithmic contributions, can be used to detect why task plans fail and share this information back into the high-level task planner (Chapter 5).

In the second part of the thesis, the factored structure is used to reason about good problem decompositions and sequences of small sampling operations inside a meta-solver (Chapter 6).

Finally, the factored structure is also fundamental in our learning-based approaches (Chapters 8 and 9). Together with suitable neural architectures, the factored structure introduces a strong inductive and relational bias in our models, reducing the sample complexity during training, improving accuracy, and enabling generalization to new and diverse problems and environments.

## 1.4. Reading Guide and Statement of Contributions

Chapter 2 introduces the main concepts and tools used as a foundation throughout the thesis: nonlinear programming, classical planning, and the Logic Geometric Programming formulation of Task and Motion Planning, together with a literature review of solvers and problem formulations for TAMP.

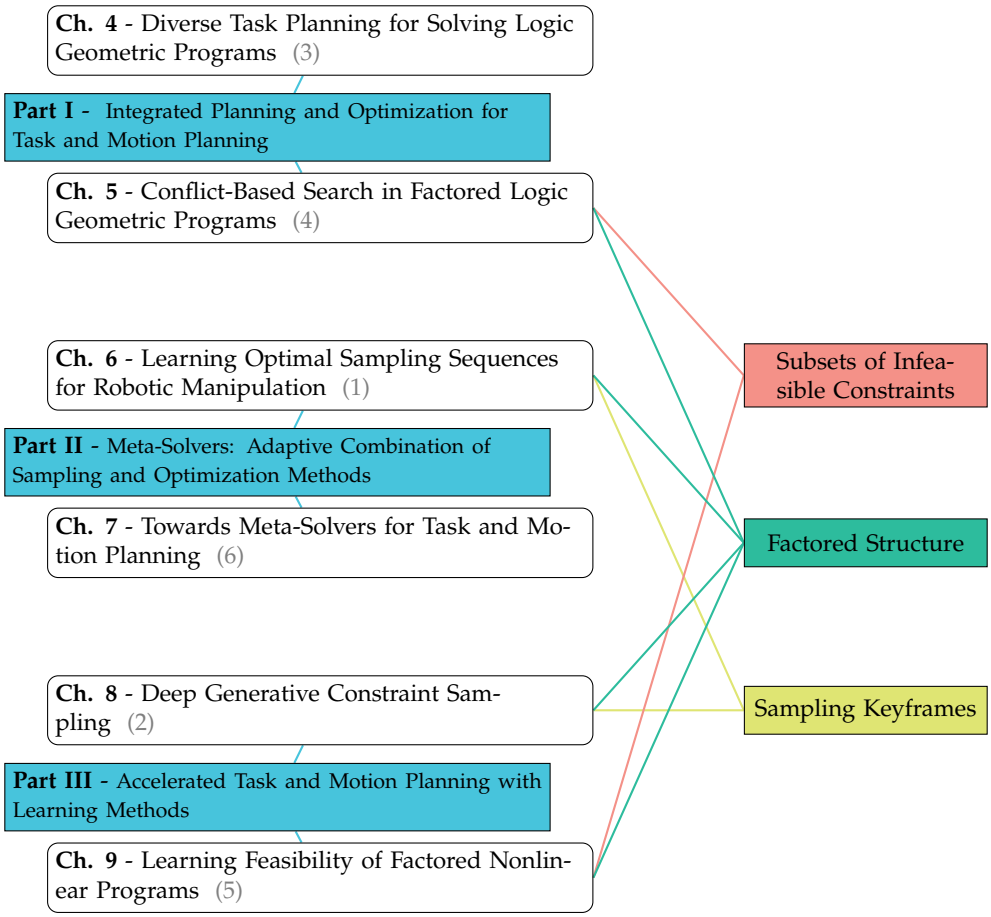



Figure 1.5.: Reading Guide – A graphical overview.

Chapter 3 provides an intuitive explanation of the factored structure inherent in trajectory optimization problems within TAMP, illustrated with multiple examples.

The core of the thesis is divided into six chapters, organized into three parts: 1 - Integrated Planning and Optimization for Task and Motion Planning, 2 - Meta-Solvers: Adaptive Combination of Sampling and Optimization Methods, and 3 - Accelerated Task and Motion Planning with Learning Methods. In the conclusion (Chapter 10), we summarize the contributions and discuss open challenges in the field of TAMP.

Despite this clear organization, several interconnections and synergies emerge between different chapters, offering a comprehensive view of TAMP from multiple perspectives. A graphical overview is provided in Fig. 1.5. Chapters are represented with white rectangles and are connected with other chapters based on the shared contributions, methodologies,

or problem settings they address, depicted with distinct color-coded boxes. For instance, the analysis of the factored structure of the trajectory optimization problems, displayed in red, is a central theme in four out of the six chapters.

Each chapter essentially reflects a different research project. The chronological order of these projects, marked with a gray number, illuminates the progression of my research journey. It aims to show how diverse perspectives and fields can converge on novel ideas and valuable contributions, often requiring one to revisit similar methodologies and problem settings with new tools and insights.

*Part I - Integrated Planning and Optimization for Task and Motion Planning*

- In Chapter 4, *Diverse Task Planning for Solving Logic Geometric Programs*, we present a systematic interface between discrete task planning and trajectory optimization for solving TAMP. Our solver detects geometric conflicts in the form of prefixes of task plans that are infeasible and blocks these prefixes in the task planner. This chapter is based on the publication [Ortiz-Haro et al., 2022c],
  - Ortiz-Haro, J. , Karpas, E., Toussaint, M., and Katz, M. (2022). Conflict-Directed Diverse Planning for Logic-Geometric Programming. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 32, pp. 279-287).
- In Chapter 5, *Conflict-Based Search in Factored Logic Geometric Programs*, we present a new factored formulation of TAMP and a second TAMP solver that uses this factored representation as a bidirectional interface between task and motion. Here, the solver can detect and encode infeasible subsets of nonlinear constraints, resulting in a more efficient interface. This chapter is based on the publication [Ortiz-Haro et al., 2022b],
  - Ortiz-Haro, J. , Karpas, E., Katz, M., and Toussaint, M. (2022). A Conflict-Driven Interface Between Symbolic Planning and Nonlinear Constraint Solving. IEEE Robotics and Automation Letters, 7(4), (pp. 10518-10525).

*Part II - Meta-Solvers: Adaptive Combination of Sampling and Optimization Methods*

- In Chapter 6, *Learning Optimal Sampling Sequences for Robotic Manipulation*, we present a meta-algorithm to solve a key subproblem of TAMP: finding the keyframe configurations for a fixed task plan. The meta-algorithm combines sampling and optimization to minimize the computational time required to generate diverse solutions. This work is based on the publication [Ortiz-Haro et al., 2021],
  - Ortiz-Haro, J. , Hartmann, V. N., Oguz, O. S., and Toussaint, M. (2021). Learning Efficient Constraint Graph Sampling for Robotic Sequential Manipulation. IEEE International Conference on Robotics and Automation (ICRA) (pp. 4606-4612).

– In Chapter 7, *Towards Meta-Solvers for Task and Motion Planning*, we present a meta-solver[2], for the comprehensive TAMP problem. This solver combines search on the task and motion levels and determines the best way to decompose the TAMP problem, deciding automatically whether it is better to use constrained sampling or joint nonlinear optimization.

*Part III - Accelerated Task and Motion Planning with Learning Methods*

– In Chapter 8, *Deep Generative Constraint Sampling*, we present a new method to generate samples on constraint manifolds using a combination of deep generative models and nonlinear optimization. We apply our framework to compute faster keyframe configurations of a fixed task plan. This work is based on the publication [Ortiz-Haro et al., 2022a],

  – Ortiz-Haro, J. , Ha, J. S., Driess, D., and Toussaint, M. (2022). Structured Deep Generative Models for Sampling on Constraint Manifolds in Sequential Manipulation. In Conference on Robot Learning (pp. 213-223). PMLR.

– In Chapter 9, *Learning Feasibility of Factored Nonlinear Programs*, we present a graph-neural model that predicts which constraints of a factored nonlinear program are infeasible. In the context of TAMP, our model can generalize to different scenes, longer manipulation sequences, more robots and objects than the example data seen during training. This chapter is based on the publication [Ortiz-Haro et al., 2023],

  – Ortiz-Haro, J. , Ha, J. S., Driess, D., Karpas, E., and Toussaint, M. (2023). Learning Feasibility of Factored Nonlinear Programs in Robotic Manipulation Planning. IEEE International Conference on Robotics and Automation (ICRA) (pp. 3729-3735).

Several of my publications as a Ph.D. student at TU Berlin have been excluded from this thesis to maintain a focus on the core contributions to Task and Motion Planning. The complete list of publications during my Ph.D. is shown in Appendix A.

[2] We plan to extend and submit the content of this chapter to a robotics or planning conference, for instance, IROS, ICRA, or ICAPS. This research has been conducted in collaboration with Erez Karpas and Marc Toussaint.

# Chapter 2

# Background

This chapter introduces the background material necessary to understand the scientific contributions that will be presented later. Our work builds on Logic Geometric Programming, an optimization-based formulation of Task and Motion Planning (TAMP), which combines nonlinear programming and classical planning.

Thus, we start with a brief presentation of nonlinear programs, including a short discussion on how to solve them and examples in the context of robotics (Section 2.1). Then, we present classical planning, introducing the factored formulation and solvers that we will use later in the thesis. We also present the Blocksworld domain, which provides a high-level abstraction of the TAMP problem, ignoring the geometric constraints (Section 2.2).

In Section 2.3, we present the Logic Geometric Programming formulation together with the state-of-the-art solver for this formulation. We conclude the background chapter with a literature review on Task and Motion Planning (Section 2.4).

## 2.1. Nonlinear Programs in Robotics

**Nonlinear programs** A nonlinear program (NLP) is an optimization problem of the form:

$$\min_{x \in \mathbb{R}^n} \quad f(x)\,, \tag{2.1a}$$

$$\text{s.t.} \quad h(x) = 0\,, \tag{2.1b}$$

$$g(x) \leq 0\,, \tag{2.1c}$$

where $x \in \mathbb{R}^n$ is an $n$-dimensional continuous vector variable, $f : \mathbb{R}^n \to \mathbb{R}$ is the cost function, $h : \mathbb{R}^n \to \mathbb{R}^l$ are the equality constraints, and $g : \mathbb{R}^n \to \mathbb{R}^m$ are the inequality

constraints. All functions $f$, $h$, and $g$ are smooth and differentiable. The abbreviation s.t. stands for "subject to".

The optimal solution minimizes the objective function while fulfilling the equality and inequality constraints. It must fulfill the first-order necessary conditions of optimality, known as the Karush–Kuhn–Tucker (KKT) conditions. KKT conditions state that if a point $x^* \in \mathbb{R}^n$ is a local minimum, then there exist vectors $\lambda^* \in \mathbb{R}^l$ and $\mu^* \in \mathbb{R}^m$, called Lagrange multipliers or dual variables, such that the following conditions hold:

$$\nabla f(x^*) + Dh(x^*)^T \lambda^* + Dg(x^*)^T \mu^* = 0\,, \tag{2.2a}$$
$$h(x^*) = 0\,, \tag{2.2b}$$
$$g(x^*) \leq 0\,, \tag{2.2c}$$
$$\mu^* \geq 0\,, \tag{2.2d}$$
$$g(x^*)^T \mu = 0\,, \tag{2.2e}$$

where $\nabla f(x^*)$ is the gradient of $f$, and $Dh(x^*)$ and $Dg(x^*)$ are the Jacobians of $h$ and $g$ evaluated at $x^*$. The KKT conditions are sufficient conditions for optimality if the NLP is a convex optimization problem, with $f$ convex, $h$ being linear, and $g$ being convex (under some regularity conditions; we refer to [Nocedal and Wright, 2006] for more technical and precise definitions and proofs).

If a vector $x_{\text{feas}} \in \mathbb{R}^n$ fulfills the constraints, i.e., $h(x_{\text{feas}}) = 0$, $g(x_{\text{feas}}) \leq 0$, it is called a feasible solution. The set of feasible solutions of (2.1) defines a nonlinear manifold

$$\mathcal{M} = \{x \in \mathbb{R}^n \mid h(x) = 0,\, g(x) \leq 0\}\,. \tag{2.3}$$

An NLP is infeasible if there are no feasible solutions. Throughout this thesis, we often omit the cost term and focus on the feasibility problem,

$$\text{find } x \in \mathbb{R}^n, \tag{2.4a}$$
$$\text{s.t. } h(x) = 0\,, \tag{2.4b}$$
$$g(x) \leq 0\,. \tag{2.4c}$$

For simplicity, we will refer to both (2.1) and (2.4) as NLPs. In fact, a practical and robust way to generate one feasible solution in (2.4) is to choose a reference value $x_{\text{ref}} \in \mathbb{R}^n$ and solve the NLP:

$$\min_{x \in \mathbb{R}^n} ||x - x_{\text{ref}}||^2 \tag{2.5a}$$
$$\text{s.t. } h(x) = 0\,, \tag{2.5b}$$
$$g(x) \leq 0\,. \tag{2.5c}$$

**Methods for solving a nonlinear program** Nonlinear programs do not have a closed-form solution, and they are usually solved with iterative methods. An extensive review is available in [Nocedal and Wright, 2006].

*Unconstrained Optimization:* First, let's consider the unconstrained optimization problem,

$$\min_{x \in \mathbb{R}^n} \quad f(x)\,. \tag{2.6}$$

We can optimize this function with a broad family of local iterative algorithms that generate a sequence $x_{k+1} = x_k + \alpha_k d_k$, starting from an initial value $x_0$. The step direction $d_k$ can be computed using the gradient $\nabla f(x_k)$ and the Hessian $\nabla^2 f(x_k)$ (second-order derivatives of the function $f$ at the current point $x_k$).

For instance, in gradient descent, $d_k$ is given by $-\nabla f(x_k)$. Using second-order information, $d_k$ is given by $-(\nabla^2 f(x_k))^{-1}\nabla f(x_k)$ in the Newton method, or $-(\nabla^2 f(x_k) + \beta_k I_d)^{-1}\nabla f(x_k)$, with $\beta_k > 0$, for a regularized Newton method. Quasi-Newton methods use $-B^{-1}\nabla f(x_k)$, where $B$ approximates the Hessian.

The step size $\alpha_k$ is either fixed or chosen adaptively to ensure that the function decreases enough at each iteration, using a line search algorithm, e.g., a backtracking line search that finds the step size that fulfills the Armijo or Wolfe Conditions [Nocedal and Wright, 2006].

In general, the sequence $x_0, x_1, \ldots, x_k$ converges to a stationary point, i.e., a point where the gradient is zero. Note that some precautions must be taken when computing the step direction and step size, see [Nocedal and Wright, 2006].

However, the point of convergence can be a local minimum, instead of a global minimum $x^* = \min_x f(x)$. When the function $f$ is convex, a local minimum is guaranteed to be the global minimum, but otherwise, this does not hold in the general non-convex case. Thus, the choice of the initial guess $x_0$ is very important, as it can lead to different local minima.

The convergence rate is linear for gradient descent and quadratic for the Newton method. The computational complexity of each iteration, with respect to the size of the vector variable problem (i.e., $n$), is linear in gradient descent and cubic in the Newton method, which requires solving a linear system of dimension $n$ in each iteration.

In robotics problems, Newton and Quasi-Newton methods converge much faster than gradient descent methods and are often preferred. Gradient descent performs poorly when the cost function has different curvatures in different directions (a very large disparity between the largest and smallest eigenvalues of the Hessian matrix).

*Constrained Optimization:* The most popular algorithms to solve (2.1) are the Augmented Lagrangian algorithm, sequential quadratic programming, interior point methods, and penalty methods. Similar to the unconstrained case, they are iterative methods that generate a sequence of points $x_k$ that converges to a stationary point.

All these methods try to find a point that fulfills the first-order necessary conditions for optimality (2.2). Similar to the unconstrained case, these methods perform only local optimization. In general, convergence to the optimum is guaranteed only if the NLP is a convex optimization problem with both a convex objective function and a convex feasible set.

In practice, in the nonlinear case, these methods might converge to a feasible local optimal point or to an infeasible point that does not fulfill the constraints, depending on the initial guess and the nonlinearity of the constraints.

Throughout this thesis, we solve NLPs using the Augmented Lagrangian algorithm. The Augmented Lagrangian algorithm solves (2.1) by solving a sequence of unconstrained optimizations. Starting from an initial guess $(x_0, \lambda_0, \mu_0)$, we update the primal and the dual variables in an iterative two-step process. At iteration $k$, primal variables are updated by solving the unconstrained optimization problem (starting from the initial guess $x_{k-1}$)

$$\min_x \mathcal{L}_A(x, \lambda_{k-1}, \mu_{k-1}, \rho), \tag{2.7}$$

where $\mathcal{L}_A$ is the Augmented Lagrangian, defined as:

$$\begin{aligned}\mathcal{L}_A(x, \lambda, \mu, \rho) = f(x) &+ \lambda^T h(x) + \mu^T g(x) + \\ &\frac{\rho}{2}\sum_{j=1}^{l} h_j(x)^2 + \frac{\rho}{2}\sum_{j=1}^{m}[g_j(x) \geq 0 \vee \mu_j > 0] g_j(x)^2\,,\end{aligned} \tag{2.8}$$

with the penalty parameter $\rho > 0$ and where the subscript $j$, e.g., $h_j(x)$, indicates the $j$-th component of a vector. Using this formulation (slight variations of the term for the inequalities are also possible), an inequality constraint acts as an equality constraint if it is not fulfilled or its dual variable is strictly positive.

After the primal variables $x_k$ have been updated, the dual variables are updated with:

$$\lambda_k \leftarrow \lambda_{k-1} + \rho h(x_k)\,, \quad \mu_k \leftarrow \max(0, \mu_{k-1} + \rho g(x_k)). \tag{2.9}$$

Additionally, it is often convenient to increase the penalty parameter $\rho$ if the constraints are not fulfilled to a desired amount. Detailed analyses of convergence and practical implementations are provided, e.g., in [Nocedal and Wright, 2006, Andreani et al., 2008, Conn et al., 2013].

**Examples of nonlinear programs in robotics** Nonlinear programs are used in robotics for a wide range of applications, including motion planning, optimal control, and inverse kinematics.

The variables in an NLP represent the configuration of the robot. For instance, the joint angles of a 7-DOF manipulator can be represented with a vector variable $x \in \mathbb{R}^7$. A finite-dimensional vector can also represent a trajectory. For example, the trajectory of a manipulator $q : [0, 1] \to \mathbb{R}^7$ can be represented by a cubic polynomial $q(t) = a + bt + ct^2 + dt^3$, resulting in the variable in the NLP $x = [a, b, c, d] \in \mathbb{R}^{4\cdot 7}$.

The nonlinear constraints in the NLP can model various constraints, including collision avoidance, kinematic, grasping, and contact constraints. The cost function can represent objectives such as energy, time, and control effort. We now present two examples of NLPs in robotics relevant to this thesis.

*Single Keyframe Optimization:* Suppose we have a 7-DOF manipulator, and our objective is to generate a configuration that picks up a box using a top grasp along the x-direction (assuming, for simplicity, that the box is aligned with the world axis). This problem can be formulated as the following NLP, where the variable $x \in \mathbb{R}^7$ represents the joint angles of the robot.

$$\begin{aligned}
\min_{x \in \mathbb{R}^7} \quad & ||x - x_{\text{ref}}||^2 , && && \text{(2.10a)}\\
\text{s.t.} \quad & p_z(x) - b_z = b/2 , && && \text{(2.10b)}\\
& a/2 \le p_x(x) - b_x \le a/2 , && && \text{(2.10c)}\\
& R(x) = I_d , && && \text{(2.10d)}\\
& p_y(x) - b_y = 0 , && && \text{(2.10e)}\\
& q_{\text{lb}} \le x \le q_{\text{ub}} , && && \text{(2.10f)}\\
& \text{sdf}(P_j(x), \text{env}) \ge 0 && j = 1, \dots, J && \text{(2.10g)}\\
& \text{sdf}(P_j(x), \text{block}) \ge 0 && j = 1, \dots, J && \text{(2.10h)}
\end{aligned}$$

where $p(x) : \mathbb{R}^7 \to \mathbb{R}^3$ is the position of the end-effector as a function of the joint values, and $R(x)$ is the rotation matrix representing the orientation of the end-effector. Here, we assume that a successful grasp requires the end-effector to have the same orientation as the box, thus $R(x) = I_d$, but more complex grasp constraints are used later throughout the thesis. Vectors $q_{\text{lb}}$ and $q_{\text{ub}}$ are the lower and upper bounds of the joint angles. The position of the block is given by $(b_x, b_y, b_z)$, with $a$ and $b$ denoting its length and height. $x_{\text{ref}}$ is a reference configuration for the robot.

$P_j(x)$ represents the collision shape of part $j$ of the robot, the position and orientation of which depend on the joint values $x$. The robot has $J$ collision parts (often, one per link). The signed-distance function, denoted with sdf, is a function that returns the minimum

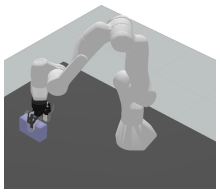


Figure 2.1.: Solution to the NLP in Eq. (2.10).

distance between the collision shape and either the environment (env) or the block (block). If the objects are in collision, sdf returns a negative value corresponding to the penetration distance. A solution to this NLP is depicted in Fig. 2.1.

*Trajectory Optimization:* In the second example, our goal is to generate a trajectory for grasping the object starting from the configuration $q_0$. We choose to parameterize the trajectory using $N$ waypoints. Our variable is now $x = [q_1, \dots, q_N] \in \mathbb{R}^{7 \cdot N}$, where $q_i \in \mathbb{R}^7$ is the joint values at waypoint $i$. The optimization problem is:

$$\min_{[q_1,\dots,q_N]\in\mathbb{R}^{7\cdot N}} \quad ||q_1 - q_0||^2 + \sum_{i=2}^{N} ||q_i - 2q_{i-1} + q_{i-2}||^2 \,, \tag{2.11a}$$

$$\text{s.t.} \quad p_z(q_N) - b_z = b/2 \,, \tag{2.11b}$$

$$a/2 \le p_x(q_N) - b_x \le a/2 \,, \tag{2.11c}$$

$$R(q_N) = Id \,, \tag{2.11d}$$

$$p_y(q_N) - b_y = 0 \,, \tag{2.11e}$$

$$q_{\text{lb}} \le q_i \le q_{\text{ub}} \,, \qquad i = 1, \dots, N \tag{2.11f}$$

$$\text{sdf}(P_j(q_i), \text{env}) \ge 0 \,, \qquad i = 1, \dots, N \quad j = 1, \dots, J \tag{2.11g}$$

$$\text{sdf}(P_j(q_i), \text{block}) \ge 0 \,. \qquad i = 1, \dots, N \quad j = 1, \dots, J \tag{2.11h}$$

The chosen cost function minimizes the sum of squared accelerations, computed using second-order backward finite differences. The constraints on the last configuration $q_N$ are the same as in the previous NLP (2.10). Joint limits and collisions are assessed at each waypoint. In this example, we only evaluate collisions at each waypoint, but collisions could also be assessed at intermediate trajectory points, which can be determined through linear interpolation. The solution is shown in Fig. 2.2.

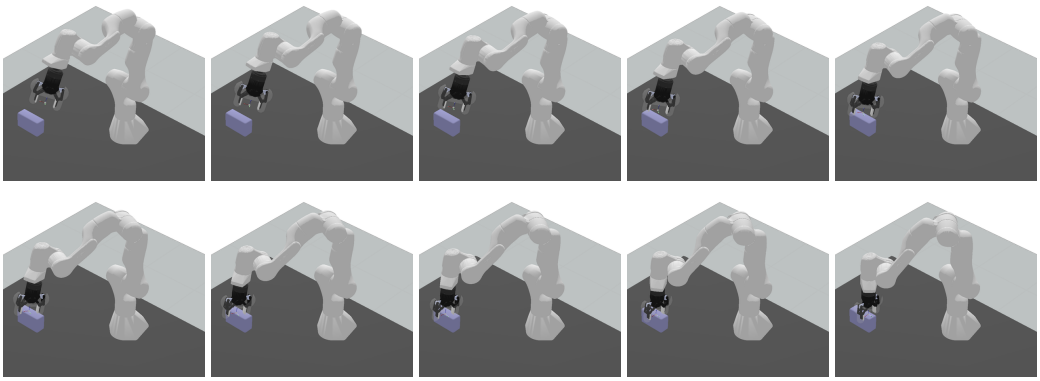


Figure 2.2.: Solution to the NLP in Eq. (2.11) using 10 waypoints ($N = 10$). Each image shows a waypoint of the trajectory (from top left to bottom right).

## 2.2. Classical Planning

Classical planning involves finding a sequence of actions to achieve a goal from an initial state. It assumes that states and actions are discrete and finite, the state is fully observable, and actions have known deterministic effects.

A classical planning model $\Pi = \langle \mathcal{S}, s_0, \mathcal{S}_G, \mathcal{A} \rangle$ comprises:

- A finite and discrete set of states $\mathcal{S}$, representing the state space.
- An initial state $s_0 \in \mathcal{S}$.
- A set of goal states $\mathcal{S}_G \subseteq \mathcal{S}$.
- A set of actions $\mathcal{A}$. The subset of actions applicable in state $s$ is given by $\mathcal{A}(s) \subseteq \mathcal{A}$, and executing action $a$ in state $s$ results in the successor state $s' = \text{succ}(s, a)$.

A feasible solution to a planning problem is a sequence $s_0, a_1, s_1, \ldots, a_K, s_K$ that transforms the initial state $s_0$ into a goal state $s_K \in \mathcal{S}_G$, where $a_k \in \mathcal{A}(s_{k-1})$ and $s_k = \text{succ}(s_{k-1}, a_k)$. The optimal solution minimizes the number of actions (assuming each action has a uniform cost).

In large problems, explicitly enumerating the state space is not feasible. In such cases, we use factored representations, in which states are factored into variables. A state is now represented as a complete value assignment to a set of variables with finite and discrete domains. The set of applicable actions $\mathcal{A}(s)$ and the successor function $\text{succ}(s, a)$ are now defined in terms of conditions and effects on these variables.

In this thesis, we use the SAS+ encoding of the classical planning problem, as referenced by [Bäckström and Nebel, 1995]. This provides a more compact and intuitive representation for

TAMP problems than the original STRIPS formulation [Fikes and Nilsson, 1971]. A notable difference is that in SAS+, variables have a discrete, finite domain, whereas in STRIPS, they are boolean.

A factored classical planning task is a tuple $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ where:

- $\mathcal{V}$ is a set of state variables. Each state variable $v \in \mathcal{V}$ has a finite domain $\mathcal{D}(v)$. A fact is a pair $\langle v, \vartheta \rangle$ of a variable $v \in \mathcal{V}$ and its value $\vartheta \in \mathcal{D}(v)$.
- An assignment to all the variables in $\mathcal{V}$ is called a state $s$. The set of all such states is denoted as $\mathcal{S}$. A partial state $p$ is a value assignment to only a subset of the variables in $\mathcal{V}$. We view a partial state $p$ as a set of facts (i.e., a set of variable-value pairs) and use $p[v]$ to denote the value of variable $v$ in $p$ (i.e., $p[v] = \vartheta$ if and only if $\langle v, \vartheta \rangle \in p$). For any partial state $p$, $\mathcal{V}(p) \subseteq \mathcal{V}$ indicates the state variables instantiated by $p$. A partial state $p$ is consistent with a state $s$ if $p \subseteq s$.
- $\mathcal{A}$ is a set of actions. Each action $a$ is a pair of partial states called preconditions $pre(a)$ and effects $eff(a)$. An action $a$ is valid in a state $s$ if $pre(a) \subseteq s$. The set of all applicable actions in $s$ is given by $\mathcal{A}(s)$. Applying $a$ on state $s$ results in the next state $s' = \text{succ}(s, a)$, where the value of variables $v \in \mathcal{V}(eff(a))$ has changed to $eff(a)[v]$.
- An initial state $s_0$.
- A partial state $g$ that defines the goal.

Similarly to the unstructured case, an action sequence $\pi = \langle a_1, \ldots, a_K \rangle$ is a valid plan if each action is applicable in the previous state ($s_k = \text{succ}(s_{k-1}, a_k)$, starting from $s_0$), and the final state satisfies the goal, that is $g \subseteq s_K$.

The Planning Domain Definition Language (PDDL) [McDermott et al., 1998] has become a standard for defining planning problems, as it is the language for the International Planning Competitions (IPC) (e.g., [Long and Fox, 2003], [Coles et al., 2012a]). In PDDL, a planning problem is defined as a pair comprising a planning domain and a problem instance. The domain defines a class of problems by specifying the set of valid action schemas and predicates in this domain. The problem instance defines the objects, on which predicates are evaluated, the initial state, and the goal state. For solving a PDDL problem, classical planners first transform the action schemas and objects into a propositional representation like STRIPS or SAS+.

The classical planning community has developed domain-independent planners that leverage the factored representation of the planning problem. A prominent approach is heuristic search, where the distance to the goal is estimated by solving a relaxed (simplified) version of the original problem. The relaxed problems, often based on the so-called delete relaxation, can be solved efficiently in polynomial time on the number of variables (instead of exponential) [Bonet and Geffner, 2001, Helmert, 2006, Hoffmann and Nebel, 2001].

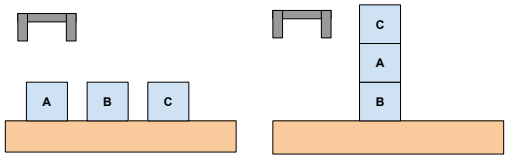



Figure 2.3.: Start and goal states in the Blocksworld problem. PDDL files are shown in Fig. 2.5.

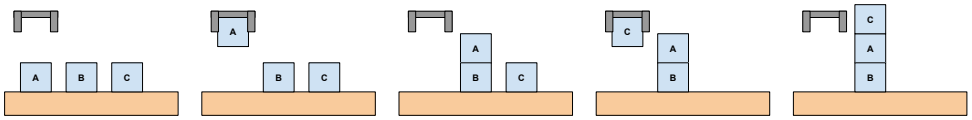



Figure 2.4.: Solution to the Blocksworld problem in Fig. 2.3.

Alternative approaches for solving classical planning problems include reductions to other formalisms such as Boolean Satisfiability (SAT) [Biere et al., 2009] and Constraint Satisfaction Problems (CSP) [Rossi et al., 2006].

**Blocksworld** Blocksworld is a classical planning problem that involves a set of blocks on a table, where the goal is to build a particular stack of blocks.

The Blocksworld PDDL domain file is shown in Fig. 2.5a. An example problem instance is shown in Fig. 2.5b (file) and Fig. 2.3 (graphical visualization). The solution to this problem is shown in Fig. 2.4.

Blocksworld is a high-level planning problem that only deals with logical relationships between blocks. On the other hand, Task and Motion Planning (TAMP) in robotics goes beyond abstract reasoning to include physical constraints. TAMP considers both the high-level discrete actions (like picking up an object) and the low-level continuous motions required to perform those tasks (such as path planning, reachability, and collision avoidance). This makes TAMP much more complex and connected to the real physical world, whereas Blocksworld remains a more abstract and simplified problem.

The success of classical planning in solving large-scale problems stems from the analysis of the factored structure inherent in most planning problems. This factorization, exemplified by the Blocksworld domain, is also readily available in Task and Motion Planning (TAMP) problems and will be exploited throughout this thesis (see Chapter 3 for a first introduction).

```
(define (domain blocksworld)
  (:predicates (on ?x ?y) (ontable ?x) (clear ?x)
               (handempty) (holding ?x))
  (:action pick-up
   :parameters (?x)
   :precondition (and (clear ?x) (ontable ?x) (handempty))
   :effect (and (not (ontable ?x)) (not (clear ?x))
                (not (handempty))  (holding ?x)))
  (:action put-down
   :parameters (?x)
   :precondition (holding ?x)
   :effect (and (not (holding ?x)) (clear ?x)
                (handempty) (ontable ?x)))
  (:action stack
   :parameters (?x ?y)
   :precondition (and (holding ?x) (clear ?y) (not (= ?x ?y)))
   :effect (and (not (holding ?x)) (not (clear ?y)) (clear ?x)
                (handempty) (on ?x ?y)))
  (:action unstack
   :parameters (?x ?y)
   :precondition (and (on ?x ?y) (clear ?x) (handempty) (not (= ?x ?y)))
   :effect (and (holding ?x) (clear ?y) (not (clear ?x))
                (not (handempty)) (not (on ?x ?y))))
)
```

(a) Domain.

```
(define (problem blocksworld-problem)
  (:domain blocksworld)
  (:objects a b c )
  (:init (handempty) (ontable a) (ontable b) (ontable c)
         (clear a) (clear b) (clear c))
  (:goal (and (clear c) (ontable b) (on c a) (on a b)))
)
```

(b) Example of a problem instance.

Figure 2.5.: Blocksworld in PDDL.

## 2.3. Logic Geometric Programming

*Can we formulate TAMP as a continuous optimization problem?* Logic Geometric Programming (LGP) is an optimization-based formulation of Task and Motion Planning (TAMP). To motivate the LGP formulation, we first show that Task and Motion Planning can be written as a single continuous-time optimization program, which can later be discretized into a nonlinear program. However, proceeding without introducing a discrete abstraction renders the problem unsolvable.

Let $\mathcal{X} = \mathbb{R}^n \times SE(3)^m$ be the configuration space of an $n$-dimensional robot and $m$ rigid objects, initially at pose $x_0 \in \mathcal{X}$. The trajectory of the robot and the movable objects can be represented with a continuous function $x(t) : [0, T] \to \mathcal{X}$, with $T$ being the terminal time. A Task and Motion Planning problem is then formulated as a continuous trajectory optimization with,

$$\min_{x(t),T} \int_0^T f_{\text{path}}\left(\bar{x}(t)\right) dt\,, \tag{2.12a}$$

$$\text{s.t. } x(0) = x_0, \tag{2.12b}$$

$$h_{\text{goal}}\left(x(T)\right) = 0,\ g_{\text{goal}}\left(x(T)\right) \leq 0, \tag{2.12c}$$

$$h_{\text{path}}\left(\bar{x}(t)\right) = 0,\ g_{\text{path}}\left(\bar{x}(t)\right) \leq 0, \quad \forall t \in [0, T] \tag{2.12d}$$

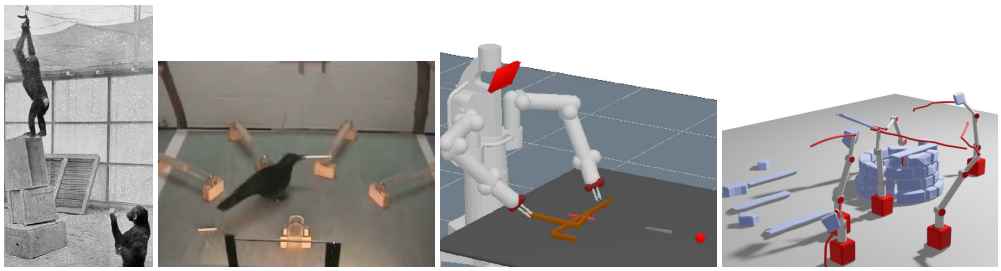
where $\bar{x}(t) = (x(t), \dot{x}(t), \ddot{x}(t))$ includes the position, velocity, and acceleration of the trajectory. Path constraints $h_{\text{path}}$ and $g_{\text{path}}$ represent the physical constraints of the physical world. For instance, objects can only move when grasped or pushed by the robot, and objects and robots should not collide with the environment. Goal constraints $\{h, g\}_{\text{goal}}$ impose constraints on the last state and are used to represent the desired final state, such as stacking objects in a tower. The cost function $f_{\text{path}}$ typically represents control effort, smoothness, or energy.

Converting the continuous-time optimization to an NLP requires a finite-dimensional representation of the trajectory. For instance, this could be achieved using a finite sequence of waypoints, and checking the path constraints on a finite set of points (e.g., the waypoints themselves), similarly to the example in Eq. (2.11).

Unfortunately, this NLP formulation is unsolvable, even for short-term horizons and few objects in the scene. The two main challenges are as follows:

1. The non-convexity of the constraints used to model all possible physics interactions (e.g., using complementarity constraints [Posa et al., 2014]) leads to disconnected and non-convex feasible sets.

2. The lack of meaningful gradients, which also stems from the generic nature of the constraints. If the initial guess for the trajectory does not interact with movable objects,



(a) (b) (c) (d)

Figure 2.6.: Logic Geometric Programming. *(a,b)* Apes and crows solving physical puzzles that require precise manipulation and long-term planning of physical interactions. *(c)* In robotics, LGP can be used to model tool-use, and diverse physical interactions, such as grasping, pushing, and throwing. *(d)* The LGP framework and the original Multi-Bound Tree Search algorithm have been extended to solve problems in the construction domain [Hartmann et al., 2022]. Images are reproduced from [Toussaint et al., 2018, Hartmann et al., 2022].

the derivatives of the path and goal constraints either lack relevant information for long-term planning, or only greedily move the robot towards every movable object.

Therefore, a local optimization method will get trapped in an infeasible local optimum and fail to find a feasible solution to (2.12). In fact, we observe that general complex behavior that requires reasoning about interacting with multiple objects cannot be computed using only local optimization methods.

*High-level and low-level abstractions in TAMP*: Introducing a high-level abstraction that defines the sequence of interactions between the robot and the objects is essential to solving the TAMP problem.

In fact, the problem can be formulated with two levels of abstraction: low-level motion planning and high-level task planning. The high-level task planning can be formulated as a classical planning task, introducing a set of discrete states and actions. But now, in addition to planning in a discrete domain, choices on the discrete level impose nonlinear constraints on the continuous trajectory, with a different sequence of high-level actions implying a different set of nonlinear constraints on the trajectory.

Therefore, instead of a single universal physics constraint, we now operate with a set of more approachable nonlinear constraints based on each discrete state and action. Though these constraints remain nonlinear, they tend to be smooth and informative, and typically guide the optimizer towards feasible solutions, as demonstrated by the diverse set of TAMP problems solved using the LGP formulation [Toussaint and Lopes, 2017, Toussaint et al., 2018] (Fig. 2.6).

**Logic Geometric Program** Formally, consider a continuous configuration space $\mathcal{X}$ (e.g., $\mathbb{R}^n \times SE(3)^m$ for an $n$-dimensional robot and $m$ rigid objects), a finite set of discrete states $\mathcal{S}$, and a finite set of discrete actions $\mathcal{A}$. Let $x_0 \in \mathcal{X}$ be the initial configuration, $s_0 \in \mathcal{S}$ be the initial discrete state, and $\mathcal{S}_g \subseteq \mathcal{S}$ be the set of discrete goal states. A Logic Geometric Program is a combined optimization problem over the continuous trajectory $x(t) : [0, KT] \to \mathcal{X}$ and the sequence of discrete states and actions $s_0, a_1, s_1, \ldots, a_K, s_K$,

$$\min_{x(t), a_{1:K}, s_{1:K}} \sum_k \int_{t_k}^{t_{k+1}} f_{\text{path}}\left(\bar{x}(t), s_k\right) dt \,, \tag{2.13a}$$

$$\text{s.t.} \quad x(0) = x_0 \,, \tag{2.13b}$$

$$h_{\text{path}}\left(\bar{x}(t), s_k\right) = 0 \,, \quad t \in [t_k, t_{k+1}], \qquad k = 0, \ldots, K-1 \tag{2.13c}$$

$$g_{\text{path}}\left(\bar{x}(t), s_k\right) \leq 0 \,, \quad t \in [t_k, t_{k+1}], \qquad k = 0, \ldots, K-1 \tag{2.13d}$$

$$h_{\text{switch}}\left(\bar{x}\left(t_k\right), s_{k-1}, s_k\right) = 0 \,, \qquad k = 1, \ldots, K \tag{2.13e}$$

$$g_{\text{switch}}\left(\bar{x}\left(t_k\right), s_{k-1}, s_k\right) \leq 0 \,, \qquad k = 1, \ldots, K \tag{2.13f}$$

$$s_k = \text{succ}\left(s_{k-1}, a_k\right) \,, \qquad k = 1, \ldots, K \tag{2.13g}$$

$$s_k \in \mathcal{S}, \quad a_k \in \mathcal{A}, \qquad k = 1, \ldots, K \tag{2.13h}$$

$$s_K \in \mathcal{S}_g \,. \tag{2.13i}$$

Here, $\bar{x}(t) = (x(t), \dot{x}(t), \ddot{x}(t))$, $t_k = Tk$ is the start time of step $k$, and $s_{1:K}, a_{1:K}$ are short notations for $\langle s_1, \ldots, s_K \rangle$ and $\langle a_1, \ldots, a_K \rangle$. The functions $(h, g)_{\text{path}}$, $(h, g)_{\text{switch}}$, and $f_{\text{path}}$ are smooth and differentiable in the continuous configuration when conditioned on the discrete states. The discrete function $\text{succ}(s, a)$ indicates the successor discrete state after applying action $a$ to state $s$. The length of the sequences $K \in \mathbb{N}$ is also subject to optimization. We remark that the discrete component of an LGP corresponds to a classical planning problem Section 2.2.

The sequence $a_{1:K}$ is called a task plan, action-skeleton, or sequence of high-level actions in the TAMP literature. Throughout this thesis, we use the term *task plan*. The continuous motion $x(t)$ is now divided into $K$ phases of duration $T \in \mathbb{R}$ (for simplicity, we assume $T$ is fixed, but it could also be optimized), with different nonlinear constraints on the configuration for each phase, based on $s_k$ (or the pair $s_{k-1}, s_k$ for switch constraints).

The key difference from the unstructured problem (2.12) is that the nonlinear functions $(f, g, h)$ are now conditioned on the discrete state $s_k$ and provide informative gradients for local optimization, as opposed to using a unique universal constraint based on complementarity or differentiable contact models. This means that the constraints $h_{\text{switch}}$ and $g_{\text{switch}}$ will be different functions of $\bar{x}$ depending on which object is grasped, pushed, or placed, as indicated by the discrete states.

**Continuous space and discrete space in LGP** Before delving into solving the Logic Geometric Program (2.13), we will clarify the meanings of the continuous space, discrete space, and the nonlinear constraints in the context of TAMP.

*Discrete Level*: The discrete states $s \in \mathcal{S}$ in LGP are used to encode the structure of the kinematic tree, which models which objects are in contact with each other. For instance, an object can be attached either to the gripper or to the table. This information is discrete because it only contains a parent's name or identifier, but it does not consider the continuous relative pose between them. Apart from special discrete symbols to represent the initial position of the objects, LGP does not introduce additional discrete symbols to represent the continuous state of the world, such as concrete values of the positions of the objects, grasps, or robot configurations, as done in other sample-based TAMP formulations (e.g., [Garrett et al., 2020]). Equivalently, we can think of the discrete states as modeling the discrete contact states of the world, specifying which objects are in contact with others. From the perspective of multimodal motion planning, discrete states in an LGP represent the different motion modes.

Discrete actions $a \in \mathcal{A}$ are the high-level actions of the task plan, such as pick, place, push, which change the kinematic structure or contact status. Discrete actions do not include any continuous parametrization. For instance, a discrete action *pick object A with Q from table* does not model the continuous grasp, e.g., the relative transformation between the gripper and the object.

*Continuous Level*: The continuous configuration $\mathcal{X} = \mathbb{R}^n \times SE(3)^m$ represents the configuration of the robot (joint values) and the objects (position and orientation). Collision avoidance, reachability, grasping, pushing, and placement constraints are modeled using the nonlinear constraints inside $(h, g)_{(\text{path,switch})}$, which vary in each motion step depending on the discrete decisions.

For instance, concerning grasping constraints, we use geometric constraints, such as aligning the end-effector with a particular axis of a box or ensuring that a point near the end-effector's palm touches the object's surface while maintaining the correct orientation. In practice, for boxes, balls, and sticks, this typically implies the existence of a stable grasp, which is then represented as a constant relative transformation until placement. For pushing constraints, we introduce additional decision variables for the forces and point-of-attack between the two interacting objects. Motion and forces are then constrained by physics equations as outlined in [Toussaint et al., 2020].

**How to solve a Logic Geometric Program?** If we fix the task plan $\langle a_1, \ldots, a_K \rangle$ in a LGP, the resulting subproblem, denoted as Trajectory-NLP($\langle a_1, \ldots, a_K \rangle$), is a continuous optimization problem:

$$\min_{x(t)} \sum_k \int_{t_k}^{t_{k+1}} f_{\text{path}}(\bar{x}(t), s_k) dt \,, \tag{2.14a}$$

s.t.

$$x(0) = x_0, \tag{2.14b}$$

$$h_{\text{path}}(\bar{x}(t), s_k) = 0, \quad t \in [t_k, t_{k+1}], \quad k = 0, \ldots, K-1, \tag{2.14c}$$

$$g_{\text{path}}(\bar{x}(t), s_k) \leq 0, \quad t \in [t_k, t_{k+1}], \quad k = 0, \ldots, K-1, \tag{2.14d}$$

$$h_{\text{switch}}(\bar{x}(t_k), s_{k-1}, s_k) = 0, \quad k = 1, \ldots, K, \tag{2.14e}$$

$$g_{\text{switch}}(\bar{x}(t_k), s_{k-1}, s_k) \leq 0, \quad k = 1, \ldots, K. \tag{2.14f}$$

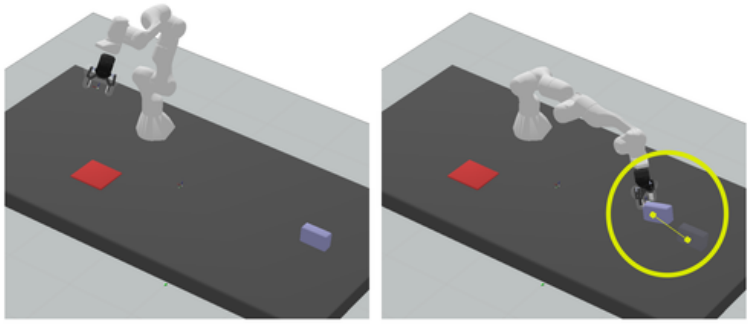
where $\langle s_0, \ldots, s_K \rangle$ is uniquely defined by $\langle a_1, \ldots, a_K \rangle$ and the fixed $s_0$ (using the discrete successor function). This optimization can be converted directly into an NLP by using a finite-dimensional representation of the trajectory, and thus, we refer to (2.14) directly as an NLP. Throughout this thesis, we represent the trajectory with a finite set of waypoints, similar to the example NLP in Eq. (2.11) of Section 2.1.

A task plan $\langle a_1, \ldots, a_K \rangle$ is said to be geometrically infeasible when (2.14) is infeasible, i.e., it has no feasible solution. This frequently occurs in TAMP, where many candidate high-level plans define constraints for the motion that can never be satisfied. For instance, the task plan $\langle$*pick object A with robot Q from A_init*$\rangle$ fails if the object is too far away or if there is an obstacle blocking the grasp. A task plan $\langle$*pick object A with robot Q from A_init*, *place object A with robot Q on red table*$\rangle$ can fail if the table is out of reach or other objects are already on the table. These two examples of motion failures are shown in Fig. 2.7.

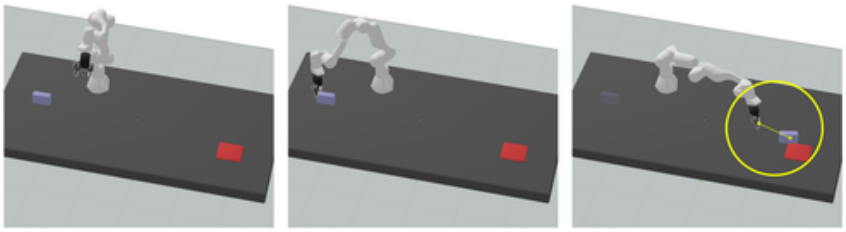
We can define relaxed versions of Trajectory-NLP($\langle a_1, \ldots, a_K \rangle$) that can be used to quickly test the feasibility while being computationally simpler. Relaxed problems remove constraints from the original problem and act as a lower bound. Thus, if the relaxed problem is infeasible, the original problem is also infeasible.

The Keyframes-NLP (*keyframes or sequence bound*) optimizes only a single configuration per phase instead of a continuous trajectory.

The optimization variables are $\{x_k \equiv x(t_k) \mid k = 1, \ldots, K\}$, which are optimized jointly, accounting for their interdependencies but without considering the continuous path between them. That is, we only evaluate the constraints at the beginning and end of each phase. This sequence of discrete configurations is called *keyframes* (the term we use in this thesis) or mode-switches and is very informative in manipulation planning, as keyframes are usually the more constrained configurations and capture with high accuracy whether a high-level



(a) Trajectory-NLP for the task plan ⟨*pick object A with robot Q from A_init*⟩ is infeasible.



(b) Trajectory-NLP for the task plan ⟨*pick object A with robot Q from A_init, place object A with robot Q on red table*⟩ is infeasible.

Figure 2.7.: Two examples of infeasible Trajectory-NLPs for two task plans in two example environments. The yellow circle highlights why the motion is infeasible. In *(a)*, the block suddenly "jumps" from the start configuration to the gripper. In *(b)*, the block "jumps" from the gripper to the red table.

task plan is geometrically feasible. The Keyframes-NLP($\langle a_1, \ldots, a_K \rangle$) is:

$$\min_{x_1,\ldots,x_K} \sum_k \tilde{f}_0(x_k) + \tilde{f}_1(x_{k-1}, x_k)\,, \tag{2.15a}$$

$$\text{s.t. } \tilde{h}_{\text{path}}(x_k, s_k) = 0, \qquad k = 1, \ldots, K \tag{2.15b}$$

$$\tilde{g}_{\text{path}}(x_k, s_k) \leq 0, \qquad k = 1, \ldots, K \tag{2.15c}$$

$$\tilde{h}_{\text{switch}}(x_k, x_{k+1}, s_k, s_{k+1}) = 0, \qquad k = 0, \ldots, K-1 \tag{2.15d}$$

$$\tilde{g}_{\text{switch}}(x_k, x_{k+1}, s_k, s_{k+1}) \leq 0, \qquad k = 0, \ldots, K-1 \tag{2.15e}$$

where $(\tilde{h}, \tilde{g})_{(\text{path,switch})}$ model the path and switch constraints but are evaluated only on the keyframe configurations instead of the full trajectory.

Importantly, in the Keyframes-NLP, we optimize the full manipulation sequence jointly. This means that we can discover, for instance, grasp locations that are good for both the pick and place keyframe, picking in places that allow for a handover, or placing an object close to the other robot for later manipulation.

In practice, before solving (2.14), it is recommended to always solve first the Keyframes-NLP (2.15), and use its solution to warm-start the trajectory in (2.14).

A looser relaxation is to optimize a single keyframe for each phase independently, without considering the interdependencies between them. The Pose-NLP($\langle a_1, \dots, a_K \rangle$) (*pose bound*) is a set of $k = 1, \dots, K$ independent optimization problems, one for each keyframe $x_k$,

$$k = 1, \dots, K \quad \min_{x_k} \tilde{f}_0(x_k, s_k)\,, \tag{2.16a}$$

$$\text{s.t. } x(0) = x_0\,, \tag{2.16b}$$

$$\tilde{h}_{\text{path}}(x_k, s_k) = 0\,, \tag{2.16c}$$

$$\tilde{g}_{\text{path}}(x_k, s_k) \leq 0\,. \tag{2.16d}$$

This is the most computationally efficient relaxation to test the feasibility of a sequence of actions but is also the least informative, as it does not consider the interdependencies between the different motion steps.

### 2.3.1. Multi-Bound Tree Search

Multi-Bound Tree Search (MBTS) [Toussaint and Lopes, 2017] is a state-of-the-art approach to solve an LGP (2.13). The discrete state and action space of the LGP formulation is explored with a search tree starting from $s_0$, where each branch represents a different sequence of discrete actions. The leaf nodes $s_g \in S_g$ are solutions to the discrete planning problem and are potential candidates for a solution to the LGP problem. Each node can be tested for feasibility by solving the continuous optimization problem induced by the sequence of actions from the root to the current node. Therefore, to find a solution to the LGP problem, we have to identify a leaf node $s_g \in S_g$ and find a feasible solution to its corresponding Trajectory-NLP($a_{1:K}$) (which might be infeasible for most candidate plans).

However, trying to solve the trajectory optimization is expensive, and the number of candidate NLPs is generally too high. To alleviate this issue, MBTS solves relaxed versions of (2.13) incrementally. The feasibility of each relaxed problem is a necessary condition for the feasibility of the Trajectory-NLP, i.e., these act as lower bounds while being computationally faster. Specifically, the two bounds are the *keyframes bound* (Keyframes-NLP (2.15)) and the *pose bound* (Pose-NLP (2.16)), which consider only a subset of variables and constraints of the full trajectory optimization problem.

The search is organized around four queues using a simple round-robin policy to process the next element of each queue. There is a queue for intermediate discrete states (explored in a breadth-first search order), two queues for computing the relaxations using the Keyframes-NLP and the Pose-NLP, and a queue for solving the Trajectory-NLP.

If a node in the tree reaches the goal, it is promoted to the relaxation queues. If a node fulfills the two relaxations, it is moved to the trajectory optimization queue. Additionally, we can also use the relaxations to prune the search tree. If the pose, keyframes, or trajectory NLP of a candidate or intermediate node fails, we can identify the prefix that is infeasible and utilize it to prune sub-branches in the search tree.

## 2.4. Related Work in Task and Motion Planning

Solvers for Task and Motion Planning can be categorized based on two complementary criteria [Garrett et al., 2021]. First, based on how motion planning and task planning are combined, we distinguish between *1) continuous-first* (where partial motions are first computed, then combined into a full TAMP solution), *2) interleaved* (with simultaneous search at both the continuous and discrete levels), and *3) discrete-first* (where candidate high-level task plans are computed first).

The second criterion focuses on the methods used to compute the motions. These are primarily *1) predefined discretization*, *2) sampling*, or *3) optimization methods*. In practice, the two criteria are closely interconnected; solvers using *optimization* typically employ a *discrete-first* search, while most *sampling* methods use *continuous-first* or *interleaved* search.

Prominent examples of sample-based methods are PDDLStream [Garrett et al., 2020] and TAMP in the Now [Kaelbling and Lozano-Pérez, 2011]. TAMP in the Now adopts a hierarchical and interleaved search approach between the discrete and continuous levels, while PDDLStream integrates constrained samplers for generating the continuous motion (e.g., grasps, collision-free paths, and inverse kinematic solutions) within PDDL-like planning. On the other hand, the study in [Ferrer-Mestres et al., 2017] showcases a pre-discretized continuous-first approach, where a set of valid configurations is integrated into task planning through precompilation.

Some sampling-based TAMP solvers reason explicitly about geometric conflicts. For example, a set of predefined predicates such as "is reachable" is used in [Srivastava et al., 2014] to combine a black-box task planner with a motion planner. The constraint-based approach in [Dantam et al., 2016] incorporates information about geometric infeasibility by blocking the full task plan or, in special cases, a pair of a discrete (partial) state and an action. Geometric infeasibility can also be evaluated efficiently using linear constraint propagation [Lagriffoul et al., 2014].

In this work, we focus on optimization-based formulations of TAMP, where Logic Geometric Programming stands as a leading general formulation [Toussaint, 2015, Toussaint et al., 2018]. Optimization-based solvers leverage nonlinear optimization to compute motions that satisfy all geometric and physical constraints, while taking into account the interdependencies in the motion. A state-of-the-art general solver for LGP is the Multi-bound Tree Search [Toussaint and Lopes, 2017], which combines discrete search with relaxed optimization problems to efficiently evaluate geometric feasibility. Other optimization-based methods, e.g., [Migimatsu and Bohg, 2020, Zhao et al., 2021, Zimmermann et al., 2020, Hadfield-Menell et al., 2016], address more specific TAMP settings (e.g., rearrangement) or a TAMP subproblem (e.g., only motion planning).

Task and Motion Planning can also be formulated as multimodal motion planning. Indeed, the difference between TAMP and multimodal motion planning is mainly a naming convention used by different authors. In practice, the concepts of *motion-modes* and *mode-transitions* in multimodal motion planning correspond to the high-level abstraction in TAMP problems. The naming convention traditionally highlights a slight difference in target applications, with TAMP focusing more on planning long manipulation sequences with multiple objects, while multimodal planning emphasizes more on problems with shorter sequences with more challenging motion planning.

Multi-Modal-PRM, proposed in the seminal work [Hauser and Latombe, 2010], builds a probabilistic roadmap (PRM, [Kavraki et al., 1996]) in different motion modes and connects these modes by sampling configurations belonging to two modes simultaneously, known as mode-switch configurations. The original multimodal framework can be extended to problems with an infinite number of modes [Hauser and Ng-Thow-Hing, 2011], asymptotic optimal planning [Vega-Brown and Roy, 2020], and heuristics to bias the search towards useful mode transitions [Kingston et al., 2020].

Problems similar to TAMP or multimodal motion planning also appear under a third distinct name in robotics literature: manipulation planning, which typically assumes contact modes using only a stable grasp (i.e., prehensile manipulation).

Similar to multimodal motion planning, most methods extend the tools of sample-based motion planning to manipulation problems. For instance, manipulation planning can be formulated as a search over a sequence of transit paths (where the robot moves freely) and transfer paths (where the robot moves while holding an object), using probabilistic roadmaps [Siméon et al., 2004]. More recently, an asymptotically optimal manipulation planner has been proposed in [Schmitt et al., 2017], and the Manipulation-RRT [Lamiraux and Mirabel, 2021] extends the classical RRT algorithm ([LaValle and Kuffner, 2001]) to plan across different contact and manipulation modes. Some algorithms focus only on specific settings within manipulation planning, such as rearrangement planning [Ota, 2004, Krontiris and Bekris, 2016, Huang et al., 2019], or navigation among movable obstacles [Stilman et al., 2007].

The multimodal nature of Task and Motion Planning (TAMP) arises from creating and breaking contacts with the environment. Such problems also appear in legged locomotion, where mixed-integer formulations can be used to optimize foot placement, gait, and joint movement [Deits and Tedrake, 2014]. To avoid the explicit combinatorial search, an alternative approach is to use local optimization methods, which, in turn, raise concerns about local optima and feasibility. Differentiable contact models [Todorov, 2011], contact invariant optimization [Mordatch et al., 2012], nonlinear programming [Posa et al., 2014], and convex relaxations [Song et al., 2021] have been used to optimize trajectories and contacts simultaneously for locomotion.

From a different perspective, and within a different research community, Task and Motion Planning can be formulated as a classical planning problem with additional numerical variables. Classical AI planners have been extended to support planning with numerical constraints on action preconditions (e.g., Metric-FF [Koehler, 1998]), and recent versions of the Planning Domain Definition Language (PDDL) include temporal planning with numerical variables [Fox and Long, 2006, Piotrowski et al., 2016, Scala et al., 2016].

For instance, the COLIN planner [Coles et al., 2012b] includes continuous linear changes of numerical variables (e.g., fixed velocities) and encodes the temporal and state evolution constraints implied by a sequence of actions as a linear program. The Scotty planner [Fernández-González et al., 2018] adds support for general convex constraints, combining discrete search with convex optimization, and the planner in [Haslum et al., 2018] extends classical planning with general state constraints. However, these general planning formulations have not been used to tackle general TAMP problems, where the dimensionality and complexity of the continuous space pose significant challenges and often require tools from motion planning and nonlinear trajectory optimization.

# Chapter 3

# Factored Structure of Task and Motion Planning

In this chapter, we analyze the factored structure that appears in Task and Motion Planning (TAMP). Specifically, we study the factorization of the nonlinear trajectory optimization problem for a fixed task plan, denoted as Trajectory-NLP($\langle a_1, \dots, a_K \rangle$) in Section 2.3 (Eq. (2.14)).

A similar analysis of the factorization of the TAMP problem is foundational in modern sample-based TAMP solvers, where it is used to define effective conditional sampling operations [Garrett et al., 2018, Garrett et al., 2021] and to propagate information about feasibility [Lagriffoul et al., 2014]. In contrast, we analyze the factored structure from the perspective of optimization-based solvers for TAMP, providing complementary insights and ideas.

One of the contributions of this thesis is the formalization of TAMP problems using our novel planning formulation, "Planning with Nonlinear Transition Constraints", which offers a factored view on LGPs. While the formal definition, details, and comprehensive analysis will be presented later in Chapter 5, this chapter provides an approachable, intuitive view of the inherent structure that appears naturally in trajectory optimization for TAMP problems.

The chapter is organized as follows: First, we introduce the factored nonlinear program formulation. Second, we discuss the factored optimization problem for the Pick and Place task plan, which will later be used as a building block in more sophisticated manipulation tasks. Third, we examine more complex examples that showcase the main benefits of our factored representation, namely, the temporal structure, sparse factorization, and composition.

## 3.1. Factored-NLP – Definition and Properties

A factored nonlinear program (Factored-NLP) is a nonlinear program (2.1) where the vector variable is factored into a set of smaller vector variables, and the cost function, the equality, and inequality constraints are also decomposed into a set of smaller cost terms and constraint functions, each depending on a small subset of the variables.

Given a set of $N$ vector variables $X = \{x_1, \ldots, x_N\}$ with $x_i \in \mathbb{R}^{n_i}$, a set of nonlinear cost functions $F = \{f_1, \ldots, f_B\}$ with $f_b : \mathbb{R}^{m_b} \to \mathbb{R}$, and a set of vector-valued constraint functions $\Phi = \{\phi_1, \ldots, \phi_A\}$ with $\phi_a : \mathbb{R}^{m_a} \to \mathbb{R}^{m'_a}$ that include both equality and inequality constraints, a Factored-NLP is the optimization problem,

$$\min_{x_1,\ldots,x_N} \sum_{f_b \in F} f_b(X_b)\,, \tag{3.1a}$$

$$\text{s.t.} \quad x_i \in \mathbb{R}^{n_i}\,, \qquad i = 1, \ldots, N \tag{3.1b}$$

$$\phi_a(X_a)\ \{\leq, =\}\ 0\,, \qquad \forall \phi_a \in \Phi \tag{3.1c}$$

where each cost function $f_b$ and constraint $\phi_a$ may depend on different subsets of variables $X_b \subseteq X$, $X_a \subseteq X$ (e.g., $X_a = \{x_1, x_3\}$ for some $a$). The notation $\{\leq, =\}$ indicates that a constraint can be either an equality or an inequality constraint, which are handled differently by nonlinear solvers.

In this thesis, we often use Factored-NLPs to reason about the infeasibility of optimization problems and to generate one or more feasible solutions that fulfill the constraints. Thus, we consider Factored-NLPs without a cost term, resulting in the feasibility problem,

$$\text{find } x_i \in \mathbb{R}^{n_i}, \qquad i = 1, \ldots, N \tag{3.2a}$$

$$\text{s.t.} \quad \phi_a(X_a)\ \{\leq, =\}\ 0, \qquad \forall \phi_a \in \Phi. \tag{3.2b}$$

A Factored-NLP is a structured representation similar to factor graphs [Frey et al., 1997, Dellaert et al., 2017], constraint systems [Rossi et al., 2006], and graphical models [Koller and Friedman, 2009]. It can be represented with a bipartite graph with two types of vertices, variables and constraints, where edges represent the dependency relations between them.

We can represent a Factored-NLP with variables $X_G = \{x_1, \ldots, x_N\}$ and constraints $\Phi_G = \{\phi_1, \ldots, \phi_A\}$ as a graph $G = (V_G, E_G)$ with vertices $V_G$ and edges $E_G$,

$$V_G = X_G \cup \Phi_G\,, \tag{3.3a}$$

$$E_G = \{(x_i, \phi_a) \mid \text{ constraint } \phi_a \in \Phi_G \text{ depends on variable } x_i \in X_G\}. \tag{3.3b}$$

Subproblems of Factored-NLPs are defined as *subgraphs* of the original graph $G$. Namely, a subset of variables and constraints is a subgraph $M \subseteq G$ of the original Factored-NLP. Likewise, a superset of variables and constraints is a supergraph $\tilde{G} \supseteq G$.

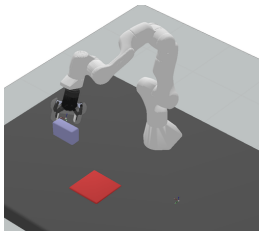
A Factored-NLP $G$ is feasible, $\mathcal{F}(G) = 1$, if the optimization problem (3.1) has a solution (i.e., if there exists a value assignment for all variables that fulfills all constraints),

$$\mathcal{F}(G) = 1 \quad \text{iff} \quad \exists\, x_i \in \mathbb{R}^{n_i}\ i = 1, \ldots, N, \text{ such that } \phi_a(X_a)\ \{\leq, =\}\ 0, \quad \forall \phi_a \in \Phi\,. \tag{3.4}$$

Otherwise, it is infeasible, with $\mathcal{F}(G) = 0$. Note that if a Factored-NLP $G$ is feasible, then all its subgraphs $M \subseteq G$ are feasible. Conversely, if a Factored-NLP is infeasible, then all its supergraphs $\tilde{G} \supseteq G$ are infeasible.

Similar to unstructured NLPs (Section 2.1), one can solve a Factored-NLP using joint non-linear optimization of all variables and constraints. If desired, the structured representation can be used for faster matrix factorization and multiplication using sparse matrices. In fact, leveraging the temporal structure of motion planning, we can reduce the computational complexity of second-order methods (Newton/Gauss-Newton) from cubic to linear in the temporal dimension.

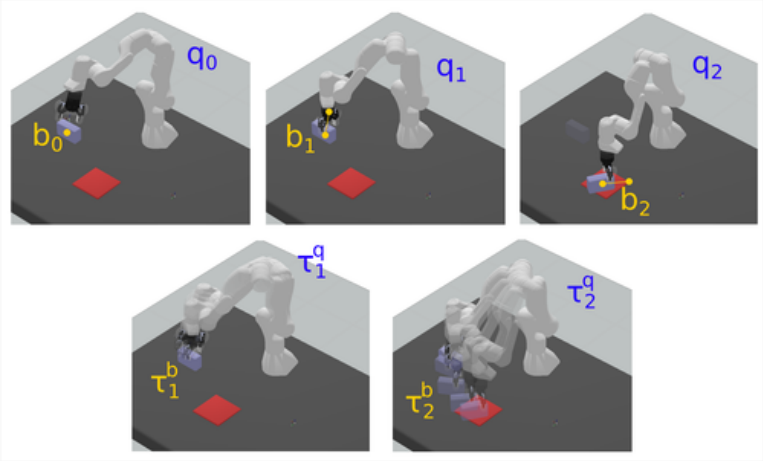
Additionally, since we now have a set of variables and constraints, we can solve for any subset of them, ignoring the other variables and constraints. In particular, we can attempt to solve the complete problem by computing subsets of variables sequentially in any order, conditioned on the variables that have already been determined, as later discussed in Chapter 6.



(a) Environment with a robot $Q$, an object $B$, and a red table.

```
pick object B with robot Q from B_init,
place object B with robot Q on red table.
```

(b) Task plan.




(c) Trajectory optimization problem. The top row illustrates the keyframes, and the bottom row shows the trajectory between keyframes. The initial configuration is $q_0$ for the robot and $b_0$ for the object. Variables $q_1$ and $b_1$ are the robot and object configurations in the pick keyframe, and $q_2$ and $b_2$ are the configurations in the place keyframe. Variables $\tau^q_{\{1,2\}}$ and $\tau^b_{\{1,2\}}$ indicate the trajectories of the robot and the object between keyframes. The images show a valid solution.

Figure 3.1.: Pick and Place – Environment, task plan, and trajectory optimization problem.

## 3.2. Pick and Place – The Basic Building Block

We analyze the trajectory optimization problem for the Pick and Place task plan, which is the basic building block in manipulation planning. The problem is shown in Fig. 3.1. (a) Shows the environment with the robot and the object, (b) shows the chosen Pick and Place task plan, and (c) represents the trajectory optimization problem that we have to solve to compute a robot motion for this task plan.

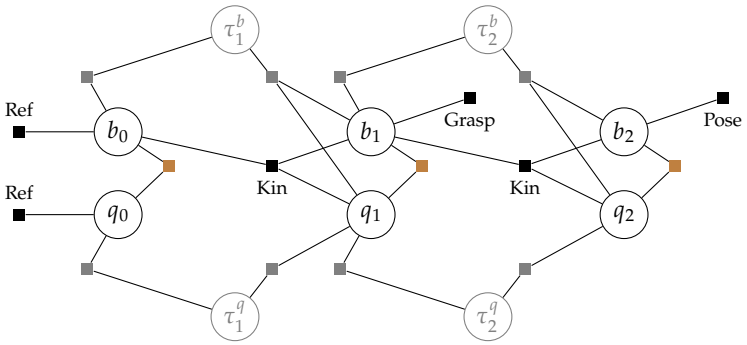



Figure 3.2.: Factored-NLP of a Pick and Place task plan (see Fig. 3.1). Circles represent variables and squares represent constraints. The vertical slices represent the initial state, the two keyframes of the manipulation plan, and the trajectories between them. Variables $q$, $b$, $\tau_q$, and $\tau_b$ are, respectively, the robot configuration, the object pose, the robot trajectory, and the object trajectory. The subindices denote the time step. The meaning of each constraint (*Ref*, *Kin*, *Grasp*, *Pose*) is explained in the main text. Brown squares represent collision avoidance constraints between the object, robot, and environment. Gray squares indicate the boundary value constraints between trajectories and keyframes. For clarity, joint limits on $q$ and additional constraints on the trajectories, such as zero velocity or collision avoidance within trajectories, are not drawn, but they are detailed in the main text.

The graphical representation of the Factored-NLP is shown in Fig. 3.2. Circles represent variables, squares represent constraints, and edges indicate the dependencies between variables and constraints. In the following, we explain the meaning of the variables and constraints that appear in the Factored-NLP in Fig. 3.2.

**Variables** Variables in the optimization problem represent the robot and object configurations in each keyframe $(q, b)$ and the trajectories between the keyframes $(\tau^b, \tau^q)$. The subscript indicates the time step. Variable $q_0 \in \mathbb{R}^7$ is the initial configuration of the robot, $q_1 \in \mathbb{R}^7$ is the configuration in the pick keyframe, and $q_2 \in \mathbb{R}^7$ is the configuration in the place keyframe.

The variable $\tau_1^q \in \mathbb{R}^{7 \cdot 20}$ is the trajectory from the start configuration to the pick keyframe, and $\tau_2^q \in \mathbb{R}^{7 \cdot 20}$ is the trajectory from the pick keyframe to the place keyframe. The trajectories are represented using a finite set of waypoints, e.g., 20, thus $\tau_1^q, \tau_2^q \in \mathbb{R}^{7 \cdot 20}$.

The variables $b$ for the object pose represent the relative transformation of the object with respect to its parent frame, as defined by the task plan. Variable $b_0 \in SE(3)$ is the initial pose with respect to the world frame, $b_1 \in SE(3)$ is the object pose with respect to the

gripper in the pick keyframe, and $b_2 \in SE(3)$ is the object pose with respect to the table in the place keyframe. We also introduce two variables for the trajectories of the object, $\tau_1^b$ and $\tau_2^b$, relative to their parent frames. Note that in the case of prehensile manipulation, such trajectories will be constrained to have zero velocity.

**Constraints** The constraints of the optimization problem are:

- $Kin(b_0, b_1, q_1)$: when the robot picks up an object, the object pose with respect to the gripper $b_1$ is determined by the position of the end-effector $p(q_1)$ and the absolute position of the object $b_0$. This is implemented with: $b_0 = p(q_1) \oplus b_1$, where $p(q) : \mathbb{R}^7 \rightarrow SE(3)$ is the pose of the end-effector with respect to the world frame, as a function of the robot configuration, and $\oplus$ is the addition operator in $SE(3)$.
- $Kin(b_1, b_2, q_2)$: similarly, when the robot places the object, the pose with respect to the table $b_2$ is determined by the position of the end-effector $p(q_2)$ and the relative grasp $b_1$.
- $Grasp(b_1)$: To ensure a stable grasp, the pose of the object with respect to the gripper must be such that the object does not fall. This can be represented with a nonlinear function based on different grasp models (e.g., we can constrain it to be a top grasp, a grasp that encapsulates the object between the two fingers, or a simpler touch-grasp).
- $Pose(b_2)$: Similarly, the placement of the object in the place keyframe is constrained to be stable. This can be represented with a nonlinear function based on different placement models (e.g., a top placement of a block on a surface).
- $Ref(q_0)$: The robot configuration in the initial state is fixed to the given value.
- $Ref(p_0)$: The object pose in the initial state is fixed to the given value.
- Collision constraints (brown squares in Fig. 3.2) are evaluated by calculating the distance between the collision shapes of the robot, movable object, and static environment. Collision avoidance constraints are also added to the trajectory variables, applying the constraint to each waypoint.
- Joint limits on the robot configurations $q_1, q_2$ and the robot trajectories $\tau_1^q, \tau_2^q$ are also added to the optimization problem (not shown in Fig. 3.2).
- Boundary value constraints (gray squares in Fig. 3.2) tie the keyframes and the trajectories together.
- Zero Velocity: the object pose is always stable with respect to its parent frame, either the world frame when it lies in the start position, or the gripper when held by the robot. This is represented with the equality constraints $Vel0(\tau_1^b)$, $Vel0(\tau_2^b)$ (not shown in Fig. 3.2).

Adding constraints between keyframes directly, e.g., $Kin(b_0, b_1, q_1)$, is essential for efficient TAMP, as it results in very informative relaxations of the Factored-NLP when removing the trajectory variables. In the case of a stable grasp, we can add equality constraints (*Kin*) directly between the variables in the keyframes, which do not depend on the intermediate trajectory. For pushing interactions, we can add inequality constraints between keyframes that over-approximate the reachability and affordances of a pushing motion.

**Cost** The Factored-NLP in Fig. 3.2 does not show any cost term, but we can add convex quadratic costs with two purposes: to favor solutions with short and smooth trajectories, and to act as a regularization in the optimization process, improving the convergence behavior and success of the optimizer. For instance,

- Squared acceleration cost on the robot trajectories. Because trajectories are represented by a finite set of waypoints, we can compute the acceleration at each waypoint using finite differences.
- Distance between robot configurations in the keyframes, e.g., $\|q_1 - q_2\|^2$.
- Regularization on robot configurations and object poses. For instance, $\|q_1 - q_{\text{ref}}\|^2$ with a reference configuration $q_{\text{ref}}$, that avoids singular configurations.

## 3.3. Complex Manipulation Sequences

In this section, we analyze the Factored-NLP for longer manipulation sequences in environments with more objects and robots. Our goal is to illustrate the scalability and unique properties of this representation, which contribute to the design of efficient planning and learning algorithms throughout the thesis. The three essential properties of our Factored-NLP formulation are:

- *Temporal Markov structure:* Constraints in the graph only connect variables from consecutive time steps, maintaining the sequential temporal structure of a planning problem.
- *Sparse factorization:* Each constraint depends only on a small subset of variables. This becomes particularly clear in scenarios involving multiple robots and objects. For instance, when a robot picks up an object, additional constraints are added between that specific robot and object, but without affecting the other objects or robots in the scene.
- *Repeatable local structure:* The optimization problem for a new manipulation task contains several small building blocks that are repeated across different task plans. For instance, when comparing the new Factored-NLPs in Figs. 3.4 and 3.5 to the

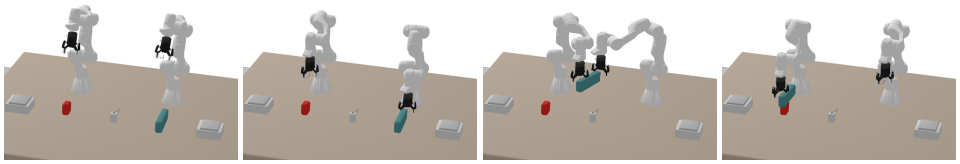


Figure 3.3.: Example domain with two objects and two robots.

Pick and Place example, similar structures with small variations are used to model handovers or the placement of one object on top of another.

To achieve these three properties in the Factored-NLP, it is fundamental to use a *non-minimal* representation of the optimization problem. Here, non-minimal means that, given a task plan, there often exists a smaller Factored-NLP equivalent to our formulation, using fewer variables and constraints. For instance, note that we have added the fixed initial state as a variable (along with constraints) to more clearly expose the temporal structure of the problem. Furthermore, when an object does not move, we also add new variables, which are constrained to be equal. While our non-minimal formulation is superior for planning and learning algorithms, solving the Factored-NLP with a non-minimal formulation incurs a small runtime penalty. If required, before employing a nonlinear solver to solve for the full Factored-NLP or a subset, one can remove and merge fixed or equal variables using a preprocessing step.

**A domain with two robots and two objects** In our second example, we consider an environment with two movable objects, $A$ and $B$, initially at *A_init* and *B_init*, and two robot manipulators, $Q$ and $W$. The high-level goal is to stack $A$ on top of $B$.

We discuss the structure of the Factored-NLP for two candidate task plans that can potentially achieve the high-level goal. Fig. 3.4 shows the Factored-NLP for the task plan $\langle$*pick B with Q from B_init*, *pick B with W from Q*, *place B with W on A*$\rangle$.

In each vertical slice of the Factored-NLP, we have continuous variables $\{a, b, q, w\}$ for keyframe configurations, and $\{\tau^a, \tau^b, \tau^q, \tau^w\}$ for trajectories. Variables $a, b$ are the poses of the two objects with respect to the parent frame in the kinematic chain, and $q, w$ are the robot joint configurations. Variables $\tau^a, \tau^b, \tau^q, \tau^w \in \mathbb{R}^{20\cdot 7}$ are the corresponding trajectories during each motion phase (represented with 20 waypoints). In comparison to the Pick and Place example, we now have four keyframe variables per vertical slice (instead of 2), and 4 vertical slices (instead of 3) because the task plan is longer. A feasible solution of the Factored-NLP is shown in Fig. 3.3.

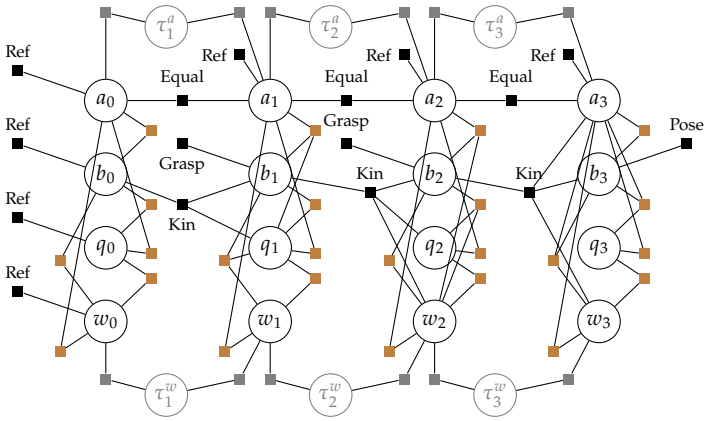



Figure 3.4.: Factored-NLP for the task plan ⟨*pick B with Q from B_init*, *pick B with W from Q*, *place B with W on A*⟩. We display all variables for the keyframe configurations $(a, b, q, w)$, and the trajectory variables only for $(\tau^a, \tau^w)$. We omit the variables $\tau^b, \tau^q$, and the constraints for zero-velocity and collisions between trajectories to keep the illustration cleaner. Brown squares represent collision avoidance constraints. Gray squares represent boundary constraints between trajectories and keyframes.

Interestingly, most of the constraints are the same as in the Pick and Place example. For instance, the collision avoidance, grasping, and positioning constraints are the same. However, this problem contains some unique, but related structures,

- In the second step, robots perform a handover. We use a *Kin* constraint, which is essentially similar to the *Kin* constraint in the Pick and Place example, but now it connects the two robots with object $B$ because the robot $W$ picks the object from the other robot $Q$.
- The *Kin* constraint when placing the object is also different because now the object is placed on top of another object.
- Finally, there is a novel *Equal* constraint between consecutive variables for objects that are not manipulated (in this case, object $A$), to ensure they remain still.

In Fig. 3.5, we show the Factored-NLP for an alternative plan: ⟨*pick B with Q from B_init*, *place B with Q on A*⟩. Note how the Factored-NLP has three columns instead of four, and a different global structure, while retaining many common local structures and relationships between variables and constraints.

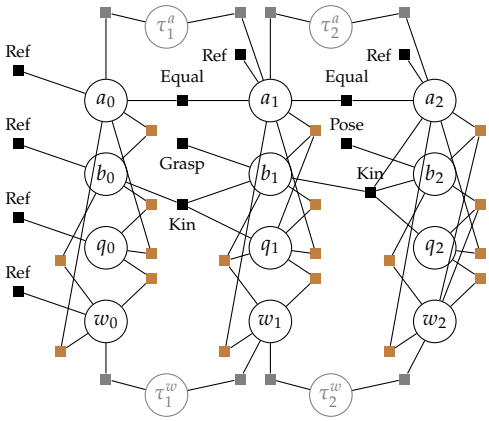



Figure 3.5.: Factored-NLP for the task plan ⟨*pick B with Q from B_init*, *place B with Q on A*⟩ in the domain referenced in Fig. 3.3. See main text and caption of Fig. 3.4.

Our novel planning formulation, *Planning with Nonlinear Constraints*, dictates which constraints appear in the graph, depending on the high-level task plan. A formal definition and analysis will be provided later in Chapter 5.

We emphasize that a single Factored-NLP cannot represent the full TAMP problem because the Factored-NLP is conditioned on the task plan, which is a priori unknown and should also be optimized within a TAMP problem. In difficult TAMP problems, most Factored-NLPs are infeasible since the candidate task plan fails when considering collisions and physics constraints. This is illustrated in our example: from the two Factored-NLPs shown in Figs. 3.4 and 3.5, only the first one is feasible in the environment shown in Fig. 3.3. The second plan would fail because the robot $Q$ cannot place object $B$ correctly on top of object $A$ because $A$ is too far.

# Part I.

# Integrated Planning and Optimization for Task and Motion Planning

# Chapter 4

# Diverse Task Planning for Solving Logic Geometric Programs

## 4.1. Introduction

In this chapter[1], we combine state-of-the-art diverse classical planning with trajectory optimization within the Logic Geometric Program (LGP) formulation.

A central challenge in Task and Motion Planning (TAMP) is the integration of information and tools from both the discrete and continuous domains. Efficient solvers for either the discrete or the continuous levels of TAMP problems are readily available, such as trajectory optimization using constrained optimization or classical planning with heuristic search. However, these tools are not directly applicable to the full problem, and their integration is neither trivial nor direct.

This limitation has led to the creation of custom TAMP solvers that can better reason about the dependencies between logic and geometry. However, these solutions often fall short in performance compared to mature, state-of-the-art solvers for each of the subproblems. The Multi-Bound Tree Search (MBTS) serves as a prominent example, being a leading solver for TAMP problems. It employs a custom tree search algorithm that explores the search space in a breadth-first manner, combining information from both geometry and logic with a multi-bound strategy.

Our new solver, *Diverse Planning for LGP*, interfaces high-level task planning with low-level trajectory optimization by identifying geometric conflicts in the form of infeasible plan

[1]This chapter is based on the paper: Ortiz-Haro, J., Karpas, E., Toussaint, M., and Katz, M. (2022). Conflict-directed Diverse Planning for Logic-Geometric Programming. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 32, pp. 279-287).

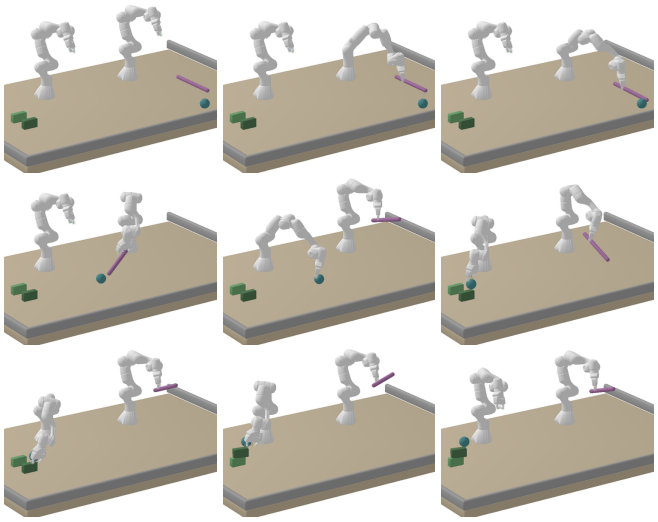


Figure 4.1.: Example of a TAMP problem solved by our algorithm. The objective is to compute the task plan and the continuous motion that achieves the high-level goal *(ball on block A) and (block A on block B)* (bottom right) from the initial configuration (top left). This requires combined logic and geometric reasoning about tool-use, pushing, and pick and place actions with two robot manipulators.

prefixes, and employs a new multi-prefix forbidding compilation to transmit this information back into the task planner. Additionally, we leverage diverse planning with a new novelty criterion for selecting candidate plans based on prefix novelty, and a metareasoning approach which attempts to extract only useful conflicts by leveraging the information gathered in the course of solving the given problem.

While Multi-Bound Tree Search can use a similar notion of geometric prefix-conflicts internally in the custom tree search, our approach enables the combination of off-the-shelf state-of-the-art PDDL planning with trajectory optimization. This combination allows us to solve problems that require complex reasoning at both the continuous and discrete levels more quickly. For instance, the manipulation task shown in Fig. 4.1 requires reasoning about tool-use, pushing, and pick and place actions with two robot manipulators.

The enhancement in the discrete search makes the solver faster, facilitating the generation of multiple candidate plans and allowing for the incorporation of new ideas and techniques from the planning community such as diverse planning and metareasoning.

We demonstrate that our combination not only accelerates the discrete search but also reduces the number of optimization problems solved in the continuous layer, resulting in faster solution times.

## 4.2. Related Work

A comprehensive review of related work in Task and Motion Planning is available in Section 2.4. The closest approaches to ours are sample-based TAMP solvers that attempt to identify geometric conflicts and encode this information back into the discrete descriptions, either using a set of predefined predicates [Srivastava et al., 2014] or by adding additional constraints to an incremental constraint satisfaction solver [Dantam et al., 2018].

We follow the LGP formulation of Task and Motion Planning and provide an alternative to the Multi-Bound Tree Search [Toussaint and Lopes, 2017] algorithm. The interface between the high-level task plan and the low-level motion, focusing on prefixes of task plans that are infeasible, is consistent across both solvers. However, the encoding techniques and tools used in the discrete search are more advanced in our approach, leading to a more efficient TAMP solver, as our experiments demonstrate.

Our solver employs tools from classical planning and conflict-based search to solve an LGP. These ideas and tools are further refined in the *Factored-NLP Planner* (Chapter 5), resulting in a more efficient bidirectional interface that exploits the factorization of TAMP problems.

## 4.3. Factorization of the Discrete State Space

*Diverse Planning for LGP* builds on top of Logic Geometric Programming (Section 2.3) and classical planning (Section 2.2). We refer to Section 2.3 for an extensive introduction to the LGP formulation and notation, which we adopt in this chapter.

In order to use off-the-shelf discrete planners, we first need a factorized representation of the discrete state space in terms of discrete variables. As exemplified in the *Blocksworld* domain (Section 2.2), this factorization is readily available if we consider an object- and robot-centric representation.

To transform the discrete search component of the LGP into a classical planning problem using the SAS+ encoding (Section 2.2), we introduce a discrete variable for each movable object and robot. Variables for objects, e.g., `parent_A` for block $A$, indicate where the object is in a discrete sense, e.g., on the table, on top of another object, or held by the gripper; but without defining the continuous relative transformation. Specifically, these variables indicate the "parent" in the kinematic tree and (implicitly) the type of interaction (e.g., stable pick by a gripper or push by a stick). Consequently, we use the term "parent", e.g., the discrete variable for object $A$ is called `parent_A`. Variables for robots indicate whether the robots are interacting with any object (e.g., the gripper can be full or empty).

Consider the scenario in Fig. 4.1 with two robots ($Q$ and $W$), two blocks ($A$ and $B$), a ball, and a stick, where robots can pick up the blocks and use the stick to push the ball. We can

represent the discrete state using six discrete variables: {`parent_A`, `parent_B`, `parent_Stick`, `robot_Q`, `robot_W`, `parent_Ball`}. The sets of possible values for each variable are:

- `parent_A` ∈ { `table`, `block_B`, `robot_Q`, `robot_W`, `A_init` },
- `parent_B` ∈ { `table`, `block_A`, `robot_Q`, `robot_W`, `B_init` },
- `parent_Stick` ∈ { `table`, `robot_Q`, `robot_W`, `Stick_init` },
- `robot_Q` ∈ { `free`, `full` },
- `robot_W` ∈ { `free`, `full` },
- `parent_Ball` ∈ { `table`, `block_A`, `block_B` , `robot_Q`, `robot_W`, `Ball_init`, `Stick` }.

A discrete state $s \in \mathcal{S}$ is a value assignment to all the variables. For instance, the initial state $s_0$ in Fig. 4.1 is:

```
parent_A = A_init,          parent_B = B_init,
parent_Stick = Stick_init,  robot_Q = free,
robot_W = free,             parent_Ball = Ball_init.
```

In a more concise way, we can also represent the initial state as a set of predicates that are true:

```
(A on A_init), (B on B_init), (Stick on Stick_init),
      (Q free), (W free), (Ball on Ball_init),
```

where, for example, (`A on A_init`) means that `parent_A = A_init`. Similarly, we can represent every discrete action $a \in \mathcal{A}$ in the LGP formulation as a pair of conditions and effects on the discrete variables. For instance:

- Action: `pick block B with robot Q from B_init`
  Conditions: `parent_B = B_init, robot_Q = free.`
  Effects: `robot_Q = full, parent_B = robot_Q.`
- Action: `place block B with robot Q on block A`
  Conditions: `parent_B = robot_Q. robot_Q = full.`
  Effects: `parent_B = block_A, robot_Q = free.`

## 4.4. Diverse Task Planning for LGP

The fundamental contribution of this chapter is the application of diverse planning at the task level of a Logic Geometric Programming (LGP) problem, yielding a varied sequence of task plans. *Diverse Planning for LGP* is an iterative Task and Motion Planning (TAMP) solver, where candidate task plans are checked for geometric feasibility by solving a trajectory

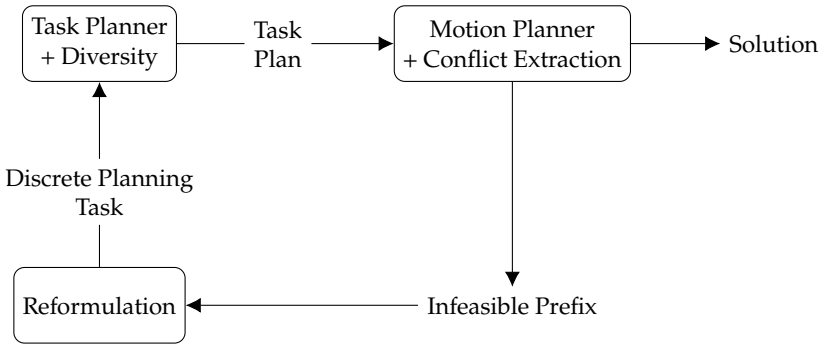



Figure 4.2.: Overview of our approach *Diverse Planning for LGP*. We combine a discrete task planner to generate task plans with a motion planner to compute the continuous trajectory. If a plan is not geometrically feasible, we extract a conflict, namely a prefix of discrete actions of the task plan, and reformulate the discrete planning task to block this prefix.

optimization problem. Crucially, if a task plan is not geometrically feasible, we extract a conflict and use this conflict to reformulate the planning task. Specifically, we build upon and extend the iterative plan forbidding approach [Katz and Sohrabi, 2020] to generate diverse and conflict-free candidate plans.

In this chapter, we address conflicts in the form of task plan *prefixes* – that is, a sequence of discrete actions $\pi = \langle a_1, \ldots, a_K \rangle$ which is applicable from the initial state at the discrete level but lacks a feasible geometric trajectory. To exploit such conflicts, in a manner akin to conflict-directed clause learning [Silva and Sakallah, 1999] or conflict-directed A* [Williams and Ragno, 2007], two steps are required. First, we must be able to efficiently extract a conflict from a task plan that is not geometrically feasible. Second, we must prevent our task planner from generating plans that contain the identified conflict as a prefix. Both of these will be elaborated on later.

The flowchart in Fig. 4.2 provides a graphical description, and Algorithm 4.1, which appears later, shows the pseudocode for our approach. Fig. 4.3 provides an example of the execution of our solver in a simplified setting.

### 4.4.1. Prefixes as Conflicts

To begin the detailed discussion of our approach, we discuss why we choose to identify prefixes as conflicts, and not a more general restriction on plans.

**Definition 4.1.** Given a sequence of discrete actions $\pi = \langle a_1, \ldots, a_K \rangle$, the prefix of length $k \leq |\pi| = K$ is denoted as,

$$\pi|_k = \langle a_1, \ldots, a_k \rangle \,. \tag{4.1}$$

**Theorem 4.1.** *Let $\pi = \langle a_1, \ldots, a_K \rangle$ be a sequence of discrete actions, such that $\pi$ is not geometrically feasible from the initial state. Then any sequence of actions $\pi'$ which contains $\pi$ as a prefix is not geometrically feasible from the initial state.*

*Proof.* Recall that a sequence $\pi = \langle a_1, \ldots, a_K \rangle$ of $K$ actions is not geometrically feasible if the nonlinear optimization Trajectory-NLP($\pi$) (Eq. (2.14)) is infeasible.

For the sake of contradiction, assume that, given an infeasible prefix $\pi$, there exists a longer task plan $\pi' = \langle \pi, a'_{1:J} \rangle$ that is feasible. If $\pi'$ is feasible, then there exists a geometric path $x(t),\ t \in [0, (K+J)T]$ that is feasible for Trajectory-NLP($\langle \pi, a'_{1:J} \rangle$). This would imply that $x(t),\ t \in [0, KT]$ is also a feasible solution for Trajectory-NLP($\pi$), because variables and constraints in the time interval $t \in [0, KT]$ are the same in Trajectory-NLP($\langle \pi, a'_{1:J} \rangle$) and Trajectory-NLP($\pi$). This is a contradiction because $\pi$ is said to be infeasible. Therefore, $\pi' = \langle \pi, a'_{1:J} \rangle$ cannot be feasible. □

For example, consider an LGP task in which a single robot $Q$ can pick and place objects $A$ and $B$ on a cluttered table. Suppose the starting sequence:

$$\langle \texttt{pick B with Q from B\_init}, \texttt{place B with Q on table} \rangle,$$

is infeasible from the initial discrete state [`parent_A=A_init`, `parent_B=B_init`, `robot_Q = free`]. Then it is safe to infer that any action sequence beginning with this sequence is also infeasible, e.g., ⟨ `pick B with Q from B_init`, `place B with Q on table`, `pick A with Q from A_init` ⟩.

However, it is *not* safe to infer that ⟨`pick B with Q from B_init`, `place B with Q on table`⟩ can never be applied. For example, it is possible that object $A$ in the initial position is obstructing the placement of object $B$. Thus, the sequence of actions:

⟨ `pick A with Q from A_init`, `place A with Q on table`, `pick B with Q from B_init`, `place B with Q on table` ⟩

might be geometrically feasible. In this example, `pick A` and `place B` form a causal link [Tate, 1977], as `pick A` supports `place B`. However, this causal link does not appear at the discrete level but only at the geometric level. It is not possible to infer a stronger conflict than prefixes without a deeper analysis of geometric feasibility (Chapter 5).

Prefix forbidding is a general and sound way to encode information from the geometric level back into the discrete level. It does not rely on hand-crafted additional predicates or checks and is therefore applicable to any sequence of actions, independently of the underlying physics or geometry model.

### 4.4.2. Forbidding Plans by Prefixes

This section describes how to prevent a discrete planner from returning task plans which begin with a given set of prefixes, found to be geometrically infeasible in a previous iteration.

Our approach builds upon previous work [Katz et al., 2018a], which has suggested a *plan forbidding reformulation*, a method of constructing a planning task with a set of valid plans being reduced by precisely the given plan. The suggested construction follows the execution of the given task plan $\langle a_1, \ldots, a_K \rangle$ and allows one to achieve the (modified) goal only once an action different from $a_k$ is applied at step $k$.

We modify this construction in two ways. First, instead of forbidding $\langle a_1, \ldots, a_K \rangle$ as a plan, we forbid it as a prefix. Thus, applying the starting sequence $\langle a_1, \ldots, a_K \rangle$ in the reformulated task leads to a dead end. Consequently, there are no plans for the reformulation with the prefix $\langle a_1, \ldots, a_K \rangle$.

Second, we simultaneously forbid multiple prefixes. While this effect can be achieved by sequentially forbidding a single prefix, the simultaneous forbidding approach yields a much more compact compilation.

The key to the simultaneous forbidding approach is building a *prefix tree* that contains all (non-dominated) prefixes. A prefix $\tilde{\pi}$ is dominated by prefix $\pi$ if $\pi$ is a prefix of $\tilde{\pi}$; in this case, it is sufficient to forbid $\pi$ and not $\tilde{\pi}$. We construct a tree $T = (N, E)$ where each node corresponds to a prefix, and there is an edge from node $\pi$ to node $\pi'$ if we can add one action to $\pi$ to yield $\pi'$. Given a set of prefixes, this tree can be efficiently constructed by adding the nodes from each prefix iteratively. Given a prefix tree, Definition 4.2 shows how to construct a planning task that forbids exactly these prefixes.

**Definition 4.2.** Let $\Pi = \langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ be a planning task using the SAS+ encoding (Section 2.2), $T = (N, E)$ be a prefix tree with $L \subseteq N$ being the leaf nodes, and $\mathcal{A}(T)$ be the set of discrete actions that appear on the prefixes in $T$. The task $\Pi_T^- = \langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$ is defined as follows:

- $\mathcal{V}' = \mathcal{V} \cup \{\overline{v}\} \cup \{\overline{v}_s \mid s \in N\}$, with $\overline{v}_s$ being binary variables and $\overline{v}$ being a ternary variable,
- $\mathcal{A}' = \mathcal{A}^e \cup \mathcal{A}^1 \cup \mathcal{A}^2 \cup \mathcal{A}^3$, where
  $\mathcal{A}^e = \{a^e \mid a \in \mathcal{A} \setminus \mathcal{A}(T)\}$, $\mathcal{A}^1 = \{a^1 \mid a \in \mathcal{A}\}$, $\mathcal{A}^2 = \{a^2 \mid a \in \mathcal{A}(T)\}$, and

$\mathcal{A}^3 = \{a^3_{(s,t)} \mid (s,t) \in E\}$ with

$$\begin{aligned}
a^e &= \langle pre(a) \cup \{\langle \overline{v}, 1\rangle\}, eff(a) \cup \{\langle \overline{v}, 0\rangle\}\rangle \\
a^1 &= \langle pre(a) \cup \{\langle \overline{v}, 0\rangle\}, eff(a)\rangle \\
a^2 &= \langle pre(a) \cup \{\langle \overline{v}, 1\rangle\} \cup \{\langle \overline{v}_s, 0\rangle \mid (s,t) \in E^a\}, \\
&\qquad\qquad eff(a) \cup \{\langle \overline{v}, 0\rangle\}\rangle \\
a^3_{(s,t)} &= \langle pre(a_{(s,t)}) \cup \{\langle \overline{v}, 1\rangle, \langle \overline{v}_s, 1\rangle\}, \\
&\qquad\qquad eff(a_{(s,t)}) \cup \{\langle \overline{v}_s, 0\rangle, \langle \overline{v}_t, 1\rangle\}\rangle \text{ if } t \notin L, \\
a^3_{(s,t)} &= \langle pre(a_{(s,t)}) \cup \{\langle \overline{v}, 1\rangle, \langle \overline{v}_s, 1\rangle\}, \{\langle \overline{v}, 2\rangle\}\rangle \text{ if } t \in L,
\end{aligned}$$

- $s'_0[v] = s_0[v]$ for all $v \in \mathcal{V}$, $s'_0[\overline{v}] = 1$, $s'_0[\overline{v}_{s_0}] = 1$, and $s'_0[\overline{v}_s] = 0$ for all $s \in N \setminus \{s_0\}$, and
- $g'[v] = g[v]$ for all $v \in \mathcal{V}$ s.t. $g[v]$ defined, and $g'[\overline{v}] = 0$.

The proof of the correctness of the compilation is similar to the proof of Theorem 6 in [Katz et al., 2018a]. The main difference between the two reformulations is in the application of $a^3_{(s,t)}$ for $t \in L$, which leads to a dead-end state.

Finally, we remark that the conflicts we extract can be encoded as PDDL 3 trajectory constraints [Gerevini et al., 2009]. These can be compiled away [Baier and McIlraith, 2006], and the above-mentioned compilation is a special case of such a compilation.

### 4.4.3. Feasibility Checking

If a candidate task plan $\pi = \langle a_1, \ldots, a_K\rangle$ is found to be geometrically feasible, then we have found a solution to our LGP task and can terminate. Otherwise, we can return the full plan $\langle a_1, \ldots, a_K\rangle$ as a conflict. We will refer to doing this as *lazy* conflict extraction.

Alternatively, we can also search for a stronger conflict, in the form of a shorter prefix of $\pi$ that is not geometrically feasible, which we refer to as *eager* conflict extraction. *Eager* conflict extraction searches for the strongest possible conflict we can extract from $\pi$, that is, the shortest prefix $\pi|_k$ which is geometrically infeasible, i.e.

$$\min\ k \text{ s.t. } \text{Feas}(\pi|_k) = 0, \tag{4.2}$$

where $\text{Feas}(\pi)$ is a binary function that returns 1 if Trajectory-NLP($\pi$) is feasible or 0 otherwise. By Theorem 4.1, $\text{Feas}(\pi|_k) \geq \text{Feas}(\pi|_{k+1})$. Therefore, we can find the strongest conflict with a binary search for the length $k$ of this prefix.

Initially, the lower bound $l$ is initialized to 0, and the upper bound $u$ is initialized to $K$. The evaluation of $\text{Feas}(\pi|_m)$, for the midpoint $m = \lfloor \frac{l+u}{2} \rfloor$, corresponds to checking with the motion planner whether the prefix up to $m$ is geometrically feasible.

As geometric feasibility checking is the most expensive computational action we perform, we cache every prefix we check and whether it is feasible or not. This cache is helpful in speeding up feasibility checking, as different discrete plans might still share a common prefix. Additionally, this cache serves as a dataset that captures the history of computational actions performed so far, which will be useful for (a) metareasoning about feasibility checking, and (b) guiding our choice of which plan to check for feasibility next when we use diverse planning. These are described in the subsequent sections. Additionally, we can use the *pose* and *keyframes* bounds (Eqs. (2.15) and (2.16)) to accelerate conflict extraction (see also Section 4.7).

## 4.5. Metareasoning for Conflict Extraction

A middle-ground approach between *lazy* conflict extraction (which does not perform any reasoning to extract conflicts) and *eager* (which attempts to find the minimal conflict) is to use metareasoning. Metareasoning [Russell and Wefald, 1991] can be used to balance the cost (the computational effort spent on extracting a conflict) and the reward (the benefits from having a stronger conflict). As the most expensive operation in our algorithm is geometric feasibility checking, we measure both the reward and the cost in terms of the number of geometric feasibility checks – either required to extract the conflict or saved by having the conflict.

We now describe the metareasoning problem we face in deciding when to stop looking for a conflict, and the overall utility we can expect to obtain. Let $\tau = \langle a_1, \ldots, a_k \rangle$ be some sequence of actions. We will denote by $r(\tau)$ the reward from adding the conflict $\tau$. Of course, this is an unknown quantity, and we will describe ways to estimate it later. Recall that during the binary search for a minimal conflict, we have a discrete plan $\pi$, and a range $[l, u]$ such that $\pi|_u$ is not geometrically feasible, while $\pi|_l$ is. Thus, we can define the metareasoning problem for a given plan $\pi$ of length $|\pi|$ as a Markov Decision Process (MDP) [Bellman, 1957] with states $S_\pi = \{\langle l, u \rangle \mid l \leq u = 0, \ldots, |\pi|\}$ – that is, each state describes the current range of the search.

The terminal states are those where the search has converged, that is, $\{\langle l, l \rangle \mid l = 0, \ldots, |\pi|\}$. The reward in state $\langle l, l \rangle$ is the reward from adding the conflict $\pi|_l$, that is, $r(\pi|_l)$. The reward from all other states is 0. As we are sure to reach a terminal state, there is no need to introduce a discount factor (that is, $\gamma = 1$).

The possible actions at state $\langle l, u \rangle$ are either to stop searching or continue searching. The decision to stop searching yields a deterministic transition to the state $\langle u, u \rangle$ – that is, we terminate and add the conflict $\pi|_u$, obtaining reward $r(\pi|_u)$.

Although the binary search always continues the search by checking the middle node ($\lfloor (l+u)/2 \rfloor$), using the metareasoning MDP allows us to consider any of the nodes between $l$ and $u$ as the next node to check. Thus, we have $u - l + 1$ possible actions – one for each node in the range.

Let us denote the probability of a sequence of actions $\tau$ being geometrically feasible by $p_f(\tau)$. Then by continuing the search to node $m$ (representing $\pi|_m$), we reach the state $\langle m, u \rangle$ with probability $p_f(\pi|_m)$, and state $\langle l, m \rangle$ with probability $1 - p_f(\pi|_m)$.

Due to the structure of this MDP, which lacks any loops, we can compute the optimal values using simple dynamic programming, starting with the terminal states, and computing the optimal value function for states with an increasing gap between the lower and upper bounds – that is, we compute the value for states $\{\langle l, l+1 \rangle \mid l = 1, \ldots, |\pi| - 1\}$, then for $\{\langle l, l+2 \rangle \mid l = 1, \ldots, |\pi| - 2\}$, and so on.

Of course, we still do not know the exact rewards or transition probabilities. In the following, we describe a data-driven method to estimate these, which allows us to compute an optimal policy for an approximate MDP.

**Data-driven estimation** Although computing $r(\tau)$ exactly is not tractable, it should be commensurate with the number of discrete plans that would be pruned by introducing the conflict $\tau$. While we do not know this number, we can estimate the fraction of plans that would be pruned by conflict $\tau$ by the fraction of plans we have discovered that have $\tau$ as a prefix. Thus, we can define the estimator,

$$\hat{r}(\tau) := \frac{|\{\pi' \mid \pi' \in C \text{ and } \tau \text{ is a prefix of } \pi'\}|}{|C|}, \tag{4.3}$$

where $C$ is the set of prefixes in our cache. As the number of matching prefixes in the numerator might be 0 (especially early on in the process), we actually add 1 to both the numerator and the denominator.

The probability of a prefix being feasible or not can also be estimated from the history of previous feasibility checks. Recall that we cache every prefix we check for feasibility. We use this cache to estimate $p_f$. We follow a type system approach [Lelis, 2013] and define a set of simple features for each prefix. Specifically, we use the length of the prefix as its only feature, and keep track of how many feasibility checks were performed for each prefix length, and how many of these turned out to be feasible – the ratio between these is our

**Algorithm 4.1** Pseudocode for *Diverse Planning for LGP.*

```
Input: LGP problem Π_LGP
Parameters: N                          ▷ Number of plans to generate at each iteration.
Π := discrete component of Π_LGP       ▷ Discrete component of an LGP, encoded in SAS+
T := ∅                                 ▷ Set of tried plans
LP := ∅                                ▷ Set of found discrete plans
MC := ∅                                ▷ Set of found conflicts
while not solved do
    Π^f := FORBID(Π, LP ∪ MC)          ▷ Forbid found plans and conflicts (Section 4.4.2)
    LP := LP ∪ Diverse-Plan(Π^f, N)    ▷ Call a diverse planner to find N new plans
    π := SELECT(LP, T)                 ▷ Select a plan to try (Section 4.6)
    feasible, traj := MOTION-Feasible? (Π_LGP, π)   ▷ Check task plan π for geometric feasibility
    if feasible then return π, traj    ▷ Return trajectory and discrete plan
    else
        T := T ∪ {π}
        conflict := FIND-CONFLICT(π)   ▷ Find a prefix of π that is infeasible (Sections 4.4.3 and 4.5)
        MC := MC ∪ {conflict}
    end if
end while
```

estimate of $p_f$, denoted $\hat{p_f}$. Combining $\hat{r}$ and $\hat{p_f}$, we can define our MDP. As our empirical results will show, this approach results in a reduction of the runtime of our solver.

## 4.6. Diversity Criteria and Complete Algorithm

To explore candidate task plans more rapidly, we use a diverse planning approach [Katz et al., 2018b]. The key idea here is to generate multiple plans at each iteration, and then choose one of them for geometric feasibility checking.

Generating a set of plans is done by applying the forbidding compilation (Definition 4.2) iteratively, as done in previous diverse planning approaches [Katz et al., 2018b].

The main question here is how to choose which plan to test next. Our approach is driven by prefixes, so it makes sense to choose a plan that has the longest novel prefix, as even if that plan fails, there is a higher chance that we will extract a short conflict from it. Furthermore, choosing a plan with a novel prefix encourages our approach to explore the

space of discrete plans, thereby covering diverse high-level approaches to the task that imply different nonlinear programs for the continuous trajectory.

Thus, we define the novelty of a plan $\pi$ with respect to a set of plans $LP$ as,

$$np(\pi, LP) := -\min\{k \in \mathbb{N} \mid \forall \pi' \in LP, \pi'|_k \neq \pi|_k\}. \tag{4.4}$$

We then choose to test the plan $\pi$ which maximizes the novelty with respect to the set of plans that were already tested for geometric feasibility, breaking ties randomly. We remark that this notion of novelty is different from previous ones, e.g., [Lipovetzky, 2021, Tuisov and Katz, 2021], and serves as a greedy selection criterion for choosing the next plan.

To summarize, Algorithm 4.1 describes our complete solver in pseudocode, and Fig. 4.3 shows an illustrative example of the execution of our algorithm in a simplified setting. We can now state the theorem proving that our approach is sound and complete.

**Theorem 4.2.** *If the underlying task planner is sound and complete, and the motion planner always finds a feasible trajectory for problems* (2.14) *if such a trajectory exists, then Algorithm 4.1 is sound and complete.*

*Proof.* The proof follows from the fact that we only identify prefixes which cannot appear at the beginning of geometrically feasible plans (Theorem 4.1), and from the correctness of the forbidding compilation (Definition 4.2). □

An important technical point is that some planners perform a relevance analysis and discard actions or state variables which they consider to be useless or redundant. For example, two actions might have the same discrete effects, and thus the planner might decide to keep only one of them. However, these actions might lead to different geometric constraints, and it may be the case that one of them is feasible while the other is not. Thus, such preprocessing techniques must be disabled when solving the discrete planning task.

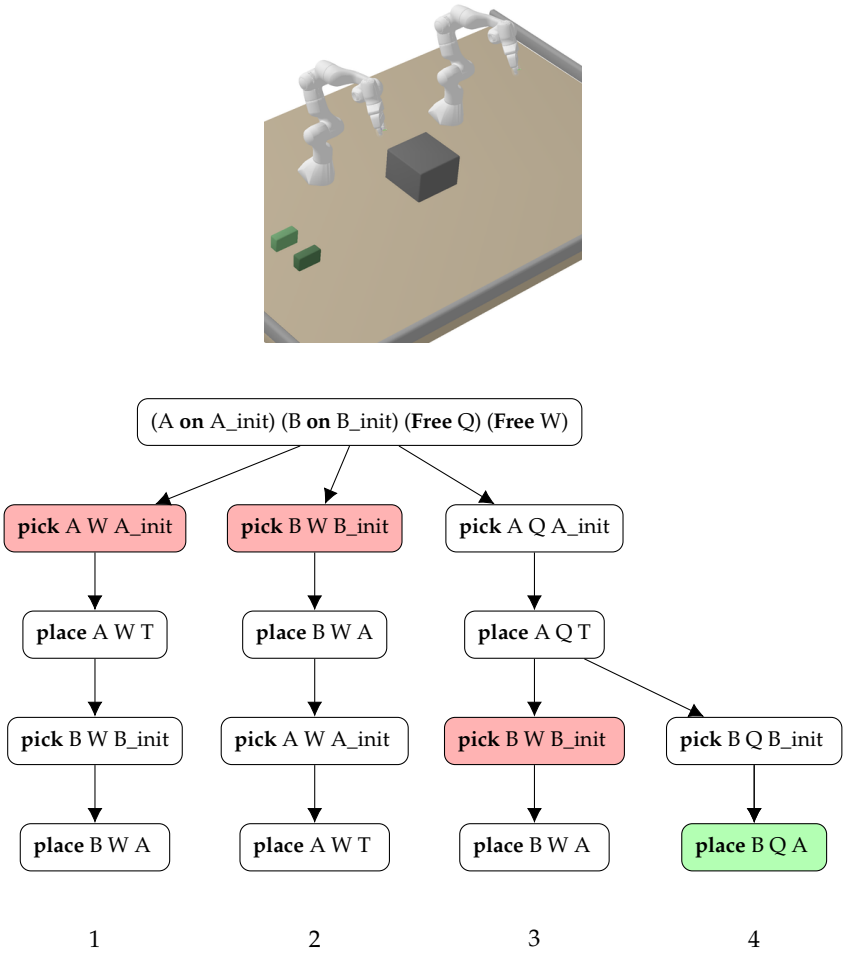



Figure 4.3.: Illustrative example of the execution of our algorithm (with $N = 1$ and *eager* conflict extraction). The scene contains two movable objects, $A$ and $B$, a table, $T$, and two robots, $Q$ and $W$, that can `pick` and `place` the objects. The goal is to stack the blocks on the table: `(A on T)` and `(B on A)`. In each iteration, the task planner has produced a task plan (1, 2, 3, and 4 in this order) that has been tested for feasibility. The motion planner returned the minimal prefix of infeasible discrete actions (highlighted in red), which is used to reformulate the planning task for the subsequent iterations. Plan number 4 (in green) is geometrically feasible.

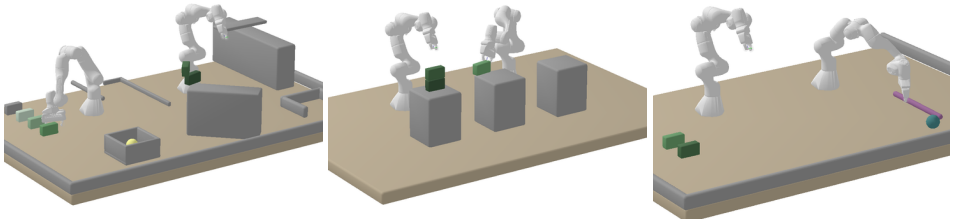


Figure 4.4.: Three domains used to evaluate our algorithm. From left to right: *Blocks*, *Hanoi* and *Push*.

## 4.7. Empirical Evaluation

### 4.7.1. Benchmarks

We use three different domains, all with two 7-DOF robotic arms (Fig. 4.4).

**Blocks** The robots can execute pick and place actions to construct a specified tower of blocks, similarly to the classical Blocksworld domain – except that the planner must also come up with motion plans. Robots can hold a stack of blocks, move the boxes, and place several objects on top of other objects. See Fig. 4.5.

**Hanoi** The robots can execute pick and place actions to solve a Tower of Hanoi problem with objects of equal size and three tables. Only the top object of each tower can be picked, and at most one tower is allowed on each table. From a logical point of view, this is more challenging than *Blocks* and requires longer action sequences, but the instances we use have fewer movable objects in the scene. See Fig. 4.5.

**Push** The robots can pick and place blocks and balls, pick up sticks, and use them as tools to push balls. The goal is to move balls and blocks to a desired discrete state, for example, stacking blocks and placing the ball on top. See Fig. 4.1.

For each domain, we generate different problems (e.g., *Blocks-{0,1,2,3,4,5}*) by modifying the goal and the number of objects, increasing complexity at both the discrete and geometric levels. Our benchmark [2] contains six problems in the domain *Blocks*, three in *Hanoi*, and eleven in *Push*.

### 4.7.2. Baselines

We compare our new approach, *Diverse Planning for LGP*, against three variations of Multi-Bound Tree Search: *MBTS-{0,1,2}* (See Section 2.3.1).

[2] Project website: https://quimortiz.github.io/ConflictPlanningLGP/

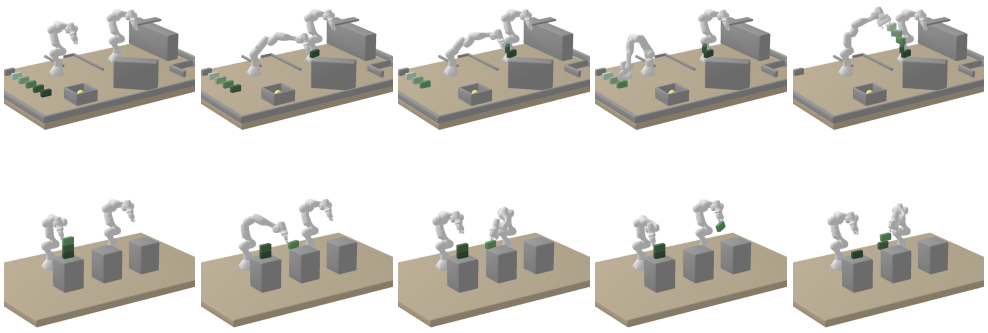


Figure 4.5.: Examples of solutions to the problems in the *Blocks* (top) and *Hanoi* (bottom) domains. A solution in the *Push* domain is shown in Fig. 4.1.

*MBTS-0* does not perform geometric checks on intermediate discrete nodes; that is, it waits until a full candidate task plan is found before solving all bounds and the complete trajectory optimization problem. *MBTS-1* and *MBTS-2* check the *pose* and *keyframes* bounds (Eqs. (2.15) and (2.16)) respectively, before expanding a discrete node in the breadth-first search.

Geometric checks during node expansion in *MBTS-1* and *MBTS-2* prune partial plans that are infeasible. This reduces the branching factor of the search and subsequent node expansions but increases the computational time spent on solving NLPs for action sequences that do not lead to the goal.

### 4.7.3. Results

We use the first iteration of LAMA [Richter and Westphal, 2010] as our underlying task planner. We ran a set of experiments comparing several versions of our approach to the baselines – all experiments were run on an AMD Ryzen 9 5980HS CPU with a 600-second time limit per run. Results are shown in Table 4.1. We omit problems *Hanoi-2* and *Blocks-5*, which were not solved by any algorithm or baseline.

**Comparison to baselines** Hypothesis: *"Our basic novel approach (N=1, Eager conflict extraction) will be faster and solve more problems than any of the MBTS baselines"*. Our method with "$N = 1$, *Eager*" solves more problems (18 vs. 12 out of 20) and is faster (16 vs. 2) than all the baselines *MBTS-{0,1,2}*. In Table 4.1, we only report *MBTS-0*, which shows better performance than the other baselines.

MBTS-0 does not solve problems that require long action sequences or where the branching factor of the tree is very high (for example, the domain *Blocks* contains 12 movable objects). Due to the uninformed behavior of Breadth-First Search, it only finds a few task plans

| | MBTS-0 | | | N=1 *Eager* | | | N=4 *Eager* | | | N=4 *Meta* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | time | pose | key | time | pose | key | time | pose | key | time | pose | key |
| Blocks-0 | **19.4**1.0 | 12.00.0 | 3.00.0 | 43.71.8 | 19.90.9 | 6.50.5 | 41.84.3 | 17.51.9 | 6.50.9 | 41.34.4 | 17.51.9 | 2.80.6 |
| Blocks-1 | - | - | - | 44.81.3 | 18.00.0 | 5.00.0 | **44.0**5.6 | 17.12.3 | 4.80.9 | 46.56.4 | 17.12.3 | 1.70.3 |
| Blocks-2 | - | - | - | 82.610.5 | 17.00.0 | 4.00.0 | **60.4**8.6 | 12.61.1 | 2.40.5 | 70.911.8 | 12.61.1 | 1.40.2 |
| Blocks-3 | - | - | - | 1115.8 | 17.01.0 | 3.40.4 | 10419.7 | 21.72.8 | 5.41.1 | **80.2**12.0 | 20.83.1 | 2.10.4 |
| Blocks-4 | - | - | - | 20033.1 | 27.18.2 | 6.12.8 | 16017.0 | 19.33.1 | 3.31.1 | **139**22.6 | 17.82.7 | 1.30.2 |
| Hanoi-0 | 10.40.4 | 13.00.0 | 4.00.0 | **7.0**0.2 | 7.00.0 | 3.00.0 | 10.01.9 | 8.00.9 | 3.70.8 | 9.11.8 | 8.61.1 | 2.90.5 |
| Hanoi-1 | 34.70.7 | 34.00.0 | 6.00.0 | 27.00.6 | 17.00.0 | 8.00.0 | 18.73.1 | 13.51.0 | 5.10.6 | **13.8**2.2 | 14.01.0 | 3.40.4 |
| Push-1 | 41.90.8 | 55.80.2 | 1.00.0 | **17.1**0.4 | 14.00.0 | 4.00.0 | 24.41.4 | 17.31.1 | 5.30.5 | 24.91.7 | 18.71.2 | 3.80.4 |
| Push-2 | 50.01.0 | 64.00.0 | 1.00.0 | 49.50.9 | 37.00.0 | 13.20.1 | 37.11.1 | 23.20.9 | 7.20.4 | **34.3**1.6 | 24.30.8 | 3.20.2 |
| Push-3 | 27.70.9 | 38.00.0 | 1.00.0 | **14.4**0.2 | 11.00.0 | 3.00.0 | 26.13.2 | 17.92.0 | 5.80.9 | 21.11.8 | 17.31.8 | 2.90.3 |
| Push-4 | 75.91.5 | 1040.0 | 2.00.0 | 71.37.8 | 41.22.8 | 15.21.5 | 32.64.1 | 20.32.2 | 5.91.0 | **30.6**2.7 | 21.32.4 | 3.10.4 |
| Push-5 | 1111.6 | 1440.1 | 1.00.0 | **20.4**0.3 | 17.00.0 | 5.00.0 | 30.82.5 | 23.72.2 | 7.40.9 | 29.42.4 | 24.42.3 | 3.20.4 |
| Push-6 | 1171.5 | 1420.0 | 1.00.0 | 64.51.2 | 50.00.0 | 17.10.1 | 45.41.0 | 29.20.7 | 9.20.4 | **45.1**1.2 | 31.81.2 | 4.60.3 |
| Push-7 | - | - | - | **68.6**4.6 | 51.33.5 | 19.01.6 | 79.15.4 | 52.64.0 | 18.01.6 | 70.42.6 | 53.02.7 | 7.40.5 |
| Push-8 | 78.31.2 | 92.00.0 | 1.00.0 | **17.0**0.4 | 13.00.0 | 3.00.0 | 32.43.6 | 26.03.0 | 7.81.1 | 32.83.7 | 28.23.6 | 4.10.6 |
| Push-9 | 24840.1 | 42367.2 | 2.50.5 | 63.31.3 | 45.00.0 | 16.00.0 | **46.2**6.2 | 32.53.7 | 10.41.4 | 49.811.9 | 39.79.5 | 5.31.7 |
| Push-10 | **12.7**0.5 | 16.00.0 | 1.00.0 | 12.80.5 | 9.00.0 | 3.00.0 | 13.81.6 | 10.41.3 | 3.30.6 | 13.21.4 | 10.51.3 | 1.60.2 |
| Push-11 | - | - | - | 61.19.3 | 25.52.3 | 10.71.4 | **26.0**2.0 | 13.40.5 | 3.80.4 | 30.55.4 | 16.72.5 | 2.60.6 |
| Total | 827 | 1138 | 24.5 | 976 | 437 | 145 | 833 | 376 | 115 | 783 | 394 | 57.4 |

Table 4.1.: Summary of the experimental results. We report the computational time in seconds (*time*), and the number of calls to the motion planner for checking the pose bound (*pose*) and the keyframes bound (*key*), with the mean over 10 randomized runs in black and the standard deviation of the mean estimator in grey. "Total" is the sum of the columns (note that the sum for *MBTS-0* is over fewer problems). A dash "–" denotes that the problem was not solved in all 10 runs.

(sometimes none), none of which are geometrically feasible. Instead, our method leverages state-of-the-art task planning to compute action sequences efficiently even in large discrete spaces, and geometric information is encoded incrementally in the planning task through our prefix forbidding reformulation.

**Analysis of diverse planning** Hypothesis: *"Diverse planning with a novelty measure will improve over incremental plan generation"*. We compare "*N=1, Eager*" (the planner produces a single plan, which is evaluated by the motion planner) and "*N=4, Eager*" (the planner produces four plans in each iteration, which are stored in a buffer; the motion planner evaluates the plan in the buffer that maximizes our novelty criteria).

$N = 4$ reduces both the overall computational time and the number of tested plans. Choosing a plan from a set of candidates with our criteria is beneficial, as it enforces novelty-based exploration in the space of candidate discrete plans. The role of prefixes and orderings in an LGP is captured accurately by our novelty measure, which outperforms alternative plan similarity metrics like action set similarity, which is inaccurate in the context of LGP, where action ordering and precedence cannot be neglected.

**Analysis of conflict extraction** Hypothesis: *"Metareasoning is faster than Eager and Lazy conflict extraction"*. For *N=4*, we compare three different methods for extracting prefix conflicts: *Eager* (finds the minimal prefix using the *keyframes* bound ((2.15))), *Lazy-pose* (an enhancement of *Lazy* that checks only the *pose* bound (2.16) to try to extract a conflict), and *Meta* (a metareasoning approach for conflict extraction).

Our metareasoning approach delivers a speedup across problems (*Meta* is better in 12 vs. *Eager* 6). The *Lazy-pose* sometimes provides small infeasible prefixes with the *pose* bound Eq. (2.16), but is slower than *Meta* and *Eager*. Finally, note that relaxation bounds of feasible NLPs are very fast to compute. This explains why, in some problems, *Eager* is faster than *Meta*, even if it performs more geometric checks in total.

## 4.8. Limitations

*Diverse Planning for LGP* shares the main limitations of the underlying LGP formulation and the previous MBTS solver, namely, the local convergence of nonlinear optimization methods. Optimization methods converge only to local optima, which might prevent finding a solution even if a problem is feasible. One way to mitigate convergence to bad local optima is to use random restarts, as solving the same problem with different initializations can improve the success rate.

To integrate random restarts into our conflict-based formulation, we can use a soft-conflict formulation. Instead of blocking prefixes in the task planner, we can penalize task plans that contain plan prefixes where the optimizer failed to find a solution. Another possible practical implementation is to use a probabilistic hard-conflict formulation, where, in each call to the task planner, we block a prefix with a probability proportional to the number of times the optimizer failed to solve the corresponding optimization problem.

Further limitations for deploying the algorithms in the real world, as often encountered in the TAMP literature, are the assumptions of accurate world information (i.e., geometry and position of the objects), a perfect forward model used for planning, and the simple geometric shapes of the objects.

## 4.9. Conclusions

In this chapter, we propose the first systematic interface between state-of-the-art task planners and nonlinear constrained path optimization methods to solve Logic Geometric Programs. A key idea of our approach is to efficiently identify geometric conflicts in the form of minimal infeasible action prefixes and incorporate this information back into the task planner through a multi-prefix forbidding compilation. Based on this general interface, we further develop a metareasoning strategy to minimize the number of calls to the motion planner and a new novelty criterion for selecting plans from a set of candidates. Our approach systematically outperforms the baseline LGP solver, solving more problems and faster, especially when the solution requires long action sequences.

This work lays the foundations for the more efficient Factored-NLP Planner, presented in Chapter 5, which also combines a discrete planner with trajectory optimization through an interface based on detecting and encoding geometric conflicts. However, instead of relying on infeasible prefixes, the Factored-NLP Planner uses a more powerful interface based on detecting and blocking subsets of infeasible nonlinear constraints in the optimization problems, which results in an order-of-magnitude improvement with respect to the Multi-Bound Tree Search.

Our results suggest that incorporating a PDDL planner into Task and Motion Planning (TAMP) solvers is crucial for enhanced performance and scalability. Besides the Factored-NLP Planner in Chapter 5, our TAMP meta-solver in Chapter 5 also employs a PDDL solver to compute a lower bound on the number of discrete actions required to reach the goal.

# Chapter 5

# Conflict-Based Search in Factored Logic Geometric Programs

## 5.1. Introduction

Despite recent advances in Task and Motion Planning (TAMP) solvers, current algorithms struggle with high-dimensional configuration spaces (e.g., multiple robots), long-horizon planning, and constrained environments that require joint optimization. A promising approach to planning in such challenging settings is to efficiently interface state-of-the-art solvers on both sides, particularly incorporating information about infeasibility from continuous solvers back to the task level.

In Chapter 4, we illustrate how geometric conflicts in the form of task plan prefixes could be encoded back into a discrete planner. Building on this foundation, we now aim to find smaller conflicts to create a more efficient interface between the task level and the motion level. To this end, in this chapter[1], we present a second iterative, conflict-based TAMP solver that combines discrete planning with nonlinear optimization with a novel bidirectional interface.

Our approach is based on identifying minimal subsets of nonlinear constraints that guarantee the infeasibility of the continuous trajectory optimization problems. This information is encoded back into the high-level task planner with a special blocking reformulation. This new interface provides a powerful enhancement, as now one conflict can directly block multiple different candidate plans–specifically, those that would generate a trajectory optimization problem containing the infeasible constraints. In contrast, our previous

[1]This chapter is based on the publication: Ortiz-Haro, J., Karpas, E., Katz, M., and Toussaint, M. (2022). A Conflict-Driven Interface Between Symbolic Planning and Nonlinear Constraint Solving. IEEE Robotics and Automation Letters, 7(4), (pp. 10518-10525).

prefix-forbidding solver, Diverse Planning for LGP, only blocked plans with a matching task plan prefix (Chapter 4). Given that the number of candidate high-level task plans grows exponentially with the number of robots and objects, an efficient interface is vital for success and scalability, as demonstrated in our evaluation.

The design of our new solver, called the Factored-NLP Planner, has required the introduction of several innovative techniques and contributions: an efficient conflict detection algorithm, a conflict-blocking reformulation of the discrete planning problem, and a precise formulation of the TAMP problem.

As a foundation for this algorithm, we first introduce an abstract Planning with Nonlinear Transition Constraints (PNTC) formulation, where a discrete task plan implies a factored nonlinear program as a sub-problem, and logical predicates of the task plan can be related to factors of the NLP. This formulation clarifies the concepts and exact assumptions our algorithm builds on and formally defines the explicit bidirectional relation between the discrete and the continuous components of the problem, which is exploited in our solver.

The PNTC formulation ensures both the correct relationship between the discrete and continuous levels, and the appropriate structure in the trajectory optimization problem. In the context of TAMP, it can be viewed as a factored variant of a Logic Geometric Program (LGP) (Section 2.3). However, in comparison to LGP, PNTC explicitly defines a factored structure of the implied NLP and a bidirectional mapping between symbols and constraint factors in the resulting NLP. This is exactly the structure we need to better inform the discrete search and is naturally available in TAMP, making our solver directly applicable to solve TAMP and LGP problems.

We evaluate our method on three robotic TAMP scenarios that present complex intrinsic logic-geometric dependencies requiring long action sequences. Generating solutions within seconds, our approach clearly outperforms previous optimization-based solvers for TAMP. We further validate the framework through real-world experiments, demonstrating that the solver computes full task and motion plans in a few seconds.

## 5.2. Related Work

A comprehensive review of related work on Task and Motion Planning is provided in Section 2.4. We now briefly position our work with respect to the most closely related approaches.

From the perspective of classical planning, the most closely related work includes [Haslum et al., 2018], which extends classical planning with general state constraints, and [Fernández-González et al., 2018], which combines discrete search with convex optimization. In contrast, the constraints of PNTC are nonlinear, defined by a sequence of discrete states and evaluated

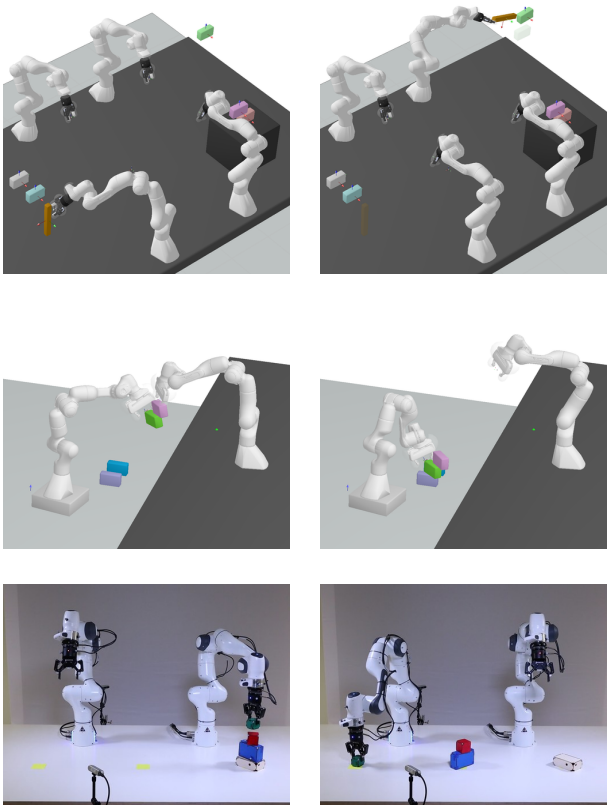


Figure 5.1.: Task and Motion Planning problems solved by our framework. *Top row*: Four robot manipulators use a stick as a tool to reach a distant block. *Middle row*: A heterogeneous team of robots builds a tower. *Bottom row*: Two real 7-DOF manipulators solve the Tower of Hanoi puzzle.

on consecutive continuous variables. This implies a nonlinear program for the whole sequence of continuous variables, which can model the complex continuous constraints in the TAMP problem efficiently.

In comparison to TAMP solvers, our method is related to conflict-based solvers, such as [Srivastava et al., 2014, Dantam et al., 2016]. These methods either use a set of predefined predicates such "is reachable" [Srivastava et al., 2014] to incorporate information about geometry, or block full plans or pairs of state-actions [Dantam et al., 2016]. Alternatively, our framework can encode any type of continuous infeasibilities that potentially involve several motion phases. In fact, instead of enumerating possible geometric failure cases, we

define nonlinear constraints to model the motion and geometry and let the solver detect which intrinsic subset is jointly infeasible.

Compared to optimization-based solvers such as the Multi-Bound tree search [Toussaint and Lopes, 2017] and *Diverse planning for LGP* (Chapter 4), we design a much more efficient interface between task planning and motion planning, as demonstrated in our experiments.

## 5.3. Problem Formulation

In this section, we introduce *Planning with Nonlinear Transition Constraints* (PNTC). PNTC is similar to the Logic Geometric Program formulation for TAMP; the main difference lies in that it provides an explicit modeling of the factorization at the discrete level and within the trajectory optimization problem. For a comprehensive introduction to the Logic Geometric Program formulation, we direct the reader to Section 2.3.

Chapter 3 presents an intuitive and insightful explanation of the natural factorization of trajectory optimization that occurs in Task and Motion Planning. PNTC will now formalize the required interface between logic and geometry to generate such structured representations. This will expose three key properties–time structure, local composition, and sparse factorization–that will be used in our solver.

**Planning with Nonlinear Transition Constraints** A *Planning with Nonlinear Transition Constraints (PNTC)* problem is a 7-tuple $\langle \mathcal{V}, \mathcal{A}, s_0, g, \Pi, \mathcal{H}, X_v \rangle$ that includes a discrete component $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ and a continuous component $\langle \mathcal{H}, X_v \rangle$, coupled through an interface $\Pi$.

- *Discrete Component*: The discrete component $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ corresponds to a classical planning problem encoded in SAS+ (Section 2.3).
- *Continuous Component*: $X_v$ is a finite set of $n$ continuous variables $\{x^1, \dots, x^n\}$. Each variable takes a value in a continuous space $\text{dom}(x^i) = \mathcal{X}^i$ (e.g., $\mathcal{X}^i = \mathbb{R}^{n_i}$). A continuous state $x \in \mathcal{X}^1 \times \dots \times \mathcal{X}^n = \mathcal{X}$ is a value assignment to all variables. In the planning problem, we use the notation $x_k$ to denote the state at step $k$, and $x_k^i$ to denote the variable $i$ at step $k$. $\mathcal{H}$ is a finite set of nonlinear, piece-wise differentiable constraint functions that are evaluated on pairs of subsets of continuous variables, $\mathcal{H} = \{\phi_b : \mathcal{X}^{b_0} \times \mathcal{X}^{b_1} \to \mathbb{R}^{n_b}\}$. The index sets $b_0, b_1 \subseteq \{1, \dots, n\}$ indicate on which subsets of variables the function $\phi_b$ depends. These functions define nonlinear constraints $\phi_b(x^{b_0}, \tilde{x}^{b_1}) \{\leq, =\} 0$ on a pair of subsets of continuous variables $(x^{b_0}, \tilde{x}^{b_1})$ (e.g., $x^{b_0} = \{x^1, x^2\}, \tilde{x}^{b_1} = \{\tilde{x}^3\}$).
- *Interface*: The discrete and continuous components of a *PNTC* are coupled through the mapping $\Pi$. Let $\mathcal{P}$ be the set of all possible partial discrete states. The mapping

$\Pi : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{H} \cup \emptyset$ with $\langle p, \tilde{p} \rangle \mapsto \phi_b(x^{b_0}, \tilde{x}^{b_1})$, maps a pair of discrete partial states $\langle p, \tilde{p} \rangle$ to a nonlinear constraint function $\phi_b$ that is evaluated on subsets of continuous variables $x^{b_0}, \tilde{x}^{b_1}$. The empty set $\emptyset$ indicates that some pairs $\langle p, \tilde{p} \rangle$ do not generate constraints. This formulation also includes constraints acting on a single state $\Pi(p) \rightarrow \phi_b(x^{b_0})$.

A solution to a PNTC is a sequence of discrete and continuous states $\langle (s_0, x_0), \ldots, (s_K, x_K) \rangle$ and discrete actions $\langle a_1, \ldots, a_K \rangle$ (starting from the fixed $s_0$ and $x_0$), such that,

$$s_k \in \mathcal{S}, \qquad k = 0, \ldots, K \tag{5.1a}$$
$$a_k \in \mathcal{A}(s_{k-1}), \qquad k = 1, \ldots, K \tag{5.1b}$$
$$s_k = \text{succ}(s_{k-1}, a_k), \qquad k = 1, \ldots, K \tag{5.1c}$$
$$g \subseteq s_K, \tag{5.1d}$$
$$x_k^i \in \mathcal{X}^i, \qquad k = 0, \ldots, K,\ i = 1, \ldots, n \tag{5.1e}$$
$$\phi_b(x_k^{b_0}, x_{k+1}^{b_1}) \{\leq, =\} 0, \qquad \phi_b \equiv \Pi(p, \tilde{p}), \forall p \subseteq s_k, \forall \tilde{p} \subseteq s_{k+1}, k = 0, \ldots, K-1. \tag{5.1f}$$

Given a fixed task plan $\langle a_1, \ldots, a_K \rangle$, the sequence of discrete states is $\langle s_0, \ldots, s_K \rangle$. The continuous states can be computed by solving the continuous feasibility program,

$$\text{find } x_k^i \in \mathcal{X}^i, \qquad k = 0, \ldots, K,\ i = 1, \ldots, n \tag{5.2a}$$
$$\text{s.t. } \phi_b(x_k^{b_0}, x_{k+1}^{b_1}) \{\leq, =\} 0, \qquad \phi_b \equiv \Pi(p, \tilde{p}), \forall p \subseteq s_k, \forall \tilde{p} \subseteq s_{k+1},\ k = 0, \ldots, K-1. \tag{5.2b}$$

Therefore, a valid task plan is only a necessary condition for the existence of a full discrete and continuous solution and, in practice, valid discrete plans often fail at the continuous level.

**PNTC and LGP for Task and Motion Planning** When comparing PNTC and LGP, we observe that PNTC is a factored formulation. The discrete state space is now factored: instead of an unstructured discrete state space $\mathcal{S}$ as in LGP (2.13), we now have a set of discrete variables $\mathcal{V}$. The continuous space in PNTC is also factorized, resulting in a Factored-NLP formulation of the trajectory optimization problems to compute the motion of robots and objects. As highlighted in Sections 2.2 and 4.3, this factorization is naturally available in TAMP problems that involve multiple objects and robots.

A technical difference with respect to LGP is that here we introduce special variables to represent trajectories between keyframes within the continuous state. Thus, a continuous state in PNTC includes both the keyframe configuration (i.e., the configuration exactly at the phase of the transition) and the trajectory from the last keyframe, while LGP uses the original configuration space as the continuous space (e.g., the joint values of the robot or the object pose for a single configuration). This modification allows us to define pairs of

discrete and continuous states and generates a beneficial structure in the Factored-NLP for conflict-based planning, which can be shown to be equivalent to the LGP formulation. The Factored-NLP in PNTC can also be viewed as a combination of the full trajectory optimization problem and the keyframes bound of LGP within a single more structured optimization problem.

Notably, PNTC decomposes the nonlinear constraints that appear in the LGP problem into a set of small constraints and introduces an explicit mapping $\Pi$ that defines which parts of the discrete plan generate which constraints.

Constraints of the form $h_{\text{path}}(x, s)$, $h_{\text{switch}}(x; s, s')$, and $\tilde{h}_{\text{switch}}(x, x'; s, s')$ in LGP (Eqs. (2.13) and (2.15)) are decomposed into a set of smaller constraints $\{\phi_b \mid \phi_b \equiv \Pi(p, \tilde{p}),\ \forall p \subseteq s, \tilde{p} \subseteq s'\}$ in PNTC (5.2).

## 5.4. Factored-NLP: a Bidirectional Interface Between Task and Motion

Given a fixed sequence of discrete states $\langle s_0, \dots, s_K \rangle$, we represent the optimization problem over the sequence of continuous variables (5.2) as a Factored-NLP, denoted by $G(\langle s_0, \dots, s_K \rangle)$.

Chapter 3 provides the basic definitions of Factored-NLPs, a detailed example for a Pick and Place task plan, and a discussion on scalability, generalization, and properties of this representation. All Factored-NLPs previously shown in Chapter 3 have been generated using the PNTC formulation that we have formally defined here.

The set of variables and constraints of the Factored-NLP $G(\langle s_0, \dots, s_K \rangle)$ is:

$$X_G = \{x_k^i \mid\ k = 0, \dots, K,\ i = 1, \dots, n\}, \tag{5.3a}$$

$$\Phi_G = \{\phi_b \mid \phi_b \equiv \Pi(p, \tilde{p}),\ \forall p \subseteq s_k, \tilde{p} \subseteq s_{k+1},\ k = 0, \dots, K-1\}. \tag{5.3b}$$

The factored structure of this Factored-NLP stems from the object and time factorization of the variable set $X_G$ and the structured dependency between variables and constraints.

Factored-NLPs from the PNTC formulation have the following properties:

**Property 5.1.** (Local time connectivity) A variable vertex $x_k^i$ is connected to constraints that are evaluated on variables with time index $k$, $k-1$, or $k+1$.

**Property 5.2.** (Factor time invariance) A sequence of partial states $\langle p_0, \dots, p_L \rangle$ induces a subgraph $M(\langle p_0, \dots, p_L \rangle) = (X_M \cup \Phi_M, E_M)$ with:

$$\Phi_M = \{\phi_b \mid \phi_b \equiv \Pi(p, \tilde{p}),\ \forall p \subseteq p_l, \tilde{p} \subseteq p_{l+1},\ l = 0, \dots, L-1\}, \tag{5.4a}$$

$$X_M = \{x_l^i \mid \exists \phi_b \in \Phi_M \text{ such that } \phi_b \text{ depends on } x_l^i,\ l = 0, \dots, L,\ i = 1, \dots, I\}. \tag{5.4b}$$

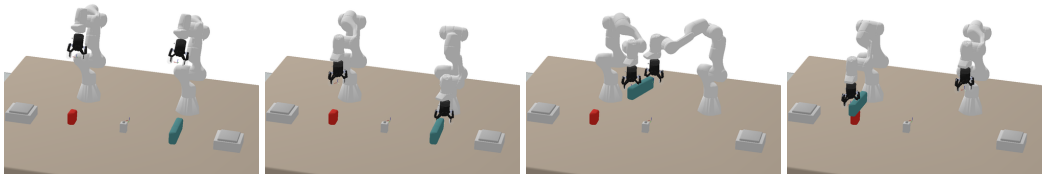


Figure 5.2.: Example domain with two objects and two robots.

Because the mapping from partial states to nonlinear constraints does not depend explicitly on the time index, if a sequence of states $\langle s_0, \ldots, s_K \rangle$ contains a sequence of partial states $\langle p_0, \ldots, p_L \rangle$ starting at any time index, that is,

$$\exists k \in \{0, \ldots, K - L\} \text{ such that } p_l \subseteq s_{k+l},\ l = 0, \ldots, L, \tag{5.5}$$

then the Factored-NLP for the sequence of partial states $M(\langle p_0, \ldots, p_L \rangle)$ is a subgraph of the Factored-NLP of the full sequence of states $G(\langle s_0, \ldots, s_K \rangle)$,

$$M(\langle p_0, \ldots, p_L \rangle) \subseteq G(\langle s_0, \ldots, s_K \rangle). \tag{5.6}$$

**Definition 5.1.** An infeasible subgraph of a Factored-NLP (i.e., a subset of variables and constraints) is minimal if, when removing any variables or constraints, the resulting optimization problem is feasible.

**Property 5.3.** The minimal infeasible subgraph is connected. If the Factored-NLP $G$ is not connected, the NLP associated with each connected component $G_j$ can be solved independently, and $\text{Feas}(G) = \bigwedge_j \text{Feas}(G_j)$.

Property 5.1 and Property 5.3 are used later for detecting minimal infeasible sets of constraints efficiently. Property 5.2 is essential in our conflict-based algorithm, as it ensures the correctness of the task reformulation, by allowing us to prune multiple candidate task plans with a single conflict.

**Example domain** In this section, we revisit the example domain shown in Section 3.3, but now provide a more formal and detailed description of the task plan, the constraints that appear in the Factored-NLP, and the relationships between them, as defined by the PNTC formulation.

Thus, we consider again a domain with a *Table* and two movable objects, $A$ and $B$, initially on *A_init* and *B_init*, and two robot manipulators, $Q$ and $W$, with the goal of stacking $A$ on top of $B$.

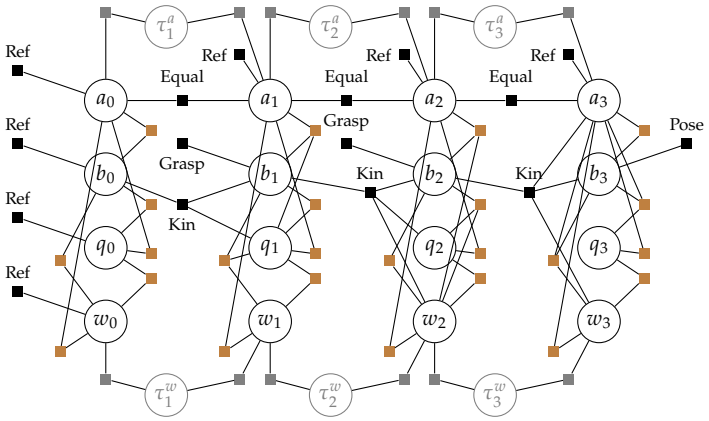



Figure 5.3.: Factored-NLP of the example domain in Fig. 5.2. Circles represent variables and squares represent constraints. We display the variables for all the keyframe configurations $(a, b, q, w)$, and the trajectories $(\tau^a, \tau^w)$ (omitting $\tau^b, \tau^q$ and factors that represent collisions between trajectories to keep the illustration cleaner). Brown squares are collision avoidance constraints. Gray squares are boundary constraints between trajectories and keyframes.

The discrete state space is factorized into four variables: `parent_A`, `parent_B`, `robot_Q`, `robot_W`. The set of possible actions is defined by two action operators: `pick` and `place`. The initial discrete state is $s_0$ = [`parent_A = A_init`, `parent_B = B_init`, `robot_Q = free`, `robot_W = free`] (see Chapter 4). A Factored-NLP is defined by a sequence of discrete states. In this example, we choose the sequence of actions,

- $a_1$: `Pick B with Q from B_init`,
- $a_2$: `Pick B with W from Q`,
- $a_3$: `Place B with W on A`.

Applying these actions from the initial discrete state $s_0$ results in the state sequence,

- $s_0$: [`parent_A = A_init`, `parent_B = B_init`, `robot_Q = free`, `robot_W = free`],
- $s_1$: [`parent_A = A_init`, `parent_B = Q`, `robot_Q = full`, `robot_W = free`],
- $s_2$: [`parent_A = A_init`, `parent_B = W`, `robot_Q = free`, `robot_W = full`],
- $s_3$: [`parent_A = A_init`, `parent_B = A`, `robot_Q = free`, `robot_W = free`].

The Factored-NLP is shown again in Fig. 5.3. We refer to Sections 3.2 and 3.3 for an explanation of variables and constraints, and we discuss here which sequences of partial discrete states imply which constraints. Constraints operate on pairs of consecutive contin-

uous variables, and the constraints that are applied depend on the values of the discrete variables.

First, note that the variables for the continuous initial state $x_0$ (i.e., $a_0, b_0, q_0, w_0$) are also added in the Factored-NLP, together with constraints *Ref* that fix their value. The mapping $\Pi : (p, \tilde{p}) \mapsto \phi$ in PNTC can be implemented as a set of rules that, given a fixed task plan, analyzes all pairs of partial states and generates the constraints in the Factored-NLP. For instance:

- `Parent_B = B_init and Parent_B' = Q` $\rightarrow Kin(b, b', q')$ This transition occurs in $s_0 \rightarrow s_1$ and generates the constraint *Kin* between variables $b_0, b_1, q_1$ in the Factored-NLP.
- `Parent_A = A_init` $\rightarrow Ref(a)$ In all discrete states $s_0, s_1, s_2, s_3$, the variable `Parent_A` has the value `A_init`; the constraint *Ref* is applied to variable $a$ in all time steps: $a_0, a_1, a_2$ and $a_3$.
- `Parent_A = A_init and Parent_A' = A_init` $\rightarrow Equal(a, a')$. In this example, object A is always in the start position, and thus we add the constraints $Equal(a_0, a_1)$, $Equal(a_1, a_2)$, and $Equal(a_2, a_3)$. Note that in this case, such constraints are redundant with $Ref(a_1)$, $Ref(a_2)$, $Ref(a_3)$. However, *Equal* constraints are necessary, e.g., when placing an object on the table, to ensure it remains still, or when holding an object for multiple time steps.
- `Parent_B = Q` $\rightarrow Grasp(b)$. In $s_1$, `Parent_B = Q`. Therefore, we add the constraint $Grasp(b_1)$.
- Collision constraints (brown squares) are added at all time steps. They account for the different structures in the kinematic chain (e.g., whether the object is held by the robot or is on the table) – resulting in a slight variation of dependencies in each vertical slice of the grasp.
- Boundary value constraints (gray squares) tie the keyframes and the trajectories at all time steps. For instance, $\tau_1^q$ is constrained to start at $q_0$, and end at $q_1$.

## 5.5. Overview: Factored-NLP Planner

Fig. 5.4 provides an overview of the Factored-NLP Planner for solving a PNTC, which we will introduce in the subsequent sections. To simplify the presentation, we briefly outline the steps of the algorithm, which are run iteratively:

1. We leverage a state-of-the-art discrete PDDL planner to find a sequence of discrete states that are valid for the current discrete planning task.

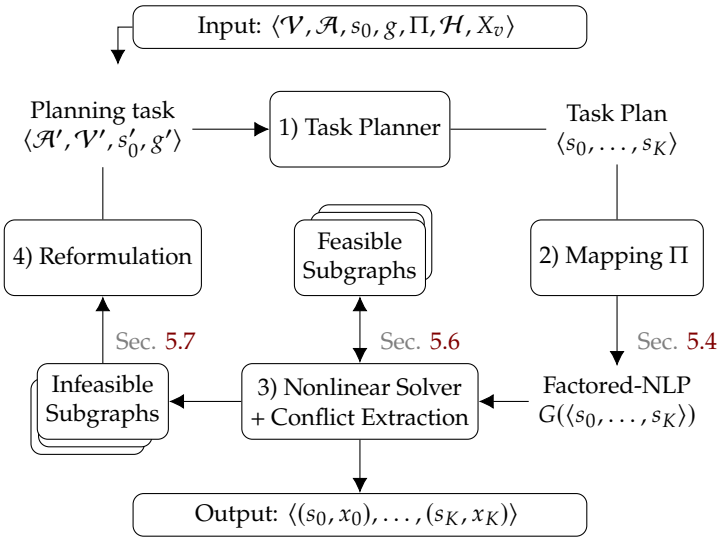



Figure 5.4.: Overview of the Factored-NLP Planner for solving a PNTC problem $\langle \mathcal{V}, \mathcal{A}, s_0, g, \Pi, \mathcal{H}, X_v \rangle$. The solution is a sequence of discrete and continuous states $\langle (s_0, x_0), \ldots, (s_K, x_K) \rangle$.

2. We generate the Factored-NLP that represents the continuous optimization problem for the continuous variables and the nonlinear constraints associated with the chosen candidate task plan.
3. An NLP solver attempts to solve the Factored-NLP. If this NLP is feasible, the algorithm terminates, and the output is a solution containing a sequence of discrete and continuous states. Otherwise, a minimal conflict in the form of a minimal infeasible subgraph (i.e., a subset of the Factored-NLP) is extracted, and all evaluated subgraphs are stored in a database as either feasible or infeasible subgraphs.
4. Finally, we reformulate the planning task to forbid all plans that would generate a Factored-NLP containing any subgraph previously determined to be infeasible.

## 5.6. Finding Small Infeasible Subgraphs

In this section, we discuss how to detect a minimal subset of infeasible constraints from a Factored-NLP (Step 3 of the Factored-NLP Planner, Fig. 5.4). In the worst case, finding an infeasible subgraph of *minimum cardinality* requires solving an NLP for each subset of constraints, $O(2^{|\Phi_G|})$ [Shoukry et al., 2018]. Conversely, a *minimal* infeasible subgraph can be found by solving a linear number of problems [Amaldi et al., 1999]. This search can

be accelerated with a divide-and-conquer strategy, with complexity $O(\log |\Phi_G|)$ [Junker, 2004]. Recently, [Shoukry et al., 2018] presented a technique for finding an approximately minimal subgraph in a convex optimization problem by solving one convex program with slack variables.

Inspired by these works, we propose an algorithm for finding small minimal infeasible subgraphs that exploits the particular structure of the Factored-NLP in our setting, namely the time structure and the semantic information contained within them, as well as the convergence point of the nonlinear optimizer.

**Double binary search on the time index** The first key insight is to exploit the time connectivity of our Factored-NLP (Property 5.1). Given an infeasible Factored-NLP $G(\langle s_0, \dots, s_K \rangle)$, we can find a minimal temporal sequence $\langle s_f, \dots, s_l \rangle$, $0 \leq f \leq l \leq K$ such that $G(\langle s_f, \dots, s_l \rangle)$ is infeasible with a double binary search that executes $O(\log K)$ calls to a nonlinear optimizer. Specifically, we first compute the minimum upper index $l$ such that $G(\langle s_0, \dots, s_l \rangle)$ is infeasible. After fixing $l$, we compute the maximum lower index $f$ such that $G(\langle s_f, \dots, s_l \rangle)$ is infeasible.

**Relaxations** Binary search on time exploits the local connectivity in the temporal dimension but does not detect the infeasible factors within an infeasible temporal sequence. To address this issue, we propose solving a set of relaxations of the Factored-NLP that evaluate only a subset of variables and constraints. Each relaxation corresponds to a subgraph of the Factored-NLP and is, therefore, a necessary condition for feasibility. The algorithm stores the infeasible relaxations as candidates for the minimal subgraph.

The relaxations depend on the semantic information of the variables and constraints and are problem-independent but domain-specific. Intuitively, we are looking for relaxations that make the graph sparser, smaller, and potentially disconnected, while keeping those constraints that define the infeasible subgraph. Section 5.8.1 presents informative relaxations in the context of Task and Motion Planning.

**Leveraging the convergence point of the optimizer** A powerful heuristic to discover a smaller infeasible subset of variables and constraints is to check the convergence point of the optimizer in an infeasible graph.

Typical optimization methods also converge for infeasible Factored-NLP $G$, and we can use the convergence point as a heuristic guess to find a subgraph of $G$ that is infeasible. Specifically, we test the subgraph spanned by the constraints violated at the convergence point, i.e., $M' = (X' \cup \Phi', E')$ where $\Phi' \subseteq \Phi_G$ is the set of constraints not fulfilled, and $X' = \{x_i^k \in X_G \mid \exists \phi_b \in \text{Neigh}(x_i^k) \text{ s.t. } \phi_b \in \Phi'\}$. If $M'$ is also infeasible, we consider only $M'$ as a candidate for the minimal infeasible subgraph.

**The complete algorithm** We combine these three ideas into one algorithm to find an infeasible subgraph. In this algorithm, we alternate between applying relaxations (each relaxation considers only a subset of variables and constraints) that potentially break the full problem into disconnected components, and computing the minimal infeasible time slice inside each connected component (with a double binary search). The convergence point of the optimizer is used to reduce the size of the output infeasible subgraph. The algorithm will return the first infeasible subgraph it finds, and therefore it is best to try the relaxations in a *loose* to *tight* order, as this will likely result in a smaller infeasible subgraph (see Alg. 1 of Appendix A in our paper for the implementation details).

Deciding whether a relaxation should be applied before or after the binary search on the time index is rather arbitrary. To this end, a relevant observation is that solving a small NLP that is feasible is usually an order of magnitude faster than checking that a larger NLP is infeasible. Thus, we try to solve numerous small and feasible problems first.

**Database of feasible subgraphs** The graph structure of the Factored-NLP is a suitable representation to share information about feasibility between different sequences of discrete states. Factored-NLPs of different task plans contain common subgraphs, which correspond to sequences of partial states that appear in both plans (potentially at different time indices).

During the execution of the Factored-NLP Planner (see Fig. 5.4), all solved subgraphs are stored either in a feasible or an infeasible database. Before solving a new nonlinear program, we check if it corresponds to a subgraph of any graph in the feasible database. This check requires a graph isomorphism test [Cordella et al., 2004], based on the adjacency structure and semantic information of the variable-vertices (the variable index $i = 1, \ldots, n$ and the name of the constraint $\phi \in \mathcal{H}$, without considering the time index). Given the available semantic information, the test is fast in practice (with complexity closer to $O\left((nK)^2\right)$ instead of the worst-case exponential).

**Infeasible subgraphs in TAMP** To conclude the section, Fig. 5.5 provides two examples of possible infeasible subgraphs of the Factored-NLP of the example domain (Fig. 9.3), together with an intuitive explanation of the underlying reason for the continuous infeasibility.

## 5.7. Reformulation of the Discrete Planning Task

In this section, we discuss how to reformulate the planning task with information about the continuous infeasibility (Step 4 of the Factored-NLP Planner, Fig. 5.4). Specifically, given an infeasible subgraph $M$, we modify the discrete planning task $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ to ensure

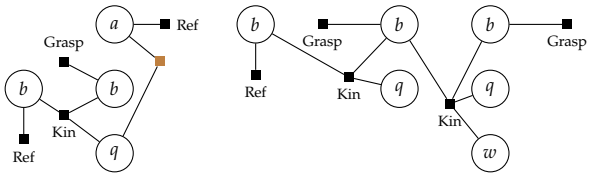



Figure 5.5.: Two examples of possible infeasible subgraphs of the Factored-NLP shown in Fig. 5.3. *Left*: the robot $Q$ cannot pick up object $B$ from its initial position if $A$ is also in the initial position, i.e., $A$ blocks the grasp of $B$. *Right*: It is not possible to pick up object $B$ with the robot $Q$ and then do a handover to robot $W$, e.g., due to kinematic constraints, robot $Q$ can only pick up the object in a certain way that prevents a handover later.

that the discrete planner will never generate plans whose Factored-NLP contains $M$. The mapping is achieved through a two-step process:

First, we translate the infeasible subgraph $M = (X_M \cup \Phi_M, E_M)$ into a sequence of discrete partial states $\langle p_0, \dots, p_L \rangle$. Recall that each constraint $\phi \in \Phi_M$ was generated by the mapping $\Pi : (p, \tilde{p}) \mapsto \phi$. We now trace this mapping back to obtain $(p, \tilde{p})$ which generated $\phi$, maintaining the relative order of the partial states. Importantly, we use $p_0$ to denote the first partial state that appears in the conflict, which could correspond to any step $k = 1, \dots, K$ of the task plan.

Given an infeasible sequence of partial states $\langle p_0, \dots, p_L \rangle$, we introduce a compilation that eliminates plans containing $\langle p_0, \dots, p_L \rangle$ starting at any time index, similarly to the plan forbidding compilation [Katz et al., 2018b]. Our compilation introduces binary variables $l = 0, \dots, L$ to indicate whether the path from $s_0$ to $s_K$ contains the infeasible sequence of partial states. Specifically, $b_l = 1$ indicates that the current path contains the first $l + 1$ elements of the infeasible sequence.

Given a planning task $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ and an infeasible sequence $\langle p_0, \dots, p_L \rangle$, the new discrete planning task is $\langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$, where:

- $\mathcal{V}' = \mathcal{V} \cup \{b_0, \dots, b_L\}$,
- $s_0' = s_0 \cup \{b_l = 0 \mid l = 1, \dots, L\} \cup \{b_0 = 1 \text{ if } p_0 \subseteq s_0;\ b_0 = 0 \text{ otherwise}\}$,
- $g' = g \cup \{b_L = 0\}$,
- $\mathcal{A}' = \{a' = \text{mod}(a),\ a \in \mathcal{A}\}$,

where $a' = \text{mod}(a)$ modifies action $a$ by adding conditional effects to ensure that if action $a$ is executed when $b_{l-1} = 1$, and executing $a$ makes $p_l$ true, then $a$ sets $b_l = 1$ and $b_{l-1} = 0$. Alternatively, if $a$ is executed when $b_{l-1} = 1$, and it does not make $p_l$ true, then $a$ sets

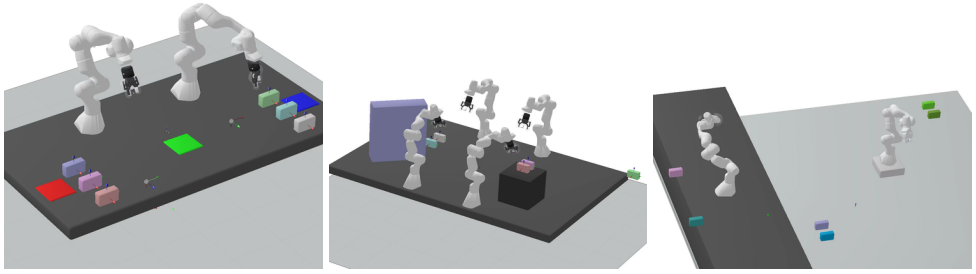


Figure 5.6.: Three environments used in the benchmark. *Left*: *Laboratory*. *Center*: *Workshop*. *Right*: *Field*.

$b_{l-1} = 0$. The last binary variable $b_L$ cannot transition from 1 to 0 (i.e., $b_L = 1$ is a dead end). The formal reformulation $a' = \text{mod}(a)$ is shown in Appendix B of our paper.

We can now state the proposition which shows that this compilation eliminates exactly all solutions which satisfy $\langle p_0, \ldots, p_L \rangle$.

**Proposition 5.1.** *Let $T = \langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ be a SAS+ planning task, $\langle p_0, \ldots, p_L \rangle$ be some infeasible sequence, and $T' = \langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$ be the reformulation described above. A plan $\pi$ is a solution of $T'$ if and only if $\pi$ is a solution of $T$ and the states along $\pi$ do not contain any sequence of states $\langle s_k', \ldots, s_{k+L}' \rangle$ such that $p_l \subseteq s_{k+l}'$ for $l = 0, \ldots, L$, starting at any $k$.*

Multiple infeasible sequences are forbidden by iterative reformulation. We are now ready to discuss the properties of our Factored-NLP Planner (Fig. 5.4):

**Theorem 5.1.** *If the underlying classical planner is sound and complete, and the nonlinear optimizer always finds a feasible solution if one exists, then the Factored-NLP Planner is sound and complete.*

*Proof Sketch:* The proof follows from the fact that any sequence we forbid cannot be part of any feasible solution (because it generates a Factored-NLP with a subgraph found to be infeasible, Property 5.2), together with the fact that our compilation eliminates only plans which contain these sequences (Proposition 5.1). The completeness of the algorithm does not require the infeasible subgraphs to be minimal, nor the mapping $\Pi$. Nevertheless, these properties are desirable for an efficient algorithm.

## 5.8. Experimental Results

Our algorithm is evaluated in three different simulated scenarios, where the goal is to move obstacles, rearrange, and stack up to six blocks to build towers with several robots. The evaluation on real robots is reported in Section 5.8.5.

1. *Laboratory (Lab)*: Two 7-DOF manipulator arms execute pick and place actions to build a tower. The solution requires handovers, regrasping, and removing obstacles. It is based on a real-world setting, Fig. 5.8.
2. *Workshop (Work)*: An extension of the *Laboratory* scenario that includes four robots and a stick that can be grasped and used as a tool to reach blocks, Fig. 5.1.
3. *Field*: Contains a fixed 7-DOF manipulator and a mobile 7-DOF manipulator, with two additional action operators: *start-move* and *end-move*, for moving the base of the mobile robot on the floor, Fig. 5.1.

For each scenario, we generate five different problems (e.g., *Lab_{1,2,3,4,5}*) by modifying the discrete goal and the initial configuration to increase complexity at both the discrete and geometric levels, while keeping the number of movable objects constant (except for the easier versions *Work_1* and *Field_1*).

### 5.8.1. Relaxations for Finding Infeasible Subgraphs

The formulation *PNTC* and the solver are general and domain-independent. Domain knowledge is introduced through the relaxations used for extracting minimal conflicts (Section 5.6). The following relaxations (applicable in any TAMP problem) are used in the benchmark scenarios:

- *Removal of trajectories*: The remaining graph only contains variables for the keyframes, considerably reducing the dimensionality of the nonlinear program while still detecting most of the geometric infeasibilities. We note that this is the most important relaxation because the dimensionality of the underlying nonlinear program is reduced considerably (a trajectory is represented with, e.g., 20 waypoints, while a keyframe variable corresponds to a single waypoint).
- *Removal of collision constraints*: Collision constraints connect all robot configuration and object pose variables at the same time step, resulting in a densely connected graph (see Fig. 9.3). Without collisions, the graph becomes sparse, and object and robot variables are only connected by grasping, kinematics, and placement constraints.
- *Removal of time consistency*: Time-consistency constraints (*Equal* in Fig. 9.3) appear when objects are not modified by a discrete action. This relaxation does not consider the long-term dependencies of the manipulation sequence and creates a sparse time structure.
- *Removal of robot variables*: The remaining graph considers only the variables for the objects, detecting infeasible placements due to collisions between objects.

Table 5.1.: Number of NLP evaluations and CPU time, averaged over 10 randomly seeded runs, with standard deviations in gray.

| | *length* | | *One-way* | | *MBTS* | | *FNPP_t* | | *FNPP_tr* | | *FNPP_trn* | | *FNPP_trng* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $N_0$ | $N$ | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* |
| Work_1 | 2 | 4 | 77.0 0.0 | 8.2 1.0 | 371 58.5 | 23.5 4.4 | 50.8 12.0 | 5.7 1.2 | 53.0 12.6 | 5.7 1.3 | 60.8 10.5 | 5.6 0.9 | 55.2 1.4 | **5.3** 0.1 |
| Work_2 | 4 | 6 | - | - | - | - | - | - | 113 40.4 | 22.7 9.2 | 94.7 1.7 | 19.7 6.0 | 86.8 1.5 | **16.7** 2.9 |
| Work_3 | 4 | 6 | - | - | - | - | - | - | 105 37.0 | **19.1** 5.3 | 95.6 1.0 | 22.1 5.9 | 86.1 1.5 | 21.4 6.1 |
| Work_4 | 8 | 10 | - | - | - | - | - | - | 282 0.0 | **53.1** 5.6 | 307 1.9 | 56.4 7.6 | 270 1.8 | 55.4 9.0 |
| Work_5 | 8 | 11 | - | - | - | - | - | - | - | - | 355 4.9 | **76.0** 7.0 | 309 5.3 | 76.8 9.4 |
| Lab_1 | 2 | 3 | 25.0 0.0 | 7.1 1.0 | 25.0 0.0 | 4.1 0.5 | 21.0 0.0 | 4.5 0.2 | 25.0 0.0 | **3.2** 0.2 | 30.0 0.0 | 3.3 0.1 | 28.0 0.0 | 3.3 0.2 |
| Lab_2 | 2 | 3 | 12.0 0.0 | 3.1 0.5 | 28.9 0.3 | 3.7 0.5 | 32.0 0.0 | 5.5 0.3 | 46.0 0.0 | 4.3 0.2 | 23.0 0.0 | **2.1** 0.1 | 21.0 0.0 | **2.1** 0.1 |
| Lab_3 | 4 | 5 | 19.0 0.0 | 8.4 1.2 | 34.0 0.0 | 18.5 2.7 | 26.0 0.0 | 5.9 0.4 | 23.0 0.0 | **3.1** 0.2 | 25.0 0.0 | 3.3 0.4 | 24.0 0.0 | 3.2 0.2 |
| Lab_4 | 4 | 9 | - | - | - | - | - | - | - | - | 70.0 0.0 | 6.5 0.6 | 60.1 3.5 | **6.3** 0.4 |
| Lab_5 | 12 | 17 | - | - | - | - | - | - | 87.0 0.0 | **18.7** 1.8 | 93.0 0.0 | 19.1 2.0 | 83.0 0.0 | 19.0 2.0 |
| Field_1 | 2 | 4 | 19.0 0.0 | 7.0 2.0 | 90.0 0.0 | 15.3 3.1 | 19.0 0.0 | 5.7 1.1 | 14.1 0.3 | 2.9 1.4 | 16.0 0.0 | **2.6** 0.3 | 16.0 0.0 | 3.1 1.0 |
| Field_2 | 2 | 6 | - | - | - | - | - | - | 46.0 0.0 | **6.1** 0.4 | 53.0 0.0 | 6.3 0.5 | 52.5 0.5 | 6.3 0.5 |
| Field_3 | 4 | 8 | - | - | - | - | - | - | 75.0 0.0 | **11.7** 1.2 | 84.0 0.0 | 12.3 1.4 | 78.6 0.5 | **11.7** 0.7 |
| Field_4 | 6 | 10 | - | - | - | - | - | - | 67.0 0.0 | **13.2** 1.5 | 77.0 0.0 | 13.6 1.6 | 76.0 0.0 | 13.5 1.3 |
| Field_5 | 6 | 11 | - | - | - | - | - | - | 282 0.0 | 56.5 6.6 | 289 1.0 | **50.7** 5.5 | 262 0.5 | 51.4 6.6 |

### 5.8.2. Benchmark

**Algorithms under comparison** We compare our approach with two different formulations that combine a discrete search with joint nonlinear optimization for solving Task and Motion Planning problems.

- *One-way interface between Top-K Planning and a nonlinear optimizer (One-way)*. This baseline combines Top-K planning [Katz et al., 2018b] to generate a set of different task plans with a nonlinear optimizer to evaluate the plans. The planner does not receive any information about the geometric reasons for infeasibility and only blocks the evaluated plans.
- *Multi-Bound Tree Search (MBTS)*. The MBTS Solver [Toussaint and Lopes, 2017] incrementally builds a tree in a breadth-first order to explore sequences of discrete actions that reach the high-level goal. Instead of solving the full continuous optimization problem directly, MBTS first computes relaxed versions (*bounds*) that consider a subset of variables and constraints (see Section 2.3.1).
- *Four Variations of our Factored-NLP Planner (in short, FNPP)*. We evaluate our full planner *FNPP_trng*, and three additional versions: *FNPP_t*, *FNPP_tr*, and *FNPP_trn* to conduct an ablation study of the algorithm to extract infeasible subgraphs. Suffixes indicate: $t$=time search, $r$=relaxation, $n$=convergence heuristic, and $g$=feasible graph database.

**Metrics** Each algorithm is run 10 times with different random seeds and a timeout of 100 seconds. For each method, we report on the number of evaluated NLPs (*NLP*) (lower is better) and the CPU time in seconds (*time*) (lower is better) in Table 5.1. "–" indicates a failure to find a solution within 100 seconds with at least a 70 % success rate.

*Time*[2] provides an objective way to compare algorithms that use different underlying methods. The number of solved NLPs is informative but does not capture the influence of the size and feasibility of NLPs on the running time of the solver.

For each problem, $N$ denotes the length of the shortest found task plan that is geometrically feasible, and $N_0$ is the length of the task plan that solves the initial discrete task (that is, without considering the continuous constraints). $N$ and $N_0$ are proxies for the difficulty: the number of candidate plans typically grows exponentially with $N$, and the difference $N - N_0$ indicates the amount of information about the continuous constraints that should be provided to the discrete planner. The approximate branching factor is 12 in *Lab_{1,2,3,4,5}*, 13 in *Field_{2,3,4,5}*, 24 in *Work_{2,3,4,5}*, 4 in *Work_1*, and 5 in *Field_1*.

[2]Experiments are conducted on a Single Core i7-1165G7@2.80GHz

**Comparison to baselines** Concerning the problems solved, *One-way* and *MBTS* can only solve the easier problems in each scenario, while *FNPP_trn/trng* solves all the problems. Our algorithm is significantly faster in the problems also solved by *One-way* and *MBTS*, because the more efficient encoding of geometric information reduces the running time.

The success rate of our planners *FNPP_trn/trng* is 100 % in all problems except for *Field_5* (80 %), *Work_4* (95 %), and *Work_5* (90 %) where the optimizer fails to solve feasible Factored-NLPs in a few runs. The performance of *FNPP_trng* is not affected by the branching factor of the underlying problem and provides good scaling with respect to $N$ and $N - N_0$. The highest computational time corresponds to *Field_5* and *Work_5*, that require a long plan and detecting of collisions between movable objects. In TAMP, the practical size of the Factored-NLPs is $O(n^2K)$ (where $K$ is the length of the action sequence, and $n$ is the number of objects and robots). The domains are modelled using a small set ($< 20$) of different types of nonlinear constraints (e.g., Fig. 9.3).

### 5.8.3. Ablation Study

- *Analysis of the relaxations: FNPP_t* detects conflicts of the form $\langle s_k, \ldots, s_{k+l} \rangle$, while *FNPP_tr* checks relaxations to generate smaller conflicts $\langle p_k, \ldots, p_{k+l} \rangle$. Small conflicts lead to more aggressive pruning of task plans and are essential to solve the harder problems (the number of solved problems is 5 versus 13 out of 15). After an ablation study of each relaxation, we observe that the *Removal of trajectories* and the *Removal of collision constraints* are the most informative relaxations.
- *Analysis of the convergence heuristic:* The results show that the convergence heuristic is important in problems that require reasoning about the collisions between movable objects, e.g., when the robot must move one object before placing another to avoid a collision. In this case, the relaxations are not informative, while the convergence point of the optimizer in these infeasible problems usually indicates which objects are in collision. *FNPP_tr* solves 13 out of 15, and *FNPP_trn* solves 15 out of 15.
- *Analysis of the database of feasible graphs: FNPP_trng* reduces the number of solved NLPs, from a total average of 1673 to 1508, but there is no improvement in computational time. We conjecture that the database approach will provide higher benefits in a setting where solving the NLPs requires more time.

### 5.8.4. Scalability

We conduct two additional experiments in the *Laboratory* scenario to explicitly evaluate the scalability of the method when increasing the number of blocks to be stacked (from 4 to 32) and the number of movable obstacles on a cluttered table (from 1 to 6).

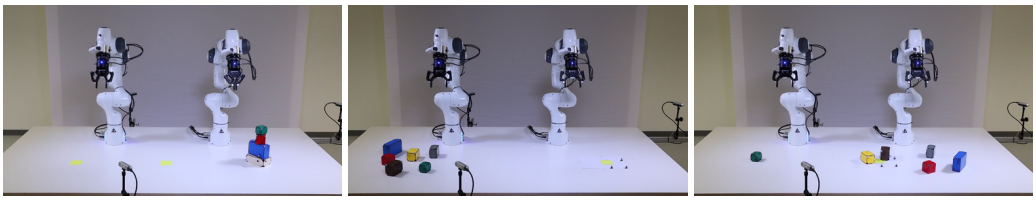


Figure 5.7.: Three scenarios in the real world evaluation. *Left*: *Hanoi-Tower*. *Center*: *Tower*. *Right*: *Obstacles-Tower*.

Experimentally, the running time of the Factored-NLP Planner scales polynomially with the number of objects and plan length, and the practical bottleneck is the time spent on solving large nonlinear programs (with cubic complexity on the number of objects and linear on the plan length).

The main weakness of our method is that the nonlinear optimizer is not guaranteed to find a solution for a (sub)Factored-NLP even if one exists, given that the nonlinear constraints define a non-convex optimization problem (which could break the assumption in Theorem 5.1). However, the extensive experiments demonstrate that the solver is efficient and reliable in relevant use-cases of TAMP.

### 5.8.5. Real-Time Planning in the Real World

We demonstrate our solver in a real-world version of the *Laboratory* environment (two 7-DOF manipulators and up to 6 movable objects, see Fig. 5.8. The solver is integrated into a *Sense-Plan-Act* pipeline, where we first perceive the scene with an external motion capture system, compute a full discrete and continuous plan, and then execute the plan.

The real-world evaluation consists of three scenarios: *Tower*, *Hanoi-Tower*, and *Obstacles-Tower*, for a total of 11 problems (see Fig. 5.7). The high-level goal is to build a tower of cubes at different locations: *Hanoi-Tower* introduces the classical Hanoi logic constraints, and *Obstacles-Tower* requires plans that first move obstacles away. The planning time differs across problems: 2.8 s to build a tower of 6 blocks in the center of the table (12 discrete actions), 8.8 s to remove two obstructing blocks and stack 4 blocks (12 actions), 9.4 s to build a Hanoi Tower (12 actions), and 27.2 s to solve a problem that requires removing obstructing blocks and transferring blocks from the left to the right side (16 actions). Recordings of planning and execution are shown on the project webpage[3].

[3] https://quimortiz.github.io/graphnlp/

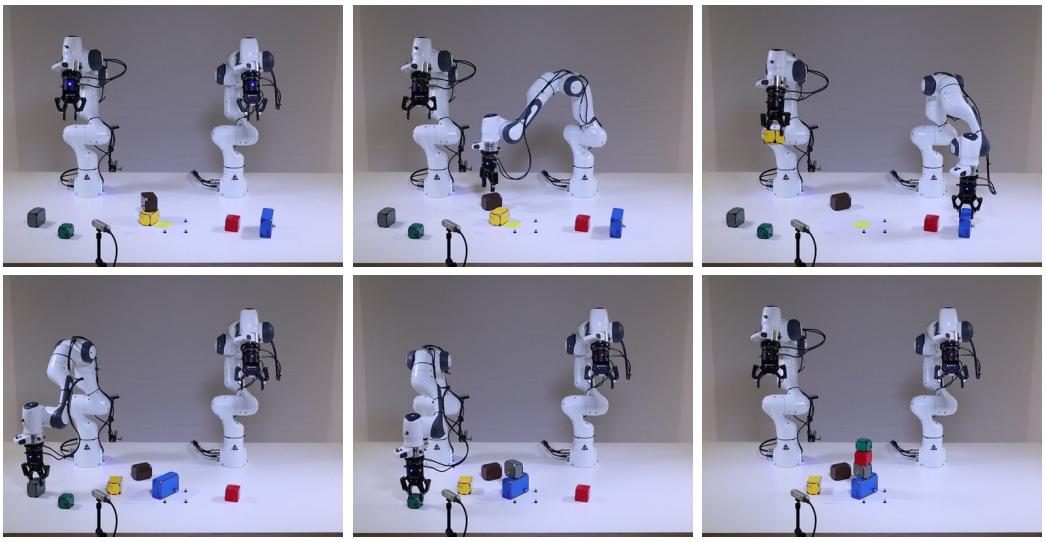


Figure 5.8.: The goal in this real-world experiment is to build the tower *blue-gray-red-green* in the central spot (highlighted in yellow). The solution, computed in only 8.88 s with our planner, requires a task plan with 12 discrete actions, moving first the *brown* and *yellow* blocks to avoid collisions.

## 5.9. Limitations

Similarly to our first contribution, *Diverse Planning for LGP*, described in Chapter 4, our TAMP solver shares the limitations of the underlying LGP formulation, namely the local convergence of optimization methods, which might prevent finding a solution even if a problem is feasible.

To address this issue, a possible solution is to store how many times a subgraph has been found to be infeasible and use this information in a soft conflict formulation (where infeasible subgraphs only penalize task plans) or in a probabilistic conflict formulation (see also the discussion in Section 4.8).

From an implementation perspective, our method requires a factored-NLP formulation where any subset of constraints can be evaluated for feasibility. This prevents the use of off-the-shelf trajectory optimization frameworks, which often assume a non-factored, unstructured trajectory optimization problem.

Solving large factored nonlinear programs with joint optimization can be inefficient and is more prone to converging to infeasible local minima. A natural way to address this issue is to combine both joint optimization and conditional constrained sampling within a single solver, bridging the gap between sample-based and optimization-based approaches

to TAMP. A foundational first step in this direction is presented in the second part of the thesis (Chapters 6 and 7), where we analyze how to design TAMP solvers that automatically select between sampling and optimization operations.

## 5.10. Conclusion

We present a solver that combines nonlinear optimization and PDDL planning for the joint optimization of discrete and continuous variables in robotic planning. The key contribution is the novel bidirectional interface between the task plan and the continuous constraints, realized through the detection of infeasible subgraphs and a reformulation to inform the task planner about subgraph infeasibility. The problem formulation is formalized as *PNTC*, which extends classical planning with nonlinear transition constraints.

Our experiments in Task and Motion Planning show that the algorithm is faster and more scalable than the Multi-Bound Tree search for LGP, while maintaining generality and using the same input information. These results are further validated in real-world experiments, where our solver generates plans for two 7-DOF robots and six objects in a few seconds.

In this chapter, we formalize the factored structure of trajectory optimization problems first presented in Chapter 3 in an illustrative manner. Besides planning, this structure is beneficial for reasoning about computational operations (see Chapter 6) and learning (see Chapters 8 and 9).

The structure of the Factored-NLPs in Chapters 6 and 9 is slightly different, as these methods use minimal representations, compressing several variables that are constant into a single variable. This results in a more compact representation but obscures the clear temporal structure presented here. On the other hand, the Factored-NLPs in Chapter 9 are built using the same formalization of Planning with Nonlinear Transition Constraints but with a redundant representation of the continuous state, which allows for better generalization across different task plans when using graph neural networks.

# Part II.

# Meta-Solvers: Adaptive Combination of Sampling and Optimization Methods

# Chapter 6

# Learning Optimal Sampling Sequences for Robotic Manipulation

## 6.1. Introduction

A core component of Task and Motion Planning (TAMP) is generating the keyframe configurations that fulfil the geometric and physical constraints of a fixed task plan. The sequence of keyframe configurations is often computed through either joint optimization or a predetermined series of sampling operations. However, both naïve sequential conditional sampling of individual variables and full joint optimization of all variables at once can be highly inefficient and non-robust, depending on the geometric environment. As an example, consider the keyframes for the task plan pick-handover-place as shown in Fig. 6.1, where obstacles, kinematic, and grasp constraints pose challenges for both sampling and optimization methods.

In this chapter[1], we present a novel algorithm that learns how to break a factored nonlinear program into smaller problems to generate solutions more efficiently.

Our method relies on a factored representation of the feasibility problems and utilizes Monte-Carlo Tree Search to learn assignment orders for the variables, with the aim of minimizing the computation time to generate feasible full samples. As an online learning algorithm, Monte-Carlo Tree Search is most valuable when multiple diverse solutions must be generated within a fixed computational budget, rather than seeking a single solution. The sparse factored representation of the nonlinear program, as presented in Chapter 3, together

[1]This chapter is based on the publication: Ortiz-Haro, J., Hartmann, V. N., Oguz, O. S., and Toussaint, M. (2021). Learning Efficient Constraint Graph Sampling for Robotic Sequential Manipulation. IEEE International Conference on Robotics and Automation (ICRA) (pp. 4606-4612).

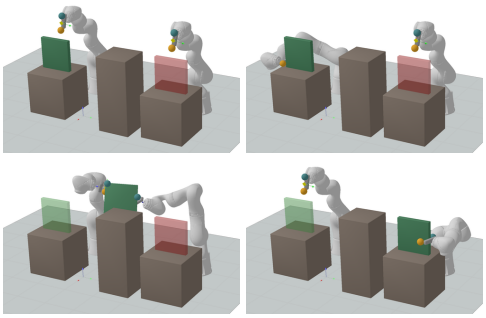


Figure 6.1.: A sequence of keyframes in the *Handover* problem. The fixed task plan is to move the green box to the target location (red) by performing an intermediate handover. Robot $Q$ picks the box (top-right) and hands it to robot $W$ (bottom-left) to place it in the goal position (bottom-right).

with local information about the dimensionality of individual variables and constraints, is used to reduce the space of possible computations.

We demonstrate that our learning method quickly converges to the best sampling strategy for a given problem, outperforming user-defined orderings or fully joint optimization, while also providing higher sample diversity.

This work is a first step towards meta-solvers for TAMP, where the ultimate goal is to develop a flexible TAMP solver that automatically selects between sequential sampling and joint optimization to compute robot motions, while also balancing the search in the discrete and continuous levels of the TAMP problem.

To this end, in this chapter we first consider the subproblem of generating keyframes for a fixed high-level manipulation plan. Though not encompassing the entire TAMP problem, this is a vital component, as keyframes are later used as waypoints for subsequent trajectory optimization or sampling-based motion planning algorithms. Especially, this setting already reveals the complex decision-making and infrastructure required to choose between sequential sampling or joint optimization operations.

In the following Chapter 7, we extend some of the ideas and techniques presented here to address the full TAMP problem, where the high-level task plan is not fixed, but also optimized.

## 6.2. Related Work

**Problem decomposition and constraint satisfaction** In continuous optimization, decompositions utilizing the underlying problem structure can enhance the performance of al-

gorithms both theoretically and practically [Dantzig and Wolfe, 1960, Benders, 1962, Boyd et al., 2011, Bertsekas, 1979].

A constraint satisfaction problem (CSP) [Rossi et al., 2006] reveals a graph dependency structure between variables and constraints and often presumes finite discrete domains for each variable. Algorithms solving CSPs can exploit the graph structure, for example, identifying connected components or trees, to efficiently assign variables [Mouhoub and Jafari, 2011]. There is growing interest in sampling solutions that satisfy CSPs [Dechter et al., 2002, Gogate and Dechter, 2006, Ermon et al., 2012]. Generating multiple solutions to the problem at hand enhances the applicability of these methods in scenarios where some constraints cannot be modeled or are evaluated only after the fact [Danna et al., 2007].

In robotics applications, such as [Driess et al., 2019, Toussaint and Lopes, 2017, Tonneau et al., 2018, Orthey et al., 2018, Hartmann et al., 2020] among many others, complex nonlinear problems are frequently decomposed into a sequence of simpler subproblems. The solutions to the simpler problems are then used to guide optimization and sampling methods toward the solution of the comprehensive problem.

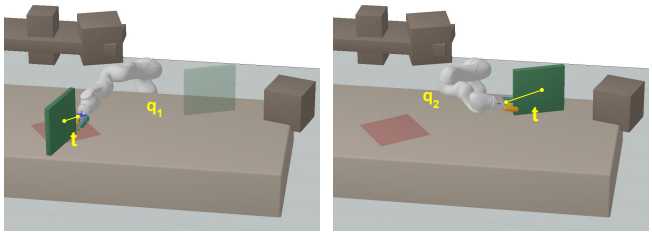
**Meta-decision processes** The optimization of computational operations is directly related to optimal metareasoning [Russell and Wefald, 1991]. Metareasoning has varied applications in heuristic search [O'Ceallaigh and Ruml, 2015, Lieder et al., 2014, Zilberstein, 2008, Karpas et al., 2018], as well as in other resource-limited planning processes [Bratman et al., 1988, Boddy and Dean, 1989]. More recently, metareasoning has been applied to reinforcement learning [Pascanu et al., 2017], temporal planning [Cashmore et al., 2018], and path planning [Mandalika et al., 2019].

In our work, we employ bandit algorithms [Auer et al., 2002, Kocsis and Szepesvári, 2006] to deliberate on the sequence of computational operations for generating solutions to a nonlinear program, where the decisions we make concern the selection of which variables to compute next.

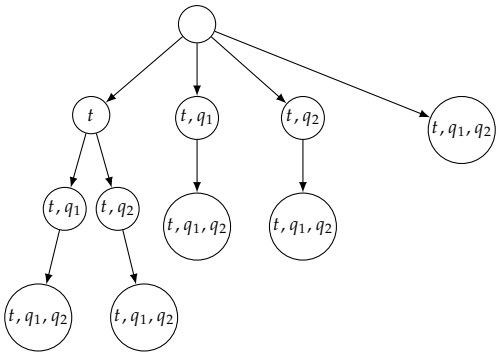
## 6.3. Sampling Sequences in the Pick and Place Task Plan

Consider the problem of computing the keyframe configurations for a Pick and Place task plan, defined by three vector variables: the robot configuration for picking and placing the object, and the relative transformation between the gripper and the object, Fig. 6.2a.

Some potential sampling sequences to generate a full solution are: (i) optimize all variables jointly, (ii) sample the relative transformation first, then compute the robot configurations, or (iii) compute the pick configuration and the relative transformation jointly first, followed by the place configuration.




(a) Visual representation of the Pick and Place problem. Left image shows the pick keyframe and right image shows the place keyframe. Variables are the robot configuration when picking $q_1$, the robot configuration when placing $q_2$, and the relative transformation between the gripper and the object $t$, which is kept constant in both keyframes because the grasp is stable.

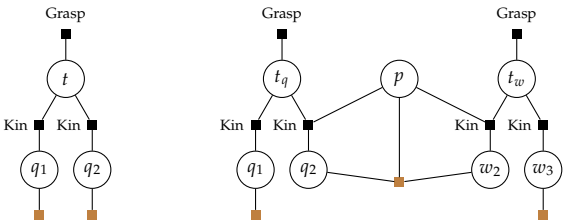



(b) Possible sampling sequences for solving the Pick and Place problem.

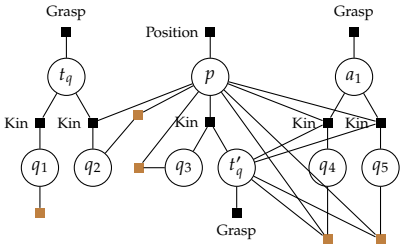
Figure 6.2.: The Pick and Place task plan.

All possible sequences of sampling operations can be represented as a tree, where each node indicates the subset of variables that have been computed, and each edge denotes a sampling operation. We illustrate the sampling tree for the Pick and Place task plan in Fig. 6.2b. For clarity, we omit sampling operations that have zero probability of success, a process detailed later in Section 6.5.2, resulting in five possible sampling sequences.

The efficacy of each strategy depends on the computational time of the individual operations, their success rates, and the probability of feasibility of subsequent sampling steps, since often partial assignments are not viable for subsequent steps. For example, a valid relative transformation between the gripper and the object might not permit an inverse kinematics solution due to collisions or the robot's reachability limits.




(a) Pick and Place. (b) Handover.




(c) Banana.

Figure 6.3.: Factored-NLPs of the three task plans considered in this chapter. Circles are variables and squares are constraints. Brown squares represent collision constraints. See the main text and Chapter 3 for a description of the variables and constraints. *Pick and Place* is used to illustrate and analyze different sampling sequences. The more complex *Handover* and *Banana* tasks are used to evaluate our method.

## 6.4. Sequential Sampling in Factored-NLPs as a Markov Decision Process

In this chapter, we present a framework to learn the optimal strategy for generating multiple diverse solutions for a Factored-NLP (Section 3.1), where optimal is defined as maximizing the number of diverse solutions found within a fixed computational time.

The Factored-NLPs of the three manipulation tasks considered in this chapter are shown in Fig. 6.3. Variables $q$ denote the configuration of the robot, with the subscript indicating different time-steps. There are two types of variables for the configuration of objects: variables $p$ denote the absolute positions of objects, and $t$ represents relative transformations with respect to the robot gripper. In *Handover* (Fig. 6.1), the configurations for the second robot are denoted with $w$. In *Banana* (Fig. 6.4), $t$ represents the grasp of the box, and $a$ represents the grasp of the banana. The meaning of each constraint is explained in detail in Chapter 3.

Note that the Factored-NLPs used in this chapter have some differences compared to the ones presented in Chapters 3 and 5. First, we only consider the keyframes of a task plan and, therefore, do not include variables to represent trajectories. Second, we use here a more compact representation of the optimization problem, where all variables that are constrained to be fixed are removed, and consecutive free variables that are constrained to be equal are condensed into a single variable – resulting in a less structured and more problem-specific representation.

Given a Factored-NLP with a set of $N$ variables $X = \{x_1, \dots, x_N\}$, $x_i \in \mathbb{R}^{n_i}$, and constraints $\Phi = \{\phi_1, \dots, \phi_L\}$, we first observe that a complete value assignment to all the variables in $X$ can be computed in different ways: variables could be assigned jointly, one by one, or following any user-defined order.

To formalize this, we define an *assignment state* $w \in W$ to indicate the set of variables of $X$ that have been assigned. The set of all possible assignment states $W$ is the power set $P(\{1, \dots, N\})$, i.e., the set of all subsets of $\{1, \dots, N\}$. For instance, an assignment state $w' = \{1, 2, 3\}$ means that values have been assigned to the variables $x_1, x_2$, and $x_3$. Conversely, $x_w$ denotes the subset of variables in $X$ indicated by $w$, e.g., $x_{w'} = \{x_1, x_2, x_3\}$.

Hence, any full assignment for all $x_i$ can be computed following a sequence of assignment states $H = (w_0, w_1, \dots, w_g)$, where $w_i \subset w_{i+1}$, and $w_0 = \{\}, w_g = W$. For any given step $i$, the sequence up to that point is represented as $H_i = (w_0, \dots, w_i)$.

A transition $w_i \rightarrow w_j$ in a sequence implies a computational operation

$$x_{w_j} = o_{w_i, w_j}(x_{w_i}), \tag{6.1}$$

that assigns numerical values to the variables indexed in $w_j \setminus w_i$, conditioned on the given values in $x_{w_i}$, which are kept fixed. If the compute operation is successful, the new assignment state is $w_j$, and the value for all computed variables so far is $x_{w_j}$ (note that $x_{w_j}$ also includes the values of $x_{w_i}$ for the variables in $w_i$).

The set $\mathcal{Y}(w_i) = \{w_j \in W \mid w_i \subset w_j\}$ denotes the assignment states that can be reached with a valid transition from $w_i$. The computational operations are implemented either with direct conditional sampling or using optimization methods initialized with a randomized guess. For instance, in the Pick and Place problem presented in the introduction, examples of such transitions include generating 6-DOF grasps or solving inverse kinematics (i.e., computing the robot joint values for a given end-effector position).

Algorithm 6.1 shows how to generate solutions from a Factored-NLP by choosing a valid transition in *assignment space* and executing the corresponding compute operation at each step. The performance of the algorithm depends heavily on how transitions are selected (our method described in Section 6.5.1 is in orange).

Each transition $w_i \rightarrow w_j$ is a conditional sampling operation, which generates samples with an unknown but defined conditional probability density $p_{w_j|w_i}(x_{w_j}|x_{w_i})$, defined on the feasible manifold of $x_{w_j}$. There are no assumptions about the shape of the distribution. We assume that the probability density is zero everywhere in case the transition conditioned on $x_{w_i}$ is infeasible. The joint probability density is

$$p_{H_j}(x_{w_j}) = p_{w_j|w_i}(x_{w_j}|x_{w_i}) \cdot p_{H_i}(x_{w_i}) \,, \tag{6.2}$$

and it depends on the history $H_j = (w_0, \ldots, w_i, w_j)$ of transitions to reach $w_j$.

A valid transition $w_i \rightarrow w_j$ between assignment states does not always produce an assignment that satisfies the associated constraints for $x_{w_j}$. There are two sources of infeasibility: *(i)* the problem conditioned on the previously assigned variable $x_{w_i}$ could be infeasible, or *(ii)* the sampling operation could fail to generate a solution even if one exists.

We can define a transition probability for transitions between the current sequence $H_i$ and the next assignment state $w_{i+1}$ (equivalently, between $H_i$ and $H_{i+1} = H_i \cup \{w_{i+1}\}$) as follows:

$$Pr\left(H_i \rightarrow H_{i+1}\right) = \hat{p}_{H_i,H_{i+1}} \cdot \frac{\int_{\Omega_{i,i+1}} p_{H_i}(x_{w_i})\, d\Omega}{\int_{\Omega_i} p_{H_i}(x_{w_i})\, d\Omega}, \tag{6.3}$$

where $\Omega_i$ is the feasible space for $x_{w_i}$ and $\Omega_{i,i+1} \subseteq \Omega_i$ is the feasible space for $x_{w_i}$ for which a joint sample $x_{w_{i+1}}$ exists. In this estimate, the success rate $\hat{p}_{H_i,H_{i+1}} \in (0,1)$ represents the probability that a solution satisfying the constraints will be found if one exists, which we estimate as constant for a given transition, rather than depending on the previous assignment $x_{w_i}$. Note that the success rate could be low for certain transitions, while it could be close to 1 for others.

**Assignment state transition as a Markov Decision Process** We can now define a Markov Decision Process (MDP): The state space is $\{H_i\} \cup \{\neg\}$, i.e., the sets of partial sequences starting from $w_0$ plus an infeasible state $\neg$. The action space contains the possible transitions between sequences, i.e., adding a new assignment state. Each action has a success rate $p_a = Pr(H \rightarrow H')$ of reaching the new sequence as defined in (6.3), (corresponding to the probability of generating an assignment satisfying the constraints) and a probability $1 - p_a$ of going to the infeasible state $\neg$. We note that the MDP has a maximum horizon length of $N$ transitions, which corresponds to assigning all the variables one by one.

We use the reward structure to obtain a concept of optimality that directly relates to our objective of maximizing the number of samples: We introduce a reward $r_g = 1$ for reaching the goal $w_g = W$ (namely, all variables have been correctly computed), and a stochastic cost, i.e., a negative reward, $r_t$ on each transition, which is the compute time the transition takes. We weigh the two costs linearly, obtaining $\lambda r_t + (1 - \lambda) r_g$ with $\lambda \in (0,1)$.

## 6.5. Choosing Computational Operations with Monte-Carlo Tree Search

We use Monte-Carlo Tree Search (MCTS, [Browne et al., 2012]) on the previously defined MDP to find the optimal sequence of computational operations.

MCTS incrementally builds a tree of possible transition sequences to find the most promising one. This is achieved by randomly selecting transitions starting from the initial state $w_0$ of the MDP, executing their corresponding conditional sampling operations, and using the reward structure of the MDP to guide the search. Thus, in our setting, the nodes of the tree built by the MCTS are sequences of assignment states.

The reward structure of the MDP guides the tree search algorithm to choose transitions that have a low computational cost and a high probability of producing a full solution to the Factored-NLP.

An estimate of the average computational cost $\hat{c} = -\sum r_t$ can be used as guidance for choosing a suitable $\lambda = \frac{1}{\hat{c}+1}$. In this case, the optimal solution in the MDP corresponds to the sequence of computational operations that maximizes the number of generated samples for the original factored nonlinear program.

### 6.5.1. Upper Confidence Tree (UCT)

In particular, we use the UCT algorithm [Kocsis and Szepesvári, 2006] to balance the exploitation of known sequences with the exploration of new sequences. UCT expands the child node $z$ that maximizes

$$Q_z + c\sqrt{\frac{\ln M_z}{m_z}}, \tag{6.4}$$

where $Q_z$ is the current estimate of the expected reward of node $z$, $M_z$ is the number of simulations that have evaluated the parent node, $m_z$ is the number of rollouts that have evaluated node $z$, and $c$ is a constant chosen by the user. The estimate of the $Q$ function for the nodes of the tree converges as the number of rollouts increases. Hence, the UCT algorithm incrementally builds a search tree, estimates the $Q$ values of each transition, and chooses actions using the upper confidence bound (6.4), leading to Algorithm 6.1.

We note that, if at least one of the sampling sequences assigns a non-zero probability density to the entire feasible manifold, so does our algorithm, since UCT never stops exploring all possible sequences.

**Algorithm 6.1** Framework to generate solutions from a Factored-NLP, with our contributions in orange.

**Input:** Factored-NLP, compute time budget $T$
$L \leftarrow \{\}$ ▷ Empty list of samples
**while** accumulated compute time $t \leq T$ **do**
  feasible $\leftarrow$ True
  $w \leftarrow w_0$ ▷ Initial empty assignment state ($w_0 = \emptyset$)
  $x_w \leftarrow x_{w_0}$ ▷ Initial empty sample ($x_{w_0} = []$)
  **while** $w \neq W$ **do**
    $w' \leftarrow$ *Choose next state from* $\mathcal{Y}(w)$ using UCT
    $x_{w'}$, feasible, reward $\leftarrow o_{w,w'}(x_w)$ ▷ Execute computational operation
    *Update UCT tree with* reward
    **if** feasible **then**
      $w \leftarrow w'$
      $x_w \leftarrow x'_w$
    **else**
      **break**
    **end if**
  **end while**
  **if** feasible **then**
    append $x_W$ to $L$
  **end if**
**end while**
**Output:** List of valid samples $L$

### 6.5.2. Pruning the Sampling Tree Using the Factored-NLP

Since the number of sequences and transitions grows exponentially with the dimension of the problem (see Table 6.1), it is not possible to naively apply MCTS to the previously defined MDP. However, most real-world problems have a sparse structure that can be leveraged to significantly reduce the set of available sequences and transitions.

We provide two examples of transitions that can be pruned. Consider a Pick and Place problem represented with the variables $\{q_1, q_2, t\}$, where $q_1, q_2$ are the robot configurations for picking and placing, and $t$ is the relative transformation between the end-effector and the object (Fig. 6.2).

- We prune $q_1 \rightarrow (q_1, t)$ and $q_2 \rightarrow (q_2, t)$. If $q_1$ or $q_2$ is sampled independently at random, the probability of correctly grasping the object is zero.
- We prune $t \rightarrow (q_1, q_2)$ because $q_1$ and $q_2$ are independent if $t$ is fixed (and it does not make sense to generate $q_1$ and $q_2$ jointly).

Table 6.1.: Number of transitions between assignment states in the three evaluated task plans, before and after transition pruning.

| | Variables | Transitions Complete | Transitions after Pruning | Pruning Ratio [%] |
|---|---|---|---|---|
| Pick and Place | 3 | 19 | 8 | 57.89 |
| Handover | 7 | 2059 | 163 | 92.09 |
| Banana | 9 | 19171 | 534 | 97.21 |

More generally, we combine knowledge of the Factored-NLP structure with the local information in each variable and constraint, i.e., the number of available degrees of freedom and the number of equality constraints, to prune a transition $w_i \rightarrow w_j$ if it fulfills any of the two criteria:

**Zero probability of success** In general, the probability of sampling variables from the ambient space such that the equality constraints are fulfilled is zero. Hence, a transition can be pruned if the number of new, linearly independent, equality constraints exceeds the number of new degrees of freedom that are added in that transition.

**Equivalence under conditional independence** Given an assignment state $w_i$, two variables in the Factored-NLP are conditionally independent if all paths that connect them contain already assigned variables. Therefore, we delete a transition that jointly samples variables that are conditionally independent with respect to those assigned in $w_i$. In such cases, joint sampling means choosing an ordering for two conditionally independent processes (which is already available as alternative valid transitions).

In Table 6.1, we show how the number of transitions is reduced in the three scenarios evaluated in this chapter.

### 6.5.3. Family of Problems and Tree Warm Start

Obstacle and object configurations impact the computational time and the success rate of the conditional sampling operations, potentially changing the optimal sampling sequence. Thus, the best sampling order depends on both the Factored-NLP and the specific problem instance.

MCTS provides a good framework to incorporate the information gathered from solutions to similar problems as a warm start. We propose to warm-start the tree search by initializing $Q_z$ of each node with the average of the $Q$ values from previous problems, and set $m_z$ to

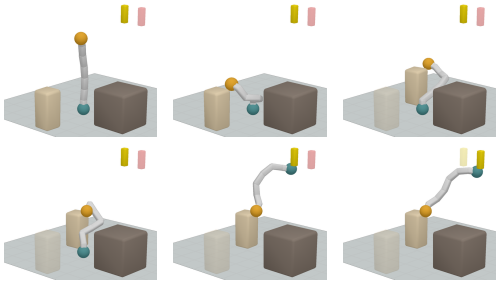


Figure 6.4.: A sequence of keyframes in one instance of the *Banana* problem. The task plan is to move the light brown box, climb on top of it, reach the banana, and place it on the red configuration. The dark brown object is an obstacle.

an equivalent count visit $m_{\text{equiv}}$, which models how confident we are with the warm start, following the approach presented in [Gelly and Silver, 2007].

## 6.6. Experimental Results

### 6.6.1. Scenarios

– *Handover*: Two robots (7D) have to collaborate to place a box in the goal configuration. Grasping is modeled with a two-finger gripper. Figure 6.1 shows a possible solution, and Fig. 6.3b shows the Factored-NLP.

– *Banana:* The hanging 'banana' has to be moved to a goal position. The robot (8D) can interact with the world using both sides of the kinematic chain as a gripper. The box has to be moved and used as a tool to stand on to be able to reach the goal. Grasping is modeled as 'grasp by touch'. Fig. 6.4 shows snapshots from a solution sequence, and Fig. 6.3c shows the Factored-NLP.

For both problems, 8 different instances (see Fig. 6.5 and Fig. 6.7) (i.e., varying obstacle and object configurations) are evaluated[2] multiple times, and unless stated otherwise, the average results are reported in the following analyses. The different instances can significantly impact the computation times and, thus, the optimal sampling sequence.

[2]Experiments are run using an Intel(R) Core(TM) i5-4200U 1.60GHz CPU.

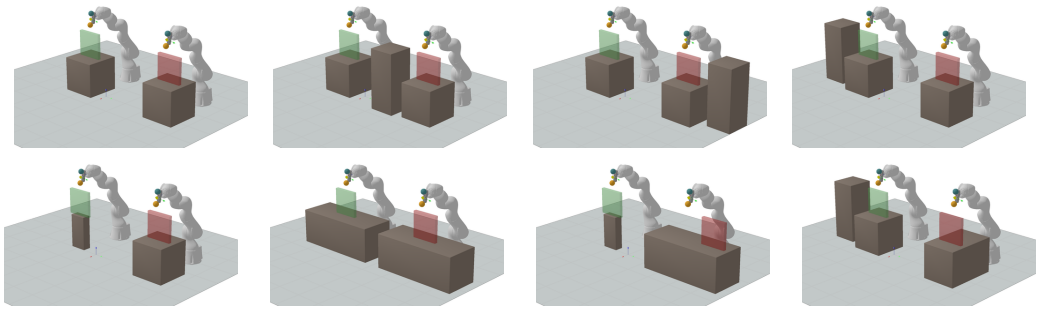


Figure 6.5.: *Handover* scenario. The obstacles (dark brown) constrain the possible grasps and handover positions of the robots. The optimal sampling sequence might vary in different scenarios, based on the number, size, and position of the obstacles.

### 6.6.2. Computational Operations

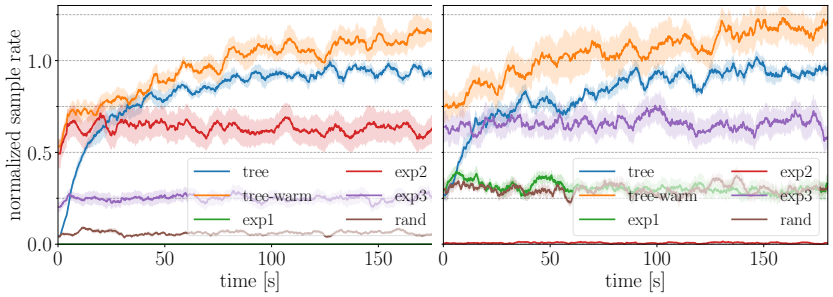
One motivation for this work is that both the computation time and the success rate of conditional sampling operations must be considered to design efficient algorithms. As an illustrative example, we report the average computational time and success rate of some sampling operations in the first instance of the *Handover* scenario: relative transformation (0.68 ms, 100 %), inverse kinematics (1.55 ms, 87 %), grasp of a fixed object (5.8 ms, 55 %), pick and place (54 ms, 46 %), and handover keyframe (68 ms, 44 %).

In practice, our algorithm finds the optimal balance between *(i)* choosing simple operations which are fast and have a high success rate but can potentially induce infeasibility in future assignments, and *(ii)* optimizing variables jointly, which considers joint feasibility but is slower and often has a lower success rate.

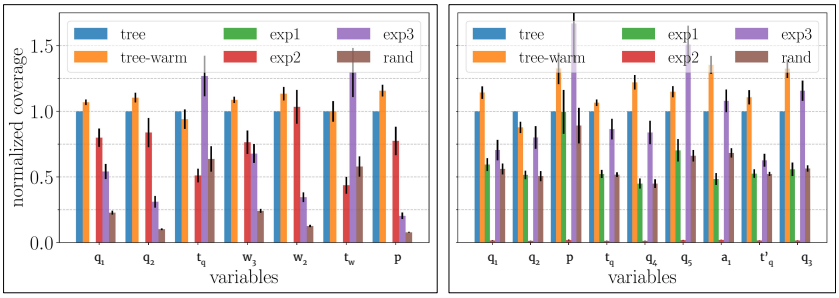
The conditional sampling operations are implemented by randomizing the initial guess of a nonlinear solver without a cost term and are solved using the Augmented Lagrangian algorithm. The initial guess covers the entire ambient space of the current partial assignment, ensuring that all possible partial solutions have a non-zero probability (see Section 6.4).

### 6.6.3. Number of Samples and Approximate Coverage

We compare the number of samples generated by our algorithms, *Tree* and *Tree-warm*, against *Expert*-sequences and *Rand*. *Tree* is our MCTS-based meta-solver (see Algorithm 6.1), including the pruning (see Section 6.5.2) and starting from an empty tree. *Tree-warm* is the same algorithm (also with pruning) but with a warm start using the average reward from previous problems (see Section 6.5.3). *Expert*-sequences are fixed sampling orders that represent different user-defined strategies (see Table 6.2). *Rand* selects computational operations at random, without learning.




(a) *Handover*: Sample rate. (b) *Banana*: Sample rate.




(c) Coverage. Left: *Handover*. Right: *Banana*.

Figure 6.6.: Sampling rates and approximated projected coverage (normalized by *Tree*), averaged over all problems instances.

**Number of samples** We plot the evolution of the sampling rate (including the MCTS overhead for our algorithms) in our two scenarios in Figs. 6.6a and 6.6b. Asymptotically, *Tree* outperforms all *Experts*, which exhibit disparate performance across the two problems. Due to the exploration of possible sampling sequences, initially, the sample rate of *Tree* is lower compared to some of the *Expert*-sequences. However, *Tree-warm* mitigates this issue due to the warm start and improves its sample rate over time. We also note that the *Expert*-sequences are more sensitive to the different problem instances.

The final sample rates achieved by *Tree-warm* in the different instances range between 7 – 15 samples/s in the *Handover* scenario and 5 – 10 samples/s in the *Banana* scenario. Examples of some of the sequences discovered by *Tree* are shown in Table 6.2.

Table 6.2.: Sampling sequences in *Banana* and *Handover* scenarios. Sampling order is left to right, each tuple (.) denotes joint sampling of the subset.

| | | |
|---|---|---|
| *Expert 1* | Handover | $(q_1, q_2, t_q, w_3, w_2, t_w, p)$ |
| Joint Opt. | Banana | $(q_1, q_2, t_q, p, q_3, t'_q, q_4, q_5, a_1)$ |
| *Expert 2* | Handover | $p, t_q, t_w, q_1, q_2, w_2, w_3$ |
| One-by-one | Banana | $p, t_q, a_1, t'_q, q_1, q_2, q_3, q_4, q_5$ |
| *Expert 3* | Handover | $(q_1, t_q), (q_2, p, w_2, t_w), (w_3)$ |
| Sequential Keyframes | Banana | $(q_1, t_q), (q_2, p), (q_3, t'_q), (q_4, a_1), q_5$ |
| *Tree* | Handover | $t_w, t_q, q_1, w_3, (q_2, p), w_2$ |
| Best found Sequence | | $(w_3, t_w), (w_2, p), (q_2, t_q), q_1$ |
| (Examples) | Banana | $(q_1, t_q), (q_2, p), (q_4, q_5, a_1, t'_q), q_3$ |
| | | $(p, q_5, a_1, t'_q), q_3, (q_1, t_q), q_2, q_4$ |

**Approximate coverage** We evaluate the coverage of the solution manifold achieved by our algorithms *Tree* and *Tree-warm*, which should achieve good coverage (i.e., a diverse set of samples) by design. Since the coverage of a nonlinear manifold embedded in a high-dimensional space cannot be evaluated reliably with a relatively low number of samples, we evaluate the *projected coverage* as a proxy measure: We project the full samples onto each variable of the sequence and compare the coverage in each of these subspaces. We discretize these subspaces and count the number of occupied cells, i.e., cells that are occupied by multiple samples are only counted once. Although an approximation, it provides useful and interpretable information about which parts of the solution-manifold are covered.

The results shown in Fig. 6.6c are normalized by the number of occupied cells in *Tree*. While some *Experts* achieve better coverage on a subset of variables, coverage by *Tree* and *Tree-warm* mostly outperform the others. This confirms our hypothesis that, with the correct design and implementation of the sampling operations, maximizing the number of samples is a good heuristic to maximize the coverage.

## 6.7. Limitations

A limitation of our framework is that the algorithm must explore inefficient computations and needs some time to converge to the best sampling order, which could prevent usage when the computational budget is limited. As demonstrated, this limitation is alleviated by warm-starting the algorithm using information from previous runs. The chosen problem setting of learning to generate solutions from online experience might not be relevant when the goal is to generate a single or a few valid solutions as quickly as possible. In this case, supervised learning methods that learn from a dataset of problems solved offline could be a better fit (see Part III of this thesis).

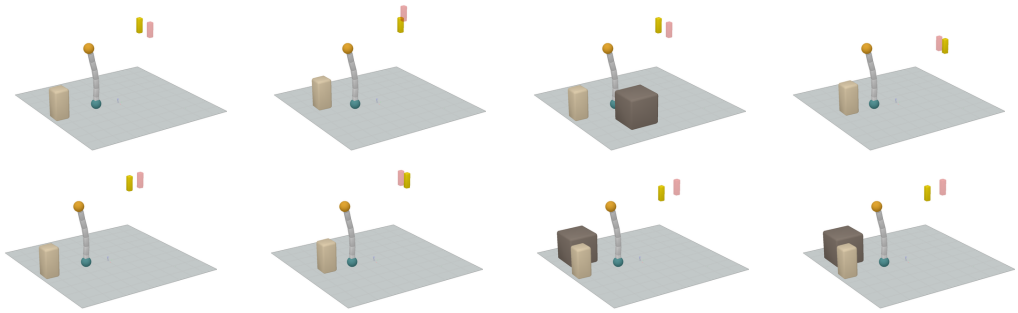


Figure 6.7.: *Banana* Scenario: Initial and goal configurations of the banana (shown in yellow and red) constrain the placement of the box (light brown). The dark brown box is a fixed obstacle.

In this chapter, we have addressed a subproblem of TAMP, in which the task plan is fixed. Integrating our ideas to combine sampling and optimization into a complete TAMP solver is challenging, as both the number of possible sampling sequences and the number of candidate task plans grow exponentially with the number of variables. The full TAMP problem requires an informed balance between the compute effort spent across both different task plans and different compute operations. In Chapter 7, we present a meta-solver to address the complete TAMP problem but, limit the number of allowed sampling operations to keep the search space (and the implementation) manageable.

## 6.8. Conclusion

We have proposed a meta-algorithm to reason about optimal decompositions of factored nonlinear programs in robotic manipulation planning. Our algorithm chooses the computational decisions, i.e., which subset of variables to conditionally sample next, to maximize the number of generated samples in a fixed computational time.

We use the method to efficiently generate a diverse set of samples for keyframes in robotic manipulation, which is an essential component in any solver for TAMP problems or multi-modal motion planning.

Our framework naturally allows us to also include cost factors in the Factored-NLP. However, we neglect cost terms because our approach is tailored to provide a diverse set of feasible samples that can be used in higher-level optimization or motion planning, where the diversity and uniform coverage of samples is an essential ingredient for ensuring completeness.

The same problem setting used in this chapter, generating diverse samples from constraint manifolds, will be revisited in Chapter 8 through the perspective of deep learning. In Chapter 8, deep generative models are trained with a dataset of solutions to similar problems and are used at runtime to compute solutions for new problems faster. In contrast to the meta-solver presented in this chapter, the neural models require that a user defines a fixed sequence of sampling operations beforehand.

# Chapter 7

# Towards Meta-Solvers for Task and Motion Planning

## 7.1. Introduction

In this chapter[1], we present a meta-solver for the comprehensive TAMP problem. Our meta-solver is a search algorithm that combines optimization and sampling computations to solve TAMP problems, guided by heuristics from the discrete task abstraction.

While most TAMP planners use *roughly equivalent* problem formulations, they differ significantly in A) how they interleave and combine search across task and motion, and B) the computational methods used to compute the continuous variables.

In particular, all TAMP planners use a predefined set of fixed computation operations to compute low-level motion, leading to disparate performance depending on the problem at hand. Solvers that rely on sampling partial solutions are inefficient when there are long-term dependencies in the low-level motion. Conversely, solvers that use joint nonlinear optimization are inefficient if the problem is highly decomposable and fail when the optimization problem has infeasible local optima.

In contrast, our meta-solver not only combines search at both the continuous and discrete levels but also reasons about *the best way to solve the continuous level*, deciding which computation operations to perform and in what order. These decisions are crucial to solving problems efficiently because the time spent on motion planning is often the biggest bottleneck for all TAMP solvers. Depending on the methods used to compute the motion, running time can vary from milliseconds to several minutes.

[1] We plan to extend and submit the content of this chapter to a robotics or planning conference, for instance, IROS, ICRA, or ICAPS. This research has been conducted in collaboration with Erez Karpas and Marc Toussaint.

This work is primarily inspired by our previous studies on meta-solvers for computing the keyframes for a fixed manipulation plan (Chapter 6), our novel factored TAMP formulation and solvers (Chapter 5), and the sampling-based TAMP solver PDDLStream [Garrett et al., 2020]. In addition to the formulation and algorithmic tools from Chapter 5, we now include conditional sampling of keyframes and use a similar mechanism to that of PDDLStream to order and prioritize computations.

Our approach represents a first step toward bridging the gap between optimization and sample-based approaches to TAMP, blending and converging towards the best strategy based on the problem at hand.

## 7.2. Related Work

An extensive discussion of related work in Task and Motion Planning is provided in Section 2.4. In contrast to state-of-the-art solvers for TAMP that use either only sampling-based or only optimization-based methods by design, our TAMP meta-solver can adaptively choose between sampling or optimization computations.

PDDLStream has been extended to include optimization operations in the Ph.D. thesis [Garrett, 2021] by merging some sequences of samplers into a larger optimization problem, resulting in a hybrid sampling and optimization algorithm. However, the algorithm does not explicitly decide when to use sampling or optimization to compute the same set of continuous variables. In contrast, our meta-solver explicitly reasons about which method is more efficient for generating the same set of continuous variables.

To merge sampling and optimization, we define a computational state and reason directly about which computational decisions to take next. Our approach is inspired by classical work on meta-reasoning and decision-making [Russell and Wefald, 1991], and its successful application in search, planning, and scheduling, for example, [Seipp et al., 2012, Shperberg et al., 2019, O'Ceallaigh and Ruml, 2015, Zilberstein, 2008, Lieder et al., 2014].

Allowing both sampling and optimization operations in our algorithm defines a search problem with an infinite branching factor to solve the original TAMP problem. Such problems can be addressed by Partial Expansion A* [Felner et al., 2012], Iterative Broadening [Ginsberg and Harvey, 1992], or Iterative Deepening [Korf, 1985]. Instead, we choose to explore a computational space with a simple A*-like algorithm that combines heuristic search with a computational level to widen the tree incrementally, similarly to PDDLStream [Garrett et al., 2020]. Interestingly, different notions of cost, heuristic, and search algorithms in the computational space result in different TAMP meta-solvers.

Recently, *Effort Level Search in Infinite Completion Trees* analyzed a computational decision process with infinite branching in the context of TAMP [Toussaint et al., 2023]. The algo-

rithm decides where to allocate compute time given a fixed set of possible computational decisions to address each component of the TAMP problem (enumerating task plans, computing keyframes with nonlinear optimization, computing paths with sample-based motion planning, and optimization). However, it does not address the trade-off between choosing conditional sampling and optimization when solving TAMP, as we do in our meta-solver.

## 7.3. The Gap Between Sampling and Optimization Approaches

Within the TAMP research community, a central concern is the large number of *slightly* different problem formulations, as every TAMP solver often introduces its own unique formulation. This has made it difficult to create hybrid methods or combine ideas from different TAMP solvers.

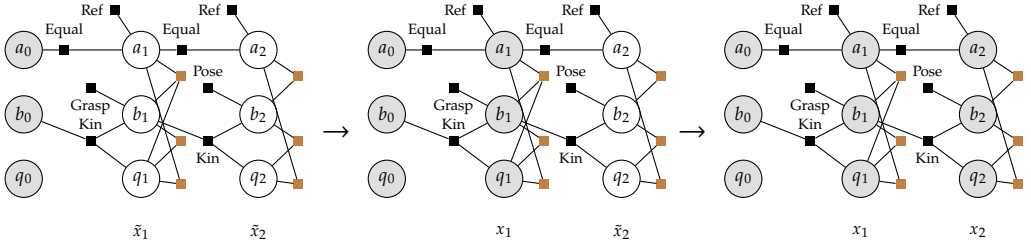
However, when comparing our recent optimization-based TAMP solver (Factored-NLP Planner, see Chapter 5) to PDDLStream, a state-of-the-art sample-based solver, one realizes that both formulations use very similar discrete abstractions and task planning, and represent the motion with equivalent variables and constraints, using constraint networks (in PDDLStream) or Factored-NLPs (in our work). Additionally, they both use discrete task planning to guide the solver towards the goal (when using the *optimistic* algorithms in PDDLStream).

The fundamental difference[2] is in how the values for the constraint networks or factored nonlinear programs are assigned. In PDDLStream, they are computed using a sequence of predefined sampling iterations, while the *Factored-NLP Planner* uses joint optimization. Both methods have advantages and disadvantages, as illustrated in Chapter 6.

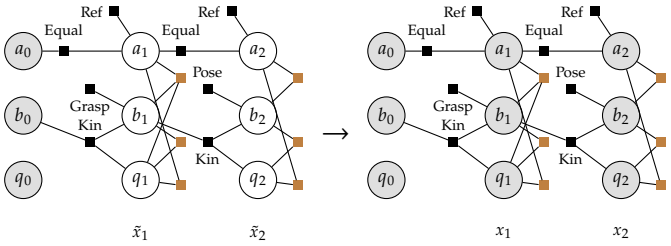
**Space of possible computations** The trade-off between sampling and optimization has been explored in Chapter 6 for a fixed task plan. However, in this chapter, we consider the full TAMP problem, which also entails finding potential task plans and balancing computational time and search effort among different candidate task plans. The focus here is on generating an initial solution to the TAMP problem rapidly, rather than producing multiple solutions for the keyframes of a fixed task plan.

Addressing the full TAMP problem is considerably more challenging than solving for a fixed task plan. Therefore, in this chapter, we confine the space of possible computations. While in Chapter 6 we considered any conceivable sequence of conditional sampling or optimization operations, here we impose the following limitations:

[2] Additional differences, though not analyzed in this chapter, include whether solvers use a conflict-based approach and whether they reuse partial solutions to compute the final solution.




(a) Pick and Place – Sampling.

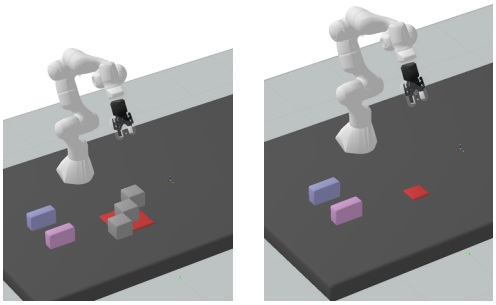



(b) Pick and Place – Optimization.

Figure 7.1.: Sampling and optimization approaches use different computations to generate keyframes for the Pick and Place task plan. From left to right, transitions from white to gray circles indicate the order in which the value of the continuous states has been computed.

- The maximum allowed fine factorization for sampling is at the level of continuous states, instead of individual variables. Hence, the algorithm cannot opt to compute only a relative transformation for the grasp or placement. Instead, it must choose between computing a single full state, e.g., the robot configuration and the grasp, or a sequence of states, e.g., the pick and place configurations.
- Computing continuous states backward in time is prohibited. For example, in a candidate task plan of Pick and Place, the continuous state for the place cannot be computed before the state for the pick (note that this was permitted in Chapter 6). Therefore, the continuous states must be computed either jointly (using joint optimization) or sequentially in a forward manner (first pick, then place).

With these two constraints, the available sequences of computations for a fixed Pick and Place task plan in an environment with one robot $Q$ and two blocks $A, B$ (refer to Fig. 7.2a) are illustrated in Fig. 7.1. In this scenario, only two possible sequences exist because the



(a) (b)

Figure 7.2.: Two representative TAMP problems, where the goal is to place the two colored blocks on the red table. Note that the two problems are identical at the discrete task level, and only the geometric scene is different (because of the size of the table and the presence of unmovable obstacles shown in color gray). Interestingly, (a) is solved more efficiently with sampling-methods, while (b) is solved better with optimization methods.

task plan of Pick and Place comprises just two steps. For a sequence of length three, the number of possible computations is four (i.e., the number of ordered partitions).

## 7.4. The TAMP Computation Tree

We first define a *computational state* and a *computation tree* (i.e., a tree of *computational states*) to model the various computations that can be performed while resolving TAMP problems with optimization or sampling-based approaches. Utilizing this computation tree, we will subsequently define a TAMP meta-solver as a particular search algorithm on the computation tree, and we will demonstrate that different TAMP solvers navigate the tree in very diverse and ingenious ways.

**Discrete-continuous states** Essentially, most TAMP solvers conceptualize the TAMP problem as a hybrid planning problem in the space of discrete-continuous states $(s, x)$. Here, we use our recent formulation, Planning with Nonlinear Transition Constraints (PNTC) (see Chapter 5), as a point of reference. Remember that in PNTC, TAMP is framed as a hybrid planning problem with discrete-continuous states $(s, x)$, where the initial state is $(s_0, x_0)$, and the goals are the states $(s_g, x_g)$ with $g \subseteq s_g$, and where the continuous state $x_g$ satisfies the nonlinear constraints $\phi(x_g; s_g)$.

A node $(s_{k+1}, x_{k+1})$ is a successor of $(s_k, x_k)$ if a discrete action $a_{k+1} \in \mathcal{A}(s_k)$ exists such that $s_{k+1} = \text{succ}(s_k, a_{k+1})$, and the pair of continuous states meet the nonlinear constraints $\phi(x_k, x_{k+1}; s_k, s_{k+1})$ set by the discrete transition $s_k \rightarrow s_{k+1}$. The vector-valued nonlinear constraints $\phi(x_k, x_{k+1}; s_k, s_{k+1})$ amalgamate all the nonlinear constraints defined by the PNTC formulation $\{\phi_b(x_k^{b_0}, x_{k+1}^{b_1}) |\ \phi_b \equiv \Pi(p, \tilde{p}), \forall p \subseteq s_k, \tilde{p} \subseteq s_{k+1}\}$.

Chapter 5 provides an in-depth discussion on the significance of discrete states, discrete actions, continuous states, and nonlinear constraints in the TAMP context. We recall that in PNTC, as one continuous state corresponds to each discrete state, the continuous state now represents the keyframe configuration and the trajectory from the preceding keyframe (instead of a single configuration).

**Computational state** However, the notion of discrete-continuous states $(s, x)$ is not sufficient to represent the different computational operations that can be performed while solving TAMP problems. From an optimization perspective, the continuous state is a *free* variable to be optimized later. From a sampling perspective, the continuous state is *fixed*, computed with a sampling operation. Thus, it cannot model the behavior of hybrid approaches that may combine both *free* continuous states to be optimized later and *fixed* continuous states that have been computed.

For designing a meta-solver, we need to define a *computational state*, a more flexible notion of *state* that represents that the continuous state can be either fixed or a free variable to be optimized later. The computational state has this designation because it models the state of computations: which parts of the problem have been computed and which have not, instead of the state of the world.

Taking the PNTC formulation and notation as a reference (Section 5.3), we define a *computational state* $N$ as a 4-tuple $(s, x, \tilde{X}, \Phi)$ where,

- $s \in \mathcal{S}$ is a discrete state. It represents the current discrete state of the world.
- $x \in \mathcal{X}$ is a fixed continuous state.
- $\tilde{X}$ is a set of free continuous states $\tilde{x}_k$ that have not been computed yet.
- $\Phi$ is a set of nonlinear constraints $\phi_b$ on the free states, which should be satisfied when computing valid values for the free states.

Importantly, in a node $N$, the fixed continuous state $x$ can correspond to the discrete state of a previous step, instead of the discrete state $s$ of the current one (see also Fig. 7.3 that appears later).

Note that here we do not consider the fine factorization of the continuous states $x \in \mathcal{X}$ into a set of variables $\{x^i \in \mathcal{X}^i\}$, and we introduce the notation $\tilde{x}$ to make an explicit distinction between states that have been already assigned, and those which have not.

The initial computational state is $N_0 = (s_0, x_0, \emptyset, \emptyset)$, i.e., a fixed continuous state $x_0$ and a fixed discrete state $s_0$ that represents the initial continuous-discrete state of the world, and no free states or nonlinear constraints.

**Expansions in a computation tree** A computational state $N = (s, x, \tilde{X}, \Phi)$ can be expanded in two different ways:

1. A **discrete expansion** with a discrete action $a$ that is applicable to the current discrete state $s$,

$$\texttt{Discrete\_Expansion}(N, a) \rightarrow N' = (s', x, \tilde{X}', \Phi'), \tag{7.1}$$

   where $s' = \text{succ}(s, a)$, $\tilde{X}' = \tilde{X} \cup \tilde{x}'$, and $\Phi' = \Phi \cup \phi'$.

   Here, $\tilde{x}'$ and $\phi'$ are the new free continuous states and the set of nonlinear constraints that represent the motion corresponding to applying the discrete action $a$ to the discrete state $s$. The constraints can depend on the state from the last time step, which can be either fixed or free.

   Therefore, the discrete expansion changes the discrete state, does not change the fixed continuous state, and introduces new free states or constraints.

   The discrete expansion is deterministic, and there is a finite number of possible expansions for each node.

2. A **numeric expansion** that assigns values to the set of free states $\tilde{X}$, subject to the constraints $\Phi$.

$$\texttt{Numeric\_Expansion}(N) \rightarrow N' = (s, x', \emptyset, \emptyset) \text{ or } \texttt{FAIL}, \tag{7.2}$$

   where $x'$ is the value assignment for the last free state in $\tilde{X}$. The values for all the continuous states in the sequence are stored, but only the last fixed state is required for future expansions.

   The numeric expansion is stochastic, can be executed infinitely, and there could be zero, a finite, or an infinite number of valid possible expansions for each node. If the constraints are not satisfiable, as is often the case in task and motion planning, the expansion fails, resulting in a dead state `FAIL`.

   Usually, nonlinear constraints define a manifold of possible solutions, resulting in an infinite branching factor for the numeric expansion.

   Even if a feasible expansion exists, we cannot guarantee that one attempt to compute values will succeed because the optimization solver might fail to find a solution. However, it could be that by repeating the operation, we can expand the node correctly. This stochastic behavior is implemented using randomized initialization and cost

functions for nonlinear optimization, randomized constraint sampling, or sample-based motion planning.

Why does it make sense to have a notion of free states? First, free states in a computational state will be optimized later, allowing us to consider the joint nonlinear constraints that depend on the variables of the next step (e.g., $\phi(x_1, x_2; s_1, s_2)$). As demonstrated by the success of optimization-based TAMP solvers, it is often beneficial to optimize the continuous variables jointly, rather than sequentially. Moreover, numeric expansions are computationally expensive, and it may be advantageous to delay them as much as possible.

A *computation tree* is a tree whose nodes are computational states, and whose edges are discrete or numeric expansions. In the context of TAMP, we denote the computation tree as the **TAMP Computation Tree**. To solve the original TAMP problem using the TAMP Computation Tree, the goal is to find a valid sequence of computational states $N_0, N_1, \ldots, N_K$ such that $N_K = (s_K, x_K, \emptyset, \emptyset)$ is a goal state, which means reaching a discrete goal state $s_K \supseteq g$, with a continuous state $x_K$ that satisfies the nonlinear constraints $\phi(x_K; s_K) = 0$, and without any free states or remaining constraints.

## 7.5. An Example of a TAMP Computation Tree and Computational States

In Fig. 7.3, we show an instance of a TAMP Computation Tree for a TAMP problem with one robot, $Q$, two movable objects, $A$ and $B$, and a red table.

The initial state $x_0$ is shown in Fig. 7.4a, and the initial discrete state is $s_0$ = [`parent_A = A_init`, `parent_B = B_init`, `gripper_Q = free`]. Given the initial state $N_0 = (s_0, x_0, \{\}, \{\})$, we can apply a discrete action $a_1$ = `pick A with Q from A_init` to obtain a new node $N_1 = (s_1, x_0, \tilde{x}_1, \Phi)$, with $\Phi = \{\phi(x_0, \tilde{x}_1; s_0, s_1)\}$ and $s_1 = \text{succ}(s_0, a_1)$.

Thus, we now have a different discrete state that assumes the discrete action will be applied successfully. However, the continuous state has not been assigned and remains as a free state subject to constraints. Similarly, we can extend the initial state with another valid action $a_1'$ = `pick B with Q from B_init`, resulting in another state $N_2$ with a set of different nonlinear constraints $\Phi'$.

In the computational state $N_1 = (s_1, x_0, \{\tilde{x}_1\}, \{\phi(x_0, \tilde{x}_1; s_0, s_1)\})$, we can decide to compute the free variables. Specifically, we want to assign values to $\tilde{x}_1$ subject to the constraints $\phi(x_0, \tilde{x}_1; s_0, s_1)$. If this operation succeeds, it results in the state $N_3 = (s_1, x_1, \{\}, \{\})$. Now, the last continuous state is $x_1$, and both the set of free states and constraints are empty. If the assignment process fails, a dead child node is created.

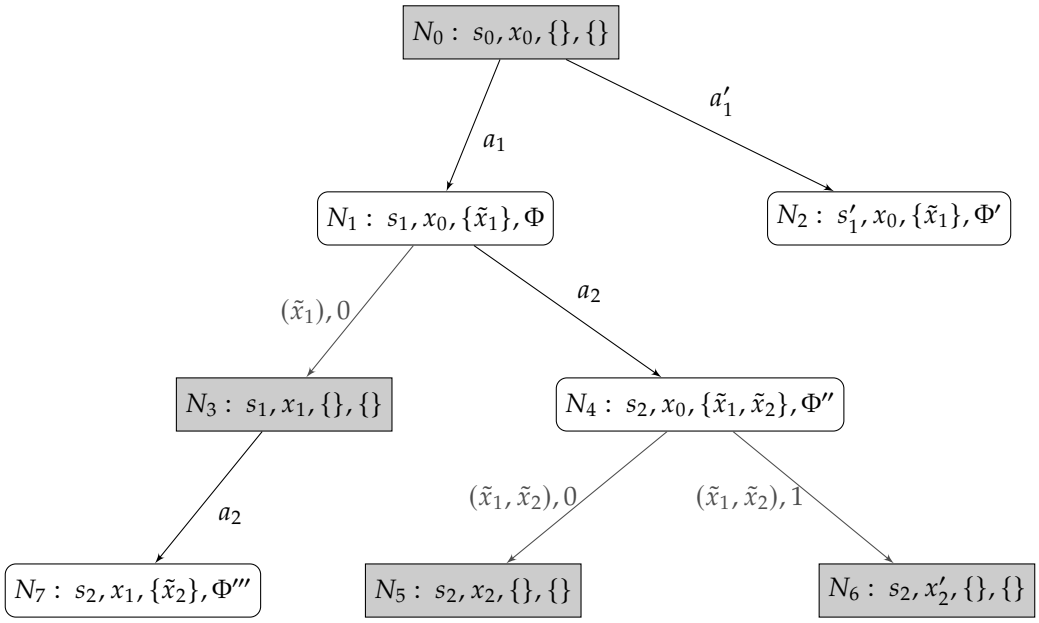
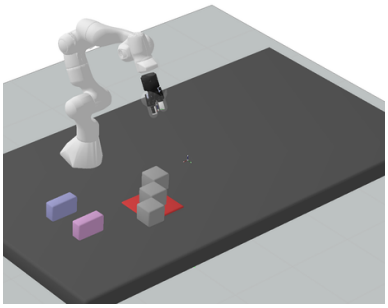


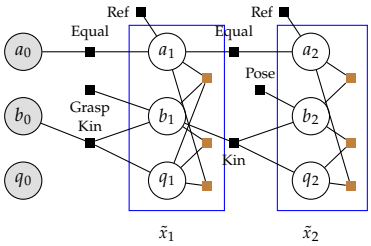

Figure 7.3.: The TAMP Computation Tree. White nodes contain both fixed continuous states and free continuous states subject to constraints . Gray nodes do not contain any free states. The color of the edges indicates the two types of expansions: numeric expansion (in gray) and discrete extension (in black). See the main text for details.

Alternatively, we can perform a discrete expansion of the node $N_1$ with new discrete actions $a_2$ = `place A with Q on red table`, resulting in $N_4 = (s_2, x_0, \{\tilde{x}_1, \tilde{x}_2\}, \Phi'')$, where $\Phi'' = \{\phi(x_0, \tilde{x}_1; s_0, s_1), \phi(\tilde{x}_1, \tilde{x}_2; s_1, s_2)\}$. The discrete state has changed to $s_2$ = [ `parent_A = red table`, `parent_B = B_init`, `gripper_Q = free`], and now we have two free states and additional constraints (Fig. 7.4b).

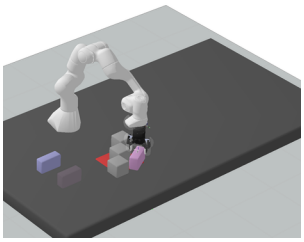
We can decide to expand $N_3$ numerically by computing the two free states jointly, resulting in the state $N_5 = (s_2, x_2, \{\}, \{\})$, with a possible $x_2$ shown in Fig. 7.4c. This operation can be repeated multiple times, resulting in different (if successful) compute states $N_6 = (s_2, x_2', \{\}, \{\})$ with $x_2'$ shown in Fig. 7.4d.



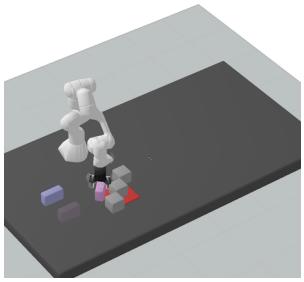
(a) Initial continuous state $x_0$.




(b) Free continuous states ($\tilde{x}_1$, $\tilde{x}_2$) subject to constraints. Each state, represented by a blue rectangle, is factorized into variables $[a, b, q]$. Nonlinear constraints are shown as squares. Gray circles represent fixed variables (in this case, the initial state).



(c) Continuous state $x_2$ for discrete state $s_2$.



(d) A different continuous state $x_2'$ for discrete state $s_2$.

Figure 7.4.: Components of computational states in the TAMP Computation Tree of Fig. 7.3 in the environment shown in Fig. 7.2a.

## 7.6. A Practical Meta-Solver for TAMP

The TAMP Computation Tree provides a framework for designing algorithms for TAMP that automatically select between joint optimization and sampling operations. With this formulation, designing a TAMP meta-solver corresponds to defining a search strategy on the tree that chooses which node to expand next.

Exploration in the TAMP Computation Tree is a daunting task. The state space of the computation tree has an explicitly infinite branching factor in the compute values operation, and the number of possible unique discrete states grows exponentially with the number of objects. In addition to these two inherent TAMP challenges, the state space in the TAMP Computation Tree is even larger, as it includes the state of computation rather than just the original discrete-continuous state space.

In this section, we propose a simple search algorithm that explores the TAMP Computation Tree in an effective manner. The goal of this algorithm is not to outperform state-of-the-art TAMP solvers, but to provide a foundational understanding and intuition of TAMP meta-solvers, which could be revolutionary for designing efficient TAMP solvers in the future.

### 7.6.1. Algorithm

Our TAMP meta-solver is a heuristic search algorithm on the space of computational states. The objective of the search algorithm is to find an optimal solution to the TAMP problem, i.e., the shortest task plan that has a valid motion, using the least compute effort.

**The computational level** To manage the infinite branching factor of the numeric expansion, we associate a computational level with each node, $\mathrm{l}(N) \in \mathbb{N}$, to incrementally enumerate the infinite branching factor, analogous to the level used in PDDLStream. Each time a node with free variables undergoes numeric expansion, we increment its level by one and reintroduce it into the open list. When we create a new node, it inherits the level of its parent node.

Incorporating the level into the node score provides an iterative-widening search algorithm that can repeat numeric expansion operations, each time at a higher cost. This level is the mechanism that allows us to explore new branches of the tree while conducting new expansions in previous nodes, resulting in different continuous states that could potentially lead to a solution.

**TAMP cost and compute cost** We distinguish between two types of costs: compute cost and TAMP cost. The TAMP cost is the cost in the original TAMP problem, assuming each

discrete action has a unit cost. The compute cost refers to the cost of the underlying compute operations.

For the compute cost, we use a straightforward cost model for the operations in the TAMP Computation Tree.

- The compute cost for a discrete expansion is zero.
- The compute cost for a numeric expansion of a node $N = (s, x, \tilde{X}, \Phi)$ equals the number of free states $|\tilde{X}|$ we attempt to compute.

Thus, the compute cost-to-go $c_c(N)$ for node $N = (s, x, \tilde{X}, \Phi)$ is the optimal compute cost required to reach a goal state, i.e., solving the original TAMP problem from $N$. It can be expressed as the sum of two components:

$$c_c(N) = |\tilde{X}| + c_{\text{TAMP}}(s)\,, \tag{7.3}$$

where $|\tilde{X}|$ is the number of free states, and $c_{\text{TAMP}}(s)$, the TAMP cost-to-go, is the length of the shortest task plan from $s$ to a goal state that is also valid for continuous motion.

We can also define a compute cost-to-come $g_c(N)$ as the number of continuous states that have been assigned, starting from the root node. The TAMP cost-to-come $g_{\text{TAMP}}(s)$ is the length of the task plan from the root to the current node.

Evaluating the function $c_{\text{TAMP}}(s)$ would require solving a TAMP problem in itself. However, we can define a lower bound $c_{\text{TASK}}(s) \leq c_{\text{TAMP}}(s)$ that disregards the geometric information and can be efficiently calculated by calling a discrete task planner.

**Score function** To balance the TAMP cost with the compute cost and progressively explore the infinite branching factor of the numeric expansions, we define a score function. The score function $f(N)$ for a node $N = (s, x, \tilde{X}, \Phi)$ is a tuple of two values:

$$f(N) \rightarrow [c_{\text{TASK}}(s) + g_{\text{TAMP}}(s) + l(N), -g_c(N)]\,. \tag{7.4}$$

The best node is selected based on the first value (lower is better), and the second value is used for breaking ties (lower is better, i.e., we prefer a node where we have already invested compute effort). If ties persist, they are broken randomly. These two components prioritize expanding nodes in task plans with potentially few discrete actions, which have not been previously attempted, and that will require less compute effort to achieve (since some states have already been computed).

**What is missing to outperform state-of-the-art TAMP solvers?** In the experiments section, we demonstrate how our simple meta-algorithm can outperform sample-based and

**Algorithm 7.1** The TAMP meta-solver.

**Input:** $N_0 = (s_0, x_0, \{\}, \{\}, l = 0)$
▷ Initial computational state (with computational level $l$)
$L \leftarrow \{N_0\}$ ▷ Open list
**while** $|L| > 0$ **do**
  $N \leftarrow$ `Choose Best`$(L)$ ▷ Node with best score $f(N)$ (Eq. (7.4))
  `Remove` $N$ `from` $L$
  **if** $|N.\tilde{X}| = 0 \;\;\&\;\; g \subseteq N.s$ **then return** $N$ ▷ We have reached the discrete goal, and all continuous states are assigned
  **end if**
  **if** $|N.\tilde{X}| > 0$ **then**
    $N' \leftarrow$ `Numeric_Expansion`$(N)$ ▷ Compute values for free variables (Eq. (7.2))
    **if** $N' \neq$ `FAIL` **then**
      $N'.l \leftarrow N.l$ ▷ Inherit level
      $L \leftarrow L \cup \{N'\}$ ▷ Add new node to open list
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
    **else**
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
      **continue**
    **end if**
  **end if**
  **for** $a \in \mathcal{A}(N.s)$ **do**
    $N' \leftarrow$ `Discrete_Expansion`$(N, a)$ ▷ Expand with discrete action (Eq. (7.1))
    $N'.l \leftarrow N.l$ ▷ Inherit level
    $L \leftarrow L \cup \{N'\}$ ▷ Add node to open list
  **end for**
**end while**

optimization-based TAMP solvers in short-horizon planning problems involving two objects, and one or two robots, but which require finding the right balance between individually sampling states or optimizing them jointly.

To match and surpass the performance of state-of-the-art TAMP solvers on larger TAMP problems, our meta-solver needs two additional ingredients to share information between the different computational states in the tree:

– *Detection and encoding of geometric conflicts* – A fundamental challenge in TAMP is that the PDDL heuristic distance does not include information about geometry and can be

rather uninformative in some contexts. To address this issue, we could incrementally integrate information about geometry back into the discrete task description. In our previous TAMP solvers, we incorporated negative information in the form of conflicts (either prefixes or subsets of infeasible nonlinear constraints) to modify the original discrete planning task. Alternatives include employing learned heuristics based on the solutions of previous similar problems or designing heuristics that, while fast to compute, can incorporate some knowledge about geometry.

– *Reusing computations* – Solving a TAMP problem involves multiple numeric expansions, which can be computationally expensive. To alleviate this, we could reuse computations from previous nodes in new expansions. For instance, if we have already computed a trajectory between two configurations, that computation could be reused in other expansions requiring the same calculation.

Incorporating these two ideas into the meta-solver (Algorithm 7.1) using the tools we developed in Chapter 5, namely conflict detection and creating a database of feasible partial solutions, is a direction for future work. Another open question for future research is how to better utilize the history of all computations, reusing information between sibling nodes, different nodes with the same discrete state, and similar task plans.

## 7.7. Analyzing and Designing TAMP Solvers with the TAMP Computation Tree

The TAMP Computation Tree is a framework that models the different computational operations performed while solving TAMP problems. Crucially, it provides a unified framework encompassing both conditional sampling and joint optimization. In the TAMP Computation Tree, conditional sampling operations correspond to sampling the continuous states one at a time, while joint nonlinear optimization involves assigning values to a sequence of free states collectively.

We propose that TAMP solvers can be conceptualized as sophisticated search strategies for navigating the TAMP Computation Tree. These strategies can share information between different computational states and store the history of all prior decisions and computations.

**Theorem 7.1** (Existence of a Search Algorithm in the TAMP Computation Tree)**.** *If the set of computational operations a TAMP solver utilizes is encompassed by the possible expansions in the TAMP Computation Tree, then a search algorithm exists within the TAMP Computation Tree that can emulate the solver's behavior.*

This theorem presents an existence argument; we cannot offer a constructive proof, meaning we cannot articulate a concise analytical function for selecting which node to expand next. The search algorithm need not be a classical one like depth-first or breadth-first search; it

**Algorithm 7.2** Optimization-based TAMP solver in the TAMP Computation Tree. Changes with respect to the meta-solver highlighted in orange.

**Input:** $N_0 = (s_0, x_0, \{\}, \{\}, l = 0)$
▷ Initial computational state (with computational level $l$)
$L \leftarrow \{N_0\}$ ▷ Open list
**while** $|L| > 0$ **do**
  $N \leftarrow$ `Choose Best`$(L)$ ▷ Node with best score $f(N)$ (Eq. (7.4))
  `Remove` $N$ `from` $L$
  **if** $|N.\tilde{X}| = 0$ & $g \subseteq N.s$ **then return** $N$ ▷ We have reached the discrete goal, and all continuous states are assigned
  **end if**
  **if** $|N.\tilde{X}| > 0$ & $g \subseteq N.s$ **then** ▷ **Optimization-based:** Only compute values for a full candidate task plan
    $N' \leftarrow$ `Numeric_Expansion`$(N)$ ▷ Compute values for free variables (Eq. (7.2))
    **if** $N' \neq$ `FAIL` **then**
      $N'.l \leftarrow N.l$ ▷ Inherit level
      $L \leftarrow L \cup \{N'\}$ ▷ Add new node to open list
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
    **else**
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
      **continue**
    **end if**
  **end if**
  **for** $a \in \mathcal{A}(N.s)$ **do**
    $N' \leftarrow$ `Discrete_Expansion`$(N, a)$ ▷ Expand with discrete action (Eq. (7.1))
    $N'.l \leftarrow N.l$ ▷ Inherit level
    $L \leftarrow L \cup \{N'\}$ ▷ Add node to open list
  **end for**
**end while**

can be a complex system that decides on operations to perform based on all prior actions. This could include identifying geometric conflicts or assessing bounds of infeasibility prior to a numeric expansion.

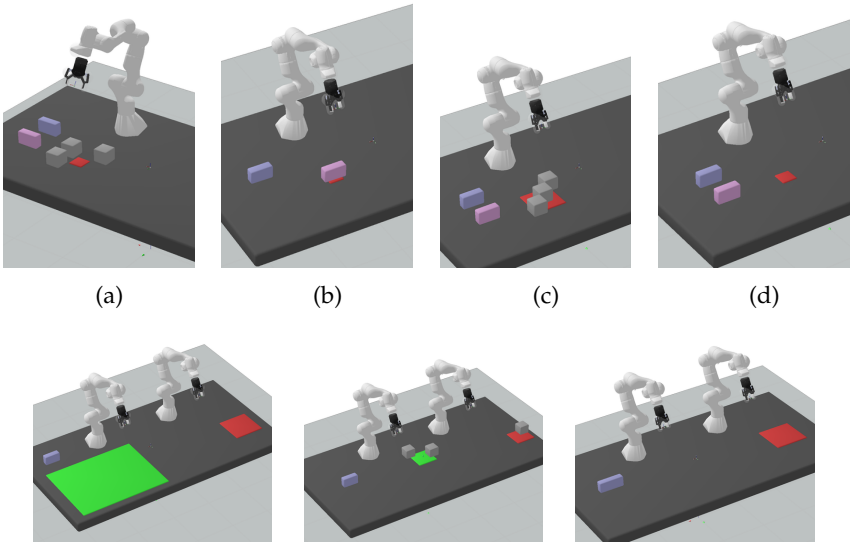
Some advanced TAMP solvers factorize TAMP problems (i.e., the full discrete and continuous state is decomposed). For clarity in this presentation, we have depicted a TAMP Computation Tree in the full configuration space, without explicitly modeling factorization. Incorporating factorization would simply require breaking down the full continuous state into variables, as demonstrated in the Planning with Nonlinear Transition Constraints formulation.

**Algorithm 7.3** Sampling-based TAMP solver in the TAMP Computation Tree. Changes with respect to the meta-solver highlighted in orange.

**Input:** $N_0 = (s_0, x_0, \{\}, \{\}, l = 0)$
▷ Initial computational state (with computational level $l$)
$L \leftarrow \{N_0\}$ ▷ Open list
**while** $|L| > 0$ **do**
  $N \leftarrow$ `Choose Best`$(L)$ ▷ Node with best score $f(N)$ (Eq. (7.4))
  `Remove` $N$ `from` $L$
  **if** $|N.\tilde{X}| = 0$ & $g \subseteq N.s$ **then return** $N$ ▷ We have reached the discrete goal, and all continuous states are assigned
  **end if**
  **if** $|N.\tilde{X}| > 0$ **then**
    $N' \leftarrow$ `Numeric_Expansion`$(N)$ ▷ Compute values for free variables (Eq. (7.2))
    **if** $N' \neq$ `FAIL` **then**
      $N'.l \leftarrow N.l$ ▷ Inherit level
      $L \leftarrow L \cup \{N'\}$ ▷ Add new node to open list
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
    **else**
      $N.l \leftarrow N.l + 1$ ▷ Increase level
      $L \leftarrow L \cup \{N\}$ ▷ Add old node to open list
      **continue**
    **end if**
  **end if**
  **if** $|N.\tilde{X}| = 0$ **then** ▷ **Sampling-Based:** Discrete expansion only in nodes without free variables
    **for** $a \in \mathcal{A}(N.s)$ **do**
      $N' \leftarrow$ `Discrete_Expansion`$(N, a)$ ▷ Expand with discrete action (Eq. (7.1))
      $N'.l \leftarrow N.l$ ▷ Inherit level
      $L \leftarrow L \cup \{N'\}$ ▷ Add node to open list
    **end for**
  **end if**
**end while**

We now illustrate how traditional sample-based or optimization-based TAMP solvers are represented as search algorithms in the TAMP Computation Tree that utilize only a subset of the available operations.

**Optimization-based TAMP solver** Within the TAMP Computation Tree framework, an optimization-based solver applies numeric expansion exclusively to computational states



(a) (b) (c) (d)

(e) (f) (g)

Figure 7.5.: Benchmark TAMP problems.

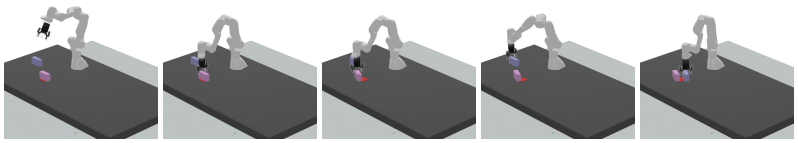
that satisfy the discrete goal—thereby optimizing all continuous states for a candidate plan collectively at the end.

Algorithm 7.2 demonstrates this approach by altering a single line in the TAMP meta-solver, highlighted in orange.

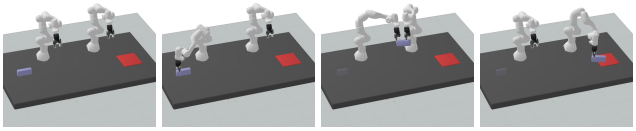
**Sampling-based TAMP solver** In the TAMP Computation Tree framework, a sampling-based solver permits only the discrete expansion of nodes without free variables. This approach limits the number of free states in a compute state to one, ensuring that motion is computed sequentially, step by step. Algorithm 7.3 implements this method by changing one line in the TAMP meta-solver, also highlighted in orange.

## 7.8. Experimental Results

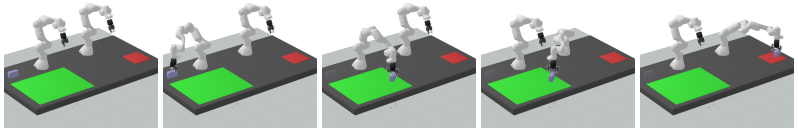
The goal of the experimental evaluation is twofold. First, we aim to present a set of small and illustrative TAMP problems that expose the balance and trade-off between optimization and sampling. Second, we demonstrate that our TAMP meta-solver bridges the gap between optimization and sample-based solvers for TAMP, outperforming them in terms of average computational time across different problems. Importantly, we do not expect our TAMP



(a)



(b)



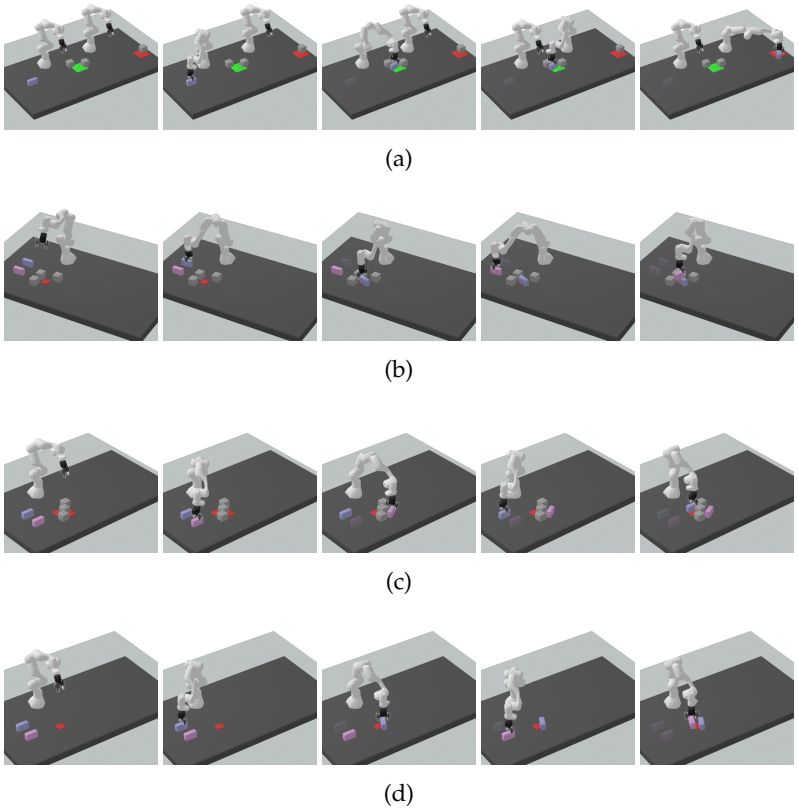
(c)

Figure 7.6.: Example solutions to the TAMP problems in Fig. 7.5 (Part 1).

solver to outperform state-of-the-art TAMP solvers in problems that involve interacting with a large number of objects. Our TAMP meta-solver, implemented as a simple yet effective A*-like search on the tree, cannot yet scale to large settings, which require more effective ways to explore the space of possible computations.

**Benchmark** Our benchmark comprises 7 problems that involve at most two robots and two objects. The maximum length of the task plan required to solve all problems is four. The main challenges are the interdependencies between the different steps of the motion and the constraints imposed by the geometry and obstacles of the environment.

1. *Blocks on a small table* (in short, small) – The task is to move the two blocks to the red table, which requires very precise placement to ensure that the two blocks fit on the table. See the environment in Fig. 7.5d and the example solution in Fig. 7.7d.
2. *Blocks on a cluttered table* (cluttered) – The task is to move the two blocks to the red table. The table is now larger, but there are three obstacles in the middle. See the environment in Fig. 7.5c and the example solution in Fig. 7.7c.
3. *Handover of a small block* (handover) – The goal is to move the object to the red table. Due to reachability constraints, the optimal solution requires a handover, which de-



(a)

(b)

(c)

(d)

Figure 7.7.: Example solutions to the TAMP problems in Fig. 7.5 (Part 2).

mands very precise grasping to avoid collisions between the end-effectors. See the environment in Fig. 7.5g and the example solution in Fig. 7.6b.

4. *Transfer of a block* (transfer) – The task is to move the block to the red table. Now, the block is very small, so the robots cannot perform a handover and should use the large auxiliary table instead. The table is large, and both robots can only reach a small subset of the table. See the environment in Fig. 7.5e and the example solution in Fig. 7.6c.

5. *Transfer of a block with obstacles* (transfer-obs) – The task is to move the block to the red table. Now, the table is cluttered with obstacles, and the auxiliary table is situated in the middle. See the environment in Fig. 7.5f and the example solution in Fig. 7.7a.

6. *Stacking blocks* (stack) – The task is to build a tower on a table with obstacles. See the environment in Fig. 7.5a and the example solution in Fig. 7.7b.

7. *Block on an occupied table* (occupied) – The task is to put a block on a table that is already filled with one block. Therefore, the optimal solution requires moving the obstructing block first. See the environment in Fig. 7.5b and the example solution in Fig. 7.6a.

Note that in the current implementation, we do not compute the continuous trajectory between keyframes; instead, we consider only the keyframe configurations as the continuous state. In tabletop environments, once the sequence of keyframes is computed, the continuous trajectory can be efficiently computed using sample-based motion planning (e.g., RRT) and/or trajectory optimization.

**Algorithms** We evaluate the TAMP meta-solver (Algorithm 7.1) against a representative optimization-based TAMP solver (Algorithm 7.2) and a sampling-based TAMP solver (Algorithm 7.3), implemented as different search strategies in the TAMP Computation Tree. The solvers will be denoted, respectively, by *Meta*, *Opt (Optimization)*, and *Sampling*.

**Evaluation metrics** As a metric, we use the compute time to find a solution. In particular, we consider only the compute time spent in the `Numeric_Expansion` operations, while neglecting the time spent computing the PDDL heuristic and managing the search queue in the TAMP Computation Tree, which is usually one or two orders of magnitude smaller.

Experiments are run 20 times, and we report the first quartile, median, and third quartile. We normalize the compute time by the median of the best-performing algorithm in each problem. In each run, we use a different random seed, which influences the initialization of nonlinear optimization and sampling operations, often resulting in very disparate compute times because numeric expansions fail or compute values that are not valid later to reach the goal.

### 7.8.1. Example Execution of the Three Algorithms

We first show how *Sampling*, *Opt (Optimization)*, and *Meta* explore the TAMP Computation Tree in different ways when solving different problems.

Fig. 7.8 shows the TAMP Computation Tree for the problem *Blocks on a small table*, and Fig. 7.9 shows the TAMP Computation Tree for the problem *Blocks on a cluttered table*, for the three algorithms. The trees correspond to a reference run in our benchmark; because all algorithms are stochastic, the tree might vary between different runs.

**Legend:** We represent computation trees following the example in Fig. 7.3. Gray squares represent compute states without free continuous states. The green square is a fixed continuous state that reaches the goal, i.e., a solution to the TAMP problem. A red square

indicates that the numeric expansion has failed. The top gray square is the initial computational state. Circles are compute states with one or more free continuous states. Blue circles are compute states that reach the discrete goal but contain free continuous states. An arrow from a circle to a square indicates a numeric expansion. The number of continuous states computed in a numeric expansion is equal to the number of circular nodes from the previous last fixed state (previous gray square). An arrow from a square or circle node to a circle node indicates a discrete expansion.

**Observations:** The *optimization* solver only performs numeric expansion on nodes that reach the discrete goal. This is reflected in the tree because only the blue circles (goal candidates) undergo numerical expansion (i.e., have children that are squares). The numeric expansion computes all the free states from the root and the circles jointly.

The *sampling* solver performs numeric expansions on all nodes that have free states and does not allow for consecutive free states. This is illustrated in the tree because a circle node is never a descendant of another circle node.

The *meta* solver combines sampling and optimization, allowing for multiple free states and numeric expansion of intermediate states, resulting in an adaptive combination of circle and square nodes in the tree.

**Performance in these problems:** The two selected problems are representative of the performance of the three algorithms in the benchmark. The initial state and the goal are the same, but the size of the table and the presence of three obstacles significantly impact performance.

*Blocks on a small table* (Fig. 7.5d) – requires joint reasoning to place the blocks precisely on the small table without collisions. The optimization solver only requires two numeric expansions because the placement of the two blocks is optimized jointly (the first expansion failed due to convergence to a poor local optimum). Conversely, the sampling solver requires multiple sampling attempts because most of the partial solutions for the placement of the first block are unsuitable for the placement of the second block. This is illustrated by the number of failed numerical expansions in the last expansion. Eventually, with enough attempts, we sample continuous states that are also compatible with the subsequent steps to reach the goal. The meta-solver initially attempts to generate a solution using sampling, but after a few failed attempts, the search algorithm automatically switches to optimization. The solution is generated by computing the last three states jointly, conditioned on the pick configuration of the first block – which allows for taking into account the tight constraints of the placement of the two blocks.

*Blocks on a cluttered table* (Fig. 7.5c) – does not require joint reasoning, and sampling is more effective because it avoids solving a larger optimization problem with multiple infeasible

| # | Problem | Opt | | | Sampling | | | Meta | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $Q_1$ | $Q_2$ | $Q_3$ | $Q_1$ | $Q_2$ | $Q_3$ | $Q_1$ | $Q_2$ | $Q_3$ |
| 1 | handover | 0.9 | 1.2 | 1.4 | 1.0 | 2.7 | 7.0 | 0.9 | 1.0 | 1.1 |
| 2 | small | 0.7 | 1.0 | 2.3 | 2.9 | 5.8 | 11.3 | 1.6 | 2.4 | 3.3 |
| 3 | cluttered | 2.1 | 3.4 | 8.8 | 0.6 | 1.0 | 1.5 | 0.7 | 2.1 | 2.9 |
| 4 | stack | 1.3 | 1.9 | 2.6 | 0.9 | 1.0 | 1.1 | 0.9 | 1.0 | 1.2 |
| 5 | occupied | 0.7 | 1.0 | 1.5 | 1.6 | 4.6 | 5.9 | 1.7 | 1.9 | 2.2 |
| 6 | transfer | 1.2 | 1.8 | 2.1 | 1.5 | 2.8 | 4.4 | 0.8 | 1.0 | 1.3 |
| 7 | transfer-obs | 5.1 | 5.7 | 9.1 | 0.9 | 1.0 | 1.1 | 2.0 | 2.6 | 2.9 |
| | Summary | 1.7 | 2.3 | 4.0 | 1.3 | 2.7 | 4.6 | 1.2 | 1.7 | 2.1 |

Table 7.1.: Computation time to find a solution in the 7 problems of the benchmark, normalized by the median of the best-performing algorithm in each problem. $Q_1$, $Q_2$, and $Q_3$ are the first quartile, the median, and the third quartile, respectively. Colors red and green highlight the worst and best algorithms for each statistic ($Q_1$, $Q_2$, and $Q_3$) in each problem. The same results are shown graphically in Fig. 7.10.

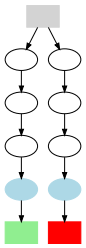
stationary points due to the three obstacles. The optimization solver requires multiple attempts to find a joint solution to the candidate plans (several red squares). Conversely, the sampling solver finds the solution with only two attempts, using less expensive computations because each step is computed individually. Here, the meta-solver behaves similarly to sampling, as the search algorithm always favors easier sampling operations at the beginning.

### 7.8.2. Comparison

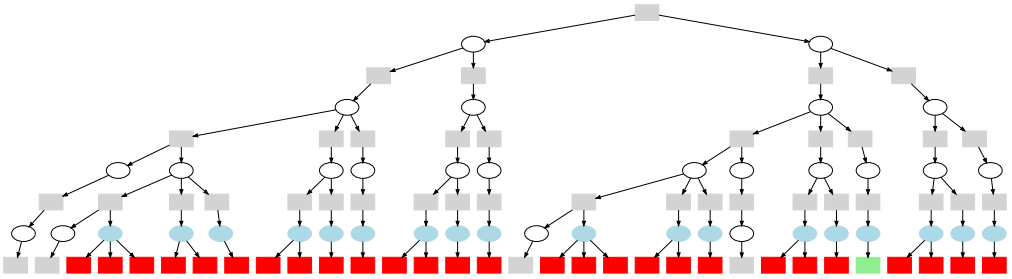
Results are shown in Fig. 7.10 and Table 7.1. From the reported statistics, we consider $Q_3$ as the most important one, as it models the worst-case performance. Minimizing the worst-case performance is critical for real applications, where the goal is to guarantee that an algorithm solves the problem quickly with a very high probability.

*Is sampling or optimization better?* When comparing only the sampling and optimization-based solvers, optimization performs better in 4 out of 7 problems. *Sampling* is better in problems that do not require reasoning about joint constraints and have obstacles (which have a more negative impact on the optimization-based solver): *Blocks on a cluttered table*, *Stacking blocks*, and *Transfer of a block with obstacles*. Optimization is better in problems that require reasoning about joint constraints but do not contain obstacles: *Blocks on a small table*, *Handover of a small block*, *Transfer of a block*, and *Block on an occupied table*.

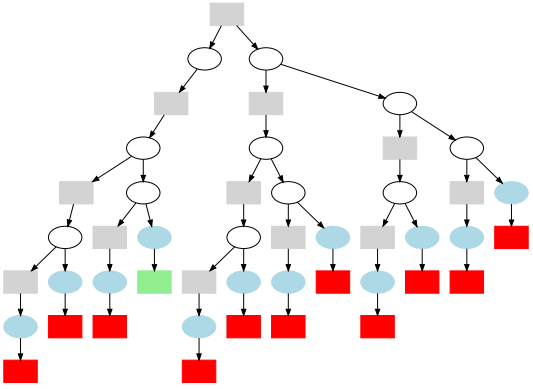
*Analysis of the performance of the meta-solver:* If we analyze the results problem by problem, the meta-solver is never the worst in any problem, confirming that providing the flexibility to choose computational operations is required if we want to solve each problem as effi-



(a) Optimization-based solver (*Opt*).

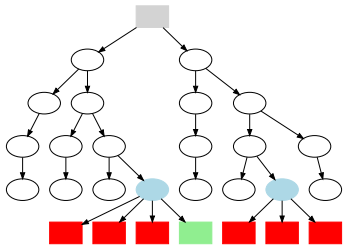


(b) Sampling-based solver (*Sampling*).

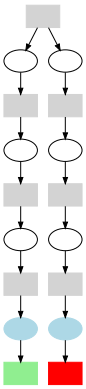


(c) Meta-solver (*Meta*).

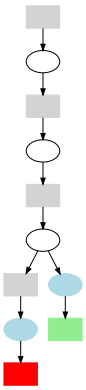
Figure 7.8.: Computation Tree in the problem *Blocks on a small table*.



(a) Optimization-based solver (*Opt*).



(b) Sampling-based solver (*Sampling*).



(c) Meta-solver (*Meta*).

Figure 7.9.: Computation Tree in the problem *Blocks on a cluttered table*.

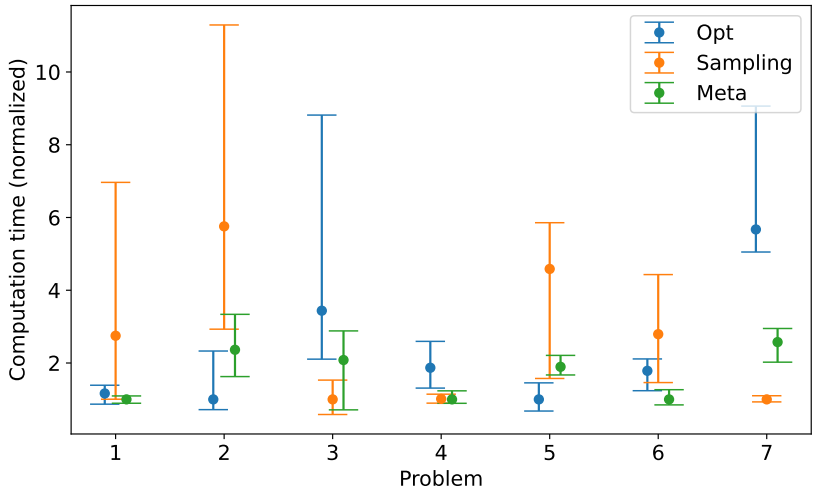



Figure 7.10.: Computation time to find a solution in the 7 problems of the benchmark, normalized by the median of the best-performing algorithm in each problem. The dot is the median, and the brackets show the interval between the first and third quartile (i.e., the 25th and 75th percentiles). The numeric results and the problem labels are shown in Table 7.1.

ciently as possible. This is indeed a very strong result, as it shows that our meta-solver can automatically balance between optimization and sampling and can choose the best strategy for each problem online, simply by trying different operations in a clever manner. Furthermore, the meta-solver outperforms both alternatives in 3 out of 7 problems.

When we summarize the results across problems using the arithmetic mean of $Q_1$, $Q_2$, and $Q_3$, *Meta* outperforms both *Sampling* and *Optimization*. The most significant difference is in the third quartile, i.e., the worst-case performance, where *Meta* is, on average, twice as fast as the baselines.

### 7.8.3. Discussion of Scalability

Preliminary tests on larger problems indicate that the *Meta* algorithm scales less effectively than *Opt* and *Sampling* in TAMP problems with more objects. The branching factor in *Meta* is higher than in *Opt* and *Sampling* because it allows for a more flexible set of operations, and the number of compute states to expand grows more rapidly. In contrast, *Sampling* and *Opt*, by design, restrict the space of possible computations, enabling them to evaluate more candidate task plans and resulting in shorter times to generate solutions in larger problems.

In our meta-solver, the current A*-like search neither shares information between nodes of different subbranches nor similar plans, nor does it utilize the history of computations, resulting in a suboptimal search strategy when the number of objects increases.

Overall, we believe that the TAMP Computation Tree is a promising framework for analyzing and designing TAMP solvers, but it requires a better scoring function for scalability. It remains unclear whether achieving scalability will necessitate additional complex reasoning to select the next node for expansion, or if a simple scoring function will suffice.

## 7.9. Limitations

The framework presented here represents more a foundational study than a practical algorithm superior to existing TAMP solvers. Indeed, our experimental study focuses primarily on TAMP problems requiring short-horizon planning.

A potential extension for scaling to larger problems involves incorporating the conflict extraction algorithm presented in Chapter 5. Upon detecting a conflict, the PDDL domain used to estimate the distance to the goal can be updated using the same reformulation as in Chapter 5. Ensuring that the detected conflict is genuine and not the result of an unlucky solver initialization, as previously discussed in the Limitations section of Chapters 4 and 5, is a significant challenge. Beyond conflicts, sharing more information between nodes belonging to the same high-level task plan, even if they are not in the same branch, is a promising idea. Also, sharing positive information between nodes, such as reusing the results of a numeric expansion in a different node, could be a potential direction for scaling the algorithm.

Moreover, we have limited the space of sampling operations to the level of a continuous state, instead of using a finer factorization in terms of single variables or backward-in-time conditional sampling, as in Chapter 6. Thus, our meta-solver is not yet exploiting the factored structure we have explored throughout this thesis. As future work, we plan to investigate the benefits of allowing this fine discretization in the numeric expansions while balancing the inherent trade-off between implementation complexity and algorithmic performance in our TAMP meta-solver.

## 7.10. Conclusion

In this chapter, we have introduced the TAMP Computation Tree, an abstract TAMP framework that reasons at the level of computational states, instead of adhering to the traditional notion of discrete-continuous states commonly used in prior TAMP solvers. A computational state encompasses both fixed and free variable states subject to nonlinear constraints,

enabling the modeling of the behavior of both optimization-based and sample-based TAMP solvers.

Leveraging this innovative framework, we have proposed a TAMP meta-solver, where the term *meta* implies that this solver can capitalize on both sampling and optimization techniques to tackle TAMP challenges, acting as a *solver of solvers*. The meta-solver is realized as a simple yet efficient heuristic search algorithm in the space of computational states, striking a natural balance between optimization and sampling-based computations.

Through a series of illustrative problems, we have demonstrated that the ideal TAMP solver varies with the specific problem, as no single numerical computation method excels across all scenarios. On a benchmark limited to small problems, our meta-solver adaptively selects the most suitable strategy for each problem, intelligently experimenting with various operations, and, on average, outperforms both optimization and sampling-based TAMP solvers across a broad spectrum of diverse problems.

# Part III.

# Accelerated Task and Motion Planning with Learning Methods

# Chapter 8

# Deep Generative Constraint Sampling

## 8.1. Introduction

Computing keyframe configurations of a manipulation sequence is a core problem in robotics and represents one of the fundamental steps in optimization-based methods for Task and Motion Planning (TAMP). It involves sampling from a constraint manifold, which can be formulated as a nonlinear optimization problem without a cost term. However, in cluttered environments with complex grasp models, these constraints become highly nonlinear, and local optimizers often get trapped in poor local optima, failing to find a feasible solution.

In Chapter 6, we addressed this challenge with a meta-solver that automatically combines joint optimization and constrained sampling to find the best sequence of computations for generating solutions. In a complementary line of research, this chapter[1] presents a method to accelerate joint nonlinear optimization using a dataset of solutions from similar problems.

Our method, Deep Generative Constraint Sampling (DGCS), combines a deep generative model for sampling close to a constraint manifold with nonlinear constrained optimization to project onto the manifold. The generative model, conditioned on the problem instance and taking a scene image as input, is trained with a dataset of solutions and a novel analytic constraint term.

An image-based representation of the problem instance (e.g., [Driess et al., 2020, Xie et al., 2019, Ebert et al., 2017, Paxton et al., 2019]) provides a fixed-size parametrization that encodes obstacles and objects for interaction, eliminating the need to engineer features for the problem instance, and can accommodate a varying number of obstacle objects.

[1] This chapter is based on the publication: Ortiz-Haro, J., Ha, J. S., Driess, D., & Toussaint, M. (2022). Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In the Conference on Robot Learning (pp. 213-223). PMLR.

Generative Adversarial Networks (GANs) [Goodfellow et al., 2014, Arjovsky et al., 2017] and Variational Autoencoders (VAEs) [Kingma and Welling, 2013] have introduced a powerful methodology for training such generative models and have shown potential to represent complex distributions in high-dimensional spaces. This work adopts the training objectives and methods of GANs, aiming to minimize the divergence between the generative and target distributions. With a set of diverse solutions from similar problems as training data, the deep generative model is trained to produce samples close to the current problem's solution manifold. These samples are then used as a warm start for the nonlinear optimizer, resulting in a combined data and model-based approach.

Despite the expressive power of function approximators and recent advancements in deep generative models, they still face limitations in generating samples from highly nonlinear and multimodal distributions. This is critical in our application, as the solution manifolds of manipulation sequences are highly nonlinear and often disconnected.

To address multimodality, accuracy, and sample complexity, we propose an extension of our generative framework to leverage the structure of factored nonlinear programs in robotic sequential manipulation, as detailed in Chapter 3. Specifically, we decompose the sampling of a full solution into a sequence of smaller sampling operations and train a separate conditional generative model for each step.

We evaluate our approach on two problems of robotic sequential manipulation in cluttered environments. Experimental results show that our deep generative model produces diverse and precise samples and outperforms heuristic warm start initialization.

## 8.2. Related Work

**Generative models in robotics** Recently, deep generative models have successfully been applied in robotics, especially in settings where problems are represented directly with images or point clouds. For instance, 6DOF grasps of complex objects can be generated using a VAE [Mousavian et al., 2019, Murali et al., 2020] conditioned on the object's point cloud.

In the context of motion and manipulation planning, generative models have been used to sample informative and collision-free configurations [Ichter et al., 2018, Ha et al., 2018, Sutanto et al., 2020, Kurutach et al., 2018, Kim et al., 2017, Simeonov et al., 2020]. These informed samples, as opposed to traditional uniform sampling, significantly improve the running times of sampling-based algorithms. In this line of research, the general goal is to train a network to directly predict partial or full solutions to a problem. In contrast, we combine learned generative models with model-based local optimization for constraint projection to achieve accurate sampling on high-dimensional nonlinear constraint manifolds.

Figure 8.1.: Sequence of keyframes (pick - handover - place) for the *Handover* problem. Our sampling framework (DGCS) combines a deep generative model that produces approximate samples (top row) conditioned on the scene (first column), with a nonlinear optimizer that projects them onto the constraint manifold (bottom row).

**Warm start in nonlinear optimization** In robotics, nonlinear optimization is used to sample on constraint manifolds and to optimize trajectories, e.g., [Toussaint et al., 2018, Mordatch et al., 2012, Winkler et al., 2018]. Recent data-based approaches aim to learn a warm start to reduce online computation time [Mansard et al., 2018, Merkt et al., 2018]. However, the mapping between problem instances and feasible manifolds is extremely nonlinear and discontinuous, presenting a fundamental challenge [Hauser, 2016, Tang and Hauser, 2018].

In settings where nonlinear programs can be represented with mixed-integer constraints, a successful approach is to learn an assignment for the integer constraints and conduct subsequent optimization [Deits et al., 2019, Bertsimas and Stellato, 2020, Cauligi et al., 2020, Wells et al., 2019, Driess et al., 2020]. Compared to our method, integer formulations use a discriminative model that is easier to train, but their generalization to problems without a clear integer structure is challenging. Recently, [Lembono et al., 2020] applied GANs to inverse kinematics, incorporating forward kinematics into the network training. In contrast, we use general analytical features in the cost term, and our framework can scale to longer sequential manipulation tasks by exploiting factorization.

## 8.3. Sampling on a Constraint Manifold

The problem we address is to generate samples from a manifold $\mathcal{M}_\tau$ parametrized by a fixed-dimensional (though potentially large, e.g., images) problem parameter $\tau \in \mathbb{R}^m$,

$$\mathcal{M}_\tau = \{x \in \mathbb{R}^n \mid h(x;\tau) = 0,\ g(x;\tau) \leq 0\}\ , \tag{8.1}$$

where $h(x;\tau) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{m_{\text{eq}}}$ and $g(x;\tau) : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^{m_{\text{ineq}}}$ are the nonlinear equality and inequality constraints, respectively, and the parameter $\tau \in \mathbb{R}^m$ represents the current problem instance and parametrizes all constraints.

In the context of TAMP, such manifolds arise once we have fixed the task plan (e.g., Eq. (2.14) in the LGP Formulation and Eq. (5.2) in the PNTC formulation). The parameter $\tau$ represents the properties of all objects in the scene, such as initial position, size, shape, and goal position. The variable $x \in \mathbb{R}^n$ concatenates all the degrees of freedom of the configurations of robots and objects in the entire manipulation sequence. The (in)equality constraints $h, g$ describe the objectives of the problem, such as collision avoidance, grasping, kinematic and pose constraints, and are parametrized by the problem instance (e.g., the position of obstacles).

A generative model $x \sim \mathbb{P}(\tau)$ that produces feasible solutions, i.e., samples $x^{(i)} \in \mathcal{M}_\tau$, is built from two components (see Algorithm 8.1): a randomized seed $x_0 \in \mathbb{R}^n$ generation and a constrained optimization algorithm that projects the seed $x_0$ onto the solution manifold via the optimization problem (8.2),

$$\min_{x \in \mathbb{R}^n} \|x - x_0\|^2 \quad \text{s.t.} \quad h(x;\tau) = 0,\ g(x;\tau) \leq 0. \tag{8.2}$$

In our current implementation, we solve (8.2) approximately by running a nonlinear optimizer from the starting point $x_0$ on the feasibility problem:

$$\text{find } x \in \mathbb{R}^n \quad \text{s.t.} \quad h(x;\tau) = 0,\ g(x;\tau) \leq 0, \tag{8.3}$$

where the initialization and internal regularization of the optimizer (Augmented Lagrangian with Gauss-Newton) provide an implicit regularization with respect to $x_0$.

Without using learning methods, the seed $x_0$ is typically sampled from a uniform or Gaussian distribution over the ambient space $\mathbb{R}^n$, resulting in an uninformed guess usually far from the solution manifold.

The projection step is a non-convex optimization problem with no guarantees of producing a feasible sample; especially for complex sequential manipulation problems, the optimization landscape often contains substantial infeasible local optima induced by the nonlinear constraints, making nonlinear projection prone to failure unless the seeds are close to the solution manifold.

**Algorithm 8.1** Sampling on a constraint manifold with deep generative models.

**Input:** Problem parametrization $\tau$, number of trials $N$
$L = \{\}$ ▷ Empty list of samples
**for** $i = 1, 2, \ldots, N$ **do**
  Sample $x_0^{(i)} \sim \mathbb{P}_\theta(\tau)$ ▷ Generate an approximate solution with deep generative models
  $x^{(i)} \leftarrow \Pi(x_0^{(i)})$ ▷ Project $x_0^{(i)}$ onto $\mathcal{M}_\tau$ with nonlinear optimization, warm-started with $x_0^{(i)}$ (Eq. (8.2))
  **if** $x^{(i)}$ is feasible **then**
    Append $x^{(i)}$ to $L$
  **end if**
**end for**
**Output:** $L$ ▷ List of valid samples

To address such difficulties, we train a seeding distribution $\mathbb{P}_\theta(\tau)$ to approximate a reference distribution of feasible solutions $\mathbb{P}_r(\tau)$, so that it can generate diverse samples close to the parametric manifold $\mathcal{M}_\tau$. These samples are then used as a warm start for the optimizer, projecting them onto $\mathcal{M}_\tau$. An example of our framework in the context of manipulation planning is shown in Fig. 8.1.

## 8.4. Training Deep Generative Models to Sample on Constraint Manifolds

Our deep generative model is denoted by $x \sim \mathbb{P}_\theta(\tau)$ with $x = G_\theta(z, \tau), z \sim \mathbb{P}_z$, where $\mathbb{P}_z$ is a multivariate Gaussian distribution, and $G_\theta$ is a neural network parameterized by $\theta$.

In contrast to the standard application of adversarial generative models for image generation, our setting also includes an analytical description of the target distribution's support, namely, the features $\phi(x;\tau) = [h(x;\tau), \max(0, g(x;\tau))]$ that characterize $\mathcal{M}_\tau$ with $\phi(x;\tau) = 0$. We consider a data-free, gradient-based optimization of the analytical constraint violation,

$$\min_{\tau} \mathbb{E} \; \mathop{\mathbb{E}}_{x \sim \mathbb{P}_\theta} \left\| \phi(x;\tau) \right\|^2 . \tag{8.4}$$

However, this approach is extremely ill-posed and can converge to a deterministic mapping $G_\theta(z, \tau) = G_{\theta,\tau}$ for all $z$, where the model disregards the noise $z$ and loses the capacity to generate a diverse distribution.

To enforce sample diversity, we regularize with respect to a reference distribution $\mathbb{P}_r$ that is diverse and has its support on the manifold, satisfying $\mathbb{E}_{x\sim\mathbb{P}_r}\left\|\phi(x;\tau)\right\|^2 = 0$. We formulate the problem as follows:

$$\min_{\tau} \mathbb{E}\ W(\mathbb{P}_\theta(\tau), \mathbb{P}_r(\tau)) + \beta \mathop{\mathbb{E}}_{x\sim\mathbb{P}_\theta}\left\|\phi(x;\tau)\right\|^2 \ , \tag{8.5}$$

where $W$ represents the Wasserstein distance, and $\beta$ belongs to $\mathbb{R}_{>0}$. Although the analytical term does not provide additional information beyond the support of $\mathbb{P}_r$, its contribution is crucial in the context of function approximation and stochastic gradient descent in non-convex optimization, as demonstrated in the experiment section.

### 8.4.1. Wasserstein Distance and Adversarial Formulation

The Wasserstein-1 (Earth Mover's distance) between two probability distributions $\mathbb{P}_r$ and $\mathbb{P}_\theta$ as defined in (8.6) is intuitively the cost of optimally transporting mass from one distribution to the other,

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \inf_{\gamma\in\Pi(\mathbb{P}_r,\mathbb{P}_\theta)} \mathop{\mathbb{E}}_{(x,y)\sim\gamma} \|x - y\| \ , \tag{8.6}$$

where $\Pi(\mathbb{P}_r, \mathbb{P}_\theta)$ denotes the set of all joint distributions with marginals $\mathbb{P}_r$ and $\mathbb{P}_\theta$.

Compared to other distance measures such as Jensen-Shannon Divergence or Total Variation, adversarial generative models employing Wasserstein distances have demonstrated superior practical stability and convergence, and are less susceptible to mode collapse [Arjovsky et al., 2017, Gulrajani et al., 2017].

Furthermore, in our application, the Wasserstein distance provides a meaningful interpretation as it reflects the geometric distance between distributions—a critical factor in enhancing the success rate of subsequent nonlinear optimization.

We minimize the objective function in (8.5) using the Wasserstein GAN formulation [Arjovsky et al., 2017, Gulrajani et al., 2017]. Employing Kantorovich duality, we transform the original definition in (8.6) into a minimax game between the critic network $D$ and the generator $G$, training both concurrently via stochastic gradient descent. Specifically, the minimax problem, incorporating our analytical feature, is:

$$\min_{G}\max_{D} \mathop{\mathbb{E}}_{\tau} \mathop{\mathbb{E}}_{x\sim\mathbb{P}_r} D(x;\tau) - \mathop{\mathbb{E}}_{x\sim\mathbb{P}_\theta} D(x;\tau) - \lambda \mathop{\mathbb{E}}_{\hat{x}\sim\mathbb{P}_{\hat{x}}} (\|\nabla D(\hat{x};\tau)\| - 1)^2 + \beta \mathop{\mathbb{E}}_{x\sim\mathbb{P}_\theta}\left\|\phi(x;\tau)\right\|^2 \ , \tag{8.7}$$

where $\beta, \lambda$ are positive real numbers, and $\hat{x}$ denotes samples interpolated between $\mathbb{P}_r$ and $\mathbb{P}_\theta$. Our reference distribution $\mathbb{P}_r(\tau)$ consists of a discrete set of solution/problem pairs $(x_i, \tau_i)$. These data points are computed offline by resolving a large number of problems with nonlinear optimization or sequential constrained sampling, starting from randomized and uninformed initializations–a labor-intensive process that necessitates multiple attempts.


(a) Factored-NLP. (b) Sampling Network.

Figure 8.2.: Factored-NLP and sampling network for the *Pick and Place* problem. (a) Circles represent variables, and squares represent constraints (see the main text and Chapter 3 for an explanation). Brown squares indicate collision constraints. (b) Arrows indicate the factorization of the joint probability and the sequence of sampling operations.

## 8.5. Structured Generative Model by Exploiting Factorization

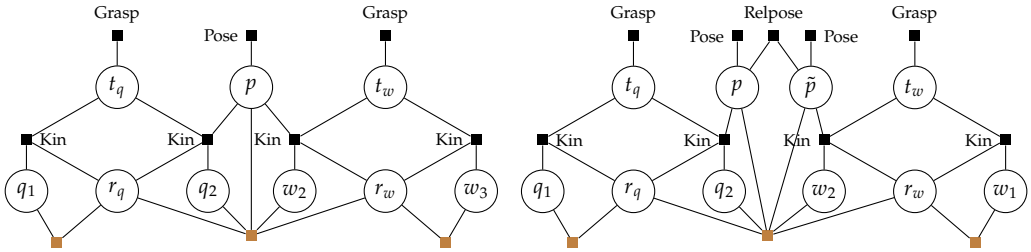
The previous sections treated $x \in \mathbb{R}^n$ as a single vector variable. While the approach for training a single generative model $x = G_\theta(z; \tau), z \sim \mathbb{P}_z$ is powerful, we can further improve scalability to large problems by exploiting a given factorization of $x$ and sequentially decomposing the sampling procedure into a sequence of conditional sampling operations, as in Bayesian networks [Koller and Friedman, 2009].

We now assume a Factored-NLP formulation of the nonlinear manifold (Section 3.1). The vector variable $x \in \mathbb{R}^n$ is factored into a set $X = \{x_1, \ldots, x_N\}$ of $N$ smaller vector variables $x_i \in \mathbb{R}^{n_i}$, and constraints are decomposed into a set of smaller nonlinear constraints $\Phi = \{\phi_1, \ldots, \phi_L\}$, each one depending only on a subset of the variables.

As illustrated in Chapter 3 for the case of TAMP, such factorization naturally arises in many applications, where each variable has some semantic and geometric meaning.

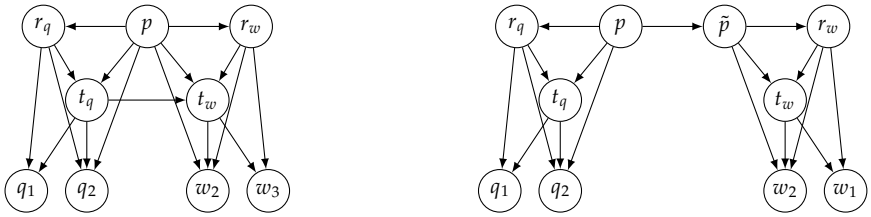
Figs. 8.2 and 8.3 show some examples of such Factored-NLPs in the context of robotic manipulation. Note that the structure of these Factored-NLPs is slightly different from the ones shown in Chapter 3 and Section 5.3. The key differences are: 1) here, all variables that are constrained to be fixed to some specific value are removed from the graph; 2) we use two distinct variables to represent object poses: $t$ for the relative grasp and $p$ for the absolute pose; and 3) consecutive variables that are constrained to be equal are merged into a single variable.

Both formulations of the Factored-NLPs are equivalent, and the differences serve only to expose a slightly different structure that is beneficial for decomposing the problem into a sequence of sampling operations. However, in comparison to Section 5.3, we lose the clear temporal and local structures.




(a) Handover. (b) Assembly.

Figure 8.3.: Factored-NLPs for the *Handover* and *Assembly* problems.




(a) Handover. (b) Assembly.

Figure 8.4.: Sampling networks for the *Assembly* and *Handover* problems.

We quickly introduce the notation used in this chapter. Variables for the configurations of robots $Q$ and $W$ are $q, w \in \mathbb{R}^7$ for the arm configuration, and $r_q, r_w \in SE(2)$ for the pose of the mobile base. Relative transformations between objects and grippers are $t_q, t_w \in SE(3)$, and absolute object positions are $p, \tilde{p} \in SE(3)$. Nonlinear constraints (explained in detail in Chapter 3), are kinematics (*Kin*), grasp, position (*Pose*), relative position (*Relpose*), and collisions (in color brown).

### 8.5.1. Directed Graphical Model and Sequential Sampling

The factored structure directly suggests a factorization of the joint probability distribution $\mathbb{P}(\tau)$ from which we want to sample. As in Bayesian networks, we can sequence the sampling if we decide on an ordering of variables that corresponds to a directed acyclic graph. Instead of learning a single $G_\theta$ to produce a full assignment with $x = G_\theta(z; \tau)$, $z \sim \mathbb{P}_z$, we learn a *conditional* model for each factor using the corresponding marginal distributions of the original data and the subset of corresponding analytical features in the Factored-NLP.

We illustrate the benefits of this factorization with the *Pick and Place* problem, as shown in Fig. 8.2. The joint probability density function $P(p, t, q_1, q_2)$ for this problem is factorized into (we omit conditioning on $\tau$ for clarity):

$$P(p, t, q_1, q_2) = P(p)\, P(t|p)\, P(q_1|t)\, P(q_2|t, p)\ , \tag{8.8}$$

where $p$ is the placement position of the object, $t$ is the relative transformation between the object and the gripper, and $q_1$, $q_2$ are the robot joint configurations at the pick and place keyframes. The factorization exploits conditional independence between $(q_1, p)$ given $t$ and $(q_1, q_2)$ given $t$. We leverage this structure by training a sequence of conditional samplers: $p \sim \mathbb{P}_p$, $t \sim \mathbb{P}_t(p)$, $q_1 \sim \mathbb{P}_{q_1}(t)$, and $q_2 \sim \mathbb{P}_{q_2}(t, p)$, as illustrated in Figs. 8.2b and 8.5. This factorization can be easily extended to longer manipulation sequences, with the ordering $p \rightarrow r \rightarrow t \rightarrow q$ (object pose, robot base, grasp, robot joint values), as shown in Fig. 8.4 for the *Handover* and *Assembly* problems.

As a side note, using the marginal distributions of a jointly consistent dataset is necessary, as only the marginals contain useful information about whether a partial assignment will admit a full solution. For example, when sampling $p \sim \mathbb{P}_p$, we aim to fulfill local constraints (e.g., *Pose*) and ensure that the value will be consistent with the future assignment of other variables. The second type of constraints cannot be evaluated efficiently but is modeled by the marginal distribution of the data.

### 8.5.2. The Advantage of Factorization for Modeling Multimodality

Generating samples from a distribution with disconnected supports using a deep generative model that receives continuous input noise $z \sim \mathbb{P}_z$ requires infinite gradients in $G_\theta$ and can only be done approximately. In these cases, training is unstable and very sensitive to hyperparameters and architecture.

We can model disconnected distributions more effectively by factoring the full joint probability into a sequence of smaller, conditional modules, as confirmed by our experimental results in Section 8.6.3. The sequencing still requires that each module has the ability to produce some degree of multimodality. However, once a module in the sequence receives disconnected input in the form of conditioning, it can successfully produce a disconnected output. As we chain modules with the ability to generate a small amount of disconnected components given continuous input, the number of possible disconnected components of the output grows exponentially with the number of modules in sequence.

Furthermore, from a practical perspective, training smaller modules has proved to be more effective. For instance, we observed that the analytical feature $\|\phi(x; \tau)\|^2$ of the joint problem can provide badly conditioned gradients when evaluated far from the manifold. This issue is alleviated when considering only subsets of constraints and variables. In our

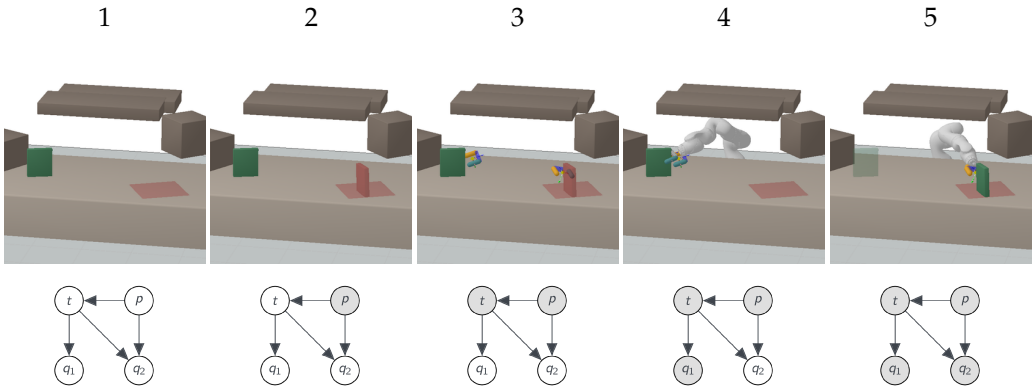



Figure 8.5.: Sequence of learned deep generative models for the *Pick and Place* problem.

preliminary experiments, we also evaluated GAN frameworks that have a mechanism to model disconnected distributions [Chen et al., 2016, Khayatkhoei et al., 2018], but we did not find significant improvements.

## 8.6. Experiments

### 8.6.1. Image-Based Problem Representation

We use an image-based representation of the problem instance $\tau$ that consists of a depth image and masks. Specifically, $\tau = \{d, m_1, m_2, m_3\}$, where $d$ is the depth image, and $m_1, m_2, m_3$ are three masks containing information from the initial object pose, the goal pose or placement region, and obstacles, respectively, as shown in Fig. 8.6. In the factored approach, each generative module receives as input only the relevant masks; for example, the sampler for the robot pick configuration receives a mask of the obstacles and the initial configuration but not the goal pose.

The main strength of the image representation is that it can generalize to different object shapes and a varying number of obstacles and shapes. Moreover, a depth camera is readily available, approximate masks can be computed with image segmentation techniques, and it provides a robust representation of sequential manipulation problems in tabletop scenarios.

### 8.6.2. Scenarios

We consider three different manipulation tasks that involve object manipulation with stable grasps:

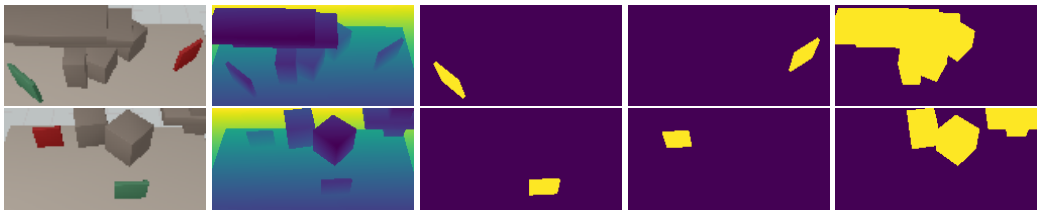


Figure 8.6.: Image-based representation of problem instances in *Assembly* and *Handover*.

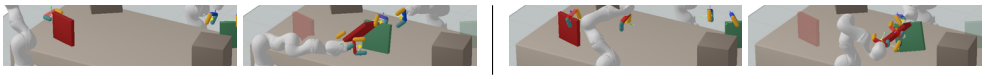


Figure 8.7.: Two samples from the deep generative model in the same instance of the *Assembly* problem are displayed. Each sample is depicted with two keyframes (pick and assembly). Both seeds lead to a feasible solution.

- *Pick and Place*: A robot must pick up an object and place it within a specified rectangular region on the table. Refer to Figs. 8.5 and 8.10 for illustrations.
- *Assembly*: Two mobile robots are required to each pick up an object and join them together. The assembly process is not prescriptive and is characterized as a manifold with constraints on both rotation and position: the objects must align perpendicularly and establish stable contact with predetermined faces of the cubes, forming a 'T' shape. Refer to Figs. 8.7 and 8.11 for illustrations.
- *Handover*: Two mobile robots collaborate to transfer the object from its starting to its target position via a handover. Refer to Figs. 8.1 and 8.12 for illustrations.

These tasks are executed on a cluttered table with obstacles varying in number from three to five. The grasp between the gripper and the object is defined by a two-fingered gripper (e.g., the gripper of the Franka Panda robot), which restricts both position and orientation. Robots and movable objects are required to avoid collisions with each other, as well as with any obstacles and the table itself. The training dataset comprises 4,000 pairs of problem-solution scenarios, computed offline using a user-defined sampling sequence to ensure diversity. Each problem is depicted by a 64x128x4 image (the network's input) and the corresponding environment (used to calculate the analytical error term during training and to conduct nonlinear optimization in the benchmark).

Variability in instances is introduced by differences in the number, position, and size of obstacles, the dimensions and location of the objects, and the goal configuration (for example, see Fig. 8.10). The instances in both the evaluation and training datasets are generated from the same distribution.

| | Seeds | | | Solutions | | |
|---|---|---|---|---|---|---|
| | Coverage | Precision | Error | Coverage | Precision | Success |
| Big NN | 0.81±0.09 | 0.7±0.11 | 8.38±1.82 | 0.58±0.12 | 0.39±0.06 | 0.46 |
| Big NN+analytical | 0.79±0.12 | 0.53±0.14 | 1.21±1.35 | 0.75±0.13 | 0.41±0.13 | 0.43 |
| Structure NN | 0.6±0.08 | 0.62±0.1 | 8.09±1.35 | 0.41±0.05 | 0.44±0.06 | 0.56 |
| Structure NN+analytical | 0.57±0.08 | 0.47±0.13 | 1.46±0.79 | 0.44±0.07 | 0.28±0.08 | 0.78 |

Table 8.1.: Ablation study in the *Pick and Place* scenario.

### 8.6.3. Ablation Study

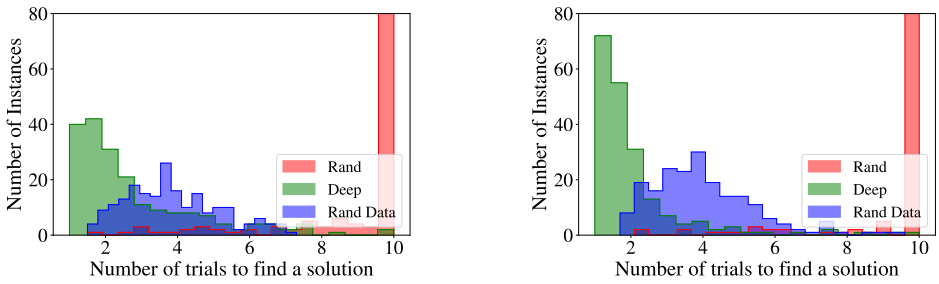
The *Pick and Place* scenario is utilized for an ablation study of the proposed generative model. We assess the contribution of the factored structure (*Big NN* vs *Structured NN*) and the analytical error term (*+analytical*). We evaluate the precision and coverage of seed samples (output from the deep generative models) and solutions (after projection with nonlinear optimization) by generating 4.000 samples for each new instance (30 in total). Results are presented in Table 8.1. We report the following metrics:

- *Error*: the constraint violation $||\phi(x;\tau)||^2$ (unitless, lower is better).
- *Precision*: the average nearest neighbour distance to a reference dataset. It models how close samples are to the real data (lower is better).
- *Coverage*: the average nearest neighbour distance from the reference dataset to the computed samples. It describes how well the learned distribution covers the reference dataset (lower is better).
- *Success rate*: the success rate of the nonlinear optimization, signifying how many samples result in a feasible solution after optimization (higher is better).

When evaluating the seeds (output of the generative model), our two contributions are vital for achieving a minimal constraint violation (analytical term) and optimal coverage (structure). Seeds from the model with both structure and the analytical term are more likely to lead to a solution (success rate). After the projection step, only samples from networks with structure exhibit good coverage.

### 8.6.4. Benchmark: Generative Models in Nonlinear Optimization

The *Assembly* and *Handover* scenarios are used to compare our generative model against two baseline methods for warm-starting (seeding) nonlinear optimizers. We analyze the number of solved problems and the number of necessary optimization runs. Measuring the number of solved nonlinear programs is an indirect way to evaluate coverage and sample quality, as both are fundamental to solving a diverse set of problems with a nonlinear




(a) Handover.

(b) Assembly.

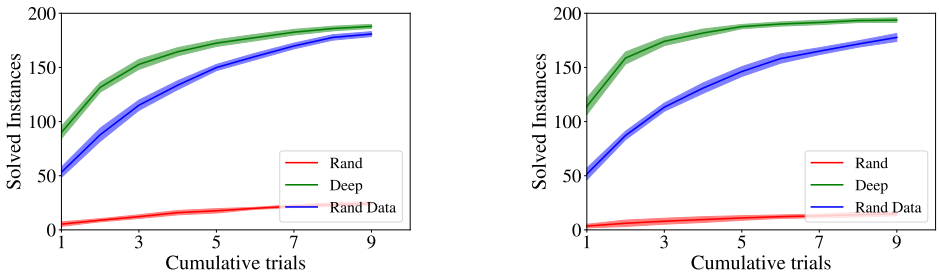
Figure 8.8.: Histogram of the estimated number of trials necessary to solve an instance (lower is better).

optimizer and preventing convergence to infeasible points. We compare our complete model (deep generative model with structure and analytical error term), in short, *Deep*, with the baselines:

- *Rand:* Randomized initial guess around a reference value.
- *Rand Data:* Choosing samples from the training dataset at random. The initial point is a feasible sample for another problem of the same family. This is actually a strong baseline because it provides diverse, informative initial seeds.

We evaluate the generative model (deep generative model + optimization) on 200 problems from the evaluation dataset. The experiments are repeated 10 times, and we report the mean and variance. We first report how many optimization trials (each trial has an independent starting point) are necessary to solve each of the test instances and plot the histogram of the mean value in Fig. 8.8. Unsolved problems are assumed to be solved with 10 trials (maximum number of trials).

In both scenarios, the proposed deep generative model outperforms the baseline warm starts, significantly reducing the number of trials required to solve the instances. First, note that *Rand* can only solve 10% of the problems when using a maximum of 10 trials. This is because the initial guess is not informative, and the nonlinear optimizer often converges to infeasible points. We now compare *Deep* against *Rand Data* and observe that *Deep* provides a 1.5-2x improvement. On average (across problems), from (Rand Data, $3.86 \pm 1.29$) to (Deep, $2.77 \pm 1.79$) in *Handover* and from (Rand Data, $3.99 \pm 1.44$) to (Deep, $2.07 \pm 1.38$) in *Assembly*. To complete the analysis, we also show the cumulative number of problems solved as we increase the number of optimization trials in Fig. 8.9.




(a) Handover. (b) Assembly.

Figure 8.9.: Cumulative number of solved problems (higher is better).

The computational overhead of evaluating the neural network is small (we produce 10 samples in 8 ms with a GPU), while most of the time is spent in optimization runs that converge to infeasible points.

## 8.7. Limitations

This chapter highlights that finding a good warm start for nonlinear optimization is an intricate problem in itself. Although the proposed method clearly outperforms randomized initialization, we observe that one of the baselines, namely warm-starting with a solution from the dataset chosen at random (and thus without considering the conditioning on the scene), also offers competitive results, given its simplicity. Thus, a fundamental question is to understand what constitutes a good warm start for nonlinear optimization and how this might depend on the problem instance. In our work, the underlying assumption is that a sample close to the solution manifold is a good warm start. In practice, we observe that it is often sufficient and beneficial to have diverse samples on different bases of attraction of the optimizer, which might vary based on the problem instance.

A future research direction is to analyze the expressivity and practical performance of different formulations and architectures for deep generative models, comparing explicit generative models (such as GANs and VAEs), energy-based implicit models, and diffusion-based models.

An inherent limitation of our approach is that generative models are only applicable to similar problems with the same number of variables and constraints. Specifically, for each class of problems evaluated in the experiments–*Pick and Place*, *Handover*, and *Assembly*–we have generated a different dataset of solutions and trained different models.

In contrast, in the following Chapter 9, we use the structure of the Factored-NLP to provide generalization across different problem classes (i.e., different types and numbers of

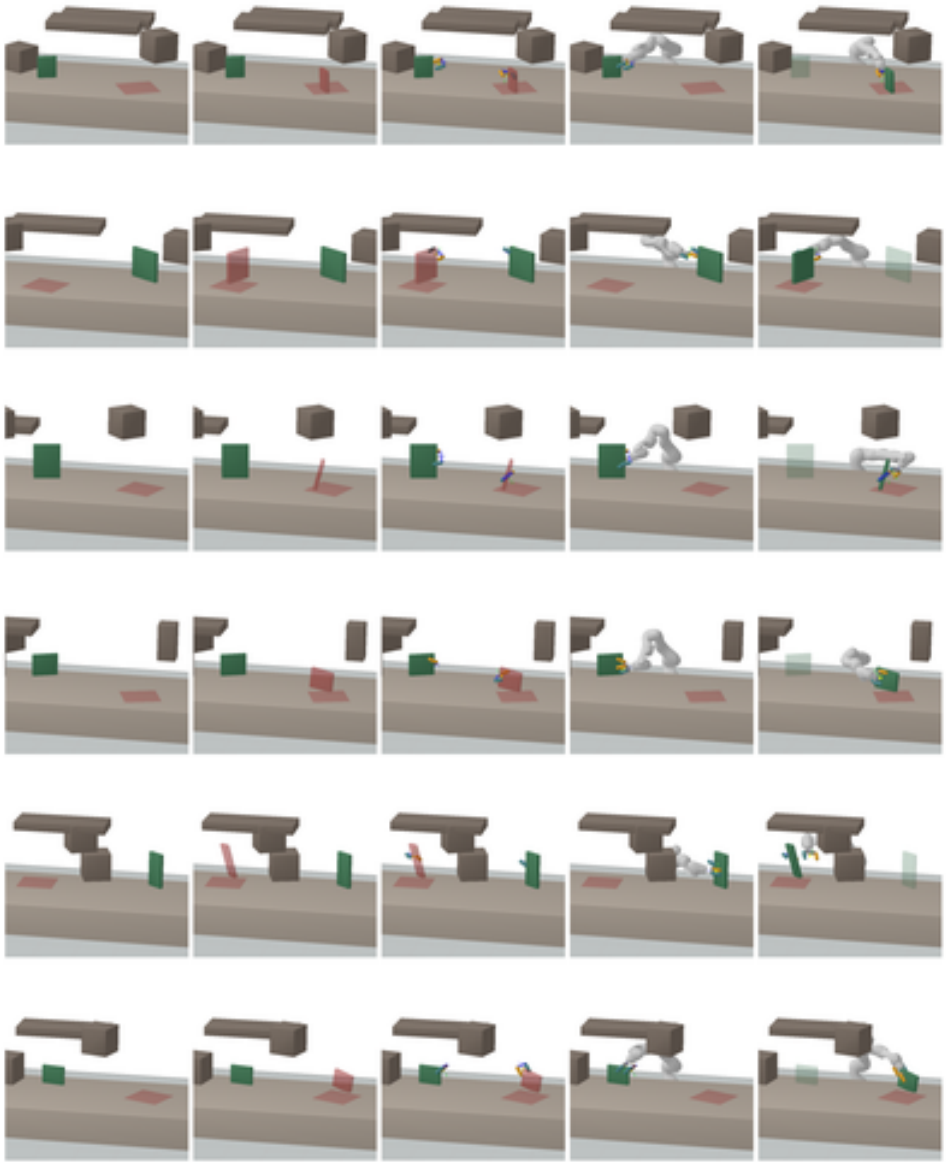


Figure 8.10.: Sequence of sampling operations (output of the deep generative model) in *Pick and Place* across six different instances. Variables are sampled in the following order: object pose on the table, grasp, robot pick configuration, and robot place configuration.

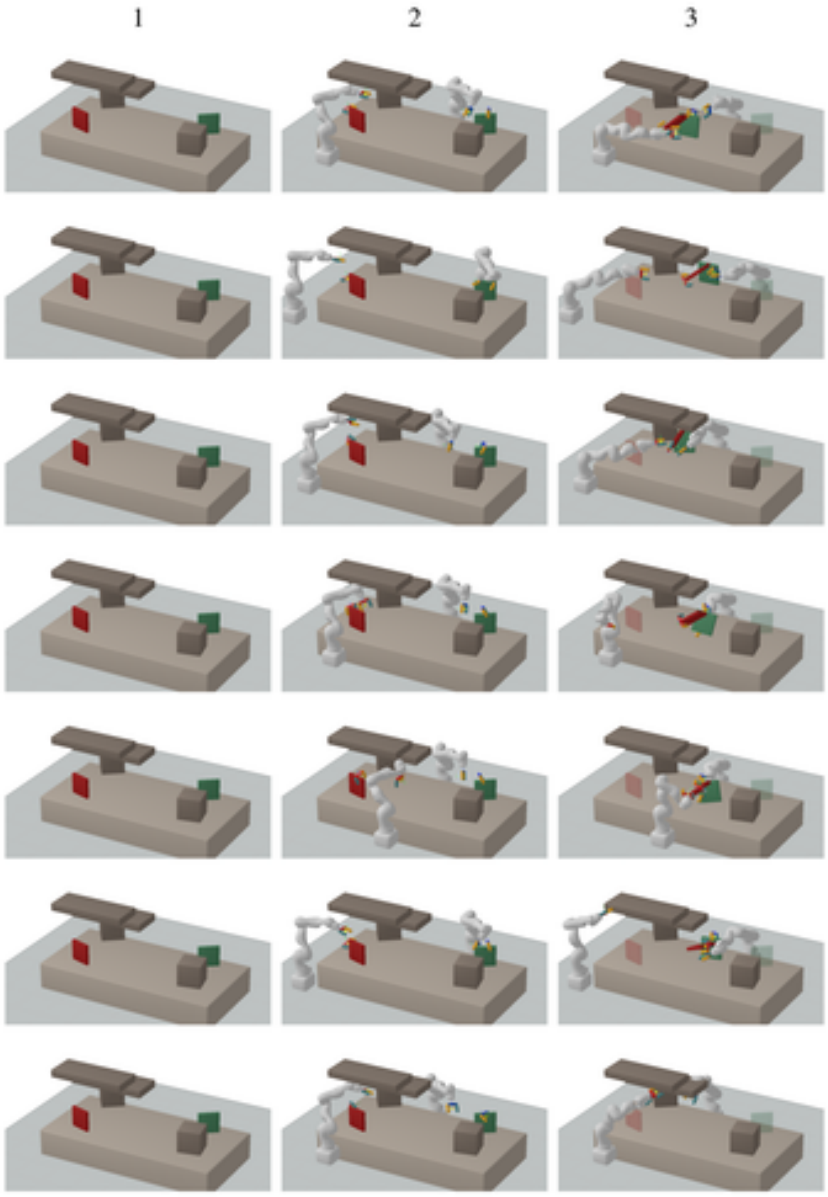



Figure 8.11.: Seven different samples (output of the deep generative model) within the same instance of the *Assembly* problem. Each row represents a different sample. Column 1 shows the problem instance; column 2, the pick keyframe; and column 3, the assembly keyframe.

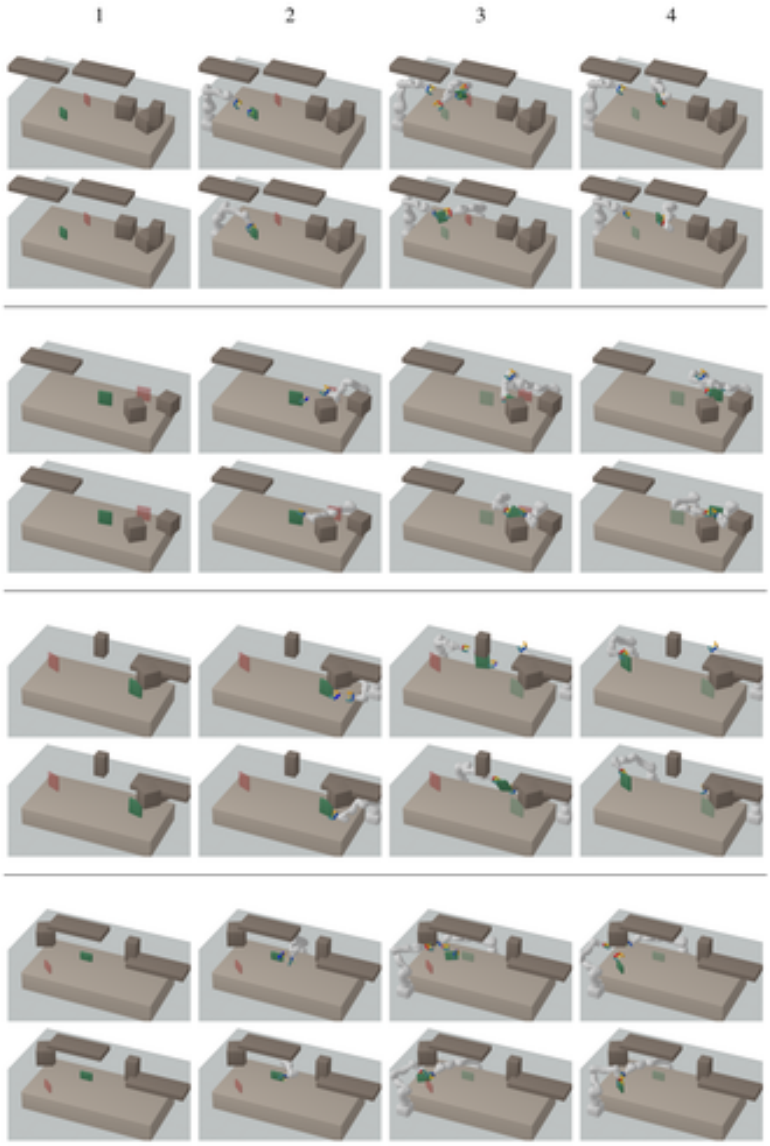



Figure 8.12.: Four pairs of initial guesses (deep generative model outputs) and solutions (post-optimization) across different instances of the *Handover* problem. In each pair of rows, the top row displays the approximate sample, and the bottom row shows the optimized solution. The first column presents the scene, while columns 2, 3, and 4 display the pick, handover, and place keyframes, respectively.

constraints and variables) by sharing and combining small modules, resulting in a single universal model for different task plans. The neural architectures and applications differ, as we predict subsets of infeasible nonlinear constraints instead of generating solutions. However, we see great potential in adapting these insights back to the generative framework.

## 8.8. Conclusion

In this work, we propose Deep Generative Constraint Sampling (DGCS), a novel approach to sampling from a constraint manifold to address challenges in robotic sequential manipulation. Our framework combines a deep generative sampling model, conditioned on an image-based representation of the problem, with a nonlinear optimizer to project samples onto the manifold. Additionally, we extend the approach to exploit a given factorization of the problem by training a sequence of conditional generative models rather than a single joint generator. Our empirical results confirm that the trained generative models outperform heuristic warm start strategies. Moreover, incorporating analytic constraints into the training of the generative model, as well as exploiting the factorization of a given problem, significantly enhances the efficiency, diversity, and precision of the sampling approach.

Our current framework integrates generative sampling using a neural network with subsequent projection through constrained optimization. A promising future direction involves exploring the possibility of embedding the optimization algorithm as the last layer of the generative model, while still maintaining good coverage and multimodality.

In this chapter, we fix the sequence of sampling operations by design when transforming the Factored-NLP into a directed sampling network. As discussed in Chapter 6, the choice of the sampling operations sequence can significantly impact performance. However, sampling operations with deep learning are conditioned on the problem scene, providing flexibility to capture interdependencies in the manipulation sequences and mitigating the drawbacks of uninformed uniform sampling.

Looking forward, we aim to develop deep generative models that directly work with the Factored-NLP, eliminating the need for manually designed directed graphical models, as this would allow for the automatic learning of the most effective decompositions.

# Chapter 9

# Learning Feasibility of Factored Nonlinear Programs

## 9.1. Introduction

When a Factored Nonlinear Program (Factored-NLP) is over-constrained or infeasible, a fundamental challenge is to extract a minimal conflict – a minimal subset of constraints that can never be fulfilled. Traditional approaches require solving several nonlinear programs, incrementally adding and removing constraints, and are thus computationally expensive.

In this chapter[1], we propose a graph neural architecture that predicts which subsets of variables and constraints are infeasible in a Factored-NLP. The model is trained with a dataset of labeled subgraphs from Factored-NLPs and can make useful predictions on larger problems than those seen during training.

As an application, we evaluate our method in robotic sequential manipulation and integrate this model into our novel TAMP solver presented in Chapter 5. The objective is to quickly determine which constraints fail in the trajectory optimization problems of candidate task plans, which is one of the computational bottlenecks of our solver.

Beyond factored nonlinear programs in TAMP, our framework is applicable for detecting infeasibility in constraint satisfaction problems, combinatorial optimization, and Boolean satisfaction (SAT), which have broad applications in robotics, planning, and scheduling.

[1]This chapter is based on the publication: Ortiz-Haro, J., Ha, J. S., Driess, D., Karpas, E., and Toussaint, M. (2023). Learning Feasibility of Factored Nonlinear Programs in Robotic Manipulation Planning. IEEE International Conference on Robotics and Automation (ICRA) (pp. 3729-3735).

The foundation of our framework is built on the factored structure of manipulation planning problems, which has been presented in Chapter 3, and further developed and analyzed with our new factored TAMP formulation in Chapter 5.

An overview of our approach is shown in Fig. 9.1. The input to our model is directly the graph representation of the Factored-NLP, including semantic information about variables and constraints (e.g., a class label), and a continuous feature for each variable that encodes geometric information about the scene. Finding the minimal infeasible subgraph (i.e., a subset of variables and constraints of the Factored-NLP) is cast as a graph node classification problem, and the predicted infeasible subsets are extracted with a connected component analysis.

By leveraging the factored structure, our model is able to predict infeasibility in longer manipulation sequences involving more objects and robots, as well as different geometric environments – a broader generalization than our deep generative models presented in Chapter 8, which were limited to a fixed high-level task plan.

Our experiments show that the model accelerates general algorithms for conflict extraction by a factor of 50, and our previous heuristic algorithm for conflict detection in TAMP by a factor of 4.

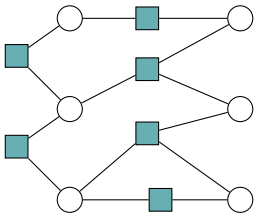
## 9.2. Related Work

**Minimal infeasible subsets of constraints** In the discrete SAT and CSP literature, a minimal infeasible subset of constraints (also called a Minimal Unsatisfiable Subset of Constraints or a Minimal Unsatisfiable Core) is usually computed by solving a sequence of SAT and MAX-SAT problems [Liffiton and Sakallah, 2008, Marques-Silva et al., 2021, Hemery et al., 2006].

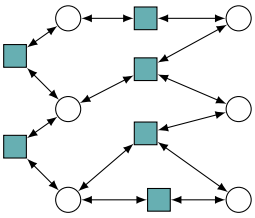
In continuous domains, a minimal infeasible subset can be found by solving a sequence of feasibility problems, adding and removing constraints, with linear complexity in the number of constraints [Amaldi et al., 1999]. This search can be accelerated with a divide and conquer strategy, with logarithmic complexity [Junker, 2004]. In convex and nonlinear optimization, we can find approximate minimal subsets by solving one optimization problem with slack variables [Shoukry et al., 2018].

In contrast, our method uses learning to directly predict minimal infeasible subsets of variables and constraints and can be combined with these previous approaches to reduce computational time.

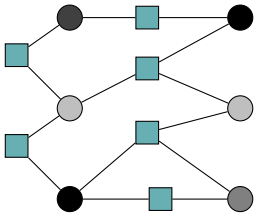
**Graph Neural Networks in combinatorial optimization** We use Graph Neural Networks (GNN) [Kipf and Welling, 2016, Battaglia et al., 2018, Ma and Tang, 2021] for learning in



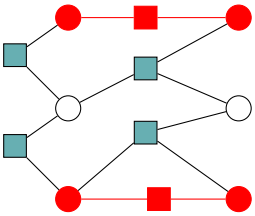
(a) Input Factored-NLP.



(b) Neural message passing.



(c) Output variable scores.



(d) Infeasible subgraphs.

Figure 9.1.: Overview of our approach to detecting minimal infeasible subgraphs in a Factored-NLP. *(a)* The input of the model is the graph representation of the Factored-NLP. Circles represent variables, and squares represent constraints. *(b)* We perform several iterations of neural message passing using the structure of the Factored-NLP. *(c)* The network outputs the probability that a variable belongs to a minimal infeasible subgraph. *(d)* We extract several minimal infeasible subgraphs using a connected component analysis.

graph-structured data. Different message passing and convolutions have been proposed, e.g., [Gilmer et al., 2017, Veličković et al., 2017]. Our architecture, targeted toward inference in factored nonlinear programs, is inspired by previous works that approximate belief propagation in factor graphs [Zhang et al., 2020, Satorras and Welling, 2020, Kuck et al., 2020].

Recently, GNN models have been applied to solve NP-hard problems [Schuetz et al., 2021], Boolean Satisfaction [Selsam et al., 2018], Max cut [Yao et al., 2019], constraint satisfaction [Toenshoff et al., 2021], and discrete planning [Shen et al., 2020, Rivlin et al., 2020, Nir et al., 2021]. Compared to state-of-the-art solvers, learned models achieve competitive solution times and scalability but are outperformed in reliability and accuracy. To our knowledge, this is the first work to use a GNN model to predict minimal infeasible subsets of constraints in a continuous domain.

**Graph Neural Networks in manipulation planning** In manipulation planning, Graph Neural Networks are a popular architecture to represent the relations between movable objects because they provide a strong relational bias and a natural generalization to include additional objects in the scene.

For example, they have been used as problem encodings to learn policies for robotic assembly [Funk et al., 2022, Ghasemipour et al., 2022] and manipulation planning [Li et al., 2020], to learn object importance and guide task and motion planning [Silver et al., 2021], and to learn dynamical models and interactions between objects [Driess et al., 2022], [Paus et al., 2020]. Previous works often use task-specific, object-centric representations, where the vertices of the graph represent the objects, and the task is encoded in the initial feature vector of each variable. Alternatively, our model performs message passing using the structure of the nonlinear program of the manipulation sequence, achieving better generalization to different task plans that fulfill different goals.

## 9.3. Formulation

### 9.3.1. Minimal Infeasible Subgraph in a Factored-NLP

Given an infeasible or over-constrained Factored-NLP $G = (X_G \cup \Phi_G, E_G)$ with variables $X_G$ and constraints $\Phi_G$ (refer to Eqs. (3.1) and (3.3)), our intention is to identify a minimal infeasible subgraph, i.e., a subset of variables and constraints that are jointly infeasible and cannot be reduced further.

To define it formally, a minimal infeasible subgraph $M = (X_M \cup \Phi_M, E_M)$ of a Factored-NLP $G = (X_G \cup \Phi_G, E_G)$, is a subset of variables $X_M \subseteq X_G$ and constraints $\Phi_M \subseteq \Phi_G$ that is

infeasible; yet, any proper subset of it is feasible:

$$M \subseteq G,\ \mathcal{F}(M) = 0,\ \mathcal{F}(M') = 1,\ \forall M' \subset M, \tag{9.1}$$

where $\mathcal{F}(M)$ denotes the feasibility of the Factored-NLP (see Eq. (3.4)), holding the value 1 if it is feasible and 0 otherwise.

In this chapter, we consider only minimal subgraphs in the form of *variable-induced* subgraphs because they enable a more compact representation. Given a graph $G$ and a subset of variables $X' \subseteq X_G$, a *variable-induced* subgraph $M = G[X'] = (X' \cup \Phi', E')$, where $\Phi' = \{\phi \in \Phi_G \mid \text{Neigh}_G(\phi) \subseteq X'\}$, is the subgraph spanned by the variables $X'$. Intuitively, $G[X']$ contains the variables $X'$ and all the constraints that can be evaluated with these variables. Our approach can be adapted to predict general subgraphs if required, by modifying the proposed variable classification to constraint classification in Section 9.3.2.

A Factored-NLP can contain multiple infeasible subgraphs, and a variable $x_i \in X_G$ can belong to multiple infeasible subgraphs. Recall that a minimal infeasible subgraph is connected, and a supergraph $\tilde{M} \supseteq M$ of an infeasible subgraph $M$ is also infeasible.

### 9.3.2. Minimal Infeasible Subgraph as Variable Classification

Let $\Omega_G = \{M_r \mid M_r \subseteq G \text{ minimal infeasible}\}$ be the set of minimal infeasible subgraphs of a Factored-NLP $G$. Instead of learning the mapping $\omega : G \mapsto \Omega_G$ directly, we propose to learn an over-approximation $\tilde{\omega}$ that can efficiently be framed as binary variable classification.

We first introduce the *variable-feasibility* function $\psi(x_i; G)$ that assigns a label $y_i \in \{0, 1\}$ to each variable $x_i \in X_G$: $y_i = 0$ if $x_i$ belongs to some infeasible subgraph and $y_i = 1$ otherwise. Given such a labeled graph, we can recover the infeasible subgraphs approximately by computing the connected components on the subgraph induced by the variables labeled 0, i.e., $G[\{x_i \in X_G \mid y_i = 0\}]$. Thus, we define the approximate mapping as

$$\tilde{\omega}(G) = \text{CCA}\left(G[\{x_i \in X_G \mid y_i = 0\}]\right), \tag{9.2}$$

where CCA denotes a connected component analysis.

The approximate mapping $\tilde{\omega}$ is exact, i.e., $\tilde{\omega} = \omega$, if the infeasible subgraphs are disconnected. If two or more infeasible subgraphs are connected, it returns their union as a minimal infeasible subgraph, i.e., $\cup\tilde{\omega} = \cup\omega$, which over-approximates the size of the original minimal infeasible subgraph. Our neural model will be trained to emulate the labels of the *variable-feasibility* function $\psi$.

We emphasize that learning the approximate function $\tilde{\omega}$ is not a real limitation. First, because the prediction will be integrated into an algorithm that can further reduce the size of the infeasible subgraph, if not already minimal, as shown later in Section 9.3.4. Second,

because finding small infeasible subgraphs, as opposed to strictly minimal, is already useful in many applications. Finally, note that $\omega$ could be transformed into a multiclass variable classification $f(x_i; G) = r_i \subseteq \{1, \ldots, R\}$, where each variable may belong to multiple classes – but this would require a complex and potentially intractable permutation invariant formulation.

### 9.3.3. GNN with the Structure of a Factored-NLP

A fundamental idea of our method is to use the structure of the Factored-NLP for message passing with Graph Neural Networks (GNN) to learn the *variable-feasibility* $\psi(x_i; G)$.

In neural message passing, each variable vertex $x_i \in X_G$ has a feature vector $z_i \in \mathbb{R}^{n_z}$ that is updated with the incoming messages of the neighboring constraints. Each $z_i$ is initialized with $z_i^0$ to encode semantic and continuous information of the variable $x_i$ (an example of how to initialize the features in manipulation planning is shown in Section 9.4.2). The update rule follows a two-step process: first, each constraint computes and sends back a message to each neighboring variable, which depends on the current features of all the neighboring variables. Second, each variable aggregates the information of the incoming messages from the constraints and updates its feature vector,

$$[\oplus \mu_{a \to i}]_{i \in N(a)} = \text{Message}_a([\oplus z_i]_{i \in N(a)}), \tag{9.3a}$$

$$z_i' = \text{Update}(\text{AGG}_{a \in N(i)}\ \mu_{a \to i}, z_i), \tag{9.3b}$$

where $\mu_{a \to i} \in \mathbb{R}^{n_\mu}$ is the message from constraint $a$ to variable $i$. The operator $[\oplus \bullet]_i$ denotes concatenation. $N(a) = \text{Neigh}_G(\phi_a)$ is the ordered set of variables connected to the constraint $\phi_a$. Conversely, $N(i) = \text{Neigh}_G(x_i)$ is the set of constraints connected to variable $x_i$. AGG is an aggregation function, e.g., max, sum, mean, or weighted average. We use max (element-wise) in our implementation. A graphical representation is shown in Fig. 9.2.

`Update` and $\texttt{Message}_a$ are small MLPs (Multilayer Perceptron) with learnable parameters. As the nonlinear constraints in the Factored-NLP are not permutation invariant or symmetric, the features $z_i$ must be concatenated in a predefined order $N(a)$ when evaluating $\texttt{Message}_a$. The function `Update` is shared by all vertices (which generalizes to Factored-NLPs with additional variables). The function $\texttt{Message}_a$ is shared between different constraints of the Factored-NLP that represent the same mathematical function, i.e., $\texttt{Message}_a = \texttt{Message}_b$ iff $\phi_a(x) = \phi_b(x) \forall x$ (which generalizes to Factored-NLPs with additional constraints). For example, in manipulation planning, all constraints that model collisions between objects will share the same `Message` MLP.

The message passing update (9.3) is performed $K$ times, starting from the initial feature vectors $z_i^0$. The feature vectors after $K$ iterations are used for feasibility prediction with a

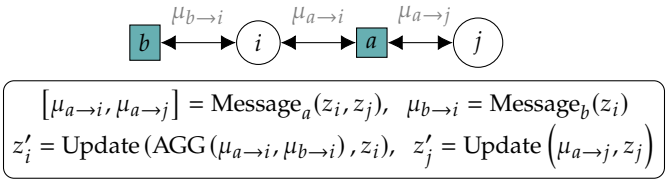



Figure 9.2.: Message passing in a Factored-NLP with two variables $(i, j)$ and two constraints $(a, b)$.

small MLP classifier,

$$\hat{y}_i = \text{Classifier}(z_i^K)\,. \tag{9.4}$$

The number of iterations is a hyperparameter of the model, and the weights of the MLPs may differ between message passing iterations $k = 1, \ldots, K$ (e.g., $\texttt{Message}_a$ at $k = 1$, denoted with $\texttt{Message}_a^1$, is different from $\texttt{Message}_a^k$ at iteration $k$). The parameters of the classifier, message, and update networks are trained end-to-end to minimize the weighted binary cross-entropy loss between the prediction $\hat{y}_i$ and the *variable-feasibility* labels $y_i$.

**Algorithm 9.1** Conflict Extraction with a Graph Neural Network.

**Input:**
Factored-NLP $G = (X_G \cup \Phi_G, E_G)$ ▷ Infeasible factored nonlinear program
$\texttt{GNN\_Model} = \{\texttt{Message}_a^k, \texttt{Update}^k, \texttt{Classifier}\}$ ▷ Learned GNN model
`Solve` ▷ Algorithm provided by the user
`Reduce` ▷ Algorithm provided by the user
**Output:** $M \subseteq G$ ▷ Minimal infeasible subgraph
$\{\hat{y}_i\} = \texttt{GNN\_Model}(G)$
$\delta \leftarrow 0.5,\ \delta_r \leftarrow 1.2$
**while True do**
    $X_\delta = \{x_i \in X_G \mid \hat{y}_i < \delta\}$ ▷ Candidate infeasible variables
    **for** $g \in \texttt{CCA}(G[X_\delta])$ **do** ▷ Connected component analysis
        `feasible` $\leftarrow \texttt{Solve}(g)$
        **if** not `feasible` **then**
            $M \leftarrow \texttt{Reduce}(g)$
            **Return** $M$
        **end if**
    **end for**
    $\delta \leftarrow \delta \times \delta_r$
**end while**

### 9.3.4. Algorithm to Detect Minimal Infeasible Subgraphs

To account for the approximation in our variable classification formulation and small prediction errors, we integrate the learned classifier into a classical algorithm to detect minimal infeasible subgraphs.

We assume the user provides the `Solve` and `Reduce` routines, which check if a Factored-NLP is feasible and compute a minimal infeasible subset of constraints, respectively. `Reduce` is an expensive routine, as it involves solving several nonlinear programs by adding and removing constraints. The number of evaluated NLPs—and therefore the computation time—depends on the size of the input graph: linear with the total number of variables according to [Amaldi et al., 1999], or logarithmic according to [Junker, 2004].

Our algorithm is outlined in Algorithm 9.1. The GNN model is evaluated once on the input Factored-NLP and computes feasibility scores, $\hat{y}_i$, for each variable. By iteratively increasing the classification threshold, $\delta$, we select the candidate infeasible variables, $X_\delta$, with a score lower than the current threshold, $\delta$. We then generate candidate infeasible subgraphs with connected component analysis on the *variable-induced* subgraph, $G[X_\delta]$, which are evaluated with `Solve`. Once an infeasible subgraph is found, we use `Reduce` to obtain a minimal infeasible subgraph.

A traditional conflict extraction approach would run `Solve` and `Reduce` directly on the input Factored-NLP. The acceleration in our algorithm, therefore, comes from evaluating these routines on small (ideally minimal) candidates. Algorithm 9.1 can be extended to compute multiple minimal infeasible subgraphs by omitting the return statement and adding a special check to avoid solving a supergraph of an infeasible subgraph identified in a previous iteration.

## 9.4. Factored-NLP for Manipulation Planning

### 9.4.1. Structure of the Factored-NLP

As an application within TAMP, we use our model to predict minimal infeasibility when computing the keyframe configurations that fulfill a high-level task plan.

When the optimization problem is infeasible, finding a minimal subset of infeasible constraints is crucial for understanding the cause of the infeasibility and providing valuable feedback to the task planner, as demonstrated in our conflict-based TAMP planner (Chapter 5).

The Factored-NLPs used in this chapter are generated using the formulation Planning with Nonlinear Transition Constraints (PNTC, Section 5.3) presented in Chapter 5, which

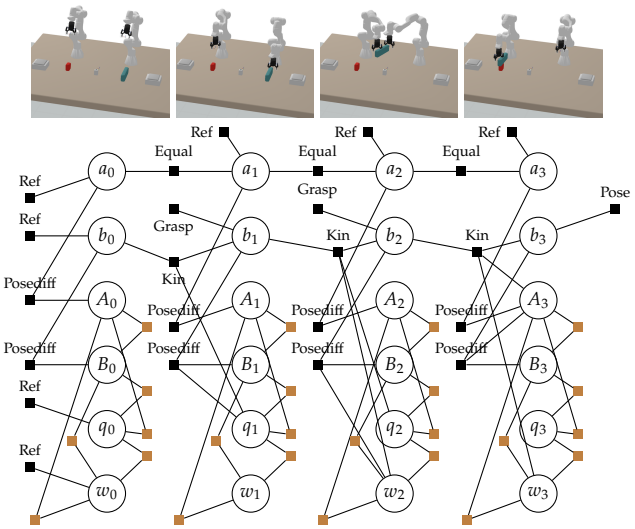



Figure 9.3.: Factored-NLP for the task plan ⟨*pick object B with robot Q from B_init*, *pick object B with robot W from robot Q*, *place object B with robot W on object A*⟩. Circles represent variables, and squares represent constraints. Each column symbolizes a keyframe of the manipulation sequence. $q, w$ are the configurations of the two robots; $A, B$ are the absolute positions of the two objects, and $a, b$ are the relative poses of these objects with respect to their parent in the kinematic tree (e.g., the table, a robot, or another object as indicated by the task plan). See the main text and Chapter 3 for an explanation of variables and constraints.

ensures a consistent local and repeatable structure to enable generalization across different nonlinear programs.

However, we employ a different, yet equivalent, formulation of the continuous space within PNTC, using two continuous variables for each movable object: one indicating the absolute position and another for the relative, along with additional constraints.

This makes the Factored-NLPs more redundant, as the absolute positions of the objects can be deduced from their relative positions and the positions of the parent frame. Nevertheless, now the Factored-NLP can be formulated using a smaller number of distinct types of nonlinear constraints. Since each type of constraint corresponds to a unique `Message` network, this formulation becomes vital for generalization in scenes with more objects.

Thus, Factored-NLPs in this chapter contain three types of variables: robot configurations, object absolute positions, and object relative positions with respect to the parent frame. Beyond the nonlinear constraints highlighted throughout the thesis, we now incorporate a new constraint type that ensures the geometric consistency between relative and absolute poses of objects (*Posediff*).

In Fig. 9.3, we display the Factored-NLP corresponding to the sequence ⟨*pick object B with robot Q from B_init*, *pick object B with robot W from robot Q*, *place object B with robot W on object A*⟩, in an environment with two robots, $Q$ and $W$, and two objects, $A$ and $B$. For comparison, the Factored-NLP using the original PNTC formulation for the same task plan is presented in Fig. 5.3.

Lastly, it is worth noting that in this chapter, we do not consider trajectory variables (e.g., $\tau_q$ in Fig. 5.3 in Chapter 5) because the keyframe variables already provide very informative information for evaluating geometric infeasibility.

### 9.4.2. Encoding of the Problem in the Initial Feature Vectors

The structure of the Factored-NLP encodes the number of objects, robots, and the task plan. The geometric description of the environment is encoded locally in the initial feature vector of each variable $z_i^0$. Specifically, the initial feature vector includes the information of unary constraints (i.e., constraints evaluated only on a single variable, which are then not added to the message passing architecture), additional semantic class information (for example, whether the variable represents an object or a robot, but without including a notion of a time index or entity), and geometric information that is relevant for the constraints (for example, the size of the objects). The dimension of $z_i^0$ is fixed, and shorter feature vectors are padded with zeros.

For example, suppose that the Factored-NLP of Fig. 9.3 is evaluated in a scene where robot $Q$ is at pose $T_Q$ = [0.32, 0.41, 0.56, 0.707, 0, 0, 0.707] , the start position of object $A$ is $T_A$ = [0.35, 0.4, 0.5, 0.707, 0, 0, 0.707] , and object $A$ is a box of size $S_A$ = [0.2, 0.3, 0.2]. Then the $z^0$ of variables $\{q_0, q_1, q_2, q_3\}$ is [1, 0, 0, 0, 0, 0, $T_Q$] , where the first six components indicate that it is a robot, and $T_Q$ is the base pose. The $z^0$ of $\{a_0, a_1, a_2, a_3\}$ is [0, 1, 0, 0, 0, 0, $T_A$] , where the first components indicate that it is a relative pose with respect to the reference position $T_A$. The $z^0$ of $\{A_0, A_1, A_2, A_3\}$ is [0, 0, 1, 0, 0, 0, $S_A$, 0, 0, 0, 0] to indicate that it is an absolute position of an object of size $S_A$.

## 9.5. Experimental Results

**Scenario** We evaluate our model in robotic sequential manipulation. The objective is to predict minimal infeasibility of task plans that build towers and rearrange blocks into different configurations, in scenarios containing a varying number of blocks, robots, and movable obstacles, in different positions. See Figs. 9.4 and 9.6. The following settings are used to generate the training dataset (4800 Factored − NLPs):

– Five movable objects: 3 blocks and 2 obstacles. Both types of objects have collision constraints, but obstacles are larger and usually block grasps or placements.

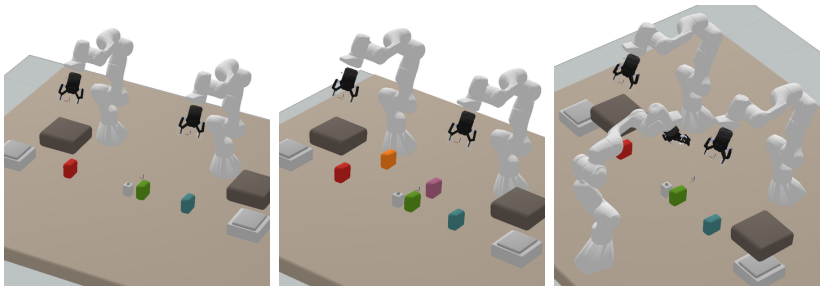


Figure 9.4.: TAMP scenarios. Obstacles are brown, blocks are colorful and tables are white. *Left*: Training Data, *Middle*: + Blocks dataset, *Right:* + Robots dataset.

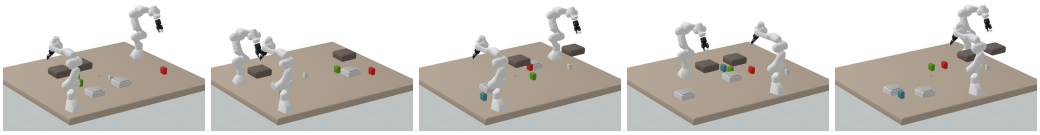


Figure 9.5.: Training Data – Different Scenes. The positions of robots, obstacles, and blocks are randomized.

- Two robots: 7-DOF Panda robot arms, which can pick and place objects using a top grasp.
- Different geometric scenes: the positions of the objects, robots, and tables are randomized.
- Manipulation sequences of lengths 4 to 7 (length of the high-level task plan).

To evaluate the generalization capabilities of the learned model, we consider three additional datasets:

- `+Robots`: we add an additional robot.
- `+Blocks`: we add two additional blocks.
- `+Actions`: this dataset contains Factored-NLPs from longer task plans (lengths of 8 to 10).

### 9.5.1. Data Generation

For training the GNN model, we need a set of Factored-NLPs with labeled variables to indicate whether they belong to a minimal infeasible subset. First, we generate a set of

interesting task plans. Second, we evaluate the manipulation sequences on random geometric scenes. To compute the feasibility labels, we adapt the conflict extraction algorithm of Chapter 5 to find up to 10 minimal infeasible subgraphs.

### 9.5.2. Accuracy of the GNN Classifier

We compare our model (*GNN*) against a Multilayer Perceptron (*MLP*) and a sequential model (*MLP-SEQ*), trained with the same dataset.

The *MLP* computes $\hat{y}_i = \text{MLP}(\tilde{z}_i^0, A, C)$, where $\tilde{z}_i^0 = [z_i^0, t_i, e_i]$ is the feature vector of the variable we want to classify. It concatenates the feature vector $z_i^0$ used in the *GNN*, with the time index $t_i$ of the variable, and a parametrization that defines the *name* of the variable $e_i$ (for instance, we represent an object with its starting pose). Note that $t_i$ and $e_i$ are not used in the *GNN* model because this information is encoded in the structure of the graph. $A$ is the encoding of the whole task plan, using small vectors to encode each token, e.g., {"pick", "block1", "l_gripper", "table"}. To account for sequences of different lengths, we fix a maximum length and add padding. $C$ is the scene parametrization and contains the position and shapes of all possible objects and robots.

We also evaluate *MLP-SEQ*, a sequential model $\text{MLP}(\tilde{z}_i^0, \text{SEQ}(A), C)$ that encodes the action sequence with a recurrent network (Gated Recurrent Units).

We first evaluate the accuracy of the models to predict if a variable belongs to a minimal infeasible subset, see Table 9.1. Our *GNN* model outperforms the alternative architectures, both in the original *Train Data* and, especially, in the extension datasets. Our model maintains a constant ~95 % success rate across all datasets, while the performance of *MLP* and *MLP-SEQ* drops to 48 % and 75 %, respectively. We also evaluate the accuracy of our model to predict infeasible subgraphs, using the proposed method that combines variable classification and connected component analysis, with the initial threshold for classification set at $\delta = 0.5$. Our model outperforms *MLP* and *MLP-SEQ*, finding between 70 % and 57 % of the infeasible subgraphs, and 30 %-50 % of the predicted subgraphs are minimal, see Table 9.2. Between 34 %-48 % of the predicted infeasible graphs are actually feasible. As shown later, these levels of accuracy, together with our iterative threshold strategy, result in a strong acceleration.

*MLP*, *MLP-SEQ*, and *GNN* have the same information to make the predictions because the Factored-NLP is a deterministic mapping of the action sequence and the geometric scene. Although the unstructured *MLP* and *MLP-SEQ* baselines could potentially learn this mapping, our experiments show that the representation does not emerge naturally, confirming that a structured model yields better generalization.

Table 9.1.: Classification accuracy. Each pair indicates the accuracy of predicting feasible and infeasible variables.

| | Train Data | + Blocks | + Robots | + Actions |
|---|---|---|---|---|
| *GNN* | (94.7, 95.4) | (96.1, 95.2) | (95.7, 95.3) | (94.6, 94.1) |
| *MLP* | (93.0, 82.2) | (93.4, 80.8) | (93.0, 80.8) | (91.0, 48.0) |
| *MLP-SEQ* | (83.5, 88.1) | (82.3, 88.8) | (82.1, 88.8) | (74.0, 75.3) |

Table 9.2.: Prediction of infeasible subgraphs. Each pair indicates the ratio "found / total" (higher is better) and "minimal / found" (higher is better).

| | Train Data | + Blocks | + Robots | + Actions |
|---|---|---|---|---|
| *GNN* | (71.2, 54.1) | (58.9, 33.3) | ( 70.2, 55.3) | ( 57.1, 41.9) |
| *MLP* | (58.5, 54.6) | (34.5, 53.2) | (55.2, 37.6) | (22.1, 35.5) |
| *MLP-SEQ* | (65.7, 26.0) | (28.6, 21.2) | (61.3, 09.5) | (36.3, 11.0) |

### 9.5.3. Finding Minimal Infeasible Subgraphs

We analyze the time required to find one minimal infeasible subgraph in an infeasible Factored-NLP with the following algorithms:

– *Oracle*, which knows beforehand the minimal infeasible subgraph and executes only a single call to `Solve` and `Reduce` on this minimal infeasible subgraph. This provides a lower bound on the compute time.

Table 9.3.: Finding one minimal infeasible subgraph, evaluated on 100 different Factored-NLPs. Each pair indicates the average number of evaluated NLPs (lower is better) and the compute time (lower is better), normalized by the results of *GNN+g1*.

| | Train Data | + Blocks | + Robots | + Actions |
|---|---|---|---|---|
| *GNN+e* | (1.57, 2.25) | (1.44, 2.09) | (1.66, 2.14) | (1.50, 2.19) |
| *GNN+g1* | (1, 1) | (1, 1) | (1, 1) | (1, 1) |
| *Oracle* | (0.83, 0.97) | (0.62, 0.79) | (0.83, 0.84) | (0.71, 0.86) |
| *Expert* | (3.66, 4.32) | (3.13, 5.06) | (4.33, 4.62) | (3.33, 4.56) |
| *General 2* | (3.50, 64.1) | (3.30, 163) | (3.50, 66.5) | (3.83, 128) |

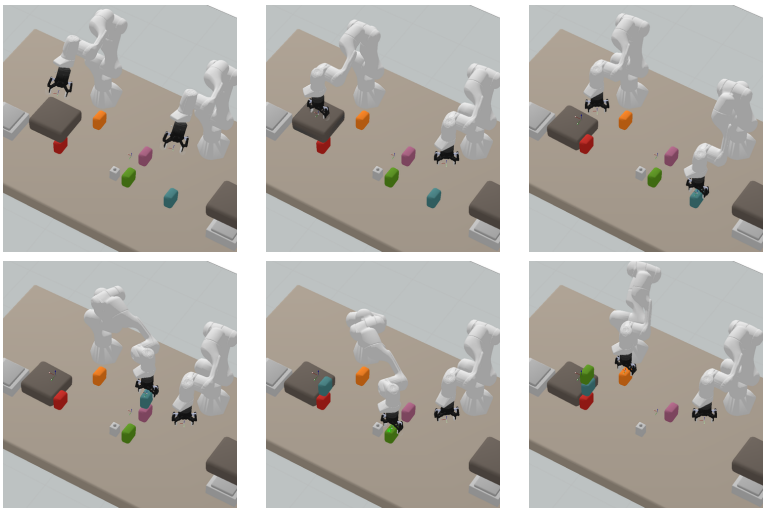


Figure 9.6.: Keyframes of a task plan in the evaluation dataset + *Actions*. Robots build a tower of blocks [*red*, *blue*, *green*, *orange*], moving first an obstacle.

- *General {1,2}*, which are generic algorithms for conflict extraction: *General 1* uses constraint filtering [Amaldi et al., 1999], and *General 2* uses *QuickXplain* [Junker, 2004].
- *Expert* is the heuristic algorithm for conflict extraction in manipulation planning presented in Chapter 5.

  It exploits the temporal structure, domain relaxations, and the convergence of the optimizer to quickly discover the conflicts.
- *GNN+{e,g1}* combines the prediction of our *GNN* model with either *Expert* or *General 1*, which are used as the `Reduce` routine in Algorithm 9.1.

Results are shown in Table 9.3. *GNN+g1* is 60-120x faster than *General 2* (which is faster than *General 1*). This highlights the benefits of our approach in domains where we can compute a dataset using *General* offline and train the model to get an order-of-magnitude improvement in new problems. *GNN+g1* is 4-5x faster than the *Expert* algorithm and only 1.2x slower than an oracle. Moreover, the acceleration provided by *GNN* is maintained in all the datasets. This confirms the good accuracy and generalization of the architecture seen in the classification results. As a side note, *Expert* is faster than *General 2* because it solves many small feasible NLPs first until it finds one that is infeasible (which is faster than solving infeasible NLPs).

### 9.5.4. Integration in a Conflict-Based TAMP Planner

We demonstrate the benefits of using deep learning to accelerate conflict extraction inside the Factored-NLP Planner (Chapter 5). Our planner iteratively generates candidate task plans, detects infeasible subgraphs of the Factored-NLP, and encodes this information back into the discrete description of the problem. For this evaluation, we define 10 TAMP problems for a reference environment in each setting: `+Actions`, `+Robots`, and `+Blocks`. We report the sum (across all 10 TAMP problems) of the number of solved NLPs (lower is better) and the computational time in the conflict extraction component of the TAMP solver. *GNN+e* (which is more robust than *GNN+g1* in this setting) takes only (8.33 s, 511 NLPs) in `+Actions`, (9.83 s, 603 NLPs) in `+Robots`, and (63.9 s, 1979 NLPs) in `+Blocks`, and is between 2 and 3 times faster than the *expert* algorithm, which requires (24.2 s, 731 NLPs), (38.7 s, 1116 NLPs), and (137.9 s, 2554 NLPs).

## 9.6. Limitations

From a practical perspective, the limitation of our approach is that the neural model primarily serves as a component within an extensive model-based pipeline for TAMP. Conflict extraction alone is not capable of generating robot motion or suggesting the next potential task plan.

Furthermore, our graph-based classifier, despite its potential for accurate predictions, requires integration into a traditional conflict extraction algorithm to refine the identified conflicts. Looking ahead, future research might branch out in three distinct and potentially conflicting directions: improving the neural models to achieve impeccable predictions, adapting the TAMP solver to handle approximate predictions, or progressively transitioning from model-based TAMP solvers with integrated learning components to fully learning-based TAMP solvers.

## 9.7. Conclusion

In this chapter, we have presented a neural model to predict the minimal infeasible subsets of variables and constraints in a factored nonlinear program. The structure of the nonlinear program is used for neural message passing, providing generalization to problems with more variables and constraints.

Our model achieves high accuracy, and the predictions can be integrated to guide and accelerate classical and heuristic algorithms for detecting minimal conflicts. As confirmed in the experiments, the neural model is directly applicable as a submodule of the Factored-NLP Planner (Chapter 5) to accelerate the conflict detection pipeline.

When compared to Deep Generative Constraint Sampling (Chapter 8), these graph neural models exhibit superior generalization across varied task plans. As such, a new research question is how to extend these ideas and insights to the generative setting.

Similar to foundational models in computer vision or natural language processing, we see great potential in investigating general models for manipulation planning, where these models can serve as backbones for different downstream tasks in any TAMP solver. In this setting, graph neural networks are an attractive architecture to combine logic and geometric information in an end-to-end, yet interpretable, manner. The structure of the graph is used to model discrete information such as the task plan and the number of objects and robots, and the feature vectors of nodes and edges are used to encode geometric information.

# Chapter 10

# Conclusions

We conclude this thesis by summarizing the main contributions and results of this work and by discussing some possible future directions and open challenges.

## 10.1. Summary of Contributions

**Factored structure of Task and Motion Planning** In Chapter 3, we present a refined factored representation of optimization problems within Task and Motion Planning. This factorization naturally arises from the temporal, object-centric, and robot-centric representations of the problem, but it had not been studied in detail in previous optimization-based TAMP solvers.

The formal definition, properties, and methods to generate these factored nonlinear programs are presented in Chapter 5. During the development of this thesis, we demonstrate the advantages and generality of this representation, illustrating how it can represent a wide range of diverse task plans in a unified manner, including problems involving multiple robots, objects, and diverse task plans.

Unintentionally, our graph representation has come to closely resemble the factored representation of PDDLStream [Garrett et al., 2020]. Exploiting this factorization is crucial for designing efficient interfaces between continuous optimization and task planning. From a sampling perspective, we can design good sampling operations that incrementally compute the motion, and from an optimization perspective, we can detect infeasible subsets of constraints and encode this information back into the task planner (Chapter 5).

We strongly believe that the factored representation of TAMP can be used to bridge the gap between optimization and sample-based approaches to TAMP (Chapters 6 and 7), and to

improve the generalization and efficiency of deep learning methods for TAMP (Chapters 8 and 9).

**Integrated planning and optimization for Task and Motion Planning** The objective of the first part of the thesis is to combine trajectory optimization and discrete task planning into a unified and efficient framework for solving TAMP problems. A fundamental challenge in solving large-scale problems with hard geometric and physical constraints is to automatically inform the task planner about motion feasibility.

In Chapters 4 and 5, we present two new conflict-based solvers for TAMP that attempt to solve the TAMP problem by evaluating candidate task plans, identifying why plans fail when considering the continuous constraints, and encoding this information back into the discrete planning problem.

Our first new solver, *Diverse Planning for LGP* (Chapter 4), detects and encodes infeasible prefixes of the task plan. Additionally, ideas from diverse planning and meta-reasoning are used to choose the most promising plans to test next and to decide how much compute effort to allocate to searching for conflicts.

In *Conflict-Based Search in Factored Logic Geometric Programs* (Chapter 5), we introduce a new factored hybrid planning formulation for TAMP, which provides a more efficient interface between discrete planning and optimization. Based on this formulation, our second solver, the *Factored-NLP Planner*, can now detect any infeasible subset of nonlinear constraints and encode this information back into the task planner, resulting in a highly efficient bidirectional interface between task planning and motion planning.

**Meta-solvers: Adaptive combination of sampling and optimization methods** In the second part of the thesis, we propose two meta-solvers: algorithms that can automatically combine two different techniques—optimization and sampling—to compute the robot motion in TAMP problems.

In *Learning Optimal Sampling Sequences for Robotic Manipulation* (Chapter 6), we begin by considering the problem of computing the motion for a fixed task plan, which corresponds to solving challenging factored nonlinear programs. Using a Monte Carlo Tree Search formulation, our method automatically discovers the best sequence of sampling and/or optimization operations to solve the problem. This adaptive algorithm outperforms both fixed sampling sequences and full nonlinear optimization used in previous work, in terms of the diversity and number of solutions found within a fixed amount of compute time.

In *Towards Meta-Solvers for Task and Motion Planning* (Chapter 7), we present a preliminary study towards a complete TAMP meta-solver that optimizes the task plan and the robot motion while automatically deciding whether it is better to use sequential sampling or joint

optimization. To this end, we first introduce a notion of a compute state that extends the traditional discrete-continuous states with additional free states subject to constraints that have not yet been computed.

Our first TAMP meta-solver is an informed search algorithm on this computational space. We show that this simple search strategy can outperform sample-based and optimization-based solvers on average compute time across diverse TAMP settings with few objects and robots. However, further research is needed to enhance the TAMP meta-solver's performance for tackling large-scale problems.

**Accelerated Task and Motion Planning with learning methods** In the third part of the thesis, we propose two novel ways to use learning to accelerate expensive operations in a TAMP solver. We assume that a dataset of solutions to similar problems can be computed offline using a model-based solver, which requires multiple expensive computations, such as solving nonlinear programs with multiple restarts or combinatorial optimization. At runtime, the learned models are used to accelerate our model-based solvers, providing speed-ups on new, unseen problems. In both contributions, exploiting the Factored-NLP representation of the problem is crucial to achieving good generalization and performance.

*Deep Generative Constraint Sampling* (Chapter 8) combines a deep learning model with nonlinear optimization to generate keyframes of manipulation sequences faster. In particular, we use Generative Adversarial Networks to produce a good warm start for nonlinear optimization, outperforming alternative warm start initialization strategies. Here, we transform the Factored-NLP into a directed graphical model, to reduce sample complexity and increase the expressivity and multimodality of the generative model.

In *Learning Feasibility of Factored Nonlinear Programs* (Chapter 9), we propose a classifier that predicts which constraints in a Factored-NLP cannot be fulfilled, a fundamental step in our conflict-based TAMP solver (Chapter 5). The model takes the structure of the Factored-NLP as direct input, together with a local encoding of the manipulation scene, providing good generalization across different task plans. Using this graph-based classifier, we can detect conflicts an order of magnitude faster than classical conflict extraction strategies.

## 10.2. Open Challenges and Future Work

In this section, we discuss open challenges and outline several promising ideas for further research in the field of Task and Motion Planning in robotics.

**TAMP benchmarks** Different TAMP methods utilize slightly varied formulations and various benchmark problems, making it challenging to compare and analyze algorithms, uti-

lize common tools, and draw on ideas from other research groups. Additionally, similar problem formulations to TAMP are often studied under different names in robotics, such as multimodal motion planning and manipulation planning, again with slightly varied formulations and heuristics tailored to different environments. There is a clear need for standardized benchmarks. Drawing inspiration from the discrete planning and reinforcement learning communities, it is evident that we need a unified way to define the problem (e.g., PDDL [McDermott et al., 1998] or the OpenAI-Gym interface [Brockman et al., 2016]), and a set of different standardized scenarios (e.g., domains in PDDL and benchmarks or datasets in RL) to evaluate our methods.

While there have been some attempts to establish TAMP benchmarks [Lagriffoul et al., 2018], the utilization of different software tools and slightly varied problem formulations has hindered broad adoption. A successful TAMP benchmark should define a common interface for stating a problem and its solution, a simulator to validate any provided solution, and interfaces that could optionally be utilized in the solvers, such as discrete abstractions and differentiable constraints.

Moreover, our research experience underscores the need to develop superior motion planning and optimization tools. Although certain algorithmic ideas, such as sample-based motion planning and trajectory optimization, have reached a mature state, the research community still requires high-quality, standalone, and open-source implementations.

**Optimization-based solvers for TAMP** In this thesis, we have presented two TAMP solvers that combine discrete planning and optimization with a conflict-based approach, effectively scaling to large-scale problems involving multiple robots and objects.

However, a fundamental issue with optimization-based approaches remains, namely, convergence to local minima. Because our solvers are conflict-based, a failure to find a solution on a feasible problem due to an unfortunate initialization compromises the completeness of our approach.

In practice, our solvers have performed very well in tabletop environments and with simple geometric shapes, where there are few local optima in the trajectory optimization problems. This assumption holds true for many relevant manipulation problems. However, for more general applications, we need to address the issue of local minima, making our solvers more robust against failed optimization attempts and allowing them to try the same problem again with a different initialization. As discussed in the limitations of our solvers, a promising future direction is to use soft or probabilistic conflict formulations.

**Nonlinear optimization in robotics** Nonlinear optimization methods are a powerful tool in robotics and have shown great success in solving robotics problems in high-dimensional spaces with complex constraints, e.g., [Winkler et al., 2018, Mordatch et al., 2012, Toussaint

et al., 2018]. Unfortunately, these results are often difficult to reproduce for non-experts, as they require technical knowledge and experience to formulate the problem correctly (e.g., the selection of variables and constraints, the scaling of each term, and the warm start).

In practice, we observe that sequential conditional sampling and sample-based motion planning are more robust to the choice of hyperparameters and the exact problem formulation. In contrast to optimization, where bad hyperparameters often lead to failure, in sampling-based algorithms, choosing hyperparameters incorrectly often means longer solution times, but solvers manage to find a solution.

A key difference is that sampling-based algorithms are often anytime algorithms, improving the success rate and solution cost with more compute time. The naïve way to convert optimization methods into an anytime algorithm is to add random restarts. However, we observe that random restarts are not informative enough to solve hard problems within a reasonable timeframe. We believe there is great potential in more intelligent restart strategies that, for instance, use both the structure of the problem and previous computations, as explored in this thesis.

Moving forward, to increase the influence and adoption of nonlinear optimization in robotics, it is essential to incorporate these advanced restart strategies into nonlinear solvers. Coupled with automatic hyperparameter tuning and appropriate scaling of costs and constraints, this opens up interesting avenues for both research and software development.

**Development of meta-solvers for TAMP – research and software infrastructure** When deploying TAMP systems in the real world, we aim to ensure that our planning algorithms consistently perform quickly across all potential problems. Achieving this robustness is only possible with TAMP solvers designed to identify the most efficient computing method for solving current problems. Our algorithms, presented in Chapters 6 and 7, represent a foundational step in this direction, yet they have some limitations in terms of scalability and applicability.

Thinking in terms of computational space and optimization over computing decisions is a very powerful idea, and we believe that this is a promising direction to pursue. However, we require more complex models to reason about computation cost and success, to share information between similar task plans, and to find intelligent ways to reuse previous computations.

Further, our research on TAMP meta-solvers highlights that, in addition to research contributions, there is a need for better software infrastructure and tools to combine sampling, discrete search, and optimization into a unified framework.

**Perception for TAMP** A limitation of our work is that we assume an extremely accurate perception module, which provides a perfect model of the world in a form that is convenient for planning. For instance, we assume knowledge of the objects' positions in the world, differentiable model-based nonlinear constraints, and a low-dimensional representation (e.g., parametric shape) of the objects we intend to manipulate. Additionally, we presume these objects have simple geometric shapes, such as cubes, cylinders, and spheres.

An attractive and structured approach to extend our TAMP solvers to include perception is to utilize neural-based perception modules that can directly map high-dimensional sensor input to the required representation for planning. Recent deep learning methods have shown great success in perception tasks, such as object detection, and pose estimation and segmentation [He et al., 2017, Kirillov et al., 2023, Redmon et al., 2016, Labbé et al., 2020].

However, small perception errors can lead to large failures in the planning and execution of manipulation tasks. An alternative approach is to learn directly manipulation features or discrete states for planning. For instance, this could involve generating a discrete state representation directly from images for high-level task plan computation [Yuan et al., 2022], or learning manipulation features that can be used directly to synthesize motion using trajectory optimization [Ha et al., 2022, Simeonov et al., 2022].

**Learning universal policies for robotic manipulation** In Part III of this thesis, we have demonstrated how to leverage learning to accelerate expensive operations in a TAMP solver. Although the proposed learned modules are integrated as small components in model-based algorithms, they have limited applications as standalone components.

A natural extension is to train models that can either solve the entire TAMP problem or, at the very least, substantial components of it, such as computing the complete motion for a given task plan. The scope of research in this area is vast, with numerous contributions, such as [Driess et al., 2021, Kase et al., 2020, Fang et al., 2019, Gupta et al., 2020, Ichter et al., 2020], among many others.

Drawing from the contributions in our thesis, we aim to utilize our structured representation to enhance efficiency and generalization across a wide array of manipulation tasks. Our future work involves applying this graph representation to develop neural universal manipulation policies, which directly (or indirectly) map the current state to the subsequent action that robots should take, all while conditioned on the high-level task plan. By leveraging the graph structure of the optimization problem, a single policy—trainable either through imitation or reinforcement learning—can be adapted to various task plans in different scenes. Graph-structured policies could be trained with diverse data across different manipulation tasks. The resulting policies could then be applied to larger problems (e.g., those involving more robots or objects), potentially outperforming recent transformer-based architectures such as [Shridhar et al., 2023, Brohan et al., 2023].

We envision that a combination of data, structure, and models will be necessary to create a universal policy with exceptional generalization capabilities, enabling it to tackle various tasks and environments. Structured policies hold the potential to merge learned features from perception and contact models—which are inherently challenging to model—with precise model-based features, such as the robot's kinematics, joint limits, or self-collision avoidance.

## 10.3. Final Remarks

In this thesis, we have studied Task and Motion Planning from a multidisciplinary perspective, combining ideas from optimization, discrete planning, and learning.

Improving TAMP solvers is not just key for the future of robotics but also a fascinating research topic. I hope this thesis has shown that TAMP is a very interesting field, requiring hybrid planning with continuous and discrete variables and constraints, and advanced reasoning about abstraction and structure. All these concepts are essential for any robot operating in our continuous world while leveraging discrete abstractions and decompositions for more effective long-term planning.

The question of whether to integrate learning into model-based reasoning is highly pertinent in the context of TAMP. On one hand, good approximate forward models and model-based solvers are readily available; on the other hand, planning with such models might be time-consuming, often too slow for real-time planning.

From a learning perspective, TAMP constitutes a compelling challenge, given the high dimensionality, requirement for long-term planning, and multimodality. Generating new data or training new neural networks for every distinct TAMP problem is impractical, underscoring the essential need for generalization abilities in learning-based TAMP.

Finding the right balance between model-based approaches and learning can become even more challenging when we consider the real world, with its complex object shapes and perception through images or point clouds. While today it is clear that a combination of learning and model-based methods in TAMP is required to solve complex manipulation problems, this equilibrium could shift in the future. Even if TAMP systems were to become predominantly based on data and neural networks, model-based TAMP would continue to play a central role, providing both a dataset of diverse solutions and the correct understanding and inductive bias for the design of efficient learning-based systems.

Beyond advancements in TAMP solvers, real-world applications will also require deeper exploration into enhanced perception, dexterous manipulation, and dynamic replanning. However, a deep understanding of model-based TAMP is instrumental, even when some real challenges are not considered or are simplified.

From a research perspective, the journey through task and motion planning in robotics has been enlightening, touching upon multiple paradigms in robotics. I hope the algorithms, formulations, and discussions presented in this thesis have piqued your interest and can serve as a starting point for further research in the field.

# Bibliography


[Amaldi et al., 1999] Amaldi, E., Pfetsch, M. E., and Trotter, L. E. (1999). Some structural and algorithmic properties of the maximum feasible subsystem problem. In *Int. Conf. on Integer Progr. and Combinatorial Optimization*.

[Andreani et al., 2008] Andreani, R., Birgin, E. G., Martínez, J. M., and Schuverdt, M. L. (2008). On augmented lagrangian methods with general lower-level constraints. *SIAM Journal on Optimization*, 18(4):1286–1309.

[Arjovsky et al., 2017] Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

[Bäckström and Nebel, 1995] Bäckström, C. and Nebel, B. (1995). Complexity results for $SAS^+$ planning. *Computational Intelligence*, 11(4):625–655.

[Baier and McIlraith, 2006] Baier, J. A. and McIlraith, S. A. (2006). Planning with temporally extended goals using heuristic search. In *International Conference on Automated Planning and Scheduling, ICAPS*.

[Battaglia et al., 2018] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Dover Publications.

[Benders, 1962] Benders, J. F. (1962). Partitioning procedures for solving mixed-variables programming problems. *Numer. Math.*, 4(1):238–252.

[Bertsekas, 1979] Bertsekas, D. P. (1979). Convexification procedures and decomposition methods for nonconvex optimization problems. *Journal of Optimization Theory and Applications*, 29(2):169–197.

[Bertsekas, 1997] Bertsekas, D. P. (1997). Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334.

[Bertsimas and Stellato, 2020] Bertsimas, D. and Stellato, B. (2020). Online mixed-integer optimization in milliseconds.

[Betts, 1998] Betts, J. T. (1998). Survey of numerical methods for trajectory optimization. *Journal of guidance, control, and dynamics*, 21(2):193–207.

[Bicchi and Kumar, 2000] Bicchi, A. and Kumar, V. (2000). Robotic grasping and contact: A review. In *Proceedings 2000 ICRA. Millennium conference. IEEE international conference on robotics and automation. Symposia proceedings (Cat. No. 00CH37065)*, volume 1, pages 348–353. IEEE.

[Biere et al., 2009] Biere, A., Heule, M., and van Maaren, H. (2009). *Handbook of satisfiability*, volume 185. IOS press.

[Billard and Kragic, 2019] Billard, A. and Kragic, D. (2019). Trends and challenges in robot manipulation. *Science*, 364(6446):eaat8414.

[Boddy and Dean, 1989] Boddy, M. and Dean, T. L. (1989). *Solving time-dependent planning problems*. Brown University, Department of Computer Science.

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, 129(1-2):5–33.

[Boyd et al., 2011] Boyd, S., Parikh, N., and Chu, E. (2011). *Distributed optimization and statistical learning via the alternating direction method of multipliers*. Now Publishers Inc.

[Bratman et al., 1988] Bratman, M. E., Israel, D. J., and Pollack, M. E. (1988). Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355.

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym. *CoRR*, abs/1606.01540.

[Brohan et al., 2023] Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Chen, X., Choromanski, K., Ding, T., Driess, D., Dubey, A., Finn, C., et al. (2023). Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*.

[Browne et al., 2012] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

[Cashmore et al., 2018] Cashmore, M., Coles, A., Cserna, B., Karpas, E., Magazzeni, D., and Ruml, W. (2018). Temporal planning while the clock ticks. In *ICAPS*.

[Cauligi et al., 2020] Cauligi, A., Culbertson, P., Stellato, B., Bertsimas, D., Schwager, M., and Pavone, M. (2020). Learning mixed-integer convex optimization strategies for robot planning and control. In *2020 59th IEEE Conference on Decision and Control (CDC)*, pages 1698–1705. IEEE.

[Chen et al., 2016] Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., and Abbeel, P. (2016). Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *CoRR*, abs/1606.03657.

[Coles et al., 2012a] Coles, A., Coles, A., Olaya, A. G., Jiménez, S., López, C. L., Sanner, S., and Yoon, S. (2012a). A survey of the seventh international planning competition. *Ai Magazine*, 33(1):83–88.

[Coles et al., 2012b] Coles, A. J., Coles, A., Fox, M., and Long, D. (2012b). COLIN: planning with continuous linear numeric change. *J. Artif. Intell. Res.*

[Conn et al., 2013] Conn, A. R., Gould, G., and Toint, P. L. (2013). *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*, volume 17. Springer Science & Business Media.

[Cordella et al., 2004] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. (2004). A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*.

[Danna et al., 2007] Danna, E., Fenelon, M., Gu, Z., and Wunderling, R. (2007). Generating multiple solutions for mixed integer programming problems. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 280–294. Springer.

[Dantam et al., 2016] Dantam, N. T., Kingston, Z. K., Chaudhuri, S., and Kavraki, L. E. (2016). Incremental task and motion planning: A constraint-based approach. In *Robotics: Science and systems*.

[Dantam et al., 2018] Dantam, N. T., Kingston, Z. K., Chaudhuri, S., and Kavraki, L. E. (2018). An incremental constraint-based framework for task and motion planning. *The International Journal of Robotics Research*, 37(10):1134–1151.

[Dantzig and Wolfe, 1960] Dantzig, G. B. and Wolfe, P. (1960). Decomposition principle for linear programs. *Operations research*, 8(1):101–111.

[Dechter et al., 2002] Dechter, R., Kask, K., Bin, E., Emek, R., et al. (2002). Generating random solutions for constraint satisfaction problems. In *AAAI*, pages 15–21.

[Deits et al., 2019] Deits, R., Koolen, T., and Tedrake, R. (2019). Lvis: learning from value function intervals for contact-aware robot controllers. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7762–7768. IEEE.

[Deits and Tedrake, 2014] Deits, R. and Tedrake, R. (2014). Footstep planning on uneven terrain with mixed-integer convex optimization. In *2014 IEEE-RAS international conference on humanoid robots*, pages 279–286. IEEE.

[Dellaert et al., 2017] Dellaert, F., Kaess, M., et al. (2017). Factor graphs for robot perception. *Foundations and Trends® in Robotics*.

[Driess et al., 2021] Driess, D., Ha, J.-S., Tedrake, R., and Toussaint, M. (2021). Learning geometric reasoning and control for long-horizon tasks from visual input. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 14298–14305. IEEE.

[Driess et al., 2020] Driess, D., Ha, J.-S., and Toussaint, M. (2020). Deep visual reasoning: Learning to predict action sequences for task and motion planning from an initial scene image. In *Robotics: Science and Systems 2020 (RSS 2020)*. RSS Foundation.

[Driess et al., 2022] Driess, D., Huang, Z., Li, Y., Tedrake, R., and Toussaint, M. (2022). Learning multi-object dynamics with compositional neural radiance fields.

[Driess et al., 2019] Driess, D., Oguz, O., and Toussaint, M. (2019). Hierarchical task and motion planning using logic-geometric programming (hlgp). RSS Workshop on Robust Task and Motion Planning.

[Ebert et al., 2017] Ebert, F., Finn, C., Lee, A. X., and Levine, S. (2017). Self-supervised visual planning with temporal skip connections. In *Conference on Robot Learning*.

[Ermon et al., 2012] Ermon, S., Gomes, C., and Selman, B. (2012). Uniform solution sampling using a constraint solver as an oracle. In *Proceedings of the Twenty-Eighth Conference on Uncertainty in Artificial Intelligence*, UAI'12, page 255–264, Arlington, Virginia, USA. AUAI Press.

[Fang et al., 2019] Fang, K., Zhu, Y., Garg, A., Savarese, S., and Fei-Fei, L. (2019). Dynamics learning with cascaded variational inference for multi-step manipulation. *arXiv preprint arXiv:1910.13395*.

[Felner et al., 2012] Felner, A., Goldenberg, M., Sharon, G., Stern, R., Beja, T., Sturtevant, N., Schaeffer, J., and Holte, R. (2012). Partial-expansion a* with selective node generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 471–477.

[Fernández-González et al., 2018] Fernández-González, E., Williams, B., and Karpas, E. (2018). Scottyactivity: Mixed discrete-continuous planning with convex optimization. *Journal of Artificial Intelligence Research*, 62:579–664.

[Ferrer-Mestres et al., 2017] Ferrer-Mestres, J., Francès, G., and Geffner, H. (2017). Combined task and motion planning as classical AI planning. *CoRR*.

[Fikes and Nilsson, 1971] Fikes, R. E. and Nilsson, N. J. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

[Fox and Long, 2006] Fox, M. and Long, D. (2006). Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*.

[Frey et al., 1997] Frey, B. J., Kschischang, F. R., Loeliger, H.-A., and Wiberg, N. (1997). Factor graphs and algorithms. In *Proceedings of the Annual Allerton Conference on Communication Control and Computing*, volume 35, pages 666–680. Citeseer.

[Funk et al., 2022] Funk, N., Chalvatzaki, G., Belousov, B., and Peters, J. (2022). Learn2assemble with structured representations and search for robotic architectural construction. In *Proceedings of the 5th Conference on Robot Learning*, volume 164 of *Proceedings of Machine Learning Research*, pages 1401–1411. PMLR.

[Garrett, 2021] Garrett, C. R. (2021). Sampling-based robot task and motion planning in the real world. In *Doctoral Thesis, MIT*.

[Garrett et al., 2021] Garrett, C. R., Chitnis, R., Holladay, R., Kim, B., Silver, T., Kaelbling, L. P., and Lozano-Pérez, T. (2021). Integrated task and motion planning. *Annual Review of Control, Robotics, and Autonomous Systems*.

[Garrett et al., 2018] Garrett, C. R., Lozano-Pérez, T., and Kaelbling, L. P. (2018). Sampling-based methods for factored task and motion planning. *The International Journal of Robotics Research*.

[Garrett et al., 2020] Garrett, C. R., Lozano-Pérez, T., and Kaelbling, L. P. (2020). Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning. In *Proceedings of ICAPS*.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280.

[Gerevini et al., 2009] Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*

[Ghasemipour et al., 2022] Ghasemipour, S. K. S., Freeman, D., David, B., Gu, S. S., Kataoka, S., and Mordatch, I. (2022). Blocks assemble! learning to assemble with large-scale structured reinforcement learning.

[Gilmer et al., 2017] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR.

[Ginsberg and Harvey, 1992] Ginsberg, M. L. and Harvey, W. D. (1992). Iterative broadening. *Artificial Intelligence*, 55(2-3):367–383.

[Gogate and Dechter, 2006] Gogate, V. and Dechter, R. (2006). A new algorithm for sampling csp solutions uniformly at random. In *International Conference on Principles and Practice of Constraint Programming*, pages 711–715. Springer.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27:2672–2680.

[Gulrajani et al., 2017] Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. (2017). Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777.

[Gupta et al., 2020] Gupta, A., Kumar, V., Lynch, C., Levine, S., and Hausman, K. (2020). Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning. In *Conference on Robot Learning*, pages 1025–1037. PMLR.

[Ha et al., 2022] Ha, J.-S., Driess, D., and Toussaint, M. (2022). Deep visual constraints: Neural implicit models for manipulation planning from visual input. *IEEE Robotics and Automation Letters*, 7(4):10857–10864.

[Ha et al., 2018] Ha, J.-S., Park, Y.-J., Chae, H.-J., Park, S.-S., and Choi, H.-L. (2018). Adaptive path-integral autoencoders: Representation learning and planning for dynamical systems. *Advances in Neural Information Processing Systems*, 31:8927–8938.

[Hadfield-Menell et al., 2016] Hadfield-Menell, D., Lin, C., Chitnis, R., Russell, S., and Abbeel, P. (2016). Sequential quadratic programming for task plan optimization. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5040–5047. IEEE.

[Hartmann et al., 2020] Hartmann, V. N., Oguz, O. S., Driess, D., Toussaint, M., and Menges, A. (2020). Robust task and motion planning for long-horizon architectural construction planning. In *Int. Conf. on Intelligent Robots and Systems (IROS)*.

[Hartmann et al., 2022] Hartmann, V. N., Orthey, A., Driess, D., Oguz, O. S., and Toussaint, M. (2022). Long-horizon multi-robot rearrangement planning for construction assembly. *IEEE Transactions on Robotics*, 39(1):239–252.

[Haslum et al., 2018] Haslum, P., Ivankovic, F., Ramirez, M., Gordon, D., Thiébaux, S., Shivashankar, V., and Nau, D. S. (2018). Extending classical planning with state constraints: Heuristics and search for optimal planning. *Journal of Artificial Intelligence Research*, 62:373–431.

[Hauser, 2016] Hauser, K. (2016). Learning the problem-optimum map: Analysis and application to global optimization in robotics. *IEEE Transactions on Robotics*, 33(1):141–152.

[Hauser and Latombe, 2010] Hauser, K. and Latombe, J.-C. (2010). Multi-modal motion planning in non-expansive spaces. *The International Journal of Robotics Research*, 29(7):897–915.

[Hauser and Ng-Thow-Hing, 2011] Hauser, K. and Ng-Thow-Hing, V. (2011). Randomized multi-modal motion planning for a humanoid robot manipulation task. *IJRR*, 30(6):678–698.

[He et al., 2017] He, K., Gkioxari, G., Dollár, P., and Girshick, R. (2017). Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969.

[Helmert, 2006] Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246.

[Hemery et al., 2006] Hemery, F., Lecoutre, C., Sais, L., Boussemart, F., et al. (2006). Extracting mucs from constraint networks. In *ECAI*.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hoffmann and Nebel, 2001] Hoffmann, J. and Nebel, B. (2001). The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302.

[Huang et al., 2019] Huang, E., Jia, Z., and Mason, M. T. (2019). Large-scale multi-object rearrangement. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 211–218. IEEE.

[Ichter et al., 2018] Ichter, B., Harrison, J., and Pavone, M. (2018). Learning sampling distributions for robot motion planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7087–7094. IEEE.

[Ichter et al., 2020] Ichter, B., Sermanet, P., and Lynch, C. (2020). Broadly-exploring, local-policy trees for long-horizon task planning. *arXiv preprint arXiv:2010.06491*.

[Junker, 2004] Junker, U. (2004). Preferred explanations and relaxations for over-constrained problems. In *AAAI*.

[Kaelbling and Lozano-Pérez, 2011] Kaelbling, L. P. and Lozano-Pérez, T. (2011). Hierarchical task and motion planning in the now. In *International Conference on Robotics and Automation (ICRA)*, pages 1470–1477.

[Karpas et al., 2018] Karpas, E., Betzalel, O., Shimony, S. E., Tolpin, D., and Felner, A. (2018). Rational deployment of multiple heuristics in optimal state-space search. *Artificial Intelligence*, 256:181–210.

[Kase et al., 2020] Kase, K., Paxton, C., Mazhar, H., Ogata, T., and Fox, D. (2020). Transferable task execution from pixels through deep planning domain learning. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10459–10465. IEEE.

[Katz and Sohrabi, 2020] Katz, M. and Sohrabi, S. (2020). Reshaping diverse planning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, pages 9892–9899. AAAI Press.

[Katz et al., 2018a] Katz, M., Sohrabi, S., Udrea, O., and Winterer, D. (2018a). A novel iterative approach to top-k planning. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, pages 132–140. AAAI Press.

[Katz et al., 2018b] Katz, M., Sohrabi, S., Udrea, O., and Winterer, D. (2018b). A novel iterative approach to top-k planning. In *ICAPS*.

[Kavraki et al., 1996] Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580.

[Khayatkhoei et al., 2018] Khayatkhoei, M., Singh, M., and Elgammal, A. (2018). Disconnected manifold learning for generative adversarial networks. *CoRR*, abs/1806.00880.

[Kim et al., 2017] Kim, B., Kaelbling, L. P., and Lozano-Pérez, T. (2017). Guiding the search in continuous state-action spaces by learning an action sampling distribution from off-target samples. *CoRR*, abs/1711.01391.

[Kingma and Welling, 2013] Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*.

[Kingston et al., 2020] Kingston, Z., Wells, A. M., Moll, M., and Kavraki, L. E. (2020). Informing multi-modal planning with synergistic discrete leads. In *IEEE International Conference on Robotics and Automation*, pages 3199–3205.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

[Kirillov et al., 2023] Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., et al. (2023). Segment anything. *arXiv preprint arXiv:2304.02643*.

[Kocsis and Szepesvári, 2006] Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

[Koehler, 1998] Koehler, J. (1998). Planning under resource constraints. In *ECAI*.

[Koller and Friedman, 2009] Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

[Korf, 1985] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25.

[Krontiris and Bekris, 2016] Krontiris, A. and Bekris, K. E. (2016). Efficiently solving general rearrangement tasks: A fast extension primitive for an incremental sampling-based planner. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3924–3931. IEEE.

[Kuck et al., 2020] Kuck, J., Chakraborty, S., Tang, H., Luo, R., Song, J., Sabharwal, A., and Ermon, S. (2020). Belief propagation neural networks. *Advances in Neural Information Processing Systems*, 33:667–678.

[Kurutach et al., 2018] Kurutach, T., Tamar, A., Yang, G., Russell, S., and Abbeel, P. (2018). Learning plannable representations with causal infogan.

[Labbé et al., 2020] Labbé, Y., Carpentier, J., Aubry, M., and Sivic, J. (2020). Cosypose: Consistent multi-view multi-object 6d pose estimation. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVII 16*, pages 574–591. Springer.

[Lagriffoul et al., 2018] Lagriffoul, F., Dantam, N. T., Garrett, C., Akbari, A., Srivastava, S., and Kavraki, L. E. (2018). Platform-independent benchmarks for task and motion planning. *IEEE Robotics and Automation Letters*, 3(4):3765–3772.

[Lagriffoul et al., 2014] Lagriffoul, F., Dimitrov, D., Bidot, J., Saffiotti, A., and Karlsson, L. (2014). Efficiently combining task and motion planning using geometric constraints. *The International Journal of Robotics Research*.

[Lamiraux and Mirabel, 2021] Lamiraux, F. and Mirabel, J. (2021). Prehensile manipulation planning: Modeling, algorithms and implementation. *IEEE Transactions on Robotics*, 38(4):2370–2388.

[LaValle and Kuffner, 2001] LaValle, S. and Kuffner, J. (2001). Rapidly-exploring random trees: Progress and prospects. *Algorithmic and Computational Robotics*, pages 303–307.

[LaValle, 2006] LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.

[Lelis, 2013] Lelis, L. H. S. (2013). Active stratified sampling with clustering-based type systems for predicting the search tree size of problems with real-valued heuristics. In *Proceedings of the Sixth Annual Symposium on Combinatorial Search,SOCS*. AAAI Press.

[Lembono et al., 2020] Lembono, T. S., Pignat, E., Jankowski, J., and Calinon, S. (2020). Generative adversarial network to learn valid distributions of robot configurations for inverse kinematics and constrained motion planning. *CoRR*, abs/2011.05717.

[Li et al., 2020] Li, R., Jabri, A., Darrell, T., and Agrawal, P. (2020). Towards practical multi-object manipulation using relational reinforcement learning. In *2020 ieee international conference on robotics and automation (icra)*, pages 4051–4058. IEEE.

[Lieder et al., 2014] Lieder, F., Plunkett, D., Hamrick, J. B., Russell, S. J., Hay, N., and Griffiths, T. (2014). Algorithm selection by rational metareasoning as a model of human strategy selection. In *Advances in neural information processing systems*, pages 2870–2878.

[Liffiton and Sakallah, 2008] Liffiton, M. H. and Sakallah, K. A. (2008). Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33.

[Lipovetzky, 2021] Lipovetzky, N. (2021). Width-based algorithms for common problems in control, planning and reinforcement learning. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4956–4960. ijcai.org.

[Long and Fox, 2003] Long, D. and Fox, M. (2003). The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59.

[Ma and Tang, 2021] Ma, Y. and Tang, J. (2021). *Deep Learning on Graphs*. Cambridge University Press.

[Mandalika et al., 2019] Mandalika, A., Choudhury, S., Salzman, O., and Srinivasa, S. (2019). Generalized lazy search for robot motion planning: Interleaving search and edge evaluation via event-based toggles. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 745–753.

[Mansard et al., 2018] Mansard, N., DelPrete, A., Geisert, M., Tonneau, S., and Stasse, O. (2018). Using a memory of motion to efficiently warm-start a nonlinear predictive controller. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2986–2993. IEEE.

[Marques-Silva et al., 2021] Marques-Silva, J., Lynce, I., and Malik, S. (2021). Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press.

[McDermott et al., 1998] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). Pddl-the planning domain definition language.

[Merkt et al., 2018] Merkt, W., Ivan, V., and Vijayakumar, S. (2018). Leveraging precomputation with problem encoding for warm-starting trajectory optimization in complex environments. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5877–5884. IEEE.

[Migimatsu and Bohg, 2020] Migimatsu, T. and Bohg, J. (2020). Object-centric task and motion planning in dynamic environments. *Robotics and Automation Letters*.

[Mildenhall et al., 2021] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. (2021). Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106.

[Mordatch et al., 2012] Mordatch, I., Todorov, E., and Popović, Z. (2012). Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics (TOG)*, 31(4):1–8.

[Mouhoub and Jafari, 2011] Mouhoub, M. and Jafari, B. (2011). Heuristic techniques for variable and value ordering in csps. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 457–464.

[Mousavian et al., 2019] Mousavian, A., Eppner, C., and Fox, D. (2019). 6-dof graspnet: Variational grasp generation for object manipulation. *CoRR*, abs/1905.10520.

[Murali et al., 2020] Murali, A., Mousavian, A., Eppner, C., Paxton, C., and Fox, D. (2020). 6-dof grasping for target-driven object manipulation in clutter. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6232–6238. IEEE.

[Nir et al., 2021] Nir, R., Shleyfman, A., and Karpas, E. (2021). Learning-based synthesis of social laws in STRIPS. In *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021*, pages 88–96. AAAI Press.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. J. (2006). *Numerical Optimization*. Springer, New York, NY, USA, 2e edition.

[O'Ceallaigh and Ruml, 2015] O'Ceallaigh, D. and Ruml, W. (2015). Metareasoning in real-time heuristic search. In *Eighth Annual Symposium on Combinatorial Search*.

[Orthey et al., 2018] Orthey, A., Escande, A., and Yoshida, E. (2018). Quotient-space motion planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8089–8096. IEEE.

[Ortiz-Haro et al., 2023] Ortiz-Haro, J., Ha, J.-S., Driess, D., Karpas, E., and Toussaint, M. (2023). Learning feasibility of factored nonlinear programs in robotic manipulation planning. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3729–3735. IEEE.

[Ortiz-Haro et al., 2022a] Ortiz-Haro, J., Ha, J.-S., Driess, D., and Toussaint, M. (2022a). Structured deep generative models for sampling on constraint manifolds in sequential manipulation. In *Conference on Robot Learning*, pages 213–223. PMLR.

[Ortiz-Haro et al., 2021] Ortiz-Haro, J., Hartmann, V. N., Oguz, O. S., and Toussaint, M. (2021). Learning efficient constraint graph sampling for robotic sequential manipulation. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4606–4612. IEEE.

[Ortiz-Haro et al., 2022b] Ortiz-Haro, J., Karpas, E., Katz, M., and Toussaint, M. (2022b). A conflict-driven interface between symbolic planning and nonlinear constraint solving. *IEEE Robotics and Automation Letters*, 7(4):10518–10525.

[Ortiz-Haro et al., 2022c] Ortiz-Haro, J., Karpas, E., Michael, K., and Toussaint, M. (2022c). Conflict-directed diverse planning for logic-geometric programming. In *Proceedings of ICAPS*.

[Ota, 2004] Ota, J. (2004). Rearrangement of multiple movable objects - integration of global and local planning methodology. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA '04. 2004*, volume 2, pages 1962–1967 Vol.2.

[Pascanu et al., 2017] Pascanu, R., Li, Y., Vinyals, O., Heess, N., Buesing, L., Racanière, S., Reichert, D., Weber, T., Wierstra, D., and Battaglia, P. (2017). Learning model-based planning from scratch. *arXiv preprint arXiv:1707.06170*.

[Paus et al., 2020] Paus, F., Huang, T., and Asfour, T. (2020). Predicting pushing action effects on spatial object relations by learning internal prediction models. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 10584–10590. IEEE.

[Paxton et al., 2019] Paxton, C., Barnoy, Y., Katyal, K. D., Arora, R., and Hager, G. D. (2019). Visual robot task planning. In *International Conference on Robotics and Automation (ICRA)*, pages 8832–8838. IEEE.

[Piotrowski et al., 2016] Piotrowski, W. M., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. (2016). Heuristic planning for pddl+ domains. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*.

[Posa et al., 2014] Posa, M., Cantu, C., and Tedrake, R. (2014). A direct method for trajectory optimization of rigid bodies through contact. *The International Journal of Robotics Research*, 33(1):69–81.

[Redmon et al., 2016] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788.

[Richter and Westphal, 2010] Richter, S. and Westphal, M. (2010). The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177.

[Rivlin et al., 2020] Rivlin, O., Hazan, T., and Karpas, E. (2020). Generalized planning with deep reinforcement learning. *arXiv preprint arXiv:2005.02305*.

[Rossi et al., 2006] Rossi, F., Van Beek, P., and Walsh, T. (2006). *Handbook of constraint programming*. Elsevier.

[Russell and Wefald, 1991] Russell, S. J. and Wefald, E. (1991). Principles of metareasoning. *Artif. Intell.*, 49(1-3):361–395.

[Satorras and Welling, 2020] Satorras, V. G. and Welling, M. (2020). Neural enhanced belief propagation on factor graphs. *CoRR*, abs/2003.01998.

[Scala et al., 2016] Scala, E., Haslum, P., Thiébaux, S., and Ramirez, M. (2016). Interval-based relaxation for general numeric planning. In *ECAI 2016*, pages 655–663. IOS Press.

[Schmitt et al., 2017] Schmitt, P. S., Neubauer, W., Feiten, W., Wurm, K. M., Wichert, G. V., and Burgard, W. (2017). Optimal, sampling-based manipulation planning. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3426–3432. IEEE.

[Schuetz et al., 2021] Schuetz, M. J. A., Brubaker, J. K., and Katzgraber, H. G. (2021). Combinatorial optimization with physics-inspired graph neural networks. *CoRR*, abs/2107.01188.

[Seipp et al., 2012] Seipp, J., Braun, M., Garimort, J., and Helmert, M. (2012). Learning portfolios of automatically tuned planners. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 22, pages 368–372.

[Selsam et al., 2018] Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2018). Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*.

[Shen et al., 2020] Shen, W., Trevizan, F., and Thiébaux, S. (2020). Learning domain-independent planning heuristics with hypergraph networks. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 574–584.

[Shoukry et al., 2018] Shoukry, Y., Nuzzo, P., Sangiovanni-Vincentelli, A. L., Seshia, S. A., Pappas, G. J., and Tabuada, P. (2018). Smc: Satisfiability modulo convex programming. *Proceedings of the IEEE*.

[Shperberg et al., 2019] Shperberg, S. S., Coles, A., Cserna, B., Karpas, E., Ruml, W., and Shimony, S. E. (2019). Allocating planning effort when actions expire. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 2371–2378.

[Shridhar et al., 2023] Shridhar, M., Manuelli, L., and Fox, D. (2023). Perceiver-actor: A multi-task transformer for robotic manipulation. In *Conference on Robot Learning*, pages 785–799. PMLR.

[Siciliano et al., 2008] Siciliano, B., Sciavicco, L., Villani, L., and Oriolo, G. (2008). *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 1st edition.

[Silva and Sakallah, 1999] Silva, J. P. M. and Sakallah, K. A. (1999). GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521.

[Silver et al., 2021] Silver, T., Chitnis, R., Curtis, A., Tenenbaum, J. B., Lozano-Perez, T., and Kaelbling, L. P. (2021). Planning with learned object importance in large problem instances using graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*.

[Siméon et al., 2004] Siméon, T., Laumond, J.-P., Cortés, J., and Sahbani, A. (2004). Manipulation planning with probabilistic roadmaps. *IJRR*, 23(7-8):729–746.

[Simeonov et al., 2020] Simeonov, A., Du, Y., Kim, B., Hogan, F. R., Tenenbaum, J., Agrawal, P., and Rodriguez, A. (2020). A long horizon planning framework for manipulating rigid pointcloud objects.

[Simeonov et al., 2022] Simeonov, A., Du, Y., Tagliasacchi, A., Tenenbaum, J. B., Rodriguez, A., Agrawal, P., and Sitzmann, V. (2022). Neural descriptor fields: Se (3)-equivariant object representations for manipulation. In *2022 International Conference on Robotics and Automation (ICRA)*, pages 6394–6400. IEEE.

[Song et al., 2021] Song, D., Fernbach, P., Flayols, T., Del Prete, A., Mansard, N., Tonneau, S., and Kim, Y. J. (2021). Solving footstep planning as a feasibility problem using l1-norm minimization. *IEEE Robotics and Automation Letters*, 6(3):5961–5968.

[Srivastava et al., 2014] Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S. J., and Abbeel, P. (2014). Combined task and motion planning through an extensible planner-independent interface layer. In *ICRA*.

[Stilman et al., 2007] Stilman, M., Schamburek, J.-U., Kuffner, J., and Asfour, T. (2007). Manipulation planning among movable obstacles. In *Proceedings 2007 IEEE international conference on robotics and automation*, pages 3327–3332. IEEE.

[Sutanto et al., 2020] Sutanto, G., Fernández, I. M. R., Englert, P., Ramachandran, R. K., and Sukhatme, G. S. (2020). Learning equality constraints for motion planning on manifolds.

[Tang and Hauser, 2018] Tang, G. and Hauser, K. (2018). Discontinuity-sensitive optimal control learning by mixture of experts. *CoRR*, abs/1803.02493.

[Tate, 1977] Tate, A. (1977). Generating project networks. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*, pages 888–893. William Kaufmann.

[Thrun et al., 2005] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press.

[Todorov, 2011] Todorov, E. (2011). A convex, smooth and invertible contact model for trajectory optimization. In *2011 IEEE International Conference on Robotics and Automation*, pages 1071–1076. IEEE.

[Toenshoff et al., 2021] Toenshoff, J., Ritzert, M., Wolf, H., and Grohe, M. (2021). Graph neural networks for maximum constraint satisfaction. *Frontiers in artificial intelligence*, 3:580607.

[Tonneau et al., 2018] Tonneau, S., Del Prete, A., Pettré, J., Park, C., Manocha, D., and Mansard, N. (2018). An efficient acyclic contact planner for multiped robots. *IEEE Transactions on Robotics*, 34(3):586–601.

[Toussaint, 2015] Toussaint, M. (2015). Logic-geometric programming: An optimization-based approach to combined task and motion planning. In *International Joint Conference on Artificial Intelligence, IJCAI* .

[Toussaint et al., 2018] Toussaint, M., Allen, K. R., Smith, K. A., and Tenenbaum, J. B. (2018). Differentiable physics and stable modes for tool-use and manipulation planning. In *Robotics: Science and Systems XIV RSS*.

[Toussaint et al., 2020] Toussaint, M., Ha, J.-S., and Driess, D. (2020). Describing physics for physical reasoning: Force-based sequential manipulation planning. *IEEE Robotics and Automation Letters*.

[Toussaint and Lopes, 2017] Toussaint, M. and Lopes, M. (2017). Multi-bound tree search for logic-geometric programming in cooperative manipulation domains. In *Int. Conf. on Robotics and Automation, ICRA*.

[Toussaint et al., 2023] Toussaint, M., Ortiz-Haro, J., Hartmann, V., Karpas, E., and Hoenig, W. (2023). Effort level search in infinite completion trees with application to task-and-motion planning.

[Tuisov and Katz, 2021] Tuisov, A. and Katz, M. (2021). The fewer the merrier: Pruning preferred operators with novelty. In *International Joint Conference on Artificial Intelligence*.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Vega-Brown and Roy, 2020] Vega-Brown, W. and Roy, N. (2020). Asymptotically optimal planning under piecewise-analytic constraints. In *Algorithmic Foundations of Robotics XII*, pages 528–543. Springer.

[Veličković et al., 2017] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. (2017). Graph attention networks. *arXiv preprint arXiv:1710.10903*.

[Wells et al., 2019] Wells, A. M., Dantam, N. T., Shrivastava, A., and Kavraki, L. E. (2019). Learning feasibility for task and motion planning in tabletop environments. *IEEE robotics and automation letters*, 4(2):1255–1262.

[Williams and Ragno, 2007] Williams, B. C. and Ragno, R. J. (2007). Conflict-directed a$^*$ and its role in model-based embedded systems. *Discret. Appl. Math.*, 155(12):1562–1595.

[Winkler et al., 2018] Winkler, A. W., Bellicoso, C. D., Hutter, M., and Buchli, J. (2018). Gait and trajectory optimization for legged systems through phase-based end-effector parameterization. *IEEE Robotics and Automation Letters*, 3(3):1560–1567.

[Xie et al., 2019] Xie, A., Ebert, F., Levine, S., and Finn, C. (2019). Improvisation through physical understanding: Using novel objects as tools with visual foresight. In *Proc. of Robotics: Science and Systems (R:SS)*.

[Yao et al., 2019] Yao, W., Bandeira, A. S., and Villar, S. (2019). Experimental performance of graph neural networks on random instances of max-cut. In *Wavelets and Sparsity XVIII*, volume 11138, pages 242–251. SPIE.

[Yuan et al., 2022] Yuan, W., Paxton, C., Desingh, K., and Fox, D. (2022). Sornet: Spatial object-centric representations for sequential manipulation. In *Conference on Robot Learning*, pages 148–157. PMLR.

[Zhang et al., 2020] Zhang, Z., Wu, F., and Lee, W. S. (2020). Factor graph neural networks. *Advances in Neural Information Processing Systems*, 33:8577–8587.

[Zhao et al., 2021] Zhao, Z., Zhou, Z., Park, M., and Zhao, Y. (2021). Sydebo: Symbolic-decision-embedded bilevel optimization for long-horizon manipulation in dynamic environments. *IEEE Access*.

[Zilberstein, 2008] Zilberstein, S. (2008). Metareasoning and bounded rationality. *AAAI Workshop - Technical Report*.

[Zimmermann et al., 2020] Zimmermann, S., Hakimifard, G., Zamora, M., Poranne, R., and Coros, S. (2020). A multi-level optimization framework for simultaneous grasping and motion planning. *IEEE Robotics and Automation Letters*, 5(2):2966–2972.

# Appendix A.

# Complete List of Publications

## Journal Papers

1. Ortiz-Haro, J., Karpas, E., Katz, M., and Toussaint, M. (2022). A Conflict-Driven Interface Between Symbolic Planning and Nonlinear Constraint Solving. IEEE Robotics and Automation Letters, 7(4), (pp. 10518-10525).

## Conferences Papers

1. Hartmann, V. N., Ortiz-Haro, J., and Toussaint, M. (2023). Efficient Path Planning In Manipulation Planning Problems by Actively Reusing Validation Effort. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS).
2. Ortiz-Haro, J., Ha, J. S., Driess, D., Karpas, E., and Toussaint, M. (2023). Learning Feasibility of Factored Nonlinear Programs in Robotic Manipulation Planning. IEEE International Conference on Robotics and Automation (ICRA) (pp. 3729-3735).
3. Braun, C. V., Ortiz-Haro, J., Toussaint, M., and Oguz, O. S. (2022). Rhh-lgp: Receding Horizon and Heuristics-Based Logic-Geometric Programming for Task and Motion Planning. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 13761-13768).
4. Kamat, J., Ortiz-Haro, J., Toussaint, M., Pokorny, F. T., and Orthey, A. (2022). Bitkomo: Combining Sampling and Optimization for Fast Convergence in Optimal Motion Planning. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 4492-4497).
5. Ortiz-Haro, J., Karpas, E., Toussaint, M., and Katz, M. (2022). Conflict-Directed Diverse Planning for Logic-Geometric Programming. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 32, pp. 279-287).

6. Hoenig, W., Ortiz-Haro, J., and Toussaint, M. (2022). db-A*: Discontinuity-Bounded Search for Kinodynamic Mobile Robot Motion Planning. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 13540-13547).
7. Ortiz-Haro, J., Ha, J. S., Driess, D., and Toussaint, M. (2022). Structured Deep Generative Models for Sampling on Constraint Manifolds in Sequential Manipulation. In Conference on Robot Learning (pp. 213-223). PMLR.
8. Ortiz-Haro, J., Hartmann, V. N., Oguz, O. S., and Toussaint, M. (2021). Learning Efficient Constraint Graph Sampling for Robotic Sequential Manipulation. IEEE International Conference on Robotics and Automation (ICRA) (pp. 4606-4612).

## Preprints

1. Ortiz-Haro, J., Hoenig, W., Hartmann, V., and Toussaint, M. (2023). iDb-A*: Iterative Search and Optimization for Optimal Kinodynamic Motion Planning. Submitted to IEEE Transactions on Robotics (T-RO).
2. Moldagalieva, A., Ortiz-Haro, J., Toussaint, M., and Hoenig, W. (2023). db-CBS: Discontinuity Bounded Conflict-Based Search for Multi-Robot Kinodynamic Motion Planning. arXiv preprint arXiv:2309.16445. Submitted to ICRA.
3. Grote, P., Ortiz-Haro, J., Toussaint, M., and Oguz, O. S. (2023). Neural Field Representations of Articulated Objects for Robotic Manipulation Planning. arXiv preprint arXiv:2309.07620. Submitted to ICRA.
4. Wahba, K., Ortiz-Haro, J., Toussaint, M., and Hoenig, W. (2023). Kinodynamic Motion Planning for a Team of Multirotors Transporting a Cable-Suspended Payload in Cluttered Environments. arXiv preprint arXiv:2310.03394. Submitted to ICRA.
5. Toussaint, M., Ortiz-Haro, J., Hartmann, V., Karpas, E., and Hoenig, W. (2023). Effort Level Search in Infinite Completion Trees with Application to Task-and-Motion Planning. Submitted to ICRA.
6. Levit, S., Ortiz-Haro, J., and Toussaint, M. (2023). Solving Sequential Manipulation Puzzles by Finding Easier Subproblems. Submitted to ICRA.

## Workshop Papers

1. Rehberg, W., Ortiz-Haro, J., Toussaint, M., and Hoenig, W. (2023). Comparison of Optimization-Based Methods for Energy-Optimal Quadrotor Motion Planning. arXiv preprint arXiv:2304.14062. Aerial Robotics Workshop ICRA.

2. Hoenig, W., Ortiz-Haro, J., and Toussaint, M. (2022). Benchmarking Sampling-, Search-, and Optimization-based Approaches for Time-Optimal Kinodynamic Mobile Robot Motion Planning. Motion Planning Workshop IROS.

### Master's and Bachelor's Theses

1. Weingart, A. (2023) Efficient Kinodynamic Motion Planning with Reinforcement Learning Policies. Master's Thesis in Computer Science (TU-Berlin). Co-supervision: Ortiz-Haro, J., and Hoenig, W.
2. Groete, P. (2023) Neural Scene Representations for Sequential Reasoning. Master's Thesis in Computer Science (TU-Berlin). Co-supervision: Ortiz-Haro, J., and Oguz, O.
3. Rehberg, W. (2022) SCP and k-Order Motion Optimization for Cooperative Multirotor Teams. Master's Thesis in Computer Science (TU-Berlin). Co-supervision: Ortiz-Haro, J., and Hoenig, W.
4. Kamat, J. (2022) Combining Sampling and Optimization for Optimal Motion Planning. Master's Thesis in Mathematics (BITS and TU-Berlin). Co-supervision: Ortiz-Haro, J., and Orthey, A.
5. Oedi, P. (2022) Constrained Sampling - A Study on Methods for Sampling from Constraint Manifolds. Master's Thesis in Computer Science (TU-Berlin). Co-supervision: Ortiz-Haro, J., and Ha, J.