Plan Optimal Collision-Free Trajectories With Nonconvex Cost Functions Using Graphs of Convex Sets

Charles L. Clark, *Student Member, IEEE*, and Biyun Xie, *Senior Member, IEEE*

***Abstract*—The recently developed approach to motion planning in graphs of convex sets (GCS) provides an efficient framework for computing shortest-distance collision-free paths using convex optimization. This new motion planner is notably more computationally efficient than popular sampling-based motion planners, but it does not support nonconvex cost functions. This article develops a novel motion planning algorithm, graph of convex sets with general costs (GCSGC), to solve this problem. A given nonconvex cost function is accurately approximated by a multiple-layer ReLU neural network and the configuration space is decomposed into a set of linear-cost regions using the hidden layers of the neural network. These linear-cost regions are intersected with a set of collision-free regions, and the resulting collision-free linear-cost regions are intersected to form the vertices and edges of the motion planner's underlying graph structure. The edge costs have a closed-form solution within each collision-free linear-cost region, but it is nonconvex, so the McCormick relaxation is applied to convexify the edge costs. Finally, a graph preprocessing technique is developed to compute a representative graph structure that acts as a heuristic for the edge costs of the underlying GCS and then simplify the underlying graph structure by removing cycles and high-cost paths, which can significantly improve the efficiency of the planner and quality of the produced trajectories. The proposed motion planner is first validated in a 2-D configuration space with comparisons between different sized neural networks with and without preprocessing, comparisons between optimal trajectories from GCSGC with shortest-distance trajectories, and comparisons between GCSGC and GCS-Sequential linear programming (SLP). The GCSGC planner is further validated in a complex 7-D configuration space by comparing to state-of-the-art multiquery (PRM*, GCS-SLP) and single-query (TrajOpt, BIT*, AIT*, RRT*) planners. The results show that the proposed motion planner is very competitive in terms of computational efficiency, trajectory cost, and memory footprint. Two physical experiments further validate the effectiveness of the proposed motion planner in real-world motion planning applications.**

***Index Terms*—Collision avoidance, convex optimization, motion and path planning, optimization and optimal control.**

Received 27 June 2025; accepted 30 August 2025. Date of publication 15 September 2025; date of current version 30 September 2025. This work was supported by the NSF under Grant #2205292 and Grant #2441654. This article was recommended for publication by Associate Editor Mehmet R Dogar and Editor Jing Xiao upon evaluation of the reviewers' comments. *(Corresponding author: Biyun Xie.)*

The authors are with the Electrical and Computer Engineering Department, University of Kentucky, Lexington, KY 40506 USA (e-mail: landon.clark@uky.edu; biyun.xie@uky.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TRO.2025.3610175, provided by the authors.

Digital Object Identifier 10.1109/TRO.2025.3610175

I. Introduction

A WIDE variety of systems, such as robot arms, drones, and autonomous vehicles, require safe paths that avoid collisions with environmental obstacles and optimal paths that minimize the path cost, such as distance or energy. This problem is highly challenging because it is difficult to ensure that the entire path is collision-free in complex and high-dimensional search spaces. In addition, optimizing an entire path is also a very difficult problem, as there are infinitely many potential paths. Many different motion planning algorithms have been designed to compute such paths, and these methods can be divided into three general categories: 1) graph-based motion planners; 2) sampling-based motion planners; and 3) optimization-based motion planners.

Graph-based motion planning is the most traditional form of motion planning, which is based on fixed graph structures. These approaches, such as Dijkstra's [1] and Floyd-Warshall [2], solve for optimal paths between two vertices in a graph using breadth-first search and the shortest distance between all pairs of vertices, respectively. A fundamental drawback of these methods is their extremely large runtimes when searching through large graphs. Heuristic-based planners, such as A* [3], [4] and D* Lite [5], [6] algorithms, utilize heuristic functions to explore low-cost paths first, offering a balance between optimality and computational efficiency. A considerable drawback for both graph-based and heurisitc-based planners is that these methods discretize the configuration space into grids, which makes solutions suboptimal for continuous configuration spaces.

An alternative to graph-based methods for continuous spaces are sampling-based approaches, such as rapidly-exploring random trees (RRT) [7], [8], [9] and probabilistic roadmaps (PRM) [10], [11], [12], which stochastically explore the configuration space to find feasible paths, and can solve for complicated paths even in high-dimensional spaces. While these methods do not provide any optimality guarantees, several variants including RRT* and PRM* produce optimal paths asymptotically [13], [14], [15], [16]. However, RRT* relies on an inefficient rewiring process of the underlying tree structure, and PRM* relies on an exhaustive nearest-neighbor search to add new edges to the underlying graph structures, to provide this guarantee. In addition, the computational efficiency of sampling-based planners greatly decreases as the dimension of the configuration space increases. The batch informed trees (BIT*) algorithm and its improved

variants (such as AIT* and EIT*) [17], [18], [19], [20], [21], improve upon RRT* and PRM* by incorporating heuristic-based searching to improve the efficiency of sampling-based planners in high-dimensional spaces. However, these algorithms' reliance on heuristic-based searching can make them sensitive to the choice of heuristic, especially for highly nonlinear cost functions.

Finally, optimization-based approaches have proven to be another effective method for planning optimal paths, especially in high-dimensional spaces. These methods have the important benefit of easily incorporating dynamic costs and constraints, such as joint torques, when solving for optimal paths. In recent years, several methods have been developed to efficiently solve for smooth and low-cost paths using covariant gradient descent [22], stochastic optimization [23], and sequential convex optimization [24] to minimize the cost of a quickly computed naïve path. The recently developed CuRobo trajectory optimization algorithm [25] combines parallelized stochastic gradient descent and parallelized traditional gradient descent to quickly produce collision-free minimum-jerk trajectories. While these methods have several important advantages, their generated paths are not guaranteed to be globally optimal.

Learning techniques can also be integrated with the above three categories of motion planners to generate trajectories quickly online. One method to achieve this is to replace individual planner components with learning methods. The most common example of this is using learned sampling distributions to replace sampling strategies of existing sampling-based motion planners [26]. The goal of these methods is to reduce the number of samples required to plan trajectories in complex and high-dimensional spaces while preserving the guarantees of the underlying planner. These samplers can be constructed using feedforward neural networks with dropout [27], convolutional neural networks [28], or transformers [29], where all of these approaches can greatly improve sample efficiency. However, when using these samplers, the underlying planner can fail to generate valid trajectories even when solutions exist, and they also inherit the computational complexities of the underlying planner. An alternative to replacing individual planner components is to generate entire trajectories directly using neural networks. Examples include using neural networks to iteratively generate path waypoints [30], [31], using neural networks to represent reactive policies [32], [33], and using diffusion models to sample feasible trajectories that satisfy task constraints [34], [35]. These methods can adapt to unseen environments, and they produce trajectories extremely efficiently. Because these methods generate trajectories using complicated neural networks, there are currently no optimality or completeness guarantees. Furthermore, all of the above methods must be trained using trajectories collected by existing motion planners, thus they cannot be considered standalone motion planners.

All of the above motion planning algorithms demonstrate an unideal compromise between computational efficiency and optimality [36]. To take advantage of the best features of each category, a new motion planning framework called motion planning in graphs of convex sets (GCS) was developed in [37]. This method combines concepts from graph-based, sampling-based, and optimization-based motion planning to efficiently plan collision-free trajectories with minimal time, length, and energy. To guarantee collision avoidance, obstacles in the task space are transferred to the joint space using nonlinear optimization to grow collision-free polytopes in the joint space [38]. Based on these collision-free regions, a graph structure is created to act as a collision-free roadmap for motion planning, where the vertices correspond to overlapping regions and the edges connect vertices belonging to the same collision-free polytopes [39]. After obtaining this roadmap, the motion planning problem can be formulated as a convex optimization problem, where the objective function is the sum of the trajectory edge costs weighted by the likelihood that they belong to the optimal path. These trajectory edges are constrained to be within the collision-free regions. Several variations of the GCS algorithm have been developed to handle motion planning problems in non-Euclidean spaces [40], solve motion planning problems with a small class of nonconvex objectives and constraints [41], and efficiently solve for optimal trajectories with several target locations [42].

A fundamental drawback of the GCS planner is its inability to handle arbitrary nonconvex cost functions. In addition, when planning using dense graph structures with many cycles and a large number of edges, the computational efficiency of the GCS planner will greatly decrease. Furthermore, the quality of the computed paths can severely decline because it is difficult to accurately determine the likelihood that each edge belongs to the optimal trajectory with minimal cost in these dense graph structures [39]. To overcome these limitations, this work proposes a new motion planner based on the GCS planner, called the graph of convex sets with general costs (GCSGC) planner. This method can optimize nonconvex cost functions for more general applications and establishes a method to efficiently simplify the underlying graph structure of the GCS planner. The main contributions of this work are as follows.

1) Developing a method to optimize nonconvex cost functions by dividing the configuration space into locally linear-cost regions using neural networks and build a graph of convex sets based on the intersection of collision-free linear-cost regions.
2) Transforming the nonconvex edge cost functions to be convex using the McCormick relaxation.
3) Proposing an online strategy to preprocess the underlying graph structure of the planner by removing cycles and high-cost paths to improve the efficiency and optimality of the motion planner.

It should be noted that while the proposed method uses neural networks when constructing its graph of convex sets, it is unlike learning-based motion planners because it does not rely on existing motion planners to produce training data.

The rest of this article is organized as follows. Section II introduces the background on computing the collision-free regions and planning trajectories using the GCS planner. The new motion planner proposed in this work is introduced in Section III. A graph preprocessing technique to simplify the underlying

graph of convex sets is developed in Section IV. Results to validate the effectiveness of the proposed method and comparisons with state-of-the-art planners are given in Section V. Finally, Section VI concludes this article.

II. Background on GCS Planner

A. Computing Collision-Free Regions

Obstacles typically exist in the workspace with simple shapes, while the shapes of these obstacles in the configuration space are extremely irregular and difficult to compute. This complicated collision-free configuration space can be approximated by a set of simple collision-free polytopes. To compute each of these collision-free polytopes, the IRIS-NP algorithm is developed in [38], which repeats a two step process to grow a polytope until all of the joint configurations within the polytope are collision-free and the volume of the polytope is maximized.

The first step of this algorithm adds hyperplanes to a polytope representing the robot's joint limits to remove any configurations where the robot's links are colliding with each other or the environment. To achieve this goal, a counterexample search is performed to find a configuration that is not collision-free and is closest to the center of the largest inscribed ellipse by solving the following nonlinear program:

$$\min_{\mathbf{q}} \ (\mathbf{q}-\mathbf{d})^\top \mathbf{C}^\top \mathbf{C}\,(\mathbf{q}-\mathbf{d}) \tag{1a}$$

$$\text{s.t. } \mathbf{A}\mathbf{q} \le \mathbf{b} \tag{1b}$$

$$\text{FK}(\mathbf{q}, \mathcal{B}^{(i)}) \cap \text{FK}(\mathbf{q}, \mathcal{B}^{(j)}) \neq \emptyset \tag{1c}$$

where $\mathbf{q}$ is the joint configuration, $\mathbf{C}$ and $\mathbf{d}$ are the shape and center of the largest inscribed ellipse, respectively, $\mathbf{A}$ and $\mathbf{b}$ represent the polytope, and $\text{FK}(\mathbf{q}, \mathcal{B}^{(i)})$ represents the points inside of body $\mathcal{B}^{(i)}$ when the robot is at joint configuration $\mathbf{q}$. Conceptually, constraints (1b) and (1c) ensure that $\mathbf{q}$ is not a collision-free configuration despite being inside of the current polytope. If a valid counterexample exists, the following hyperplane

$$\mathbf{a}_j = \frac{\mathbf{C}^\top\mathbf{C}(\mathbf{q}-\mathbf{d})}{\|\mathbf{C}^\top\mathbf{C}(\mathbf{q}-\mathbf{d})\|_2}, \ b_j = \mathbf{a}_j^\top \mathbf{q} - \delta \tag{2}$$

where δ is a small positive constant, is added to the current polytope. This hyperplane separates the polytope from the configuration $\mathbf{q}$. This procedure is repeated until several consecutive unsuccessful counterexample searches are performed.

After these hyperplanes have been added to the polytope, the largest inscribed ellipse of the polytope computed using the above counter-example search is determined by solving the following convex program:

$$\max_{\widetilde{\mathbf{C}},\mathbf{d}} \ \log\det \widetilde{\mathbf{C}} \tag{3a}$$

$$\text{s.t. } \widetilde{\mathbf{C}} \in \mathbb{S}^n_{++} \tag{3b}$$

$$\left\|\widetilde{\mathbf{C}}\mathbf{a}_i\right\|_2 + \mathbf{a}_i^\top \mathbf{d} \le b_i, \ 1 \le i \le l \tag{3c}$$

where $\widetilde{\mathbf{C}} = \mathbf{C}^{-1}$, the set $\mathbb{S}^n_{++}$ is the set of positive definite and symmetric $n \times n$ matrices, and l is the number of rows in $\mathbf{A}$ and $\mathbf{b}$. Both of these steps are repeated until the volume of the largest inscribed ellipse does not change.

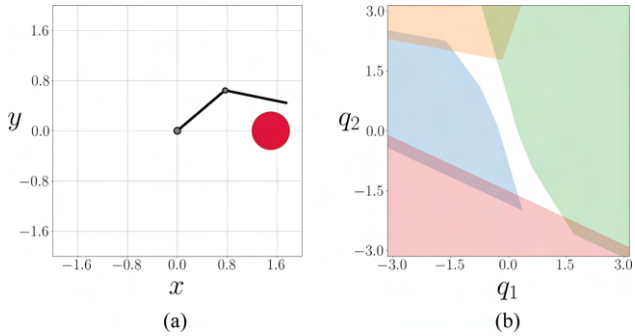


Fig. 1. Planar 2R robot with one spherical obstacle in its workspace is shown in (a). The corresponding collision-free space of this robot is shown in (b) represented by four colored polytopes.

An example of collision-free regions computed by the above algorithm is shown in Fig. 1. These regions are used to approximate the collision-free configuration space of the planar 2R robot with one spherical obstacle in its workspace shown in Fig. 1(a). The corresponding collision-free regions are represented as the four colored polytopes shown in Fig. 1(b).

B. GCS Motion Planner

Based on the above computed collision-free regions, the GCS motion planner was developed in [37] to solve for optimal collision-free paths. First, the GCS planner develops a graph structure to represent the connections between different collision-free regions in the configuration space. Next, the GCS planner solves for a set of potentially optimal edges between collision-free regions by minimizing the costs of the paths in the graph structure, which are defined as the sum of the costs of the edges belonging to each path. This can be accomplished by turning off high-cost edges and minimizing the cost of the remaining edges. This procedure is represented as the following convex program:

$$\min_{\mathbf{z}_e,\mathbf{z}'_e,\phi_e} \ \sum_{e\in\mathcal{E}} \tilde{\ell}_e(\mathbf{z}_e, \mathbf{z}'_e, \phi_e) \tag{4a}$$

$$\text{s.t. } \sum_{e\in\mathcal{E}_s^{\text{out}}} \phi_e = \sum_{e\in\mathcal{E}_g^{\text{in}}} \phi_e = 1 \tag{4b}$$

$$\sum_{e\in\mathcal{E}_v^{\text{out}}} \phi_e \le 1 \qquad \forall\, v \in \boldsymbol{\mathcal{V}}\setminus\{s,g\} \tag{4c}$$

$$\sum_{e\in\mathcal{E}_v^{\text{in}}} \begin{bmatrix}\mathbf{z}'_e \\ \phi_e\end{bmatrix} = \sum_{e\in\mathcal{E}_v^{\text{out}}} \begin{bmatrix}\mathbf{z}_e \\ \phi_e\end{bmatrix} \quad \forall\, v \in \boldsymbol{\mathcal{V}}\setminus\{s,g\} \tag{4d}$$

$$\mathbf{z}_e \in \widetilde{\mathcal{Q}}_u(\phi_e), \ \mathbf{z}'_e \in \widetilde{\mathcal{Q}}_v(\phi_e) \quad \forall\, e=(u,v) \in \mathcal{E} \tag{4e}$$

$$0 \le \phi_e \le 1 \qquad \forall\, e \in \mathcal{E} \tag{4f}$$

where $\mathcal{E}$ and $\boldsymbol{\mathcal{V}}$ are the edge and vertex sets of the underlying graph structure, the variables ϕ_e is the flow associated with the edge e which represents the likelihood that this edge belongs to the optimal trajectory. The variables $\mathbf{z}_e = \phi_e \mathbf{q}_e$ and $\mathbf{z}'_e = \phi_e \mathbf{q}'_e$,

where $\mathbf{q}_e$ and $\mathbf{q}'_e$ are the start and end configuration of the edge e, respectively. The function $\tilde{\ell}_e(\mathbf{z}_e, \mathbf{z}'_e, \phi_e) = \ell_e(\mathbf{q}_e, \mathbf{q}'_e)\phi_e$ represents the perspective of the cost of edge e with respect to ϕ_e, the vertices s and g represent the start and goal configurations, respectively, and $\widetilde{\mathcal{Q}}_u(\phi_e)$ is the perspective cone of $\mathcal{Q}_u$ (intersection of collision-free regions) with respect to ϕ_e. After solving (4), the resulting values of ϕ_e can be interpreted as the likelihood that the edge e belongs to the globally optimal path. Therefore, a set of paths $\mathcal{P}$ between the start configuration and the goal configuration can be constructed by randomly sampling edges based on the probability distribution induced by the solution to (4). This procedure can be repeated until $\mathcal{P}$ is sufficiently large. Then, the optimal vertices associated with the above computed paths and constrained within the intersections of the collision-free regions are determined by solving the following lightweight convex program:

$$\min_{\mathbf{q}_e, \mathbf{q}'_e} \sum_{e \in \mathcal{E}_p} \ell_e(\mathbf{q}_e, \mathbf{q}'_e) \tag{5a}$$

$$\text{s.t. } p \in \mathcal{P} \tag{5b}$$

$$\mathbf{q}_e \in \mathcal{Q}_u,\ \mathbf{q}'_e \in \mathcal{Q}_v \qquad \forall\, e = (u, v) \in \mathcal{E}_p \tag{5c}$$

$$\mathbf{q}'_d = \mathbf{q}_e, \qquad d = (u, v),\ e = (v, w) \in \mathcal{E}_p \tag{5d}$$

where the set $\mathcal{E}_p$ is the set of edges belonging to the path p. Finally, the optimal trajectory computed by the GCS motion planner is the lowest cost trajectory after solving the above optimization problem for all of the potential paths.

III. GCSGC Motion Planner

A. Overview

This recently developed GCS planner provides a very efficient framework for solving optimal collision-free trajectories, but it can only optimize convex cost functions. A new motion planner, called the GCSGC planner, is proposed in this section to plan optimal collision-free trajectories with nonconvex cost functions. To optimize nonconvex cost functions, they are first approximated by piecewise linear functions, and the configuration space is divided into linear-cost regions in Section III-B. The underlying graph structure of the planner is developed based on the intersection of these linear-cost regions with the collision-free regions in Section III-C. While the cost of each configuration is then linear within each linear-cost region, the cost of each edge in the graph structure is nonconvex. Thus, a convex relaxation based on McCormick envelopes is developed in Section III-D. The optimal collision-free trajectories are finally computed by solving two convex optimization problems based on the developed graph of convex sets in Section III-E.

B. Linearizing Nonconvex Cost Functions and Dividing Configuration Space Into Linear-Cost Regions

A key feature of neural networks that use the ReLU activation function is their ability to approximate complex nonlinear functions as piecewise linear functions, as proved in [43] and [44]. Therefore, these neural networks can be used to linearize nonconvex cost functions. Given a neural network with multiple hidden layers using the ReLU activation function, the neural network output $z \in \mathbb{R}$, i.e., the nonconvex cost function relative to each individual configuration $\mathbf{q} \in \mathbb{R}^n$, can be computed by the following steps:

$$\mathbf{y}^{(1)} = \sigma\left(\mathbf{W}^{(1)}\mathbf{q} + \mathbf{b}^{(1)}\right) \tag{6a}$$

$$\mathbf{y}^{(i)} = \sigma\left(\mathbf{W}^{(i)}\mathbf{y}^{(i-1)} + \mathbf{b}^{(i)}\right),\ \text{for } 2 \le i < L \tag{6b}$$

$$z = (\mathbf{w}^{(L)})^\top \mathbf{y}^{(L-1)} + b^{(L)} \tag{6c}$$

where L represents the number of hidden layers, $\mathbf{W}^{(i)} \in \mathbb{R}^{N_i \times N_{i-1}}$ represents the weights of the ith hidden layer and N_i is the number of neurons in the ith hidden layer, $\mathbf{b}^{(i)} \in \mathbb{R}^{N_i}$ represents the bias of the ith hidden layer, $\sigma(x) = \max\{0, x\}$ is the ReLU activation function applied elementwise, $\mathbf{w}^{(L)} \in \mathbb{R}^{N_{L-1}}$ represents the output weights, and $b^{(L)} \in \mathbb{R}$ represents the output bias.

It is apparent from (6b) that if $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(i-1)} + b_j^{(i)} \le 0$, where $\mathbf{w}_j^{(i)}$ is the jth row of $\mathbf{W}^{(i)}$ and $b_j^{(i)}$ is the jth element of $\mathbf{b}^{(i)}$, then $\mathbf{y}_j^{(i)} = 0$. As a result, $\mathbf{w}_j^{(i)}$ and $b_j^{(i)}$ do not effect the output of the ith hidden layer. Thus, the output of the first hidden layer can be rewritten as a linear function of $\mathbf{q}$, expressed as follows:

$$\mathbf{y}^{(1)} = \widetilde{\mathbf{W}}^{(1)}\mathbf{q} + \widetilde{\mathbf{b}}^{(1)} \tag{7}$$

where $\widetilde{\mathbf{W}}^{(1)}$ and $\widetilde{\mathbf{b}}^{(1)}$ represent $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ with the rows satisfying $(\mathbf{w}_j^{(1)})^\top \mathbf{q} + b_j^{(1)} \le 0$ set to zero. Likewise, the output of the ith hidden layer can be rewritten as a linear function of $\mathbf{y}^{(i-1)}$, expressed as follows:

$$\mathbf{y}^{(i)} = \widetilde{\mathbf{W}}^{(i)}\mathbf{y}^{(i-1)} + \widetilde{\mathbf{b}}^{(i)} \tag{8}$$

where $\widetilde{\mathbf{W}}^{(i)}$ and $\widetilde{\mathbf{b}}^{(i)}$ represent $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$, respectively, with the rows satisfying $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(i-1)} + b_j^{(i)} \le 0$ set to zero. Thus, the neural network output z can finally be written as a linear function of $\mathbf{q}$, expressed as follows:

$$z = (\mathbf{w}^{(L)})^\top \left[\left(\prod_{i=1}^{L-1} \widetilde{\mathbf{W}}^{(i)} \right) \mathbf{q} + \left(\prod_{i=2}^{L-1} \widetilde{\mathbf{W}}^{(i)} \right) \widetilde{\mathbf{b}}^{(1)} + \ldots + \widetilde{\mathbf{W}}^{(L-1)} \widetilde{\mathbf{b}}^{(L-2)} + \widetilde{\mathbf{b}}^{(L-1)} \right] + b^{(L)}. \tag{9}$$

If a set of configurations $\mathcal{H}$ have the same $\widetilde{\mathbf{W}}^{(i)}$ and $\widetilde{\mathbf{b}}^{(i)}$ for all of the hidden layers $1 \le i < L$, then these configurations form a region where the nonconvex cost function can be simplified as a linear function of the input configuration, as shown in (9). These regions are referred to as linear-cost regions. A linear-cost region $\mathcal{H}$ can be determined by analyzing the effect of the weights and biases on the neural network output for the configurations within $\mathcal{H}$. For each hidden layer i, if $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \le -b_j^{(i)}$, where $\mathbf{y}^{(\mathbf{i-1})}$ is a linear function of $\mathbf{q} \in \mathcal{H}$, then the weight vector and bias will not affect the linear cost function, but the linear cost function is still only valid when this inequality is satisfied. Alternatively, if this inequality is not satisfied, then this weight vector and bias will affect the network output.

As such, there are two types of inequalities which form $\mathcal{H}$ for each hidden layer i, the first is any row j of $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ such that $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \leq -b_j^{(i)}$ and the second is any row k of $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ such that $(\mathbf{w}_k^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \geq -b_k^{(i)}$. The inequalities of all of the hidden layers form the linear-cost region $\mathcal{H}$ as follows:

$$\mathcal{H} = \left\{ \mathbf{q} : \begin{bmatrix} \mathbf{S}^{(1)}\mathbf{W}^{(1)}\mathbf{q} \\ \mathbf{S}^{(2)}\mathbf{W}^{(2)}\mathbf{y}^{(1)} \\ \vdots \\ \mathbf{S}^{(L-1)}\mathbf{W}^{(L-1)}\mathbf{y}^{(L-2)} \end{bmatrix} \leq \begin{bmatrix} \mathbf{S}^{(1)}\mathbf{b}^{(1)} \\ \mathbf{S}^{(2)}\mathbf{b}^{(1)} \\ \vdots \\ \mathbf{S}^{(L-1)}\mathbf{b}^{(L-1)} \end{bmatrix} \right\} \tag{10}$$

where $\mathbf{S}^{(i)}$ is a diagonal matrix such that $s_{jj}^{(i)} = -1$ if $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \leq -b_j^{(i)}$ and $s_{jj}^{(i)} = 1$ otherwise. Conceptually, the matrix $\mathbf{S}^{(i)}$ acts to convert the inequality $(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \geq -b_j^{(i)}$ to the inequality $-(\mathbf{w}_j^{(i)})^\top \mathbf{y}^{(\mathbf{i-1})} \leq b_j^{(i)}$, which allows $\mathcal{H}$ to be defined in the standard form $\mathbf{Ax} \leq \mathbf{b}$. Furthermore, each linear-cost region $\mathcal{H}$ can be uniquely determined by the $\mathbf{S}^{(i)}$ matrices, for $1 \leq i < L$, alone.

Based on the above definition of linear-cost regions, the configuration space can be divided into many polytopes by the neural network hyperplanes, where the nonconvex cost function becomes a linear function of the configurations within each polytope. To solve for all of these polytopes, an arbitrary configuration is first used to solve for an initial linear-cost region that this configuration belongs to. Once this polytope has been computed, all of its potential neighbors can be solved for by flipping each diagonal element of all of its associated $\mathbf{S}^{(i)}$ matrices. For each potential neighbor, if no configurations satisfy (10), then this neighbor is not valid. Otherwise, this neighbor represents a valid linear-cost region, and its unvisited neighbors will be generated and checked for validity. This process concludes when there are no more unvisited neighbors left to check.

A simple example in a 2-D configuration space with the nonconvex eggbox cost function $f(q_1, q_2) = \sin(q_1)\sin(q_2) + 1.25$ shown in Fig. 2 is used to illustrate the above concepts. This cost function is used to train three different neural networks of increasingly large size. The first network uses a single hidden layer with seven neurons, and its output is shown in Fig. 3(a), and its estimation error is shown in Fig. 3(b). The second network uses three hidden layers with seven neurons each, and its output is shown in Fig. 3(c), and its estimation error is shown in Fig. 3(d). The largest neural network uses four hidden layers with ten neurons each, and its output is shown in Fig. 3(e), and its estimation error is shown in Fig. 3(f). The range of the cost function is $[-1, 1]$, and the mean absolute error of each neural network is 0.048, 0.008, and 0.005, respectively. The maximum absolute error of each network is 0.21, 0.11, and 0.036, respectively. The configuration space divided by each neural network into linear-cost regions is shown in Fig. 4, where the small network produces 12 regions, as shown in Fig. 4(a), the medium network produces 336 regions, as shown in Fig. 4(b), and the large network produces 655 regions, as shown in Fig. 4(c). It is clear that the error decreases with increases in the complexity of the neural network, however, the number of linear-cost regions can quickly become quite large. An appropriate neural network structure can be chosen based on the tradeoff between the number of linear-cost regions and the accuracy of the approximated cost function.

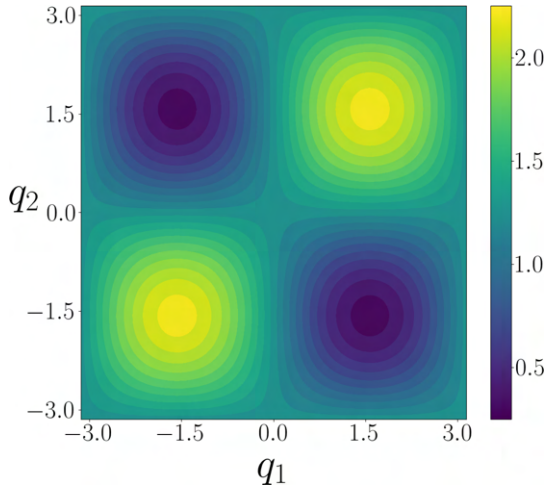


Fig. 2. Eggbox function, $f(q_1, q_2) = \sin(q_1)\sin(q_2) + 1.25$, is given, where the output of the function at a configuration (q_1, q_2) is demonstrated by its associated color.

C. Building Graph of Convex Sets and Computing Optimal Trajectories

These linear-cost regions are intersected with collision-free regions to build a graph of convex sets used to solve for collision-free paths that minimize a nonconvex cost function. The first step in building this graph structure is intersecting the linear-cost regions, as shown by the colored regions $\mathcal{H}_i$ in Fig. 5(a), with a set of previously computed collision-free regions, as shown by the pink regions $\mathcal{Q}_i$ in Fig. 5(b). These computed intersections represent collision-free linear-cost regions, as shown by the six colored regions in Fig. 5(c). Next, these collision-free linear-cost regions are intersected to form the vertices of the underlying graph structure of the motion planner, as shown by the white areas $\mathcal{V}_i$ in Fig. 5(d). Because the linear-cost regions intersect only at shared hyperplanes, these regions can be expanded a small amount to ensure their intersections have some volume such that all of the vertices in the underlying graph structure can be represented using only inequality constraints. The dashed lines in Fig. 5(d) represent the connectivity between these convex sets. Together, these vertices and their connections form the underlying graph of convex sets, denoted as $\mathcal{G} = (\boldsymbol{\mathcal{V}}, \mathcal{E})$, where $\boldsymbol{\mathcal{V}}$ is the set of vertices and $\mathcal{E}$ is the set of edges representing the connectivity between these vertices. A region path p represents a sequence of intersections of collision-free linear-cost regions that a trajectory will travel through. For example, the region path of the blue trajectory, which starts at the green configuration and ends at the red configuration, is $(\mathcal{V}_1, \mathcal{V}_2, \mathcal{V}_3, \mathcal{V}_4, \mathcal{V}_5)$ in Fig. 5(e).

After the graph of convex sets is built, the optimal trajectory between a given start configuration and goal configuration can be computed in a similar fashion to the GCS motion planner. First, a set of region paths that most likely contain the globally optimal

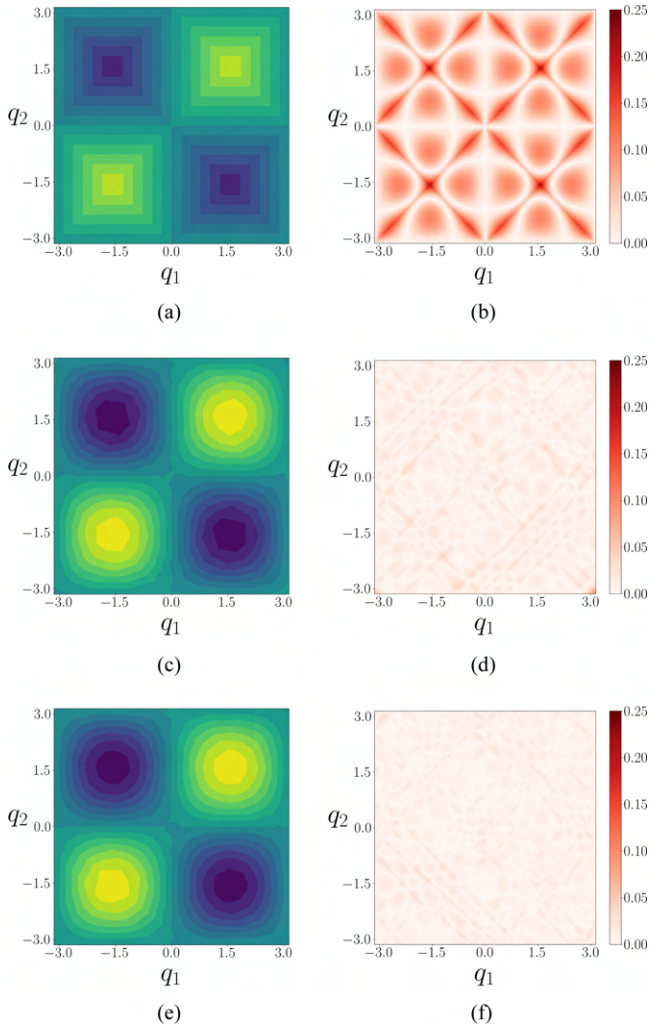


Fig. 3. Performance of three different neural networks used to approximate the eggbox function is shown. The output of the small, medium, and large neural networks are shown in (a), (c), and (e), respectively. The approximation error of the small, medium, and large neural networks are shown in (b), (d), and (f), respectively.

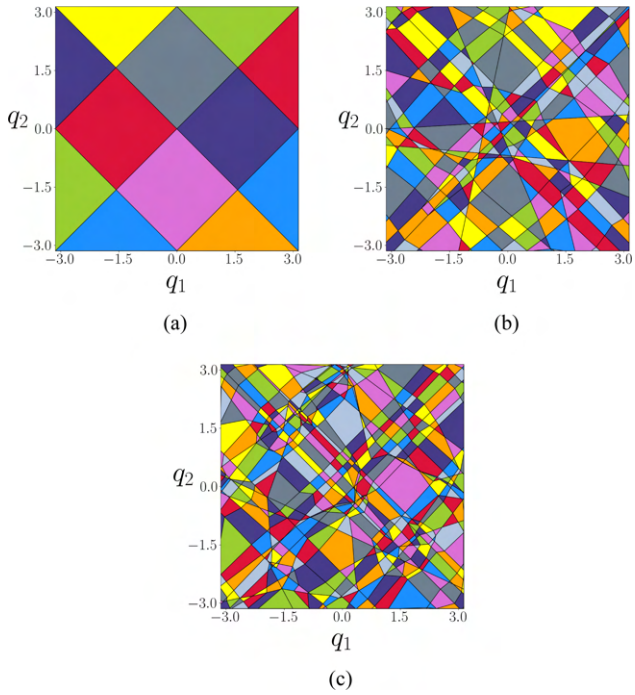


Fig. 4. (a) Configuration space is divided into 12 linear-cost regions by the small neural network. (b) 336 linear-cost regions by the medium neural network. (c) 655 linear-cost regions by the large neural network.

trajectory must be determined. After this set has been computed, the lowest cost trajectory within each region path is computed. Finally, the costs of the computed trajectories are compared, and the lowest cost trajectory is selected as the globally optimal trajectory.

To compute the set of region paths that most likely contain the globally optimal trajectory, the minimum network flow problem and shortest path problem are combined into a single optimization problem, which minimizes the costs of the trajectories within different region paths [39]. To accomplish this, flow variables ϕ_e are used to represent the likelihood that the globally optimal trajectory moves through the edge $e = (u, v)$ which connects regions $\mathcal{V}_u$ and $\mathcal{V}_v$. In addition, to convexify this optimization problem after introducing these flow variables, perspective functions are used to combine joint configurations with the edge flow variables into a set of new decision variables, denoted $\mathbf{z}_e = \phi_e \mathbf{q}_e$ and $\mathbf{z}'_e = \phi_e \mathbf{q}'_e$ for each edge e. This optimization problem is almost identical to (4), where only the intersections of collision-free regions, $\mathcal{Q}_u$ and $\mathcal{Q}_v$ from (4e), are replaced by the intersections of collision-free linear-cost regions, $\mathcal{V}_u$ and $\mathcal{V}_v$, as follows:

$$\mathbf{z}_e \in \widetilde{\mathcal{V}}_u(\phi_e),\ \mathbf{z}'_e \in \widetilde{\mathcal{V}}_v(\phi_e)\ \forall\ e = (u, v) \in \mathcal{E} \tag{11}$$

where $\widetilde{\mathcal{V}}_u(\phi_e)$ is the perspective cone of $\mathcal{V}_u$ with respect to ϕ_e. After solving this optimization problem, the region paths that potentially contain the globally optimal trajectory are computed by randomly building region paths from the start configuration to the goal configuration based on the flow variables. The probability that a region $\mathcal{V}_v$ is added to a region path currently at $\mathcal{V}_u$ is equal to the flow between these two regions normalized by the flow between $\mathcal{V}_u$ and all of its neighbors [37]. This process is repeated until sufficiently many region paths have been solved for.

After a set of region paths, denoted as $\mathcal{P}$, are computed, the optimal trajectory within each region path $p \in \mathcal{P}$ can be computed by determining the configuration within each vertex $\mathcal{V}_i \in p$ to travel through which minimizes the cost of the trajectory. This optimization problem is almost identical to (5), where only the intersections of collision-free regions, $\mathcal{Q}_u$ and $\mathcal{Q}_v$ from (5c), are replaced by the intersections of collision-free linear-cost regions, $\mathcal{V}_u$ and $\mathcal{V}_v$, as follows:

$$\mathbf{q}_e \in \mathcal{V}_u,\ \mathbf{q}'_e \in \mathcal{V}_v\ \forall\ e = (u, v) \in \mathcal{E}_p \tag{12}$$

where the set $\mathcal{E}_p$ is the set of edges in the region path p. The globally optimal collision-free trajectory can then be solved for by performing the above optimization for all $p \in \mathcal{P}$.

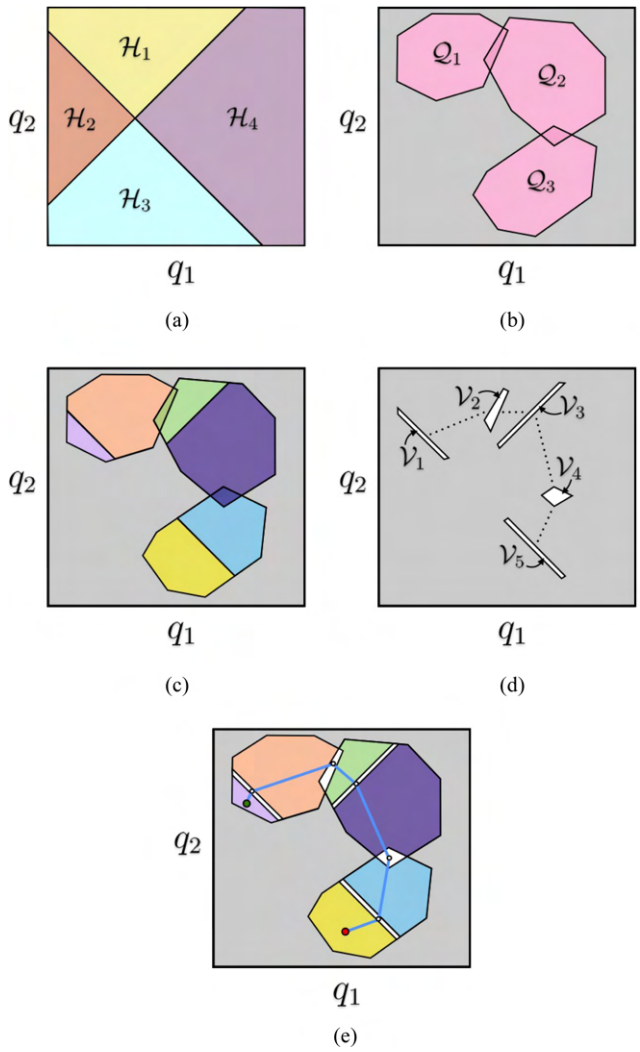


Fig. 5. Process of constructing the graph structure for the proposed motion planner is given. A set of four linear-cost regions in (a) are intersected with three collision-free regions in (b) to create the six collision-free linear-cost regions in (c). The intersections of these six collision-free linear-cost regions form the vertices of the graph structure, as shown by the white regions in (d), and the dashed lines represent the connections between these vertices. These vertices and connections form the graph of convex sets used by the proposed motion planner. An example trajectory from the green starting configuration to the red goal configuration is given in (e).

D. Convexifying Edge Costs Using McCormick Relaxation

For a general cost function c, the cost function of an edge e along a path is defined as follows:

$$\ell_e(\mathbf{q}_e, \mathbf{q}'_e) = \int_0^L c\left(\left(1 - \frac{\tau}{L}\right)\mathbf{q}_e + \frac{\tau}{L}\mathbf{q}'_e\right)\, d\tau \tag{13}$$

where $L = \|\mathbf{q}_e - \mathbf{q}'_e\|_2$ represents the length of the edge, the variables $\mathbf{q}_e$ and $\mathbf{q}'_e$ represent the start and end configurations of the edge. Conceptually, this edge cost function represents the accumulation of the cost function c while traveling along a straight line between $\mathbf{q}_e$ and $\mathbf{q}'_e$. This formulation is the most common definition of trajectory costs for general cost functions in motion planning. For example, it is used to maximize obstacle clearance, maximize global dexterity, and minimize energy consumption [45], [46]. This edge cost function can be complicated based on the properties of the cost function c, and thus there is no general closed-form solution. Because the edges in the underlying graph structure lie completely in the same linear-cost region, as defined above, the edge cost function can be easily computed as follows:

$$\ell_e(\mathbf{q}_e, \mathbf{q}'_e) = \int_0^L \mathbf{p}_e^\top \left(\left(1 - \frac{\tau}{L}\right)\mathbf{q}_e + \frac{\tau}{L}\mathbf{q}'_e\right) + o_e\, d\tau \tag{14}$$

where $\mathbf{p}_e$ and o_e are the parameters of the linear cost function inside of the region containing both $\mathbf{q}_e$ and $\mathbf{q}'_e$. This function has the following closed-form solution:

$$\ell_e(\mathbf{q}_e, \mathbf{q}'_e) = \left(\frac{1}{2}\mathbf{p}_e^\top (\mathbf{q}_e + \mathbf{q}'_e) + o_e\right) \cdot \|\mathbf{q}_e - \mathbf{q}'_e\|_2 . \tag{15}$$

This closed-form solution can be treated as the product of two convex functions, the linear cost of edge e, defined as $f_e(\mathbf{q}_e, \mathbf{q}'_e) = \frac{1}{2}\mathbf{p}_e^\top(\mathbf{q}_e + \mathbf{q}'_e) + o_e$ and the length of edge e, defined as $h_e(\mathbf{q}_e, \mathbf{q}'_e) = \|\mathbf{q}_e - \mathbf{q}'_e\|_2$. This edge cost function is nonconvex as the product of the two convex functions is not necessarily convex. If this edge cost function is used in (4a), the trajectory optimization problem is no longer convex. To convexify this motion planning problem, an auxiliary variable t_e is first introduced to represent edge cost function (15), such that $t_e = \ell_e(\mathbf{q}_e, \mathbf{q}'_e)$. This nonconvex equality constraint can then be replaced by several convex inequality constraints by applying the McCormick relaxation of factorable functions as follows:

$$t_e \le \overline{\mathcal{C}}_e(\mathbf{q}_e, \mathbf{q}'_e)\ \forall\, e \in \mathcal{E} \tag{16a}$$

$$t_e \ge \underline{\mathcal{C}}_e(\mathbf{q}_e, \mathbf{q}'_e)\ \forall\, e \in \mathcal{E} \tag{16b}$$

where $\overline{\mathcal{C}}_e(\mathbf{q}_e, \mathbf{q}'_e)$ and $\underline{\mathcal{C}}_e(\mathbf{q}_e, \mathbf{q}'_e)$ represent the McCormick envelope of the decision variable t_e. As stated in Section III-C, to incorporate the flow variables with this new edge cost formulation and retain the convexity of this optimization problem, the perspective of the edge cost t_e with respect to the flow variable ϕ_e must be used, i.e., $\tilde{t}_e = \phi_e t_e$. Therefore, $\tilde{t}_e$ must be constrained by the perspective of the McCormick envelope with respect to ϕ_e as follows:

$$\tilde{t}_e \le \overline{\mathcal{C}}_e(\mathbf{z}_e, \mathbf{z}'_e, \phi_e)\ \forall\, e \in \mathcal{E} \tag{17a}$$

$$\tilde{t}_e \ge \underline{\mathcal{C}}_e(\mathbf{z}_e, \mathbf{z}'_e, \phi_e)\ \forall\, e \in \mathcal{E}. \tag{17b}$$

To compute these constraints, the convex and concave envelopes of f_e and h_e need to be computed. The intersection of both concave envelopes is the upper limit of $\tilde{t}_e$, and the intersection of both convex envelopes is the lower limit. The convex and concave envelopes of f_e and h_e can be computed by using the following formulas [47]:

$$\tilde{t}_e \le \overline{f}_e(\mathbf{z}_e, \mathbf{z}'_e)h_e^M + \overline{h}_e(\mathbf{z}_e, \mathbf{z}'_e)f_e^m - \phi_e f_e^m h_e^M \tag{18a}$$

$$\tilde{t}_e \le \overline{f}_e(\mathbf{z}_e, \mathbf{z}'_e)h_e^m + \overline{h}_e(\mathbf{z}_e, \mathbf{z}'_e)f_e^M - \phi_e f_e^M h_e^m \tag{18b}$$

$$\tilde{t}_e \ge \underline{f}_e(\mathbf{z}_e, \mathbf{z}'_e)h_e^m + \underline{h}_e(\mathbf{z}_e, \mathbf{z}'_e)f_e^m - \phi_e f_e^m h_e^m \tag{18c}$$

$$\tilde{t}_e \ge \underline{f}_e(\mathbf{z}_e, \mathbf{z}'_e)h_e^M + \underline{h}_e(\mathbf{z}_e, \mathbf{z}'_e)f_e^M - \phi_e f_e^M h_e^M \tag{18d}$$

IEEE Transactions on Robotics (T-RO) paper, presented at ICRA 2026, Vienna, Austria. Cite as T-RO paper.

where the functions $\overline{f}_e$ and $\overline{h}_e$ are the concave overestimators of f_e and h_e, respectively. The functions $\underline{f}_e$ and $\underline{h}_e$ are the convex underestimators of f_e and h_e, respectively. The variables f_e^m and h_e^m are the minimum values of f_e and h_e, respectively. The variables f_e^M and h_e^M are the maximum values of f_e and h_e, respectively. Inequalities (18a) and (18b) represent the concave envelope of the edge cost function, while inequalities (18c) and (18d) represent the convex envelope. Because f_e is a linear function being concave and convex simultaneously, f_e is its own concave overestimator $\overline{f}_e$ and convex underestimator $\underline{f}_e$. In addition, its minimum value f_e^m and maximum value f_e^M can be solved for using convex optimization, ensuring that $\mathbf{q}_e$ and $\mathbf{q}'_e$ are always contained within their respective intersections of collision-free linear-cost regions. Because h_e is convex and not concave, its convex underestimator $\underline{h}_e$ is itself, and its concave overestimator $\overline{h}_e$ is defined as the largest possible distance between the starting and end configurations of the edge e. In addition, its minimum value h_e^m can also be computed using convex optimization. Its maximum value h_e^M can be determined by comparing all of the combinations of the extreme points of the region containing $\mathbf{q}_e$ with the extreme points of the region containing $\mathbf{q}'_e$.

An illustrative example of the application of the McCormick relaxation to convexify the edge cost in (15) is given. This example uses two connected 1-D vertices q_e and q'_e, bounded from -0.8 to 0.5 and 1.5 to 2.5, respectively. The linear cost of configurations along the edge between these two vertices is defined as $c(q) = 0.3q + 0.1$. Using (15), the edge cost function can be obtained as $\ell_e(q_e, q'_e) = [0.15(q_e + q'_e) + 0.1] \cdot \|q_e - q'_e\|_2$. The corresponding optimization problem to minimize the edge cost is given as follows:

$$\min_{q_e, q'_e} \ \ell_e(q_e, q'_e) = [0.15\,(q_e + q'_e) + 0.1] \cdot \|q_e - q'_e\|_2 \quad (19a)$$

$$\text{s.t. } q_e \in [-0.8,\ 0.5] \quad (19b)$$

$$q'_e \in [1.5,\ 2.5]\,. \quad (19c)$$

Fig. 6(a) shows the optimization landscape of this problem, where the gray box represents the valid values of the two vertices q_e and q'_e, the colored surface represents the edge cost for all possible combinations of q_e and q'_e, and the red point represents the global minimum of this problem.

By applying the proposed McCormick relaxation, the above optimization problem can be reformulated as follows:

$$\min_{q_e, q'_e} \ t_e \quad (20a)$$

$$\text{s.t. } q_e \in [-0.8,\ 0.5] \quad (20b)$$

$$q'_e \in [1.5,\ 2.5] \quad (20c)$$

$$t_e \le 0.50\,(q_e + q'_e) + 0.33 + 0.21\,(q'_e - q_e) - 0.68 \quad (20d)$$

$$t_e \le 0.15\,(q_e + q'_e) + 0.1 + 0.55\,(q'_e - q_e) - 0.55 \quad (20e)$$

$$t_e \ge 0.15\,(q_e + q'_e) + 0.1 + 0.21\,(q'_e - q_e) - 0.21 \quad (20f)$$

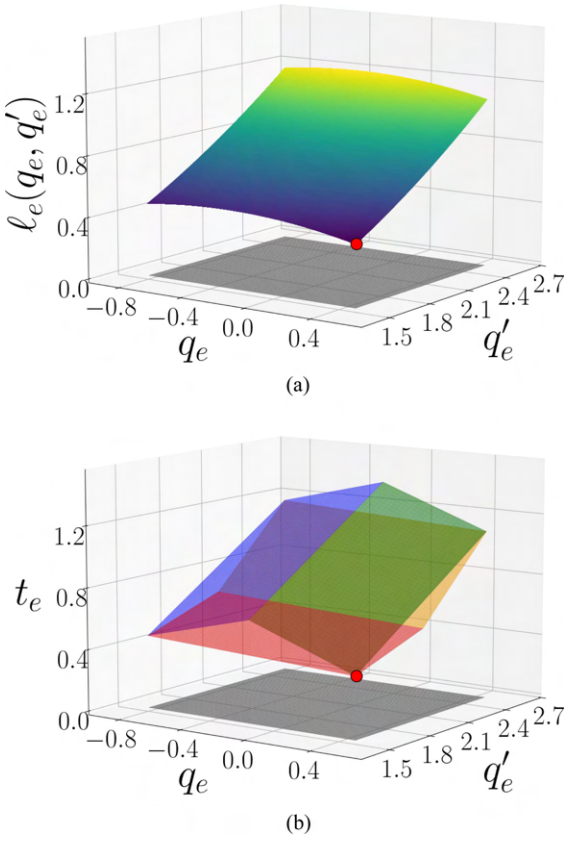


Fig. 6. Illustrative example of the McCormick relaxation applied to convexify the edge cost function of two connected 1-D vertices is shown. The edge costs of all possible combinations of the two vertices are shown in (a), where the red point represents the global minimum of the edge cost function. The McCormick relaxation of this edge cost function is shown as the four colored hyperplanes in (b), where the red point represents the global minimum within the McCormick envelopes.

$$t_e \ge 0.50\,(q_e + q'_e) + 0.33 + 0.55\,(q'_e - q_e) - 1.82. \quad (20g)$$

The concave envelopes of the auxiliary variable t_e from (20d) and (20e) are shown in Fig. 6(b) as the blue and green hyperplanes, respectively. The convex envelopes from (20f) and (20g) are shown as the red and orange hyperplanes, respectively. Finally, the global minimum is shown as the red point, which coincides with the global minimum of the original nonconvex optimization problem. This illustrative example demonstrates the effectiveness of the proposed convex relaxation of the edge cost function.

For motion planning problems that require a robot's motion to satisfy certain differential constraints, such as velocity and acceleration limits or continuous differentiability, Bezier curves are an effective method to model such trajectories. Thus, an extension of the above McCormick relaxation to edges represented by Bezier curves is briefly discussed. A Bezier curve $\mathbf{r}_e$ is defined as follows:

$$\mathbf{r}_e(t) = \sum_{i=0}^{d} \beta_{i,d}(t) \cdot \mathbf{q}_e^{(i)} \quad (21)$$

where $0 \leq t \leq 1$ represents the normalized time parameter along the curve, $\mathbf{q}_e^{(i)}$ represents the ith control point of the Bezier curve, and $\beta_{i,d}(t)$ represents the ith Bernstein basis polynomial of degree d evaluated at t. The first derivative of $\mathbf{r}_e(t)$ with respect to t is also a Bezier curve, defined as follows:

$$\dot{\mathbf{r}}_e(t) = \sum_{i=0}^{d-1} d \cdot \beta_{j,d-1}(t) \cdot \Delta \mathbf{q}_e^{(j)} \tag{22}$$

where $\Delta \mathbf{q}_e^{(j)} = \mathbf{q}_e^{(j+1)} - \mathbf{q}_e^{(j)}$. Given an edge e that travels completely within a linear cost region with a configuration cost function $c(\mathbf{q}) = \mathbf{p}_e^\top \mathbf{q} + o_e$, the edge cost of the Bezier curve can be computed as follows:

$$\ell_e(\mathbf{r}_e(\cdot)) = \int_0^1 \left(\mathbf{p}_e^\top \mathbf{r}_e(t) + o_e\right) \cdot \|\dot{\mathbf{r}}_e(t)\|_2 \; dt. \tag{23}$$

The upper bound of this edge cost can be computed as follows [37]:

$$\ell_e(\mathbf{r}_e(\cdot)) \leq \sum_{i=0}^{d} \sum_{j=0}^{d-1} \left(\mathbf{p}_e^\top \mathbf{q}_e^{(i)} + o_e\right) \cdot \left\|\Delta \mathbf{q}_e^{(j)}\right\|_2 \cdot \int_0^1 d \cdot \beta_{i,d}(t) \cdot \beta_{j,d-1}(t)\, dt. \tag{24}$$

This upper bound can be expressed in the form $\ell_e(\mathbf{r}_e(\cdot)) \leq \mathbf{f}^\top \mathbf{B} \mathbf{h}$, where $\mathbf{f} \in \mathbb{R}^{d+1}$ and $f_i = \mathbf{p}_e^\top \mathbf{q}_e^{(i)} + o_e$; $\mathbf{B} \in \mathbb{R}^{d+1 \times d}$ and $b_{i,j} = \int_0^1 d \cdot \beta_{i,d}(t) \cdot \beta_{j,d-1}(t)\, dt$, which has a closed-form solution [48]; and $\mathbf{h} \in \mathbb{R}^d$ and $h_j = \|\Delta \mathbf{q}_e^{(j)}\|_2$. Let $\mathbf{g}^\top := \mathbf{f}^\top \mathbf{B}$, then the edge cost is upper bounded by

$$\ell_e(\mathbf{r}_e(\cdot)) \leq \mathbf{g}^\top \mathbf{h} = \sum_{i=0}^{d-1} g_i \cdot h_i. \tag{25}$$

Then, the auxilliary variable T_e can be defined as

$$T_e = \sum_{i=0}^{d-1} t_e^{(i)} = \sum_{i=0}^{d-1} g_i \cdot h_i. \tag{26}$$

Finally, the same McCormick relaxation from (16)–(18) can be applied to each of the d nonconvex equality constraints $t_e^{(i)} = g_i \cdot h_i$, where $t_e^{(i)}$ is constrained within the concave and convex envelopes of $g_i \cdot h_i$ and is used to estimate $g_i \cdot h_i$. This McCormick relaxation can be used to minimize the edge cost of an edge represented by a Bezier curve.

E. Formulating Convex Problems to Solve for Optimal Trajectories

By applying the methods above to linearize the configuration cost function, build the graph of convex sets, and convexify the edge cost functions, the motion planning problem can be finally formulated as two convex optimization problems. First, by applying the McCormick relaxation, the optimization problem in (4) that determines the optimal region paths is reformulated as the following convex program:

$$\min_{\mathbf{z}_e, \mathbf{z}'_e, \phi_e, \tilde{t}_e} \quad \sum_{e \in \mathcal{E}} \tilde{t}_e \tag{27a}$$

$$\text{s.t.} \sum_{e \in \mathcal{E}_s^{\text{out}}} \phi_e = \sum_{e \in \mathcal{E}_g^{\text{in}}} \phi_e = 1 \tag{27b}$$

$$\sum_{e \in \mathcal{E}_v^{\text{out}}} \phi_e \leq 1 \qquad \forall\, v \in \boldsymbol{\mathcal{V}} \setminus \{s, g\} \tag{27c}$$

$$\sum_{e \in \mathcal{E}_v^{\text{in}}} \begin{bmatrix} \mathbf{z}'_e \\ \phi_e \end{bmatrix} = \sum_{e \in \mathcal{E}_v^{\text{out}}} \begin{bmatrix} \mathbf{z}_e \\ \phi_e \end{bmatrix} \qquad \forall\, v \in \boldsymbol{\mathcal{V}} \setminus \{s, g\} \tag{27d}$$

$$\mathbf{z}_e \in \widetilde{\mathcal{V}}_u(\phi_e),\ \mathbf{z}'_e \in \widetilde{\mathcal{V}}_v(\phi_e) \qquad \forall\, e = (u, v) \in \mathcal{E} \tag{27e}$$

$$0 \leq \phi_e \leq 1 \qquad \forall\, e \in \mathcal{E} \tag{27f}$$

$$\tilde{t}_e \leq \overline{f}_e(\mathbf{z}_e, \mathbf{z}'_e) h_e^M + \overline{h}_e(\mathbf{z}_e, \mathbf{z}'_e) f_e^m - \phi_e f_e^m h_e^M \qquad \forall\, e \in \mathcal{E} \tag{27g}$$

$$\tilde{t}_e \leq \overline{f}_e(\mathbf{z}_e, \mathbf{z}'_e) h_e^m + \overline{h}_e(\mathbf{z}_e, \mathbf{z}'_e) f_e^M - \phi_e f_e^M h_e^m \qquad \forall\, e \in \mathcal{E} \tag{27h}$$

$$\tilde{t}_e \geq \underline{f}_e(\mathbf{z}_e, \mathbf{z}'_e) h_e^m + \underline{h}_e(\mathbf{z}_e, \mathbf{z}'_e) f_e^m - \phi_e f_e^m h_e^m \qquad \forall\, e \in \mathcal{E} \tag{27i}$$

$$\tilde{t}_e \geq \underline{f}_e(\mathbf{z}_e, \mathbf{z}'_e) h_e^M + \underline{h}_e(\mathbf{z}_e, \mathbf{z}'_e) f_e^M - \phi_e f_e^M h_e^M \qquad \forall\, e \in \mathcal{E}. \tag{27j}$$

Then, the optimization problem in (5) that optimizes the trajectory within a given region path p can also be reformulated as follows:

$$\min_{\mathbf{q}_e, \mathbf{q}'_e, t_e} \quad \sum_{e \in \mathcal{E}_p} t_e \tag{28a}$$

$$\text{s.t.}\ p \in \mathcal{P} \tag{28b}$$

$$\mathbf{q}_e \in \mathcal{V}_u,\ \mathbf{q}'_e \in \mathcal{V}_v \qquad \forall\, e = (u, v) \in \mathcal{E}_p \tag{28c}$$

$$\mathbf{q}'_d = \mathbf{q}_e, \qquad d = (u, v),\ e = (v, w) \in \mathcal{E}_p \tag{28d}$$

$$t_e \leq \overline{f}_e(\mathbf{q}_e, \mathbf{q}'_e) h_e^M + \overline{h}_e(\mathbf{q}_e, \mathbf{q}'_e) f_e^m - f_e^m h_e^M \qquad \forall\, e \in \mathcal{E} \tag{28e}$$

$$t_e \leq \overline{f}_e(\mathbf{q}_e, \mathbf{q}'_e) h_e^m + \overline{h}_e(\mathbf{q}_e, \mathbf{q}'_e) f_e^M - f_e^M h_e^m \qquad \forall\, e \in \mathcal{E} \tag{28f}$$

$$t_e \geq \underline{f}_e(\mathbf{q}_e, \mathbf{q}'_e) h_e^m + \underline{h}_e(\mathbf{q}_e, \mathbf{q}'_e) f_e^m - f_e^m h_e^m \qquad \forall\, e \in \mathcal{E} \tag{28g}$$

$$t_e \geq \underline{f}_e(\mathbf{q}_e, \mathbf{q}'_e) h_e^M + \underline{h}_e(\mathbf{q}_e, \mathbf{q}'_e) f_e^M - f_e^M h_e^M \quad \forall\, e \in \mathcal{E}. \tag{28h}$$

The final optimal collision-free trajectory generated by the proposed GCSGC motion planner is the lowest cost trajectory among the trajectories optimized using (28) that are contained within the region paths computed by (27).

IV. Simplifying the Graph of Convex Sets to Improve Online Computational Efficiency

A. Overview

The graph structure developed in Section III-C is more complicated than the graph structure of the original GCS motion planner as the collision-free regions are further divided into collision-free linear-cost regions. This is particularly true when high approximation accuracy of the nonconvex cost function is required, as seen in Fig. 4(c). This complicated graph structure can greatly decrease the online computational efficiency of the motion planner because the graph structure contains many edges that are unlikely to belong to the optimal trajectory. Therefore, a graph preprocessing technique is developed in this section to simplify the graph of convex sets. To simplify the underlying graph structure of the planner, a representative graph structure is first computed in Section IV-B to provide a heuristic for the edge costs of the graph of convex sets. Using this representative graph structure, an online graph preprocessing technique is developed in Section IV-C to remove high-cost paths and cycles once a start and a goal configuration have been provided. Finally, the optimal trajectory is computed by the GCSGC algorithm in (27) and (28) using the above simplified graph structure. The flowchart of the entire proposed motion planner is shown in Fig. 7.

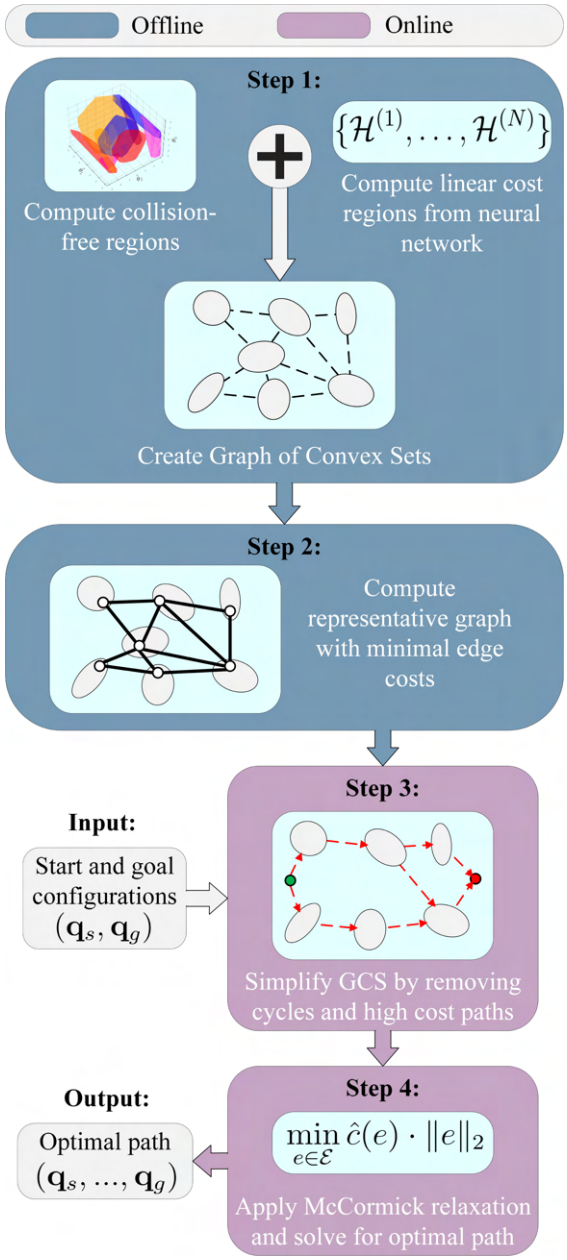


Fig. 7. Flowchart of the proposed motion planner GCSGC is shown.

B. Solving for the Representative Graph Structure

Because the vertices of the graph of convex sets are the intersections of the collision-free linear-cost regions, the edge costs are not known until a point inside each of these regions is determined. To determine the optimal points inside these regions that provide an effective heuristic for the edge costs, the edge costs between a point and all of its neighbors should be minimized. This is equivalent to minimizing the sum of all of the edge costs of the entire graph structure, which is defined as follows:

$$\min_{\mathbf{q}_u, \mathbf{q}_v} \quad \sum_{(u,v) \in \mathcal{E}} \ell_e(\mathbf{q}_u, \mathbf{q}_v) \tag{29a}$$

$$\text{s.t. } \mathbf{q}_u \in \mathcal{V}_u \qquad \forall\, \mathcal{V}_u \in \boldsymbol{\mathcal{V}}\,. \tag{29b}$$

The above optimization problem is large and nonconvex because the number of vertices in the graph of convex sets can be very large and the edge cost is nonconvex. Sequential linear programming (SLP) is an effective choice to solve this large nonconvex problem [49].

A 2-D illustrative example is given to demonstrate the concept of the representative graph structure. The four collision-free regions of the planar 2R robot from Fig. 1(b) are intersected with the 12 linear-cost regions from Fig. 4(a) to produce the graph of convex sets shown in Fig. 8(a). The resulting graph of convex sets contains 49 vertices and 167 edges. The centers of the intersections of the collision-free linear-cost regions and their respective edges used to initialize the above sequential linear program are shown in Fig. 8(b). The representative graph structure solved by (29) is shown in Fig. 8(c). The nodes of both graph structures are shown as white dots, while the edges are shown as colored lines, where the color corresponds to the edge's cost. It can be seen that the nodes in the top-right corner (a high-cost region) in Fig. 8(c) attempt to optimize their edge costs by moving close to each other to reduce their distance, while also avoiding moving into the highest cost region. The nodes spread out in the top-left corner (a low-cost region) in

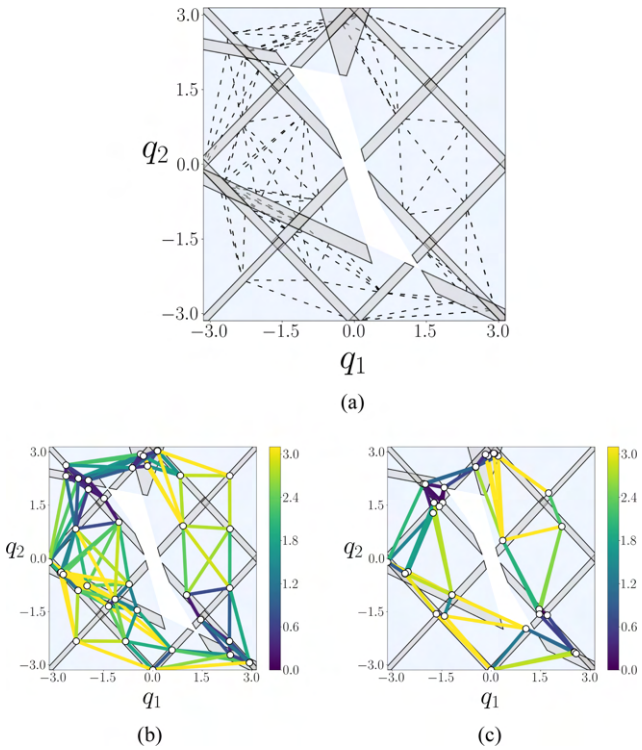


Fig. 8. (a) Graph of convex sets created by intersecting the collision-free regions in Fig. 1(b) and the linear-cost regions in Fig. 4(a). (b) Initial representative graph structure whose nodes are the centers of the intersections of the collision-free linear-cost regions. (c) Final representative graph structure after minimizing the graph's collective edge costs.

Fig. 8(b) move to the lowest cost region to reduce both their distance and individual configuration cost.

C. Simplifying the Graph of Convex Sets by Removing Cycles and High-Cost Paths for a Given Task

Based on the edge costs in the representative graph structure, an online graph preprocessing technique is developed to simplify the graph of convex sets once a start and a goal configuration have been provided. To accomplish this, the method removes high-cost paths to reduce the number of candidate paths that the GCSGC planner needs to handle. In addition, this method removes cycles by solving for directed paths, which removes many of the cycles contained in the complex undirected graph of convex sets. This method will finally produce a much simpler and directed graph of convex sets, which significantly improves the online computational efficiency of the GCSGC planner.

The procedure to simplify the graph of convex sets is as follows. First, after the start and goal configurations, $\mathbf{q}_s$ and $\mathbf{q}_g$, are given, they are added to the representative graph structure. Next, the minimum path cost between each vertex in the representative graph and the goal is then computed using the Dijkstra's algorithm. Then, the k-shortest directed paths from the start configuration to the goal configuration are estimated based on the minimum path cost from each vertex to the goal. Finally, all of the edges that are not in the k-shortest paths are removed from the original graph of convex sets, which results in a much smaller graph structure containing very few cycles and keeping the potentially optimal paths.

Because this procedure must be done online and directly computing the k-shortest paths is not efficient enough for large graphs [50], [51], a method for estimating the k-shortest paths is developed. This method iteratively builds low-cost paths, starting at $\mathbf{q}_s$, by randomly adding neighboring vertices based on the lowest-cost path from these neighbors to the goal configuration $\mathbf{q}_g$ until this goal configuration is reached. At the most recently added vertex u, all of its neighboring vertices $\{v : (u, v) \in \mathcal{E}_u\}$ are checked to determine if the lowest potential cost when traveling down each of these vertices is within a certain range of the lowest cost in the representative graph using the following inequality:

$$C(s, v) + \widehat{C}(v, g) \leq \alpha \cdot C_{\text{min}} \tag{30}$$

where the function $C(s, v)$ represents the cost from the starting configuration to v, the function $\widehat{C}(v, g)$ represents the minimum path cost from v to the goal g, and C_{min} represents the cost of the lowest cost path from s to g in the representative graph. The value $\alpha \geq 1$ is used to determine the largest allowed path cost with respect to C_{min}. In addition, to avoid creating paths with cycles, vertices that have already been added to the current path cannot be added again. After any infeasible neighboring vertices have been removed, one of the remaining vertices is randomly selected and added to the path using the following probability distribution:

$$p(v|u) = \frac{\exp(-C(u, v) - \widehat{C}(v, g))}{\sum_{(u,w)\in\mathcal{E}_u^*} \exp(-C(u, w) - \widehat{C}(w, g))} \tag{31}$$

where the edge set $\mathcal{E}_u^*$ represents all of the edges (u, v) beginning with the vertex u which satisfy constraint (30) and v is not in the current path. The likelihood of a vertex being added to the currently computed path is inversely proportional to the minimum potential path cost when traveling through this vertex. The above method for building paths is then repeated k times to produce a set of edges that are used during the trajectory optimization process.

An illustrative example of implementing this method to simplify the underlying graph structure of Fig. 8(a) is shown in Fig. 9. The simplified graph structures using $\alpha = 1.0$, $\alpha = 1.02$, and $\alpha = 1.11$ are shown in Fig. 9(a)–(c), respectively. All three of these examples use $k = 1000$. The gray regions represent the vertices of the simplified graph structures, while the red arrows connecting these vertices are the directed edges of the simplified GCSs. The original graph of convex sets containing 49 vertices and 167 undirected edges (equivalent to 334 directed edges) is simplified to graph structures with 10 vertices and 9 directed edges when $\alpha = 1.0$, 29 vertices and 107 directed edges when $\alpha = 1.02$, and 43 vertices and 211 directed edges when $\alpha = 1.11$. The optimal trajectory from the green starting configuration to the red goal configuration can be efficiently computed using the respective simplified graph structures, as shown by the blue curves, which takes 0.002, 0.08, and 0.22 s for $\alpha = 1.0$, $\alpha = 1.02$, $\alpha = 1.11$, respectively. The costs of these optimal trajectories for the respective values of α are 10.37, 10.25, and

IEEE Transactions on Robotics (T-RO) paper, presented at ICRA 2026, Vienna, Austria. Cite as T-RO paper.

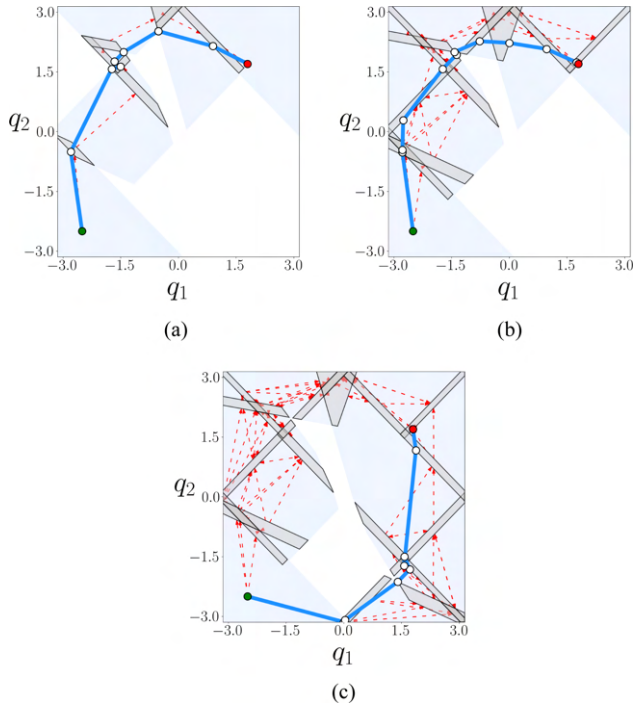


Fig. 9. Example of the graph preprocessing method is shown for a 2-D configuration space. The graph structure from Fig. 8(a) is simplified in (a)–(c), with $\alpha = 1.0$, $\alpha = 1.02$, and $\alpha = 1.11$, respectively. The gray regions are the vertices of the simplified graph structures and the red arrows represent the directed connections between these vertices. Optimal trajectories are planned using the simplified graph structure in (a)–(c) from the green configuration to the red configuration, shown by the blue curves, which takes 0.002, 0.08, and 0.22 s, respectively.

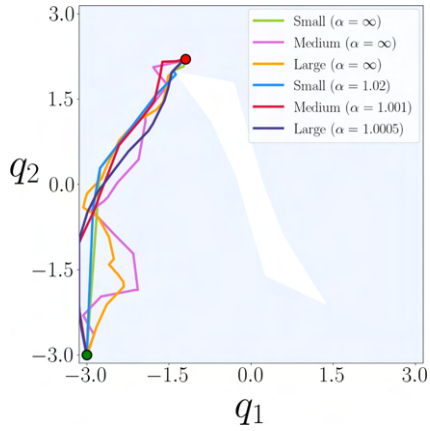


Fig. 10. Trajectories produced by the GCSGC algorithm using small, medium, and large neural networks both with and without preprocessing are shown for an example task. The green point represents the starting configuration for the task, and the red point represents the goal configuration.

10.26. These results indicate that it is important to choose an appropriate value for α. In this case, $\alpha = 1.02$ produces a simple enough graph of convex sets while also producing the lowest cost trajectory. When α is too small, such as in Fig. 9(a), it is highly likely that the above graph simplification method will remove too many regions and potentially optimal region paths that the truly optimal trajectory might belong to. Conversely, when α is too large, such as in Fig. 9(c), the simplified graph structure will retain too many regions and suboptimal region paths that can decrease the online computational efficiency of the motion planner. In addition, the added complexity of the underlying graph structure can cause the relaxations in (27) to loosen, and thus, decrease the quality of the produced trajectory.

V. Results

A. Two-Dimensional Experiments

1) Comparison Between Different Sized Neural Networks With and Without Preprocessing: To demonstrate the effects of different sized neural networks and the benefits of preprocessing on the proposed GCSGC motion planner, the collision-free regions from Fig. 1(b) are intersected with the small, medium, and large neural networks from Fig. 4 to create three different GCS. The small graph of convex sets contains 49 vertices and 167 edges, the medium graph of convex sets contains 881 vertices and 4039 edges, and the large graph of convex sets contains 1272 vertices and 5110 edges. Using a fixed start configuration, 1000 goal configurations were randomly sampled to create a set of planning tasks, where the eggbox cost function should be minimized while moving from the start to the goal.

First, the optimal trajectories are computed using the original GCS. The average trajectory costs and average planning times for the three different graph structures are shown in Table I. In terms of computational efficiency, the online computational time increases with increases in the size of the neural networks and their corresponding GCS. In addition, the accuracy of the approximated cost function increases with increases in the neural network size. However, this does not lead to lower trajectory costs because larger GCS reduce the tightness of the convex relaxation of the flow variables in (27f). This can be seen by the fact that the small graph of convex sets notably outperforms the two larger GCS.

Next, the preprocessing techniques from Section IV are applied, and the optimal trajectories are computed using each simplified graph of convex sets, as shown in Table I. The values $\alpha = 1.02$ and $k = 1000$, $\alpha = 1.001$ and $k = 1250$, and $\alpha = 1.0005$ and $k = 1500$ were used during the preprocessing stage of the small, medium, and large GCS, respectively. For each graph of convex sets, both the trajectory costs and computational efficiency significantly improved when preprocessing was applied. In this experiment, it can be seen that the medium and large GCS perform almost identically, and both outperform the small graph of convex sets.

Fig. 10 shows the trajectories produced by each of the GCS both with and without preprocessing for one of the tasks. The large graph of convex sets with preprocessing produced the lowest cost trajectory, which is shown as the purple curve. It can be seen that this trajectory quickly moves out of the high-cost region in the bottom left, and then moves into the lower cost region in the top left.

2) Comparison Between Optimal Trajectories From GCSGC With Shortest-Distance Trajectories: Although the trajectory cost depends on the length of the trajectory, low-cost trajectories

TABLE I
COMPARISON BETWEEN DIFFERENT SIZED NEURAL NETWORKS WITH AND WITHOUT PREPROCESSING

Metric	No Pre-processing			Pre-processing		
	Small ($\alpha = \infty$)	Medium ($\alpha = \infty$)	Large ($\alpha = \infty$)	Small ($\alpha = 1.02$)	Medium ($\alpha = 1.001$)	Large ($\alpha = 1.0005$)
Trajectory Cost	7.03 ± 2.63	8.97 ± 4.81	8.00 ± 4.39	6.02 ± 2.17	5.75 ± 2.04	5.75 ± 2.03
Online Time [s]	0.23 ± 0.07	8.97 ± 1.95	9.94 ± 1.07	0.12 ± 0.08	0.42 ± 0.29	0.44 ± 0.26

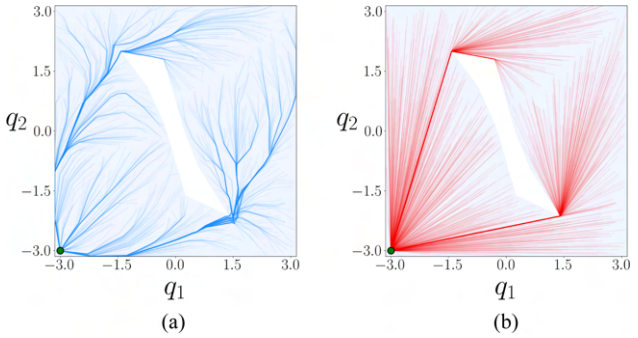


Fig. 11. Trajectories for 1000 planning tasks with a fixed start location and random goal locations are given, where the blue trajectories in (a) are the lowest cost trajectories computed by the GCSGC motion planner and the red trajectories in (b) are the shortest distance trajectories computed by the GCS motion planner.

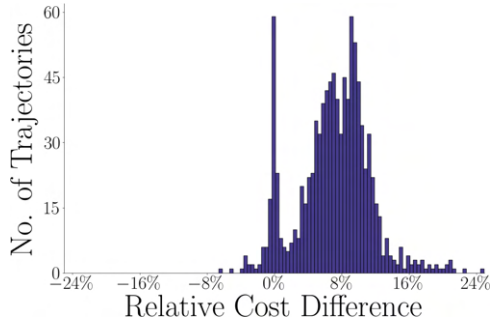


Fig. 12. Number of trajectories within different ranges of the relative difference in trajectory cost between GCSGC and the shortest distance trajectories are shown.

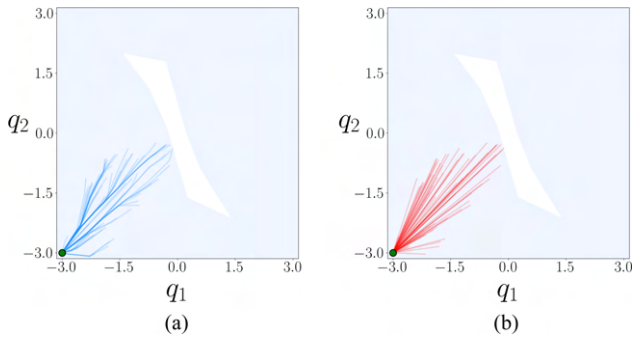


Fig. 13. 7% of tasks where the trajectories computed by the GCSGC planner (a) are higher cost than the shortest-distance trajectories (b) are shown.

do not necessarily minimize distance. Thus, optimal trajectories must make a tradeoff between traveling shorter distances and moving through lower cost configurations. To analyze the proposed method's ability to effectively make this tradeoff, the trajectories produced by the proposed GCSGC motion planner are compared with the shortest distance trajectories. The medium-sized graph of convex sets is used in this experiment, where this graph structure and its representative graph were computed in 15.9 s. Using a fixed start configuration $[-3, -3]$, 1000 goal configurations were randomly sampled to create a set of planning tasks. The proposed planner, GCSGC, was used to compute the optimal collision-free trajectories using $\alpha = 1.001$ and $k = 1500$. To provide a baseline for comparisons, the original GCS motion planner is used to compute the shortest distance collision-free trajectories for each task. The average online computational time for the GCSGC motion planner and the GCS motion planner to compute each trajectory was 0.62 and 0.017 s, respectively.

The trajectories computed by the GCSGC motion planner are shown as the blue curves in Fig. 11(a), while the trajectories computed by the GCS motion planner are shown as the red curves in Fig. 11(b). It is clear from these figures that the GCSGC planner produces much more complicated trajectories as it attempts to avoid high-cost regions (the bottom-left and top-right regions) and move into low-cost regions (the bottom-right and top-left regions). The costs of the resulting trajectories from both planners are computed using the actual eggbox cost function. The differences in the trajectory costs are grouped into different ranges, and the number of trajectories within each range is shown in Fig. 12. A positive difference corresponds to the trajectory computed by GCSGC having a lower cost than the trajectory computed by GCS, and a negative difference corresponds to the opposite case.

Over 93% of the trajectories computed by GCSGC have a lower cost than the trajectories computed by GCS. For the other 7% of the tasks, the GCSGC trajectories are shown in Fig. 13(a) and the GCS trajectories are shown in Fig. 13(b). It can be seen that this is only the case when the distance between the start and goal locations is very short, and thus, it is lower cost to directly move through the high-cost region, as opposed to taking a long motion around it. The GCSGC planner does not exactly produce these short straight-line trajectories because it uses an approximation of the eggbox function. However, the produced GCSGC trajectories are still very similar to the straight-line trajectories. In contrast, approximately 22% of the GCSGC trajectories have costs more than 10% lower than the corresponding GCS trajectories, which are shown in Fig. 14(a) and (b), respectively. It can be seen that these tasks typically have a large distance between the start and goal locations, and the lowest cost trajectories quickly move out of the high-cost region in the bottom left and enter into the low-cost regions in the top left and bottom right. These 2-D experiments show the

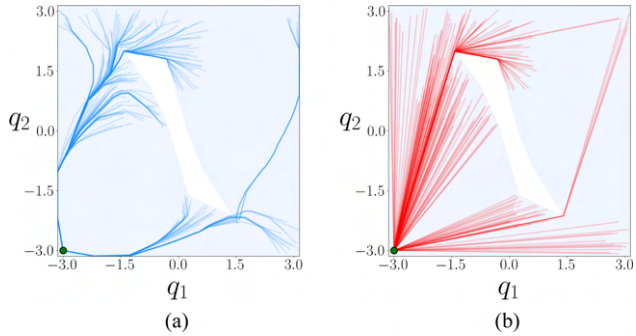


Fig. 14. 22% of tasks where the trajectories computed by the GCSGC motion planner (a) are more than 10% lower in cost than the shortest distance trajectories (b) are shown.

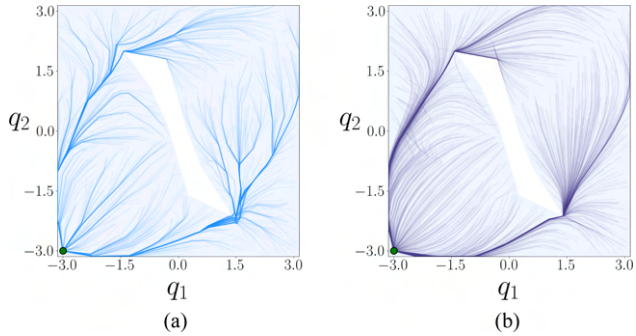


Fig. 15. Trajectories for 1000 planning tasks with a fixed start location and random goal locations are given, where the blue trajectories in (a) are the lowest cost trajectories computed by the GCSGC motion planner and the purple trajectories in (b) are the lowest cost trajectories computed by the GCS-SLP motion planner.

advantages of the trajectories computed by the proposed motion planner compared to the shortest-distance trajectories.

3) Comparison Between GCSGC and GCS-SLP: The benefits of the proposed GCSGC motion planner, which uses linearized cost functions and McCormick relaxations to convexify this complicated nonconvex problem, are demonstrated against the direct application of nonlinear programming methods to compute optimal trajectories. The baseline nonlinear programming approach used in this experiment is constructed by combining the traditional GCS formulation to handle collision avoidance with an SLP-based approach to minimize the nonconvex edge costs. In particular, SLP is used to locally optimize the nonconvex edge costs within the original GCS formulation. SLP is an effective and efficient choice for this nonconvex optimization problem, where all of the constraints are linear in these experiments.

To compare the performance of the GCSGC motion planner with the GCS-SLP algorithm, the same 1000 tasks from the previous experiment are used. Therefore, the results for the GCSGC planner are identical to the results of the previous experiment, as shown in Fig. 15(a). The optimal trajectories computed by the GCS-SLP algorithm using the HiGHS linear program solver [49] are shown in Fig. 15(b). The average online computational time for the GCSGC motion planner and the GCS-SLP motion planner to compute each trajectory was 0.62 and 3.56 s, respectively. This almost six-times improvement in computational efficiency demonstrates the large performance benefits that can be achieved by the proposed GCSGC planner over nonlinear programming approaches. The differences in the trajectory costs are grouped into different ranges, and the number of trajectories within each range is shown in Fig. 16. A positive difference corresponds to the trajectory computed by GCSGC having a lower cost than the trajectory computed by GCS-SLP, and a negative difference corresponds to the opposite case. It can be seen that the difference in trajectory cost is very small, where 95% of all of the comparable trajectories have a 5% or less relative difference in cost. However, some trajectories produced by the GCS-SLP algorithm have relative costs over 20% larger than the costs of the comparable trajectories computed by GCSGC. This is because the GCS-SLP algorithm can become stuck in local minima based on the initial conditions provided to the algorithm.

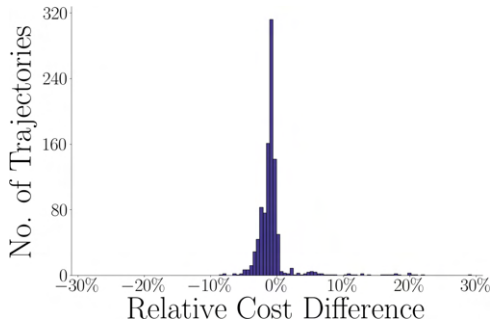


Fig. 16. Number of trajectories within different ranges of the relative difference in trajectory cost between GCSGC and GCS-SLP.

B. Seven-Dimensional Experiments

1) Experiment Setup: To further validate the performance of the proposed motion planner in high-dimensional spaces, several experiments are performed using a 7-DoF Kinova Gen3 robot arm. The GCSGC motion planner in (27) and (28) is implemented using the CVXPY library [52], a popular library for formulating convex optimization problems in the Python programming language. Once formulated, it is solved using the Mosek solver [53]. All of the graph computations, such as their representation, storage, and preprocessing, are performed using the NetworkX library [54], a popular and efficient graph analysis Python library.

The tasks in these experiments are designed to test the motion planner's ability to reorganize a bookshelf while simultaneously avoiding collisions and minimizing some cost function. There are five tasks in total, the first task requires the robot arm to move from a start configuration outside the bookshelf to a goal configuration inside the bookshelf to grab a large toy bear. The next task is to move the large toy bear to the top of the bookshelf. After this, the robot arm moves back into the bookshelf to grab a small toy bear. This small toy bear is then moved to the shelf below it. Finally, the fifth task moves the robot arm back to its starting position. The environment for these experiments is shown in

Fig. 17. Environment for the bookshelf reorganizing tasks.

Fig. 17. To represent the collision-free configuration space for this environment, six collision-free regions were computed in 986 s using the IRIS-NP algorithm introduced in Section II-A.

To show the effectiveness of the proposed motion planner, it is compared to several other state-of-the-art motion planners, which are divided into the categories multiquery and single-query. These motion planners are compared in terms of their computational efficiency, memory footprint, and the quality of their produced trajectories. To ensure that the computational efficiency of the tested motion planners is fairly compared, all of the planners are tested using the same computer, with an Apple M2 Pro CPU and 16 GB of RAM.

2) Comparison With Multiquery Algorithms: Because the underlying graph structure of the GCSGC motion planner can be reused for any tasks within the same environment, it is classified as a multiquery planner. Therefore, it is first compared to the current state-of-the-art asymptotically optimal multiquery algorithm, PRM*. The PRM* algorithm uses an offline stage to build a planning roadmap by randomly sampling joint configurations and testing for collision-free connections between these sampled configurations. The online stage of the PRM* algorithm is performed by adding the start and goal configurations to the roadmap and then using a graph search method, such as the Djikstra's algorithm, to compute an optimal collision-free trajectory within the roadmap. The PRM* algorithm in these experiments is implemented using the OMPL library [55], which is a popular open-source motion planning library. In addition, similar to the 2-D experiments, the GCSGC planner is also compared to the GCS-SLP algorithm.

A highly nonlinear and nonconvex cost function, the postfailure dexterity after an arbitrary joint is locked, is used in this comparison. It is defined as follows [56]:

$$\mathcal{K}(\mathbf{q}) = \min_{f=1}^{n} \{{}^{f}\sigma_m(\mathbf{q})\} \tag{32}$$

where n is the number of joints, ${}^{f}\sigma_m(\mathbf{q})$ is the minimum singular value of the robot's Jacobian at configuration $\mathbf{q}$ after joint f is locked, which represents the worst-case dexterity after joint f fails. Based on this metric at each configuration, the cost of the entire trajectory is defined as follows:

$$C(\mathbf{r}(\cdot)) = \int_0^T \beta - \mathcal{K}(\mathbf{r}(t))\, dt \tag{33}$$

where the function $\mathbf{r}$ represents a trajectory as a function of time t with duration T, and β is chosen such that it is greater than the largest possible value of $\mathcal{K}(\mathbf{q})$, thus ensuring the trajectory cost is always positive. It can be seen that a trajectory with a lower cost represents a safer trajectory after an arbitrary joint failure.

To approximate the above cost function, a neural network with 4 hidden layers and 6, 6, 8, and 8 neurons in each respective layer was used. The network is trained using collision-free configurations as the inputs and the respective costs at the configurations as the outputs. The mean squared error loss function is minimized using the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (L-BFGS) algorithm, and the neural network reaches a minimum loss of 0.009, after 55 s of training, which is an acceptable level of accuracy . The above neural network divides the configuration space into 5118 linear-cost regions, and these linear-cost regions are then intersected with the six collision-free regions to form the underlying graph of convex sets used by the GCSGC motion planner in this experiment. This graph of convex sets contains 30 973 vertices and 390 997 edges. After the underlying graph structure was constructed, the representative graph structure was computed, where these steps were completed in 4337 s, and the entire offline process took 5378 s.

The construction of the PRM* roadmap in this experiment is similar to [37], where this process is divided into two offline phases. The first phase connects the centers of the intersections of collision-free regions together to form a basic skeleton for the roadmap. This basic skeleton is further developed in the second phase by randomly sampling configurations inside the six collision-free regions and testing for collisions along the connections between these configurations also using the collision-free regions. The final roadmap contains 103 692 vertices and 3 378 980 edges. The total offline time of the PRM* algorithm is 5407 s, which is almost identical to the offline time of the GCSGC planner.

The trajectory costs, online computational time, and memory footprint of the GCSGC planner with $k = 1000$ and $\alpha = 1.0$, $\alpha = 1.02$, and $\alpha = 1.05$, PRM*, and GCS-SLP are shown in Table II . The trajectory costs of the proposed GCSGC planner are significantly lower than the trajectory costs of PRM* for all values of α, and in particular the trajectory costs when $\alpha = 1.02$. The trajectory costs of the GCSGC planner for all values of α are lower than the trajectory costs of the GCS-SLP planner for four of the five tasks, where GCS-SLP performs slightly better in task 2. The online computational time of the GCSGC planner has two parts: 1) the time used by the graph preprocessing stage; and 2) the time taken to solve the two convex optimization problems, (27) and (28). The online computational time of the GCSGC planner with $\alpha = 1.0$ is the lowest for four of the five tasks, and it is roughly 10 to 15 times faster than PRM* in these cases. PRM* is the fastest algorithm in Task 4, however it is only 0.003 s faster than the GCSGC planner with $\alpha = 1.0$. This fast runtime is due to the extremely short distance trajectory for this

TABLE II
COMPARISON BETWEEN GCSGC PLANNER[1] AND MULTIQUERY PLANNERS PRM*[2] AND GCS-SLP[3]

Task	Metric	GCSGC (α = 1.0)	GCSGC (α = 1.02)	GCSGC (α = 1.05)	PRM*	GCS-SLP
1	Trajectory Cost	2.03	**1.98**	2.06	2.73	2.08
	Online Time [s]	**0.012 + 0.008 = 0.02**	0.23 + 1.33 = 1.56	0.27 + 3.5 = 3.77	0.28	337.7
	Memory [MB]	167 + 0.94 = 167.94	167 + 32 = 199	167 + 81 = 248	1217	**4.64**
2	Trajectory Cost	3.29	3.24	3.28	4.44	**3.09**
	Online Time [s]	**0.021 + 0.007 = 0.028**	0.22 + 0.92 = 1.14	0.35 + 3.1 = 3.45	0.42	352.6
	Memory [MB]	167 + 0.71 = 167.71	167 + 12 = 179	167 + 48 = 215	1217	**4.28**
3	Trajectory Cost	1.78	**1.70**	**1.70**	3.71	2.31
	Online Time [s]	**0.014 + 0.006 = 0.02**	0.18 + 0.32 = 0.5	0.21 + 1.47 = 1.68	0.30	466.9
	Memory [MB]	167 + 0.59 = 167.59	167 + 4.9 = 171.9	167 + 29 = 196	1217	**4.28**
4	Trajectory Cost	0.5	**0.49**	0.58	1.69	1.15
	Online Time [s]	0.006 + 0.004 = 0.01	0.15 + 0.056 = 0.206	0.16 + 0.41 = 0.57	**0.007**	314.5
	Memory [MB]	167 + 0.44 = 167.44	167 + 2.7 = 169.7	167 + 12 = 179	1217	**4.33**
5	Trajectory Cost	1.86	**1.78**	1.81	2.27	2.67
	Online Time [s]	**0.025 + 0.009 = 0.034**	0.31 + 2.16 = 2.47	0.38 + 5.1 = 5.48	0.35	417.9
	Memory [MB]	167 + 0.96 = 167.96	167 + 45 = 212	167 + 79 = 246	1217	**4.69**

[1]The training time for the neural network approximating the post-failure dexterity cost function was 55 s; the time taken to compute the collision-free regions was 986 s; and the time taken to compute the graph of convex sets for the GCSGC motion planner was 4337 s. The total offline time for the GCSGC motion planner was 5378 s.
[2]The offline time for the PRM* motion planner was 5407 s.
[3]The offline time (computing the collision-free regions) for the GCS-SLP motion planner was 986 s.

task, where the PRM* trajectory is comprised of only four nodes including the start and goal configurations. The GCS-SLP is significantly slower than both the GCSGC and PRM* planners for all five tasks, where GCSGC ($\alpha = 1.0$) is at least 12 000 times faster than GCS-SLP. For the GCSGC planner, it can be seen that increases in α can cause an increase in the algorithm's online runtime due to the increase in the complexity of the simplified graph structure, but even the slowest runtimes are still reasonable for many applications. This leads to a tradeoff between the quality of the computed trajectory and the time taken to plan it. The optimal value of α can be determined based on the priority of the trajectory quality versus the computational efficiency for a specific application.

For low-resource systems, although GCS-SLP is quite memory efficient, it is not a feasible choice due to its computational inefficiency. As both the GCSGC and PRM* motion planners rely on large graph structures in order to produce high quality trajectories, their memory footprints must be considered. The GCSGC memory entries in Table II contain two parts: 1) the memory used by the representative graph structure; and 2) the memory used during the optimization stage. It can be seen that the GCSGC motion planner uses significantly less memory than the PRM* motion planner. This is because the representative graph structure is used during the graph pre-processing which requires much less memory than the entire underlying graph of convex sets. In addition, only the regions needed to represent the simplified graph structure are loaded into memory for solving the optimization problems. On the contrary, the PRM* graph structure contains roughly three times more vertices and nine times the number of edges than the representative graph structure. This experiment demonstrates the advantages of the proposed GCSGC motion planner in terms of trajectory quality, computational efficiency, and memory footprint.

Example end-effector trajectories obtained by following the joint trajectories produced by the tested motion planners for the above experiment are shown in Fig. 18. These trajectories were planned to complete Task 2, which requires the robot to move a toy bear from the middle shelf to above the bookshelf. The blue, purple, and red curves represent the end-effector trajectories computed by the GCSGC planner with $\alpha = 1.02$, the PRM* planner, and the GCS-SLP planner, respectively. It can be seen that all of these trajectories are collision-free. However, the trajectory produced by the PRM* planner moves closer to the boundary of the reachable workspace, which are regions with low postfailure dexterity. The simulation video of the robot motions planned by both planners for all five tasks is available on YouTube.[1]

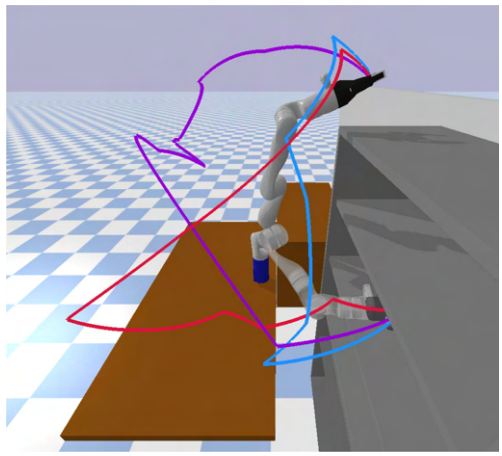


Fig. 18. End-effector trajectories computed by the GCSGC planner, the PRM* planner, and the GCS-SLP planner for Task 2 of the multiquery comparisons experiment are shown as the blue, purple, and red curves, respectively.

3) Comparison With Single-Query Algorithms: Some additional comparisons are conducted between the GCSGC motion planner and several state-of-the-art single-query motion planners. There are two main categories of single-query planners used in this experiment: 1) optimization-based (TrajOpt); and

[1][Online]. Available: https://youtu.be/9VPRXp06uKk

TABLE III
COMPARISON BETWEEN GCSGC PLANNER[1] AND SINGLE-QUERY PLANNERS

Task	Metric	GCSGC (α = 1.0)	GCSGC (α = 1.04)	GCSGC (α = 1.075)	TrajOpt	BIT*	AIT*	RRT*
1	Trajectory Cost	**2.88**	2.89	**2.88**	2.72	3.01	2.97	3.01
	Online Time [s]	**0.0017 + 0.003 = 0.0047**	0.13 + 0.59 = 0.72	0.09 + 1.01 = 1.1	63.4	420	660	420
	Valid	Yes	Yes	Yes	No	Yes	Yes	Yes
2	Trajectory Cost	**1.67**	1.68	2.16	1.75	2.71	2.44	1.95
	Online Time [s]	**0.015 + 0.008 = 0.023**	0.43 + 5.5 = 5.93	0.47 + 8.6 = 9.07	86.7	900	1080	1020
	Valid	Yes	Yes	Yes	No	Yes	Yes	Yes
3	Trajectory Cost	**0.9**	0.91	1.26	1.08	0.94	1.24	0.94
	Online Time [s]	**0.003 + 0.005 = 0.008**	0.14 + 1.4 = 1.54	0.32 + 7.92 = 8.24	19.3	780	480	1260
	Valid	Yes	Yes	Yes	Yes	Yes	Yes	Yes
4	Trajectory Cost	1.92	1.13	**1.11**	0.9	1.29	1.33	N/A
	Online Time [s]	**0.003 + 0.008 = 0.011**	0.26 + 0.33 = 0.59	0.2 + 0.92 = 1.12	22.4	540	480	3600
	Valid	Yes	Yes	Yes	No	Yes	Yes	No
5	Trajectory Cost	3.43	2.78	2.8	2.34	2.64	2.92	**2.15**
	Online Time [s]	**0.002 + 0.003 = 0.005**	0.12 + 0.32 = 0.44	0.14 + 0.94 = 1.08	74.3	1200	1140	1500
	Valid	Yes	Yes	Yes	Yes	Yes	Yes	Yes

[1]The training time for the neural network approximating the gravity vector cost function was 33 s; the time taken to compute the collision-free regions was 986 s; and the time taken to compute the graph of convex sets for the GCSGC motion planner was 3241 s. The total offline time for the GCSGC motion planner was 4260 s.
[†]Note that boldface is used to represent the optimal result only for valid trajectories.

2) sampling-based (BIT*, AIT*, RRT*). A custom implementation of TrajOpt is used, where the sequential convex programs are solved using Mosek, and the continuous collision detection method in [24] is implemented using PyBullet [57]. The following hyperparameters are used: 25 nodes per trajectory, the safe distance 0.005 m, the collision check distance 0.05 m, the initial penalty coefficient 0.1, the penalty scaling 25.0, and all tolerances are set to 0.0001. The OMPL implementations of the BIT*, AIT*, and RRT* algorithms are used with the following hyperparameters: k-nearest neighbors search, rewiring factor equal to 1.001, and the maximum range and goal bias for RRT* are set to $\pi/2$ and 0.1, respectively. These sampling-based methods attempt to solve for the lowest cost trajectory until 300 s pass without finding a better trajectory, with an upper limit of 1 h of planning time.

Another highly nonlinear and nonconvex cost function, the norm of the gravity vector, is used in this experiment. It is the function $\mathbf{g}$ in the following dynamic equation for a robot arm [58]

$$\mathbf{M}(\mathbf{q})\ddot{\mathbf{q}} + \mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})\dot{\mathbf{q}} + \mathbf{g}(\mathbf{q}) = \boldsymbol{\tau} \tag{34}$$

where $\dot{\mathbf{q}}$ is the velocity of the robot arm, $\ddot{\mathbf{q}}$ is the acceleration, $\boldsymbol{\tau}$ is the applied joint torques, $\mathbf{M}(\mathbf{q})$ is the inertia matrix, $\mathbf{C}(\mathbf{q}, \dot{\mathbf{q}})$ is the centrifugal and Coriolis matrix, and $\mathbf{g}(\mathbf{q})$ is the gravity vector of the robot arm. Based on this configuration cost, the cost of the entire trajectory is defined as follows:

$$C(\mathbf{r}(\cdot)) = \int_0^T \|\mathbf{g}(\mathbf{r}(t))\|_2 \, dt. \tag{35}$$

Minimizing this trajectory cost will reduce the energy required to compensate for gravity.

To approximate the above cost function, a neural network with 3 hidden layers and 7 neurons in each layer was used. Using the mean squared error loss function, the network reaches a minimum loss of 0.003 in 33 s, which can approximate the given cost function very accurately. This neural network divides the configuration space into 3659 linear-cost regions, and these regions are intersected with the six collision-free regions to form the underlying graph of convex sets with 23 973 vertices and 363 519 edges, where these steps were completed in 3241 s. The total offline time used by the GCSGC planner in this experiment is 4260 s.

The trajectory costs, online computational time, and the validity (collision free) of the computed trajectories for the GCSGC planner with $k = 1000$ and $\alpha = 1.0$, $\alpha = 1.04$, and $\alpha = 1.075$, TrajOpt, BIT*, AIT*, and RRT* are given in Table III. For Tasks 1, 2, and 3, the GCSGC planner with $\alpha = 1.0$ produces the lowest cost trajectories among the valid trajectories. In Task 4, the GCSGC planner with $\alpha = 1.075$ produces the lowest cost trajectory among the valid trajectories. Even though the GCSGC planner with $\alpha = 1.04$ does not produce the lowest cost trajectories, it is always very close to the lowest cost. In Task 5, RRT*, BIT*, and TrajOpt produce lower cost trajectories because some low-cost configurations were not contained within the six collision-free regions used to build the graph of convex sets, and thus they are not reachable by the GCSGC motion planner. This problem can be solved by computing an approximate cover of the entire collision-free configuration space by using the method introduced in [59].

Regarding the online computational efficiency, GCSGC obviously performs the best because of the useful information provided by the graph structure computed offline. If the offline computational time (4260 s) were to be factored into the GCSGC results, the combined times for all of the five tasks for each sampling-based method would be close to the cumulative offline and online times of the GCSGC planner. This shows that single-query algorithms are effective choices for a single task or dynamic environments, but the GCSGC planner is a more powerful planner when several tasks need to be completed in the same environment. TrajOpt also performs well both in terms of computational efficiency and trajectory quality. However, it cannot guarantee that the computed trajectories are collision-free, as it only successfully plans two trajectories in this experiment. This is because the continuous collision detection checks in TrajOpt use the convex hull of the link geometries at two adjacent waypoints, which does not necessarily enclose all of the link

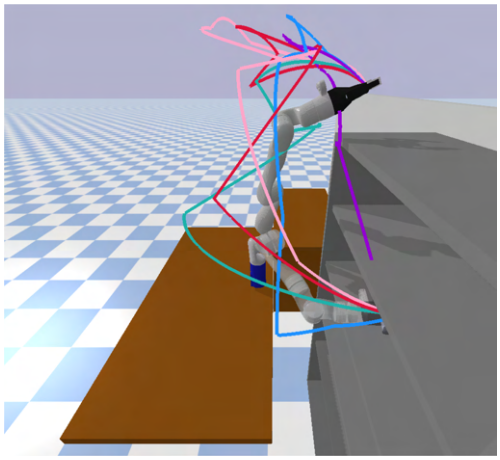


Fig. 19. End-effector trajectories computed by GCSGC, TrajOpt, BIT*, AIT*, and RRT* for Task 2 of the single-query comparisons experiment are shown as the blue, purple, green, pink, and red curves, respectively.

geometries between these two waypoints. In addition, RRT* could not find a collision-free trajectory after 1 h for Task 4.

Example end-effector trajectories obtained by following the joint trajectories produced by the tested motion planners for Task 2 of the above experiment are shown in Fig. 19. The blue, purple, green, pink, and red curves represent the trajectories computed by the GCSGC planner with $\alpha = 1.04$, TrajOpt, BIT*, AIT*, and RRT*, respectively. It can be seen that the trajectories computed by GCSGC and the sampling-based methods are collision-free, while the trajectory computed by TrajOpt collides with the top shelf. In addition, in contrast with the other trajectories, the trajectory computed by the GCSGC planner moves out of the shelf very quickly and immediately moves into an upright position. At these positions, the associated joint configurations have gravity vectors with very small magnitudes, and thus only a small amount of torque is required to compensate for gravity. The simulation video of the robot motions planned by all of the planners for all five tasks is available on YouTube.[2]

4) Physical Experiments: Two physical experiments are conducted on a real Kinova robot to further demonstrate the effectiveness of the proposed motion planner. In the first physical experiment, the cost function optimized by the motion planner is the norm of the gravity vector defined in (35). The robot first moves from an arbitrary configuration, as shown in Fig. 20(a), to a given initial configuration outside of the bookshelf, as shown in Fig. 20(b). Meanwhile, two toy bears are placed at arbitrary locations inside the bookshelf, shown in Fig. 20(b). An Azure Kinect RGB-D camera is then used to detect the bears and estimate their poses by employing the YOLOv11 computer vision algorithm [60], as shown in Fig. 20(c). Once the positions of these bears are determined, the corresponding goal configurations are computed by solving the inverse kinematics constrained within the six collision-free regions. The GCSGC planner with $\alpha = 1.0$ is used to plan an optimal collision-free trajectory from the initial configuration to the goal configuration corresponding

[2][Online]. Available: https://youtu.be/9VPRXp06uKk

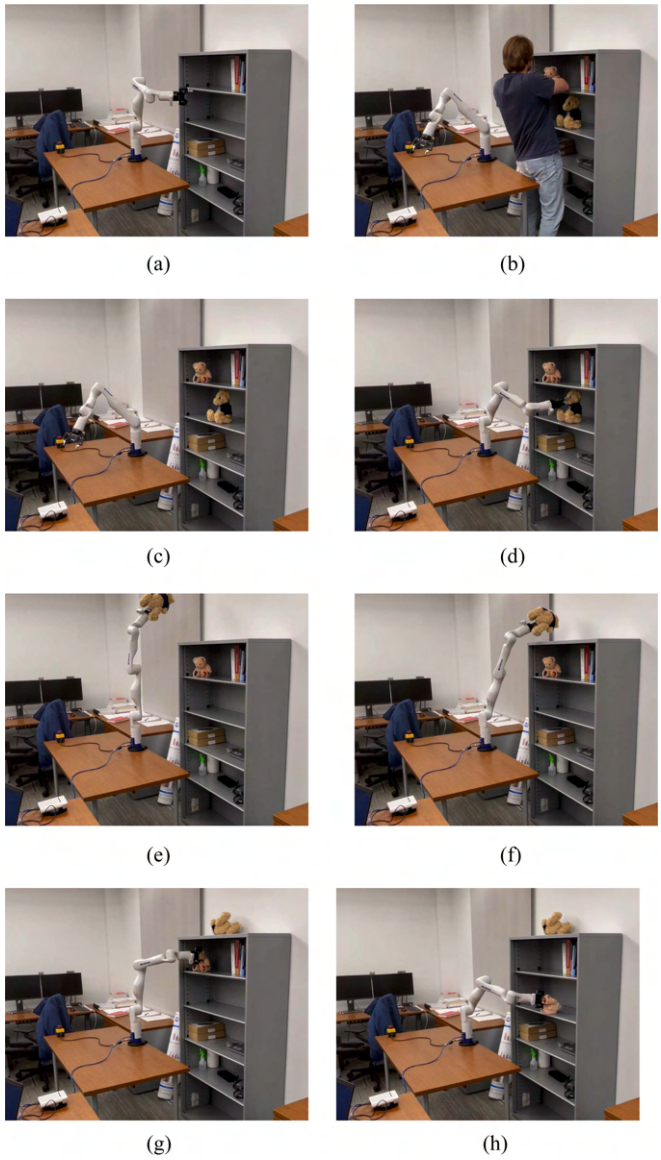


Fig. 20. Physical experiment of the bookshelf reorganizing tasks is shown, where the nonconvex cost function is the norm of the gravity vector. (a) The robot at an arbitrary configuration and the object detection camera are shown in (a). The robot moves to a given initial configuration while two bears are being placed at arbitrary locations inside of the bookshelf in (b). The camera system then determines the location of the bears and the GCSGC planner computes the trajectory to grasp the lower bear in (c). The robot reaches the first bear in (d), and a trajectory to move the bear to the top of the bookshelf is computed. This trajectory first moves the arm to an upright position to reduce the norm of the gravity vector, as shown in (e), before placing the bear on top of the bookshelf in (f). Finally, the robot grasps the second bear in (g) and moves it to the middle shelf in (h).

to the bear at the lower position. By following this trajectory, the robot moves into the bookshelf and grabs the bear, as seen in Fig. 20(d). After finishing this trajectory, the GCSGC planner immediately computes a new trajectory to move the bear to the top of the bookshelf. In this computed trajectory, the robot first moves to an upright position to reduce the norm of the gravity vector, as seen in Fig. 20(e), before finally moving the bear to the top of the bookshelf, as seen in Fig. 20(f). After dropping the

bear above the bookshelf, another trajectory is computed to reach the second bear inside of the bookshelf, as shown in Fig. 20(g). A final trajectory is computed to move this bear to the middle shelf, as shown in Fig. 20(h). The second physical experiment is identical to the first, except the cost function optimized by the GCSGC planner is the postfailure dexterity, as defined in (33). Videos for both physical experiments are available on YouTube.[3] From the videos, it can be seen that the GCSGC planner can compute high-quality trajectories very efficiently.

VI. Conclusion

This article develops a new motion planner called the GCSGC planner to compute low-cost and collision-free trajectories for nonconvex cost functions. This planner is created by building a graph structure whose vertices and edges are obtained by intersecting a set of collision-free regions with a set of linear-cost regions obtained from an ReLU neural network used to approximate the nonconvex cost function. A graph preprocessing technique is then developed to improve the efficiency of the planner and the quality of the computed trajectories. This proposed motion planner is first validated using a simple 2-D configuration space, where the GCSGC planner is constructed from different sized neural networks and tested with and without preprocessing. This experiment demonstrates that medium-sized neural networks with preprocessing can produce optimal trajectories with relatively small online runtimes. The trajectories produced by the GCSGC planner are then compared with the shortest distance trajectories. Over 93% of the 1000 trajectories computed by the GCSGC planner are lower cost than the comparable shortest distance trajectories. Finally, the GCSGC planner is compared to a nonlinear programming method GCS-SLP, where the GCSGC planner produced similar quality trajectories roughly six times faster than GCS-SLP.

The proposed motion planner was further tested in a complicated 7-D configuration space (Kinova Gen3 robot) using two different cost functions. The results show that with a similar offline computational cost to another multiquery planner, PRM*, the GCSGC planner performs better in terms of online computational efficiency, trajectory cost, and memory footprint. In addition, the GCSGC planner produced lower cost trajectories than GCS-SLP in four of five tasks, while also computing these trajectories around 12 000 times faster. Compared to several state-of-the-art single-query algorithms (TrajOpt, BIT*, AIT*, and RRT*), the GCSGC planner is able to compute the optimal collision-free trajectories for four of the five tested tasks. In this other task, RRT*, TrajOpt, and BIT* are able to compute lower cost trajectories because the collision-free regions used in the GCSGC planner do not cover some of the low-cost regions that the other trajectories computed by the single-query algorithms traveled through. In terms of computational efficiency, the cumulative offline and online time for all five tasks of the GCSGC motion planner was close to the cumulative time of each of the sampling-based planners. Finally, two physical experiments were conducted to demonstrate the proposed method's effectiveness in real-world motion planning applications. In future works, the proposed GCSGC motion planner will also be extended to operate in dynamic environments by using nonconvex constraints to handle dynamic obstacles [41].

[3][Online]. Available: https://youtu.be/9VPRXp06uKk

References

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, New York, NY, USA: Assoc. Comput. Machinery, 2022, pp. 287–290.

[2] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, Art. no. 345, 1962.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968.

[4] S. Sedighi, D.-V. Nguyen, and K.-D. Kuhnert, "Guided hybrid A-star path planning algorithm for valet parking applications," in *Proc. 5th Int. Conf. Control, Automat. Robot.*, 2019, pp. 570–575.

[5] S. Koenig and M. Likhachev, "D* lite," in *Proc. 18th Nat. Conf. Artif. Intell.*, 2002, pp. 476–483.

[6] J. Jin, Y. Zhang, Z. Zhou, M. Jin, X. Yang, and F. Hu, "Conflict-based search with D* lite algorithm for robot path planning in unknown dynamic environments," *Comput. Elect. Eng.*, vol. 105, 2023, Art. no. 108473.

[7] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Comput. Sci. Dept., Iowa State Univ., Ames, IA, USA, Res. Rep. 98-11, 1998.

[8] J. J. Kuffner and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *Proc. ICRA. Millennium Conf. IEEE Int. Conf. Robot. Automat. Symposia Proc.*, vol. 2, 2000, pp. 995–1001.

[9] A. J. La Valle, B. Sakcak, and S. M. LaValle, "Bang-bang boosting of RRTs," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2023, pp. 2869–2876.

[10] J.-C. Latombe, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. Robot. Automat.*, vol. 12, no. 4, pp. 566–580, Aug. 1996.

[11] R. Bohlin and L. E. Kavraki, "Path planning using lazy PRM," in *Proc. ICRA. Millennium Conf. IEEE Int. Conf. Robot. Automat. Symposia*, vol. 1, 2000, pp. 521–528.

[12] C. Voss, M. Moll, and L. E. Kavraki, "A heuristic approach to finding diverse short paths," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2015, pp. 4173–4179.

[13] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.*, vol. 30, no. 7, pp. 846–894, 2011.

[14] O. Arslan and P. Tsiotras, "Use of relaxation methods in sampling-based algorithms for optimal motion planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2013, pp. 2421–2428.

[15] D. Kim, Y. Kwon, and S.-E. Yoon, "Dancing PRM: Simultaneous planning of sampling and optimization with configuration free space approximation," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2018, pp. 7071–7078.

[16] R. Shome, K. Solovey, A. Dobson, D. Halperin, and K. E. Bekris, "dRRT*: Scalable and informed asymptotically-optimal multi-robot motion planning," *Auton. Robots*, vol. 44, no. 3/4, pp. 443–467, 2020.

[17] J. D. Gammell, S.S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *Proc. IEEE/RSJ Int. Conf. Intel. Robots Syst.*, 2014, pp. 2997–3004.

[18] J. D. Gammell, S.S. Srinivasa, and T. D. Barfoot, "Batch Informed Trees (BIT*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2015, pp. 3067–3074.

[19] M. P. Strub and J. D. Gammell, "Adaptively Informed Trees (AIT*): Fast asymptotically optimal path planning through adaptive heuristics," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2020, pp. 3191–3198.

[20] M. P. Strub and J. D. Gammell, "Advanced BIT* (ABIT*): Sampling-based planning with advanced graph-search techniques," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2020, pp. 130–136.

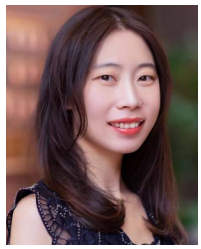
[21] M. P. Strub and J. D. Gammell, "Adaptively informed trees (AIT*) and effort informed trees (EIT*): Asymmetric bidirectional sampling-based path planning," *Int. J. Robot. Res.*, vol. 41, no. 4, pp. 390–417, 2022.

[22] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2009, pp. 489–494.

[23] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal, "STOMP: Stochastic trajectory optimization for motion planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2011, pp. 4569–4574.
[24] J. Schulman et al., "Motion planning with sequential convex optimization and convex collision checking," *Int. J. Robot. Res.*, vol. 33, no. 9, pp. 1251–1270, 2014.
[25] B. Sundaralingam et al., "CuRobo: Parallelized collision-free robot motion generation," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2023, pp. 8112–8119.
[26] B. Ichter, J. Harrison, and M. Pavone, "Learning sampling distributions for robot motion planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2018, pp. 7087–7094.
[27] A. H. Qureshi and M. C. Yip, "Deeply informed neural sampling for robot motion planning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2018, pp. 6582–6588.
[28] J. Wang, W. Chi, C. Li, C. Wang, and M. Q.-H. Meng, "Neural RRT*: Learning-based optimal path planning," *IEEE Trans. Automat. Sci. Eng.*, vol. 17, no. 4, pp. 1748–1758, 2020.
[29] J. J. Johnson, U. S. Kalra, A. Bhatia, L. Li, A. H. Qureshi, and M. C. Yip, "Motion planning transformers: A motion planning framework for mobile robots," 2021, *arXiv:2106.02791*.
[30] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," in *Proc. Int. Conf. Robot. Automat.*, 2019, pp. 2118–2124.
[31] M. J. Bency, A. H. Qureshi, and M. C. Yip, "Neural path planning: Fixed time, near-optimal path generation via oracle imitation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2019, pp. 3965–3972.
[32] M. Dalal, J. Yang, R. Mendonca, Y. Khaky, R. Salakhutdinov, and D. Pathak, "Neural MP: A generalist neural motion planner," 2024, *arXiv:2409.05864*.
[33] A. Fishman, A. Murali, C. Eppner, B. Peele, B. Boots, and D. Fox, "Motion policy networks," in *Proc. Conf. Robot Learn.*, 2023, pp. 967–977.
[34] K. Saha et al., "EDMP: Ensemble-of-costs-guided diffusion for motion planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2024, pp. 10351–10 358.
[35] J. Carvalho, A. T. Le, M. Baierl, D. Koert, and J. Peters, "Motion planning diffusion: Learning and planning of robot motions with diffusion models," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2023, pp. 1916–1923.
[36] D. González, J. Pérez, V. Milanés, and F. Nashashibi, "A review of motion planning techniques for automated vehicles," *IEEE Trans. Intell. Transp. Syst.*, vol. 17, no. 4, pp. 1135–1145, Apr. 2016.
[37] T. Marcucci, M. Petersen, D. von Wrangel, and R. Tedrake, "Motion planning around obstacles with convex optimization," *Sci. Robot.*, vol. 8, no. 84, 2023, Art. no. eadf7843.
[38] M. Petersen and R. Tedrake, "Growing convex collision-free regions in configuration space using nonlinear programming," 2023, *arXiv:2303.14737*.
[39] T. Marcucci, J. Umenberger, P. Parrilo, and R. Tedrake, "Shortest paths in graphs of convex sets," *SIAM J. Optim.*, vol. 34, no. 1, pp. 507–532, 2024.
[40] T. Cohn, M. Petersen, M. Simchowitz, and R. Tedrake, "Non-Euclidean motion planning with graphs of geodesically-convex sets," in *Proc. Robot.: Sci. Syst.*, 2023, pp. 1–14.
[41] D. von Wrangel and R. Tedrake, "Using graphs of convex sets to guide nonconvex trajectory optimization," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2024, pp. 9863–9870.
[42] S. Morozov et al., "Multi-query shortest-path problem in graphs of convex sets," 2024, *arXiv:2409.19543*.
[43] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function," *Neural Netw.*, vol. 6, no. 6, pp. 861–867, 1993.
[44] R. Arora, A. Basu, P. Mianjy, and A. Mukherjee, "Understanding deep neural networks with rectified linear units," 2016, *arXiv:1611.01491*.
[45] Y. Nakamura and H. Hanafusa, "Optimal redundancy control of robot manipulators," *Int. J. Robot. Res.*, vol. 6, no. 1, pp. 32–42, 1987.
[46] D. Devaurs, T. Siméon, and J. Cortés, "Optimal path planning in complex cost spaces with sampling-based algorithms," *IEEE Trans. Automat. Sci. Eng.*, vol. 13, no. 2, pp. 415–424, Apr. 2016.
[47] A. Mitsos, B. Chachuat, and P. I. Barton, "McCormick-based relaxations of algorithms," *SIAM J. Optim.*, vol. 20, no. 2, pp. 573–601, 2009.
[48] M. Acikgoz and S. Araci, "A study on the integral of the product of several type Bernstein polynomials," *IST Trans. Appl. Math.-Modelling Simul.*, vol. 1, no. 1, pp. 10–14, 2010.
[49] Q. Huangfu and J. J. Hall, "Parallelizing the dual revised simplex method," *Math. Program. Computation*, vol. 10, no. 1, pp. 119–142, 2018.
[50] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quart. Appl. Math.*, vol. 27, no. 4, pp. 526–530, 1970.
[51] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, no. 2, pp. 652–673, 1998.
[52] S. Diamond and S. Boyd, "CVXPY: A python-embedded modeling language for convex optimization," *J. Mach. Learn. Res.*, vol. 17, no. 1, pp. 2909–2913, 2016.
[53] M. ApS, "Mosek optimizer API for python," *Version*, vol. 9, no. 17, pp. 6–4, 2022.
[54] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), Tech. Rep., 2008.
[55] I. A. Şucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robot. Automat. Mag.*, vol. 19, no. 4, pp. 72–82, Dec. 2012. [Online]. Available: https://ompl.kavrakilab.org
[56] K. M. Ben-Gharbia, A. A. Maciejewski, and R. G. Roberts, "Kinematic design of redundant robotic manipulators for spatial positioning that are optimally fault tolerant," *IEEE Trans. Robot.*, vol. 29, no. 5, pp. 1300–1307, Oct. 2013.
[57] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," 2016.
[58] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*. Hoboken, NJ, USA: Wiley, 2020.
[59] 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 *Proc. IEEE Int. Conf. Robot. Automat.*, 2024, pp. 10359–10365.
[60] G. Jocher and J. Qiu, "Ultralytics yolo11," 2024. [Online]. Available: https://github.com/ultralytics/ultralytics

Charles L. Clark (Student Member, IEEE) received the B.Sc. degree in computer science from Xavier University, Cincinnati, Ohio, USA, in 2020. He is currently working toward the Ph.D. degree in electrical engineering with IRA Lab, University of Kentucky, Lexington, KY, USA.

He is currently a Research Assistant with IRA Lab, University of Kentucky. His research interests include machine learning, redundant robots, fault-tolerant robots, and robot motion planning.



Biyun Xie (Senior Member, IEEE) received the B.S. and the Ph.D. degrees in mechanical engineering from the Beijing University of Technology, Beijing, China, in 2009 and 2015, respectively, and the Ph.D. degree in electrical engineering from Colorado State University, Fort Collins, CO, USA, in 2019.

She is currently an Assistant Professor of Electrical and Computer Engineering with the University of Kentucky, Lexington, KY, USA. She has authored and coauthored over 30 papers.

Dr. Xie serves as the Associate Editor for IEEE/RSJ International Conference on Intelligent Robots and Systems, International Conference on Robotics and Automation, and IEEE TRANSACTIONS ON ROBOTICS. She also serves as the Associate Vice President of the IEEE Robotics and Automation Society (RAS) Technical Activities Board and Co-Chair of the IEEE RAS TC on Algorithms for Planning and Control of Robot Motion.