A Additional Experiments

A.1 Driving Domain

We also evaluate our method on a driving domain. In this problem, the vehicle is attempting to make an unprotected left turn, and there is both cross and oncoming traffic. An example of this domain is depicted in Figure 9.

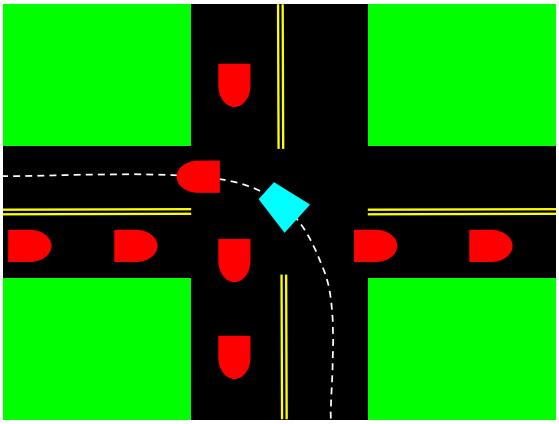

Fig. 9. The robot (blue trapezoidal vehicle) making an unprotected left turn along the dotted white curve. Each obstacle (red rounded vehicles) exists in space-time and has uncertain speed. There is cross traffic going to the right blocking the robot's path before entering the intersection, oncoming traffic going downwards blocking the robot's path before exiting the intersection, and an obstacle vehicle in front. The robot must choose when it is safest to cut between vehicles, keeping in mind that going too fast risks collision with the front vehicle. There are a total of 12 obstacle vehicles.

The geometric curve the vehicle will follow is fixed, but the vehicle has the option to proceed forward or wait at each timestep. Hence, the graph is a 2D lattice where one dimension is progress along the curve (40 steps) and the other dimension is time (100 timesteps). This graph is fed as input to the graph search algorithms, each of which returns a trajectory that indicates when the robot should be moving and when it should be stopping. Practically speaking, a solution trajectory makes two choices: which vehicles to cut between when entering the intersection, and which vehicles to cut between when leaving the intersection. The performance of these methods are compared in Figure 10. We also measured the effect of the collision horizon on these performance metrics, depicted in Figure 11. We find that M_0 and running MCR on sampled obstacles produce plans of similar quality, although M_0 is significantly faster. As before, setting the shadows to be equal is suboptimal in nearly all of the problems in this domain because there are too many obstacles, so it must choose an overly conservative shadow for each one. Overall, M_1 appears to be the most generally attractive option, as it is optimal in every instance, although in this domain increasing the collision horizon does not appear to increase runtime significantly.

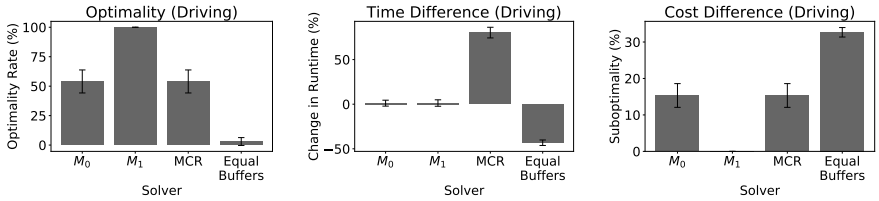

Fig. 10. The optimality rate (percent of problem instances where the algorithm returns an optimal solution), runtime, and planning risk of each method. Runtime and cost are depicted as the difference compared to the optimal planner M_{12} to control for the variance between problem instances.

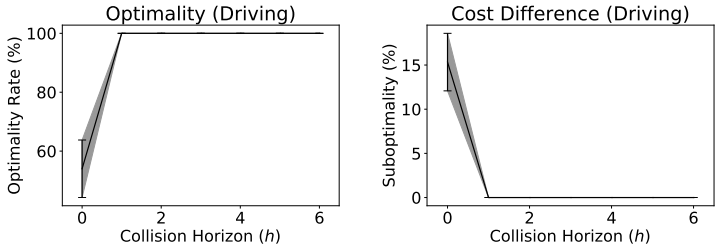

Fig. 11. The optimality and planning risk of M_h for each collision horizon. Cost is depicted as the percent difference compared to the optimal planner M_{12} to control for the variance in difficulty between problem instances. There was no significant difference in runtime across different values of h, so the runtime graph is omitted. Increasing the collision horizon past 6 shows no noticeable change in behavior, so we have cropped the graphs for clarity.

A.2 Minimum Constraint Removal for Manipulation Planning

We also evaluate our algorithm as a minimum constraint removal planner. Our experimental domain is identical to the one in Section 4.1, but the obstacles are deterministic and the task is instead to find the path with the fewest collisions. This task is very practically relevant in manipulation planning domains to determine which obstacles must be moved out of the way in order to perform a given operation.

As described before, Hauser [9] presented two algorithms, a greedy planner and an exact planner, which are equivalent to M_0 and M_{24}, respectively (note that M_{24} is equivalent to M_∞ when there are at most 24 obstacles). Our algorithm is compared for different settings of the collision horizon in Figure 12.

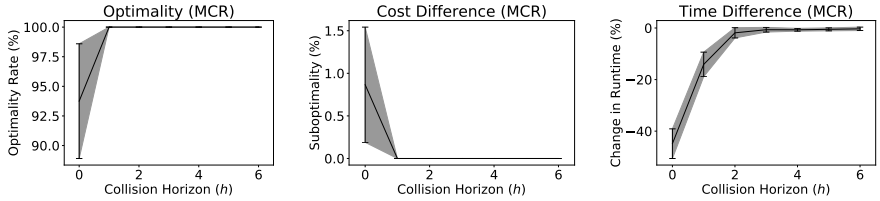

Fig. 12. The optimality, runtime, and planning risk of M_h for each collision horizon. Runtime and cost are depicted as the difference compared to the optimal planner M_{24} to control for the variance in difficulty of different problem instances. Increasing the collision horizon past 6 up to 24 shows no noticeable change in behavior, so we have cropped the graphs for clarity.

Similar to minimum-risk planning, we find that M_0 is already near optimal, and that M_1 closes the gap. As a result, it is unclear whether the collision horizon is bounded for this domain, or if it is just highly likely to be small. As before, M_1 strikes a good balance of optimal performance and quick runtime.

B Step-by-step Example of Algorithm

B.1 Problem Setup

Here we walk through a step-by-step example of running Algorithm 1 on a small problem. Suppose we have a graph with vertices v_1, v_2, v_3, v_4 and edges

$$\begin{aligned} e_1 &= (\mathbf{v_1}, \mathbf{v_2}) \\ e_2 &= (\mathbf{v_1}, \mathbf{v_3}) \\ e_3 &= (\mathbf{v_2}, \mathbf{v_3}) \\ e_4 &= (\mathbf{v_3}, \mathbf{v_4}) \\ e_5 &= (\mathbf{v_3}, \mathbf{v_2}) \end{aligned} \tag{1}$$

Then let there be 2 obstacles o_1 and o_2, each with two risk levels of .01 and .05, where

$$\begin{aligned} &f_{o_1}(\{e_1\}) = f_{o_1}(\{e_3\}) = f_{o_1}(\{e_4\}) = f_{o_1}(\{e_5\}) = .05 \\ &f_{o_1}(\{e_2\}) = f_{o_1}(\{e_4\}) = 0 \\ &f_{o_2}(\{e_2\}) = .01 \\ &f_{o_2}(\{e_1\}) = f_{o_2}(\{e_3\}) = f_{o_2}(\{e_4\}) = 0 \end{aligned} \tag{2}$$

B.2 Algorithm Execution

We will now walk through M_1 starting at $\mathbf{s} = \mathbf{v_1}$ and ending at $\mathbf{t} = \mathbf{v_4}$. The referenced psudeocode may be found in Algorithm 1.

Iteration 1 We start with an empty closed set (line 1) and an open set containing the start vertex $\mathbf{v_1}$ (line 2). When we enter the main loop (line 3), we assign $\mathbf{u} = \mathbf{v_1}$ with empty τ and C (line 8). $\mathbf{u} \neq \mathbf{t}$, and so we do not end here (line 9).

Now we get to line 16. The closed set is empty, and so the first clause evaluates to true since there does not exist any $(\mathbf{w}, C'')$ in closed. For the second clause, we can set $M = \{\} \leq C$, which satisfies $|M| \leq 1$ and also there not existing any $(\mathbf{w}, C'')$ in closed. Hence, the overall predicate evaluates to true.

We then add $(\mathbf{v_1}, \{\})$ to the closed set (line 17). For outgoing edge e_1 (line 19), we set $C' = \{(o_1, .05)\}$ (line 20) and add $(\mathbf{v_2}, [e_1], C')$ to the open set with key .05 (line 21). For outgoing edge e_2 (line 19), we set $C' = \{(o_2, .01)\}$ (line 20) and add $(\mathbf{v_3}, [e_2], C')$ to the open set with key .01 (line 21).

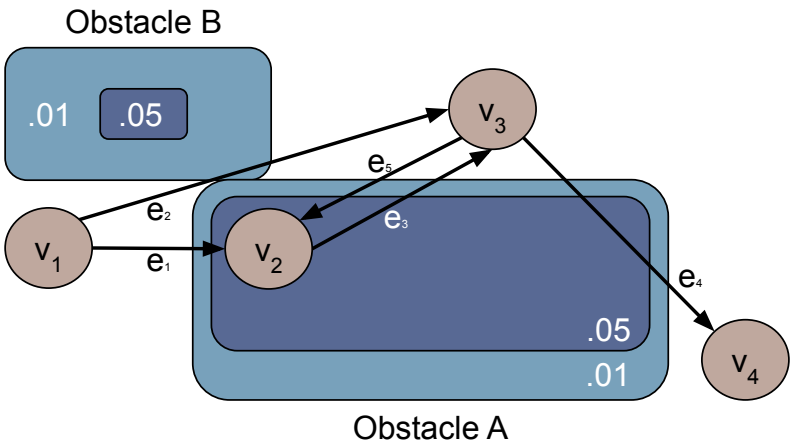

Fig. 13. End of iteration 1.

The set of closed states at the end of this iteration is depicted in Figure 13.

Iteration 2 Now we come back to the top of the main loop (line 3). We assign $\mathbf{u} = \mathbf{v_3}$ with $\tau = [e_2]$ and $C = \{o_2, .01\}$ (line 8). $\mathbf{u} \neq \mathbf{t}$, and so we do not end here (line 9).

There is no closed state at vertex $\mathbf{v_3}$, and so the predicate evaluates to true (line 16).

We then add $(\mathbf{v_3}, \{(o_2, .01)\})$ to the closed set (line 17). For outgoing edge e_4 (line 19), we set $C' = \{(o_1, .05), (o_2, .01)\}$ (line 20) and add $(\mathbf{v_4}, [e_2, e_4], C')$ to the open set with key .06 (line 21). For outgoing edge e_5 (line 19), we set $C' = \{(o_1, .05), (o_2, .01)\}$ (line 20) and add $(\mathbf{v_2}, [e_2, e_5], C')$ to the open set with key .06 (line 21).

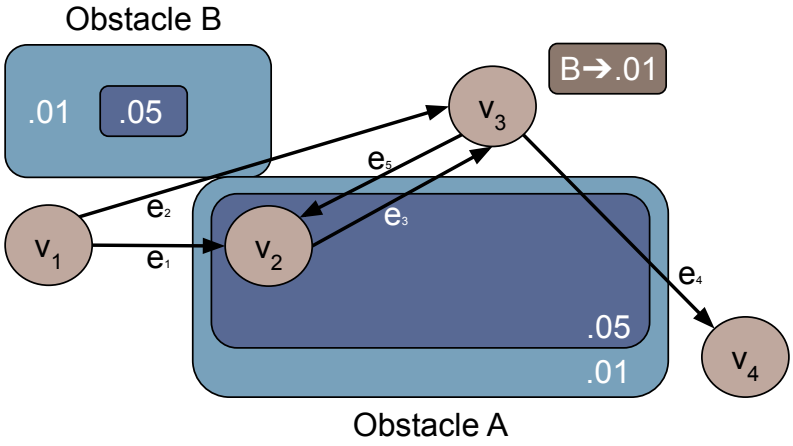

Fig. 14. End of iteration 2.

The set of closed states at the end of this iteration is depicted in Figure 14.

Iteration 3 We come back to the top of the main loop (line 3). We assign $\mathbf{u} = \mathbf{v_2}$ with $\tau = [e_1]$ and $C = \{o_1, .05\}$ (line 8). $\mathbf{u} \neq \mathbf{t}$, and so we do not end here (line 9).

There is no closed state at vertex $\mathbf{v_2}$, and so the predicate evaluates to true (line 16).

We then add $(\mathbf{v_2}, \{(o_1, .05)\})$ to the closed set (line 17). For outgoing edge e_3 (line 19), we set $C' = \{(o_1, .05)\}$ (line 20) and add $(\mathbf{v_3}, [e_1, e_3], C')$ to the open set with key .05 (line 21).

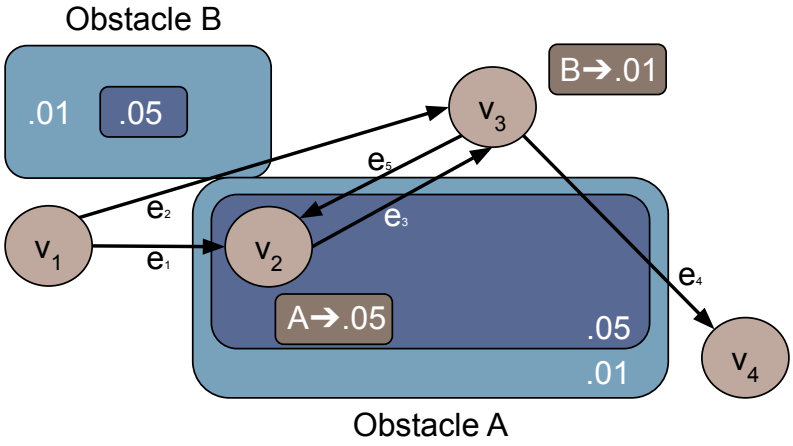

Fig. 15. End of iteration 3.

The set of closed states at the end of this iteration is depicted in Figure 15.

Iteration 4 Now we come back to the top of the main loop (line 3). We assign $\mathbf{u} = \mathbf{v_3}$ with $\tau = [e_1, e_3]$ and $C = \{o_1, .05\}$ (line 8). $\mathbf{u} \neq \mathbf{t}$, and so we do not end here (line 9).

There is only closed state $(\mathbf{v_3}, \{(o_2, .01)\})$ at vertex $\mathbf{v_3}$. $\{(o_2, .01)\} \not\leq C$ and we can set $M = C$ satisfying $|M| \leq 1$ and $M \not\leq \{(o_2, .01)\}$. Therefore, the predicate evaluates to true (line 16).

We then add $(\mathbf{v_3}, \{(o_1, .05)\})$ to the closed set (line 17). For outgoing edge e_4 (line 19), we set $C' = \{(o_1, .05)\}$ (line 20) and add $(\mathbf{v_4}, [e_1, e_3, e_4], C')$ to the open set with key .05 (line 21). For outgoing edge e_5 (line 19), we set $C' = \{(o_1, .05)\}$ (line 20) and add $(\mathbf{v_2}, [e_1, e_3, e_5], C')$ to the open set with key .05 (line 21).

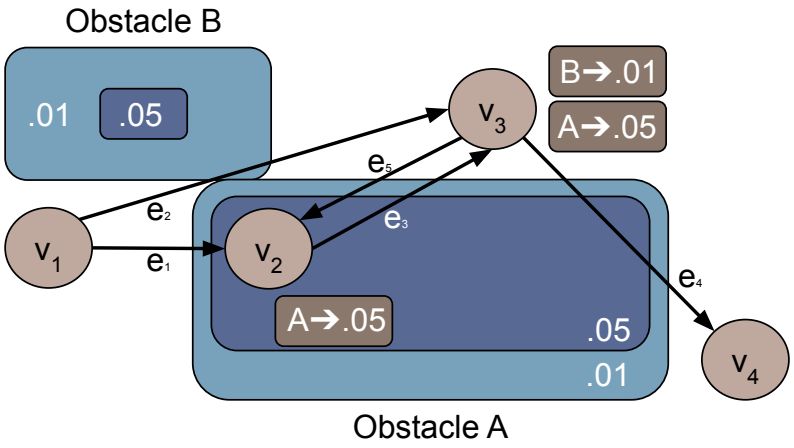

Fig. 16. End of iteration 4.

The set of closed states at the end of this iteration is depicted in Figure 16.

Iteration 5 We come back to the top of the main loop (line 3). We assign $\mathbf{u} = \mathbf{v_2}$ with $\tau = [e_1, e_3, e_5]$ and $C = \{o_1, .05\}$ (line 8). $\mathbf{u} \neq \mathbf{t}$, and so we do not end here (line 9).

There is only closed state $(\mathbf{v_2}, \{(o_1, .05)\})$ at vertex $\mathbf{v_3}$. $\{(o_1, .05)\} \leq C$ and so the predicate evaluates to false (line 16).

Because we did not need to expand this state, the closed set remains unchanged, and so it is still as depicted in Figure 16.

Iteration 6 Finally, we reach the top of the main loop (line 3) with $\mathbf{u} = \mathbf{v_4}$, $\tau = [e_1, e_3, e_4]$ and $C = \{o_1, .05\}$ (line 8). $\mathbf{u} = \mathbf{t}$, and so we return the solution $[e_1, e_3, e_4]$ (line 9).

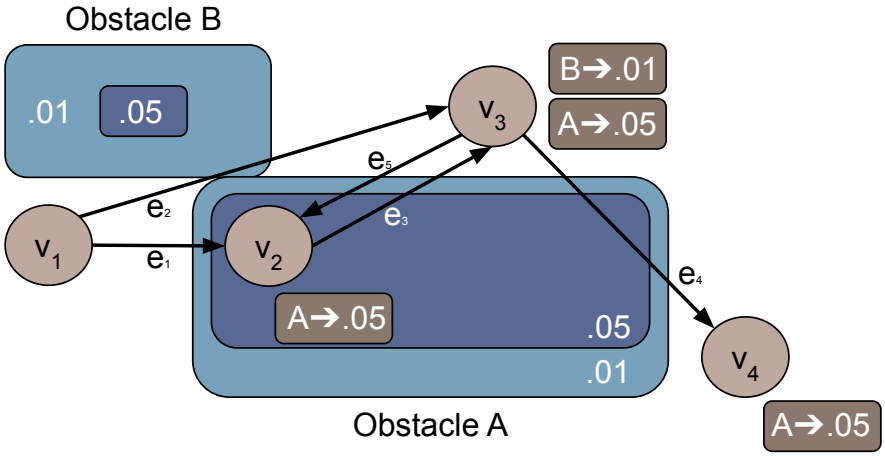

Fig. 17. End of iteration 6.

The set of closed states at the end of this iteration is depicted in Figure 17.

C Theoretical Guarantees for Motion Planning

While motion planning has been shown to be PSPACE-hard [23], the community has developed algorithms which are able to solve many practical motion planning problems. Even though we cannot guarantee that any algorithm is both efficient (polynomial time) and complete (guaranteed to find a solution), we can provide lighter guarantees. The goal of this section is to provide background on different theoretical guarantees relevant to motion planning with obstacle uncertainty.

C.1 Completeness

An algorithm is said to be *complete* if it is always able to find a solution if one exists. Unfortunately, many algorithms which depend on heuristics are not complete and can fail in certain scenarios. For example, optimization based motion planners can fail to find a solution if they are not initialized with a trajectory whose homotopy class contains a valid trajectory.

When working with sampling based motion planners, we work with a criteria known as *probabilistic completness*, a property introduced with the RRT algorithm [16]. While we cannot guarantee that any algorithm efficiently returns a valid solution if one exists, we can ensure the algorithm is probabilistically complete.

Definition 9 (Probabilistically Complete Motion Planning Algorithm). *A motion planning algorithm takes a set of obstacles, a start state* $\mathbf{s}$*, and a goal state* $\mathbf{t}$ *as input and generates a trajectory that does not intersect any obstacle as output. A planning algorithm is probabilistically complete if, with* n *samples, the probability that it finds a safe trajectory approaches* 1 *as* n *approaches* ∞*.*

Probabilistic completeness essentially means the algorithm will eventually find a solution if one exists.

Many sampling based algorithms, including RRTs, guarantee probabilistic completeness as long as there exists a solution that meets certain conditions. In particular, RRTs and many other sampling based motion planners are only probabilistically complete if there exists a solution trajectory in the topological interior of free configuration space. Since a similar condition is necessary for the algorithm presented in this paper, we develop this condition without explicitly relying on topology below.

This condition can be articulated as the existence of a path in the δ-interior of the free space X_{free} where X_{free} is a bounded subset of $\mathbb{R}^n$.

Definition 10 (δ-interior [13]). *A state $x \in X_{free}$ is in the δ-interior of X_{free} if the closed ball of radius δ around x lies entirely in X_{free}.*

A sampling based algorithm can only succeed if it always has a non-zero probability of sampling a waypoint leading to more progress. One way to ensure this is requiring the existence of a solution where every waypoint has a ball of nonzero diameter around it, guaranteeing that said ball has non-zero measure and the algorithm will eventually draw a sample in said ball.

When there exists a path in the δ-interior of free space for some $\delta > 0$, many sampling based motion planners are probabilistically complete. However this formulation does not extend well to the domain with uncertain obstacles; there is no concept of "free space" because the locations of the obstacles are not known. Instead we will use the equivalent view of inflating the path instead of shrinking the free space.

Definition 11 (δ-inflation). *The δ-inflation of the set X is the set $Y = \bigcup_{x \in X} \{y \mid d(x, y) \leq \delta\}$, where $d(x, y)$ is any distance metric.*

We note that in the deterministic setting, if a trajectory is in the δ-interior of X_{free}, then the δ-inflation of the trajectory is entirely in X_{free}. This allows us to consider problems with the following regularity condition: there exists a δ-inflated trajectory that has a low risk of collision.

Definition 12 (ϵ-safe δ-inflated trajectory). *A trajectory τ is an ϵ-safe δ-inflated trajectory if its δ-inflation intersects an obstacle with probability at most ϵ.*

While the standard RRT requires the existence of a trajectory in the interior of free space in order to be probabilistically complete, the algorithm in this paper requires the existence of an $\epsilon-$safe $\delta-$inflated trajectory in order to be probabilistically complete.

C.2 Graph Restriction Hardness

When solving motion planning without uncertainty, once the algorithm has identified a graph that contains a solution, the problem is essentially solved. Algorithms like Djikstra's algorithm and A* can be applied out of the box to find the minimum cost path within said graph.

We refer to the hardness of finding a solution restricted to a graph the *graph restriction complexity* of a problem. Since we can apply Djikstra's algorithm to solve motion

planning without uncertainty, the graph restriction complexity is P. This is crucial to enabling the fast solving of motion planning in practice. Sampling based planners tackle the problem in two [sometimes alternating] phases. The first phase involves sampling a graph. The second involves checking if the graph contains a solution, and finding the lowest cost one if it does. Motion planning problems that are "easy" for sampling based planners are ones where the first phase is easy. The hardness of the second phase is exactly the graph restriction complexity.

One could hope that with the right approximations, a similar pattern could work for motion planning with obstacle uncertainty. In [3], the authors develop a notion of confidence intervals around obstacles with the aim of using them as an approximation enabling efficient planning. Unfortunately, the graph restriction complexity of planning with obstacle uncertainty with shadows is still NP-hard, even in two dimensions [25]. In other words, even if you are able to efficiently identify a graph containing the solution, it is not easy to find the solution in this graph unless P=NP.

This paper presents an algorithm that shows that the graph-restriction complexity of MRMP is P when the collision horizon is constant. Under these conditions, the same paradigm as for standard motion planning applies. One can sample a graph in configuration space just like with a standard sampling based motion planner and then use the presented graph search algorithm in order to solve MRMP. This allows us to adapt the standard sampling based motion planners to be able to solve MRMP.

D Proofs

Proof of Theorem 4:

Proof. Let ϵ be the associated cost of the trajectory generated by $M_h(G, O, \mathbf{s}, \mathbf{t})$ and let τ_* be any optimal trajectory, with associated cost ϵ_*. Because each obstacle is distributed independently from other obstacles, we begin by considering the risk incurred by each one separately. For a given obstacle o, suppose $S_o^{(\tau_*)}$ is the trajectory τ_* split into the fewest segments such that for each $\tau_i \in S_o^{(\tau_*)}$, $H_o^{(\tau_i)} \leq h$. For each time τ_i enters an obstacle level with edge (u, v), the planning tree generated by M_h must contain a state $\hat{s}_u$ at vertex u with memory containing the preceding h collisions in τ_i since such a state is reachable (given that τ_i reaches it) and would not be skipped unless another previously closed state at u already contained the preceding h collisions. Because $H_o^{(\tau_i)} \leq h$, we know that the preceding h collisions are sufficient to determine the marginal risk of each collision. Then M_h will at some point expand edge $(\hat{s}_u, \hat{s}_v)$, where $\hat{s}_v$ is the state at vertex v still with memory containing the preceding h collisions in τ_i and with cost from o no more than $f_o(\tau_i)$. Hence, M_h computes the marginal risk of this obstacle for this subtrajectory correctly. Then the total computed risk from obstacle o for trajectory τ_* is at most

$$\sum_{\tau_i \in S_o^{(\tau_*)}} f_o(\tau_i)$$

Hence, the algorithm would assign the overall risk of trajectory τ_* as at most

$$\widetilde{\epsilon} \leq \sum_{o \in O, \tau_i \in S_o^{(\tau_*)}} f_o(\tau_i)$$

Because M_h greedily expands nodes in order of computed cost, it would only select a different trajectory if its computed cost $\hat{\epsilon} \leq \widetilde{\epsilon}$. We know that M_h can only overestimate the cost of a trajectory (due to not taking into account the optimal set of past collisions), not underestimate, so the cost of the trajectory it returns is at most $\hat{\epsilon}$. Finally, since $f_o(\tau_*) = \max_{\tau_i \in S_o^{(\tau_*)}} f_o(\tau_i)$ and

$$\epsilon_* = \sum_{o \in O} f_o(\tau_*) = \sum_{o \in O} \max_{\tau_i \in S_o^{(\tau_*)}} f_o(\tau_i)$$

we are left with the following bound:

$$\epsilon \leq \hat{\epsilon} \leq \widetilde{\epsilon} \leq \epsilon_* + \sum_{o \in O} (|S_o^{(\tau_*)}| - 1) f_o(\tau_*)$$

E Additional Runtime Analysis

Although the runtime of the exact algorithm (i.e. M_k) has poor asymptotic complexity according to our analysis, in our experiments we observe that in practice it seems to not be that much slower than the approximate versions. We would like to investigate why that would be the case, since it is a surprising result.

We define an extension of the irreducible constraint removal (ICR) defined by Hauser [9] for MCR domains.

Definition 13 (minimal reachable memory (MRM)). *A memory C is a minimal reachable memory for a configuration $\mathbf{u}$ if there exists a trajectory from the initial configuration $\mathbf{s}$ to $\mathbf{u}$ that does not collide with any shadow not in C, and there is no lower memory $C' < C$ such that there exists such a trajectory that does not collide with any shadow not in C'.*

Hauser [9] observe that the pruning of non-ICR states eliminates a large number of states, leading to practical efficiency for many MCR problem instances. In our algorithm, we prune non-MRM states (note that when using our algorithm as an MCR planner, this is equivalent to pruning of non-ICR states), so we would like to quantify how much of an effect the pruning has on the overall runtime.

Let U denote the set of risk memories associated with all states for configuration $\mathbf{u}$ that are expanded by M_k. Suppose $C_1, C_2 \in U$, and M_k expanded C_1 before C_2. If $C_1 \leq C_2$, then C_2 would not have been expanded since $(\mathbf{u}, C_2)$ is a strictly worse state than $(\mathbf{u}, C_1)$, hence a contradiction. But if $C_2 \leq C_1$, then C_2 would have been expanded before C_1, which is also a contradiction. Therefore, C_1 and C_2 are incomparable, and so U is an antichain.

For the purpose of simplifying the following math, we will continue only for the case where $L = 1$ (i.e. equivalent to MCR), but we believe the general orders of magnitude of the effect to be similar for larger L. In this case, the maximum antichain is the set of subsets of size $\frac{k}{2}$, which has size $\binom{k}{\frac{k}{2}}$. However, the actual antichain the algorithm produces is usually much smaller.

One reason is that it selects ICR sets rather than those in the maximum antichain. If most ICR sets have size approximately λ, then the size of U would be reduced to

approximately $\binom{k}{\lambda}$. This is the extent of the effect of pruning on the runtime, and it does not appear to sufficiently explain the low runtime we see in practice (in the boxes MCR domain, we see typical ICR sizes of on the order of around 8, which would still suggest a hundreds-of-thousands-fold runtime multiplier over the greedy algorithm).

We speculate that the topology of the obstacle placements induce additional constraints on which memories are reachable. For example, if the obstacles are arranged into a 2D grid, you would not be able to reach a shadow in the corner without also reaching some set of shadows in-between. Computational experiments suggest that this would reduce the size of U to a manageable number for the size of problems we have shown here, Exact counts for an MCR domain where $\mathbf{s}$ and $\mathbf{t}$ are δ cells apart and the algorithm is limited to reaching Δ obstacles (e.g. due to the algorithm terminating upon reaching the goal by passing through Δ obstacles) shown in the below table. Beyond the boundaries of the table, we expect that the size of U has asymptotic complexity at least $o\left(\delta^{\text{poly}(\Delta-\delta)}\right)$.

		Δ				
		8	9	10	11	12
δ	6	31	31	203	203	823
	7	1	43	43	375	375
	8	1	1	57	57	647
	9	0	1	1	73	73
	10	0	0	1	1	91

We expect the boxes domain to exhibit similar properties due to the dense placement of the obstacles (we estimate that in the boxes MCR domain the goal is reached with cost around 10 and the average obstacle is reached with cost around 8), although likely in an inexact manner. A potential line of future work could be to analytically quantify the impact of the obstacle topology on the runtime of the algorithm, and determine whether that sufficiently explains the empirically low runtime we have observed. Additionally, it would be helpful to evaluate the runtime of our algorithm on problems where δ and $\Delta - \delta$ are much larger to see if it continues to be efficient. This would also provide further intuition as to what is driving the hardness of the general problem, and which special cases can be solved efficiently when an exact solution is required and the collision horizon is unknown.