Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling (ICAPS 2017)

Efficient Motion Planning for Problems Lacking Optimal Substructure

Oren Salzman, Brian Hou, Siddhartha Srinivasa
The Robotics Institute Carnegie Mellon University Pittsburgh, PA [*][†]

Abstract

We consider the motion-planning problem of planning a collision-free path of a robot in the presence of risk zones. The robot is allowed to travel in these zones but is penalized in a super-linear fashion for consecutive accumulative time spent there. We suggest a natural cost function that balances path length and risk-exposure time. Specifically, we consider the discrete setting where we are given a graph, or a roadmap, and we wish to compute the minimal-cost path under this cost function. Interestingly, paths defined using our cost function do not have an optimal substructure. Namely, subpaths of an optimal path are not necessarily optimal. Thus, the Bellman condition is not satisfied and standard graph-search algorithms such as Dijkstra cannot be used. We present a path-finding algorithm, which can be seen as a natural generalization of Dijkstra's algorithm. Our algorithm runs in $O\left((n_B \cdot n)\log(n_B \cdot n) + n_B \cdot m\right)$ time, where n and m are the number of vertices and edges of the graph, respectively, and n_B is the number of intersections between edges and the boundary of the risk zone. We present simulations on robotic platforms demonstrating both the natural paths produced by our cost function and the computational efficiency of our algorithm.

Introduction

In this paper, we explore motion-planning problems where an agent has to compute the least-cost path to navigate through *risk zones* while avoiding obstacles. Travelling these regions incurs a penalty which is *super-linear* in the traversal time. We call the class of problems *Risk Aware Motion Planning (RAMP)* and define a natural cost function which simultaneously optimizes for paths that are both short and reduce consecutive exposure time in the risk zone.

We are motivated by real-world problems involving *risk*, where continuous exposure is much worse than intermittent exposure. Examples include pursuit-evasion where sneaking in and out of cover is the preferred strategy, and visibility planning where the agent must ensure that an observer or operator is minimally occluded.

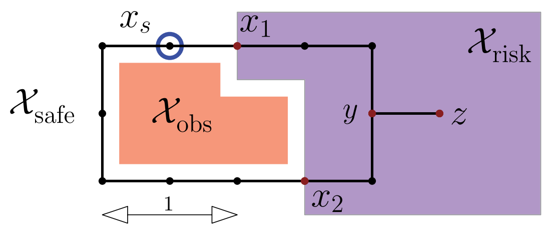

x_s
x_1
$\mathcal{X}_{\text{risk}}$
$\mathcal{X}_{\text{safe}}$
$\mathcal{X}_{\text{obs}}$
y
z
x_2
1

Figure 1: Example for which optimal paths computed using our cost function (Eq. 1) do not have an optimal substructure. Distance between two consecutive grid points is 0.5. The minimal-cost path from x_s to y passes through x_1 and has a cost of $0.5 + (e^{1.5} - 1) \approx 3.98$ while the path passing through x_2 has a cost of $3 + (e^{1} - 1) \approx 4.78$. In contrast, the minimal-cost path to z does path through x_2 and has a cost of $3 + (e^{1.5} - 1) \approx 6.48$ while the path passing through x_1 has a cost of $0.5 + (e^{2} - 1) \approx 6.88$.

Interestingly this setting (where continuous exposure is much worse than intermittent exposure) is not limited to planning: consider a large number of processes running in a shared-memory environment. Here, we would like to *quantify* the cost of running each process. When a process writes to the shared memory, it is required to use a mutex mechanism, possibly blocking other processes. Clearly, as the blocking-time increases, the probability that other processes have to stay idle, increases. Thus, as in the previous examples, the cost of running the process is its duration with a super-linear penalty proportional to accumulative consecutive times where the process writes to the shared memory.

Although practically useful, our cost function for RAMP suffers from one fundamental algorithmic challenge: *optimal plans do not posses optimal substructure*.

We explain with an example and an analogy. Consider Fig. 1, where an agent, traversing a graph, starts at x_s and must reach z while avoiding $\mathcal{X}_{\text{obs}}$. The cost of traversing through $\mathcal{X}_{\text{safe}}$ is linear in the distance travelled while the cost of traversing $\mathcal{X}_{\text{risk}}$ is super-linear in the distance travelled (formalized in Sec.).

Now consider two snails (an homage to Pohl (Pohl 1969)) taking two different paths racing to z: snail A passes through x_1 and y, and snail B through x_2 and y. If there were no risk zones, and the snails move at constant speed,

[*]This work was (partially) funded by the National Science Foundation IIS (#1409003), Toyota Motor Engineering & Manufacturing (TEMA), and the Office of Naval Research.

[†]{osalzman, bhou1,ss5}@andrew.cmu.edu.

the optimal substructure implies that when B reaches y and notices that A has passed through it already (snails leave personalized slime trails) it has no hope of catching up and should give up. In other words, the first snail that reaches z is *also* the first snail that reaches every intermediate point along the optimal path to z. Optimal substructure is critical for search algorithms (Dijkstra 1959) to efficiently track all "promising" snails (Sec.).

Unfortunately, our cost function for RAMP does *not* posses optimal substructure. Imagine that snails passing through a risk region accumulate muck on their foot. The more time a snail spends in a risk region, the more muck it accumulates, and the *slower* it gets. Now, although B reaches y later than A, it has less muck on it, having spent less time in the risk zone than A, and actually *catches up and overtakes* A to reach z first.

We define a *boundary point* to be at the boundary of a risk-free and a risk region (like x_1 and x_2 in our example) and let n_B denote the number of boundary points in our graph.

We can now address RAMP via two key insights:

1. Any optimal path from a risk-free start to a risk-free goal that passes through a risk region can be decomposed into (i) optimally reaching a boundary point, (ii) optimally traversing the risk region, and (iii) optimally reaching the goal from the exit (of the risk region). As a consequence, if we store $O(n_B^2)$ pairwise optimal paths (one for each pair of boundary points), we could create an augmented graph which can be used to directly solve RAMP.
2. All snails that pass through one specific boundary point satisfy the optimal substructure condition. In other words, the only way a snail can overtake another is if it enters the risk region via a different boundary point. As a consequence, we *only* need to track $O(n_B)$ extra snails.

Fuelled by these insights, we propose two new fundamentally different algorithms (Sec.) that solve RAMP efficiently by augmenting different data structures used by Dijkstra's algorithm:

Our *precomputation-based algorithm* creates an augmented graph with boundary vertices and their precomputed optimal paths added to the original graph, incurring a time complexity of $O\left(n_B n \cdot \log n + n_B^2 \cdot \log n + n_B m\right)$. Here n and m are the number of vertices and edges on the graph, respectively.

Our *incremental algorithm* is a strict generalization of Dijkstra's algorithm to account for risk regions. It maintains an augmented priority queue to account for the $O(n_B)$ boundary points, incurring a time complexity of $O\left(n_B n \cdot \log n + n_B m\right)$.

Interestingly the (asymptotic) running time of both algorithms is identical unless the $n_B^2 \log n$ component of the precomputation dominates the running time. This happens when $n_B \log n = \omega(m)$. Intuitively, the precomputation algorithm slows down if there are many boundary *vertices*, whereas the incremental algorithm slows down if there are many risk-zone *edges*.

We evaluate our algorithms in Sec. . Our experiments show that our cost function naturally balances path length and risk-exposure times. Furthermore we demonstrate the advantage of our incremental algorithm over the precomputation-based approach. Finally, we discuss future work in Sec. .

Related work

The RAMP problem lies on the intersection between several disciplines which we will briefly review. In its general form, our problem can be seen as an instance of the *motion-planning problem* (LaValle 2006; Choset et al. 2005). Indeed, in this work we follow the *sampling-based paradigm* (Sec.). This calls for computing a discrete graph which is then traversed by a *path-finding* algorithm (Sec.).

Computing minimal-risk paths in high-dimensional spaces is related to motion-planning under *risk constraints* (Sec.). When the search domain is two dimensional, alternative, more efficient approaches exist (Sec.).

Sampling-based motion planning

The basic motion-planning problem calls for moving a robot R in a workspace cluttered with static obstacles from a start position to a target one while avoiding obstacles and minimizing some cost function. Typically, R is abstracted as a point in a *configuration space* $\mathcal{X}$ which is subdivided into free and forbidden regions (Lozano-Pérez 1983). The problem then reduces to computing a minimal-cost collision-free path for a point in $\mathcal{X}$.

For high-dimensional problems, even computing a path, let alone an optimal one, becomes computationally hard (Reif 1979). Thus, a common approach is to approximate $\mathcal{X}$ using a graph, or a roadmap, G. One such example is the Probabilistic Roadmap Planner or PRM (Kavraki et al. 1996). Here vertices of G are points sampled in $\mathcal{X}$ and two "close-by" vertices are connected by an edge if the straight line connecting the two does not intersect obstacles in $\mathcal{X}$. A query is then answered by running a shortest-path algorithm on G. Under certain assumptions, the cost of solutions obtained by this algorithm converge to the cost of the optimal solution as the number of samples grow (Karaman and Frazzoli 2011; Solovey, Salzman, and Halperin 2016).

Path planning

Planning a shortest path on a given graph is a well studied problem. Given a graph with a cost function on its edges, the shortest-path problem asks for finding a path of minimum cost between two given vertices. When the cost function has an optimal substructure, efficient algorithms such as Dijkstra (Dijkstra 1959), A* (Hart, Nilsson, and Raphael 1968) and their many variants can be used.

In certain applications, including our setting, this is not the case. For example in (Tsaggouris and Zaroliagis 2004) every edge is associated with two attributes, say cost and resource, and there is a non-linear objective function which is convex and non-decreasing

Our problem is also closely related to multi-objective path planning. Here, we are given a set of cost functions and we are interested in finding a set of paths that captures the trade-off (the so-called Pareto curve) among the several objectives.

In general, the number of efficient solutions may be exponential in the problem size (Ehrgott and Gandibleux 2000). However, a fully polynomial-time approximation scheme (FPTAS) can be found, even for the case where the cost functions are non additive (Tsaggouris and Zaroliagis 2006). For recent results on multi-objective and multi-constrained non-additive shortest path problems, see e.g. (Reinhardt and Pisinger 2011) and references within.

Planning under risk constraints

Planning under risk constraints has been studied in several motion-planning settings. A common approach to formulate this problem is to assign some risk values to regions. Paths are considered by the planning algorithm only if the risk obtained in different regions are below some predefined thresholds (Ono, Williams, and Blackmore 2013; Ono et al. 2015; Chow et al. 2015).

Planning under risk constraints was also considered for the specific problem of Autonomous Underwater Vehicle (AUV). Here the cost function used was the sum of risk values at waypoints along a given path and the domain was two-dimensional (Pereira et al. 2013).

Planning under risk constraints is closely related to the problem of planning under uncertainty. A common approach is to minimize the estimated collision probability of a path (Liu and Ang 2014; Sun et al. 2016). Additional constraints are often added such as requiring some notion of smoothness (Müller and Sukhatme 2014).

Planning in low dimensions

When planning occurs in low dimensions, efficient algorithms exist (see, e.g., (Halperin, Salzman, and Sharir 2016) for a survey). We briefly mention several variants which are related to the problem we consider.

One such example is computing minimal-cost paths for weighted planar regions (Mitchell and Papadimitriou 1991). Here, a planar space is subdivided into different regions, where each region is assigned a positive weight. The length of a path is defined as the weighted sum of (Euclidean) lengths of the subpaths within each region. The problem is conjectured to be computationally hard, thus the focus of the community has been on approximate algorithms. For an overview of recent results, as well as a generalization of the problem, see (Sun and Reif 2007; Jaklin, Tibboel, and Geraerts 2014).

Another example is computing paths which are simultaneously short and stay away from the obstacles (Wein, van den Berg, and Halperin 2008). It is not clear if this problem is NP-Hard, however an FPTAS for this problem is known (Agarwal, Fox, and Salzman 2016).

Problem formulation

Let $\mathcal{X}$ denote the d-dimensional configuration space, $\mathcal{X}_{\text{free}}$ the collision-free portion of $\mathcal{X}$ and $\mathcal{X}_{\text{obs}} = \mathcal{X} \setminus \mathcal{X}_{\text{free}}$. Let $\mathcal{X}_{\text{risk}} \subset \mathcal{X}_{\text{free}}$ and $\mathcal{X}_{\text{safe}} = \mathcal{X}_{\text{free}} \setminus \mathcal{X}_{\text{risk}}$ denote the risk and risk-free zones, respectively. We assume that $\mathcal{X}_{\text{risk}}$ and $\mathcal{X}_{\text{free}}$ are open and closed sets, respectively. See Fig. 2(a).

A trajectory $\gamma : [0, T_\gamma] \to \mathcal{X}_{\text{free}}$ is a continuous mapping between time and configurations. The image of a trajectory is called a path. By a slight abuse of notation we refer to $\gamma[t', t'']$ as the sub-path connecting $\gamma(t')$ and $\gamma(t'')$ for $0 \leq t' \leq t'' \leq T_\gamma$. Finally, we assume that both endpoints of the path lie in the risk-free zone. Namely, $\gamma(0), \gamma(T_\gamma) \in \mathcal{X}_{\text{safe}}$.

Given a trajectory γ, and some time $t \in [0, T_\gamma]$, let $t' \leq t$ be the latest time such that $\gamma(t') \in \mathcal{X}_{\text{safe}}$. Notice that if $\gamma(t) \in \mathcal{X}_{\text{safe}}$ then $t' = t$. We define the *current exposure time* of γ at t as $\lambda_\gamma(t) = t - t'$. Namely, if $\gamma(t) \in \mathcal{X}_{\text{risk}}$ then $\lambda_\gamma(t)$ is the time passed since γ last entered $\mathcal{X}_{\text{risk}}$. If $\gamma(t) \in \mathcal{X}_{\text{safe}}$ then $\lambda_\gamma(t) = 0$.

We are now ready to define our cost function. Let γ be a trajectory and $f(x)$ any function such that $f(x) = \omega(x)$ and $f(0) = 1$. The cost of γ, denoted by $c_f(\gamma)$ is defined as

$$c_f(\gamma) = \int_{t\in[0,T_\gamma]} f(\lambda_\gamma(t))|\dot{\gamma(t)}|dt. \tag{1}$$

Eq. 1 penalizes continuous exposure to risk in a super-linear fashion (hence the requirement that $f(x) = \omega(x)$). As $f(0) = 1$, the cost of traversing the risk-free zone is simply path length. See Fig. 3 for a conceptual visualization of the current exposure time and our cost function.

Equipped with our cost function we can formally state the risk-aware motion-planning problem:

P1 *Risk-aware motion-planning problem (RAMP)* Given the tuple $(\mathcal{X}_{\text{safe}}, \mathcal{X}_{\text{risk}}, \mathcal{X}_{\text{obs}}, x_{\text{s}}, x_{\text{g}}, f)$, where $x_{\text{s}}, x_{\text{g}} \in \mathcal{X}_{\text{safe}}$ are start and target configurations, compute $\underset{\gamma\in\Gamma}{\arg\min}\; c_f(\gamma)$ with Γ the set of all collision-free trajectories connecting x_{s} and x_{g}

As mentioned, RAMP is a generalization of the motion-planning problem (where there are no risk zones) which is known to be PSPACE-Hard (Reif 1979). Thus, in this paper we concentrate on the discrete version of the problem:

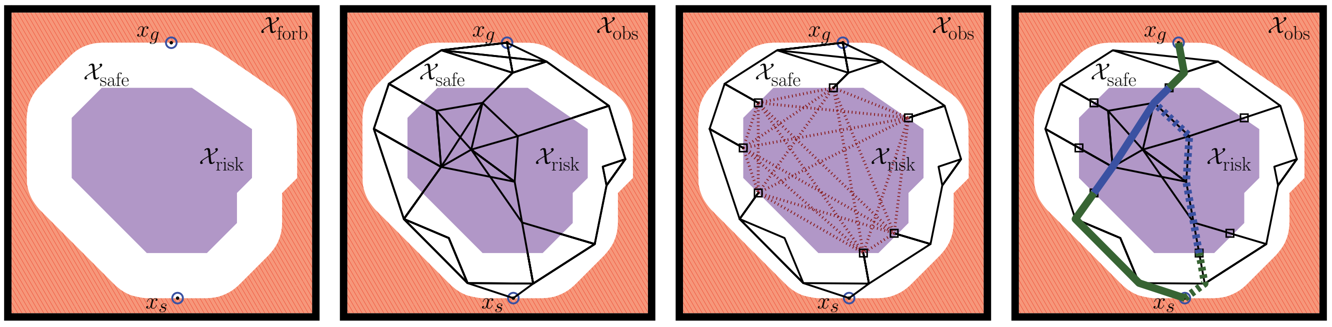
P2 *discrete Risk-aware motion-planning problem (dRAMP)* Given the tuple $(\mathcal{X}_{\text{safe}}, \mathcal{X}_{\text{risk}}, \mathcal{X}_{\text{obs}}, x_{\text{s}}, x_{\text{g}}, G, f)$, where $G = (V, E)$ is a roadmap embedded in the C-space such that $x_{\text{s}}, x_{\text{g}} \in V$, compute $\underset{\gamma\in\Gamma_G}{\arg\min}\; c_f(\gamma)$ with Γ_G the set of all collision-free paths[1] in G connecting x_{s} and x_{g}. See Fig. 2(b).

To simplify the discussion, in the rest of this paper we assume that the robot is moving in constant speed and we use $f(x) = e^x$. Thus, we can re-write Eq. 1 as

$$c(\gamma) = \int_{t\in[0,T_\gamma]} e^{\lambda_\gamma(t)}dt. \tag{2}$$

Using the assumption that the robot is moving in constant speed, we will use the terms duration of a trajectory and path length interchangeably (here we measure path length as the Euclidean distance). Further exploiting this assumption and by a slight abuse of notation we will also use Eq. 2 to define the cost of a path (and not of a trajectory).

[1]Note that we use paths to define curves in $\mathcal{X}$ and as sequence of edges in a graph. These definitions coincide as the graph is embedded in $\mathcal{X}$.



x_g
$\mathcal{X}_{\text{forb}}$
$\mathcal{X}_{\text{safe}}$
$\mathcal{X}_{\text{risk}}$
x_s
(a)
x_g
$\mathcal{X}_{\text{obs}}$
$\mathcal{X}_{\text{safe}}$
$\mathcal{X}_{\text{risk}}$
x_s
(b)
x_g
$\mathcal{X}_{\text{obs}}$
$\mathcal{X}_{\text{safe}}$
$\mathcal{X}_{\text{risk}}$
x_s
(c)
x_g
$\mathcal{X}_{\text{obs}}$
$\mathcal{X}_{\text{safe}}$
$\mathcal{X}_{\text{risk}}$
x_s
(d)

Figure 2: (a) A two-dimensional space $\mathcal{X}$ consisting of obstacles (red polygons) and a risk region (purple region), defined as all points which are farther away from the obstacles than a predefined distance (see Sec. for a motivation regarding this scenario). (b) Probabilistic roadmap G sampled in $\mathcal{X}$. (c) The roadmap G' built by taking G, adding all border points (hollow squares) as vertices and replacing all existing edges in $\mathcal{X}_{\text{risk}}$ with an edge between every pair of border points (red polylines) representing the cost of travelling between the two in $G \cap \mathcal{X}_{\text{risk}}$. (d) Minimal-cost path (green and blue for edges in $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$, respectively). Notice that this is *not* the shortest path, which has high exposure to $\mathcal{X}_{\text{risk}}$ (depicted in dashed green and dashed blue).

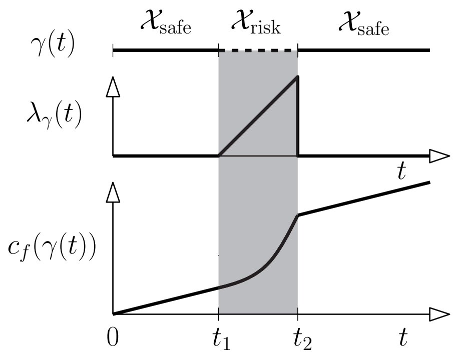


Figure 3: Relation between a trajectory $\gamma(t)$ (top), recent exposure time $\lambda_\gamma(t)$ (middle) and cost $c_f(\gamma(t))$ (bottom) as a function of time. In $t \in [0, t_1]$, γ stays in $\mathcal{X}_{\text{safe}}$, hence $\lambda_\gamma(t) = 0$ and the cost grows linearly with time. At $t = t_1$, γ enters $\mathcal{X}_{\text{risk}}$, $\lambda_\gamma(t)$ grows linearly and the cost grows super-linearly. At $t = t_2$, γ leaves $\mathcal{X}_{\text{risk}}$, $\lambda_\gamma(t) = 0$ and the cost returns to growing linearly.

Preliminaries

In this section we review Dijkstra's path-finding algorithm (using a min-priority queue) which relies on the cost function to have an optimal substructure. We then continue to discuss our cost function—why it does not have an optimal substructure and what properties it does have.

Dijkstra's shortest-path algorithm

Dijkstra's algorithm computes the minimal-cost path between a given start vertex x_s and all other vertices in a graph. Returning to our snails, the algorithm can be intuitively described as follows: a snail starts at x_s moving at constant speed. Every time it reaches a vertex, it splits into multiple snails, one for each outgoing edge. If a snail reaches a vertex which was already reached by another snail (identified by the existing trail of slime), it retracts into its shell and stops moving. Clearly, the distance travelled by the first snail to reach any vertex u is the minimal distance to reach u.

The problem with the aforementioned process is that it is continuous. Dijkstra's algorithm uses discrete times where each time-step represents the event that a snail reaches a specific vertex. The key difference is that here only one snail moves at a time and his movement spans the entire length of an edge. Processing an event is similar to the continuous version: if a snail reaches a vertex which was already reached by another snail, it retracts into it's shell and stops its progress. Otherwise, it splits into multiple snails, one for each outgoing edge, and the time the snail is intended to reach the edge's endpoint is computed. This is registered as a new event.

The implementation of the aforementioned process is via a min-cost priority queue $\mathcal{Q}$. Each entry $\tau = (u, c, p)$ in the queue represents the time, or cost, c that a snail is to reach the vertex u through the parent p. We emphasize that for each vertex u, only the *current best* path (or snail) is maintained, together with its cost. Initially, this value is only known for x_s. For every other vertex this value is unknown, thus the event is initialized to have infinite cost. After all such events are inserted into the queue, the minimal cost entry (u, c, p) is removed and if it is the goal, the process terminates. If not, then for every neighboring edge (u, v) that is collision free, the cost to reach v via u is computed. If it represents a shorter path to reach v, than v's current entry, v's entry together with its location in the priority queue are updated.

Properties of our cost function

Recall that our cost does not have an optimal substructure (see Sec. and Fig. 1). This implies that, for any vertex u, we need to consider not only the optimal path to reach u, but also all paths that pass through u and may be part of a minimal-cost path to reach some future vertex v (that pass through u). To do so, we must characterize this set of paths. We start by noting the following properties of Eq. 2:

Observation 1 *Let γ be a trajectory that lies completely within $\mathcal{X}_{\text{safe}}$, then $\forall t\ \lambda_\gamma(t) = 0$ and the cost of the trajectory is simply its duration T_γ.*

Observation 2 *Let γ be a trajectory that lies completely within $\mathcal{X}_{\text{risk}}$, then $\forall t\ \lambda_\gamma(t) = t$ and the cost of the trajectory is $e^{T_\gamma} - 1$.*

Using the fact that Obs. 1 and 2 hold for any maximally connected subpath in $\mathcal{X}_{\text{safe}}$ or $\mathcal{X}_{\text{risk}}$ we can rewrite Eq. 2. Namely, let $t_0 = 0 < t_1 < \ldots < t_n = T_\gamma$ such that $\forall i \geq 0$, $\gamma[t_{2i}, t_{2i+1}] \subset \mathcal{X}_{\text{safe}}$ and $\gamma[t_{2i+1}, t_{2i+2}] \subset \mathcal{X}_{\text{risk}}$ then[2]

$$c(\gamma) = \sum_i (t_{2i+1} - t_{2i}) + \sum_i \left(e^{t_{2i+2} - t_{2i+1}} - 1\right). \quad (3)$$

We characterize the set of paths that our algorithm will have to consider using the notion of *domination*. Given two trajectories γ_1, γ_2 that start at x_s and end in some vertex u, we say that γ_1 *dominates* γ_2 if it will always be more beneficial to use γ_1 and not γ_2 to reach some future vertex v. Namely, γ_1 dominates γ_2 if $c(\gamma_1) \leq c(\gamma_2)$ and $\lambda_{\gamma_1}(T_{\gamma_1}) \leq \lambda_{\gamma_2}(T_{\gamma_2})$. The set of all trajectories Γ_u that start at x_s and end in u where no trajectory dominates any other trajectory is said to be a *useful* set of trajectories.

Understanding path domination and the maximal size of a useful set of trajectories will be key in understanding our algorithms and bounding their running time. We note several properties of such trajectories:

Lemma 1 *Let $u \in \mathcal{X}_{\text{safe}}$ with γ_u the minimal-cost path to reach u from x_s. Then γ_u dominates any other trajectory γ'_u that reaches u.*

Proof: Since $u \in \mathcal{X}_{\text{safe}}$ we have that $\lambda_{\gamma_u}(T_{\gamma_u}) = \lambda_{\gamma'_u}(T_{\gamma'_u}) = 0$. Furthermore, γ_u is a minimal-cost path, thus $c(\gamma_u) \leq c(\gamma'_u)$. Hence, by definition, γ_u dominates γ'_u. $\square$

Lemma 2 *Let $u \in \mathcal{X}_{\text{risk}}$ with γ_u the minimal-cost path to reach u from x_s. Let $\phi(u)$ be the point on the boundary of $\mathcal{X}_{\text{safe}}$ through which γ_u last entered $\mathcal{X}_{\text{risk}}$. Then, γ_u dominates any trajectory γ'_u that reaches u with $\phi(u)$ as the last point on the boundary of $\mathcal{X}_{\text{safe}}$ through which γ'_u last entered $\mathcal{X}_{\text{risk}}$.*

Proof: Let t_1 and t'_1 be the times that γ_u and γ'_u reach $\phi(u)$, respectively. If $c(\gamma_u[0, t_1]) > c(\gamma'_u[0, t'_1])$, by Eq. 3, we can replace subpath $\gamma_u[0, t_1]$ with $\gamma'_u[0, t'_1]$ in γ_u and reduce its cost which contradicts γ_u being a minimal-cost path. The same argument holds for subpaths $\gamma_u[t_1, T_{\gamma_u}]$ and $\gamma'_u[t'_1, T_{\gamma'_u}]$. Thus, we have that $\lambda_{\gamma_u}(T_{\gamma_u}) \leq \lambda_{\gamma'_u}(T_{\gamma'_u})$ and γ_u dominates γ'_u. $\square$

[2]There is a slight inaccuracy here as $\mathcal{X}_{\text{risk}}$ is an open set and thus $\gamma[t_{2i+1}, t_{2i+2}]$ cannot be fully contained in $\mathcal{X}_{\text{risk}}$. However this inaccuracy does not change our results and is intentionally used for ease of exposition.

Minimal-cost path-finding algorithm

In this section we address problem P2, namely how to efficiently compute a minimal-cost path between two vertices in a given roadmap. As we have seen, in Dijkstra's algorithms (and its many variants), each vertex u can have only one useful path which will dominate all other paths to reach u. Lemma 1 and 2 imply that in our setting this holds for vertices in $\mathcal{X}_{\text{safe}}$ but for vertices in $\mathcal{X}_{\text{risk}}$, the number of useful paths may be as large as the number of edges entering $\mathcal{X}_{\text{risk}}$. This gives rise to two different algorithmic approaches:

The first approach (Sec.) is to directly use Eq. 3 by splitting the graph into subgraphs that are fully contained in $\mathcal{X}_{\text{safe}}$ and subgraphs that are fully contained in $\mathcal{X}_{\text{risk}}$. We then precompute the graph distance between all points that lie on the border of $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$ (we call these *border points*) restricted to moving only in $\mathcal{X}_{\text{risk}}$. Using these distances allows us to define a new graph, which has an edge between every pair of border points with weights assigned using the precomputed distances. We can then run any shortest-path algorithm on the new graph without having to consider the multiple useful paths of vertices in $\mathcal{X}_{\text{risk}}$. This is analogous to having our snails roam the original graph and considering a traversal of $\mathcal{X}_{\text{risk}}$ as one discrete event.

The first algorithm (Sec.), which requires preprocessing the entire graph, can be seen as a warm-up for our efficient path-finding algorithm (Sec.). This algorithm essentially runs a Dijkstra-type search without any preprocessing. To do so, for vertices within $\mathcal{X}_{\text{risk}}$, it efficiently maintains all useful paths. Here we can envision our snails entering $\mathcal{X}_{\text{risk}}$ as in Dijkstra's algorithm. However, when one snail reaches a vertex already traversed by another snail, it only retracts into its shell if the other snail dominates it.

Both algorithms have to distinguish between $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$. Thus, we start by defining the *refined roadmap* (Sec.) and then continue to detail each algorithm.

Refined roadmap

An edge e is said to be a *border edge* if it straddles $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$. Namely, if $e \cap \mathcal{X}_{\text{safe}} \neq \emptyset$ and $e \cap \mathcal{X}_{\text{risk}} \neq \emptyset$. For simplicity of exposition, we assume that every border edge of G intersects the boundary of $\mathcal{X}_{\text{risk}}$ exactly once. We denote this point $\phi(e)$ and call it a *border point*. Set $E_{\text{border}} \subseteq E$ to be the set of all border edges and $V_{\text{border}} = \bigcup_{e \in E_{\text{border}}} \phi(e)$ to be all the border points.

Given a roadmap $G = (V, E)$, define the *refined roadmap* $\tilde{G} = (\tilde{V}, \tilde{E})$ such that $\tilde{V} = V \cup V_{\text{border}}$ and $\tilde{E} = (E \setminus E_{\text{border}}) \bigcup \{(u, \phi(e)), (\phi(e), v) \mid (u, v) \in E_{\text{border}}\}$. Namely, the refined roadmap is the roadmap defined by adding all border points to the original set of vertices and subdividing border edges accordingly.

Minimal-cost planning via precomputinon

To compute the shortest path in G, we start by constructing $\tilde{G}$. For each border point, we run Dijkstra's algorithm restricted to $\mathcal{X}_{\text{risk}}$. This gives us a mapping $\mathcal{T} : V_{\text{border}} \times V_{\text{border}} \to \mathbb{R}_{\geq 0}$ that denotes shortest *distances*, or traversal times, of paths that stay strictly in $\mathcal{X}_{\text{risk}}$ (if no such path exists, then the mapping returns ∞). Thus, the cost of the shortest path between two border points u, v that stays strictly in $\mathcal{X}_{\text{risk}}$ is $e^{\mathcal{T}(u,v)} - 1$. Now, construct the graph $G' = (V', E')$ where $V' = (V \cap \mathcal{X}_{\text{safe}}) \bigcup V_{\text{border}}$, and $E' = (E \cap \mathcal{X}_{\text{safe}}) \bigcup \{(u, \phi(e)) \mid (u,v) \in E_{\text{border}}\} \bigcup \{(u,v) \mid u, v \in V_{\text{border}} \text{ and } \mathcal{T}(u,v) < \infty\}$. Namely, V' consists of all vertices that are risk free or are border points. E' contains three types of edges: (i) all original edges that are risk free, (ii) edges from $\tilde{G}$ that start at a risk-free vertex and end at a border point, and (iii) new edges connecting each pair of border points within the same risk zone. The weights of edges in G' are simply the weights of the edges in $\tilde{G}$ if they are of the first two types or $e^{\mathcal{T}(u,v)} - 1$ if the edge (u, v) is of the third type. See Fig. 2(c). Finally, we run any shortest-path algorithm between x_s and x_t in G'.

This algorithm requires preprocessing the entire graph and computing distances between all pairs of border points. As we will see, this may incur unnecessary computations. We continue with a Dijkstra-type algorithm that computes paths in a just-in-time manner.

Minimal-cost planning via incremental search

Recall that Dijkstra's algorithm makes use of a priority queue $\mathcal{Q}$ with entries of the form (u, c, p). Our algorithm will also store entries in $\mathcal{Q}$ which we call *Risk-Aware Shortest Paths* entries, or RASP entries. Each such RASP entry represents a dominating path. Following Lemma 1, for each vertex $u \in \mathcal{X}_{\text{safe}}$ we need one such entry. Following Lemma 2, for each vertex $u \in \mathcal{X}_{\text{risk}}$ we need at most one entry for each border point of u's risk region.

Specifically, each entry $\tau = (u, c, t, \lambda, p, \phi)$ will represent a dominating path, or trajectory γ_τ, to reach a vertex $u[\tau] = u$ (implicitly defined by the parent pointer $p[\tau] = p$), its cost and duration (stored as $c[\tau] = c$ and $t[\tau] = t$, respectively), its current exposure time at time $t[\tau]$ (stored as $\lambda[\tau] = \lambda$) and the last border point ($\phi[\tau] = \phi$) that γ_τ passed through if $u \in \mathcal{X}_{\text{risk}}$ (NIL if $u \in \mathcal{X}_{\text{safe}}$).

The algorithm (Alg. 1) starts by initializing RASP entries (lines 1-4) and inserting them into the min-priority queue $\mathcal{Q}$ (lines 5-6). Entries in $\mathcal{Q}$ are ordered according to their cost. We iteratively pop the min-cost entry τ from $\mathcal{Q}$ and set $u = u[\tau]$ to be the vertex associated with the entry (line 8). If it is the goal vertex, then a minimal-cost path has been found and the algorithm terminates (lines 9-10). If not, we consider each of its neighbors v, and if the edge connecting the two is collision free we expand the trajectory γ_τ to v (line 12). This trajectory $\gamma_{\tau_{\text{tmp}}}$ is represented by a temporary RASP entry τ_{tmp}. As in Dijkstra's algorithm, we check if it improves the current-best path to reach v. Here we restrict ourselves to paths that enter $\mathcal{X}_{\text{risk}}$ through a specific border point $\phi[\tau_{\text{tmp}}]$. If this is the case (line 13-15), we update the relevant RASP entry and decrease it's cost in $\mathcal{Q}$ (lines 14-15)

We now detail the `expand` operation (line 12). Specifically, let τ be the entry popped from $\mathcal{Q}$ with $u = \tau[u]$ its associated vertex. Let v be its neighbor and let $\Delta_t(e)$ denote the length of an edge e. We describe the content of the new RASP entry τ_v according to whether u and v are in $\mathcal{X}_{\text{safe}}$ or $\mathcal{X}_{\text{risk}}$. See extended version of this paper (Salzman, Hou, and Srinivasa 2017) for a visualization of how the RASP lists are maintained by the algorithm.

Algorithm 1 `incremental_search` (G, x_s, x_g)

```
1: $\tau_{x_s,\text{NIL}} = (x_s, 0, 0, 0, \text{NIL}, \text{NIL})$;  $T \leftarrow \{\tau_{x_s,\text{NIL}}\}$
2: for all $v \in V \setminus \{x_s\}$ do
3:     for all $\phi \in V_{\text{border}}$ do
4:         $\tau_{v,\phi} = (v, \infty, \infty, \infty, \text{NIL}, \phi)$;  $T \leftarrow T \cup \{\tau_{v,\phi}\}$
5: for all $\tau \in T$ do
6:     $\mathcal{Q}$.add_with_priority($\tau$)

7: while $|\mathcal{Q}| > 0$ do
8:     $\tau \leftarrow \mathcal{Q}$.extract_min();  $u \leftarrow u[\tau]$
9:     if $u = x_g$ then
10:        return extract_path($\tau$)

11:    for all $v$ s.t. $(u,v) \in E$ and $(u,v) \notin \mathcal{X}_{\text{obs}}$ do
12:        $\tau_{\text{tmp}} \leftarrow$ expand($\tau, v$);  $\tau_v \leftarrow \tau_{v,\phi[\tau_{\text{tmp}}]}$
13:        if $c(\tau_{\text{tmp}}) < c[\tau_v]$ then
14:            $\tau_v \leftarrow \tau_{\text{tmp}}$
15:            $\mathcal{Q}$.decrease_priority($\tau_v$)
```

Case (i) $u \in \mathcal{X}_{\text{safe}}$ and $v \in \mathcal{X}_{\text{safe}}$: We set $\tau_v = (u, c[\tau] + \Delta_t(u,v), t[\tau] + \Delta_t(u,v), 0, \text{NIL}, \text{NIL})$.

Case (ii) $u \in \mathcal{X}_{\text{safe}}$ and $v \in \mathcal{X}_{\text{risk}}$: We compute the border point $\phi = \phi(u,v)$ and the lengths $\Delta_t(u,\phi)$ and $\Delta_t(\phi, v)$. We then compute the RASP entry which represents the path reaching v using u as its parent. This entry τ_v will have, $c[\tau_v] = c[\tau] + \Delta_t(u,\phi) + e^{\Delta_t(\phi,v)} - 1$, $t[\tau_v] = t[\tau] + \Delta_t(u,v)$, $\lambda[\tau_v] = \Delta_t(\phi, v)$ and $\phi[\tau_v] = \phi$.

Case (iii) $u \in \mathcal{X}_{\text{risk}}$ and $v \in \mathcal{X}_{\text{risk}}$: We set $c[\tau_v] = c[\tau] + e^{\lambda[\tau]} \cdot (e^{\Delta_t(u,v)} - 1)$, $t[\tau_v] = t[\tau] + \Delta_t(u,v)$, $\lambda[\tau_v] = \lambda[\tau] + \Delta_t(u,v)$ and $\phi[\tau_v] = \phi[\tau]$.

Case (iv) $u \in \mathcal{X}_{\text{risk}}$ and $v \in \mathcal{X}_{\text{safe}}$: We compute the border point $\phi(u,v)$ and the lengths $\Delta_t(u, \phi(u,v))$ and $\Delta_t(\phi(u,v), v)$. Similar to case (ii) we set $c[\tau_v] = c[\tau] + e^{\lambda[\tau]} \cdot (e^{\Delta_t(u,\phi(u,v))} - 1) + \Delta_t(\phi(u,v), v)$, $t[\tau_v] = t[\tau] + \Delta_t(u,v)$, $\lambda[\tau_v] = 0$ and $\phi[\tau_v] = \text{NIL}$.

Practical implementation Similar to our descriptions of Dijkstras algorithm (Sec.), we traded practical efficiency with ease of exposition (this has no effect on the asymptotic runtime of the algorithm). In practice, we can initialize $\mathcal{Q}$ to contain only the RASP entry associated with x_s. Other RASP entries can be created on the fly only when they are first constructed by the `expand` operation (lines 12-15).

Additionally, Lemma 2 states that a given vertex can have at most n_B useful trajectories. In practice, this number may be much smaller. Thus, before inserting a RASP entry τ to $\mathcal{Q}$, we can check if it is dominated by any other entry τ' in $\mathcal{Q}$ with $u[\tau] = u[\tau']$.

Finally, the same algorithm can be transformed into an A*-type algorithm by using a heuristic that estimates the cost-to-go and ordering $\mathcal{Q}$ according to the sum of the cost-to-come and the estimated cost-to-go.

Computational complexity

In this section we discuss the computational complexity of our search algorithms. We assume that testing if an edge is collision free and computing border points and distances take constant time. We note that while this assumption is common in search algorithms such as Dijkstra and A*, in many motion-planning applications, these operations often dominate the (practical) running time of search algorithms (LaValle 2006).

Recall that $n_B \leq m$ is the number of border points in G and that our algorithm that uses precomputation (Sec.), runs (i) Dijkstra's algorithm from every border point (restricted to vertices within within $\mathcal{X}_{\text{risk}}$) (ii) adds an edge between every two border points in the same connected component and (iii) runs a shortest-path algorithm on the new graph G'. Step (i) takes $O\left(n_B \cdot ((n + n_B)\log(n + n_B) + m)\right)$ time. We then add $O(n_B^2)$ edges to our new graph in step (ii). This implies that the number of vertices n' and edges m' of G' is $n' = O(n + n_B)$ and $m' = O(m + n_B^2)$. Thus, step (iii) takes $O(n' \log n' + m')$. To summarize, our precomputation-based algorithm takes

$$O\left(n_B n \cdot \log n + n_B^2 \cdot \log n + n_B m\right).$$

Our incremental search algorithm (Sec.) has complexity identical to Dijkstra's except that there may be at most $n \cdot n_B$ RASP entries in $\mathcal{Q}$ (and not n). Moreover, each outgoing edge of a vertex u can be expanded once for each of u's useful paths which is at most $O(n_B)$. Thus, our incremental search algorithm runs in time, $O\left((n \cdot n_B)\log(n \cdot n_B) + n_B \cdot m\right)$ which is equal to

$$O\left(n_B n \cdot \log n + n_B m\right).$$

Interestingly, the (asymptotic) running time of the two algorithms is identical unless the $n_B^2 \log n$ component dominates the running time of the first algorithm. This happens when $n_B \log n = \omega(m)$. However, in practice, in the precomputation of the first algorithm we often compute paths in $\mathcal{X}_{\text{risk}}$ that will not be used. Moreover, this requires testing in advance which edges are in $\mathcal{X}_{\text{risk}}$, which is an expensive operation in practice. This is demonstrated in Sec. .

Evaluation

In this section we visualize our cost function and demonstrate the behavior of our algorithm. All algorithms were implemented using the Open Motion Planning Library (OMPL 1.2.1) (Şucan, Moll, and Kavraki 2012) running on a 4.0-GHz Intel Core i7 processor with 16 GB of memory. Source code is publicly available at https://github.com/personalrobotics/ompl_rasp .

For each experiment, we constructed a roadmap and precomputed for each vertex and each edge whether it is collision-free and whether it is in the risk zone. This allows us to compare the time that graph operations take for each of the algorithms.

Our first set of experiments is motivated by the early Viking sailing expeditions: For centuries, sailing was done primarily by coastal navigation, where the sea vessel stayed within sight of the coast. Gradually, the art of open-seas navigation was developed, relying on more uncertain factors such as visibility to the sun, moon and the stars. Thus, we model the sea and the land as the free and forbidden regions, respectively. Any point closer (further) than a predefined distance from the shore is modelled as the safe (risk) region, respectively.

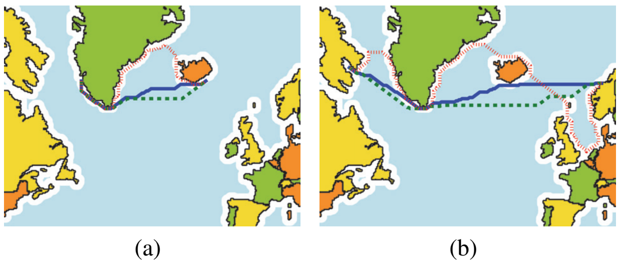


Figure 4: Navigating the seas: land masses are regarded as obstacles, regions next to the coastal line (white) and open seas (light blue) are $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$, respectively. We visualize paths produced using our cost function (solid blue), shortest paths (dashed green), and minimal-risk paths (dotted red). Figure best viewed in color.

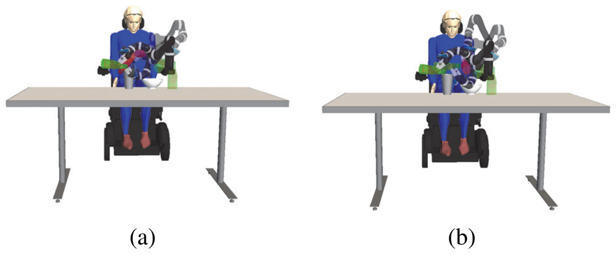


Figure 5: A disabled user moving a bottle using a robotic arm in the presence of obstacles. The trajectory of the arm moves between "safe" and "risk" regions where the bottle is visible (colored green) and non-visible (colored red) to the user, respectively. Snapshots are taken at intermediate points along the path. (a) Shortest path. (b) Minimal-cost path.

Fig. 4 depicts maps with different queries. For each query, we use a 201×201 eight-connected grid as a roadmap. We then compute the minimal-cost path computed using Eq. 2, the shortest (Euclidean) paths and the minimal-risk path that minimizes time spent in $\mathcal{X}_{\text{risk}}$. As we can see, our cost function serves as a natural interpolation between the two opposing metrics.

We present running times in Table 1. Computing minimal-cost paths results in larger computation times when compared to computing shortest paths. For our incremental-based algorithm, this is roughly a $4\times$ or $5\times$ slowdown. For our precomputation-based algorithm, this is slower by a factor of several thousands. Not surprisingly, the lion's share of the algorithm's running time is dedicated to computing shortest paths between pairs of points on the boundary of $\mathcal{X}_{\text{risk}}$.

Our second scenario is motivated by assistive robotics. Consider a robot arm performing a task such as pouring juice from a bottle, while receiving inputs from a user such as when to stop pouring. During the motion, the robot's end effector is moving between regions that are either visible or

Scenario	Dijkstra	Precomputation-based	Incremental-based
Vikings (Fig. 4(a))	0.03 ± 0.001	204.35 ± 8.73	0.11 ± 0.006
Vikings (Fig. 4(b))	0.06 ± 0.004	207.42 ± 9.97	0.31 ± 0.034
Assistive Fig. 5	0.113 ± 0.008	—	0.003 ± 0.008

Table 1: Running time (in seconds) comparing Dijkstra, computing the shortest path, with our precomputation-based algorithm (Sec.) and our incremental algorithm (Sec.) computing minimal-cost paths. For the precomputation-based algorithm, roughly 98% of the running time is spent on computing distances between pairs of boundary points on the Viking scenarios while on the assistive care scenario it terminated due to insufficient memory. Times reported are the average over 50 different runs together with one standard deviation.

occluded to the user.

Specifically, in this motion-planning problem configurations in $\mathcal{X}_{\text{risk}}$ are points occluded from the viewpoint of a user sitting in a wheelchair. We computed a Halton graph with 10,000 vertices (a typical-size roadmap in such motion-planning settings) and ran Dijkstra's algorithm as well as our path-finding algorithms. Results, depicted in Fig. 5 demonstrate how the shortest path traverses the risk zone for a long duration while minimal-cost paths enter it for a short period of time.

Timing results, reported in Table 1 show that our incremental-based algorithm is actually faster than Dijkstra's algorithm. This is because in this scene there is a large risk region with high cost and a "narrow passage" that reduces risk exposure. Our cost function naturally guides the search towards this promising region. In contrast, Dijkstra searches in cost-to-come space and exposes more vertices. Our precomputation-based algorithm, on the other hand, terminated due to insufficient memory.

Conclusions and future work

Many interesting research questions arise from our problem formulation. The first, relates to the roadmap generation: Using (Karaman and Frazzoli 2011), we can describe the necessary conditions for solutions obtained using PRM to converge to an optimal solution. Applying the same analysis to our setting is not straightforward. This is partially due to the fact that the proof used in (Karaman and Frazzoli 2011) assumes that the source and target configurations are in the roadmap. Consider an optimal path $\gamma*$ that passes in and out of risk zones. A naive attempt to use the aforementioned proof is to subdivide the path into sections that are fully contained within $\mathcal{X}_{\text{safe}}$ and fully contained within $\mathcal{X}_{\text{risk}}$ and argue that asymptotically, the roadmap will converge to each of these subpaths. However, the points where $\gamma*$ moves from $\mathcal{X}_{\text{safe}}$ to $\mathcal{X}_{\text{risk}}$ (and vice-versa) are *not* in the roadmap. We believe, that under certain assumptions on the structure of $\mathcal{X}_{\text{risk}}$ this may be done but many details should be carefully addressed.

Since our problem is single-shot, a possible approach to solve RAMP (and not dRAMP) is not to construct a roadmap but a tree, rooted at the initial configuration. Here, an RRT*-type algorithm (Karaman and Frazzoli 2011) may be used in order to asymptotically converge to the optimal solution.

Another interesting question relates to the setting where $\mathcal{X} \subset \mathbb{R}^2$, i.e., when planning is restricted to the plane. Here, we are interested in understanding what complexity class does our problem fall in. It is well known that planning for shortest paths in the plane amid polygonal obstacles can be computed in $O(n \log n)$ time, where n is the complexity of the obstacles (see (Mitchell 2016) for a survey). When computing shortest paths amid polyhedral obstacles in $\mathbb{R}^3$, or in $\mathbb{R}^2$ when there are constraints on the curvature of the path, the problem becomes NP-Hard (Canny and Reif 1987; Kirkpatrick, Kostitsyna, and Polishchuk 2011). Furthermore, the Weighted Region Shortest Path Problem, which is closely related to our problem (Sec.), is unsolvable in the Algebraic Computation Model over the Rational Numbers (De Carufel et al. 2014). If our problem is NP-Hard, as we conjecture, then a reduction, possibly along the lines of (Canny and Reif 1987) should be provided together with an approximation algorithm. Here, a possible approach would be to sample the boundary of $\mathcal{X}_{\text{risk}}$, similar to (Agarwal, Fox, and Salzman 2016).

Finally, our work assumed that the dimension of $\mathcal{X}_{\text{safe}}$ and $\mathcal{X}_{\text{risk}}$ are d, the dimension of $\mathcal{X}$. However, we envision our cost function being used in situations where this assumption does not hold. Consider compliant motion planning or fine motion (Lozano-Perez, Mason, and Taylor 1984) where a robot reduces uncertainty by making and maintaining contact with the environment. Here, we would like to penalize for *not* being in contact with an obstacle feature. Thus, $\mathcal{X}_{\text{safe}}$ induces a manifold of lower dimensionality than $\mathcal{X}$ which raises many interesting questions.

References

Agarwal, P. K.; Fox, K.; and Salzman, O. 2016. An efficient algorithm for computing high-quality paths amid polygonal obstacles. In *Symposium on Discrete Algorithms*, 1179–1192.

Canny, J. F., and Reif, J. H. 1987. New lower bound techniques for robot motion planning problems. In *Symposium on Foundations of Computer Science*, 49–60.

Choset, H.; Lynch, K. M.; Hutchinson, S.; Kantor, G.; Burgard, W.; Kavraki, L. E.; and Thrun, S. 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementation.* MIT Press.

Chow, Y.; Ghavamzadeh, M.; Janson, L.; and Pavone, M. 2015. Risk-constrained reinforcement learning with percentile risk criteria. *CoRR* abs/1512.01629.

De Carufel, J.-L.; Grimm, C.; Maheshwari, A.; Owen, M.; and Smid, M. 2014. A note on the unsolvability of the weighted region shortest path problem. *Computational Geometry* 47(7):724–727.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1(1):269–271.

Ehrgott, M., and Gandibleux, X. 2000. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR-Spektrum* 22(4):425–460.

Halperin, D.; Salzman, O.; and Sharir, M. 2016. Algorithmic motion planning. In *Handbook of Discrete and Computational Geometry, 3rd Edition.* Chapman and Hall/CRC. chapter 50. http://www.csun.edu/~ctoth/Handbook/HDCG3.html.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics* 4(2):100–107.

Jaklin, N.; Tibboel, M.; and Geraerts, R. 2014. Computing high-quality paths in weighted regions. In *Motion in Games*, 77–86.

Karaman, S., and Frazzoli, E. 2011. Sampling-based algorithms for optimal motion planning. *Int. J. Robot. Res.* 30(7):846–894.

Kavraki, L. E.; Svestka, P.; Latombe, J.-C.; and Overmars, M. H. 1996. Probabilistic roadmaps for path planning in high dimensional configuration spaces. *IEEE Trans. Rob.* 12(4):566–580.

Kirkpatrick, D. G.; Kostitsyna, I.; and Polishchuk, V. 2011. Hardness results for two-dimensional curvature-constrained motion planning. In *Canadian Conference on Computational Geometry*.

LaValle, S. M. 2006. *Planning algorithms*. Cambridge University Press.

Liu, W., and Ang, M. H. 2014. Incremental sampling-based algorithm for risk-aware planning under motion uncertainty. In *2014 IEEE International Conference on Robotics and Automation*, 2051–2058.

Lozano-Perez, T.; Mason, M. T.; and Taylor, R. H. 1984. Automatic synthesis of fine-motion strategies for robots. *Int. J. Robot. Res.* 3(1):3–24.

Lozano-Pérez, T. 1983. Spatial planning: A configuration space approach. *IEEE Trans. Computers* 32(2):108–120.

Mitchell, J. S. B., and Papadimitriou, C. H. 1991. The weighted region problem: Finding shortest paths through a weighted planar subdivision. *Journal of the ACM* 38(1):18–73.

Mitchell, J. S. B. 2016. Shortest paths and networks. In *Handbook of Discrete and Computational Geometry, 3rd Edition.* Chapman and Hall/CRC. chapter 31. http://www.csun.edu/~ctoth/Handbook/HDCG3.html.

Müller, J., and Sukhatme, G. S. 2014. Risk-aware trajectory generation with application to safe quadrotor landing. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3642–3648.

Ono, M.; Pavone, M.; Kuwata, Y.; and Balaram, J. 2015. Chance-constrained dynamic programming with application to risk-aware robotic space exploration. *Auton. Robots* 39(4):555–571.

Ono, M.; Williams, B. C.; and Blackmore, L. 2013. Probabilistic planning for continuous dynamic systems under bounded risk. *J. Artif. Intell. Res. (JAIR)* 46:511–577.

Pereira, A.; Binney, J.; Hollinger, G. A.; and Sukhatme, G. S. 2013. Risk-aware path planning for autonomous underwater vehicles using predictive ocean models. *J. Field Robotics* 30(5):741–762.

Pohl, I. 1969. *Bi-directional and heuristic search in path problems*. Ph.D. Dissertation, Dept. of Computer Science, Stanford University.

Reif, J. H. 1979. Complexity of the mover's problem and generalizations (extended abstract). In *Symposium on Foundations of Computer Science*, 421–427.

Reinhardt, L. B., and Pisinger, D. 2011. Multi-objective and multi-constrained non-additive shortest path problems. *Computers & OR* 38(3):605–616.

Salzman, O.; Hou, B.; and Srinivasa, S. S. 2017. Efficient motion planning for problems lacking optimal substructure. *CoRR* abs/1703.02582.

Solovey, K.; Salzman, O.; and Halperin, D. 2016. New perspective on sampling-based motion planning via random geometric graphs. In *Robotics: Science and Systems*.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine* 19(4):72–82. http://ompl.kavrakilab.org.

Sun, Z., and Reif, J. H. 2007. On robotic optimal path planning in polygonal regions with pseudo-euclidean metrics. *IEEE Trans. Systems, Man, and Cybernetics, Part B* 37(4):925–936.

Sun, W.; Torres, L. G.; van den Berg, J.; and Alterovitz, R. 2016. Safe motion planning for imprecise robotic manipulators by minimizing probability of collision. In Inaba, M., and Corke, P., eds., *International Symposium on Robotics Research*, 685–701.

Tsaggouris, G., and Zaroliagis, C. D. 2004. Non-additive shortest paths. In *European Symposium on Algorithms*, 822–834.

Tsaggouris, G., and Zaroliagis, C. D. 2006. Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications. In *International Symposium on Algorithms*, 389–398.

Wein, R.; van den Berg, J. P.; and Halperin, D. 2008. Planning high-quality paths and corridors amidst obstacles. *International Journal of Robotics Research* 27(11-12):1213–1231.