Fast Multi-query Planning in Graphs of Convex Sets

by

Savva Morozov

S.B., Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2025

Authored by: Savva Morozov
Department of Electrical Engineering and Computer Science
January 24, 2025

Certified by: Russell L. Tedrake
Professor of Electrical Engineering and Computer Science,
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Fast Multi-query Planning in Graphs of Convex Sets

by

Savva Morozov

Submitted to the Department of Electrical Engineering and Computer Science
on January 24, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

ABSTRACT

Planning in Graphs of Convex Sets (GCS) is a recently developed optimization framework that seamlessly integrates discrete and continuous decision making. It naturally models and effectively solves a wide range of challenging planning problems in robotics, including collision-free motion planning, skill chaining, and control of hybrid systems. In this thesis, we study the multi-query extension of planning through GCS, motivated by scenarios where robots must operate swiftly within static environments. Our objective is to precompute optimal plans between predefined sets of source and target conditions, in an effort to enable fast online planning and reduce GCS solve times.

Our solution consists of two stages. Offline, we use semidefinite programming to compute a coarse lower bound on the problem's cost-to-go function. Then, online, this lower bound is used to incrementally generate feasible plans by solving short-horizon convex programs. We demonstrate the effectiveness of our approach through a variety of experimental domains: collision-free motion planning for a warehouse robot arm, item sorting for a top-down suction gripper, and footstep planning for a bipedal walker. In particular, in a warehouse-like scenario involving a seven-joint robot arm, our method generates higher-quality paths up to 100 times faster than existing motion planners.

Thesis supervisor: Russell L. Tedrake
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Before we proceed, I want to express my heartfelt gratitude to and acknowledge those without whom this work would not have been possible.

First, I want to thank my advisor, Russ Tedrake, for his invaluable guidance and intellectual inspiration; for teaching me the importance of precision and clarity in thinking and communication; for motivating me to formulate my questions and problems with rigor, to strip away the fog of hype and ambiguity, and to ground my ideas in specific and precise terms; for showing me the value of deep understanding over superficial familiarity; for encouraging me to seek connections with adjacent fields and contextualize my work in the broader body of research. Thank you, Russ, for this unique opportunity to spend my days immersed in textbooks, tinkering with code, and wrestling with ideas over a piece of paper, surrounded by the smartest people I know.

I am also deeply grateful to Pablo Parrilo for the invaluable meetings we've had with him and Russ. Pablo's ability to draw connections and his incredible depth of mathematical insight have been a source of great intellectual inspiration. The meetings we had with Pablo and Russ were transformative, challenging me to think more deeply and communicate more effectively. Thank you, Russ and Pablo, for the numerous discussions we had, where we spent hours dissecting the second slide out of a thirty-slide presentation, because I had overlooked the core insights hidden in the details. These moments were humbling and profoundly instructive, and I'm incredibly grateful for your patience and wisdom.

I also owe special thanks to Tobia Marcucci for his guidance in thinking, writing, and presenting ideas. Thank you for helping me sharpen my communication, articulate my thoughts clearly, and generally find the right way to convey and think about ideas, drawing meaningful connections with existing research. Thank you for not letting me settle for "eh, good enough" and spending hours helping me find just the right words. Your insistence on precision and excellence has been invaluable.

To Bernhard Paus Græsdal and Alex Amice, thank you for our productive discussions, for your thoughtful feedback, and curious, stimulating conversations. You have been instrumental in shaping this work. To Nick, David, and Terry, thank you for your insights about life, love, research, work, and meaning. And to my other labmates — Pang, Jack, Max, Tommy, Lu, Boyuan, Lirui, Adam, Abhinav, Shao, and Rebecca — thank you for your conversations and camaraderie.

To my parents, Masha and Aleksei, thank you for supporting me throughout my life and for encouraging all the challenging ideas I've pursued. Your fascination with my work, even when it involves something as seemingly simple as moving and rotating boxes, means the world to me. To my brother Steve, thank you for our engaging discussions about research

and for sharing your intellectual curiosity and inspiring mine. And to the rest of my family, thank you for your unwavering love and support.

To my sweet wife Mary, I love you more than I can express with words. My work, my research, and this entire journey are driven by my desire to be a better husband and a better provider for you and for our future children, my love. You are the reason for my determination and the heart of my happiness.

Finally, I give thanks to Jesus Christ, our Lord, to our Heavenly Father, and to the Holy Spirit, for the gift of life, for the gift of purpose, and the opportunity to pursue this work. Ultimately, this effort is for Your glory: this work, at its core, is an act of service to You. By striving to do good research, I seek to serve You.

Contents

Chapter 1

Introduction

1.1 Motivation

A Graph of Convex Sets (GCS) is a recently developed generalization of a directed graph where each vertex is paired with a convex set and a convex cost function [1]. Traversing a GCS involves selecting a point from the convex set at each vertex and incurring a corresponding convex cost, while edges impose additional convex costs and constraints that couple adjacent vertex points. Many problems in classical graph theory, such as the shortest-path problem, the traveling-salesman problem, the minimum-spanning-tree problem, and others, have been extended to the GCS [2]. Among these, the Shortest-Path Problem (SPP) in GCS [1] has received particular attention due to its applications in robotics. The objective is to find a discrete path through the graph, together with the continuous vertex points along this path, that minimize the cumulative edge and vertex costs. The SPP in GCS naturally models problems where discrete and continuous decision-making are interleaved, making it a powerful tool for robotics applications, including optimal control [1], collision-free motion planning [3–5], planning through contact [6], and other problems [7, 8]. Prior work [9–11] also highlighted that searching for a path (a sequence of distinct vertices) in a GCS can be limiting. Allowing vertex revisits naturally models many planning problems in robotics where actions, skills, or behaviors must be repeated. This is formalized in the Shortest-Walk Problem (SWP) in GCS, where the vertices may be repeated and different points can be selected upon each vertex visit. Together, we refer to the shortest-path and the shortest-walk problems in GCS as *planning in GCS*.

While planning in GCS is NP-hard [1, Section 9.2], effective solution methods have been proposed in [1, 11]. However, despite its versatility, planning in GCS can be too slow for real-time applications on high-dimensional robotic systems. As a motivating example, consider a 7-DoF KUKA iiwa robot arm, repeatedly performing online motion planning in a static environment. When the environment is simple, the GCS is small, and the planning queries can be solved quickly, in under 50ms [3]. However, when the environment is complex and the configuration space must be covered thoroughly, as in Fig. 1.1, the GCS becomes large, and the planning queries can take up to 600ms. This is not practical for high-productivity applications, such as robot arms in warehouses, where the company's income is nearly proportional to the operational speed.

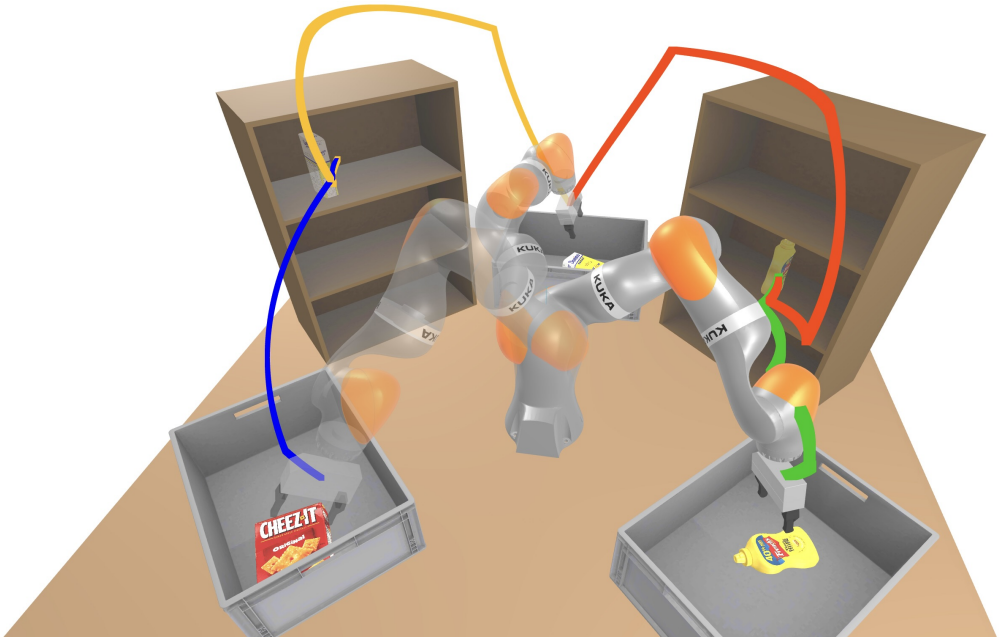


Figure 1.1: Robotic arm in a simulated environment, tasked with moving items between shelves and bins. Shown are four queries for collision-free motion planning.

1.2 Statement of work

In an effort to reduce solve times for online shortest-path and shortest-walk queries in GCS, in this thesis, we seek an efficient way of precomputing optimal plans between given sets of source and target conditions. We formulate this problem as a generalization of planning in GCS that is analogous to the all-pairs and many-to-many generalizations of the classical SPP in an ordinary graph [12, Ch. 23]. Our solution contains two phases. Offline, we solve a semidefinite program that synthesizes a piecewise-convex-quadratic lower bound on the problem's cost-to-go function, with each convex quadratic piece defined over the convex set associated with each GCS vertex. We derive tractable cost-to-go synthesis programs tailored to both the shortest paths and the shortest walks in GCS. Then, online, we use this cost-to-go lower bound to guide an incremental search algorithm to quickly retrieve a path or a walk. Specifically, we use a multi-step lookahead greedy policy, guided by the cost-to-go lower bound, to determine the next vertex to visit. A path or a walk are thus obtained incrementally, one vertex at a time. Convexity of quadratic lower bounds at each vertex allows us to evaluate this greedy policy by solving a set of small convex programs in parallel, which can be done quickly and efficiently at run-time. Although the quadratic lower bound at each vertex can be coarse, using the lookahead policy is equivalent to producing a piecewise-quadratic lower bound over each vertex, which can be very expressive. As a result, the obtained plans are typically nearly optimal in practice. Additionally, provably-optimal plans can be obtained using the generalization of the classical A* search algorithm [11], at the expense of extra computational cost. Applied to the complex scenario shown in Figure 1.1, our method requires just 6s of offline computation to produce the cost-to-go lower bounds. Subsequent online queries take 2-11ms, which is up to two orders of magnitude faster than solving a planning in GCS query

from scratch or obtaining plans via sampling-based competitors. Our experimental results highlight that our approach effectively produces high-quality solutions to a wide range of problems in robotics and control, including collision-free motion planning, skill chaining, and control for hybrid dynamical systems.

1.3 Organization

The remainder of this thesis is organized as follows.

In Chapter 2, we review the SPP and the SWP in GCS. We begin with their classical counterparts in ordinary discrete graphs, define the GCS framework and the shortest-path and walk problems in a GCS, and highlight the similarities and differences between the two.

In Chapter 3, we introduce and study multi-query planning in GCS. We define the problem and present our two-stage solution: the offline cost-to-go synthesis phase, described in Section 3.4, and the online incremental search phase, described in Section 3.5.

In Chapter 4, we evaluate our framework through various numerical examples. We start with a simple 2D illustration to build intuition, then analyze the impact of the cost-to-go coarseness and greedy policy horizon on solution quality, and demonstrate the effectiveness of our approach on three challenging robotic systems.

Finally, we conclude by discussing the key takeaways and limitations in Chapter 5.

Chapter 2

Planning in Graphs of Convex Sets: Shortest Paths and Shortest Walks

We begin with a background chapter on planning in GCS. In Section 2.1, we review the classical SPP in a discrete graph and explain why the SWP is redundant in this context, as it reduces to the SPP. Next, in Section 2.2, we introduce the GCS optimization framework and formally define both the SPP and the SWP in GCS. In GCS, these two problems are meaningfully different: in Section 2.3, we highlight the differences and similarities between the two problems and discuss key properties that will be relevant for the subsequent chapters.

2.1 Shortest walks and paths in an ordinary graph

2.1.1 Walks and paths in a graph

Let $G = (\mathcal{V}, \mathcal{E})$ be a directed graph with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$. Given a source vertex s and a target vertex t, an *s-t walk* is a sequence of vertices $w = (v_0, \dots, v_K)$, such that $v_0 = s$, $v_K = t$, and each consecutive pair of vertices (v_k, v_{k+1}) is connected by an edge $e_k \in \mathcal{E}$. We define $\mathcal{E}_w = \big((v_0, v_1), \dots, (v_{K-1}, v_K)\big)$ as the sequence of edges traversed by w.

A *path* is a walk where all vertices must be distinct. We emphasize this difference between walks and paths: a walk may revisit vertices and edges, whereas a path cannot. Throughout the thesis, we generally use w to denote walks and p to denote paths. We define $\mathcal{W}_{s,t}$ to be the set of all s-t walks in G, and $\mathcal{P}_{s,t}$ to be the set of all s-t paths. We refer to a walk or a path that traverses $K+1$ vertices as a K-step walk or path. When speaking about walks, we will often add a superscript K to $\mathcal{W}^K_{s,t}$ to specifically denote the set of K-step s-t walks.

2.1.2 Classical Shortest-Path Problem (SPP)

Let us associate with every edge $e \in \mathcal{E}$ a non-negative edge cost $c_e \in \mathbb{R}_+$, and with every vertex $v \in \mathcal{V}$ a non-negative vertex cost $c_v \in \mathbb{R}_+$. A shortest path p between the vertices s and t minimizes the sum of the edge and vertex costs along the path:

$$\min_p \quad \sum_{v \in p} c_v + \sum_{e \in \mathcal{E}_p} c_e \quad \text{s.t.} \quad p \in \mathcal{P}_{s,t}.$$

The optimal value of this program is called the *cost-to-go* between s and t, and is denoted by $J^*_{s,t}$. *The principle of optimality* [13] holds in this context, stating that every subpath of a shortest path is itself a shortest path; this forms the foundation for many efficient solution algorithms to this problem.

2.1.3 Classical Shortest-Walk Problem (SWP)

Similarly to the shortest path, a shortest walk w between the vertices s and t minimizes the sum of the edge and vertex costs along the walk:

$$\min_{w} \quad \sum_{v \in w} c_v + \sum_{e \in \mathcal{E}_w} c_e \quad \text{s.t.} \quad w \in \mathcal{W}_{s,t}.$$

The shortest-walk problem is rarely defined in the literature, as it is redundant: it is well-known that for a graph with non-negative edge and vertex costs, this problem always admits an optimal solution that is a path. This is because revisiting a vertex creates a cycle of non-negative cost and makes no progress towards the target. This cycle can thus be removed without increasing the total cost of the walk. As such, we define the shortest-walk problem in an ordinary graph not because it is studied in its own right, but to mirror the exposition of analogous problems in the GCS context.

2.2 Shortest walks and paths in Graphs of Convex Sets

2.2.1 Graph of Convex Sets (GCS)

A GCS is a directed graph $G = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$ and edge set $\mathcal{E}$, where each vertex $v \in \mathcal{V}$ is paired with a compact, convex set $\mathcal{X}_v$ and a continuous, convex, non-negative cost function $l_v : \mathcal{X}_v \to \mathbb{R}_+$. Similarly, each edge $e = (u, v) \in \mathcal{E}$ is also paired with a convex set $\mathcal{X}_e \subseteq \mathcal{X}_u \times \mathcal{X}_v$ and a continuous, convex, non-negative cost function $l_e : \mathcal{X}_e \to \mathbb{R}_+$. When traversing a GCS, we must select a point $x_v \in \mathcal{X}_v$ upon a visit to vertex v and incur the cost $l_v(x_v)$. When moving along an edge $e = (u, v)$, the adjacent points (x_u, x_v) must satisfy the constraint $(x_u, x_v) \in \mathcal{X}_e$, and we incur the edge cost $l_e(x_u, x_v)$ [2].

2.2.2 Walks and paths in a GCS

Let s, t be a pair of source and target vertices, and let $\bar{x}_s \in \mathcal{X}_s$, $\bar{x}_t \in \mathcal{X}_t$ be a pair of source and target points in the corresponding source and target convex sets. Let $w = (v_0, \ldots, v_K) \in \mathcal{W}_{s,t}$ be a K-step s-t walk through the graph G. We denote a *trajectory that corresponds to a walk* w to be a sequence of vertex points $\tau = (x_0, \ldots, x_K)$ that satisfies the following constraints:

$$x_0 = \bar{x}_s, \; x_K = \bar{x}_t, \tag{2.1a}$$

$$(x_{k-1}, x_k) \in \mathcal{X}_{e_k}, \qquad \forall k = 1, \ldots, K. \tag{2.1b}$$

In words, we require that the trajectory start at point $\bar{x}_s$, end at point $\bar{x}_t$, and that each consecutive pairs of points (x_{k-1}, x_k) must lie in the corresponding edge constraint set $\mathcal{X}_{e_k}$,

where $e_k = (v_{k-1}, v_k)$. Note that the assumption $\mathcal{X}_{e_k} \subseteq \mathcal{X}_{v_{k-1}} \times \mathcal{X}_{v_k}$ also ensures that the relevant vertex constraints $x_k \in \mathcal{X}_{v_k}$ are satisfied along the walk. Given a walk w and a pair of source and target points $\bar{x}_s, \bar{x}_t$, we denote the set of trajectories τ that satisfy (2.1a) and (2.1b) as $\mathcal{T}_w(\bar{x}_s, \bar{x}_t)$.

We refer to the tuple (w, τ) as a *walk in a GCS*: it is a walk w through the graph G, paired with a sequence of corresponding vertex points τ that satisfies the edge and vertex constraints of the GCS. Note that, just as vertices and edges in a walk may be revisited, distinct continuous points may be selected when revisiting the same vertex.

We denote with $l(w, \tau)$ the sum of the edge and vertex costs along the walk (w, τ) and refer to it as the cost of the walk in the GCS:

$$l(w, \tau) = \sum_{k=0}^{K} l_{v_k}(x_k) + \sum_{k=1}^{K} l_{e_k}(x_{k-1}, x_k).$$

Finally, a *path in a GCS* is defined similarly: it is a tuple (p, τ), where $p \in \mathcal{P}_{s,t}$ and $\tau \in \mathcal{T}_p(\bar{x}_s, \bar{x}_t)$, with the only distinction being that the vertices and edges along the path cannot be revisited. Since every path in a GCS is also a walk in a GCS, all definitions above introduced for walks naturally extend to paths as well.

2.2.3 Shortest-Path Problem in a GCS

The shortest s-t path in a GCS between points $\bar{x}_s$ and $\bar{x}_t$ is a path of minimal cost. It is the solution to the following optimization problem:

$$\min_{(p,\tau)} \quad l(p, \tau) \tag{2.2a}$$

$$\text{s.t.} \quad p \in \mathcal{P}_{s,t}, \quad \tau \in \mathcal{T}_p(\bar{x}_s, \bar{x}_t). \tag{2.2b}$$

The optimal solution to (2.2) is a tuple (path p and trajectory τ). We denote the optimal objective value of (2.2) as $J^*_{s,t}(\bar{x}_s, \bar{x}_t)$ and refer to it as the *path cost-to-go* from point $\bar{x}_s$ of vertex s to point $\bar{x}_t$ of vertex t.

2.2.4 Shortest-Walk Problem in a GCS

Similarly, the shortest s-t walk in a GCS between points $\bar{x}_s$ and $\bar{x}_t$ is a walk of minimal cost. There are subtle differences between searching for a shortest walk and a shortest path in a GCS. To help us emphasize these differences, we define the shortest walk to be the solution to the following, slightly different optimization problem:

$$\inf_{K} \min_{(w,\tau)} \quad l(w, \tau) \tag{2.3a}$$

$$\text{s.t.} \quad w \in \mathcal{W}^K_{s,t}, \quad \tau \in \mathcal{T}_w(\bar{x}_s, \bar{x}_t). \tag{2.3b}$$

We have a two-level optimization here: we seek the shortest K-step walk (w, τ) at the inner level, and take the infimum over K at the outer level. We thus optimize over the discrete number of steps K, the walk through the graph, and the trajectory along this walk.

The reason for the subtle difference in the definition of the shortest-walk and shortest-path problems in GCS is that a finite shortest walk may not actually exist. Indeed, there are simple GCS instances where no finite walk is optimal, but the infimum in the optimization problem (2.3) does exist in the limit. This is illustrated in Section 2.3.2. Even if the optimal walks are infinite, the optimization problem (2.3) remains well-defined. We denote the optimal objective value of (2.3) as $J'_{s,t}(\bar{x}_s, \bar{x}_t)$ and refer to it as the *walk cost-to-go* from point $\bar{x}_s$ of vertex s to point $\bar{x}_t$ of vertex t. We emphasize this difference in notation: J' denotes the shortest-walk cost-to-go, while J^* represents the shortest-path cost-to-go.

2.3 Key properties, differences, and similarities

We now review the key properties of the shortest paths and shortest walks in GCS that are relevant to the subsequent chapters.

2.3.1 Shortest walks need not be paths

As discussed in Section 2.1.3, for an ordinary graph with non-negative edge and vertex costs, the SWP reduces to the SPP because revisiting a vertex adds a non-negative cycle and makes no progress towards the target. The same is not true in a GCS. When traversing a GCS, revisiting the same vertex can be advantageous, as demonstrated by the following example.

Example 1. Consider the simple 2D problem depicted in Figure 2.1a. This GCS has 5 vertices $\mathcal{V} = \{s, a, b, c, t\}$ and 8 edges, drawn in green. The convex sets $\mathcal{X}_s, \mathcal{X}_c, \mathcal{X}_t$ are points, $\mathcal{X}_b$ is a segment, and $\mathcal{X}_a$ is a square. For every edge $e = (u, v)$, the edge cost is defined as $l_e(x_u, x_v) = 1 + ||x_u - x_v||_2^2$: the first term penalizes the number of steps taken, while the squared displacement term penalizes the size of each step. There are no vertex costs, nor are there any additional edge constraints. The solutions to the SPP and the SWP in this GCS are shown in Figures 2.1b and 2.1c respectively. To avoid revisiting vertices, the shortest path (orange) is forced to take larger steps, incurring large penalties. By taking smaller steps and revisiting vertices, the shortest walk (blue) achieves a lower cost. Thus, though cycles in the GCS still have a non-negative cost, they can actually make progress towards the target.

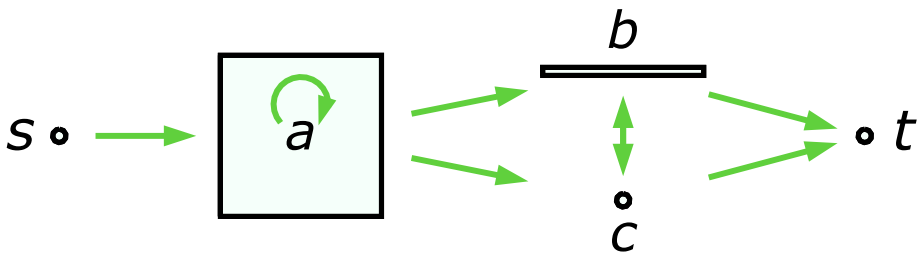
2.3.2 Shortest walks need not be finite

Consider again the GCS in Example 1. If the edge cost was $l_e(x_u, x_v) = ||x_u - x_v||_2^2$, then the optimal walk would involve taking infinitely many small steps through the vertex a. As a result, no finite walk is optimal, though the solution does exist in the limit. This issue is the reason for the infimum over the number of steps K in program (2.3).

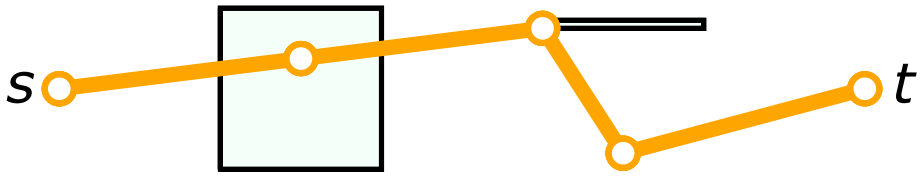
The following is a simple sufficient condition for the optimal walk, if one exists, to be finite:

$$\min\{l_e(x_u, x_v) \mid (x_u, x_v) \in \mathcal{X}_e\} > 0, \tag{2.4}$$

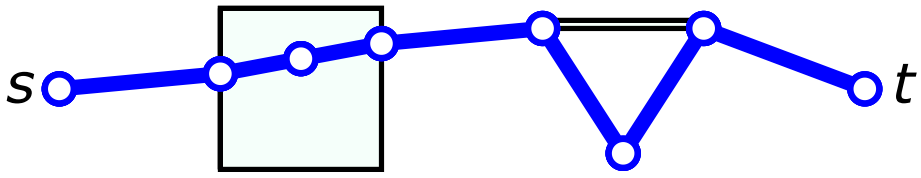
for every edge $e = (u, v) \in \mathcal{E}$. This condition ensures that the cost of every step in the walk is bounded below by some positive value; thus an infinite walk must incur infinite cost, and cannot be optimal. For practical purposes, this condition can be easily satisfied by adding



(a) A GCS embedded in $\mathbb{R}^2$.



(b) The shortest path between vertices s and t (orange) costs 35.



(c) The shortest walk between vertices s and t (blue) costs 28. Vertices a and b are both revisited along the walk: a is visited three times consecutively, and b is visited twice non-consecutively.

Figure 2.1: A GCS where the shortest walk is not a path.

a small $\epsilon > 0$ to every edge cost l_e. In what follows, we assume that the condition above holds; this assumption is both reasonable and non-intrusive, especially since we are targeting robotics applications.

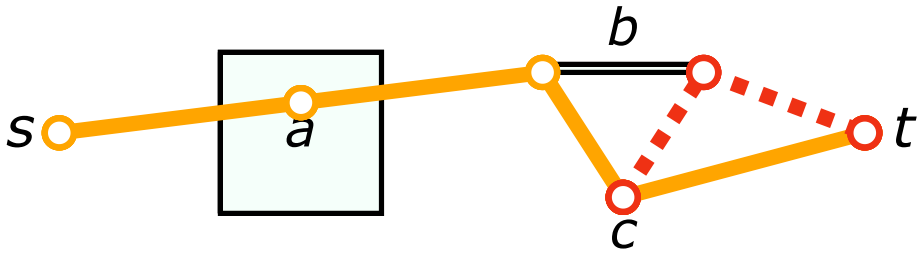
Assumption 1. We assume that the condition in (2.4) holds, ensuring that the shortest-walk problem, if feasible, has a finite optimal solution.

2.3.3 Principle of optimality

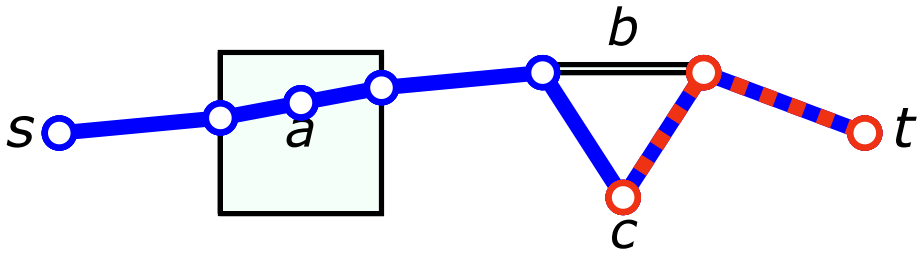
For the classical SPP, the principle of optimality holds and the optimal policy is independent of past decisions, which simplifies the problem and enables many efficient solution algorithms. While this property holds for the SWP in GCS, it does not hold for the SPP in GCS.

Example 2. Consider again the GCS instance from Example 1, pictured in Figure 2.1a.

Due to the constraint that vertices cannot be revisited, the principle of optimality does not hold for the SPP in GCS. This is demonstrated in Fig. 2.2a, where we plot the optimal s-t path in orange and the optimal c-t path in red. The c-t subpath of the optimal s-t path (orange) is not the optimal c-t path (red), and thus the principle of optimality does not hold. We also observe that the optimal policy for the SPP in GCS is a function of the set of



(a) For the shortest paths, the optimal policy at vertex c depends on previous decisions. If b has been visited already, the optimal decision is to go to t (orange), otherwise it is to go to b (red). Principle of optimality does not hold: c-t subpath of the optimal s-t path is not itself optimal.



(b) For the shortest walks, the optimal policy is independent of previous decisions. Shown are the optimal s-t (blue) and c-t (red) solutions to the relaxed problem. Principle of optimality holds: c-t subwalk of the optimal s-t walk is itself optimal.

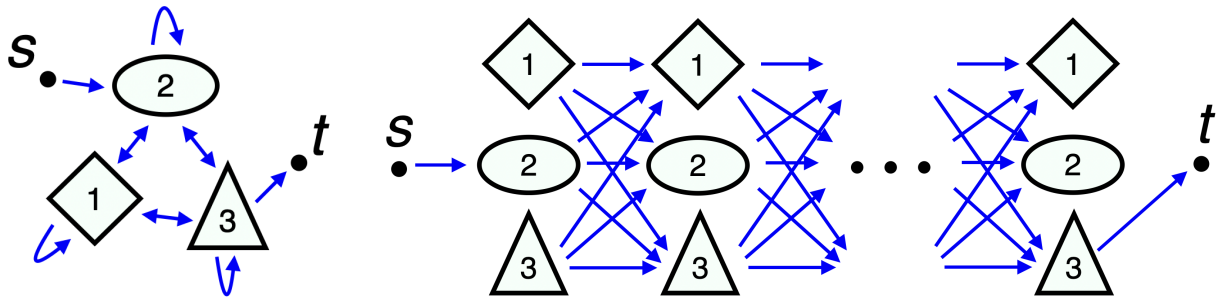
Figure 2.2: Principle of optimality holds for the shortest walks but not the shortest paths.

previously visited vertices. Consider the optimal decision at vertex c: if b was visited before, the optimal decision is to go to t (orange), otherwise the optimal decision is to go to b (red). The optimal decision policy thus depends on the set of previously visited vertices.

For the shortest walks in GCS, the principle of optimality holds, stating that every subwalk of the optimal walk is itself optimal. Indeed, if a subwalk was not itself optimal, then it could be replaced with the actual optimal subwalk, resulting in a walk of lower cost than the original shortest walk — a contradiction. This is visualized in Fig. 2.2b, where the optimal s-t and c-t solutions to the SWP in GCS are shown in blue and red respectively. Since the principle of optimality holds, the optimal decision policy is independent of the history of previously visited vertices.

2.3.4 Computational complexity

Both SPP and SWP in GCS are NP-hard. In [2, §9.2], the authors reduced the well-known NP-complete 3SAT problem [14] to the shortest path in an acyclic GCS, thus proving that SPP in GCS is NP-hard. This reduction proves that the shortest walks in GCS are NP-hard as well. Indeed, since every walk is a path in an acyclic graph, the SPP and the SWP in an acyclic GCS are equivalent, and the NP-hardness of the shortest-walk problem follows.



(a) (b)

Figure 2.3: A K-step optimal walk in the GCS in (a) corresponds to a K-step optimal path in the layered GCS in (b), constructed by duplicating the original vertices across $K-1$ layers. Thus, the SWP in GCS in (a) can be solved via SPP queries over progressively larger GCS instances in (b) as $K \to \infty$.

2.3.5 Existing solution methods

Given that both SPP and SWP in GCS are NP-hard, it is unlikely that there is an efficient (polynomial-time) solution to either of them. For this reason, existing methods either yield heuristic solutions or have exponential solve time.

The SPP in GCS can be reformulated as a Mixed-Integer Convex Program (MICP) with a strong convex relaxation [1]: using a rounding strategy from [3], this relaxation often yields near-optimal solutions. This particular formulation has proven to be very effective in practice and has been implemented in the software packages `Drake` [15] and `gcspy`.

Unlike the shortest paths in GCS, the optimization tools for the shortest walks are not as strong. Fundamentally, this has to do with the fact that length of an optimal walk can be arbitrarily long and can be difficult to guess from the problem statement. Thus, to compute the solution (w, τ) via a single optimization problem, one would need to pre-allocate a fixed number of decision variables and a fixed amount of memory, which inherently requires guesswork. For this reason, solution methods for shortest walks are typically either enumerative or incremental in nature.

A naive enumerative way to find the shortest walk in a GCS is to compute K-step optimal walks for progressively larger K and maintain the best solution. As $K \to \infty$, this converges to the infimum in (2.3). A K-step optimal walk can be obtained by solving an SPP in a different GCS, where we duplicate vertices as a way to allow revisits, as shown in Figure 2.3. Given the original GCS in Figure 2.3a, we construct a new layered GCS in Figure 2.3b, with each layer containing duplicates of the original vertices, and consecutive layers connected based on the original edges. Solving the SPP in this layered GCS yields a K-step optimal path, equivalent to a K-step optimal walk in the original GCS. This layered construction was considered [2, §10.2.3], where it was shown to be computationally expensive. Moreover, for the shortest walks, this approach is intractable as it requires solving SPP queries over increasingly larger GCS instances.

Incremental search methods provide a unified framework for solving both the SPP and SWP in GCS [9–11, 16]. The authors in [11] generalize the well-known A* algorithm to the GCS, employing a heuristic approximation of the cost-to-go function to guide the search.

When a good heuristic is available, greedy search with backtracking [9] is enough to quickly produce effective heuristic solutions. In this thesis, we use incremental search to solve multi-query planning in GCS, building on our earlier work [9] that was originally published in the proceedings of the 16th international Workshop for Algorithmic Foundations of Robotics. The performance of incremental search methods heavily depends on the quality of the guiding heuristic; in this thesis, we develop effective heuristics tailored to both shortest paths and walks in GCS.

Chapter 3

Multi-query planning in Graphs of Convex Sets

In this chapter, we introduce and study the problem of multi-query planning in GCS. In Section 3.1, we formally define multi-query SPP and SWP in GCS. We formulate these problems by generalizing the classical all-pairs SPP, which seeks shortest paths between every pair of vertices in an ordinary graph. We review relevant literature in Section 3.2 and outline our solution framework in Section 3.3, extending the classical all-pairs SPP methodology to the GCS setting. Our approach has two phases: an offline phase (Section 3.4), where we precompute lower bounds on the cost-to-go function over the GCS, and an online phase (Section 3.5), where solutions are recovered incrementally through graph search.

3.1 Problem statement

Let $\mathcal{S} \subset \mathcal{V}$ be a set of source vertices and $\mathcal{T} \subset \mathcal{V}$ be a set of target vertices. The goal of multi-query SWP in GCS is to find the shortest walks between every pair of source and target points $\bar{x}_s \in \mathcal{X}_s$ and $\bar{x}_t \in \mathcal{X}_t$, and every pair of source and target vertices $s \in \mathcal{S}$ and $t \in \mathcal{T}$. Analogously, the goal of multi-query SPP in GCS is to find the shortest paths between every pair of source and target points and vertices. Note that since both the shortest-walk and the shortest-path problems in GCS are NP-hard, their multi-query generalizations are at least NP-hard as well.

3.2 Related literature

Multi-query generalizations of the classical SPP, such as the many-to-many SPP, which seeks shortest paths between specified sets of source and target vertices, and the all-pairs SPP[1], which seeks shortest paths for every vertex pair in the graph, are foundational in graph theory and algorithms research [12, Ch. 23]. The widespread applicability of these problems has led to a rich body of work focused on their solutions.

[1]In the literature, the acronym APSP is more commonly used for the All-Pairs Shortest Path problem; to reduce the number of unique acronyms in this thesis, we use *all-pairs SPP* instead of APSP.

At the core of these solutions is the Bellman's principle of optimality [17], which states that every subpath of a shortest path is itself a shortest path. This principle enables the use of recursive methods for solving the classical SPP. The shortest path from a vertex s to a target t can be determined recursively, by evaluating the cost of a shortest path from each successor v of vertex s, and greedily selecting the best one among them. Thus, the cost of the shortest paths between pairs of vertices, known as the cost-to-go function, is sufficient for optimal decision making. This recursive relationship is encapsulated in the celebrated Bellman equation, which characterizes the optimality condition on this problem's cost-to-go function. The cost-to-go can be computed efficiently by solving the Bellman equation recursively via dynamic programming [18].

Unfortunately, explicit solutions to the Bellman equation exist only in a handful of contexts. In the purely discrete graph setting, efficient algorithms such as Floyd-Warshall [19, 20] Johnson's algorithm [21], and others [22] compute the cost-to-go function into a matrix, often referred to as the distance oracle [23]. Despite their efficiency, the storage requirement for the cost-to-go grows quadratically with the graph size (while the computation time is roughly cubic in graph size), making these methods impractical for large graphs. For this reason, approximate distance oracles and the shortest paths have been studied extensively [24, 25]; see [22, p. 5.3.2] for a recent review.

For continuous-space systems, a notable example where the exact solution is known is the Linear Quadratic Regulator (LQR) in control theory [26, 27]. When the system has linear dynamics and the cost function is quadratic, the cost-to-go function is quadratic and can be computed by solving the famous algebraic Riccati equation [28–31]. When linear constraints are placed on the system's control inputs, the problem is known as the explicit Model Predictive Control (MPC), and the cost-to-go is this setting is known to be piecewise quadratic [32]. However, the number of pieces grows exponentially with the number of linear constraints and the horizon of the problem, and is thus intractable for all but the simplest scenarios [33].

For general continuous-space control systems, solving the Bellman equation is intractable. This has motivated extensive research into approximate methods for computing the cost-to-go.

Linear programming-based approaches approximate the cost-to-go function by solving an optimization problem over a finite set of points, thus discretizing the continuous space [34–37]. However, these methods suffer from the curse of dimensionality: for the cost-to-go description to be effective, the number of discretization points typically grows exponentially with the dimension of the configuration space and environment's complexity [38].

Other techniques utilize functional approximations over the continuous space. Among these, neurodynamic programming represents an influential class of techniques, approximating the cost-to-go function using neural networks. The literature on this subject is vast; we refer the reader to [39] and [40] for classic textbooks. Leveraging the expressive power of neural networks, these methods provide tractable solutions for high-dimensional continuous spaces. Effective methods for approximately enforcing the Bellman equation in these setting have been developed, iteratively refining the approximations; see [41, p. 6.3][42–45] for a handful of classic techniques. While effective, these approaches often demand meticulous hyperparameter tuning and are notably sensitive to a multitude of factors like initialization, network architecture, training schedules, and specific application setting.

Semidefinite programming (SDP) and sum-of-squares (SOS) techniques offer a principled

convex framework for constructing polynomial approximations to the cost-to-go function [46–48]. Relatedly, a substantial body of work in the controls literature has employed SOS-based methods for verifying and synthesizing Lyapunov functions [49–52] and control-barrier functions [53, 54]. Although polynomials are much less expressive than neural networks, they are computationally tractable and robust. The approach proposed in this paper uses similar techniques, extending their applicability to Graphs of Convex Sets (GCS). Specifically, we develop a computationally efficient framework for approximating the cost-to-go function over GCS, enabling the solution of multi-query SPP and SWP in this setting.

3.3 Solution outline

We propose a general solution strategy for both multi-query shortest-walk and shortest-path problems in GCS. We proceed in two phases. Offline, we compute a coarse lower bound on the cost-to-go function between relevant pairs of GCS vertices. Then online, we use these cost-to-go lower bounds to guide a greedy incremental search algorithm, quickly recovering a solution one vertex at a time.

Our strategy is inspired by a similar methodology commonly used for solving the classical all-pairs SPP. Before describing our solution, we briefly outline this classical all-pairs SPP methodology and discuss the key challenges in extending it to the GCS setting.

3.3.1 Classical all-pairs SPP review

The goal of the classical all-pairs SPP is to compute the shortest paths between all pairs of vertices in the graph. Efficient solutions to this problem leverage the principle of optimality. Instead of computing the full path for each pair of vertices, it suffices to compute only the immediate successor along each path, which crucially depends solely on the current and target vertices. The full path can then be constructed incrementally, one vertex at a time.

It is common to implicitly encode the solution to the classical all-pairs SPP into the cost-to-go function $J^*_{v,t}$ for every pair of vertices v and t. The successor is then computed by greedily picking a vertex that minimizes the one-step lookahead with respect to the cost-to-go. Given the current and target vertices u and t the greedy decision policy π selects the next vertex along the shortest u-t path:

$$\pi(u,t) \quad = \quad \arg\min_{v} \quad c_u + c_e + J^*_{v,t} \tag{3.1a}$$

$$\text{s.t.} \quad e = (u,v) \in \mathcal{E}. \tag{3.1b}$$

When the exact cost-to-go $J^*_{v,t}$ is available, this greedy policy is the optimal decision policy. Naturally, if only an approximate or heuristic cost-to-go is available, the greedy policy yields heuristic and not necessarily optimal solutions. Optimal solutions can still be obtained via incremental search (e.g., A*).

3.3.2 Limitations when generalizing to GCS

Shortest walks in GCS. Since the principle of optimality holds for the SWP in GCS, the solution to this problem can also be efficiently encoded via the cost-to-go function. If we had

access to this cost-to-go, a GCS generalization of the greedy policy (3.1) would constitute the optimal decision policy for this problem.

Unfortunately, similar to the explicit MPC [32, 33], the exact shortest-walk cost-to-go function can be arbitrarily complex even for simple GCS instances; computing it is intractable. Instead, we produce a coarse piecewise-quadratic lower bound on the cost-to-go. Similar to the classical all-pairs SPP, we are then able to produce quick heuristic solutions via greedy search, or provably optimal solutions via a GCS generalization of the A* algorithm.

Shortest paths in GCS. Unlike the classical all-pairs SPP and the SWP in GCS, the optimal decision policy for the SPP in GCS depends on the set of previously visited vertices (as demonstrated in Section 2.3.3 and Example 2). Explicitly capturing this dependency requires $2^{|\mathcal{V}|-1}$ different cost-to-go functions per vertex: one for every possible set of previously visited vertices, which is combinatorially intractable. Moreover, similar to the shortest walks in GCS and the explicit MPC, these cost-to-go functions can be arbitrarily complex in general, and thus intractable to compute exactly.

We proceed with the same general approach described above: we synthesize a coarse lower bound on the cost-to-go and use incremental search to obtain a solution. Similarly, we can quickly obtain heuristic paths with a greedy policy, or provably optimal paths via A*.

3.4 Offline phase: synthesis of cost-to-go lower bounds

We now develop the optimization problems that synthesize shortest-walk and shortest-path cost-to-go lower bounds over the GCS. First, in Section 3.4.1, we derive the Bellman equation for the shortest walks in GCS, which gives a recursive condition on optimality of the cost-to-go function in this setting. Next, in Section 3.4.2, we solve this Bellman equation, obtaining the exact shortest-walk cost-to-go function as the solution to an infinite-dimensional Linear Program (LP). We then extend and strengthen this program to produce (not necessarily tight) lower bounds on the shortest-path cost-to-go in Section 3.4.3. Finally, in Section 3.4.4, we present a tractable numerical approximation to these infinite-dimensional LPs, producing piecewise-quadratic cost-to-go lower bounds.

For clarity of presentation, we make some simplifying assumptions. We refer the reader to Appendix A for the extension of our method where these assumptions are lifted.

Assumption 2. First, we assume that we have just one source vertex and one target vertex, i.e., $\mathcal{S} = \{s\}$ and $\mathcal{T} = \{t\}$. Second, we assume that the convex set $\mathcal{X}_t$ corresponding to the target vertex t is a single point: $\mathcal{X}_t = \{\bar{x}_t\}$. Since we have a unique target point $\bar{x}_t$, we also simplify the notation and refer to the shortest-path cost-to-go function $J^*_{v,t}(x_v, \bar{x}_t)$ as $J^*_v(x_v)$, and the shortest-walk cost-to-go $J'_{v,t}(x_v, \bar{x}_t)$ as $J'_v(x_v)$.

3.4.1 Bellman equation for the shortest walks

As discussed in Section 2.3.3, the principle of optimality holds for the SWP in GCS: every subwalk of a shortest walk is itself a shortest walk. Leveraging this property, we derive the Bellman equation that characterizes the cost-to-go function for this problem.

Recall that for every vertex $v \in \mathcal{V}$ and point $x_v \in \mathcal{X}_v$, $J'_v(x_v)$ denotes the shortest-walk cost-to-go from the point x_v to the fixed target point $\bar{x}_t$. Consider a point x_u of vertex u, and let the point x_v of vertex v be next along a shortest walk from x_u (where vertices u, v need not be distinct, since this is a walk). Then by the principle of optimality, the cost-to-go $J'_u(x_u)$ must be the sum of the incurred costs $l_u(x_u) + l_{(u,v)}(x_u, x_v)$ and the subsequent cost-to-go $J'_v(x_v)$. Furthermore, since the transition to x_v of vertex v is optimal, it must minimize this sum among all other feasible transitions. This is summarized in the Bellman equation below:

$$\begin{aligned} J'_u(x_u) \quad = \quad & \min_{x_v,\, v} \quad l_u(x_u) + l_e(x_u, x_v) + J'_v(x_v) \\ & \text{s.t.} \quad e = (u, v) \in \mathcal{E}, \\ & \qquad (x_u, x_v) \in \mathcal{X}_e. \end{aligned} \tag{3.2}$$

Although the minimization above should generally be replaced with the inf, the Assumption 1 ensures that the optimal walks are finite, making the minimization in (3.2) valid.

3.4.2 Cost-to-go lower bounds for the shortest walks

To solve the Bellman equation (3.2), we draw inspiration from the well-known linear programming approach [34–37], which searches for cost-to-go lower bounds by imposing a relaxed inequality version of the Bellman equation. Similar to these methods, the optimization problem we derive in this section is also an LP, but it is infinite-dimensional and cannot be immediately solved numerically. Later in Section 3.4.4 we develop a tractable finite-dimensional approximation conducive to numerical methods.

The program we formulate searches over the space of lower-bounds on the shortest-walk cost-to-go function J'_v. For each vertex $v \in \mathcal{V}$, we denote these lower bounds as $J_v : \mathcal{X}_v \to \mathbb{R}$. Note that we search over the space of functions J_v, not over the individual points x_v. To ensure that J_v is a lower bound on J'_v, we impose a relaxed inequality version of the Bellman equation (3.2) as a constraint:

$$J_u(x_u) \leq l_u(x_u) + l_e(x_u, x_v) + J_v(x_v),$$

for every edge $e = (u, v) \in \mathcal{E}$ and for every feasible pair of points $(x_u, x_v) \in \mathcal{X}_e$. This inequality states that $J_u(x_u)$ is no higher than the incurred vertex and edge costs $l_u(x_u) + l_e(x_u, x_v)$ plus the subsequent value $J_v(x_v)$. Constraining $J_t(\bar{x}_t) = l_t(\bar{x}_t)$ at the target, the resulting functions J_u must be lower bounds on the cost-to-go J'_u, that is: $J_u(x_u) \leq J'_u(x_u)$ for all points $x_u \in \mathcal{X}_u$ and vertices $u \in \mathcal{V}$. To make J_s a tight lower bound on the cost-to-go J'_s at the source vertex s, we maximize the average value of J_s over the source set $\mathcal{X}_s$. We obtain the following optimization problem:

$$\max_{\{J_v\}_{v \in \mathcal{V}}} \quad \int_{\mathcal{X}_s} J_s(x) d\phi_s(x) \tag{3.3a}$$

$$\text{s.t.} \quad J_v : \mathcal{X}_v \to \mathbb{R}, \qquad \forall v \in \mathcal{V}, \tag{3.3b}$$

$$J_u(x_u) \leq l_u(x_u) + l_e(x_u, x_v) + J_v(x_v), \qquad \forall e = (u, v) \in \mathcal{E}, \quad \forall (x_u, x_v) \in \mathcal{X}_e, \tag{3.3c}$$

$$J_t(\bar{x}_t) = l_t(\bar{x}_t), \tag{3.3d}$$

where ϕ_s in the objective (3.3a) is a probability distribution of anticipated source conditions over the set $\mathcal{X}_s$ [2]. Constraints (3.3c) and (3.3d) enforce that J_v is a lower bound on J'_v for every vertex $v \in \mathcal{V}$, while the integral in (3.3a) maximizes the weighted average of J_s over $\mathcal{X}_s$, effectively "pushing up" the cost-to-go lower bound at the source vertex. The objective function is maximized when $J_s(x_s) = J'_s(x_s)$ for all $x_s \in \mathcal{X}_s$, and thus the optimal solution to program (3.3) yields tight lower bounds on the shortest-walk cost-to-go over the source set.

Simultaneously maximizing the average value of J_v across all vertices $v \in \mathcal{V}$ yields tight lower bounds on the shortest-walk cost-to-go over the entire GCS. This is similar to the classical many-to-one SPP, lifting one of the simplifications made in Assumption 2. Further details and extensions are provided in Appendix A.

We note that program (3.3) naturally generalizes the well-known cost-to-go synthesis LP for the classical SPP. When each convex set $\mathcal{X}_v$ is a single point, the SWP in GCS reduces to the classical SWP in an ordinary graph, which in turn reduces to the classical SPP, where functions J_v are defined at single points and represented by a single decision variable per vertex. Program (3.3) reduces to the well-known classical cost-to-go synthesis LP:

$$\begin{aligned} \max_{\{J_v\}_{v\in\mathcal{V}}} \quad & J_s \\ \text{s.t.} \quad & J_u \le l_u + l_e + J_v, \qquad \forall e = (u,v) \in \mathcal{E}, \\ & J_t = l_t. \end{aligned}$$

Similar to this purely discrete classical setting, optimization problem (3.3) is also an LP; however, it searches over the space of functions and is therefore infinite-dimensional. In Section 3.4.4, we develop a tractable finite-dimensional approximation to program (3.3).

3.4.3 From walks to paths

We now turn to cost-to-go synthesis for the SPP in GCS. As shown in Section 2.3.3 and Example 2, the principle of optimality does not hold for the shortest paths in GCS. Due to the constraint that vertices cannot be revisited along the path, previous decisions now have a bearing on the subsequent ones: that is, the optimal decision policy at every point and every vertex depends on the set of visited vertices. Consequently, the shortest-path cost-to-go function is also history-dependent. Explicitly capturing this dependency requires $2^{|\mathcal{V}|-1}$ different functions per vertex: one for every set of possibly visited vertices, which is combinatorially intractable. Instead, we opt to compute a single heuristic lower bound on the cost-to-go function, which is then used to guide an incremental search policy.

The SWP cost-to-go function produced in program (3.3) already provides a valid lower bound for the SPP cost-to-go: indeed, since every path is a walk, the cost of a shortest path is at least that of a shortest walk. In this section, we strengthen program (3.3) to yield tighter

[2] At this stage, the choice of distribution ϕ_s is irrelevant so long as it is positive over the source set $\mathcal{X}_s$. If it is, then the optimal solution to the program (3.3) is a tight lower bound on the shortest-walk cost-to-go function: $J_s = J'_s$. This is because J_s has arbitrary expressive power, as we search over the infinite-dimensional space of functions. However, when finite-dimensional approximations are introduced in Section 3.4.4, the expressive power of J_s becomes finite, and the choice of ϕ_s becomes important, as it effectively determines what part of the set $\mathcal{X}_s$ the expressive power should be spent on.

lower bounds on the SPP cost-to-go. Empirically, these tighter lower bounds serve as better heuristics when producing paths via incremental search.

The lower bound on the cost-to-go is obtained using the following optimization problem:

$$\max_{\{J_v, h_v\}_{v\in\mathcal{V}}} \quad \int_{\mathcal{X}_s} J_s(x) d\phi_s(x) \tag{3.4a}$$

$$\text{s.t.} \quad J_v : \mathcal{X}_v \to \mathbb{R}, \qquad \forall v \in \mathcal{V}, \tag{3.4b}$$

$$J_u(x_u) \le l_u(x_u) + l_e(x_u, x_v) + h_v + J_v(x_v), \qquad \forall e = (u, v) \in \mathcal{E}, \quad \forall (x_u, x_v) \in \mathcal{X}_e, \tag{3.4c}$$

$$J_t(\bar{x}_t) = l_t(\bar{x}_t) - \sum_{v\in\mathcal{V}} h_v, \tag{3.4d}$$

$$h_v \ge 0 \qquad \forall v \in \mathcal{V}. \tag{3.4e}$$

We now detail the differences between programs (3.3) and (3.4), and then prove the validity of the lower bound produced in program (3.4) in Lemma 1 below.

The objective function (3.4a) and the constraint (3.4b) remain unchanged: for each vertex $v \in \mathcal{V}$, we associate a function J_v that serves as a lower bound on the cost-to-go J_v^*, while the integral in the objective maximizes the weighted average of J_s over the source set $\mathcal{X}_s$.

In (3.4e), we introduce a non-negative penalty h_v for every vertex $v \in \mathcal{V}$. This penalty is meant to discourage revisits to vertex v, which is a way to relax the constraint that a path must not visit any vertex more than once.

To implement the penalty h_v, we increment the edge cost l_e for every edge $e \in \mathcal{E}$ that enters vertex v. This is formalized in (3.4c), which states that for every edge $e = (u, v)$ and a feasible transition $(x_u, x_v) \in \mathcal{X}_e$, the value $J_u(x_u)$ is a lower bound on the sum of the vertex cost $l_u(x_u)$, penalty-incremented edge cost $l_e(x_u, x_v) + h_v$, and the subsequent cost-to-go lower bound $J_v(x_v)$. As written, the non-negative penalty h_v increases the cost of the edges leading into vertex v, thereby discouraging visits to v. However, since our goal is to only discourage subsequent vertex revisits, we need to waive the penalty h_v once. This is achieved by setting the cost-to-go lower bound $J_t(\bar{x}_t)$ to $l_t(\bar{x}_t) - \sum_{v\in\mathcal{V}} h_v$ in constraint (3.4d). Upon reaching the target vertex, we incur the target vertex cost $l_t(\bar{x}_t)$ and subtract the sum of all vertex penalties, effectively waiving the penalties once per vertex. We now show that these constraints produce lower bounds on the cost-to-go function.

Lemma 1. *Let $\{J_u, h_u\}_{\forall u\in\mathcal{V}}$ be a feasible solution of problem (3.4). Then*

$$J_u(x_u) \le J_u^*(x_u) \text{ for all } u \in \mathcal{V}.$$

Proof. Consider a feasible solution to program (3.4), and let u be some vertex. Let (p, τ) be the shortest path in GCS from a point $x_u \in \mathcal{X}_u$ to the target point $\bar{x}_t$ of target vertex t. Since vertex sequence p is a path, it contains no repeated vertices. Adding the constraint (3.4c) along the edges $\mathcal{E}_p$ of this optimal path, we have:

$$J_u(x_u) \le \sum_{e=(v,w)\in\mathcal{E}_p} \Big(l_v(x_v) + l_e(x_v, x_w) \Big) + \sum_{v\in p} h_v + J_t(\bar{x}_t), \tag{3.5}$$

where x_v, x_w are the vertex variables of the optimal trajectory τ corresponding to the path p (note: here w is used to represent a vertex, not a walk). Constraint (3.4d) states that $J_t(\bar{x}_t) = l_t(\bar{x}_t) - \sum_{v \in \mathcal{V}} h_v$, while the sum of the edge costs $l_e(x_v, x_w)$ along the optimal path p is by definition the cost-to-go $J^*_u(x_u)$. Substituting and rearranging terms, we obtain:

$$J_u(x_u) + \sum_{v \notin p} h_v \;\leq\; J^*_u(x_u). \tag{3.6}$$

Since the penalties h_v are non-negative by (3.4e), the conclusion follows. □

By maximizing the weighted average of J_s in the objective function (3.4a), the program (3.4) seeks the best possible lower bound J_s on the cost-to-go J^*_s, up to the relaxation gap introduced by the vertex penalties. This gap is clear from (3.6): for $x_s \in \mathcal{X}_s$, the sum of the off-the-optimal-path penalty terms $\sum_{v \notin p} h_v$ need not to be zero, so $J_s(x_s)$ need not be a tight lower bound on $J^*_s(x_s)$. In other words, recall that, upon reaching the target, we waive the penalties h_v for every vertex $v \in \mathcal{V}$. As a result, we do not just waive the first-time penalties on vertices along the optimal path p, we also waive the off-the-path penalties $\sum_{v \notin p} h_v$, which were never accrued in the first place[3]. Waiving these off-the-path penalties introduces the gap between J_s and J^*_s.

Lemma 1 proves that program (3.4) generates a lower bound on the shortest-path cost-to-go function. Observe that by setting $h_v = 0$ for every $v \in \mathcal{V}$, we remove the penalties on vertex revisits, and recover the program (3.3). Thus the optimal solution to (3.3) is a feasible solution to (3.4), and the lower bound obtained in this section is at least as strong as the shortest-walk lower bound derived earlier.

3.4.4 Numerical approximation via semidefinite programming

We now produce approximate solutions to the shortest-walk and shortest-path cost-to-go synthesis programs (3.3) and (3.4). Since the SWP cost-to-go synthesis program (3.3) is a special case of (3.4) (obtained by setting vertex penalties $h_v = 0$), we focus just on the later program. We restrict each function J_v to be convex quadratic, which allows us to cast (3.4) as a tractable Semidefinite Program (SDP). SDPs are mathematical programs where the objective function is linear and the constraints are either linear or linear matrix inequalities (LMIs). To help with the presentation, we first state without proof three well-known facts.

Lemma 2 (e.g., [55, App. A.1]). *A quadratic function $f : \mathbb{R}^n \to \mathbb{R}$ is non-negative if and only if it is representable as a Positive-Semidefinite (PSD) quadratic form:*

$$f(x) = \begin{bmatrix} 1 \\ x \end{bmatrix}^\top Q \begin{bmatrix} 1 \\ x \end{bmatrix} \quad \text{for some} \quad Q \succeq 0.$$

Lemma 3 ([55, Section 3.2.4]). *Let $\mathcal{X} = \{x \in \mathbb{R}^n \mid g_i(x) \geq 0,\ \forall i = 1, \ldots, m\}$. The function $f : \mathbb{R}^n \to \mathbb{R}$ is non-negative on the set $\mathcal{X}$ if there exists $\lambda \in \mathbb{R}^m_+$, such that $f(x) - \sum_{i=0}^m \lambda_i g_i(x)$ is non-negative for every $x \in \mathbb{R}^n$.*

[3] Note that if the source set was a single point ($\mathcal{X}_s = \{\bar{x}_s\}$), then this gap would be zero, and the lower bound J_s would be tight at $\bar{x}_s$. Though we still cannot solve this program exactly, as it is an infinite-dimensional LP.

Corollary 1. *Suppose that in Lemma 3, the function f is quadratic, and all g_i functions are affine or convex quadratic. Then we can apply Lemma 2 to verify Lemma 3 via an LMI, i.e., we can verify if f is non-negative over $\mathcal{X}$ by searching for a PSD matrix in an affine subspace.*

Using these facts, we proceed to cast program (3.4) as an SDP.

Defining cost-to-go lower bounds in (3.4b). We restrict lower bounds J_v per vertex $v \in \mathcal{V}$ to be convex quadratic functions. By Lemma 2, searching for such functions is equivalent to searching for appropriate PSD matrices Q_v. The decision variables are thus the coefficients of the quadratic polynomials. As a result, we produce a coarse convex quadratic lower bound on the optimal cost-to-go function at every vertex v.

Constraint (3.4e) is already linear, and constraint (3.4d) is linear in the coefficients of the quadratic polynomial J_t and the decision variables h_v. These constraints are thus already suitable for the SDP.

Enforcing the lower-bound constraint (3.4c). To apply Corollary 1 to enforce this constraint, we impose additional restrictions. First, we restrict vertex and edge sets $\mathcal{X}_v$ and $\mathcal{X}_e$ to be intersections of ellipsoids and polyhedra. We also restrict vertex and edge costs l_u, l_e to be quadratic, ensuring that the expression in (3.4c) is quadratic. For non-quadratic edge and vertex costs, such as the Euclidean distance, we use quadratic approximation instead. Applying Corollary 1, we verify constraint (3.4c) with an LMI.

The objective function (3.4a). Since J_s is a quadratic polynomial, the integral in (3.4a) is linear in the coefficients of J_s[4], which are the decision variables of the program. Therefore, the objective function (3.4a) is linear in the decision variables, as required for the SDP.

Empirically, we found quadratic lower bounds to be a good balance between computational complexity and expressive power. Note that higher-degree polynomial lower bounds J_v can be synthesized via the Sums-of-Squares (SOS) hierarchy [56–58]. However, in practice, the resulting programs tend to be prohibitively expensive. On the other hand, restricting J_v to be affine yields a program that almost exactly matches the dual to the convex relaxation of the SPP in GCS, discussed in [1, App. B]. In other words, solving the SPP in GCS already gives a coarse affine cost-to-go lower bound that can be used to solve multi-query SPP in GCS. In Section 4.2, we show that empirically, these affine lower bounds have significantly less expressive power than the quadratic lower bounds. Although quadratic cost-to-go lower bounds are still coarse, we find them surprisingly effective in practice. In particular, we are able to effectively mitigate this coarseness by using multi-step lookaheads in the greedy search policy, which effectively produces piecewise quadratic lower bounds on the cost-to-go function at every vertex. This is demonstrated in Section 4.1.

[4]Specifically, the objective is a product of the coefficients of J_s and the moments of the distribution ϕ_s of anticipated initial conditions over the source set $\mathcal{X}_s$. Since the expressive power of the polynomial lower bound J_s is limited, ϕ_s serves as a preference weighting on the quality of the lower bound over $\mathcal{X}_s$. Consequently, ϕ_s should be chosen to prioritize parts of $\mathcal{X}_s$ where higher accuracy of the lower bound is most critical.

3.5 Online phase: incremental search

After synthesizing lower bounds on the cost-to-go functions, we extract shortest walks and paths using incremental search. Motivated by fast online solve times, we employ greedy search, which generates high-quality solutions by taking locally-optimal decisions at each step, greedily descending along the cost-to-go function. To mitigate the coarseness of the quadratic lower bounds and better approximate the optimal decision policy, we use multi-step lookahead greedy search, where n-step optimal decision sequences are computed at each step, and the first decision is executed.

While fast and effective, greedy search provides no guarantees on optimality or completeness. We note that for provably optimal walks or paths, best-first search can be used (such as a GCS generalization of A* [11]). This method guarantees finding an optimal solution, though in the worst case, its computational cost grows exponentially with the length of the solution. We refer the reader to Appendix B for further details on best-first search.

We now formulate the multi-step lookahead greedy policy for the shortest paths and walks in GCS, generalizing the one-step greedy policy (3.1) from the classical all-pairs SPP.

Greedy policy for the shortest paths. Suppose that at runtime, we are given a source vertex $v_0 \in \mathcal{S}$ and a source point $x_0 \in \mathcal{X}_{v_0}$. At iteration k of the policy rollout, let (v_k, x_k) be the current vertex and vertex point, and let $p_k = (v_0, \ldots, v_k)$ be the path so far. The greedy policy, which we define shortly, uses this information to produce the next vertex v_{k+1} and the corresponding vertex point x_{k+1}. We then advance to the next iteration. The rollout terminates when we reach the target vertex t, where we must select the target point $\bar{x}_t$. At each iteration of the policy rollout, to determine the next vertex and point to go to, we solve a greedy lookahead optimization problem with respect to the cost-to-go lower bounds.

Given a vertex v, a set of visited vertices p, and a lookahead horizon n, let $\mathcal{P}(v, p, n)$ be the set of all candidate n-step paths that originate at v and do not include vertices in p. Here we also consider candidate paths that terminate at the target vertex t early: we include paths that start at v, end at t, and have fewer than n steps. We define the set $\mathcal{P}(v, p, n)$ to include all such paths as well. The n-step lookahead program is then defined as follows:

$$\min_{(p,\tau)} \quad l(p,\tau) + J_{v_{k+n}}(x_{k+n}) - l_{v_{k+n}}(x_{k+n}) \tag{3.7a}$$

$$\text{s.t.} \quad p \in \mathcal{P}(v_k, p_k, n), \tag{3.7b}$$

$$\tau \in \mathcal{T}_p(x_k, x_{k+n}). \tag{3.7c}$$

This program considers candidate n-step decision sequences (p, τ) that originate at (v_k, x_k), and among them selects the one that minimizes the n-step lookahead cost-to-go.

We now explain program (3.7) line by line. In (3.7b), we consider all candidate n-step paths $p = (v_k, \ldots, v_{k+n})$ that start at the vertex v_k and do not include vertices p_k that have already been visited. The set $\mathcal{P}(v_k, p_k, n)$ encompasses all such paths. In (3.7c), for each candidate path p, we consider a trajectory $\tau = (x_k, \ldots, x_{k+n})$ along this path, which starts at the fixed point x_k and ends at a variable point x_{k+n}. The set $\mathcal{T}_p(x_k, x_{k+n})$ describes all such feasible trajectories. If the candidate path p ends with the target vertex t, we modify the

constraint (3.7c) to be $\tau \in \mathcal{T}_p(x_k, \bar{x}_t)$, so that the trajectory τ ends on the target point $\bar{x}_t$. Together, path p and trajectory τ are a candidate n-step decision sequence through the GCS.

Through the objective (3.7a), we seek the decision sequence that minimizes the n-step lookahead cost-to-go. The objective function consists of three terms. The first term $l(p, \tau)$ is the sum of the vertex and edge cost incurred along the n-step path. The second term $J_{v_{k+n}}(x_{k+n})$ is the lower bound on the remaining cost-to-go from the last vertex v_{k+n} and point x_{k+n}. To avoid double-counting, the third term $-l_{v_{k+n}}(x_{k+n})$ subtracts the vertex cost incurred from visiting v_{k+n}, as it is counted in both $l(p, \tau)$ and $J_{v_{k+n}}(x_{k+n})$. Having obtained the optimal n-step decision sequence (p, τ), we take just the first step, transition to (v_{k+1}, x_{k+1}), and advance to the next iteration. The policy rollout terminates once we reach the target vertex.

We emphasize that this multi-step lookahead formulation is key for mitigating the coarseness of the quadratic cost-to-go lower bound J_v. This is because an n-step lookahead from vertex v effectively produces a piecewise-quadratic lower bound on the optimal cost-to-go J_v^* over $\mathcal{X}_v$, which has significantly more expressive power. While these lower bounds can still be loose in theory, multi-step lookaheads enable effective decision-making in practice.

Modifications for the shortest walks. The policy for the shortest walks is nearly identical, with two minor changes. First, in (3.7b), instead of searching for a path that originates at (v_k, x_k), we search for a walk, allowing vertex revisits. Second, the policy rollout need not terminate upon reaching the target vertex t, as this vertex may also be revisited. Instead, we have to consider multiple options: one where the target point $\bar{x}_t$ is selected and the rollout is terminated, and one where some other point is selected from the target set $\mathcal{X}_t$ and the policy rollout continues.

Efficient implementation. To actually compute the minimizer of (3.7), we solve multiple convex programs in parallel, one for each candidate n-step lookahead path, then select the best path. For each candidate path $p = (v_k, \ldots, v_{k+n}) \in \mathcal{P}(v_k, p_k, n)$, we solve the following convex program to find the optimal trajectory $\tau = (x_k, \ldots, x_{k+n})$ along this candidate path:

$$\begin{aligned} \min_{\tau} \quad & l(p, \tau) + J_{v_{k+n}}(x_{k+n}) - l_{v_{k+n}}(x_{k+n}) \\ \text{s.t.} \quad & \tau \in \mathcal{T}_p(x_k, x_{k+n}). \end{aligned}$$

To make the convex structure of this program more clear, we expand $l(p, \tau)$ and $\mathcal{T}_p$ below:

$$\min_{(x_{k+1}, \ldots, x_{k+n})} \quad \sum_{m=k+1}^{k+n} \Big(l_{v_{m-1}}(x_{m-1}) + l_{e_m}(x_{m-1}, x_m) \Big) + J_{v_{k+n}}(x_{k+n}) \tag{3.8a}$$

$$\text{s.t.} \quad (x_{m-1}, x_m) \in \mathcal{X}_{e_m}, \qquad \forall m = k+1, \ldots, k+n. \tag{3.8b}$$

Here, the decision variables are the continuous vertex points $(x_{k+1}, \ldots, x_{k+n})$ along the candidate path; note that point x_k is not a decision variable: it remains fixed. In (3.8b), we enforce the edge constraints on the consecutive points; recall that vertex constraints $x_m \in \mathcal{X}_{v_m}$ are enforced implicitly due to the assumption $\mathcal{X}_{e_m} \subseteq \mathcal{X}_{v_{m-1}} \times \mathcal{X}_{v_m}$. By definition, these constraints are convex. The objective function (3.8a) is also convex, since it is the sum

of convex functions: convex vertex and edge costs along the path and the convex cost-to-go lower bound $J_{v_{k+n}}(x_{k+n})$. We emphasize that this is the reason why we chose to synthesize convex cost-to-go lower bounds: to ensure that the objective function (3.8a) is convex. As a result, program (3.8) is convex, and can be solved quickly and efficiently at run-time.

To speed up online computation, parametric programming is used offline to pre-build these n-step lookahead programs into binaries, thus avoiding the compilation overhead at run-time. After synthesizing the cost-to-go lower bounds, we precompile a parametric program for every possible n-step decision sequence, with x_k as the parameter and $(x_{k+1}, \ldots, x_{k+n})$ as the decision variables. Program (3.8) is convex in both parameters and decision variables, as per disciplined convex parametric programming [59]. At run-time, we solve these pre-built programs in parallel using multithreading, enabling very fast execution in practice.

Lastly, several techniques can be employed to further reduce the number of convex programs solved at run-time. One effective approach is to prune n-step decision sequences that are inherently infeasible, regardless of the initial point. This can be achieved by solving a simple feasibility program offline for each candidate n-step path p:

$$\text{find } x_0, x_n, \tau \quad \text{s.t.} \quad \tau \in \mathcal{T}_p(x_0, x_n).$$

Sequences for which this program is infeasible can be safely discarded. Additionally, for many robotic systems, certain candidate sequences can be heuristically pruned due to their impracticality or lack of relevance. For example, paths that involve highly inefficient detours or explore dead-ends can often be identified and discarded offline.

Post-processing. To further improve the quality of walks and paths obtained via greedy search, we apply two post-processing steps upon termination of the policy rollout. First, we extract the vertex sequence $(v_0, \ldots, v_t)$ and re-optimize the continuous vertex variables $(x_0, \ldots, x_t)$, so as to produce a trajectory that is optimal within this walk or path. Second, we use short-cutting, attempting to reduce the vertex sequence. Specifically, we look for non-consecutive vertices v_k and v_m that are connected by an edge $e = (v_k, v_m)$, with $k < m$. If such vertices are found, we consider a new vertex sequence $(v_0, \ldots, v_k, v_m, \ldots, v_t)$, re-optimize the corresponding trajectory, and accept the new solution if it is feasible and has a lower cost. For the shortest walks, this short-cutting procedure tends to eliminate unnecessary cycles along the walk. Empirically, we observed short-cutting to substantially improve the solution quality of both walks and paths. However, it can be computationally expensive due to a potentially large number of short-cutting options, as well as the overhead of constructing programs at run-time. Thus, the extent of short-cutting should be tailored to the available time budget.

Recursive feasibility. Finally, we note that the lookahead program (3.7) is not guaranteed to be recursively feasible. If we end up in a vertex where the lookahead program has no solution, we backtrack to a previous vertex that has a different feasible candidate outgoing edge, and retry from there. As a result, our planner is sound but not complete: it is not guaranteed to produce a solution, but every solution it produces is feasible. A complete and provably-optimal planner can be attained via best-first search — see Appendix B for details.

Chapter 4

Experimental demonstrations

In this section, we evaluate the proposed solution method through a series of experiments. First, in Section 4.1, we present a simple example to illustrate the algorithm's offline and online phases. Next, Section 4.2 examines how the coarseness of the cost-to-go lower bounds and the horizon of the multi-step lookahead greedy policy affect the performance of our algorithm. In Section 4.3, we apply our approach to the motivating example from Figure 1.1, a robot arm performing motion planning in a warehouse setting, and compare its performance against PRM, its natural sampling-based competitor. Finally, in Section 4.4 we illustrate the applicability and effectiveness of our approach across several challenging robotics systems.

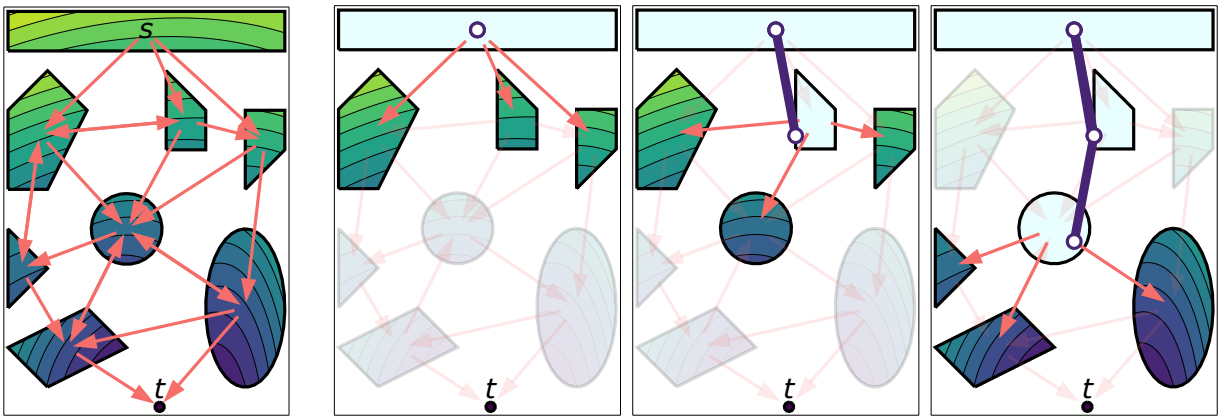
Unless otherwise specified, all experiments were run on a desktop computer with a 4.5Ghz 16-core AMD Ryzen 9 processor and 64GB 4800MHz DDR5 memory. We use Mosek 10.2.1 [60] to solve all convex programs in this section.

4.1 Visual intuition

To illustrate our approach, we consider a simple two-dimensional GCS problem in Figure 4.1. We have a graph G with $|\mathcal{V}| = 9$ vertices, $|\mathcal{E}| = 25$ edges, including multiple cycles. The geometry of the convex sets $\mathcal{X}_v$ can be deduced from Figure 4.1a; no edge constraints $\mathcal{X}_e$ are used. The edge costs $l_e(x_u, x_v) = \|x_u - x_v\|_2^2$ are the squared Euclidean distance, and there are no vertex costs. The source vertex s is a box at the top, the target vertex t is a single point at the bottom, and the goal is to find the shortest paths between them.

Offline we compute the convex quadratic lower bounds on the shortest-path cost-to-go function at every vertex. The contour plots of these cost-to-go lower bounds are visualized in Figure 4.1a. Then at run-time, we are given a specific point in the source set. In Figure 4.1b, we depict the first three iterations of the 1-step lookahead rollout of the greedy policy (3.7). At each iteration, we expand the neighbors of the current vertex and greedily select the next vertex v and the vertex point x_v that minimize the objective (3.7a). The rollout proceeds until the target vertex t is reached.

We evaluate the quality of the cost-to-go lower bounds and the resulting solutions in Figure 4.2. The optimal shortest path cost-to-go function J_s^* (green) is piecewise-quadratic. Naturally, the convex quadratic lower bound J_s (purple) is a poor lower bound to J_s^*. However, the quality of the lower bound is greatly improved via multi-step lookaheads (solid lines,



(a) Offline: synthesize a cost-to-go lower bound over the GCS. Contour plots are shown.

(b) Online: at each iteration, we evaluate all candidate n-step paths from the current vertex ($n=1$ shown) and greedily select the decision that minimizes the n-step lookahead cost-to-go. The first three iterations are shown, as the path is built incrementally.

Figure 4.1: Illustration of our methodology. The GCS instance is embedded in $\mathbb{R}^2$, with the source vertex at the top and the target vertex at the bottom. The edges are shown as red arrows, and the edge length is the squared Euclidean distance.

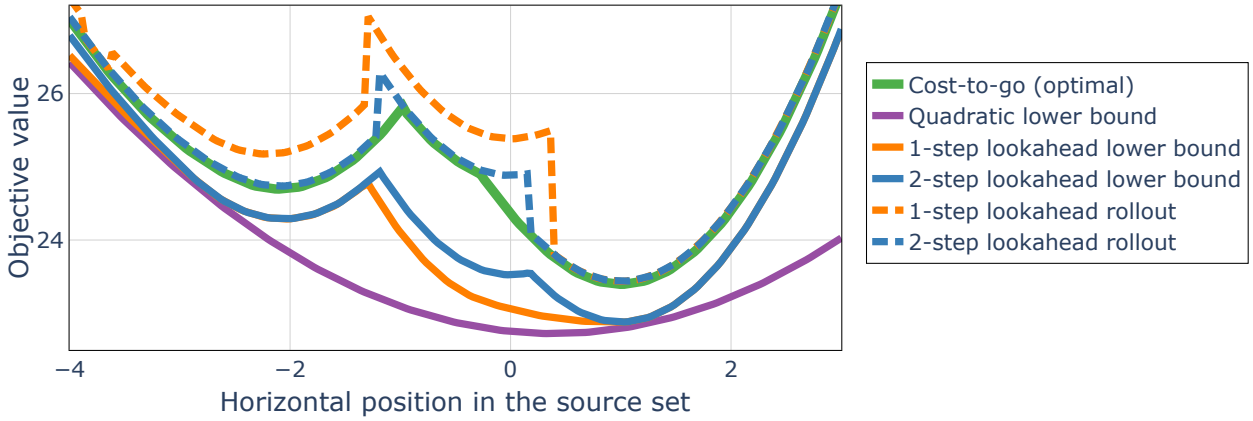
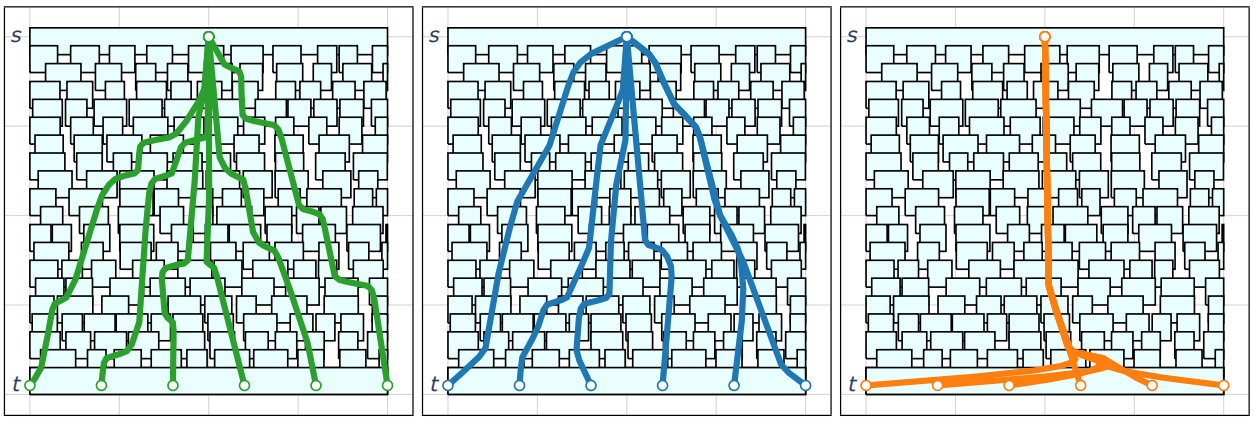


Figure 4.2: Comparison of lower and upper bounds on the cost-to-go over a horizontal slice of the source set $\mathcal{X}_s$ from Figure 4.1a. The shortest-path cost-to-go function J_s^* (green) is piecewise-quadratic. Convex quadratic lower bound J_s (purple) is naturally a poor lower-bound. Multi-step lookaheads (solid orange, blue) produce tighter piecewise-quadratic lower bounds. Upper bounds on the cost-to-go are obtained by rolling out the multi-step lookahead policy (dashed orange, blue), which produces near-optimal solutions.

orange for 1-step, blue for 2-step). A horizon-n lookahead produces a piecewise-quadratic lower bound to J_s^*, with up to as many quadratic pieces as there are different n-step paths from the source vertex s. Though neither 1-step nor 2-step lookahead lower bounds are tight, they successfully capture the general cost-to-go landscape, and are sufficient for near-optimal decision making. The costs of the rollouts of the greedy policy are plotted as dashed lines; in particular, 2-step lookahead rollouts (blue) attain optimal solutions nearly always.



(a) Optimal solution. (b) Quadratic lower bounds. (c) Affine lower bounds.

Figure 4.3: A 3-step lookahead policy with quadratic J_v (blue) yields diverse vertex paths resembling the optimal solutions (green). A 3-step lookahead with affine J_v (orange) follows a single vertex sequence regardless of the target point, accruing much higher cost.

4.2 Polynomial lower-bound and lookahead horizon: effects on solution quality

In this section, we analyze how the coarseness of the cost-to-go lower bounds and the lookahead horizon impact solution quality. First, we show that multi-step lookaheads with quadratic J_v yield near-optimal solutions in very large graphs. Second, we demonstrate that quadratic lower bounds significantly outperform the affine ones, which are available from the dual of the convex relaxation of the SPP in GCS [1, App. B].

We consider a randomly generated environment depicted in Figure 4.3. We assign a GCS vertex v for each teal box. Each convex set $\mathcal{X}_v$ is the set of control points of a cubic Bézier curve within the box (see [3] for more details). The GCS vertices are connected by a pair of edges if the corresponding teal boxes overlap. The resulting graph has 190 vertices and 540 edges. For each edge, we constrain the vertex Bézier curves to be differentiable at the transition point. The path cost is the sum of squared Euclidean distances between the consecutive control points of the Bézier curves. The source vertex s is at the top, and the target vertex t is at the bottom, and our goal is to find shortest paths between random source and target points.

We synthesize the quadratic and affine lower bounds on the shortest-path cost-to-go over the GCS, which takes 6s and 2s respectively. We then uniformly sample 120 pairs of source and target conditions, and rollout the greedy policy using different lower bounds and lookahead horizons. Optimal solutions are obtained by solving the MICP formulation of the SPP in GCS. Numerical results are reported in Table 4.1.

Table 4.1 shows that our approach scales well to large problem instances, yielding much better solve times than the SPP in GCS. A 2-3 step lookahead policy with a quadratic cost-to-go lower bound produces near-optimal solutions (8-9% median suboptimality) in under 10ms. The SPP in GCS produces slightly better solutions (7% median suboptimality), but due to the size of the graph, the solve-time increases to over 1000ms. For large graph instances,

Solution method	Optimality gap, %	Solve time, ms	Failure rate, %
Quadratic J_v, 1-step	20.0 (62.1)	3 (3)	0.0
Quadratic J_v, 2-step	9.4 (22.3)	4 (4)	0.0
Quadratic J_v, 3-step	8.8 (15.7)	5 (6)	0.0
Affine J_v, 1-step	157.1 (N/A)	2 (657)	27.2
Affine J_v, 2-step	142.4 (418.8)	3 (914)	14.0
Affine J_v, 3-step	80.2 (348.3)	5 (808)	9.9
Affine J_v, 8-step	11.9 (37.4)	169 (1996)	3.3
Affine J_v, 9-step	7.0 (26.2)	388 (2454)	0.0
SPP in GCS	6.9 (12.0)	716 (1051)	0.0

Table 4.1: Impact of the degree of J_v and lookahead horizon on performance, over 120 queries for the GCS in Figure 4.3. We report optimality gaps (ratio between solution cost and optimal cost), solve times, and failure rates (rollout policy is terminated after 10,000 iterations). We report median values, with the 75th percentile in the parenthesis. Low-horizon lookahead policies with quadratic lower bounds yield near optimal solutions, perform much better than the affine lower bounds.

incremental search through the graph via the multi-query SPP in GCS achieves competitive solution quality while reducing solve times by up to two-three orders of magnitude.

Finally, Table 4.1 shows that quadratic lower bounds with short-horizon lookaheads offer a good balance between expressive power and solve times. A 3-step lookahead policy with affine lower bounds has a median suboptimality of 80.2%, compared to 8.8% with quadratic lower bounds. Achieving similar solution quality with affine lower bounds requires a lookahead horizon of 8-9 steps, but the resulting rollouts take significantly more time. Figure 4.3 shows that 3-step lookahead rollouts with affine lower bounds fail to capture the diversity of optimal solutions. Additionally, low-horizon lookahead policies with affine lower bounds often fail to produce solutions within a reasonable number of iterations, as demonstrated by the failure rate statistics. Overall, we observe that the lookahead policies with quadratic lower bounds perform much better than those with affine ones.

We remark that the GCS used in this experiment is notably large, consisting of 190 vertices and 540 edges, with numerous cycles that complicate path planning. Despite this, quadratic cost-to-go lower bounds remain highly informative. Using only a 3-step lookahead greedy policy with respect to these lower bounds, we achieve near-optimal solutions, thus illustrating that our approach scales effectively to large problem instances.

4.3 Case-study: collision-free motion planning for a robot arm in a warehouse setting

We now evaluate our approach in a realistic scenario that reflects the motivation for our work: quickly and effectively solving planning problems for a robot in a fixed environment. We study

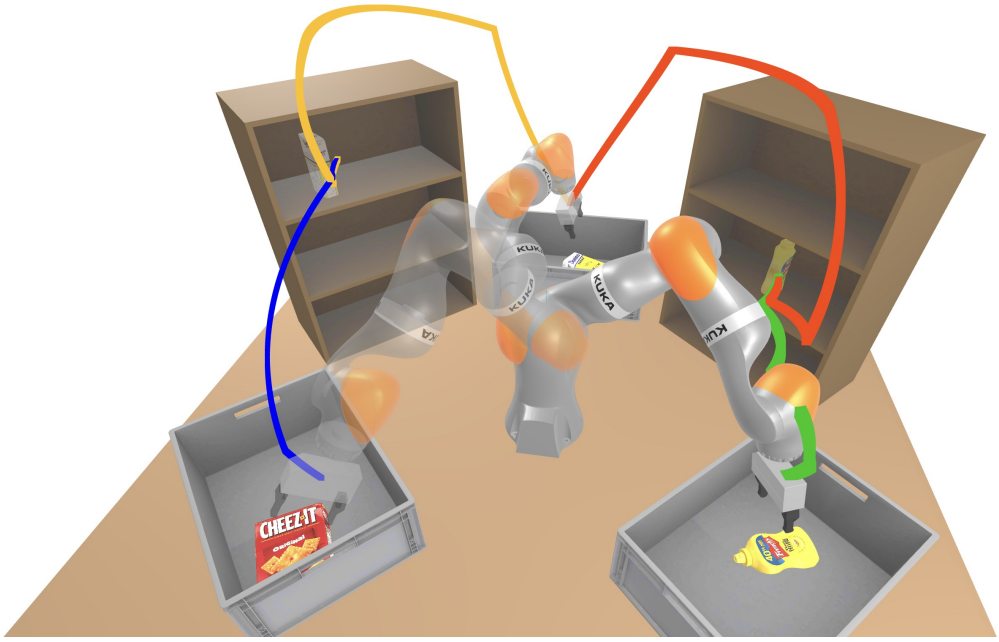


Figure 4.4: We repeat Figure 1.1 here for convenience and visual reference. Pictured is a robotic arm in a simulated environment, tasked with moving items between shelves and bins. Shown are four shortest-path queries for collision-free motion planning.

multi-query collision-free motion planning for the KUKA iiwa robotic arm (Figure 4.4), tasked with moving virtual items between shelves and bins. Our methodology requires minimal additional offline computation, while delivering significant online speed up with negligible solution quality reduction.

To use GCS for this problem, we follow the methodology from [3]. We first produce an approximate polytopic decomposition of the 7-dimensional collision-free configuration space of the arm. This is done via the IRIS-NP algorithm [61], and we use IRIS clique seeding [62] to obtain polytopes inside the shelves and bins. We assign a GCS vertex v per polytope in this decomposition. The convex set $\mathcal{X}_v$ is the set of linear segments contained within the collision-free region, with the segment represented by its endpoints. The vertex cost l_v is given by the Euclidean distance of the linear segment. Two GCS vertices are connected by an edge if the corresponding regions overlap, and additional edge constraints are added to ensure path continuity between adjacent segments; no edge costs are used. The resulting graph contains 23 vertices and 68 edges. We define 12 source vertices (6 shelves, 2 vertices per shelf) and 3 target vertices (inside the left, front, and right bins), and seek shortest paths between the source and target points. To generate the quadratic lower bounds on the shortest-path cost-to-go function, we use the generalization of (3.4) discussed in Appendix A.

We evaluate our algorithm in this multi-query scenario: at run-time, the arm is given a random next position to go to, alternating between shelves and bins. We rollout a 1-step lookahead policy to generate paths from shelves to bins, and reverse them to obtain paths from bins to shelves. We evaluate our approach on a total of 120 queries. We compare our algorithm against solving the SPP in GCS from scratch, as well as against the shortcut PRM (sPRM) algorithm, which is its natural sampling based multi-query competitor. We

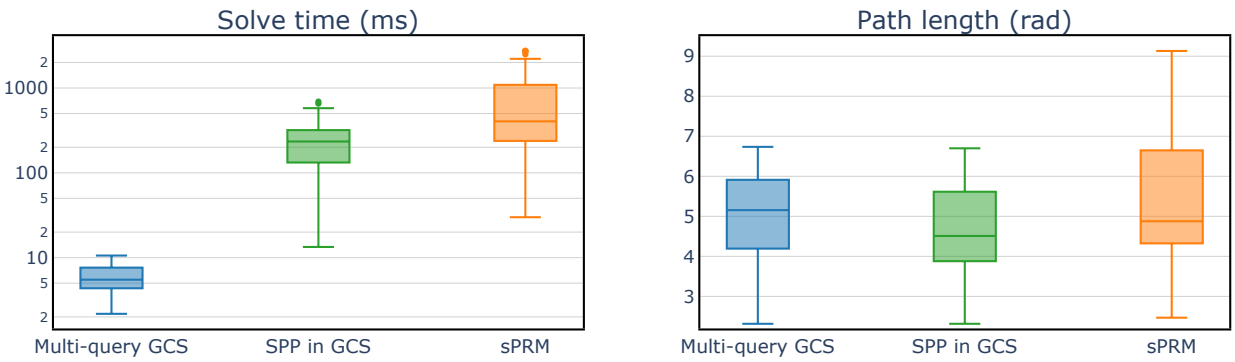


Figure 4.5: For the robot arm scenario in Section 4.3, we compare path length and solve time performance between the multi-query SPP in GCS, single-query SPP in GCS, and shortcut PRM over 120 queries. The offline phases take 106s, 100s, and 0.9s respectively. Multi-query SPP in GCS is on average 40 times faster than the SPP in GCS, with minimal reduction in solution quality. Compared to sPRM, our multi-query approach is on average 110 times faster.

use a high-performance implementation of sPRM based on [63], producing a large roadmap with 10,000 vertices. Our solutions are visualized in Figure 4.4; performance comparison is provided in Figure 4.5. Similar to how the quality of the PRM solutions depends on the density of the PRM, the quality of solutions obtained with GCS depends on the quality of the polytopic decomposition of the collision-free configuration space. We thus make no claims about the optimality of the solutions in this section.

Offline, generating shortest-path cost-to-go lower bounds takes only 6 seconds, which is just 6% of the time that it takes to generate the polytopic decomposition necessary to use GCS in the first place. Then online, our policy rollouts are very fast, with a median solve time of 5ms and a maximum of 11ms (we report parallelized solver time). Our method is on average 40 times faster than the SPP in GCS, producing paths that are only 7% longer on average. Compared to sPRM, our method is on average 110 times faster and produces paths that are 5% shorter on average. We achieve consistent performance in both solve time and path length, unlike sPRM, which shows high variance in both. Overall, compared to these state-of-the-art baselines, multi-query SPP in GCS reduces online solve times significantly, with minimal compromise in solution quality.

4.4 Qualitative demonstrations on robotic systems

In this section, to further demonstrate the applicability and effectiveness of our methodology, we qualitatively evaluate our approach across two broad categories of robotic planning problems. These include skill chaining, exemplified by item sorting with a suction gripper, and optimal control of hybrid systems, demonstrated through bipedal footstep planning. For each category, we formulate a general problem statement, recast it as planning in GCS, and apply our multi-query framework to an experimental setup. We note that the following experiments are run on a laptop with an Apple M1 MAX chip with 16GB of RAM, which is slower than the desktop machine used in the previous sections.

4.4.1 Skill chaining

Problem statement

Consider a robotic system controlled by a discrete set of continuously parameterized skills (also commonly referred to as motion primitives, actions, or behaviors in the literature) that use low-level control policies to transition between configurations. Abstracting away the low-level dynamics of these policies, the goal of skill chaining is to select a sequence of skills and the corresponding control parameters that achieve the target state [64, 65]. In the literature, related families of problems include sequential composition [66, 67] and Task and Motion Planning (see [68] for a comprehensive review).

Formally, given an n-dimensional configuration space, we define each skill π via a set $\mathcal{Q}_\pi \subset \mathbb{R}^{2n}$ of feasible configuration transitions $(q, q') \in \mathcal{Q}_\pi$ that can be achieved by this skill. Note that alternative definitions in the literature [66–71] describe skills through preconditions (or a pre-image, domain, initiation set) and effects (or a reachable set, goal set, termination condition), but these are generally interchangeable. Each skill also has an associated cost function $c_\pi : \mathcal{Q}_\pi \to \mathbb{R}_+$, where $c_\pi(q, q')$ is the cost of the transition from configuration q to q' under this skill. We assume that we are given sets of start and target configurations $\mathcal{C}_s \subset \mathbb{R}^n$ and $\mathcal{C}_t \subset \mathbb{R}^n$. Then, at run-time, we are presented with a pair of start and target configurations $\bar{q}_s \in \mathcal{C}_s, \bar{q}_t \in \mathcal{C}_t$, and our goal is to find a sequence of skills $(\pi_1, \ldots, \pi_K)$ and the sequence of corresponding transitions $\big((q_0, q_1), \ldots, (q_{K-1}, q_K)\big)$, such that $q_0 = \bar{q}_s$, $q_K = \bar{q}_t$, and each transition (q_{k-1}, q_k) is achieved via the skill π_k.

Various solution strategies for this problem have been developed in the literature. A common approach alternates between sampling discrete skills and continuous transitions (control parameters), guided by strong heuristics [72–76]. However, to be effective in complex environments, these methods often rely on costly hand-crafted samplers and may stall without them. In particular, these sampling-based approaches struggle when the desired poses lie on a lower-dimensional manifold, as the probability of sampling such poses may be zero. To more effectively explore the space of continuous transitions and better inform discrete search, other approaches use optimization-based subroutines [77–80]. Our GCS-based formulation is in this vein.

SWP in GCS transcription

To use GCS, we need the sets of source and target configurations $\mathcal{C}_s, \mathcal{C}_t$, the sets $\mathcal{Q}_\pi$ of transitions under each skill, and the cost functions c_π to be convex. If they are not, we assume that convex approximations or decompositions are available. In our GCS formulation, each skill π corresponds to a vertex with a convex set $\mathcal{X}_\pi = \mathcal{Q}_\pi$ and a vertex cost $l_\pi = c_\pi$. Visiting a vertex that correspond to skill π is thus equivalent to executing some transition (q, q') and incurring the cost $c_\pi(q, q')$. An edge connects two vertices if their skills can be chained: that is, if there exist configurations q_0, q_1, q_2 such that $(q_0, q_1) \in \mathcal{Q}_{\pi_1}$ and $(q_1, q_2) \in \mathcal{Q}_{\pi_2}$. Ensuring that the end point q_1 of the first skill is also the start point of the second skill requires adding an appropriate edge constraint. We then add a start vertex for the set of start configuration $\mathcal{C}_s$ and connect it to the vertices that represent skills executable from any $\bar{q}_s \in \mathcal{C}_s$. We add a target vertex in a similar fashion. Given a pair of start and target configurations $\bar{q}_s \in \mathcal{C}_s, \bar{q}_t \in \mathcal{C}_t$, the shortest walk in this GCS is exactly the solution to the skill chaining

problem: it is a sequence of skills $(\pi_1, \ldots, \pi_K)$ together with the corresponding sequence of transitions $\big((q_0, q_1), \ldots, (q_{K-1}, q_K)\big)$, where $q_0 = \bar{q}_s$ and $q_K = \bar{q}_t$.

Item sorting for a top-down suction gripper

To provide a concrete example, we now consider a variation of the canonical pick-and-place problem, illustrated in Figure 4.6. Our robot is a suction-cup gripper with a fixed vertical orientation, described by a 1D horizontal position. The environment contains three movable rectangular objects, described by their width, height, and the horizontal position. The robot can pick and place objects using top grasps at their centers, or flip objects clockwise and counterclockwise by grasping their corners. The task is to sort all objects into the target region (a green horizontal interval). At run-time, we are given the target interval, the objects' dimensions, and their initial positions, and the objective is to determine a sequence of actions that successfully sorts the items.

We cast this problem as a shortest walk in a GCS, following the construction above. We define the configuration space to include the horizontal positions of the objects and the arm, along with the objects' dimensions and the endpoints of the target interval. The robot's skills are defined as follows: (1) moving the arm from one position to another while the objects remain intact, (2) grasping an object at its center and placing it at a new collision-free location, and (3) flipping an object clockwise or counterclockwise by grasping its corner, thereby swapping its width and height. The cost of skill execution is defined as 1 plus the arm's horizontal displacement, and the sets $\mathcal{X}_\pi$ are defined to capture the feasible transitions under each skill. Naturally, these sets are not convex: this is due to the collision avoidance requirements and the combinatorial nature of selecting objects for manipulation. To address this, we decompose the skills into convex sub-skills, resulting in 6 arm movement skills, 3 pick-and-place skills, and 6 object-flipping skills. We define the source set to include all possible collision-free configurations, and the target set to include all possible collision-free configurations where the objects are inside the target region. Naturally, these sets are also non-convex: we decompose the source set into convex subsets and take the convex hull of the target set. The resulting GCS contains a vertex for every sub-skill, one target vertex, and 6 source vertices: a total of 22 vertices and 120 edges. We maximize the average value of the quadratic cost-to-go lower bounds over the source and target vertices, which takes only 20 seconds. The resulting structure efficiently supports shortest-walk queries for a variety of source conditions.

At run-time, the initial configuration is provided, and the objective is to move the objects into the target region. Notably, we do not prescribe a specific target configuration to achieve: rather, it is selected greedily during the policy rollout. We simply select the next skill to execute so as to greedily go down the cost-to-go lower bound (using a 2-step lookahead horizon) until the rollout is terminated at the target vertex.

In Figure 4.6, we demonstrate an example solution to the SWP in this GCS. We're given a random initial condition (top left). Here, object 1 is already in the target region, but it needs to be moved and reoriented to also fit object 2. This is a challenging puzzle, as we can barely fit all three objects within the target region. For this reason, sampling-based planners would struggle to find any solution at all: due to near-zero probability of sampling a feasible target configuration. In contrast, our approach produces an optimal solution and avoids

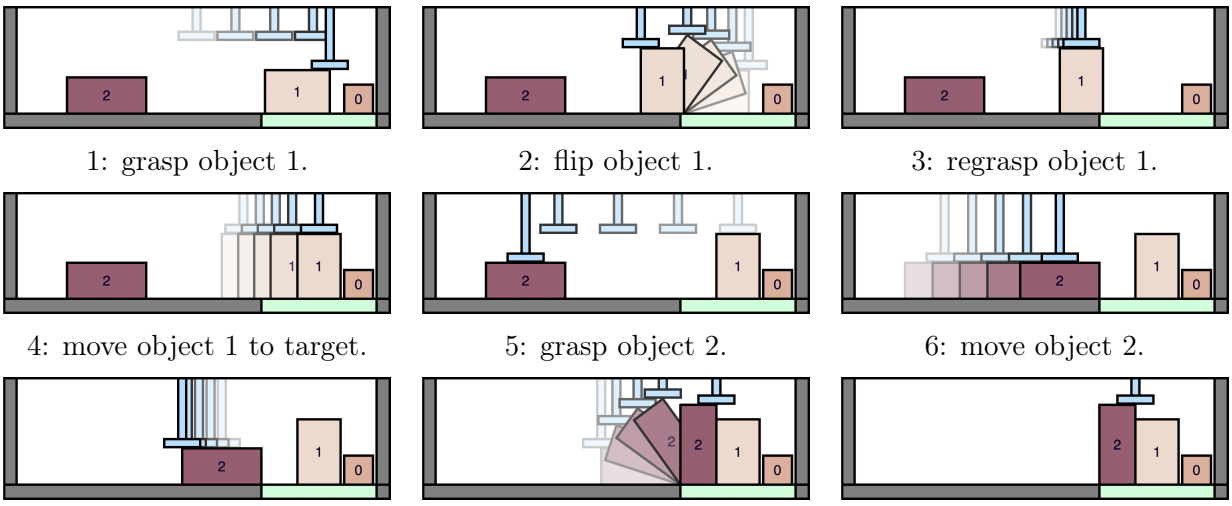


Figure 4.6: An 8-step plan where the top-down suction-cup arm is tasked with sorting three objects into the green target region.

unnecessary skill executions: for instance, object 0 remains intact, as it already satisfies the target conditions and doesn't need to be moved to fit the other objects. With appropriate pre-building of the programs used during incremental search, the solve times for producing this and similar plans range from 0.5 to 1 second.

4.4.2 Hybrid optimal control

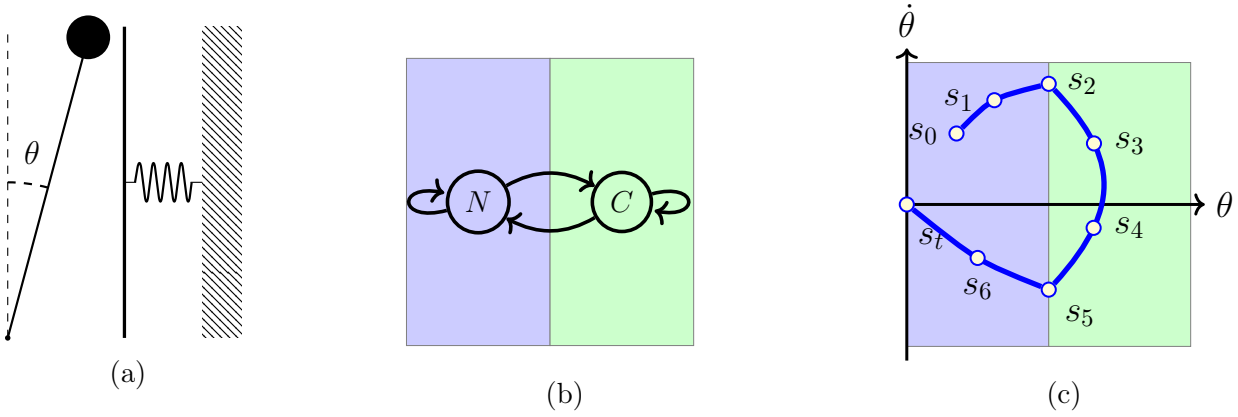
Problem statement

Many challenging problems in robotics, such as footstep planning, planning through contact, and dexterous manipulation, involve systems with hybrid dynamics. It is well known that such systems can be approximated arbitrarily-well with a Piecewise Affine (PWA) dynamical model [81–83]. Motivated by this, we consider the problem of optimal control for discrete-time PWA dynamical systems. We refer the reader to [84] for a recent review of approaches for hybrid systems control, which highlights the SPP in GCS as an effective and competitive strategy. Below we produce a shortest-walk formulation, which may offer an even more effective strategy.

Let $\mathcal{S}$ and $\mathcal{A}$ be our system's state and control spaces, and let the state-space be partitioned into closed, polyhedral sets $\mathcal{S} = \cup_i \mathcal{S}_i$, commonly referred to as modes. A PWA control system evolves according to different affine dynamics depending on the mode that the system is in. That is, the system's dynamics at time-step n are governed by:

$$s_{n+1} = A_i s_n + B_i a_n + c_i, \text{ if } s_n \in \mathcal{S}_i, \ a_n \in \mathcal{A}.$$

Executing control input a_n at state $s_n \in \mathcal{S}_i$ of mode i incurs the mode-specific stage cost $l_i(s_n, a_n)$. The PWA optimal control problem seeks a state, control, and mode trajectories between source and target states $\bar{s}_0$ and $\bar{s}_t$, satisfying the PWA dynamics and minimizing



(a) (b) (c)

Figure 4.7: An actuated pendulum with a soft wall (a) can be approximated as a PWA system with two modes (no-contact and contact), which can be modeled as a GCS with two vertices and four edges (b). An optimal state-space trajectory for regulating the pendulum to the equilibrium position $s_t = (0, 0)$ can be computed by solving a SWP in GCS, shown in (c).

the total stage cost. In the multi-query generalization, the goal is to find state, control, and mode trajectories between any pair of states in the given sets of initial and target conditions.

SWP in GCS transcription

Hybrid optimal control can be naturally cast as a shortest walk in a GCS. For every mode i, we define a GCS vertex i with a convex set $\mathcal{X}_i = \mathcal{S}_i \times \mathcal{A}$. The source vertex is added similarly, while the target is just the set of target states. Two vertices are connected with an edge if there exists a feasible transition between some pair of states in the corresponding modes. Affine dynamics are imposed as edge constraints, and the convex stage cost l_i is added as a vertex cost. The shortest walk in this GCS is a vertex sequence w, corresponding to a PWA mode trajectory, and a sequence of points τ, corresponding to state and control trajectories.

As a simple and concrete visual example, we demonstrate this construction using a pendulum with a soft wall — a canonical benchmark for control through contact — illustrated in Figure 4.7. The system has two contact modes: C for contact and N for no-contact. Prior to adding the source and target vertices, the resulting GCS consists of two vertices and four edges, as shown in Figure 4.7b. The dynamics are imposed on the edges by linearizing the nonlinear dynamics for each mode. In particular, note that the linear dynamics along the edges (N, C) and (C, N) are different. An example trajectory is depicted in Figure 4.7c.

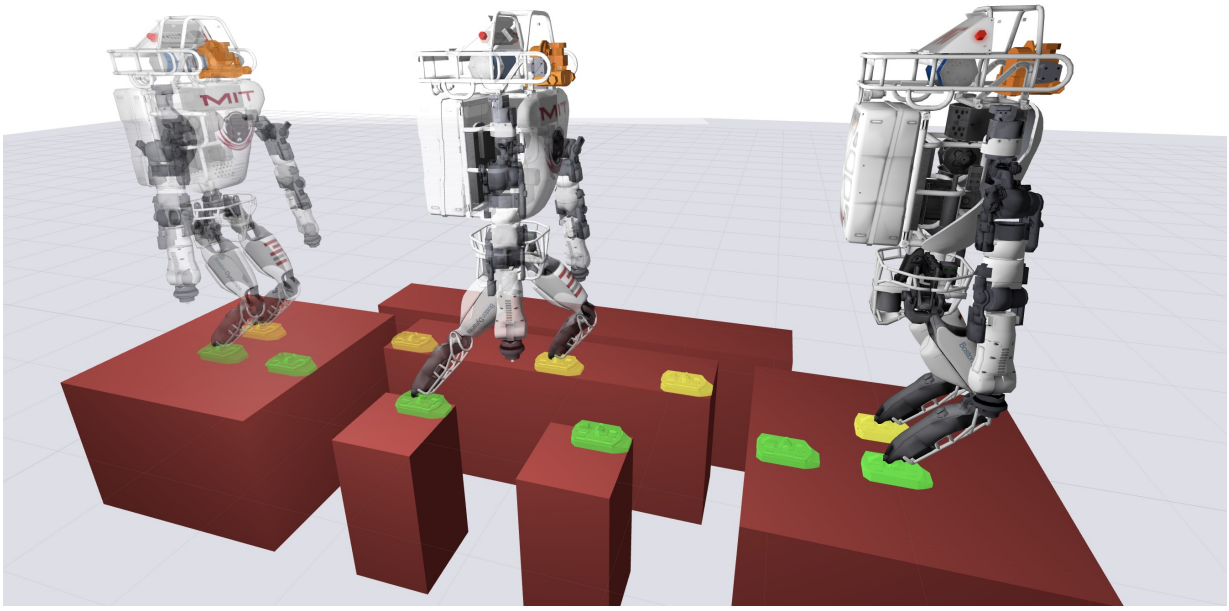
Footstep planning for a ZMP walker

We now turn to a more complex robotic example: footstep planning for a bipedal robot navigating over stepping stones in a flat 2D x-y plane. The robot, shown in Figure 4.8, must reach the target by planning a sequence of footsteps and contact forces through the stepping stones, ensuring stability and avoiding foot collisions.

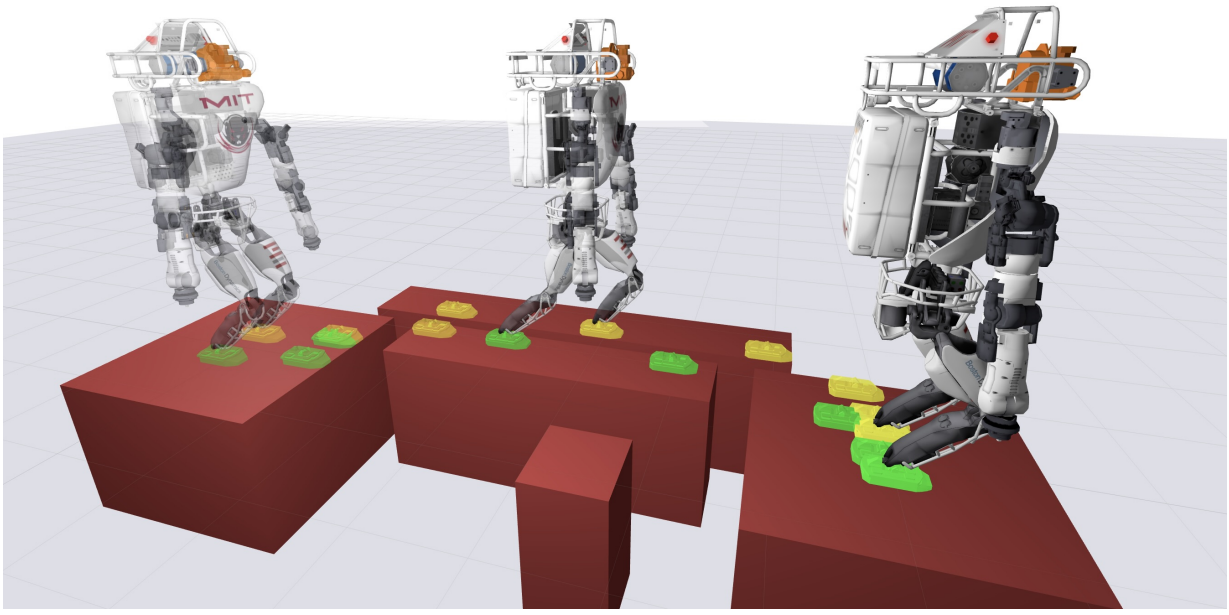
We use the the well-known Zero-Moment Point (ZMP) formulation to model the robot's dynamics; see [85, 86] for classic reviews. We constrain the robot's Center of Pressure (CoP)

(which is also the ZMP) to remain within the support polygon formed by the feet — this ensures that the robot can generate the ground reaction forces necessary to maintain dynamic stability. We also constrain the ground reaction forces to lie inside the friction cone. We assume that the robot has massless legs, that the acceleration along the vertical axis is zero, and that the robot does not rotate (i.e., zero angular acceleration). This results in an affine relationship between the robot's CoP and CoM dynamics. The dynamics become piecewise affine when we account for footstep planning, with three primary contact modes: both feet on the ground, only the left foot, or only the right foot. Foot placement on stepping stones introduces up to $O(N^2)$ potential contact modes (N options for each foot), but many are infeasible due to constraints on the distance between the feet. In practice, the modes scale as $O(DN)$, where D is the average number of adjacent stones. Additional constraints ensure collision avoidance between the feet. The resulting PWA system jointly considers safe footstep placement, contact forces, centroidal dynamics, and stability enforced by the ZMP condition.

Following the formulation above, we cast the discrete-time trajectory planning problem for this PWA system as the SWP in a GCS. A GCS vertex is added for each PWA mode, and a target vertex is added to represent the desired location. This results in a GCS with 24 vertices and 58 edges for the scenario in Figure 4.8a, and 21 vertices and 50 edges for the scenario in Figure 4.8b. We compute quadratic cost-to-go lower bounds, maximizing their average value across all vertices, which takes 2.3 and 3.7 seconds respectively. Guided by these lower bounds, the 2-step lookahead greedy search generates an 10-step plan in Figure 4.8a within 110ms and a 16-step plan in Figure 4.8b within 330ms. Multi-step lookahead greedy search effectively functions as a receding-horizon control policy, counteracting disturbances and mitigating the effects of imperfect plan tracking. At run-time, we can rollout greedy search until we run out of the fixed time budget, execute the first step, and iteratively replan from the resulting state. Similarly, the computed quadratic cost-to-go lower bounds can also be effectively used as part of a finite horizon MPC policy.



(a) A footstep plan for a bipedal robot, navigating a set of stepping stones.



(b) With one stone removed, the robot must take a longer detour.

Figure 4.8: Visualizations of the footstep plans across stepping stones for the Atlas bipedal robot. Our approach jointly optimizes for safe footstep placement, contact forces, and centroidal dynamics, while maintaining stability enforced by the ZMP condition. Both plans are produced in under 330ms on a laptop, and can be run online in MPC fashion as a policy.

Chapter 5

Conclusions

5.1 Summary

In this thesis, we considered the problem of multi-query planning in Graphs of Convex Sets. We generalized the classical all-pairs shortest-path problem to the GCS, and developed practical approximate numerical methods for solving the resulting problem, both for the shortest walks and the shortest paths in GCS. We demonstrated that a coarse lower bound on the cost-to-go with a multi-step lookahead greedy policy effectively produce near-optimal solutions, while significantly reducing solve times. Our methodology scales well to high-dimensional scenarios and large graph instances, enabling practical applications in multi-query robotic settings, including collision-free motion planing, skill chaining, and optimal control for hybrid systems.

5.2 Limitations

First, using GCS inherently requires the effort to construct it, which can be computationally intensive and may involve manual tuning. This is the case with generating collision-free regions with the IRIS algorithm in Section 4.3, convex skill decomposition in Section 4.4.1, and PWA decomposition of the dynamics in Section 4.4.2. While potentially tedious, this stage is an unavoidable part of working with the GCS framework.

Next, the offline cost-to-go synthesis step requires solving a potentially large SDP. For large graphs and high-dimensional robotic systems, these programs can become massive, and conic solvers like MOSEK [60] and Clarabel [87] often struggle to handle them. One key challenge is the significant memory required to store and manipulate big constraint matrices, cones, and solution matrices, which can quickly exceed available RAM for particularly large GCS instances. Additionally, these solvers rely heavily on numerical linear algebra, and depending on the choice of convex sets and constraints, the resulting matrices may be sparse or ill-conditioned, leading to numerical instability.

Finally, achieving the fast solve times reported for the online greedy search stage requires pre-compiling programs into binaries and solving them in parallel, as described in Section 3.5. Specifically, we assumed the ability to solve up to 10 programs in parallel and reported simulated parallelized solve times based on this assumption. Achieving these solve times in

practice would require additional engineering effort, which should be considered as part of the overall implementation cost.

Appendix A

Cost-to-go synthesis: extensions and variations

We briefly remark on various natural generalizations to programs (3.3) and (3.4). In particular, we focus on (3.4), as (3.3) can be derived from it by setting the vertex penalties to zero.

1. Suppose the set of source vertices $\mathcal{S}$ has more than one vertex. To simultaneously "push up" lower bounds J_s per vertex $s \in \mathcal{S}$, we add extra integral terms to the objective function (3.4a).

2. Suppose the target set $\mathcal{X}_t$ is not a singleton, but a compact convex set. First, we modify the constraint (3.4b) to search for $J_{v,t} : \mathcal{X}_v \times \mathcal{X}_t \to \mathbb{R}$. Here, the function $J_{v,t}(x_v, x_t)$ is a lower bound on the cost-to-go of the shortest path from x_v of vertex v to x_t of vertex t. Similarly, the probability distribution $\phi_{s,t}$ is now supported on $\mathcal{X}_s \times \mathcal{X}_t$: it is the probability distribution over anticipated source-target pairs (x_s, x_t), so as to push up on $J_{s,t}(x_s, x_t)$. The lower-bound constraint (3.4c) is adjusted to include $x_t \in \mathcal{X}_t$:

$$J_{u,t}(x_u, x_t) \leq l_u(x_u) + l_e(x_u, x_v) + h_v + J_{v,t}(x_v, x_t),$$

   for all edges $e = (u, v) \in \mathcal{E}$, and all points $(x_u, x_v) \in \mathcal{X}_e$ and $x_t \in \mathcal{X}_t$. Finally, the target constraint (3.4d) is adjusted to be $J_{t,t}(x_t, x_t) = l_t(x_t) - \sum_{v \in \mathcal{V}} h_v$, for all $x_t \in \mathcal{X}_t$.

3. The scalar vertex penalty h_v is generalized to be a non-negative function of the target state x_t, that is: $h_{v,t} : \mathcal{X}_t \to \mathbb{R}_+$. We thus replace h_v with $h_{v,t}(x_t)$ in (3.4c) and update the constraint (3.4d) as follows:

$$J_{t,t}(x_t, x_t) = l_t(x_t) - \sum_{v \in \mathcal{V}} h_v(x_t),$$

   further tightening the resulting lower bounds.

4. Suppose the set of target vertices $\mathcal{T}$ has more than one vertex. To obtain the cost-to-go lower bounds for every pair of vertices $v \in \mathcal{V}$ and $t \in \mathcal{T}$, we solve multiple programs (3.4) in parallel, one per target vertex $t \in \mathcal{T}$.

5. In general, the greedy policy (3.7) is also a function of target vertex t and target point x_t. The relevant adjustments to the policy are straight-forward.

6. Other penalties, similar to the vertex visitation penalties h_v, can be added to improve the quality of the lower bounds. For instance, consider a 2-cycle with edges (u, v) and (v, u). We can add edge penalties $h_{u,v} = h_{v,u}$ for traversing either edge. By subtracting $h_{u,v}$ from the cost-to-go lower bound at the target, we effectively ensure that no penalty is incurred for traversing just one (but not both) of the edges. This can be extended to cycles of arbitrary length.

Appendix B

Best-first search for provably-optimal solutions

We now outline an alternative incremental search policy to the greedy policy discussed in Section 3.5. Although greedy search quickly and effectively produces solutions to the SPP and the SWP in GCS, it is heuristic (not guaranteed to return optimal solutions), and is not complete (may fail to yield a solution if one exists). Best-first search addresses both of this problems. We describe the algorithm for the shortest paths; the algorithm for the shortest walks is nearly identical, with the modifications mirroring the ones discussed in Section 3.5.

Suppose that at run-time, we are given a source vertex $s = v_0 \in \mathcal{S}$ and a source point $\bar{x}_s = x_0 \in \mathcal{X}_{v_0}$, and our goal is to find a shortest path to the point $\bar{x}_t$ of the target vertex t. We maintain a priority queue of candidate subpaths $p = (v_0, \dots, v_k)$, initiated simply with the subpath $p = (v_0)$. Each subpath in the queue is a candidate prefix to the shortest path in this GCS: a candidate for the beginning of the vertex sequence that may lead to a full solution. Given a candidate subpath $p = (v_0, \dots, v_k)$, we can evaluate its value $J(p)$ by solving the following optimization problem:

$$J(p) = \min_{\tau} \quad l(p, \tau) + J_{v_k}(x_k) - l_{v_k}(x_k) \tag{B.1a}$$

$$\text{s.t.} \quad \tau = (x_0, \dots, x_k) \in \mathcal{T}_p(\bar{x}_s, x_k), \tag{B.1b}$$

which searches for an optimal subtrajectory τ along the subpath p and uses the cost-to-go lower bound $J_{v_k}(x_k)$ to estimate the cost of the remainder of the shortest path. In the language of A*, $l(p, \tau)$ is the cost-to-come, $J_{v_k}(x_k)$ is the lower bound on the cost-to-go, and we subtract $l_{v_k}(x_k)$ to avoid double counting, just as in Section 3.5. Thus, for a candidate subpath p, $J(p)$ is the lower bound on the cost of a full path between $(s, \bar{x}_s)$ and $(t, \bar{x}_t)$ that is prefixed with p.

At each iteration of the best-first search algorithm, we pop from the priority queue a candidate subpath p with the lowest value $J(p)$. We then examine the last vertex v_k of this subpath and consider every vertex v that can be reached from v_k (i.e., $(v_k, v) \in \mathcal{E}$) that has not already been visited ($v \notin p$). For each such vertex v, we extend the subpath p by appending v to it, thus generating a new candidate subpath. If the program (B.1) is infeasible for this new candidate subpath, then there exists no feasible trajectory along that subpath, and it can be safely discarded. New candidate subpaths that are feasible are added to the

queue, and the process continues by popping the next candidate subpath from the queue. If a path from s to t is popped, the constraint (B.1b) is updated to $\tau \in \mathcal{T}_p(\bar{x}_s, \bar{x}_t)$ to ensure that the trajectory ends with the target point $\bar{x}_t$. If the resulting program is feasible, then we terminate the process: the resulting tuple (p, τ) is guaranteed to be a shortest path in this GCS. This is because all remaining candidate subpaths in the queue have costs at least as high as the current path, rendering further exploration unnecessary. If the queue becomes empty, then we must have considered every feasible candidate subpath that originates at $(s, \bar{x}_s)$, yet none of them reach $(t, \bar{x}_t)$. We thus have a proof that no solution exists and can terminate the search.

The number of candidate subpaths considered during best-first search depends on the quality of the cost-to-go lower bounds; in general, it can be exponential in the length of the shortest path (or walk). For this reason, we rely on this approach for obtaining ground-truth optimal solutions, not to obtain fast online queries. We note that the number of considered subpaths can be reduced via dominance checks, which allow pruning some candidate subpaths, as described in [11].

References

[1] T. Marcucci, J. Umenberger, P. Parrilo, and R. Tedrake. "Shortest paths in graphs of convex sets". In: *SIAM Journal on Optimization* 34.1 (2024), pp. 507–532.

[2] T. Marcucci. "Graphs of Convex Sets with Applications to Optimal Control and Motion Planning". PhD thesis. Massachusetts Institute of Technology, 2024.

[3] T. Marcucci, M. Petersen, D. von Wrangel, and R. Tedrake. "Motion planning around obstacles with convex optimization". In: *Science robotics* 8.84 (2023), eadf7843.

[4] D. von Wrangel and R. Tedrake. "Using Graphs of Convex Sets to Guide Nonconvex Trajectory Optimization". In.

[5] T. Cohn, M. Petersen, M. Simchowitz, and R. Tedrake. "Non-Euclidean Motion Planning with Graphs of Geodesically-Convex Sets". In: *Robotics: Science and Systems* (2023).

[6] B. P. Graesdal, S. Y. Chia, T. Marcucci, S. Morozov, A. Amice, P. A. Parrilo, and R. Tedrake. "Towards Tight Convex Relaxations for Contact-Rich Manipulation". In: *Robotics: Science and Systems* (2024).

[7] A. G. Philip, Z. Ren, S. Rathinam, and H. Choset. "A Mixed-Integer Conic Program for the Moving-Target Traveling Salesman Problem based on a Graph of Convex Sets". In: *arXiv preprint arXiv:2403.04917* (2024).

[8] V. Kurtz and H. Lin. "Temporal Logic Motion Planning With Convex Optimization via Graphs of Convex Sets". In: *IEEE Transactions on Robotics* 39.5 (2023), pp. 3791–3804.

[9] S. Morozov, T. Marcucci, A. Amice, B. P. Graesdal, R. Bosworth, P. A. Parrilo, and R. Tedrake. "Multi-query shortest-path problem in graphs of convex sets". In: *arXiv preprint arXiv:2409.19543* (2024).

[10] R. Natarajan, C. Liu, H. Choset, and M. Likhachev. "Implicit graph search for planning on graphs of convex sets". In: *arXiv preprint arXiv:2410.08909* (2024).

[11] S. Y. C. Chia, R. H. Jiang, B. P. Graesdal, L. P. Kaelbling, and R. Tedrake. "GCS*: Forward Heuristic Search on Implicit Graphs of Convex Sets". In: *arXiv preprint arXiv:2407.08848* (2024).

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

[13] R. Bellman. "Dynamic programming". In: *science* 153.3731 (1966), pp. 34–37.

[14] R. M. Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[15] R. Tedrake and the Drake Development Team. *Drake: Model-based design and verification for robotics*. 2019. URL: https://drake.mit.edu.

[16] K. Sundar and S. Rathinam. "A* for Graphs of Convex Sets". In: *arXiv preprint arXiv:2407.17413* (2024).

[17] R. Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515.

[18] D. Bertsekas. *Dynamic programming and optimal control*. Vol. 4. Athena scientific, 2012.

[19] R. W. Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), pp. 345–345.

[20] S. Warshall. "A theorem on boolean matrices". In: *Journal of the ACM (JACM)* 9.1 (1962), pp. 11–12.

[21] D. B. Johnson. "Efficient algorithms for shortest paths in sparse networks". In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13.

[22] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah. "A survey of shortest-path algorithms". In: *arXiv preprint arXiv:1705.02044* (2017).

[23] M. Thorup and U. Zwick. "Approximate distance oracles". In: *Journal of the ACM (JACM)* 52.1 (2005), pp. 1–24.

[24] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. "Fast estimation of diameter and shortest paths (without matrix multiplication)". In: *SIAM Journal on Computing* 28.4 (1999), pp. 1167–1181.

[25] D. Dor, S. Halperin, and U. Zwick. "All-pairs almost shortest paths". In: *SIAM Journal on Computing* 29.5 (2000), pp. 1740–1759.

[26] R. E. Kalman et al. "Contributions to the theory of optimal control". In: *Bol. soc. mat. mexicana* 5.2 (1960), pp. 102–119.

[27] B. D. Anderson and J. B. Moore. *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

[28] D. Kleinman. "On an iterative technique for Riccati equation computations". In: *IEEE Transactions on Automatic Control* 13.1 (1968), pp. 114–115.

[29] A. Laub. "A Schur method for solving algebraic Riccati equations". In: *IEEE Transactions on automatic control* 24.6 (1979), pp. 913–921.

[30] W. F. Arnold and A. J. Laub. "Generalized eigenproblem algorithms and software for algebraic Riccati equations". In: *Proceedings of the IEEE* 72.12 (1984), pp. 1746–1754.

[31] P. Lancaster. *Algebraic Riccati Equations*. Oxford Science Publications/The Clarendon Press, Oxford University Press, 1995.

[32] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos. "The explicit linear quadratic regulator for constrained systems". In: *Automatica* 38.1 (2002), pp. 3–20.

[33] A. Alessio and A. Bemporad. "A survey on explicit model predictive control". In: *Nonlinear Model Predictive Control: Towards New Challenging Applications* (2009), pp. 345–369.

[34] A. S. Manne. "Linear programming and sequential decisions". In: *Management Science* 6.3 (1960), pp. 259–267.

[35] E. V. Denardo. "On linear programming in a Markov decision problem". In: *Management Science* 16.5 (1970), pp. 281–288.

[36] P. J. Schweitzer and A. Seidmann. "Generalized polynomial approximations in Markovian decision processes". In: *Journal of mathematical analysis and applications* 110.2 (1985), pp. 568–582.

[37] D. P. De Farias and B. Van Roy. "The linear programming approach to approximate dynamic programming". In: *Operations research* 51.6 (2003), pp. 850–865.

[38] W. B. Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons, 2007.

[39] D. Bertsekas. "Neuro-dynamic programming". In: *Athena Scientific* (1996).

[40] R. S. Sutton. "Reinforcement learning: An introduction". In: *A Bradford Book* (2018).

[41] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[42] R. S. Sutton. "Learning to predict by the methods of temporal differences". In: *Machine learning* 3 (1988), pp. 9–44.

[43] V. Konda and J. Tsitsiklis. "Actor-critic algorithms". In: *Advances in neural information processing systems* 12 (1999).

[44] M. Riedmiller. "Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method". In: *Machine learning: ECML 2005: 16th European conference on machine learning, Porto, Portugal, October 3-7, 2005. proceedings 16*. Springer. 2005, pp. 317–328.

[45] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[46] J. B. Lasserre, D. Henrion, C. Prieur, and E. Trélat. "Nonlinear optimal control via occupation measures and LMI-relaxations". In: *SIAM journal on control and optimization* 47.4 (2008).

[47] Y. Wang, B. O'Donoghue, and S. Boyd. "Approximate dynamic programming via iterated Bellman inequalities". In: *International Journal of Robust and Nonlinear Control* 25.10 (2015).

[48] L. Yang, H. Dai, A. Amice, and R. Tedrake. "Approximate Optimal Controller Synthesis for Cart-Poles and Quadrotors via Sums-of-Squares". In: *IEEE Robotics and Automation Letters* (2023).

[49] S. Prajna, P. A. Parrilo, and A. Rantzer. "Nonlinear control synthesis by convex optimization". In: *IEEE Transactions on Automatic Control* 49.2 (2004), pp. 310–314.

[50] S. Prajna, A. Papachristodoulou, and F. Wu. "Nonlinear control synthesis by sum of squares optimization: A Lyapunov-based approach". In: *2004 5th Asian control conference (IEEE Cat. No. 04EX904)*. Vol. 1. IEEE. 2004, pp. 157–165.

[51] R. Tedrake, I. R. Manchester, M. Tobenkin, and J. W. Roberts. "LQR-trees: Feedback motion planning via sums-of-squares verification". In: *The International Journal of Robotics Research* 29.8 (2010), pp. 1038–1052.

[52] A. Majumdar and R. Tedrake. "Funnel libraries for real-time robust feedback motion planning". In: *The International Journal of Robotics Research* 36.8 (2017), pp. 947–982.

[53] A. Clark. "Verification and synthesis of control barrier functions". In: *2021 60th IEEE Conference on Decision and Control (CDC)*. IEEE. 2021, pp. 6105–6112.

[54] H. Wang, K. Margellos, and A. Papachristodoulou. "Safety verification and controller synthesis for systems with input constraints". In: *IFAC-PapersOnLine* 56.2 (2023), pp. 1698–1703.

[55] G. Blekherman, P. A. Parrilo, and R. R. Thomas. *Semidefinite optimization and convex algebraic geometry*. SIAM, 2012.

[56] P. A. Parrilo. *Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization*. California Institute of Technology, 2000.

[57] P. A. Parrilo. "Semidefinite programming relaxations for semialgebraic problems". In: *Mathematical programming* 96 (2003), pp. 293–320.

[58] J. B. Lasserre. "Global optimization with polynomials and the problem of moments". In: *SIAM Journal on optimization* 11.3 (2001), pp. 796–817.

[59] M. Grant, S. Boyd, and Y. Ye. *Disciplined convex programming*. Springer, 2006.

[60] M. ApS. *The MOSEK optimization toolbox for MATLAB manual. Version 10.1.* 2024. URL: http://docs.mosek.com/latest/toolbox/index.html.

[61] M. Petersen and R. Tedrake. "Growing convex collision-free regions in configuration space using nonlinear programming". In: *arXiv preprint arXiv:2303.14737* (2023).

[62] P. Werner, A. Amice, T. Marcucci, D. Rus, and R. Tedrake. "Approximating robot configuration spaces with few convex sets using clique covers of visibility graphs". In: *International Conference on Robotics and Automation* (2024).

[63] C. Phillips-Grafflin. "Common robotics utilities". In: URL: https://github.com/ToyotaResearchInstitute/common_robotics_utilities.

[64] R. S. Sutton, D. Precup, and S. Singh. "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning". In: *Artificial intelligence* 112.1-2 (1999), pp. 181–211.

[65] G. Konidaris and A. Barto. "Skill discovery in continuous reinforcement learning domains using skill chaining". In: *Advances in neural information processing systems* 22 (2009).

[66] R. R. Burridge, A. A. Rizzi, and D. E. Koditschek. "Sequential composition of dynamically dexterous robot behaviors". In: *The International Journal of Robotics Research* 18.6 (1999), pp. 534–555.

[67] R. Tedrake et al. "LQR-trees: Feedback motion planning on sparse randomized trees." In: *Robotics: Science and Systems*. Vol. 2009. 2009.

[68] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez. "Integrated task and motion planning". In: *Annual review of control, robotics, and autonomous systems* 4.1 (2021), pp. 265–293.

[69] R. E. Fikes and N. J. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving". In: *Artificial intelligence* 2.3-4 (1971), pp. 189–208.

[70] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. Sri, A. Barrett, D. Christianson, et al. "PDDL| the planning domain definition language". In: *Technical Report, Tech. Rep.* (1998).

[71] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[72] S. Cambon, R. Alami, and F. Gravot. "A hybrid approach to intricate motion, manipulation and task planning". In: *The International Journal of Robotics Research* 28.1 (2009), pp. 104–126.

[73] L. P. Kaelbling and T. Lozano-Pérez. "Hierarchical task and motion planning in the now". In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 1470–1477.

[74] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. Russell, and P. Abbeel. "Combined task and motion planning through an extensible planner-independent interface layer". In: *2014 IEEE international conference on robotics and automation (ICRA)*. IEEE. 2014, pp. 639–646.

[75] A. Krontiris and K. E. Bekris. "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)*. IEEE. 2016, pp. 3924–3931.

[76] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling. "PDDLStream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning". In: *Proceedings of the international conference on automated planning and scheduling*. Vol. 30. 2020, pp. 440–448.

[77] M. Toussaint. "Logic-Geometric Programming: An Optimization-Based Approach to Combined Task and Motion Planning." In: *IJCAI*. 2015, pp. 1930–1936.

[78] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada. "SMC: Satisfiability modulo convex programming". In: *Proceedings of the IEEE* 106.9 (2018), pp. 1655–1679.

[79] D. Hadfield-Menell, C. Lin, R. Chitnis, S. Russell, and P. Abbeel. "Sequential quadratic programming for task plan optimization". In: *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE. 2016, pp. 5040–5047.

[80] E. Fernandez-Gonzalez, B. Williams, and E. Karpas. "Scottyactivity: Mixed discrete-continuous planning with convex optimization". In: *Journal of Artificial Intelligence Research* 62 (2018), pp. 579–664.

[81] E. Sontag. “Nonlinear regulation: The piecewise linear approach”. In: *IEEE Transactions on automatic control* 26.2 (1981), pp. 346–358.

[82] E. D. Sontag. “Interconnected automata and linear systems: A theoretical framework in discrete-time”. In: *International Hybrid Systems Workshop*. Springer. 1995, pp. 436–448.

[83] A. Bemporad and M. Morari. “Control of systems integrating logic, dynamics, and constraints”. In: *Automatica* 35.3 (1999), pp. 407–427.

[84] J. Lee, H. Im, and A. Atamtürk. “Strong Formulations for Hybrid System Control”. In: *arXiv preprint arXiv:2412.11541* (2024).

[85] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa. “Biped walking pattern generation by using preview control of zero-moment point”. In: *2003 IEEE international conference on robotics and automation (Cat. No. 03CH37422)*. Vol. 2. IEEE. 2003, pp. 1620–1626.

[86] M. Vukobratović and B. Borovac. “Zero-moment point—thirty five years of its life”. In: *International journal of humanoid robotics* 1.01 (2004), pp. 157–173.

[87] P. J. Goulart and Y. Chen. “Clarabel: An interior-point solver for conic programs with quadratic objectives”. In: *arXiv preprint arXiv:2405.12762* (2024).