BOUNDED UNCERTAINTY ROADMAPS

EHSAN REHMAN

M.Comp (NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgments

I would definitely thank my supervisor, David Hsu, the most. He always had something interesting to say: some good advice and encouragement even during the periods when results were hard to come by. His ability to see the underlying causes from experimental results, was inspiring. He guided and led me when I needed it, and gave total freedom at some other times.

The National University of Singapore provided all the needed resources, which are hard to mention within a reasonable space: libraries, internet, labs, offices etc etc. It all made it possible to concentrate on research without too many problems. Sure there are complaints; all of us complain one time or the other, but that is natural. I would say that in NUS it is much easier for things to be worse than what they currently are, and harder for them to be any better, because the conditions are already quite good.

I am thankful to my parents who encouraged me to attempt a PhD in the first place. In fact their encouragment and support goes much farther back than that. They have encouraged me towards critical thinking, and the Sciences and Mathematics since the very beginning. My colleagues Hanna Kurniawati and Saurabh Garg were also very helpful. They have given me useful advice on multiple, important occasions during my research.

Contents

Abstract

In this thesis, we give algorithms to plan the motion of the robot when the position and shapes of the obstacles are not known precisely, as well as these obstacles can change dynamically over time. These changes could be to the number of obstacles, their shape, as well as to their position.

The Robot Motion Planning problem is: Find a valid sequence of actions of the robot, taking it to the given goal configuration, while avoiding collisions with obstacles. Probabilistic Roadmaps (PRMs) are state-of-art motion planners solving many hard problems. PRMs make a graph (roadmap) in the free-space of the robot, and then search for the shortest path in the roadmap from source to goal. However classical PRMs assume that the environment is known with perfect accuracy, and does not change over time. These assumptions are not true in practice.

We first give the Bounded Uncertainty Roadmap (BURM), extending PRMs to the case when the environment is not known with perfect accuracy: The coordinates of obstacle vertices are now specified via probability distributions, $f(v)$. The novelty in BURM is that we do not compute the exact probabilities of collision at robot configurations, but only maintain rough bounds on them. Bounds are refined to different degrees of resolution in

different parts of the configuration space of the robot, depending on their relevance in finding the best path through the roadmap. This makes BURM quite efficient. Part of these refinements are performed by a hierarchical subdivision of the geometry of the robot and obstacles, and computing probabilities of collision over these subdomains; while part of it is achieved using Monte Carlo Integration.

We extend BURM to give the Dynamic BURM (DBURM). Now the environment can change dynamically, which we model by changing $f(v)$. After every change, a certain 'local consistency condition' may be violated at some nodes in the roadmap. DBURM identifies the non-consistent nodes and iteratively makes them consistent. Once all nodes are consistent, the shortest path is easy to compute. DBURM is shown to be more efficient than re-running BURM after every change. Similar re-planning algorithms existed before, but not for the case when edges in the roadmap are annotated by cost *bounds*. This is different and more challenging.

Suppose the expected costs of edges, $c(e)$, in a graph are unknown. We can not always compute definite bounds on $c(e)$ as above, but by sampling the costs we can compute an interval that contains $c(e)$ with a high probability. In such a case we can only return a path that is optimal with a high probability. We give the Probably Approximately Correct BURM (PACBURM) that makes a further small sacrifice while making a large gain in efficiency: PACBURM returns a path that with a high probability is *close in cost* to the optimal path. Hence PACBURM returns a *probably approximately optimal* path. We do not know of any other graph algorithm in AI literature that computes a probably approximately optimal path in such a setting.

List of Tables

List of Figures

Chapter 1

Introduction

In the real world many robots need to *move* in order to perform their task. Hence in order to achieve the task, we need to plan the motion of the robot. This plan could be pre-programmed if the robot always stays in a pre-specified environment, and always executes a known task. However this is not usually true: Robots keep venturing into new environments and carry out a variety of tasks that are not known in advance. It is therefore imperative that the robot be able to autonomously plan its motion. The motion planning problem is: To plan a path getting the robot from its initial configuration to the given goal configuration while avoiding collision with obstacles.

In this thesis we give efficient algorithms for generating motion plans, when the coordinates of the obstacles in the environment are not known with perfect precision, and may even change dynamically. Later we generalize our technique to return a *probably approximately* optimal path in a graph where the expected costs of edges are not known precisely. That is, the path reutrned by the algorithm is within a cost of δ_1 of the optimal path, with a

probability of $(1 - \delta_2)$, where δ_1 and δ_2 are small positive numbers.

1.1 The Probabilistic Roadmap Planner and its assumptions

We first define what we mean by *number of degrees of freedom*, and the *configuration space* of the robot, which are important concepts in motion plannning. The number of degrees of freedom (dofs) is the minimum number of parameters required to specify the configuration of the robot. For example a polygon in 2 dimensions that can translate, has 2 dofs, as its x and y coordinates are enough to specify its exact configuration. If it could also rotate, it would have 3 dofs, as we now need to specify the angle as well. Another example: A robot arm that is fixed at its base, but can rotate at the wrist and elbow has 2 dofs. If the base could also translate in (x, y, z), it would have 5 dofs. Each configuration of the robot is represented by a point in the *configuration space*, C of the robot. Hence the number of dimenstions of C equals the number of degrees of freedom of the robot.

The first motion planners were *Complete planners*. These planners return a valid path whenever one exists from the source to the goal, and indicate no path exists otherwise. All known complete planners take time that is exponential in the number of DOFs of the robot. Known complete planners are quite inefficient if the robot has more than 4 dofs, and many useful robots unfortunately do have more than 4 dofs

Probabilistic Roadmap planners (PRM) [44, 45] are *probabilistically* com-

plete planners and have been successful in efficiently solving many hard problems, even for robots with large number of dofs. This efficiency comes from the fact that PRMs do not attempt an exact representation of the high dimensional configuration space, C of the robot. In fact, PRMs approximate this space with a graph, and then carry out a shortest path search in this graph from initial to goal node. This graph is known as the 'roadmap' in motion planning literature. PRM first samples C for configurations that are not in collision. These samples become the nodes of the roadmap. Then it connects a subset of these nodes by *free* edges; i.e. the robot would not collide with an obstacle when carrying out the motion along the edge. These nodes and edges define the roadmap.

As PRMs represent a high dimensional configuration space by a simple roadmap, PRMs are going to be very efficient for many motion planning problems with high dimensional configuration space. However the classical PRMs make a couple of assumptions that are not true in practice:

1. PRMs assume that the environment is perfectly known. That is, we know the coordinates of the obstacles with perfect precision.

2. PRMs assume that the environment is static. That is, the environment is not changing over time.

1.1.1 Sources of imprecision in obstacle coordinates

The imprecision in the obstacle coordinates (point 1 above) comes from a couple of sources: errors in robot sensors and errors in robot control. The environment map is constructed by a robot that travels in the given envi-

ronment, detecting environment features. Hence the coordinates of a given feature depends on the coordinates of the robot, as well as the readings of its sensors. Typically neither is known accurately. The position of the robot can be in error because of wheel slipping and unevenness of the floor, among other reasons [89]. Therefore the robot does not have precise knowledge of the distance travelled, and hence its position. The sensors are also prone to errors: sensors such as sonar range scan, and laser range scan, can have errors because of limited resolution of the sensors, atmospheric effects, reflective surfaces, and others. [89]

These imprecisions imply that rather than assigning exact coordinates to the detected environment features, we should assign a probability distribution: Given a feature f, and coordinates (x, y), the probability distribution specifies the probability that f is at (x, y). There are algorithms such as Extended Kalman Filters, among others, that are used to specify the feature coordinates in terms of probability distributions [89].

1.2 Extending the PRMs: the contributions

The simplicity and efficiency of PRMs means, that it would be of great utility if we can extend the PRMs so that the two assumptions discussed in section 1.1, are no longer made by the algorithms. This is the main aim of this thesis.

As explained above, we need to specify the coordinates of the environment features via probability distributions, and there exist algorithms that do so. We do not implement such algorithms, and assume that an environment

map with such probabilisitic data is given to us. We model the probabilistic data by specifying the coordinates of the obstacle *vertices* by probability distributions. The reasons for choosing vertices is that the obstacles very often are, or are modeled as, polygons. Now given a list of vertices of the polygon where every two adjacent vertices are connected by an edge, the shape and position of the polygon is completely determined by the position of the vertices. Hence it suffices to model uncertainty of shape and position, by assigning uncertainties at the vertices.

We extend the PRM and give a planner, BURM (Bounded Uncertainty Roadmap) [29] in chapter 3 to plan under imprecisely known environment. BURM builds a roadmap like PRM, but now the expected cost of traversing an edge depends on the probabilities of collision when traversing it. The main contribution of BURM is that rather than computing these probabilities exactly, it keeps an upper and lower bounds on them. The bounds are refined only if it helps in identifying the optimal path. The refinements are done partially by Monte Carlo Integration to compute the probabilities, but partially by dividing the geometry of the robot and obstacle to produce sub-domains, and then the probability of collision is computed over these subdomains. This hierarchical refinement is another contribution of BURM.

We also give a planner DBURM (Dynamic BURM) that extends BURM to environments that can alter. Now in BURM, vertex coordinates are specified via probability distributions. So we model a change in the environment, by altering some of these probability distributions. It is important to recompute the optimal path in the roadmap after such a dynamic change, because the previously computed path may be invalid, or even if it is still valid the

robot might pay a high cost in terms of collision while traversing it. One option is to re-run BURM from scratch after every dynamic change. However there might be a lot of information from our previous runs of BURM, that was not affected by the dynamic change, and could be reused. DBURM follows this methodology and repairs the computed shortest path tree, at only those nodes whose shortest path to goal might have been affected by the dynamic change. Similar replanning algorithms exist in literature [86, 48] for usual graphs. The contribution of DBURM is that it gives a replanning algorithm for the case when edge weights are *intervals* that can be tightened by more computation. The reason that they are intervals is because just like in BURM, we are computing bounds on probabilities of collision, which induce bounds on the expected cost of traversing the edge.

The BURM and DBURM make a roadmap, where the expected costs of edges are unknown, but contained in known intervals *with probability 1.* These intervals can be tightened by more computations. The idea of keeping bounds that can be tightened, is useful and general enough to have been discovered and applied in other domains [66]. However in a general scenario, we can not compute an interval that bounds the expected cost with *certainty.* But we can indeed compute an interval that contains the expected cost with a *high probability.* In Statistics, such intervals are called as *confidence intervals* of the expected cost. If the interval bounds the expected cost with a probability $(1 - \alpha)$, it is called as the $(1 - \alpha)$ confidence interval of the expected cost.

Suppose that we are given a graph such that the expected costs of edges are not known, as well as the cost of traversing an edge can vary; for e.g. the

time taken by a car to traverse a road can vary. In this setting it is natural to compute *confidence intervals* of the expected costs of edges. These intervals may be tightened by taking more samples, and we would like to return a close-to-optimal path without having to take too many samples. The best we can do is to return a path that is optimal with a high probability. We introduce the Probably Approximately Correct BURM (PACBURM) algorithm, that returns a path p, such that with a high probability p has a cost that is *close to the cost* of the optimal path. Hence we sacrifice the quality of the returned path, but in chapter 5 we show that making just a small sacrifice improves the running time by up to two orders of magnitude. Currently, we do not know of any generic algorithm that returns a probably approximately optimal path, by computing the confidence intervals of expected costs of edges.

1.3 Outline

In the next chapter we give a survey of related work. We give the BURM algorithm in chapter 3, that finds the optimal path through a roadmap when the position of obstacles are not known exactly. The BURM algorithm does not compute the exact edge weights, which depend on an exact computation of the probability of collision. Rather, it keeps bounds on the edge weights. These bounds can be refined, but after paying additional computation costs. The refinements are performed conservatively; only if it helps in identifying the optimal path. In chapter 4 we give the DBURM algorithm, which is an extension of the BURM algorithm - now we allow the environment to change dynamically. After every dynamic change DBURM efficiently recomputes

the optimal path through the roadmap. Similar re-planning algorithms exist in literature, but not when the edge weights are intervals. Lastly, in chapter 5 we give the Probably Approximately Correct BURM, that returns a probably approximately optimal path, in a graph. The setting is that the expected costs of edges are not known, but we can generate samples of costs. Hence we can compute $(1 - \alpha)$ confidence intervals of the expected costs. By keeping on sampling we can hope to tighten these intervals. In this chapter we prove and show that a small sacrifice in the quality of the returned path, can lead to a large improvement in the running time.

Chapter 2

Related Work

In this chapter we review the related work. We start with some early techniques in the domain of motion planning, leading to the successful Probabilistic Roadmaps (PRMs). Although PRMs have been quite successful in solving some difficult motion planning problems, the classic PRMs have two main drawbacks. PRMs assume:

1. Perfect knowledge of the environment

2. A static environment that does not change.

Our algorithm in chapter 3 extends PRMs to deal with point (1), and the algorithm in chapter 4 extends PRMs to deal with both (1) and (2). Therefore we also review other planners that do not assume (1) or (2). However note that in all the planners that we survey that extend PRMs, the following idea is missing: Keep bounds on the expected costs of edges rather than computing them exactly, and hierarchically refine the bounds but only if it helps in identifying the optimal path. This is the idea used in BURM, and

DBURM extends it for replanning in environments that are non-static.

In section 2.1 we note that motion planning problems are inherently hard, and mention some early motion planning methodologies. Efforts to develop planners that were efficient enough to be useful in practice, led to the Probabilistic Roadmap algorithm, which we describe in section 2.2. In section 2.3 we give an overview of motion planners that plan under uncertainty, in the knowledge of environment, and that in the resulting states of the robot after executing an action. These planners therefore do not make the assumption (1) of PRMs above. Our Bounded Uncertainty Roadmap algorithm is most related to planners under uncertainty of environment (sec 2.3.2) and planners that extend PRMs for uncertainty in environment (sec 2.3.1). We also review Markov decision processes(MDPs) and partially observable-MDPs (POMDPs) that are quite general techniques for motion planning uncertainty, but they are computationally expensive. In section 2.4 we review planners for dynamic environments; that is, these planners do not make assumption (2) above.

2.1 Early motion planning methodologies

Robot Motion Planning was identified as an area of research in the 1970*s*. In the simplest case the planner has to generate motion leading the robot from its starting configuration to the given goal configuration, while avoiding colliding with any obstacles. This problem is known to be PSPACE Hard [77]. This means that solving motion planning may require an unacceptable amount of time and space for non-trivial problems. *Complete planners* con-

struct a motion strategy whenever one exists, and indicate no such strategy exists otherwise. Because of the PSPACE hardness of the problem, we can't expect complete planners to be very efficient. Indeed, all known complete planners (see [52] for examples) run in time that increases exponentially with increasing number of degrees of freedom (dofs) of the robot.

Complete planners generally perform poorly on robots with more than 4 dofs, and such robots are widespread in the real world. To solve this we need to look at incomplete planners, that may have a high rate of success, but might also fail to generate a valid path even when one exists. Potential field approach [52] is one such methodology. It gives the obstacles a repulsive potential, and the goal an attractive potential. The robot is guided by these attractive and repulsive forces. However the robot can get stuck in local minima generated by these potential gradients. To alleviate this problem, randomized sampling was used [3] to attempt to guide the robot out of the local minimum if it gets stuck: In order to escape the local minimum after getting stuck, the robot executes small random steps. This led to planners that use randomized sampling from configuration space to build search trees, as the following section explains.

2.2 Planning by sampling from configuration space

The idea of randomized sampling was extended to the whole of planning process [44, 45]. Before explaining it, we first define the configuration space

C of the robot. It is the space where each configuration of the robot is represented as a point. Hence the number of dimensions of C is equal to the number of dofs of the robot.

In the probabilistic roadmap (PRM), the planner first samples uniformly at random from C. Given a configuration q, there is a primitive $free(q)$ which returns true, if and only if q is not in collision with any obstacle. Samples that are in collision with obstacles are rejected. The other samples become the nodes of a graph in C. The starting and goal configurations are also part of the node set. After some number of samples these nodes are then connected by straight line edges: if two nodes are within a given radius of each other in C, and if the straight line connector between them is collision free, then the two nodes are connected by this edge. Hence we now have a graph with a set of nodes and edges connecting them. This graph in motion planning literature is called the *roadmap*. As the nodes and edges of the roadmap are collision free, any path generated by following edges of the roadmap, is valid. Hence a valid path is returned by searching for a path in the roadmap from the start to goal node. Although quite simple, the PRM has been quite successful, and has efficiently solved many motion planning problems for robots with large number of dofs.

What PRM does is to avoid exploring C exactly, and represent it by the much simplified approximation of a roadmap. Note that PRM is not a complete planner. It is only probabilistically complete: for any given configuration space, the probability that a path exists and would not be found by the PRM, converges to 0 as the number of samples increase.

Sometimes we are interested in performing just a single query on a given

C. The Rapidly Exploring Random tree(RRT) [50], and the Expansive Space tree [35], solve this problem by making a *tree* in the configuration space and expanding it until the source and destination configurations are connected [50, 35]. In these techniques a new node is sampled, and then is connected to an already existing node x in the tree, by a collision free edge. The tree is therefore 'pulled' into unexplored regions of the configuration space, and the expansion is continued until both the starting and ending configurations are connected to the tree.

As noted at the beginning of this chapter, one of the two main shortcomings of the classic PRM is the assumption that the environment of the robot is perfectly known: $free(q)$ can always determine with certainty whether the configuration q is collision-free or not. However in the real world knowledge is often imperfect, and therefore we also need to consider seriously the issue of planning under uncertainty. The source of this imperfection is generally (i) imperfection in robot sensing and acting. That is robot sensors can not detect the position of objects with perfect precision, nor can they carry out an action with perfect precision. (ii) imperfect knowledge of the environment. The next section reviews planners under uncertainty.

2.3 Planning under uncertainty

Motion planning under uncertainty is an important problem in robotics and has been studied widely [52, 53, 14]. As noted above, in robot motion planning, uncertainty arises from two main sources: (i) noise in robot control and sensing and (ii) imperfect knowledge of the environment.

There were two early approaches to motion planning under uncertainty in motion and sensing. One was the two phase approach [9, 58] in which the plan was first constructed assuming no uncertainty. The plan is then locally patched by inserting complementary actions or sensory readings. It is a bit restrictive as we can't be certain if the plan can be locally patched. The second approach is the *preimage backchaining* [59]. In this we first identify the configurations, S, from which a commanded action is guaranteed to reach the goal. Then we identify the configuration S' from where the robot can reach S. Working iteratively like this, the planner continues until it reaches the starting configuration. We then have a sequence of commands leading from the initial to goal. However, the complexity of this method is double exponential in the number of plan steps [11].

One reason that preimage backchaining is inefficient, is because of the interaction of motion uncertainty and sensing uncertainty. The goal has to be reached despite motion uncertainty, and recognized despite sensing uncertainty. This recursive relationship was noted in [71]. Erdmann [22] decoupled the two, by assuming that a certain subset of the goal configuration can be recognized by the robot regardless of the way it was achieved. Another way is to reduce the uncertainty in robot position by using *landmarks* as explained below.

One way of reducing uncertainty in position many practical situations is the use of landmarks in the environment. For example if a robot can sense a known building in the environment, then it would reduce the uncertainty in the current position of the robot [55]. This method has been used along with preimage backchaining to give more efficient algorithms [54, 8]. In [54]

the algorithm is polynomial time. The environment has obstacle disks and landmark disks, and the number of obstacle disks is $O(l)$, where l is the number of landmark disks. It is also assumed that the robot has perfect localization in the landmark disks. In [26] the concept of landmarks was used to introduce the first planner taking into account both uncertainty and non-holonomic constraints of the robot. In [79] landmarks as well as the probability that they will be obstructed by moving people etc, is considered. Hence the robot should go along a path where the successful identification of landmarks is high, as well as the path reaches the goal in a short distance. In [28] the robot localizes with help of *linear* landmarks (such as a wall, as oppossed to a point landmark such as a building).

2.3.1 Extending sampling based planners for uncertainty

Due to the high success of sampling based motion planners, many algorithms have attempted extending sampling based planners for uncertainty.

PRMs have been extended for uncertainty [94, 10, 61, 74]. In [74] the edges of the roadmap are the mean of the 'belief' of the current state of the robot, and are annotated with a covariance matrix - the covariance of the belief. Hence the roadmap is now in belief space and can use a mixture of both objectives: short path, and small covariance. This hence extends PRMs to planning under motion uncertainty. [94, 10, 61] deal with an uncertainty of the environment and are explained later.

In a PRM under uncertainty, an interesting problem is to find the short-

est path where the probability of collision is smaller than the user defined threshold. This is certainly not an easy problem and is proven to be NP-Hard w.r.t number of nodes [36]. However in [37] it is solved for the case of a robot with base pose uncertainty. The main idea used there is, if an edge in the roadmap has a probability of collision beyond a threshold, then any path containing it, need not be considered.

RRTs have also been extended to deal with motion uncertainty [5, 60, 46, 91, 70]. In [60] the tree is extended by adding nodes just as in RRT. However, now rather than directly adding an edge between the given two nodes, robot motion under a given command is simulated several times under different likely conditions, and the end states noted. The resulting states are grouped into clusters, and each cluster is treated as a node in the tree. The likelihood of reaching a node can be calculated using the likelihood of being in the starting conditions, from where the robot can reach that node. The extension of the tree is biased towards those configurations, q, where there is a greater probability of reaching q from the starting configuration. This biasing was given in [91]. In [46] the common technique of Monte-Carlo simulations is not used, and instead uncertainty is represented using SRSM (stochastic surface response method [38]). This uses a series of Gaussian random variables to represent uncertainty. The unknown terrain/robot parameters in these distributions can be estimated by a limited number of simulation. In [5], it is assumed that we have the stochastic dynamic and the stochastic observation model of the robot, and that they can be modeled as Gaussian noise. Given this information, and a path p, the probability distributions of the state and control input of the robot along p, are computed during the

planning phase. The method can therefore be used to assess the quality of a path with respect to a variety of measures. In [88] preimage backchaining is used to extend RRTs to handle uncertainty. (a) it first generates a tree in C like the RRT (b) efficient algorithms for computing Lyapanov functions [39, 69], are used as tools to help compute the preimages – and can be used even for 'very complicated dynamical systems'.

2.3.2 Planning under uncertainty in environment

There have been planners for planning when environments are not known perfectly [21, 94, 10, 61, 65]. In [21] the configuration space is divided into a grid, and each cell of the grid is annotated with the probability that the cell is occupied by obstacles. Assuming the uncertainty in robot control is negligible, one can then find a path with minimum collision cost using Dijkstra's or A* algorithm [18, 81]. In [94, 10, 61] the sampling based motion planning (PRMs, RRTs) is extended to deal with uncertainty. In [10], the robot is guided to explore those regions of the environment that would result in a large information gain about the environment and help in constructing the roadmap, by defining an expected utility of carrying out a given sensing action. In [61] the position of any vertex of an obstacle is specified only imprecisely, via probability distributions. A roadmap is then built and the edges of the roadmap are annotated with the expected cost of traversing the edge. In order to compute this, the planner also needs to compute the probability of collision along any edge of the roadmap.

2.3.3 MDPs and POMDPs

Markov decision processes (MDP) [31, 81] is a principled approach to solve planning under motion uncertainty. MDPs model reality as follows. Given the robot in state s, there is a set of actions available to the robot. However the result of each action is not known with certainty. For example, performing action a in state s, may lead the robot to state s' with *probability* 0.3. To state s'' with probability 0.4 etc. These outcome probabilities of each action in each possible state is known as the 'transition model'. Given a transition model, we need to construct a strategy that from any state s, recommends the action that maximizes expected benefit. For e.g. the robot should take action a in state s, action a' in state s_2 and so on. The value iteration and policy iteration algorithms [4, 31] are quite useful in constructing arbitrarily close approximations to the perfect strategy, in a reasonable amount of time.

MDPs have also been used to extend PRMs to under motion uncertainty [2]. Here from each node in the roadmap, possible actions of the robot are simulated many times to construct the transition model. Similar to above, the transition model specifies the probability of ending at a node n_2, given the action a was performed at node n_1. This transition model hence defines an MDP which can be solved by standard procedures.

For uncertainty in both sensing and motion, or if the map is not known with perfect precision, MDP is not enough. Because in these cases, the robot does not know the *current* state with certainty. Partially Observable MDPs (POMDPs) [85] extend MDPs to represent this. Here neither the result of a motion is known with uncertainty, nor the current state. In the field of

Artificial Intelligence POMDPs were introduced via [12, 40].

In summary POMDPs are similar to MDPs, because now although the robot does not know its state, it does know its *belief state*. So we can now treat belief state in POMDP, the way states are treated in MDP. The belief state is a probability distribution over all possible states. For example: given the current observation there is a probability of 0.2 that the robot is in state 1; a probability of 0.4 that it is in state 2 and so on. That is, it is the 'belief' about the current state of the robot. Similar to MDP, we wish for an optimal strategy: given the current belief, the robot should take sensing or motion actions to reach another belief state. The actions taken by the robot should be optimal in the sense of maximizing expected reward.

In MDPs each state has a value: the expected reward from this state onwards, following the optimal strategy. As the states are finite, the values can be kept in a table and updated by the algorithms. However in POMDPs the belief is a continuous high-dimensional probability distribution, hence we can't treat the beliefs one-by-one and compute their values. However these value functions are piece-wise linear convex [85]. The number of pieces in this piecewise-linear function can grow fast as the steps in the planning horizon increase. Figuring out efficient ways to compute the linear pieces making this function is hence important, and there have been various algorithms introduced to compute the non-dominated linear pieces efficiently [83, 63, 13, 12, 97]. After that, the value iteration algorithm from MDP can be adapted to solve POMDPs as well.

Note that the *dimensionality* of the belief space is the number of possible *states* of the robot which can be quite large. Solving a POMDP exactly is

indeed exponential in the number of states [68]. The above algorithms hence are not very efficient on moderate sized number of states. This led to various approximate POMDP approaches. One of them is the Augmented MDP [79, 80], which compresses the belief space into a more compact representation, by assuming that the belief can be summarized by a sufficient statistic. Another class of approximate POMDPs is the point-based POMDPs [73, 98], where only a small representative set from the belief space is used by the algorithm. This idea led to the point-based value iteration algorithm [72]. How to choose these representatives via sampling has also been explored in literature [84, 82, 51]. In [51] an attempt is made to sample only in the sub-space of the belief space, which is reachable by the optimal strategy. These approximate POMDPs can now solve some problems efficiently, with tens of thousands of states. However do note that uncertainty in environment (covered above), can lead to an infinite number of possible states, and hence POMDPs are not well suited for those kinds of problems as yet.

2.4 Planning under dynamic changes

We now come to planners that do not make assumption (2) of the PRMs that were given at the beginning of this chapter: that the environment is static.

In the real world, as the robot moves towards its goal configuration, the environment may change. If the motion of the obstacles is known, then we need to consider moving obstacles while planning for the path. On the other hand if the motion is unpredictable, then it may necessitate *re-planning* while executing the plan. That is, when a change is observed, we re-plan a valid

path from the current position of the robot to goal, before continuing the robot along the path to goal.

2.4.1 Planning when the motion of obstacles is known

In order for a fail-proof plan to be built before execution, the motions of the obstacles must be known. Hence we will asssume that for now. As motion planning is itself a hard problem, one expects motion planning under moving obstacles to be even harder. Indeed, in [78] it is shown that in three dimensional workspace, with arbitrary number of rotating obstacles, the problem is NP-hard if there are no restrictions on the velocity of the robot, and PSPACE-hard if the velocity modulus is bounded.

To plan under moving obstacles, one has to take time into consideration, which can be done by adding a time dimension to the configuration space of the robot. Hence planning is done in this configuration-time-space, called the CT-space. However there is one additional constraint: the path can never go backwards along the time axis, as time is irreversible.

Sampling based motion planners for static environment can be extended to plan in the CT-space; just that we have to be careful not to go backwards in the time axis. Planning when the velocity modulus of the robot is bounded, have also been considered in literature [52, 42]. The velocity-tuning method [42] is a two-phased approach. Planning is done assuming static obstacles in the first phase, and then the velocity is 'tuned' to avoid collisions in the second phase. The second phase adapts the visibility-graph method [52] for static environments. Other approaches adapt the approxi-

mate cell decomposition [52] for static environments, to problems involving moving obstacles, both when the robot's velocity modulus is unbounded, or bounded.

2.4.2 Adapting graph search methods

So far we have assumed that the motions of the obstacles is known. For the case that obstacle motions are unpredictable, we need a planning methodology that reacts to changes in the environment or to the motion of a given obstacle. As sampling based planners have been highly successful for planning under static environment, one could adapt them for the current problem as well by *re-planning* the path whenever a change is observed.

2.4.3 Rebuilding the solution if a moving obstacle invalidates the previous solution

We would need to re-establish our solution, if a moving obstacle invalidates the previously computed solution. Recall from above that some planners build a tree in the configuration space, rather than a graph to solve the query. We gave two references for that. Each of those techniques has been expanded to deal with the possibility that a moving obstacle may invalidate the solution.

In [33] the planning is performed for a robot with kinematic and/or dynamic constraints, and the obstacles are assumed to be moving with known trajectory. The planning is performed in the space of configuration $\times$ time. The methodology remains the same as constructing a tree in configuration

space (see 2.2), just that the tree is now being made in the space conf $\times$ time. If an obstacle deviates from its expected trajectory, then the planner is alerted and the path to the goal (in the space of conf $\times$ time) is recomputed. In dynamic RRT [23] if an obstacle moves, firstly those edges of the tree are discarded which are now in collision with some obstacle. Then while keeping the rest of the tree intact, the tree is regrown until a solution is again found. In [90] planning is done in small incremental steps, and the algorithm alternates between planning and execution phases. The advantage is that we do not make much assumptions about the position or trajectory of obstacles, as planning is now online.

2.4.4 Recalculating an 'optimal' path through a roadmap after dynamic changes

In section 2.4.3 the tree in C was regrown until a valid path from source to goal was re-established. However in the case of PRMs, even if a solution still exists through the roadmap after changes in the environment, we may want to recompute the *optimal* path through this graph from the start to the goal configuration; i.e. the path in the graph with the smallest cost. A naive approach would be to re-run the algorithm from scratch whenever a change in the environment is observed. A smarter methodology would allow to keep much of the formerly computed result, and only change those parts of the solution that got affected by the change in the environment.

In a general form, we want to build an extension to Dijkstra's [18] algorithm, so that if edge costs alter, the optimal path from source to goal is

recomputed in an efficient way. This has been done (for e.g. [75, 76, 27]). Early algorithms treated edge cost increases and edge cost decreases, separately, some of them performing one update at a time. In [75] batch updates could be processed. An experimental study can be found in [17] verifying that these algorithms are much more efficient than re-running the Dijkstra algorithm from scratch after changes to edge costs.

In the AI literature, the above ideas were introduced via the Dynamic A* algorithm and its variants [86, 48, 87, 24]. These algorithms keep a priority queue of nodes, just like in Dijkstra's algorithm. However, the difference is in what nodes go into the queue, and how are they prioritized. If dynamic changes to the environment are detected, the optimal path from a node x to goal may get altered, but only for a select few nodes in the graph. The attempt is therefore for only those nodes to enter the queue, whose optimal path has possibly altered. Also, the priority of a node in the queue depends on not just its current estimated cost to goal, but the minimum cost in a certain history of the node. This is because dynamic changes may have altered the cost-to-goal of the node in various ways. The Dynamic A* - lite algorithm [48] notes a consistency condition, which if satisfied at all nodes ensures that we know the shortest path tree. Hence the idea of these algorithms really is, to repair this consistency condition only at nodes where it became violated after the dynamic changes in the environment.

None of the above algorithms however consider imperfect environment maps, which is what we do in the Dynamic BURM (DBURM) algorithm (chapter 4). DBURM is closely related to the above algorithms, however DBURM considers uncertainty of the environment. Next, for efficiency DBURM

does not calculate the exact probabilities of collision along an edge, but keeps upper and lower bounds, that are refined if required. Hence edge costs are represented as intervals rather than values. This interval representation of costs, makes it much more challenging than the case in the algorithms above. Also, our local consistency condition is different to the algorithms above.

Chapter 3

Bounded Uncertainty Roadmaps

We introduce the Bounded Uncertainty Roadmap (BURM) algorithm, which extends the Probabilistic Roadmap algorithm to plan the robot path when the positions of obstacles are uncertain. Such uncertainties often arise because information about the environment, acquired via robot sensors or environment map, is not perfectly accurate. The aim is to find a path from the source configuration to destination, that has a small expected cost of collision. BURM builds a roadmap in the configuration space of the robot, and identifies the path through the roadmap that has the least expected cost of going from source to destination.

Because of uncertainties in position of the obstacles, the expected cost of a path depends on the *probabilities* of collision along the path. The main contribution of BURM is not to compute the exact probability of collision at a given robot configuration, but to keep bounds on it. Bounds can be

iteratively refined by performing more computation. Bounds at a given configuration are refined only if it help in identifying the optimal path in the roadmap. Part of this refinement is done by hierarchically dividing the geometry of the robot and obstacles into smaller pieces, giving us subdomains over which probability of collision is to be computed. Some part of the refinement is also performed by Monte Carlo Integration.

In our experimental results we compare BURM to the classic PRM; and to what we call as the Monte Carlo Uncertainty Roadmap (MCURM). MCURM does not use iterative refinement of probability bounds, hence it attempts to compute the exact probability of collision along all edges it encounters while searching for the optimal path in the roadmap. The classic PRM on the other hand, completely disregards uncertainty. Experiments show that DBURM is around $150 - 200$ times faster than MCURM, and surprisingly only around 3 times slower than PRM, which returns a much more expensive path as it disregards uncertainty.

In the following section we describe how we extend from a roadmap where edge costs are known values, to Bounded Uncertainty Roadmap, where edge-costs are bounded within known intervals. In section 3.3 we define the path cost function under such a setting. As computing the function exactly is computationally expensive, we also describe how to reduce the required computation. In section 3.4 we show how to find the optimal path from a given source to destination in a Bounded Uncertainty Roadmap. The aim is quite similar to Dijkstra's shortest path algorithm [18, 15], but now the expected costs of edges are not known exactly, and we only know that they are contained within given intervals. The intervals can be refined when needed.

The details of how to refine bounds is given in section 3.5. Finally we give the implementation details and experimental results in sections 3.6 and 3.7 respectively.

3.1 Extending Roadmaps, to Bounded Uncertainty Roadmap

Probabilistic sampling of configurations from the configuration-space of the robot, is a highly successful approach for motion planning of robots with many degrees of freedom. Sampling-based motion planning algorithms typically assume that input environments are perfectly known in advance. This assumption is reasonable in carefully engineered settings, such as robot manipulators on manufacturing assembly lines. However, as robots venture into new application domains at homes or in offices, environment maps are often acquired through sensors subject to substantial noise. It is essential for sampling-based motion planning algorithms to take into account uncertainty in the environment maps during planning so that the resulting motion plans are relevant and reliable.

In sampling-based motion planning, a robot's configuration space $\mathcal{C}$ is assumed to be known and represented implicitly by a geometric primitive $\texttt{Free}(q)$, which returns true if and only if the robot placed at q does not collide with obstacles in the environment. The main idea is to capture the connectivity of $\mathcal{C}$ in a graph, usually called a *roadmap*. The nodes of the roadmap correspond to collision-free configurations sampled randomly from

$\mathcal{C}$ according to a suitable probability distribution. There is an edge between two nodes if the straight-line path between them is collision-free.

Now suppose that the shapes or poses of obstacles in the environment are not known exactly, but are modeled as a probability distribution π_c of possible shapes and poses. Then $\texttt{Free}(q)$ cannot always determine whether q is collision-free or not: it depends on the distribution of obstacle poses and shapes. Instead of relying on $\texttt{Free}(q)$, we need to compute the *probability* that q is collision-free with respect to π_c, and instead of a usual roadmap, we construct an *uncertainty roadmap* U by annotating roadmap edges with probabilities that they are collision-free (Fig. 3.1).

After building the uncertainty roadmap, we process path planning queries by finding a path in U that has the least expected cost of traversal. The cost of traversing a path is defined in terms of its Euclidean length, as well as the number of expected collisions if the robot traverses the path. (see section 3.3).

3.1.1 From Uncertainty Roadmaps to Bounded Uncertainty Roadmaps

Unfortunately, constructing a complete uncertainty roadmap U incurs high computational cost, as it is difficult to compute collision probabilities efficiently. We use two ideas to overcome this difficulty, which we briefly explain below, and explain in greater detail in the later sections.

Firstly, observe that a path planning query may be answered without knowing the complete uncertainty roadmap. Instead of computing the ex-

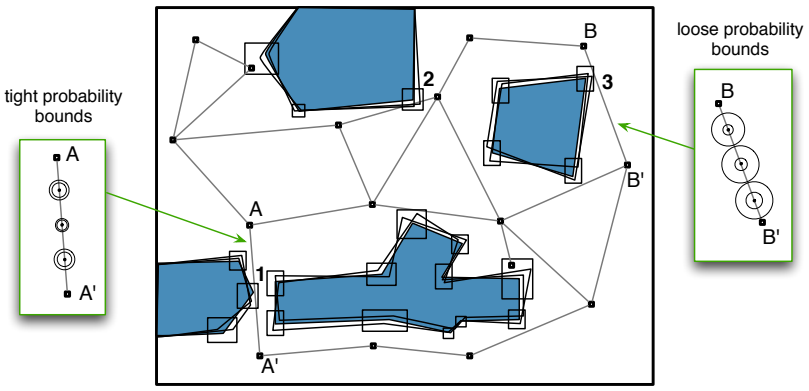


Figure 3.1: An uncertainty roadmap. The positions of polygon vertices are uncertain and modeled as probability distributions. The rectangular boxes around the vertices indicate the regions of uncertainty. In the insets, for each configuration marked along a roadmap edge, there are two circles indicating the upper and lower bounds on the probability that the configuration is collision-free. The size of the circles corresponds to the probability value. In region 1, the two circles have similar size, indicating that the probability bounds are tight.

act collision probabilities for the edges of U, we maintain upper and lower bounds on the probabilities and refine the bounds incrementally as needed. We call such a roadmap a *bounded uncertainty roadmap* (BURM). See the insets in Fig. 3.1 for an illustration. The key idea of our approach is to evaluate uncertainty, represented by the collision probability bounds, at multiple resolutions in different regions of the configuration space, depending on their relevance for finding a best path in U. Often, the critical decision of favoring one path over another depends on the uncertainty in localized regions only, for example, in narrows passages where the robot must operate in close proximity of the obstacles. It is thus sufficient to evaluate uncertainty accurately only in those regions and only to the extent necessary to choose a best path. Consider the example in Fig. 3.1. If we want to go form A to A',

it is important to evaluate the precise collision probabilities in region 1 in order to decide whether to take the risk of going through the narrow passage or to make a detour. Knowing the precise collision probabilities in regions 2 and 3 is much less relevant for this decision. Thus, evaluating collision probabilities at different resolutions hierarchically leads to drastic reduction in computation time by avoiding unnecessarily computing the exact collision probabilities.

Secondly, note that computing collision probabilities effectively requires integrating over a high-dimensional distribution π_c of obstacle shapes and poses. The *dimensionality* of π_c depends on the geometric complexity of the environment obstacles. Consider, for example, a simple two-dimensional environment consisting of 10 line segments. Each line segment is specified by its endpoints, whose positions are uncertain. We then need to integrate over a distribution of $10 \times 2 \times 2 = 40$ dimensions! To overcome this difficulty, we break down the high-dimensional integral into a series of lower-dimensional ones as we explain in section 3.3.1.

3.2 Modeling Position Uncertainty

Let us start with two-dimensional environments. The obstacles are modeled as polygonal objects, each consisting of a set of primitive geometric features—line segments for two-dimensional environments. The endpoints of the line segments may not be known precisely and are modeled as probability distributions with finite support, such as truncated Gaussians. See Fig. 3.1 for an illustration. In Fig. 3.1 the rectangles around the vertices bound the

position of the vertex with probability 1. Environment maps of this kind can be obtained by, for example, feature-based extended Kalman filtering (EKF) mapping algorithms [89]. For generality, we model the robot in exactly the same way. In three-dimensional environments, the representation is similar, but the primitive geometric feature are triangles rather than line segments.

3.3 The Path Cost Function

Given the above representation of the obstacles and the robot, we can construct an uncertainty roadmap U in the robot's configuration space $\mathcal{C}$. The nodes of U are configurations sampled at random from $\mathcal{C}$. For every pair of nodes u and u' that are close enough according to some metric, there is an edge in U, representing the straight-line path between u and u'. Recall that in classic motion planning, sampling-based algorithms construct a roadmap whose nodes and edges are guaranteed to be collision-free, and the goal is to find a collision-free path in the roadmap. In our setting, due to the uncertainty, we cannot guarantee that the nodes and edges of U are collision-free, and there may exist no path that is collision-free with probability 1. So instead, we want to find a path with minimum cost according to a suitable cost function. A cost function may incorporate various properties of the desired path. To be specific, our cost function depends on two factors: the collision probability and the path length. This allows us to trade off the distance that the robot must travel against the risk of collision.

3.3.1 The weight of an edge

Definition of W(e)

We define the weight of an edge e in the roadmap as follows:

$$W(e) = \ell(e) + \mathbf{E}[C(e)], \tag{3.1}$$

where $\ell(e)$ is the length of e and $C(e)$ is the cost of collision for e. $\mathbf{E}[C(e)]$ denotes the expected collision cost, and the expectation is taken over π_c, the probability distribution of the obstacle and robot geometry. This cost function assumes that collision is tolerable and we want to trade off the risk of collision against the robot's travel distance. Paths that do not conform to this assumption, *e.g.*, those that penetrate through the interior of obstacles, must be excluded.

Reducing Computation of $W(e)$ by (a) Breaking down the integral into a series of lower dimensional integrals

In the discussion below, we use the terminology of the geometric 'features' of robot and obstacles. For the case of 2-dimensional environment, we define 'features' as line segments belonging to the robot or obstacle. However, we remark that the discussion in the rest of this section is independent of how exactly a feature is defined.

To obtain $W(e)$, we need to calculate $\mathbf{E}[C(e)]$. Doing so directly is extremely difficult, because it involves computing the probability of collision. In order to compute the probability of collision at a given configuration, we need to integrate over π_c, a high-dimensional distribution whose dimensionality is proportional to the number of geometric features describing the

obstacles and the robot, as illustrated by the example in section 3.1.1. Furthermore, we must perform this integration for every edge of U. Instead of this, we break down the integration process into several steps.

We define $C(e)$ as:

$$C(e) = \int_{q \in e} C(q)\, dq,$$

where we slightly abuse the notation and use $C(q)$ to denote the cost of collision at configuration q. Following the usual practice in sampling-based motion planning, we discretize the edge e into a sequence of of configurations $(q_1, q_2, \ldots, q_n)$ at a fixed resolution and approximate the integral by

$$C(e) = \sum_{i=1}^{n} C(q_i)\, dq. \tag{3.2}$$

Hence by linearity of expectation we get $\mathbf{E}[C(e)] = \sum_{i=1}^{n} \mathbf{E}[C(q_i)]$

So far not much has changed as $\mathbf{E}[C(q_i)]$ involves computing the probabilities of collision, which are high dimensional integrals and difficult to compute. However, recall that the end-points of the geometric features of the polygons that make the environment and robot, are modeled as probability distributions with *finite* support. Hence we need to integrate over the poses of only those geometric features of the environment, that are close enough to the robot at q_i, to have a non-zero probability of collision with the robot. This by itself can drastically reduce the dimensionality of the integral (and summation). However, we go a step further.

Given the robot in configuration q_i, denote the two feature sets by S for the obstacles and S' for the robot. We define the collision cost of the robot at a given configuration as the sum of collision costs of all pairs of geometric

features $s \in S$ and $s' \in S'$. This can model, for example, the preference that configurations with fewer feature pairs in collision are more desirable. (See section 3.3.2 for further discussion). In formula, we have

$$C(q) = \sum_{s \in S, s' \in S'} C_{s,s'}(q), \tag{3.3}$$

where $C_{s,s'}(q)$ is the collision cost for the feature pair s and s' when the robot is placed at configuration q. Now as $\mathbf{E}[C(q)] = \sum_{s \in S, s' \in S'} \mathbf{E}[C_{s,s'}(q)]$, hence when computing the probabilities of collision, we now need to consider only a pair of geometric features at a time. In 2-dimensional workspace it implies 2 endpoints of 2 line segment; hence an 8-dimensional integral.

Combining Eqs. (3.1–3.3) and using the linearity of expectation, we get $W(e) = \ell(e) + \sum_{i=1}^{n} \sum_{s \in S, s' \in S'} \mathbf{E}[C_{s,s'}(q_i)]$. Let $I_{s,s'(q)}$ denote the event that s and s' intersect when the robot is placed at q. Then $\mathbf{E}[C_{s,s'}(q_i)] = \alpha \, \mathrm{P}(I_{s,s'(q)})$, where α is the cost of collision when a pair of features intersect. In practice, α is adjusted to reflect our willingness to take the risk of collision in order to shorten the robot's travel distance. To summarize, the weight of an edge e is given by

$$W(e) = \ell(e) + \sum_{i=1}^{n} \sum_{s \in S, s' \in S'} \alpha \, \mathrm{P}(I_{s,s'(q_i)}), \tag{3.4}$$

and the cost of a path γ in U is the sum of the weights of all edges contained in γ.

Reducing Computation of $W(e)$ by (b) Maintaining Bounds

To compute the cost of a path, each edge of an uncertainty roadmap U must carry a set of probabilities $\mathrm{P}(I_{s,s'(q_i)})$. Although s and s' are primitive geometric features of constant size, computing $\mathrm{P}(I_{s,s'(q_i)})$ exactly is still

expensive. If s and s' are both uncertain, then the computation requires integration over a distribution of 8 dimensions for two-dimensional features and 18 dimensions for three-dimensional features. To reduce the computational cost, we maintain upper and lower bounds on $\mathrm{P}(I_{s,s'(q_i)})$ rather than calculate the exact probability. Thus each edge e of U carries a set of probability bounds on $\mathrm{P}(I_{s,s'(q_i)})$ for each configuration $q_i \in e$ resulting from the discretization of e, and for each pair of features $s \in S$ and $s' \in S'$ at q_i. We call such a roadmap a *bounded uncertainty roadmap* or BURM for short. The probability bounds are refined incrementally by subdividing the integration domain hierarchically when searching for the optimal path.

3.3.2 The definition of cost - is it useful?

Above we defined the collision cost of the robot at a given configuration, as the sum of collision costs of all pairs of geometric features $s \in S$ and $s' \in S'$. But what can $C(q_i) = \sum_{i=1}^{n} \sum_{s \in S, s' \in S'} C(s, s')$ model? Firstly, as was mentioned before, this can model that configurations with fewer feature pairs in collision are more desirable. Each robot feature has an importance, and each collision of this feature with any of the obstacle-features, has a cost. However one may argue that the cost should depend on not only whether it collides, but how much of force is exerted on the robot-feature due to the collision.

We first look at our definition cost-of-collision in a somewhat more generic way. We are calculating the cost at a given configuration by breaking it down into feature-pair collisions. Therefore any cost-function that it calculates

would have the property of being expressable in terms of functions of feature-pairs. That is: $C(q) = g(\cup_{i,j} f_{i,j}(s'_i, s_j))$. Here $C(q)$ is the cost of collision at a configuration, s'_i is a feature of the robot, and s_j is a feature of the obstacle. Both g and f denote functions - hence g is a function of all the $f_{i,j}$. Any cost-model that is not expressable as above, can't be (by definition) broken down into feature-pair considerations. On the other hand if it is expressable in such a form, then although some details would vary, but we can indeed break down the computation of $C(q)$ into feature-pair consideration as we have done in this chapter.

As we said above, if $C(q) = g(\cup_{i,j} f_{i,j}(s_i, s'_j))$ then we can indeed break down the computation of $C(q)$ into feature-pair consideration. One natural idea is to use this cost function to incorporate 'force' exerted on the robot by the obstacles, as this force can indeed be broken down in the above fashion. In order to incorporate force exerted on a robot-feature due to a feature-pair collision, we can define $C(q) = \sum_{i,j} F(i,j)$, where F is the force exerted on feature s'_i of the robot due to the collision with obstacle feature s_j in the configuration q. The one question remaining is how to define $F(i,j)$ by looking at the geometry of s_i and s_j at a given configuration.

There can be a few ways to approximate $F(i,j)$ using the geometry of s_i and s'_j. One way is to define force in terms of penetration of s'_i into s_j. One definition of 'penetration' for a pair of polygons (or polytopes) in literature is: the minimum rotational and/or translational distance that would cause the robot and obstacle to become disjoint [19, 1, 96]. The algorithms in literature to compute the penetration for convex polygons are reasonably fast, all running in time less than $O(n^2)$ where n is the total number of

vertices of the two polygons. For our purpose we require the penetration distance of two line segments in 2D workspace, or of two triangles in 3D workspace, which are simpler than the generic problem, as well as have a small total number of vertices.

3.4 Path Planning with BURMs

3.4.1 Overview

Suppose that we are given the (uncertain) geometry of the obstacles and the robot in the representation described in section 3.2. Our goal is to find a minimum-cost path between a start configuration q_{s} and a goal configuration q_{g}. Conceptually, there are two steps, First, we construct a BURM U with trivial probability bounds by sampling the robot's configuration space $\mathcal{C}$. Next, we tighten up the probability bounds incrementally and search for a minimum-cost path in U.

The first step is similar to the usual sampling-based motion planning algorithms. We sample a set of configurations from $\mathcal{C}$ according to a suitable probability distribution and insert the sampled configurations along with q_{s} and q_{g} as nodes of U. We then create an edge for every pair of nodes that are sufficiently close according to some metric. We filter out those nodes and edges that are in collision. Here collision is defined with respect to the mean geometry of the obstacles and the robot, which means that the primitive geometric features representing the obstacles and the robot are all at their mean positions. The purpose of filtering is to exclude those paths that cause the robot to pass through the interior of the obstacles. It is well known that

the probability distribution for sampling $\mathcal{C}$ is crucial, and there is a lot of work on effective sampling strategies for motion planning. See [14, 34, 53] for comprehensive surveys. There is also recent work on how to adapt the sampling distribution when the environment map is uncertain. we do not address the issue of sampling strategies. BURMs can be used in combination with any of the existing sampling strategies.

In the second step, we search for a minimum-cost path in U using a variant of Dijkstra's algorithm. While Dijkstra's algorithm deals with path cost, a BURM contains only bounds on path cost. When there are two alternative paths, we may not be able to decide which one is better, as their bounds may "overlap". To resolve this, we need to refine the probability bounds in a suitable way. The details are described in section 3.5.

3.4.2 Searching for a Minimum-Cost Path

Given a BURM U, we search for a minimum-cost path in U using a variant of Dijkstra's algorithm. A sketch of the algorithm is shown in Algorithm 1. For each node u in U, we maintain the lower bound $\underline{K}(u)$ and upper bound $\overline{K}(u)$ on the minimum-cost path from q_{s} to u. Recall that every edge e of U carries a set of probability bounds on $I_{s,s'(q_i)}$, for every $q_i \in e$ resulting from the discretization of e and every feature pair $s \in S$ and $s' \in S'$. Let $\underline{\mathrm{P}}(I_{s,s'(q_i)})$ and $\overline{\mathrm{P}}(I_{s,s'(q_i)})$ denote the lower and upper bounds on the probability of $I_{s,s'(q_i)}$, respectively. Using these bounds, we can calculate the lower bound $\underline{W}(e)$ and upper bound $\overline{W}(e)$ on the edge weight for each edge $e \in U$. By the definition, $\underline{K}(u)$ and $\overline{K}(u)$ can then be obtained by summing up the bounds on the edge weights. Like Dijkstra's algorithm, we insert each node

Algorithm 1 Searching for a minimum-cost path in a BURM.

1: For every node u of a BURM U, initialize the lower and upper bounds on the cost of the minimum-cost path from q_s to u: $\underline{K}(u) = 0, \overline{K}(u) = 0$ if $u = q_\mathrm{s}$, and $\underline{K}(u) = -\infty, \overline{K}(u) = +\infty$ otherwise.
2: Insert all nodes of U into a priority queue Q_u.
3: **while** Q_u is not empty **do**
4: Find in Q_u a node u such that $\overline{K}(u) \leq \underline{K}(v)$ for all $v \in Q_u$, where $v \neq u$. Remove u from Q_u.
5: **if** u = q_g **then return**.
6: **for** every node v incident to u **do**
7: Discretize the edge between u and v at a given resolution into a sequence of configurations $q_i, i = 1, 2, \ldots$.
8: For every q_i, invoke `FindColEvents`(q_i, S, S') to find feature pairs $s \in S$ and $s' \in S'$ that are likely to have $\mathrm{P}(I_{s,s'(q_i)}) > 0$. For each such feature pair, set $\underline{\mathrm{P}}(I_{s,s'(q_i)}) = 0$ and $\overline{\mathrm{P}}(I_{s,s'(q_i)}) = 1$, and insert $I_{s,s'(q_i)}$ into Q_e.
9: Set $\underline{K}_u(v) = \underline{K}(u) + \underline{W}(u, v)$ and $\overline{K}_u(v) = \overline{K}(u) + \overline{W}(u, v)$.
10: **while** the two intervals $(\underline{K}_u(v), \overline{K}_u(v))$ and $(\underline{K}(v), \overline{K}(v))$ overlap **do**
11: `RefineProbBounds`$(Q_\mathrm{e}, U, q_\mathrm{s}, u, v)$.
12: **if** $\overline{K}_u(v) < \underline{K}(v)$ **then**
13: Set $\underline{K}(v) = \underline{K}_u(v)$ and $\overline{K}(v) = \overline{K}_u(v)$.
14: Update Q_u, using the new bounds on the cost of the minimum-cost path to v. Call `RefineProbBounds` if needed.

u of U into a priority queue Q_u, which is implemented as a heap, and then dequeue them one by one until a minimum-cost path to q_g is found. As we do not know the exact values of expected edge costs, we can not use the path cost from q_s to u as the priority value of u. Instead we use the path cost bounds $(\underline{K}(u), \overline{K}(u))$ as the priority key. Given two nodes u and v, we say that u has a higher priority if $(\overline{K}(u) \leq \underline{K}(v))$. Now this is only possible if the bound intervals $(\underline{K}(u), \overline{K}(u))$ and $(\underline{K}(v), \overline{K}(v))$ do not overlap. Hence in our implementation of Q_u, if u and v are being prioritized, and their bound intervals overlap, then the probabilities of collision along the two paths are

refined until the bounds no longer overlap. At such time we can establish which node has a higher priority. With such a definition of priority, the node u with the highest priority in the heap, would satisfy $\overline{K}(u) \leq \underline{K}(v)$ for all $v \in Q_u$ where $v \neq u$, which is the condition on line 4.

As we have mentioned in section 3.3.1, computing collision probabilities requires integration over a high-dimensional distribution and is very expensive computationally. We use two techniques for efficient computation of the probability bounds. In line 8 of Algorithm 1, `FindColEvents` find feature pairs $s \in S$ and $s' \in S'$ such that s and s' are likely to intersect with non-zero probability, when the robot is placed at q_i (see section 3.5.1 for details). For each such feature pair, we attach an initial probability bound of $[0, 1]$ to the event $I_{s,s'(q_i)}$ and insert $I_{s,s'(q_i)}$ into a set Q_{e} as a candidate for probability bound refinement in the future. Observe that usually, at each configuration, only a small number of feature pairs are in close proximity and likely to intersect with non-zero probability. Therefore, this step drastically reduce the number of collision probability bounds that need to be calculated. To search for the intersecting feature pairs efficiently, we exploit a hierarchical representation of the geometry of the obstacles and the robot (see section 3.5.2) and quickly eliminate most of the feature pairs that are guaranteed to have zero collision probability.

In lines 11 and 14 of Algorithm 1, `RefineProbBounds` refines probability bounds. While searching for a minimum-cost path in U, we may encounter two paths γ and γ' and must decide which one has lower cost. If the bound intervals on the cost of γ and γ' overlap, refinement of the probability bounds becomes necessary. To do so, we find all events $I_{s,s'(q)}$ in Q_{e} such that q lies

in an edge along γ or γ'. We then refine the probability bounds on these events (see Section 3.5.2), until we can determine the path with lower cost.

To focus on the main issue and keep the presentation simple, Algorithm 1 uses Dijkstra's algorithm for graph search. Informed search, such as the A* algorithm with an admissible heuristic function, is likely to give better results. In our case, one possible heuristic function is the Euclidean distance between two configurations.

3.5 Refining the probability bounds of collision between feature pairs

In this section we describe our computation of probability bounds. We restrict ourselves to the two-dimensional case. The basic idea generalizes to the three-dimensional case in a straightforward way, but the details are more involved.

3.5.1 The `FindColEvents` routine

For the two-dimensional case, `FindColEvents` (Algorithm 1, line 8) finds feature pairs $s \in S$ and $s' \in S'$ that are likely to have intersection probability $\mathrm{P}(I_{s,s'(q)}) > 0$. Here S is the set of features of the obstacles, and S' is the set of features of the robot, when the robot is placed at configuration q. As in 2-D we define features to be line segments, s and s' are line segments.

`FindColEvents` quickly eliminates most of the line segment pairs where $\mathrm{P}(I_{s,s'(q)}) = 0$. Recall that we model the endpoints of these line segments

as probability distributions with finite support. Without loss of generality, we assume that the support regions are rectangular. Let $R(s)$ denote the endpoint regions for a line segment s and $H(s)$ denote the convex hull of the endpoint regions. Now if $H(s) \cap H(s') = \emptyset$, then clearly, the probability of collision of s and s' is 0. See Fig. 3.2(a) for an illustration.

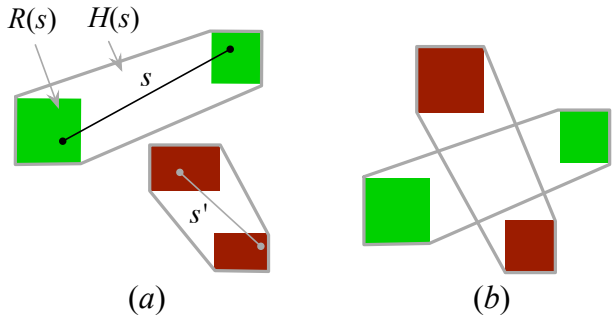


Figure 3.2: The line segment pair s and s' intersect with (a) probability 0 and (b) probability 1. (See also theorem 3.5.1)

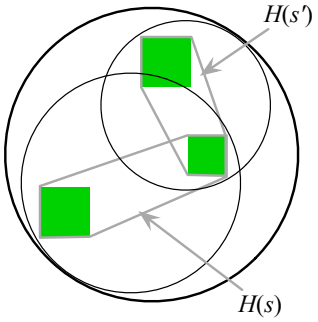


Figure 3.3: A sphere tree over two uncertain line segments.

If S and S' contain m and n line segments, respectively, it takes $O(mn)$ time to check all line segment pairs $s \in S$ and $s' \in S'$. This is very time-consuming, as we need to invoke `FindColEvents` repeatedly at many configurations. To

improve efficiency, we apply a well-known technique from the collision detection literature and build bounding volume hierarchies over the geometry of the obstacles and the robot. There are many different types of bounding volume hierarchies. See [57] for a survey. We have chosen the sphere tree hierarchy, though other hierarchies, such as the oriented bounding box (OBB) tree, can be used as well. Specifically, we build two sphere trees for S' and S, respectively. Each leaf of a sphere tree contains the convex hull $H(s)$ for a line segment s, and each internal node v contains a sphere that encloses the geometric objects in the children of v (Fig. 3.3). Clearly $H(s)$ and $H(s')$ can intersect only if their enclosing sphere intersect. It then follows that s and s' intersect with non-zero probability only if the spheres enclosing $H(s)$ and $H(s')$ intersect. By traversing the sphere trees hierarchically, we can quickly eliminate most of the line segment pairs that have zero intersection probability and reduce the cost of checking a quadratic number of line segment pairs to a much smaller number. We omit the details of constructing sphere tree hierarchies and traversing them for collision detection, as they are well documented elsewhere (see, *e.g.*, [57]).

In summary, by exploiting a hierarchical representation, `FindColEvents` efficiently identifies most line segment pairs with intersection probability 0 and reduce the trivial probability bound of $[0, 1]$ to $[0, 0]$ for all of them together. For the remaining line segment pairs, which are usually small in number, their probability bounds are further refined when necessary (see next subsection).

3.5.2 Hierarchical Refinement of Collision Probability Bounds

We now consider the problem of refining the bounds on the intersection probability $\mathrm{P}(I_{s,s'(q)})$ for some line segment pair $s \in S$ and $s' \in S'$. To simplify the notation, we will omit the parameter q and assume that s' is translated and rotated suitably.

Computing $\mathrm{P}(I_{s,s'})$ is in essence an integration problem. Let x_1 and x_2 be the endpoints of s, and let x_3 and x_4 be the endpoints of s'. Suppose that x_i has probability density function $f_i(x_i)$ with rectangular support regions R_i. We can calculate the probability that s and s' intersect by integrating over $R_1 \times \cdots \times R_4$:

$$\mathrm{P}(I_{s,s'}) = \int_{R_1 \times \cdots \times R_4} A(x_1, \ldots, x_4)\ f_1(x_1)dx_1 \cdots f_4(x_4)dx_4, \qquad (3.5)$$

where $A(x_1, \ldots, x_4)$ is an index function that is 1 if and only if s and s' intersect. In two-dimensional environments, this integral is 8-dimensional.

To evaluate this integral, we decompose the integration domain $R_1 \times \cdots \times R_4$ hierarchically into a set of subdomains such that in each subdomain, the index function A is constant. By summing up the probability mass associated with all the subdomains where A is 1, we get the value for $\mathrm{P}(I_{s,s'})$. During the hierarchical decomposition process, we maintain three lists of subdomains: (i) subdomains where A is always 1, (ii) subdomains where A is always 0, and (iii) subdomains where A has mixed values (0 or 1). Interestingly, these three lists provide an upper bound and a lower bound on $\mathrm{P}(I_{s,s'})$ at any moment during the decomposition process. Let p_1 and p_2 be the probability mass

associated with subdomains in list (i) and (ii), respectively. Clearly, we have $p_1 \leq \mathrm{P}(I_{s,s'}) \leq 1 - p_2$. The probability mass associated with subdomains in list (iii) is $1 - p_1 - p_2$. It represents the gap between the upper and lower bounds. To refine the bounds, we simply take a subdomain from list (iii) and decompose it further until some of the refined subdomains can be assigned to either list (i) or list (ii).

To decompose an integration domain $R_1 \times \cdots \times R_4$, we take a horizontal or vertical cut on one or more of the endpoint regions R_i and obtain a set of subdomains $R_1' \times \cdots \times R_4'$ such that R_i' is rectangular and $R_i' \subseteq R_i$ for $i = 1, 2, 3, 4$. Using Theorem 3.5.1 below, we can easily determine whether the index function A has constant value on a subdomain and assign the subdomain to the appropriate list.

Theorem 3.5.1 *Let s and s' denote two line segments with uncertain endpoint positions in two dimensions. For a line segment s, let $H(s)$ denote the convex hull of the end point regions. Let $R(s)$ and $R(s')$ be the rectangular end point regions.*

(1) *If $H(s) \cap H(s') = \emptyset$, then s and s' intersect with probability* 0.

(2) *If $H(s) \cap H(s') \neq \emptyset$, $R(s) \cap H(s') = \emptyset$, and $R(s') \cap H(s) = \emptyset$, then s and s' intersect with probability* 1

Proof To prove part (1), observe that since $H(s) \cap H(s') = \emptyset$, line segments s and s' has no intersection for all pairs s and s' whose endpoints lie in $R(s)$ and $R(s')$, respectively. This immediately implies that the intersection probability is 0.

Now we prove part (2). Let R_1, R_2, R_3, R_4 be the four end point regions (see fig. 3.4). Let's first assume that there is a pair of intersecting line segments s_1 and s_2, where the end-points of s_1 lie in R_1 and R_2, and those of s_2 lie in R_3 and R_4 as shown.

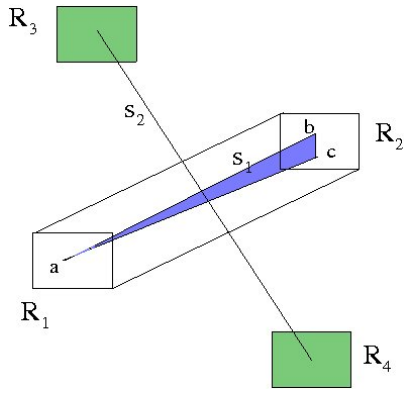


Figure 3.4: Two colliding segments, when $R(s) \cap H(s') = \emptyset$, and $R(s') \cap H(s) = \emptyset$. $R(s)$ are the end point regions R_1 and R_2. H(s) is their convex hull. Also, R(s') are the end point regions R_3 and R_4. H(s') is their convex hull

Let the two vertices of s_1 be labelled as a and b as shown in the figure. Next, keeping a constant, choose any other vertex c as shown. We will now show that s_2 also intersects with ac.

The three points a, b, c define a triangle(we include the possibility of a degenerate triangle). Note that s_2 enters this triangle by intersecting ab, but s_2 also needs to exit this triangle. Else $R(s')$ would be intersecting abc, which contradicts the hypothesis that $R(s') \cap H(s) = \emptyset$. Note now that s_2 can not intersect with the segment bc, else R_2 would intersect with $H(s')$ Hence we conclude that s_2 intersects with the only remaining edge of the triangle, ac.

But c was chosen arbitrarily from R_2, hence showing that s_2 intersects with *all* line segments az, where $z \in R_2$. For each z, we can reverse the argument treating z as we treated a above, to conclude that s_1 intersects with *all* segments s_i where the end points of s_i lie in R_1 and R_2.

So we conclude, that if s_2 intersects one line segment s_1, then it intersects *all* line segments s_i as defined above. But now we can reverse the roles of s and s' in this argument. Hence, as every line segment s_i intersects with s_2, *each* s_i intersects with *all* line segments s_j, where the end points of s_j lie in R_3 and R_4. This then shows that if there is one pair of intersecting segments, then the probability of intersection is 1.

All that needs to be done is to show that such a pair of segments indeed exists. But that is straightforward. Let p be a point at the boundary of the intersection of the two convex hulls. p must be a point that belongs to the boundary of $H(s)$ as well as the boundary of $H(s')$. Now consider the boundary line segment of $H(s)$ that contains p, and the boundary line segment of $H(s')$ that contains p. These two boundary line-segments hence are the pair of segments we wanted. ∎

There are various strategies to decompose a rectangular integration domain. The main goal is to assign each subdomain to list (i) or (ii) and avoid unnecessarily decomposing into a large number of tiny subdomains. One possible decomposition procedure, based on the quadtree [16], always cuts an endpoint region in the middle either horizontally or vertically (Fig. 3.5a). This is simple to implement, but does not always results in the best decomposition, as it may unnecessarily cut a domain into small pieces. Our

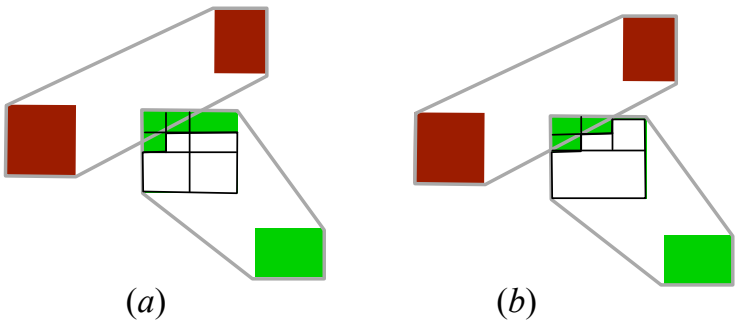


Figure 3.5: Decomposing the integration domain for intersection probability calculation. (a) The quadtree-based procedure. (b) Our procedure that takes into account the geometry of intersecting line segments. The example shows that after roughly a same number of cuts, our procedure identifies a large part of the domain for which no further decomposition is needed.

decomposition procedure uses the geometry of the two intersecting line segments s and s' to decide where to cut. This results in better decomposition, but the trade-off is that each decomposition step is slightly more expensive. To determine how to cut, our procedure enumerates several cases that depend on the relative positions of the endpoint regions and convex hulls for s and s'. (See 3.6 for more details.) An example decomposition is shown in Fig. 3.5b. In the experiments, our decomposition procedure usually gives slightly better performance than the quadtree-based procedure.

An alternative way of evaluating the integral in (3.5) is to perform Monte Carlo integration [41] by sampling from the integration domain $R_1 \times \cdots \times R_4$. Each sample consists of four points $x_1, \ldots, x_4$ with $x_i \in R_i$. Let A_i be the value of the index function A for the ith sample, and p be the value of the

integral in (3.5). Then an estimate of p is given by

$$p_N = \frac{1}{N} \sum_{i=1}^{N} A_i. \tag{3.6}$$

The values $A_i, i = 1, 2, \ldots, N$ are in fact a set of independent and identically distributed (i.i.d.) random variables. Under a wide range of sampling distributions, the mean of A_i is equal to p. Let V_A be the variance of A_i. Note that A_i has a Bernoulli distribution, and therefore $V_A \leq 0.25$. By (3.6), p_N is a random variable with mean p and variance V_A/N. We can then apply Chebychev's inequality and obtain

$$\mathrm{P}\left(|p_N - p| \geq (1/\delta)(V_A/N)^{-1/2}\right) \leq \delta, \tag{3.7}$$

which implies that p_N converges to p at the rate $O(N^{-1/2})$. More precisely, for any δ arbitrarily small, we can determine the number of samples, N, needed to ensure that the estimate p_N does not deviate too much from p. So instead of maintaining upper and lower bounds on the collision probabilities, we can choose N large enough to get sufficiently accurate estimates for all the collision probabilities and find a minimum-cost path with high probability. Unfortunately, using Monte Carlo integration this way is not efficient (see Section 3.7), as it uses the same number of samples for estimation everywhere.

An interesting method is to combine probability bound refinement and Monte Carlo integration. We start by decomposing the integration domain as described earlier. When the probability mass associated with a subdomain is small enough, we apply the Monte Carlo method with a small number of samples to get an estimate and close the gap between the upper and lower

bounds. Strictly speaking, if we do this, we cannot guarantee that the algorithm finds a minimum-cost path in U. However, if we use a sufficient number of samples for Monte Carlo integration, we can provide the guarantee with high probability. Furthermore, even when the algorithm fails to find a minimum-cost path, the cost of the resulting path is still a good approximation to the minimum cost. The reason is that due to the bound in (3.7), we make a mistake only when two paths have very similar cost. We use this combined method in our implementation of the algorithm, and it achieves better performance better than one that uses pure probability bound refinement.

3.6 Implementation Details

3.6.1 Mixing Monte-Carlo Integration and Hierarchical Refinement

In our implementation of BURM, we mixed Monte Carlo integration, and hierarchical refinement of bounds as we now explain. Suppose we are refining the bounds on the probability of intersection between the segments s and s'. Initially the bounds are trivial: [0,1]. We refine the bounds by dividing the integral into subdomains, as explained below. However, if the subdomain has a probability mass of $m \leq 0.4$, then we do not perform any more hierarchical refinements, but perform $m \times 100$ Monte Carlo simulations to close the gap on the probability bounds. The proportion of samples that were in collision, is then taken to be the probability of collision over the subdomain.

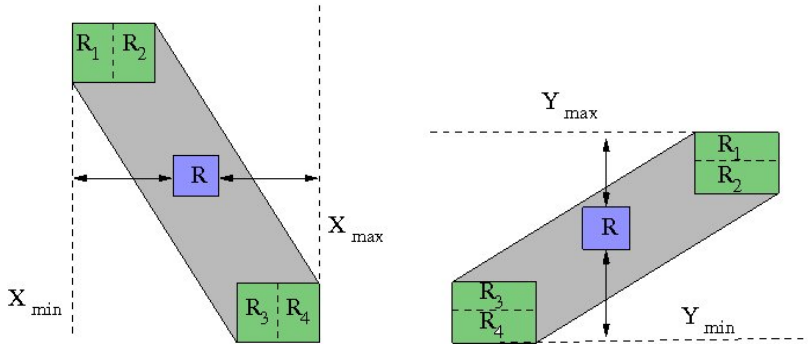
3.6.2 Details of Hierarchical Refinement

We put this in implementation details, as there are various possible hierarchical refinements. We implemented one, and do not claim it's the best. Recall that we perform hierarchial refinement only if the probability of collision is neither 0 nor 1. Therefore we are going to assume that for the rest of this section.

Geometrically speaking, hierarchical refinement concerns a pair of hulls of the end point regions (see fig. 3.2 and theorem 3.5.1). We attempt to divide the rectangles in such a way, that some of the subdomains have probability of collision 0, or probability of collision 1. The idea is to make individual checks cheap to compute, and yet having reasonable heuristics, so that within a few divisions we can get to subdomains with probability of collision 0 or 1 as shown in fig. 3.2.

We break it down into three cases. First suppose that some end-point region R is entirely contained within the hull of the other segment as shown in fig. 3.6. We are showing just 3 of the 4, end point regions. In figure (a) we show the distances from R to x_{min} and x_{max}. Denote the smaller of these distances by x_{dist}. We define y_{dist} similarly. If x_{dist} is smaller than y_{dist}, which is the case in (a), then we divide the two end point regions into R_1, R_2, R_3 and R_4 as shown. Note that this would give us four subdomains: R_1R_3 , R_1R_4 , R_2R_3 and R_2R_4. In case (b) y_{dist} is smaller and in this case we divide along the horizontal axis as shown.

Note that to reduce the combinatorial explosion of the number of subdomains, we divide only *one* of the rectangles as shown above, and do not



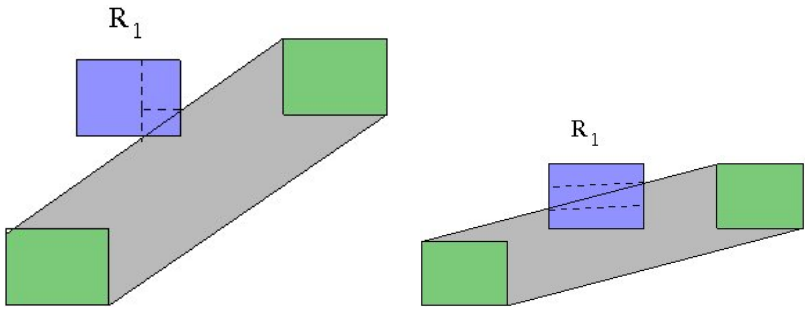
(*a*) Case 1: $x_{dist} \leq y_{dist}$ (*b*) Case 2: $x_{dist} > y_{dist}$

Figure 3.6: One of the end point regions is contained in the convex hull

divide any other rectangles. If the above case is not true for any of the four rectangles then we move onto second case as shown in the figure below. Here one of the rectangles intersects with a line segment that does *not* belong to any other end point region as shown in fig. 3.7. The intersecting rectangle is divided into 3 domains along the dotted lines as shown. Note that in both the examples shown below, one of the subdomains lies outside the convex hull, which should help in quickly refining the probability using theorem 3.5.1. Note that the examples do not cover the case, when the intersecting segment is along one of the diagonals of the rectangle. In this third case (not shown in figure), we just divide the rectangle into 4 equal subdomains, each having half the width and height of the original rectangle.

If both the above cases are false, then there must be two intersecting rectangles. In this case, we just perform $100 \times m$ Monte Carlo simulations to close the gap on the probability bounds. Here m is the probability mass of the four rectangles; that is, the probability that the 4 vertices would be in the 4 rectangles. Note that m may not be 1, as we might have reached this case

after some hierarchical subdivisions. The probability of collision assigned to this mass is then $mn_{coll}/100$, where n_{coll} is the number of samples in which the two line segments intersected.



(*a*) Case 1 (*b*) Case 2

Figure 3.7: End point region intersecting with a segment that does *not* belong to any other end point region

3.7 Experiments

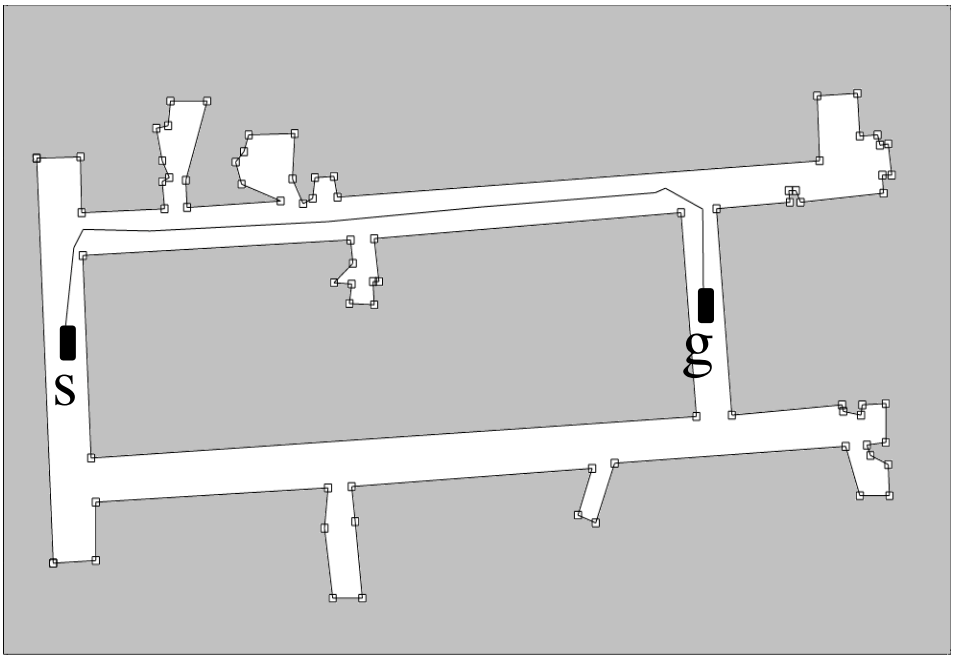
To test the effectiveness of our approach, we compared the performance of our algorithm and with two alternatives. One algorithm, MCURM, is similar to ours. It also builds an uncertainty roadmap. However, instead of refining the probability bounds incrementally when necessary, it estimates the exact probabilities using Monte Carlo integration. In our tests, MCURM uses 100 samples to evaluate each intersection probability $\mathrm{P}(I_{s,s(q)})$. The other algorithm that we compared is Lazy-PRM [6], which does not take into account uncertainty during planning.

In our tests, all three algorithms use the same sampling strategy, which is a hybrid strategy consisting of the bridge test and the uniform sampler [32]. We ran the algorithms on each test case and repeated 30 times independently.

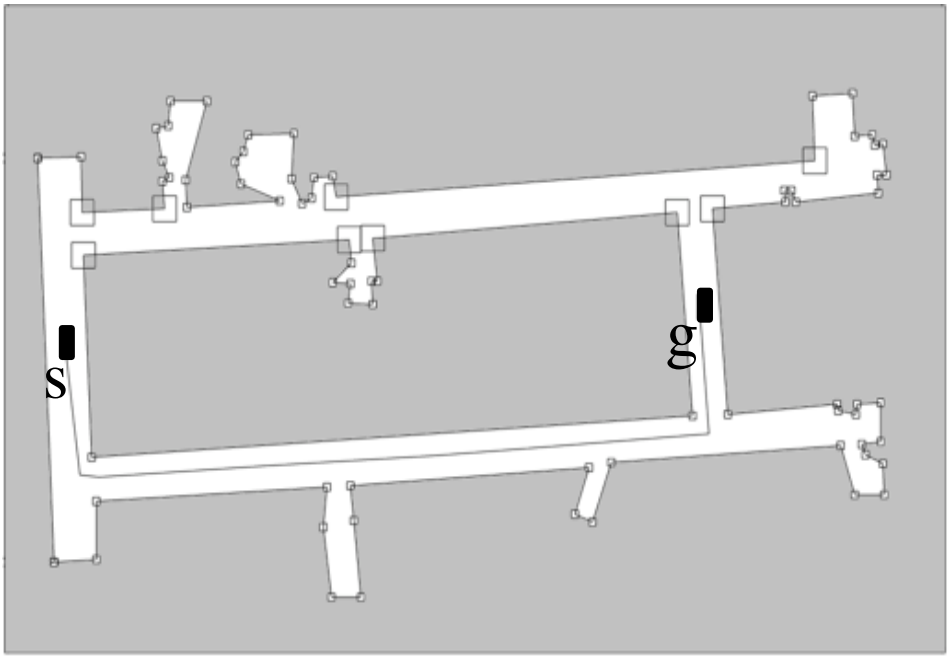
The performance statistics reported here are the averages of 30 runs. In each run, the three algorithms used the same set of sampled configurations. So the performance difference results from the way they find a minimum-cost path in an (uncertainty) roadmap rather than random variations in sampling the configuration space.

In the figures below the boxes around the vertices mark the support regions of the probability distributions modeling the endpoint positions of line segments forming the obstacle boundaries.

The test results are shown in figures 3.8 , 3.9 and Table 3.1. In test environments 1–3, the robot has a rectangular shape and only translates. In these three tests, the environments are similar. The robot essentially chooses between two corridors to go from the start to the goal position. The main differences among the tests are (i) the level of uncertainty in the obstacle geometry and (ii) the robot start position. In test environment 1 figure (fig 3.8(a)), the uncertainty level is low and roughly the same everywhere. So the robot chooses the upper corridor, based mainly on the path length consideration. However, it is interesting to observe that although a shortest path with respect to the path length normally touches obstacle boundaries, our minimal-cost path stays roughly in the middle of the corridor. It does so to avoid collision due to the uncertainty in obstacle geometry. In test environment 2, the upper corridor has substantially higher uncertainty than the lower corridor. On balance, it is better for the robot to choose the slightly longer, but safer lower corridor (fig 3.8(b)). In test environment 3, the uncertainty in obstacle geometry remains the same as that in test environment 2,

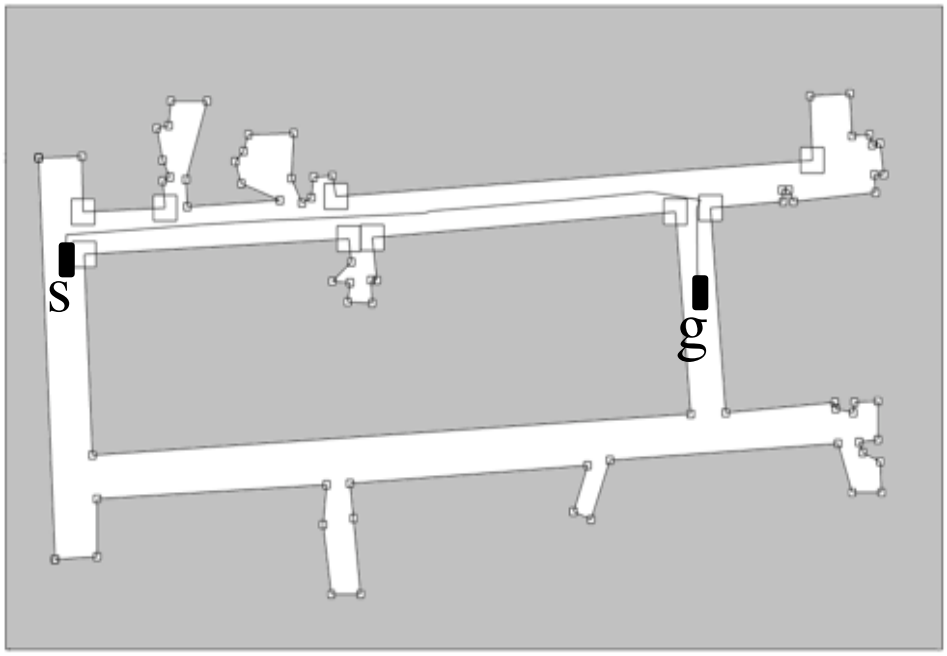


(*a*) Test environment 1.

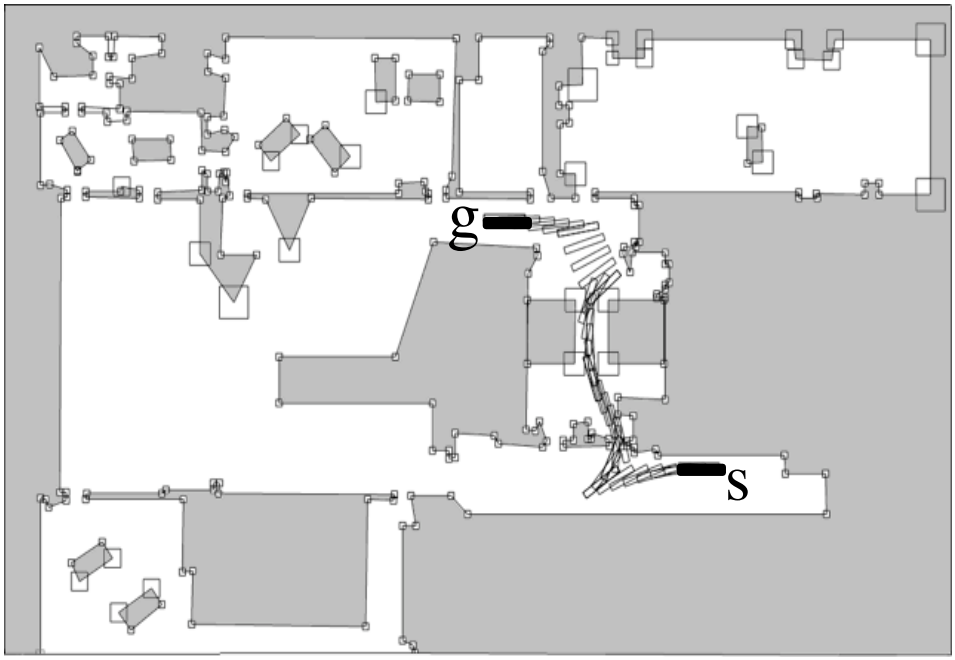


(*b*) Test environment 2.

Figure 3.8: Test Environments 1 and 2



(*a*) Test environment 3.



(*b*) Test environment 4.

Figure 3.9: Test Environments 3 and 4

Table 3.1: Performance statistics.

Test Env.	No. Nodes	Cost			Time (s)		
		BURM	MCURM	Lazy-PRM	BURM	MCURM	Lazy-PRM
1	300	700	699	786	15.1	3,132	5.9
2	300	742	742	1,063	19.3	2,893	5.9
3	300	761	760	961	19.4	3,016	5.1
4	500	544	542	706	18.6	2,878	9.4

but the start position for the robot moves higher. The robot again decides to go through the upper corridor, because despite the higher collision risk of the upper corridor, it is much shorter than the lower corridor (fig 3.9(a)).

In test environment 4, the robot can both translate and rotate. To reach its goal, the robot can either take the risk of collision and squeeze through the narrow passage or make a long detour. It is not obvious which choice is better. The answer depends, of course, on the cost of collision. In this case, the robot decides to take the riskier, but shorter path (fig 3.9(a)). Interestingly, our algorithm finds two paths of similar cost, depending on the set of sampled configurations. One path veers to the right (fig 3.9(b)) when it approaches the obstacle near the lower entrance to the narrow passage, and the other veers to the left. This is in fact not surprising, because regardless of whether the path veers to the the left or right, the uncertainty that the path encounters remains similar and the path length does not differ by much.

Now let us look at the performance statistics. For each test case, Table 3.1 lists the number of nodes in the (uncertainty) roadmap, the running times, and the cost of the paths found by the three algorithms.

BURM and MCURM find paths with almost the same cost. BURM is,

however, 140–200 times faster. This clearly demonstrates the advantage of the approach of evaluating uncertainty hierarchically at multiple resolutions.

The comparison between BURM and Lazy-PRM is even more interesting. As expected, BURM finds paths with lower cost as it takes uncertainty into account during planning. However, it is somewhat surprising that BURM is not much slower than Lazy-PRM: BURM is only 2-3 times slower than Lazy-PRM, while it is at least 140 times faster than MCURM. The reason is that uncertainty comes into play in deciding the best path when the robot operates in close proximity of the obstacles. This usually happens in localized regions of the configuration space only. BURM takes advantage of this by maintaining bounds on collision probabilities rather than calculating the exact probabilities. It refines these bounds incrementally by exploiting bounding volume hierarchies built over the geometry of the obstacles and the robot and by hierarchically decomposing the integration domain for collision probability calculation. The running time comparison with Lazy-PRM provides further evidence on the advantage of our approach.

To better understand the behavior of BURM, we applied it to an environment similar to that in tests 1-3 and varied the uncertainty level in the obstacle geometry. The resulting bounded uncertainty roadmaps are shown in Fig. 3.10 (at the end of the chapter). The edges of the roadmaps are colored to indicate how tight the associated collision probability bounds are. The roadmap in Fig. 3.10a serves as a reference point for comparison. The uncertainty is low in both the upper and lower corridors, and the collision probability bounds are refined to various degrees. As the uncertainty gets higher in the upper corridor, the collision probability bounds there are tight-

ened to differentiate the quality of the paths (Fig. 3.10b). It is also interesting to observe that the upper corridor is not explored as much, because the paths in the lower corridor are far better. Finally, as the uncertainty in the lower corridor also increases, both corridors must be explored, and the collision probability carefully tightened in order to determine the best path.

Bounded uncertainty roadmaps can be used with any existing sampling strategies. To demonstrate this, we performed additional tests by varying the sampling strategy used and the number of nodes in the uncertainty roadmap. We tried two additional strategies: the uniform sampler and the Gaussian sampler [7]. For all sampling strategies, as the roadmap size increases, the running time increases correspondingly. Two plots of representative results are shown in Fig. 3.11 (at the end of the chapter). They indicate that the cost of the minimum-cost path found decreases with the roadmap size up to a certain point and then stabilizes. So one way of minimizing the path cost is to run our algorithm in an "anytime" fashion by gradually adding more nodes to the roadmap. The effect of different sampling strategies is more pronounced in more complex environments. In test environment 2, when the number of roadmap nodes is sufficiently large, the results obtained by the different sampling strategies are comparable. When the number of nodes is small, the Gaussian sampler does not behave very well, as it biases sampling towards the obstacle boundaries, resulting in high collision cost. In the more complex test environment 4, which contains several narrow passages, the hybrid bridge test and the Gaussian sampler have clear advantages over the uniform sampler. Just as in classic motion planning, effective sampling strategies for constructing uncertainty roadmaps are important and require

further investigation.

3.8 Summary

In this chapter we introduced the notion of a bounded uncertainty roadmap and used it to extend sampling-based algorithms for planning under uncertainty in environment maps. BURM builds a roadmap and then finds the optimal path through the roadmap, like the classical PRM. However due to the uncertainty, the cost of an edge now depends on *probabilities* of collision.

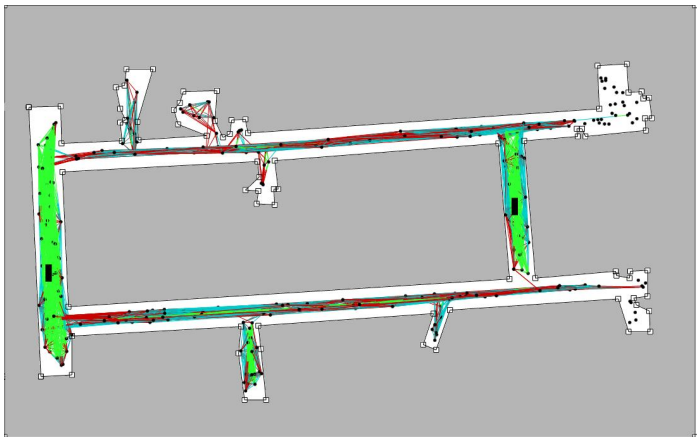
In order to evaluate the probabilities of collision we need to evaluate a high dimensional integral the dimensionality of which is equal to the number of vertices in the envrionment and robot. The main idea is to make this computation much more efficient. It is performed by firstly breaking down the high dimensional integral into a series of low dimensional integral. And then, we maintain *bounds* on the probabilities of collisions, rather than computing the exact values. The bounds are refined only if it helps in identifying the optimal path. Some part of this refinement is achieved by hierarchically dividing the geometry of the robot and environment into subdomains, over which probability of collision is to be computed. While some of the refinements use Monte Carlo Integration.

The cost of collision at a configuration is defined as the sum of the cost of feature-pair collisions. Here a feature pair consists of a colliding pair of features, one from the robot and the other from the obstacle. Hence when calculating probabilities BURM needs to consider only a pair of features at

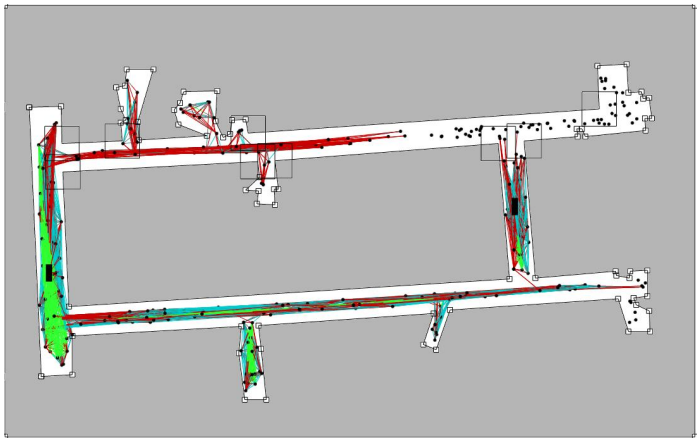
a time, greatly reducing the dimensionality of the integral. We remark that for as long as the cost of collision at a configuration, is defined in such a way so that it can be expressed in terms of feature-pair collision, the basic idea of considering only one pair of features at a time would work.

Given a feature pair, the bounds on the probability of collision are hierarchically refined by geometric considerations. The probability of collision is computed by dividing the integration domain into subdomains. This division is done in such a way so as to make it likely that the probabilities of collision in some of these subdomains would be trivial; i.e. either 0 or 1. This makes it possible to close the gap on the probability bounds. After some iterations of this geometric division, the rest of the gap is closed by Monte Carlo integration.

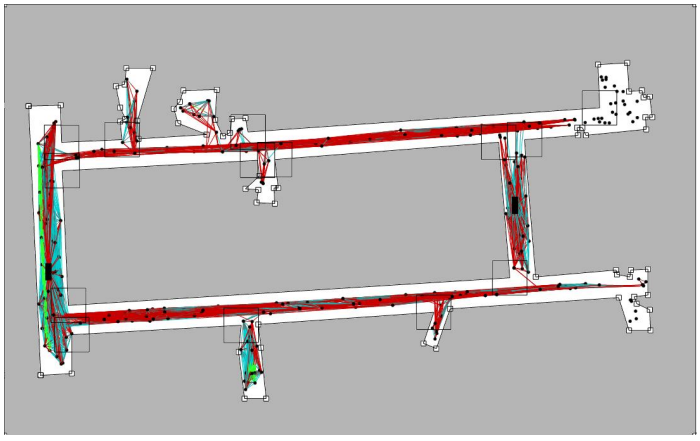
Our approach greatly improves planning efficiency. Experimental results have demonstrated that it is highly effective, and up to a 150-200 times better than MCURM, which uses Monte Carlo integration to compute probabilities of collision for every feature pair, for the robot configurations along the edges that are explored when searching for the optimal path. Also, BURM is only around 3 times slower than the classical PRM, which does not consider uncertainty at all and hence returns a much more inferior path in terms of cost-of-collision.



(*a*)

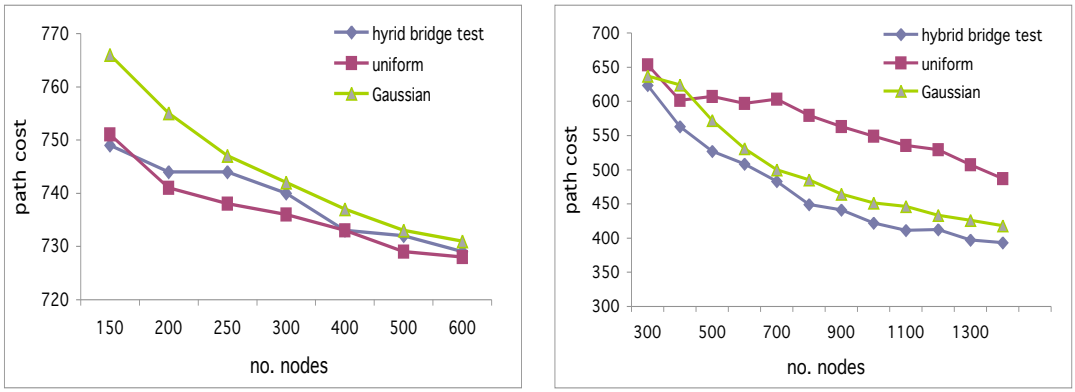


(*b*)



(*c*)

Figure 3.10: Color-coded BURMs. Red, gray, and blue-green marks edges with tight, intermediate, and loose collision probability bounds, respectively. Bright green marks edges with collision probability 0.



(*a*) Test environment 2. (*b*) Test environment 4.

Figure 3.11: The change in the cost of the minimum-cost path found as a function of the sampling strategy and the number of nodes in the uncertainty roadmap.

Chapter 4

The Dynamic BURM: Path Planning in a Dynamic Environment with an Imperfect Map

Probababilistically sampling a robot's configuration space is a powerful approach for motion planning of robots with many degrees of freedom. However, sampling-based motion planning algorithms often assume a known static environment, as well as perfect robot control and sensing. In practice, an environment map constructed from sensor data is often inaccurate, and the environment changes over time. It is essential for motion planner algorithms to take into account such environment uncertainties and generate plans that are robust during the execution.

To deal with inaccurate environment maps, the previous chapter intro-

duced the Bounded Uncertainty Roadmap (BURM). Like most sample based planners, BURM builds a roadmap in the robot's configuration space, and finds the optimal path in the roadmap. For efficiency, BURM does not calculate expected costs of edges exactly, but maintains bounds on them. These bounds can be refined with more computation.

In this chapter, we introduce the Dynamic BURM algorithm (DBURM), which extends BURM to a dynamic environment: The obstacles can change position and shape over time. Hence DBURM builds a roadmap and identifies the optimal path in it, when the information about the position and shapes of obstacles has uncertainty, as well as these position and shapes can change over time. These changes are modeled by changing the probability distributions that specify the coordinates of obstacle vertices.

As BURM maintains bounds on the expected costs of the edges, we can consider edge weights to be *intervals* containing the expected costs. Now in DBURM we no longer assume that the environment is static. A change to the environment can alter the expected costs of some edges, invalidating the correctness of the corresponding intervals, as the previously computed intervals may no longer contain the expected cost.

In conclusion, our problem can be put as: come up with a graph search algorithm that finds the optimal path, when the edge weights are refinable intervals that contain the expected costs of edges, and the expected costs can alter dynamically. The difference from BURM is that now expected costs can alter dynamically.

One way to solve the problem is to re-run BURM whenever a change in the environment is detected. However DBURM attempts at a more efficient

solution by locally 'repairing' the shortest path tree rather than recomputing it from scratch. We first extend the (known) local-consistency conditions of graphs whose edge weights are values, to graphs whose edge weights are intervals: if all nodes are locally consistent, then the shortest path tree is easy to construct. DBURM only considers the non-consistent nodes, 'repairs' their consistency, and re-establishes the solution. This is a common strategy that has also been applied in D* algorithm and its variants [86, 48, 87, 24]. However in DBURM we face the added challenge that the edge weights are now intervals rather than values, which makes it more difficult to identify the non-consistent nodes, as well as repair their consistency.

Our experimental results show that DBURM is a more efficient option, than re-running of BURM. In most experiments, DBURM refines the bounds by 1.5 to 2 times less, before identifying the optimal path in the roadmap.

4.1 Overview of the chapter

The rest of the chapter is organized as follows. The next two sections are: Problem Description and Algorithm Overview. From section 4.4 it gets more detailed. In section 4.4 we motivate the definition of local consistency condition in DBURM, which is an extension of the consistency conditions in the usual graphs where edge-costs are *values*. Section 4.5 goes into the details by giving the formal definition of the consistency conditions, and then proving that if all nodes are locally consistent, then the shortest path tree is easy to construct. Section 4.6 gives the DBURM algorithm and proves that it terminates, and at that time the shortest path tree has been computed. In

section 4.7 we show how to make DBURM more efficient by terminating it earlier: before the entire shortest path tree has been computed, but after having found the optimal path from the given source to destination. In the end we give experimental results and conclusion.

4.2 Problem Description

We assume that the robot has a 2-D map of the environment. The obstacles in the environment are modeled as polygons, whose shapes and positions are not known exactly. Each vertex of a polygonal obstacle is represented as a probability distribution with finite support (Fig. 3.1). This representation is general and makes no assumption on the parametric form of the distribution. If the vertex distribution follows a specific parametric form, *e.g.*, Gaussian, we can further exploit it for computational efficiency. For simplicity, the robot is modeled in the same way.

In our setting, the vertex distributions in the map may change for two reasons: the robot receives new sensor information to refine the map, and the obstacles move. Either way, the map is updated to reflect these changes, as soon as they occur. The robot's objective is to reach a specified goal location by following an "optimal" path based on the current map. The cost of a path accounts for both the path length and the robot's risk of colliding with obstacles. In this work, we focus on computing the optimal path efficiently and assume that the map is updated suitably using standard methods [89].

4.3 Algorithm Overview

Similar to BURM (chapter 3), DBURM builds a roadmap U in the robot's configuration space $\mathcal{C}$. Each node of U represents a robot configuration sampled at random from $\mathcal{C}$, and each edge represents a straight-line path between two nodes that are sufficiently close to each other according to a suitable metric. The expected cost, $\hat{w}(e)$ of travering edge e is defined as :

$$\hat{w}(e) = \ell(e) + \mathbf{E}[c(e)], \tag{4.1}$$

where $\ell(e)$ is the length of the straight-line path represented by e and $\mathbf{E}[c(e)]$ is the robot's expected collision cost while traversing this path. The expected value must be computed with respect to the joint vertex distribution, which is simply the cross-product of the distributions for the individual vertices and is $2n$-dimensional for a map with n obstacle vertices. Calculating the expected value with respect to such a high-dimensional probability distribution may incur a prohibitive computational cost, but we can nevertheless do it efficiently by taking advantage of reasonable indpendence assumptions and evaluating the expected value hierarchically at multiple resolutions. (see chapter on BURM).

The cost of a path γ in U is the sum of the weights for the edges along γ. Our goal is to find a minimum-cost path from the robot's current position to the goal. There are two interesting challenges here.

First, we do not want to compute the edge weights of U exactly. As noted earlier, doing so incurs high computational cost. Instead, we construct upper and lower bounds, *i.e.*, an *interval*, on the collision probability for each

edge e. This leads to corresponding bounds on the expected collision cost $\mathbf{E}[c(e)]$. We initalize the collision probability interval as $[0, 1]$. While searching for the optimal path, we may tighten up these intervals when necessary by performing additional computation. (See the BURM chapter for further details.)

Second, we do not want to recompute the optimal path afresh whenever the environment map changes. We certify an optimal path by establishing a set of local consistency conditions at the nodes of U. A node is *consistent*, if it satisfies the local consistency condition. An optimal path is found, if all the nodes are consistent. When the environment map changes, we place all the inconsistent nodes into a queue. We dequeue these nodes one by one and make them consistent. However, the processing may cause other nodes to become inconsistent, and we must identify these additional nodes and queue them up for future processing. This idea of incremental computation is complicated by the fact that the exact edge weights in U are not known exactly and are represented as intervals. To re-establish consistency, we may need to tighten up some of these intervals and must select them with care to avoid unnecessary tightening and wasted computation. Indeed, one distinguishing feature of DBURM is to combine incremental computation with BURM's interval representation of uncertain environment maps.

As BURM is basically a graph where edge-weights are intervals, in the next two sections we extend the local consistency conditions of the usual graphs, to the DBURM setting: when the edge-costs are intervals.

4.4 Attempting to extend the local consistency conditions of the usual graphs

Our purpose here is to extend the local consistency conditions from the usual graphs where edge weights are values, to graphs where edge weights are *intervals*. The local consistency conditions for the usual graphs is a known result[48]. Each node x maintains an estimate of the cost-to-go, $c(x)$: the estimated cost of the optimal path from x to goal. This estimate is correct if all nodes satisfy the local consistency conditions.

$$c(x) = \begin{cases} 0 & \text{if } x = \text{goal;} \\ min_y\{w(x,y) + c(y)\} & \text{otherwise.} \end{cases}$$

Here $w(x, y)$ is the weight of the edge connecting the nodes x and y.

There are two problems in carrying over: $c(x) = min_y\{w(x, y) + c(y)\}$ in its exact form to DBURM. The first one is easy: in our case the edge weights are *intervals*, so we first need to define what it means to add two intervals, and take the min element from a set of intervals. We do so in section 4.4.1. The second problem is algorithmic: we can not enforce an equality of intervals like above, without significant computational overhead. As edge weights are constantly being tightened by DBURM, in order to enforce *equality* above, each time we would have to propagate the tightened edge weight through the graph: tightening of $w(x, y)$ implies updating $c(x)$, which in turn implies updating $c(.)$ for other nodes, etc.

Fortunately as we will see, a weaker consistency condition suffices. We first define some interval arithmetic.

4.4.1 Operations on intervals

In DBURM the edge weights, $W(e)$, and estimated cost-to-go, $C(x)$ are *intervals*. Note that we indicate intervals by upper case alphabets.

As $W(e)$ are intervals that bound the expected cost of traversing the edge, we want to define interval addition, so that adding the $W(.)$ of edges along a path, give the bounds on the expected cost of traversing the *path*. This is straightforward: Let X be the interval (a, b) and Y be the interval (c, d). By $X + Y$ we denote the interval $(a + c, b + d)$

Definition : *W(edge)* and *W(path)*

By $W(x, y)$ we denote the weight (interval) of the edge connecting nodes x and y. Similarly if p is a valid path to goal, we define $W(p)$ as: $\sum_{e \in p} W(e)$. Here the sum is taken over the edges in the path. If p is not a valid path to goal, we define $W(p) = \infty$.

Note that the weight above is with an upper-case W and represents an *interval*. As we are assuming bi-directional edges, there is no difference between $W(x, y)$ and $W(y, x)$.

Given a set of cost-intervals, S, how would we choose that interval in S, which represents the smallest expected cost? Let $\overline{X}$ and $\underline{X}$ denote the upper bound and lower bound of the interval X. If S is a set of intervals then $X = \min\{S\}$, if $X \in S$ and $\overline{X} \leq \underline{Y}$ for all *other* $Y \in S$. If there is no such element in S, then a minimum element for S is not well-defined.

In the section to follow, the important equation is eqn no. 4.3, and the main theorem is theorem 4.5.1.

4.5 The Local Consistency Condition

We first state the local consistency condition for the usual graphs; when edge weights are values. Here $w(x, y)$ denotes the weight of the edge connecting nodes x and y. We assume that the edge weights are strictly greater than 0. We say that a node x is locally consistent if it satisfies the following equation.

$$c(x) = \begin{cases} 0 & \text{if } x = \text{goal;} \\ min_y\{w(x,y) + c(y)\} & \text{otherwise.} \end{cases} \tag{4.2}$$

This is the same equation as given before but we stated here again for easy reference.

It is a known result that if all nodes are locally consistent then $c(x)$ equals the cost of the shortest path from x to goal. Hence one could easily construct the shortest route by defining for every node x, $\pi(x) = \arg\min_y\{w(x,y) + c(y)\}$ if $\min_y\{w(x,y)+c(y)\} < \infty$, and setting $\pi(x) = \emptyset$ otherwise. Then for every node x that has a path to goal $x \to \pi(x) \to \pi^2(x) \ldots goal$ would be an optimal path from x to goal.

Definition : Local Consistency A node x is said to be locally consistent if it satisfies the following equation.

$$\begin{aligned} C(x) &= [0,0] \text{ if } x = \text{goal, else} \\ C(x) &\supseteq min_y\{W(x,y) + C(y)\} \end{aligned} \tag{4.3}$$

Here $C(x)$ is an estimate of an interval that contains the expected cost

of the optimal path from x to goal; similar to the concept of $c(x)$ above. By $\supseteq$, we mean that $C(x)$ is an interval containing the min interval. We assume that $\underline{W}(x, y) > 0$ for all edges (x, y), and that the min interval above is well-defined.

Note that if the min interval in equation 4.3 is not well defined, then x is non-consistent by definition. The importance of 'local consistency' to the computation of the shortest path tree, can be understood by reading the statement of theorem 4.5.1

Next we need to define the notion of an optimal path from x to goal, when edge weights are intervals. So far we have used the notion that the optimal path is the path with the least expected cost of traversal. The definition below is a bit different, but is equivalent if the expected cost of every edge e, is contained within its edge weight-interval, $W(e)$, which is what we assume in BURM and DBURM.

Definition : optimal path

Suppose we assign each edge e, with a cost $w(e)$ such that $w(e) \in W(e)$. Define $w(path) = \sum_{e \in path} w(e)$. Then p_{opt} is an *optimal path* from x to goal if $w(p_{opt}) \leq w(p_2)$ under any such cost assignment, where p_2 is *any* path in the graph from x to goal.

Note that as $W(e)$ bounds the mean-cost of traversing the edge e, an optimal path from x to goal, is also a path from x to goal with the smallest expected-cost of traversal.

We defined $\pi(x)$ at the beginning of section 4.5 such that if x has a path to goal, then the path $(x, \pi(x), \pi^2(x) \ldots)$ is an optimal path from x

to goal. Motivated by this, we would define $\pi(x)$ below for the case when edge weights are intervals. At a given point in time in DBURM, the path $(x, \pi(x), \pi^2(x) \ldots)$ may not be the optimal path from x to goal, but we'll show that when the algorithm terminates, it's indeed the optimal path for nodes of interest.

But first, we need to figure what properties should $\pi(x)$ satisfy so that the sequence of $\pi(.)$ above *does* trace out an optimal path. For that we have theorem 4.5.1. Below, when we say that an interval $I = \infty$, we mean that $I = [\infty, \infty]$. We choose this notation for brevity. Note that if $I_2 \supseteq I_1$, and $I_1 = \infty$ then $I_2 = \infty$ as well.

Definition If $\pi(x) = y$, we say that y is a *parent* of x.

Theorem 4.5.1 *Suppose all nodes are locally consistent. Let* $\pi(x) = \emptyset$ *if* $x = goal$*. Otherwise let* $\pi(x) = \arg\min_y \{W(x, y) + C(y)\}$ *if this min interval* $\neq \infty$*, and* $\pi(x) = \emptyset$ *otherwise.*

We claim that the edges connecting nodes to their parents make a shortest path tree. That is, $x, \pi(x), \pi^2(x) \ldots goal$ *is an optimal path from* x *to goal, if a path from* x *to goal exists. If no such path exists, then* $\pi(x) = \emptyset$*.*

The proof depends on some lemmas which are given immediately after the proof of the theorem. We chose this sequence just to give the main result as quickly as possible.

Proof The claim is trivially true for the goal node. If $x \neq goal$, then by lemmas 4.5.3, 4.5.4, 4.5.5, the sequence $\pi(x), \pi^2(x) \ldots$ exists iff x is in the same connected component as the goal, and this sequence always ends at

the goal node. This means that $\pi(.) = \emptyset$ for all nodes which are not in the same connected component as the goal. As nodes that are not in the same component as the goal, have no path to goal, the statement of the theorem is satisfied for such nodes.

What remains is to prove the statement of the theorem for nodes that are in the same connected component as the goal. Let $W(x, y)$ be any edge weight and $w(x, y) \in W(x, y)$. For each edge (x, y) choose $w(x, y)$ arbitrarily from $W(x, y)$. Let $p(x)$ be the path $x, \pi(x), \pi^2(x) \ldots goal$. Next, define $c(x) = \sum_{(x,y)\in p(x)} w(x, y)$.

Consider now the same graph, but with edge costs as $w(x, y)$ and cost estimates at a node x being $c(x)$. Therefore we now have the usual graph where edge costs are values. Also note, that we did not change any $\pi(.)$ when going from the original graph to this new graph. All nodes hence satisfy $c(x) = w(x, \pi(x)) + c(\pi(x))$. All that we now need to show is that the conditions in (1) are satisfied at all nodes in this connected component. Because then all nodes would be consistent by (1), as well as $\pi(x) = \arg\min_y\{w(x, y) + c(y)\}$. This would show that if we instantiate the edges with *any* cost value taken from within their cost-intervals, the set of $\pi(.)$ still define a shortest path tree.

Now to prove that conditions in (1) are indeed satisfied. Let z be a node in the same component as the goal. By lemma 4.5.2, $c(z) \in C(z)$ for all such nodes. Therefore, $w(x, y) + c(y) \in \{W(x, y) + C(y)\}$. Now as we defined $\pi(x) = \arg\min_y\{W(x, y) + C(y)\}$, it means that $\pi(x) = \arg\min_y\{w(x, y) + c(y)\}$ as well. As we assigned $c(x) = w(x, \pi(x)) + c(\pi(x))$, this shows that $c(x) = min_y\{w(x, y) + c(y)\}$. Hence the nodes satisfy the

consistency conditions in (1). ∎

Lemma 4.5.2 *Suppose all nodes are locally consistent, and that $\pi(x)$ is defined as in theorem 4.5.1. If $(x, \pi(x), \pi^2(x)\ldots)$ leads to goal, then with $c(x)$ and $w(x)$ as defined in the proof of theorem 4.5.1, $c(x) \in C(x)$.*

Proof We prove by induction along the path $(x, \pi(x), \pi^2(x)\ldots goal)$. We start from goal and go towards x.

The statement is obviously true for the goal node, as $C(goal) = [0, 0]$ by definition. Now suppose that the statement is true for $\pi(y)$, i.e. $c(\pi(y)) \in C(\pi(y))$. Hence we have (i) By local consistency, $C(y) \supseteq W(y, \pi(y)) + C(\pi(y))$, (ii) $w(y, \pi(y)) \in W(y, \pi(y))$ and (iii) $c(\pi(y)) \in C(\pi(y))$. Hence $w(y, \pi(y)) + c(\pi(y)) \in C(y)$. As we assign $c(y) = w(y, \pi(y)) + c(\pi(y))$, this proves that $c(y) \in C(y)$. Hence by induction, the statement is true for all nodes in the path $(x, \pi(x), \pi^2(x)\ldots goal)$. ∎

Lemma 4.5.3 *Suppose all nodes are locally consistent, and that $\pi(x)$ is defined as in theorem 4.5.1. We claim the sequence $\pi(x), \pi^2(x), \pi^3(x)\ldots$ can not have any loops. That is, no node repeats twice in this sequence.*

Proof By contradiction. As $C(x) \supseteq W(x, \pi(x)) + C(\pi(x))$. Hence $\underline{C}(x) \leq \underline{W}(x, \pi(x)) + \underline{C}(\pi(x)) \leq \underline{W}(x, \pi(x)) + \underline{W}(\pi(x), \pi^2(x)) + \underline{C}(\pi^2(x))$ etc. Continuing in the same way, if we have a loop such as:
$(x, \pi(x), \pi^2(x), \ldots, x)$, then we get

$$\underline{C}(x) \leq \underline{W}(x, \pi(x)) + \underline{W}(\pi(x), \pi^2(x)) + \underline{W}(\pi^2(x), \pi^3(x))\ldots + \underline{C}(x)$$

As $\underline{W}(x, y) > 0$ for all edges, this is a contradiction. ∎

Lemma 4.5.4 *Suppose all nodes are locally consistent, and that $\pi(x)$ is defined as in theorem 4.5.1. If $\pi(x) \neq \emptyset$, then the sequence $\pi(x), \pi^2(x), \pi^3(x) \ldots$ must end at the goal.*

Proof We prove by contradiction. Suppose $\pi(x) \neq \emptyset$, and that the sequence $\pi(x), \pi^2(x) \ldots$ does not end at the goal. By lemma 4.5.3 this sequence can not have any loops. Hence if the sequence does not end at the goal, then it ends at some last node $z \neq goal$. Now $\pi(z) = \emptyset$, hence $C(z) = \infty$. But then $C(\pi^{-1}(z)) = \infty$ for all nodes in $\pi^{-1}(z)$. Similarly for $\pi^{-2}(z)$. Continuing backwards in this way, we conclude that $C(x) = \infty$. But in such a case we assign $\pi(x) = \emptyset$. Contradiction.

Lemma 4.5.5 *Suppose all nodes are locally consistent, and that $\pi(x)$ is defined as in theorem 4.5.1. If $x \neq goal$, then $\pi(x) = \emptyset$ iff x is not in the same connected component as the goal.*

Proof x is not in the same connected component $\implies$ $\pi(x) = \emptyset$, has been proved by the above two lemmas. (because the sequence of $\pi(.)$ pointers can not have any loops and must end at the goal).

We only need to prove: $\pi(x) = \emptyset \implies x$ is not in the same connected component as the goal. We prove the contrapositive. That is, x is in the same connected component as the goal $\implies \pi(x) \neq \emptyset$.

We prove by induction. First consider all nodes which are one hop away from goal. It would suffice to show that for each such node x there exists a node y such that $C(y) \neq \infty$, and $W(x, y) \neq \infty$. Because in such a case, the *min* interval in equation 4.3 can not be ∞. For the nodes one hop away

from the goal, such a node is the goal as $C(goal) = [0, 0]$. Therefore if x is one hop away from goal, $C(x) \neq \infty$ and hence $\pi(x) \neq \emptyset$.

Now suppose that the result is true for all nodes within n hops of the goal. But then if a node x is $n + 1$ hops away, then it is a neighbor of some node y which is n hops away. Then applying similar reasoning as above, we conclude that for all nodes $n + 1$ hops away from goal, $\pi(x) \neq \emptyset$. ∎

4.6 The DBURM algorithm

In the previous section we showed that if all nodes are consistent and $\pi(x)$ satisfies certain properties, then $x \to \pi(x) \to \pi^2(x) \dots goal$ is an optimal path from x to goal. Motivated by this we define a parent pointer $\pi(x)$ for each node x. At a given time it may not be satisfying the properties in theorem 4.5.1. However we will eventually prove that when the algorithm terminates, all $\pi(.)$ do indeed satisfy those properties, and hence $x, \pi(x), \pi^2(x) \dots$ is an optimal path from x to goal.

Definition : The path *p(x)*

For any node x, we define $p(x)$ as the path $x, \pi(x), \pi^2(x) \dots$

We first give the offer(x, y) routine. Note that this routine is quite similar to the relax(x, y) as defined in Dijkstra's algorithm in popular texts. However we chose the name 'offer' as we believe it is a more natural choice of words. In offer(x, y), x offers the path $y \to x \to \pi(x) \to \pi^2(x) \dots goal$ to y. y may select this path by setting $\pi(y) = x$. However it selects the offer only if the cost of its current path $p(y)$, is more than that of the offered path.

It would not make much sense to perform all the computations in offer(x, y) if $\pi(y) = x$ is already true, and therefore we don't do that (see line 1 of offer())

Algorithm 2 offer(Node x, Node y)

1: **if** $\pi(y) = x$ **then**
2: return.
3: Let p_{offered} be the path $y \rightarrow p(x)$.
4: collect the edges constituting p_{offered} and $p(y)$.
5: **if** The interior of the intervals $W(p_{\text{offered}})$ and $W(p(y))$ intersect **then**
6: Tighten the intervals $W(p_{\text{offered}})$ and $W(p(y))$ until the interiors of the two intervals do not intersect.
7: **for** All nodes z along p_{offered} and $p(y)$ **do**
8: Update $C(z)$: Set $C(z) = W(p(z))$, if $p(z)$ is a path to goal. Else set $C(z) = \infty$ and $\pi(z) = \emptyset$.
9: **if** $\overline{W}(p_{\text{offered}}) < \underline{W}(p(y))$ **then**
10: Set $\pi(y) = x$ and $C(y) = W(y, x) + C(x)$

On line no. 8 above, we update the cost-intervals, $C(.)$ of all nodes along the two paths. In the case we find that one of those paths does not end at goal, we set $\pi(.) = \emptyset$. Note that the $\pi(.)$ sequence can not have any loops. This is because before we change $\pi(y)$ above, we narrow the two $W(path)$ until they do not intersect. A loop therefore would imply that $\overline{W}(p') \leq \underline{W}(p)$, where p is a path, and p' is a path that is a strict subset of path p. As we assume that $\underline{W}(e) > 0$ for all edges e, this is a contradiction.

Hence the only way that the path does not lead to goal is that it is of the following type: $(x, \pi(x) \ldots \pi^n(x), \emptyset)$, where $\pi^n(x) \neq goal$. In this case some edges in the graph must have become invalid, so that the weights of those edges are now ∞. This change had been propagated to $\pi^n(x)$ (as $\pi^{n+1}(x) = \emptyset$). but not to the other nodes in that path. Whenever we find such a path on line 9, we set $\pi(.)$ of all the nodes in the path to $\emptyset$.

Note that in offer() we are comparing $W(.)$ of the paths, which is the sum of the $W(e)$ where e is an edge in the path. Therefore we need line 4 above to know the $W(path)$, as well as to able to tighten the bounds on line 7 (see section 3.5 for details of how we tighten the bounds). Below is the DBURM algorithm.

Definition We say that a node y is a neighbor of node x, if $W(x, y) \neq \infty$.

Algorithm 3 DBURM

1: make roadmap.
2: Set $C(x) = \infty$, $\pi(x) = \emptyset$ for all nodes.
3: Set $C(goal) = [0, 0]$ and insert goal node into queue
4: **while** Robot is not at goal **do**
5: **while** queue is not empty **do**
6: Remove a node x from queue
7: For all nodes $z \in p(x)$, set $C(z) = W(p(z))$ if $p(z)$ is a path to goal. Else set $C(z) = \infty$, and $\pi(z) = \emptyset$.
8: offer(y, x) for all neighbors y of x
9: offer(x, y) for all neighbors y of x
10: For all nodes z such that $\pi(z) = x$, set $C(z) = W(z, x) + C(x)$,
11: While running lines 6-10, for all nodes $z \neq x$, if $C(z)$ changed at some point such that $C_{new}(z) \not\subseteq C_{old}(z)$. Then insert z into queue.
12: **while** no changes are observed and robot is not at goal **do**
13: Move the robot along $p(r)$, which is the path $r \rightarrow \pi(r) \rightarrow \pi(\pi(r)) \ldots goal$.
14: Reinitialize $W(edge)$ for all edges that could be affected by the environment change, and insert all nodes adjacent to these edges into queue.

We would like to show that when the queue in DBURM becomes empty then all nodes satisfy the hypothesis of theorem 4.5.1. That is: (i) all nodes are locally consistent. (ii) $\pi(x) = \emptyset$ if $x = goal$. Otherwise $\pi(x) = \arg\min_y\{W(x, y) + C(y)\}$ if this min interval $\neq \infty$, and $\pi(x) = \emptyset$ otherwise.

If this is indeed true, then the conclusion of theorem 4.5.1 would also be true, and for any node x, $p(x)$ would be the optimal path from x to goal.

Definition We say that a cost interval, W (or C) has been *modified* , if the bounds of W change such that $W_{new} \not\subseteq W_{old}$.

Cost-modification in our setting has a similar role to cost-modification in D* and Dijkstra. For one, note that in DBURM algorithm above, we insert a node x into queue when $C(x)$ *modifies* . In order to prove our desired claim given above, that $p(x)$ is the optimal path from x to goal when the queue becomes empty, we begin with the following two lemmas.

(*Note that the statement of all lemmas and theorems assumes that the state of execution is not 'inside' the while loop on line* 5.)

Lemma 4.6.1 *Suppose that the cost is 'inconsistent' between a parent-child in the following way: Let* $\pi(y) = x$*, and suppose* $C(y) \not\supseteq W(y, x) + C(x)$*. Then* x *must be on queue.*

Proof We prove the contrapositive: Suppose $\pi(y) = x$, and that x is not on queue. Then $C(y) \supseteq W(y, x) + C(x)$.

Note that *whenever* $C(y)$ is updated, the condition $C(y) = W(y, \pi(y)) + C(\pi(y))$ is true immediately afterwards (see line 8 of offer() and line 10 of DBURM). Now we always update $C(y)$ if $\pi(y)$ changes. So the previous time we updated $C(y)$ we had: $C(y) = W(y, x) + C(x)$. Let this time be t. It suffices to prove that $W(y, x)$ and $C(x)$ have not modified since t.

There could be many iterations of the while loop on line 5, when this node x was the one chosen for removal on line 6. Let L_x be the most recent

iteration of such iterations. It is clear that $t' < t$, as x updates $C(.)$ for all its children on line 10. As x has not entered the queue since t' by definition of t', $W(y,x)$ and $C(x)$ have not modified since t'. Otherwise x would be reinserted into the queue. Hence $W(y,x)$ and $C(x)$ have not modified since t. This proves the lemma. ∎

Definition : **Reject**

We say that x 'rejects' a path offer from y when

1. $\pi(x) = y$, but then $\pi(x)$ is updated so that $\pi(x) \neq y$. (Hence x preferred another option).

2. We perform offer(y,x), and at the end of this offer we have $\pi(x) \neq y$. Therefore y does not become the parent of x despite offering the path to x. To be specific, in this case, we define the 'time' of rejection as the time at the end of this offer(y,x).

Lemma 4.6.2 *Let x be a node. Assume x was inserted into queue at some time, but currently neither x nor y is on the queue. We claim if $\pi(x) \neq y_i$, then $\overline{C}(x) \leq \underline{W}(x,y_i) + \underline{C}(y_i)$*

Proof As x was inserted into queue, x must have been removed from queue as well. x may have been selected for removal (on line 6 of DBURM) in several iterations. Let L_x be the last such iteration. Let t be the time at the end of line 7 in L_x. We will follow what happened in L_x.

Note that $C(x) = W(p(x))$ after line 7. As none of the edge-weights $W(x,y)$ can modify inside the loop starting on line 5 of DBURM, $W(p(x))$ can only become tighter after line 7. Hence, for as long as $p(x)$ does not

change, $\overline{C}(x)$ can not increase. On the other hand, if $p(x)$ changes to a new path, then the upper bound of that new path is even smaller. We conclude that $\overline{C}(x)$ can not increase during L_x after executing line 7.

Let y_i be any neighbor such that $\pi(x) \neq y_i$, and t_i be the most recent time when x rejected the offer from y. It is clear that $t_i > t$. As $\overline{C}(x)$ has been non-increasing since t, we conclude:

$$\overline{C}_{curr}(x) \leq \underline{W}_{t_i}(x, y_i) + \underline{C}_{t_i}(y_i)$$

By C_{curr} or W_{curr} we denote the respective intervals 'now' (at the current time). We claim that the above inequality is still true, in the sense:

$$\overline{C}_{curr}(x) \leq \underline{W}_{curr}(x, y_i) + \underline{C}_{curr}(y_i)$$

This is because of the three points below:

1. $\overline{C}(x)$ is non-increasing since t_i.

2. $\underline{W}(x, y_i)$ has been non-decreasing since t_i. This is because $W(x, y_i)$ has not modified since t_i, else x would be inserted into queue. Contradicting the definition of L_x.

3. $\underline{C}(y_i)$ has been non-decreasing since t_i. This is because $C(y_i)$ has not modified since t_i. Note that if t_i occurred inside an iteration where y_i was chosen for removal from queue, then t_i occurred on line 9, and $C(y)$ can not modify after taking the best offer() on line 8. Hence if

$\underline{C}(y_i)$ decreases after t_i, then as $C(y)$ would *modify* outside such an iteration, y_i would be inserted into queue at a time $t' > t_i$. As y_i is currently not in queue, it would be removed from queue at a still later time. This contradicts that t_i was the last time when x rejected the path offer of y_i.

This proves the claim ∎.

We now prove an important result of this section. Note the similarity between the statement of this theorem and that of 4.5.1.

Lemma 4.6.3 *Suppose $x \notin Q$, as well as none of its neighbors are on queue. Then x is locally consistent. Also, $\pi(x) = \emptyset$ if $x = goal$. Otherwise $\pi(x) = \arg\min_y\{W(x,y) + C(y)\}$ if this min interval $\neq \infty$, and $\pi(x) = \emptyset$ otherwise.*

Proof We break it down into two cases.

CASE (A)

First suppose that x was never inserted into queue. Hence $C(x)$ never *modified* , and hence $C(x) = \infty$, and $\pi(x) = \emptyset$. This also means that the neighbors of x never offered a path that had a cost $< \infty$. Now by lemma 4.6.4, $C(y) = \infty$. Hence we conclude that $C(y) = \infty$ for all neighbors y of x. Also as x was never inserted into queue $C(x) = \infty$, and $\pi(x) = \emptyset$. Its straightforward to verify that x satisfies the conclusion of the theorem in this case.

CASE (B)

Now we assume that x was inserted at some time into queue. By lemma

4.6.2, if $\pi(x) \neq y_i$, then:

$$\overline{C}(x) \leq \underline{W}(x, y_i) + \underline{C}(y_i) \tag{4.4}$$

First suppose $\pi(x) = \emptyset$, then it must be that $C(x) = \infty$. (Whenever we update $C(x) \neq \infty$, then we set $\pi(x)$ to the relevant neighbor). Collecting these conclusions we get: $C(x) = \infty$, $\pi(x) = \emptyset$, and $\overline{C}(x) \leq \underline{W}(x, y_i) + \underline{C}(y_i)$ over all neighbors y_i of x. It is easy to verify that x satisfies the statement of the theorem.

Now suppose $\pi(x) \neq \emptyset$. We need to show that $C(x) \supseteq C(\pi(x)) + W(x, \pi(x))$. But as x and $\pi(x)$ are not in queue, by lemma 4.6.1 this is indeed true. Because of inequality 4.4 above, we can now conclude $C(\pi(x)) + W(x, \pi(x)) = argmin_y\{C(\pi(y)) + W(y, \pi(y))\}$. This proves the claim ∎

Lemma 4.6.4 *If x has never been inserted into the queue, and y is a neighbor of x, then $C(y) = \infty$*

Proof By contradiction. Suppose $C(y) \neq \infty$.

1. y must have entered the queue as $C(y)$ modified . As y is currently not in queue, y was removed.

2. y may have been selected for removal (on line 6 of DBURM) in several iterations. Let L_y be the last such iteration. Now the edge weights can not modify to cost of ∞ inside L_y. Also no edge weight $W(\_, y)$ has modified since after L_y. Else y would be inserted into queue again, contradicting the definition of L_y. Hence x has remained a neighbor of y ever since line 6 of L_y.

Hence y must have offered a path to x on line 9 of L_y. Therfore it must be that $C(y) = \infty$ and $\pi(y) = \emptyset$ at the end of line 8 of L_y. Now $C(y)$ can not *modify* after line 8 in L_y (as y has already looked at all offers, $p(x)$ remains constant after line 8). Hence $C(y) = \infty$ at the end of L_y. As $C(y)$ has not modified since (else y would be inserted once more into queue), $C(y)$ is still ∞. Contradiction. ∎

Now we come to a main theorem in this chapter.

Theorem 4.6.5 *DBURM is correct and terminates, regardless of the order in which we dequeue the nodes (line* 6*).*

Proof Lemma 4.6.3 shows that if the queue becomes empty, then the hypothesis of theorem 4.5.1 is satisfied, and hence $x, \pi(x), \pi^2(x) \ldots$ defines the optimal path from x to goal, or $\pi(x) = \emptyset$ if no such path exists. Hence DBURM is correct.

We claim that the loop on line 5 of DBURM must terminate. This is because whenever a node x leaves the queue, $C(x) = W(p(x))$. Hence if x enters the queue again, $C(x)$ must have modified . As weights of edges do not modify inside the loop on line 5, $C(x)$ will modify only if $p(x)$ changes to a strictly better path (Note the strict inequality on line 12 of offer()). This can happen only a finite number of times. Eventually x must stop entering the queue. Applying this reasoning over all nodes in the graph, proves the claim. ∎

4.7 Terminating the while loop on line 5 earlier

Many times what we are interested in, is to have the optimal path from a given source node, to the goal. In such a case it could be quite inefficient to compute the shortest path tree in its entirety. Maybe we could start computing the shortest path tree, and then stop the process as soon as we know that the optimal path from the given source has been successfully computed. But how would we know that?

Many graph search algorithms like Dijkstra's and D* keep a queue of nodes (we also do in DBURM), and a key $k(x)$ is defined for every node in the queue. In Dijkstra and A*, once $\min_{x \in queue} k(x) > c(src)$, where c is the estimated cost-to-go and src is the source node, then the optimal path from src has already been computed.

4.7.1 Defining a key, $k(x)$ for every node x

Motivated by the above we also define a key, $k(x)$ for every node. Our motivation is to terminate the graph-search algorithm (the while loop on line 5) earlier; once we know that $p(src)$ is an optimal path from src to goal, where src is the source node. Therefore $\min_{x \in Q} k(x)$ should indicate that no offer from any nodes in the queue can now be made that can potentially improve the current path $p(src)$ from source to goal. Hence, intuitively, the value $\min_{x \in queue} k(x)$ should handle the following aspects:

1. It should lower bound the path with the least cost, that can be offered

by any node in the queue. Hence we would expect $\underline{W}(p(x))$ or $\underline{C}(x)$ to feature in the definition of the key.

2. But note that in a dynamic setting $\underline{W}$ and/or $\underline{C}$ is not enough to know the least cost that can be offered. For example, it might be that x rejected a path from y of cost 8, because at that time $p(x)$ had a cost 7. However as we are in a dynamic setting, the edge costs along $p(x)$ may have increased since then. It could be that now $C(p(x)) = 9$. So x should look in its neighborhood (note line 8 in DBURM), and get the offer from y again. Therefore, we also need to keep track of the path costs that were rejected before, but now maybe good enough to be accepted. Now whenever x rejects an offer from y, we have $\overline{C}(x) \leq \underline{W}(x, y) + \underline{W}(p(y))$. Therefore we can expect a certain history of values of $\overline{C}(x)$ to feature in the definition of $k(x)$, in order to lower bound the offers from neighbors of x.

Now we give a formal definition for the key. We first define $h(x)$; a certain history of the values of $\overline{C}(x)$.

Definition : $h(x)$

Consider those iterations of the while loop (on line 5), when the given node x was chosen for removal on line 6. There could be several such iterations. Let L_x be the most recent of such iterations. Let t be the time when we just finished executing line 7 in L_x, or $t = -\infty$ if there was no such iteration. We define $h(x)$ as the minimum value of $\overline{C}(x)$ since t.

Note that if x is currently in queue, then $h(x)$ is smaller than the smallest value of $\overline{C}(x)$ since x most recently entered the queue.

The reason that line no. 8 features in the above definition is, because we want to keep track of the smallest outdated value of $\overline{C}(x)$ that might have been used to update the children of x; i.e. those nodes z where $\pi(z) = x$. The important fact is: if $\overline{C}(z) \leq h(x)$, then $\pi(z) \neq x$.

Definition : k(x)

$$k(x) = \min\{h(x), \underline{C}(x)\} \tag{4.5}$$

We defined $k(x)$ is to terminate the while loop on line 5 before the queue becomes empty, but after the optimal path from the source node has been found. That is when DBURM terminates, the path $p(src)$ should be the optimal path from src to goal. Define $\underline{k}$ as : $\underline{k} = \min_{x \in queue} k(x)$. We claim that DBURM remains correct if we terminate the while loop on line 5 earlier by changing line 5 to: $while(k(src) \geq \underline{k})$. That is, once this condition becomes false, we can terminate the while loop as $p(src)$ is the optimal path from src to goal. We prove this result later in theorem 5.8.2, as almost a corrollary of the correctness of the more general PACBURM in chapter 5.

4.8 Experiments

In this section, DBURM refers to Algorithm 2 improved with the priority queue described in Section 4.7

4.8.1 The BURM* algorithm

We compare DBURM to BURM*, an improved version of BURM for dynamic environments. BURM is designed for static environments. Whenever a change in the environment map occurs, it re-initializes all the collision probability intervals to $[0, 1]$ and computes the optimal path afresh. However, this may be unnecessary as the change may be local and only affect a small subset of collision probability intervals. To improve computational efficiency, BURM* identifies these intervals in a conservative manner and only re-initializes them to $[0, 1]$, while retaining the values of other intervals. If the edge is far enough form the environmental change so that its cost can't be affected, the weight (interval) $W(e)$ of the edge is retained.

4.8.2 Results

We tested DBURM in five different environments with robots having up to four degrees of freedom. DBURM substantially outperformed BURM* in Test 1–4. DBURM did not perform as well in Test 5, leading to the observation that the priority order in the queue interacts with the interval representation of an uncertain environment representation in interesting ways.

The environments in the figures are shown with small boxes around the vertices of the obstacles. These boxes indicate the finite support of the vertex distributions: the underlying position of the vertex could be anywhere in the box. The probability distributions used were 'gaussian-type': the probability mass was highest in the center of the box, and tailed off towards the edges.

The gray obstacles shown in the figures, are the ones we get when all the vertices placed at the mean positions of their respective distributions.

The algorithms would first solve the query in the initial environment, and compute the optimal path. Then as the robot is traveling along its path, changes would be made to the environment, forcing the algorithm to recompute the optimal path to goal.

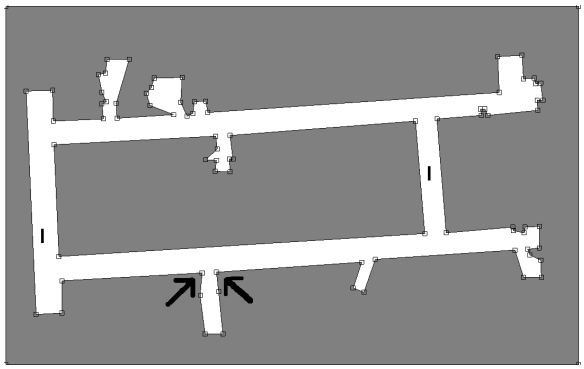
Experiments 1 and 2

The environment and robot for these experiments are shown in figs 4.1(a) and (b). The robot has 2 degrees of translation. It is shown as a thin rod on the left and the right side of the environments, which are the starting and goal configurations of the robot respectively.

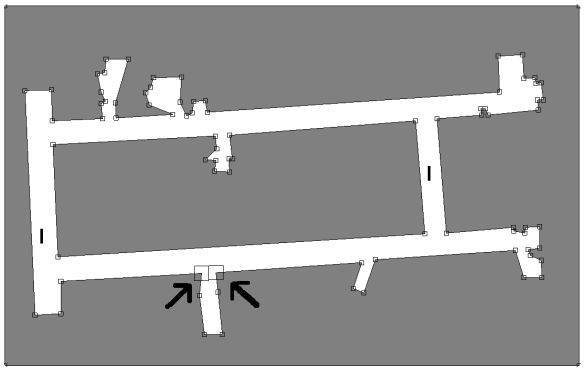
The difference between the environments of 4.1(a) and 4.1(b), are the sizes of two vertex distributions in the lower passage. In figure 4.1(b) there is an increased uncertainty (bigger boxes) around two of the vertices (see arrows in fig 4.1).

In experiment 1 the starting environment was 4.1(a), and the dynamic change to the environment led to 4.1(b). In experiment 2 it was the other way round. Therefore in experiment 1 the uncertainty increased, and in experiment 2 it decreased. The robot would go via the lower passage, and the dynamic change was made just before the robot would pass over the two vertices mentioned above (see the two arrows in fig 4.1(a) and (b)).

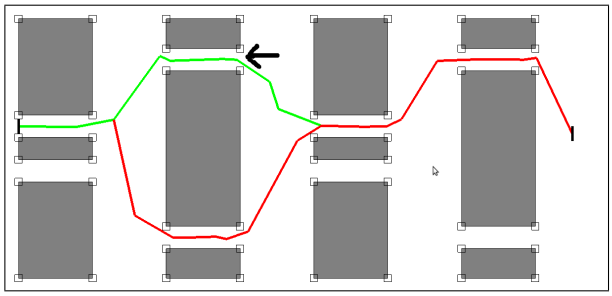
Results are shown in table 4.1. All results are the average over 30 runs. 'Nodes' is the average number of nodes that were inserted into the priority



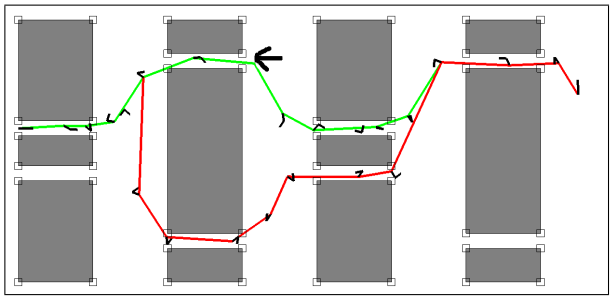
(a) Lower uncertainty at the two vertices indicated by the arrows



(b) Higher uncertainty at the two vertices indicated by the arrows



(c) Experiment 3: Robot with 2 dofs. The passage indicated by the arrow was blocked before the robot entered it.



(d) Experiment 4: Robot with 4 dofs. The passage indicated by the arrow was blocked before the robot entered it.

Figure 4.1: Environments for the first four experiments

queue. 'CCs' are the collision checks: the number of segment-segment intersection tests ($\times 10^5$) performed by the algorithms to tighten the edge weight intervals, $W(e)$, in the roadmap. In these experiment the CCs are directly proportional to the amount by which $W(e)$ were tightened. Hence CC's indicate the amount of additional information required by the algorithm before identifying the optimal path. (For details of how intersection tests are used to tighten $W(e)$, refer to BURM (chapter 3). Time $T(s)$, given in seconds, is the time taken to compute the desired path from the current to the goal configuration *after* the dynamic changes.

Table 4.1: DBURM vs BURM* (experiments 1 & 2)

	Experiment 1			Experiment 2		
Algo	Nodes	CCs	T(s)	Nodes	CCs	T(s)
DBURM	122	7.5	2.31	123	2.4	0.84
BURM*	219	11.3	3.24	221	5.6	1.70

The results show that DBURM performs measurably better than BURM* in both CCs as well as time. In the second experiment, DBURM takes less than half the number of CCs and around half as much of time.

Experiments 3 and 4

In these experiments we block a narrow passage on the path of the robot, just before the robot would enter the passage. The robot had to go from the start configuration shown at the left end of figures 4.1(c) and 4.1(d), to the goal configuration at the other end of the environment. The arrows in figure 4.1(c) and 4.1(d) indicate the narrow passage that was blocked.

Table 4.2: DBURM vs BURM* (experiments 3 & 4)

	2 dof robot			4 dof robot		
Algo	Nodes	CCs	T(s)	Nodes	CCs	T(s)
DBURM	237	2.1	1.42	231	4.0	2.43
BURM*	379	5.8	2.26	372	6.7	2.85

The figures also show the optimal paths computed both before and after the blocking of the passage. As can be seen the robot was forced to go from the passage towards the lower half of the environment after the upper passage was blocked.

Figure 4.1(c) shows the experiment with a robot with two degrees of freedom, while figure 4.1(d) shows the experiment with a robot with 4 degrees of freedom. In the case of the 4-dof robot, the figure also shows the configurations of the robot along the paths.

The results are in table 4.2. DBURM performs measurably better in these experiments. Note again that DBURM clearly performed lesser CCs, with almost 3 times lesser number of CCs in the experiment with 2-dof robot.

Experiment 5

In this experiment we give an example of where DBURM could perform worse. The robot has 3 degrees of freedom; 2 for translation and 1 for rotation. The robot had to go from the horizontal position towards the top of the environment (see fig 4.2), to the horizontal position at the bottom as shown. The starting environment for this experiment is the same as in chapter 3, figure 3.9(b). However, for the dynamic change, we block the

Table 4.3: Performance statistics for experiment 5

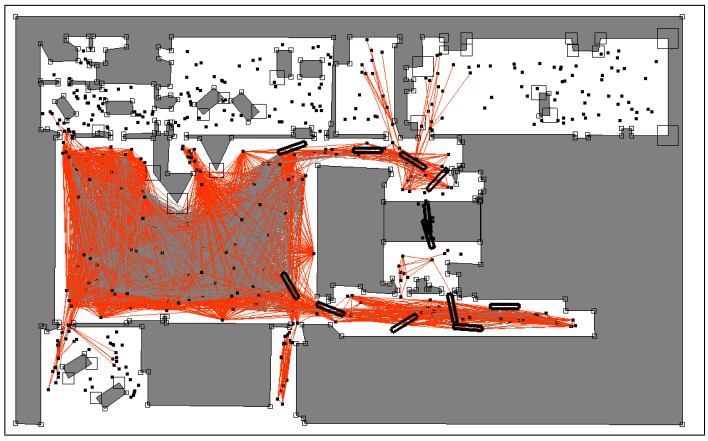
DBURM			BURM*		
Nodes	CCs	T(s)	Nodes	CCs	T(s)
172	9.3	29.9	293	7.7	28.1

critical narrow passage on its path, forcing the robot to go the longer way after this dynamic change to the environment.

Figs 4.2(a) and 4.2(b) also show the roadmaps explored by the algorithms. Red edges are those whose cost-intervals were tightened by the algorithms. Gray are those that were too far from any obstacle, and the probability-of-collision was easily assigned as 0 without much computations.

Note the two compartments on the top right hand side of the environment: DBURM explored more edges than BURM*. Why? Recall that DBURM does not tighten the $W(.)$ or $C(.)$ intervals while prioritizing on queue. Hence there is only a partial order of intervals on the queue. In other words, if x is at the head of the queue, $W(x)$ may not be the min $W(.)$ interval amongst the nodes on queue (see definition of min under Interval Operations in 4.4). DBURM also had to perform more CC's than BURM* (see table 4.3).

(a) DBURM: Roadmap after dynamic change



(b) BURM*: Roadmap after dynamic change

Figure 4.2: Experiment 5

4.9 Summary

In this chapter we presented the DBURM algorithm for path planning in a dynamic environment with an imperfect environment map. DBURM builds on top of the BURM algorithm of chapter 3 which was for imperfect maps but in a static environment. Hence like BURM, DBURM builds a roadmap in the robot's configuration space, and then identifies the optimal path in the roadmap.

Like BURM, the expected costs of edges are not calculated exactly, but only the upper and lower bounds of the costs are maintained. These bounds are refined conservatively; only if such information is useful in identifying the optimal path in the roadmap. Hence we can consider edge weights to be intervals, that contain the expected cost of the corresponding edge.

We first extended the local consistency condition from the usual graphs where edge weights are values, to graphs where edge weights are intervals. The local consistency condition, if satisfied by all nodes in a graph, implies that the shortest path tree has been computed. We extended that condition to a graph where edge costs are intervals, and proved similarly: that if all nodes are locally consistent then the shortest path tree has already been computed.

We then presented the DBURM algorithm for path planning in a dynamic environment with an imperfect environment map. DBURM maintains interval bounds on a robot's collision probabilites and tightens up the bounds, when necessary, to identify an optimal path. When an envionment change invalidates a previously optimal path, DBURM updates the path and re-

establish its optimality by "repairing" the local consistency condition at a selected subset of roadmap nodes. One distinguishing feature of DBURM is to combine this incremental computation with the interval representation of uncertain environment maps.

We compared DBURM to BURM* which is a smart re-running of BURM. BURM* re-runs BURM after a dynamic change in the environment is observed. However BURM* maintains the cost bounds of those edges in the roadmap, whose cost could not have been affected by the given dynamic change. Hence it does not need to refine those bounds all over again. Nevertheless, in most of our tests, DBURM outperformed BURM* by 1.5 to 2 times in terms of time taken. In most experiments DBURM also refined the cost bounds by 1.5 to 2 times less before identifying the optimal path. We also gave one example in which DBURM did not perform as good and explained the cause for the underperformance. Like some other replanning algorithms, DBURM does not immediately discard the estimated cost to go, $C(x)$ after the dynamic change, and uses the former $C(x)$ to prioritize the nodes on the queue. If the new costs, and the former costs happen to be very different, then this prioritization may end up looking at more nodes than required. This possible disadvantage is also shared by other replanning algorithms [86, 48].

Chapter 5

PACBURM : Efficiently finding a Probably Approximately Optimal path

It happens many times that we are given a graph but we do not know the expected costs of edges. Typically the problem is solved by running simulations - the traversal of edge e is simulated n number of times, hence generating a sample of n costs that were incurred when traversing e. These average of these samples is usually taken as the expected costs of e, $\hat{w}(e)$. However accurately speaking, this is not true. We can at best only compute an interval that bounds $\hat{w}(e)$ with a probability of $(1 - \alpha)$, where $\alpha > 0$. Such an interval is called as the $(1 - \alpha)$ confidence interval of $\hat{w}(e)$. The general situation we are considering is that we do not know the expected costs of edges, but we can generate a sample of costs incurred. These samples may be available due to historical data, or may be obtained by simulations. The

problem is to return a useful path in this situation.

The idea is not to take too many samples, while returning a path that with a high probability, is the optimal path. By optimal path, we mean the path with the least expected cost. Further, if we allow the graph search algorithm a 'tolerance', we can return a path that is *close* in cost to that of the optimal path, with a high probability. Introducing such tolerance would be a useful idea if the error incurred is small, while gain in efficiency is large. We introduce this tolerance in our algorithm and both our theoretical analysis, as well as experimental results, show that this is indeed a very useful idea. As mentioned above, the usual idea of running simulations generates a $(1 - \alpha)$ confidence interval, and therefore our idea of 'tolerance' can be used to reduce the number of simulations we need, while the path returned remains of an acceptable quality. We do not know of another algorithm that returns a close-to-optimal path, by computing the confidence intervals of the expected costs of edges.

In this chapter we give the Probably Approximately Correct BURM (PACBURM), which returns a path p_{pac}, that can be characterized as follows. Define the underlying optimal path, p_{opt}, as the path with the least expected cost from source to goal. Let $|E|$ be the number of edges in the graph. Then with probability $(1 - \alpha)^{|E|}$, $\hat{w}(p_{pac}) - \hat{w}(p_{opt}) \leq L\epsilon$. Here $\hat{w}$ is the expected cost, L is the number of edges in the optimal path, p_{opt}, and ϵ is the level of tolerance that was used. Note that L can be trivially bounded by $|V|$, the number of nodes in the graph. The tolerance ϵ, and the confidence level α, can be made arbitrarily small, but leads to increasing the running time of the algorithm.

In our experiments the value of $(1-\alpha)^{|E|}$ is maintained at > 0.90. The results indicate that by introducing the tolerance, $\hat{w}(p_{pac}) - \hat{w}(p_{opt})$ can be maintained to within 1-5% of $\hat{w}(p_{pac})$, while the running time gets reduced by up to two order of magnitude.

5.1 Introduction

In the DBURM algorithm, the expected cost of traversing an edge is not known accurately, and the edges are annotated with an interval, $W(e)$, that contains the expected cost of traversing e. Usually however we can only obtain a $(1-\alpha)$ confidence interval of the expected cost, where $\alpha > 0$. For example we may want to find out the expected time taken for a car to traverse a road. If we are given samples of time taken by cars to traverse the road, we can compute $(1-\alpha)$ confidence interval of the expected time taken. As another example, consider that we want to find the expected cost of collision for a robot on traversing an edge e. In order to do so we may run simulations and gather sample data of costs incurred. Again, using these samples we can generate a $(1-\alpha)$ confidence interval of the expected cost of traversing e.

Therefore we need to make changes to DBURM so that it handles the case when $W(e)$ is the $(1-\alpha)$ confidence interval of the expected cost of edge e. Like many graph algorithms we also assume that the edge weights are positive. In our setting this translates to: The lower bounds of the confidence intervals are positive.

With such an algorithm we could claim that when the algorithm terminates: The path that the algorithm returns is the optimal path from s to

goal, *if* for every edge e, $\hat{w}(e) \in W(e)$. That is, the expected cost of e is contained within its confidence interval. By making α sufficiently small, we can claim that the path returned is optimal with a high probability. However experiments indicate that such an algorithm can take a long time to run.

In order to make things faster we introduce a 'tolerance' ϵ. In graph search algorithms there is usually a subroutine where we compare the costs of two paths p_1 and p_2 and choose the better path. In the context of PACBURM we would take samples and hence tighten the interval costs of p_1 and p_2, until the two intervals do not intersect in their interior. After that it would be easy to choose the path with lesser cost. However, when using a tolerance of ϵ, PACBURM tightens the intervals, until the intersection of the two intervals has a width of $\leq \epsilon$. This tolerance is bound to improve efficiency, as narrowing confidence intervals needs a lot of samples, if the width of the confidence interval is already small - although we would sacrifice the quality of the returned path.

We now quantify the quality of the path returned by PACBURM. Let p_{pac} be the path returned, and p_{opt} be the optimal path; the path with the least expected cost from source to goal. Let $\hat{w}(p)$ denote the expected cost of path p. We later show that with probability $(1-\alpha)^{|E|}$, $\hat{w}(p_{pac}) - \hat{w}(p_{opt}) \leq L\epsilon$, where L is the number of edges in the optimal path, and $|E|$ is the total number of edges. Therefore PACBURM returns a path that is probably *approximately* optimal. Trivially, L can be bounded by $|V|$, the number of nodes in the graph. However this is only a theoretical limit, and experiments show that the actual regret is much smaller than $|V|\epsilon$. The tolerance ϵ, and confidence level α, can be made arbitrarily small but leads to increasing the

running time of the algorithm.

In our experiments $(1-\alpha)^{|E|}$ is maintained reasonably high at > 0.90. The experiments indicate that by introducing the tolerance ϵ, the efficiency can improve by two orders of magnitude, while $\hat{w}(p_{pac}) - \hat{w}(p_{opt})$ is kept to within 5% of $\hat{w}(p_{pac})$.

5.2 Overview of the Chapter

We first give related work in section 5.3. In section 5.4 we give the offer() subroutine of PACBURM. It is in this section that the tolerance ϵ gets defined, a key idea in searching for the probably approximately optimal path. Next, in section 5.5 we give the main PACBURM algorithm. The fact that the algorithm terminates with probability 1 is proved in section 5.6, while the statements and proofs about the quality of the path returned by PACBURM, are in section 5.7. We postponed the correctness result of DBURM in chapter 4. As all theorems and proofs of PACBURM carry to DBURM as well, we easily prove the correctness and termination of DBURM in section 5.8. We then give the experiments in section 5.9. In the last section we give a short summary of the chapter.

5.3 Related Work

With PACBURM algorithm we generalize our BURM and DBURM algorithms, and consider the general situation of searching for an optimal path in a graph where edge weights are confidence intervals of the expected cost

of the edge. We review some of the other techniques where edge costs are annotated with intervals.

5.3.1 Fuzzy shortest path problem

Shortest path problem when edge costs are not known precisely, has been studied in the domain of Fuzzy Set Theory [95]. In fuzzy set theory membership of an element x, to a set A, is denoted by a membership *value*. The greater the value the more the 'belongness' of x to A. Note that usually we only have two possibilities: $x \in A$ and $x \notin A$. Hence the indicator function of the set A, $A(x)$, which in non-fuzzy theories takes only the value 0 (if $x \notin A$), and 1 (if $x \in A$), would now take values from the range $[0, 1]$. Translating it to our problem, if an edge weight w is not known with certainty then it can be thought of as a *fuzzy number* – which is a fuzzy set with some reasonable restrictions on its indicator function [20]

Fuzzy shortest path problem was first studied in [20]. The problem was solved by defining what it means to take a minimum over a set of fuzzy number, and adapting Floyd's and Bellman-Ford algorithms [15] for the usual graphs, to the fuzzy scenario. But the minimum path length may not correspond to any real path in the given graph. The problem was circumvented by redefining the problem slightly [47]. Now the shortest path length was a fuzzy number, but each value in the fuzzy set with positive membership grade corresponds to some path in the graph.

With the difficulties as above in formulation of fuzzy shortest path, one could restrict attention to an interval [67], which is a type of fuzzy number.

In [67] an ordering of intervals was defined using both the centers, as well as width of the intervals. Namely, given intervals A and B, with centers a^c, b^c, and widths a^w, b^w respectively, the definition was: $A \preceq_{\alpha,\beta} B \Leftrightarrow b^c - a^c \geq \alpha(b^w - a^w)$ and $b^c - a^c \geq \beta(b^w - a^w)$. Then an algorithm was given to find a minimal element amongst the paths in the graph from source to destination.

In [93], given an edge e in the graph, the weight of e was a fuzzy number as before, but its indicator function was defined using the $1 - \alpha$ confidence-interval based on past statistical data of the edge weight. This indicator function was shaped like a triangular distribution, whose value was zero outside the confidence interval, and whose peak was at $\overline{c}$, the statistical mean of the past data. These fuzzy numbers were 'de-fuzzified', by defining an average over the indicator function. Hence the problem was reduced to the classical shortest path problem and solved.

5.3.2 Robust shortest paths

Suppose we don't know the edge costs precisely, but know the intervals containing the costs. Sometimes we are interested in the shortest path according to the *robust*, or the *robust deviation* criterion [49]. The 'robust' path criterion, is to find the path that minimizes the maximum path length between the origin and destination nodes, across all possible realizations of edge weights. The problem in its general form is NP-Complete [49]. The 'robust deviation' criterion is to find the path p that minimizes, the maximum difference between the length of p and the shortest path in the graph, over all possible realizations of the data. In other words, p minimizes the maximum regret.

The robust deviation criterion is also NP-Hard [99]. Despite the hardness of the problem algorithms exist [64, 43], that solve for the robust deviation problem.

5.4 Defining tolerance: The offer() sub-routine

5.4.1 Recalling some definitions

We first recall some definitions. Each edge e is annotated with an interval $W(e)$. In the context of PACBURM, $W(e)$ is a $(1-\alpha)$ confidence interval of the expected cost of traversing e. Given any interval W, we denote the upper bound by $\overline{W}$ and the lower bound by $\underline{W}$. Given a path p to goal, we define $W(p) = \sum_{e \in p} W(e)$. This sum of intervals is defined by taking the respective sums of the upper and lower bounds; in other words: $\overline{W}(p) = \sum_{e \in p} \overline{W}(e)$ and $\underline{W}(p) = \sum_{e \in p} \underline{W}(e)$. In the case that p is a sequence of edges that does not end at the goal, we define $W(p) = [\infty, \infty]$. Given any node x, $\pi(x)$ is the parent of x. During the execution of PACBURM, the current estimate of the optimal path from x to goal is the path $p(x) = (x, \pi(x), \pi^2 \dots goal)$, and is continually updated. Lastly, for every node x, we maintain an interval $C(x)$ which is an estimate of $W(p(x))$.

5.4.2 The offer() subroutine

Recall that in DBURM the offer(x, y) subroutine compares the costs of two paths from y to goal. It then updates the parent pointer of y, $\pi(y) = x$, if doing so reduces the cost of $p(y)$. In order to determine which of the two

competing paths has the smaller cost, DBURM narrows the $W(.)$ intervals of the two paths until the two intervals do not overlap. At such a time it is easy to choose the path with the smaller cost: If p_1 and p_2 are the two paths under comparison, then p_1 has the smaller cost if $\overline{W}(p_1) \leq \underline{W}(p_2)$.

Narrowing the $(1-\alpha)$ confidence intervals to such an extent can be quite expensive, and this fact is indeed borne out by our experimental results. A natural idea is to allow a 'tolerance' of ϵ: if the two intervals are intersecting over a sub-interval of width $\leq \epsilon$, then PACBURM should immediately choose the 'better' path without further tightening of intervals.

Of course with this ϵ tolerance, we can't be sure which of the two paths has the smaller cost, so we do as follows. Let p_1 and p_2 be the two paths under comparison. If $\overline{W}(p_1) \leq \underline{W}(p_2) + \epsilon$ then p_1 is considered as the path with the smaller cost. It is possible that both p_1 and p_2 satisfy this criterion. Now note line 9: we change the pointer $\pi(y)$ if $p_{offered}$ can be considered as the better path; however the current path, $p(y)$, can *not* be considered as the better path. This is to avoid needlessly changing the parent pointers.

On line 6 we take more samples. The way we implement this is by choosing that edge $e \in p_{offered} \cup p(y)$, which has the greatest width of $W(e)$. We then take a pre-specified number of samples from the cost distribution of e, before going back to the condition on line 5.

Algorithm 4 offer(Node x, Node y)

1: **if** $\pi(y) = x$ **then**
2: return.
3: Let p_{offered} be the path $y \rightarrow p(x)$.
4: collect the edges constituting p_{offered} and $p(y)$.
5: **if** The interior of the intervals $W(p_{\text{offered}})$ and $W(p(y))$ intersect **then**
6: Attempt to tighten the intervals $W(p_{\text{offered}})$ and $W(p(y))$ by taking more samples, until either $\overline{W}(p_{\text{offered}}) \leq \underline{W}(p(y)) + \epsilon$ OR $\overline{W}(p(y)) \leq \underline{W}(p_{\text{offered}}) + \epsilon$.
7: **for** All nodes z along p_{offered} and $p(y)$ **do**
8: Update $C(z)$: Set $C(z) = W(p(z))$, if $p(z)$ is a path to goal. Else set $C(z) = \infty$ and $\pi(z) = \emptyset$.
9: **if** $\overline{W}(p(y)) > \underline{W}(p_{\text{offered}}) + \epsilon$ **then**
10: Set $\pi(y) = x$ and $C(y) = W(y, x) + C(x)$

5.5 Probably Approximately Optimal path with PACBURM

In this section we give the Probably Approximately Correct BURM (PACBURM) algorithm.

Line 5 expresses the termination condition for the while loop. Each node in queue is assigned a key, $k(x)$, and $\underline{k}$ is the minimum key value among all the nodes on queue. *src* is the source node. The definition of $k(x)$ is the same as in chapter 4. The definition would be repeated again in section 5.7.

On line 8 of PACBURM, x is improving its path, $p(x)$, by looking at the offers of its neighbors. Then on line 9, x attempts to improve the paths, $p(y)$, of its neighbors by offering its current path to its neighbors.

Other than the fact that PACBURM uses an updated version of offer() (given earlier), PACBURM is basically DBURM but when the edge weight intervals, $W(e)$, are $(1 - \alpha)$ confidence intervals of the expected cost. The

Algorithm 5 PACBURM

1: make roadmap.
2: Set $C(x) = \infty$, $\pi(x) = \emptyset$ for all nodes.
3: Set $C(goal) = [0, 0]$
4: insert goal node into queue
5: **while** $(\overline{C}(src) > \underline{k})$ **do**
6: Remove a node x from queue
7: For all nodes $z \in p(x)$, set $C(z) = W(p(z))$ if $p(z)$ is a path to goal. Else set $C(z) = \infty$, and $\pi(z) = \emptyset$.
8: offer(y, x) for all neighbors y of x
9: offer(x, y) for all neighbors y of x
10: For all nodes z such that $\pi(z) = x$, set $C(z) = W(z, x) + C(x)$,
11: While running lines 6-10

1. For all nodes $z \neq x$, if $C(z)$ *modified* between lines 6-10 then insert z into queue.

2. if $\overline{C}(x)$ increased on line 8, or $C(x)$ modified on line 9, insert x back into queue.

3. If $W(e)$ modified for some edge e, insert the nodes adjacent to e into the queue.

important difference is that $\alpha \neq 0$. This calls for a couple of changes to DBURM, which are points 2 and 3 on line 11. Recall the definition of *modify* . An interval $W(.)$ is said to *modify* if it changes in a way so that $W_{new} \not\subseteq W_{old}$. We now give an intuitive reasoning for line 11 below.

1. As the edge weights are $(1 - \alpha)$ confidence intervals, they may *modify* while sampling. That is, $W_{new}(e) \not\subseteq W_{old}(e)$ for some edge e. In DBURM this could happen only when there were dynamic changes to the environment, at which point we would insert the nodes adjacent to e into the queue. In PACBURM we have to be prepared for this event at all times (see condition 3 on line 11).

2. Note that when updating the value of $C(x)$, we always set $C(x) = W(p(x))$. Now suppose the $W(e)$ intervals, where e is any edge, are only getting tighter as we sample. Then $\overline{C}(x)$ is non-increasing during line 8, as $p(x)$ may change but only if we find a path with a smaller cost. Also, $C(x)$ can only get tighter on line 9, as $p(x)$ can not change on line 9, and $W(p(x))$ can only get tighter. However, as our assumption is false and confidence intervals are not always getting tighter, these two properties might be violated. If so, we can no longer be sure that x has chosen the best 'offer' from its neighbors. Hence we insert x back into the queue. (see condition 2 of line 11).

Note that in point (1) above, what we are saying is that dynamic changes can happen in PACBURM all the time. This is why we extended PACBURM from DBURM rather than BURM, as BURM does not handle the case of dynamic changes. Most of the rest of the chapter is dedicated to proving

the termination and correctness of the PACBURM algorithm. Namely, that PACBURM 'almost surely' terminates. And when PACBURM terminates, then with a high probability, the path returned, $p(src)$, has an expected cost that is close to the cost of the optimal path.

5.6 Termination of PACBURM

The main results of this section are theorems 5.6.6 and 5.6.7. These are: that PACBURM terminates when the underlying distributions of edge costs have a bounded support; and that PACBURM 'almost surely' terminates if the distributions have unbounded support, but finite population variance.

PACBURM samples edge costs, in attempt to tighten the $(1 - \alpha)$ confidence intervals of the expected cost of edges. The question of whether PACBURM terminates, can be transformed to the easier of question of whether PACBURM terminates when $W(e)$ intervals are not changing any more. In other words the relevant confidence intervals are narrow enough, so that PACBURM does not need to sample any more.

In the first subsection we prove termination when $W(e)$ are not changing. In the second subsection we explore the question of when would $W(e)$ stop changing. In the final subsection we give the main termination theorems.

5.6.1 Termination when $W(e)$ are not changing

Theorem 5.6.1 *PACBURM terminates in a finite number of iterations if the intervals $W(e)$ are not changing.*

Proof Suppose the bounds of $W(e)$ are not changing for every edge e.

Recall that ϵ is the tolerance in offer(). Define p_1 to be a *strictly better path* than p_2, if $\overline{W}(p_1) - \underline{W}(p_2) \leq \epsilon$, *and* $\overline{W}(p_2) - \underline{W}(p_1) > \epsilon$. Denote this relation by $p_1 <_\epsilon p_2$. Note that in offer(), we update the path $p(y)$ only if we find a strictly better path (line 9 of offer()).

Suppose x is the node removed from queue on line 6. At the end of this iteration $C(x) = W(p(x))$, as we make this update at the end of every offer(). Now x would be inserted again into the queue, only if $C_{new}(x) \not\subseteq C_{old}(x)$ at some point. As $W(e)$ and $W(path)$ for any given e and *path* are not changing, these conditions can become true only if $p(x)$ is updated to a strictly better path.

Hence x would be re-inserted only if x finds a strictly better path to goal. By lemma 5.6.2 below, $<_\epsilon$ relation can not contain any cycles. Therefore x would find a strictly better path to goal only a finite number of times. So eventually x stops entering the queue. The reasoning is true for all nodes, and hence the queue eventually becomes empty, and the condition on line 5 becomes false. ∎

First we note that $<_\epsilon$ relation as defined in the proof of theorem 5.6.1 is indeed *not* transitive. As an example, take $\epsilon = 0.2$, and consider the intervals, $a = [1, 1.4]$, $b = [1.1, 1.2]$, $c = [0.1, 1.3]$. Now note that $c <_\epsilon b$, and $b <_\epsilon a$. However $c \not<_\epsilon a$. Nevertheless $<_\epsilon$ is a non-cyclical relation which is all we needed in the proof of the theorem above.

Lemma 5.6.2 *Assume $W(e)$ intervals are not changing. Then $<_\epsilon$ relation as defined in the proof of theorem 5.6.1 can not contain any cycles.*

Proof First we prove that it doesn't contain any 3-cycles. The proof for n-cycles, for any given integer n, is an easy extension of this proof.

We prove by contradiction. Suppose $p_1 <_\epsilon p_2$ and $p_2 <_\epsilon p_3$. Also, suppose that $p_3 <_\epsilon p_1$.

As $p_1 <_\epsilon p_2$, we get:

$$\text{(i) } \overline{W}(p_1) - \underline{W}(p_2) \leq \epsilon \quad \text{(ii) } \overline{W}(p_2) - \underline{W}(p_1) > \epsilon$$

As $p_2 <_\epsilon p_3$, we get:

$$\text{(iii) } \overline{W}(p_2) - \underline{W}(p_3) \leq \epsilon \quad \text{(iv) } \overline{W}(p_3) - \underline{W}(p_2) > \epsilon$$

As $p_3 <_\epsilon p_1$, we get:

$$\text{(v) } \overline{W}(p_3) - \underline{W}(p_1) \leq \epsilon \quad \text{(vi) } \overline{W}(p_1) - \underline{W}(p_3) > \epsilon$$

Let $|W(p_2)|$ denote the width of the interval $W(p_2)$. Adding the width of $W(p_2)$ to equation (v) we get:

$$\begin{aligned}
&(\overline{W}(p_3) - \underline{W}(p_2)) + (\overline{W}(p_2) - \underline{W}(p_1)) \leq \epsilon + |W(p_2)| \\
&\Rightarrow 2\epsilon < \epsilon + |W(p_2)| \\
&\Rightarrow |W(p_2)| > \epsilon
\end{aligned}$$

On the other hand, adding the width $|W(p_2)|$ to equation (vi) we get:

$$\begin{aligned}
&(\overline{W}(p_1) - \underline{W}(p_2)) + (\overline{W}(p_2) - \underline{W}(p_3)) > \epsilon + |W(p_2)| \\
&\Rightarrow 2\epsilon > \epsilon + |W(p_2)| \\
&\Rightarrow |W(p_2)| < \epsilon
\end{aligned}$$

As $|W(p_2)|$ can not be both greater and less than ϵ, this gives a contradiction, hence proving that there are no 3-cycles.

The proof that there are no n-cycles, for any given n, is very similar. Suppose $p_1 <_\epsilon p_2 <_\epsilon p_3 <_\epsilon \ldots p_{n-1} <_\epsilon p_n <_\epsilon p_1$. Now take the two equations defining $p_n <_\epsilon p_1$. Rather than adding $W(p_2)$ as we did above, we would now add $\sum_{i=2}^{n-1} W(p_i)$. Adding $\sum_{i=2}^{n-1} |W(p_i)|$ to the equation $\overline{W}(p_n) - \underline{W}(p_1) \leq \epsilon$ we get:

$$\begin{aligned}
& (\overline{W}(p_n) - \underline{W}(p_{n-1})) + \ldots (\overline{W}(p_2) - \underline{W}(p_1)) \leq \epsilon + \sum_{i=2}^{n-1} |W(p_i)| \\
\Longrightarrow \quad & (n-1)\epsilon < \epsilon + \sum_{i=2}^{n-1} |W(p_i)| \\
\Longrightarrow \quad & \sum_{i=2}^{n-1} |W(p_i)| > (n-2)\epsilon \qquad (5.1)
\end{aligned}$$

On the other hand when we add $\sum_{i=2}^{n-1} |W(p_i)|$ to the equation $\overline{W}(p_1) - \underline{W}(p_n) > \epsilon$, we get:

$$(\overline{W}(p_1) - \underline{W}(p_2)) + \ldots (\overline{W}(p_{n-1}) - \underline{W}(p_n)) > \epsilon + \sum_{i=2}^{n-1} |W(p_i)|$$

$$\implies \quad (n-1)\epsilon > \epsilon + \sum_{i=2}^{n-1} |W(p_i)|$$

$$\implies \quad \sum_{i=2}^{n-1} |W(p_i)| < (n-2)\epsilon \tag{5.2}$$

Together it implies that $\sum_{i=2}^{n-1} |W(p_i)|$ is both less than and greater to $(n-2)\epsilon$. A contradiction. ∎

5.6.2 When do $W(e)$ stop changing?

We proved that if the edge costs are not sampled, then PACBURM terminates in a finite number of steps. This leads to the question, when does the algorithm stop sampling costs of a given edge.

Denote the width of an interval W by $|W|$. Recall that ϵ is the tolerance we are using in PACBURM. We denote the number of vertices in the graph by $|V|$. We first observe the following.

Lemma 5.6.3 *The cost of edge e is not sampled if $|W(e)| \leq \epsilon/|V|$. Hence $W(e)$ does not change once $|W(e)| \leq \epsilon/|V|$.*

Proof In the offer() sub-routine, when comparing the costs of paths p_1 and p_2, we sample the cost of that edge $e_k \in (p_1 \cup p_2)$, whose width $|W(e_k)|$ is the greatest. However if this greatest width is $\leq \epsilon/|V|$, then $|W(p_1)|$ and $|W(p_2)|$ are both $\leq \epsilon$. Hence $W(p_1)$ and $W(p_2)$ can not overlap in a subinterval with width greater than ϵ. But in such a case, offer() chooses the 'better' path

without sampling. This shows that if $|W(e_k)| \leq \epsilon/|V|$, then its cost is not sampled. ∎

By the lemma above, we have changed the question 'Do $W(e)$ stop changing in a finite time?' to 'Do we have $|W(e)| \leq \epsilon/|V|$ for all edges e, after some finite time?' The answer to this question depends on the way confidence intervals, $W(e)$ are being computed. If the width of $W(e)$ for every e converges to 0, then PACBURM would terminate.

For simplicity, we assume that the $(1-\alpha)$ confidence intervals are approximated by the popular technique: $[\bar{x} - z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}} \ , \ \bar{x} + z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}]$. Here S is the sample variance, n_s is the number of samples, and $\bar{x}$ is the sample mean. $z_{\frac{\alpha}{2}}$ is the usual notation for value of the inverse CDF of Standard Normal distribution at $\alpha/2$. Therefore in PACBURM $W(e)$ would be equal to this interval.

We consider two cases: (a) Probability distributions of edge costs have bounded support (b) Probability distributions have unbounded support, but a finite variance.

Lemma 5.6.4 *The width of the interval* $[\bar{x} - z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}} \ , \ \bar{x} + z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}]$ *converges to* 0 *at a rate of* $O(1/\sqrt{n_s})$ *if the distribution has bounded support.*

If the distribution has unbounded support and finite population variance, then this width 'almost surely' converges to 0*. In other words, given any positive number* δ*, there exists an integer* N*, such that after* N *samples,* $pr(|W| > \delta) < \delta$*. Here* $|W|$ *denotes the width of the interval above.*

Proof If the distribution has bounded support then the sample variance S is bounded. It is straightforward to see that the width of the interval in that

case is proportional to $1/\sqrt{n_s}$.

If the distribution has unbounded support, then the sample variance can be made arbitrarily large. However that happens with a probability measure of 0: By lemma 5.6.5 below, S 'almost surely' converges to σ. Here σ is the population variance, and is a finite number. Hence the width of the interval almost surely converges to $K\sigma/\sqrt{n_s}$, which converges to 0 as n_s increases. The conclusion follows. ∎

==========

Following is a definition, and the lemma that was used in the proof of theorem 5.6.4.

Definition (Convergence in Probability) Let $\{X_n\}$ be a sequence of random variables and let X be a number. We say that X_n converges in probability to X if for all $\epsilon > 0$

$\lim_{n\to\infty} P(|X_n - X| \geq \epsilon) = 0$. If so, we write $X_n \xrightarrow{P} X$

Next we quote a known result (see pp. 206 of [30])

Lemma 5.6.5 *Let S_n^2 be the sample variance after n samples. Assume $E(x_1^2) < \infty$, where $E(x_1^2)$ is the expected value of the square of one sample. Then $S_n^2 \xrightarrow{P} \sigma^2$.*

==========

5.6.3 The main Termination theorems

As above, for this section we assume that the confidence interval $W(e)$ is approximated by the interval: $W(e) = [\bar{x} - z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}} \ , \ \bar{x} + z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}]$. As above,

we consider two cases (a) The probability distribution has bounded support (b) The distribution has unbounded support but finite variance.

Theorem 5.6.6 *Suppose the distributions of the edge costs have bounded support, and that confidence intervals are approximated by the interval* $W(e) = [\bar{x} - z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}} \quad , \quad \bar{x} + z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}]$. *Then PACBURM terminates.*

Proof A direct result of the first part of lemma 5.6.4.

Inthe case of unbounded support but finite variance, by lemma 5.6.4, the width $|W(e)|$ converges almost surely to 0. We hence get the following result.

Theorem 5.6.7 *Suppose that the underlying cost distributions of edges can have unbounded support, but finite population variance. Also that* $W(e) = [\bar{x} - z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}, \bar{x} + z_{\frac{\alpha}{2}}.\frac{S}{\sqrt{n_s}}]$. *Then PACBURM almost surely terminates.*

Proof We know that the width of each $W(e)$ converges in probability to 0. That is, for every edge e_i, there exists a number n_i, such that after n_i samples of cost of e_i, $pr(|W(e_i)| \leq \epsilon/|V|) > (1-\delta)^{1/|V|}$. Therefore by lemma 5.6.3, $W(e_i)$ stops changing after n_i samples with a probability of $(1-\delta)^{1/|V|}$. This is true for every edge. Hence after $\sum_i n_i$ samples, with a probability of at least $(1-\delta)$, the width of *every* $W(e)$ is $\leq \epsilon/|V|$. Hence by theorem 5.6.1 PACBURM terminates in a finite number of steps after taking $\sum_i n_i$ samples. ∎

5.7 Quality of path returned by PACBURM

Having dealt with termination of PACBURM we now want to find out the quality of the path returned when PACBURM terminates. This section is

devoted to proving theorem 5.7.1 (and hence corrollary 5.7.2), which are given below.

Let $\hat{w}(path)$ be the expected cost of traversing the path. Let p_{opt} be the optimal path: the path from source to goal with the least expected cost. Let L be the number of edges in p_{opt}, and ϵ be the tolerance that we are using in offer(). Recall that the returned path by PACBURM is $p(src)$, a path from src to goal. In this section we prove the following theorem.

Theorem 5.7.1 *Suppose that $W(e)$ are the $(1-\alpha)$ confidence intervals of the mean cost of edge e. Then when PACBURM terminates, with probability of at least $(1-\alpha)^{|E|}$ we have: $\hat{w}(p(src)) - \hat{w}(p_{opt}) \leq L\epsilon$.*

Note that L above can be trivially bounded by n, the number of nodes in the graph, so we have as corollary:

Corollary 5.7.2 *Suppose $W(e)$ are the $(1-\alpha)$ confidence intervals of the mean cost of edge e. Then when PACBURM terminates, with probability of at least $(1-\alpha)^{|E|}$ we have: $\hat{w}(p(src)) - \hat{w}(p_{opt}) \leq n\epsilon$, where n is the number of nodes in the graph.*

5.7.1 Proof

In order to prove theorem 5.7.1 we prove the following which easily implies the statement of theorem 5.7.1.

Theorem 5.7.3 *Suppose to each edge e we assign a weight $w(e)$, such that $w(e) \in W(e)$. Let p_{pac} be the path returned by PACBURM. Then for any*

path p_{alt} from src to goal, $w(p_{pac}) - w(p_{alt}) \leq L\epsilon$, where L is the number of edges in p_{alt}.

For the remainder of this section, s is the source node, p_{pac} is the path returned by PACBURM, p_{alt} is an alternate path from s to goal, L is the number of edges in p_{alt}. For definitions of $p(x)$, $C(x)$, $\pi(x)$ and $W(.)$, see section 5.4.1. We also suppose that PACBURM has terminated, and we have assigned a cost $w(e) \in W(e)$ for each edge e. $w(path)$ is defined as $\sum_{e \in path} w(e)$.

Note that when PACBURM terminates, $p_{pac} = p(s)$. The reason for introducing the extra term p_{pac} would become clear below.

==========

***Idea of the proof**: Suppose we change $p(s)$ after termination of PACBURM by changing some of the parent pointers $\pi(.)$. Note that we need to change at most L parent pointers to make $p(s) = p_{alt}$. Now when we compare two path in offer(), we choose the better path within a tolerance of ϵ. This implies that if 'relevant' paths have been compared, then each time we change a pointer $\pi(.)$, the cost of $w(p(s))$ would reduce by at most ϵ. Hence if all 'relevant' paths have been compared, then indeed $w(p_{pac}) - w(p_{alt}) \leq L\epsilon$.*

However, if the latest costs of all 'relevant' paths have not *been compared, then we can't use the above reasoning. But in this case some of the nodes along these paths would be on queue. Now if a node is on queue, then a path containing that node, can not have a small cost. Therefore in this case too, we will show that the regret can be bounded by $L\epsilon$.*

==========

We are going to break the proof of 5.7.3 into three sub-cases. Case (A) is when for every node x in p_{alt}, $p(x)$ does not contain any node on the queue. In the language used above, this is the case when "all relevant paths have been compared."

CASE (A)

We begin by a lemma which bounds the improvement of $w(p(s))$ when we change a single parent pointer.

Lemma 5.7.4 *Suppose when PACBURM terminates, $p(x)$ and $p(y)$ do not contain any nodes on queue. Now if we change $\pi(x)$ and assign $\pi(x) = y$. Then $w(p(x)) \leq w(p_{new}(x)) + \epsilon$. Here $p_{new}(x)$ is the update path of $p(x)$ after we change $\pi(x)$.*

Proof By corollary 5.7.12, if $p(x)$ and $p(y)$ do not contain any nodes on queue, and $\pi(x) \neq y$, then $\overline{W}(p(x)) \leq \underline{W}(x, y) + \underline{W}(p(y)) + \epsilon$. This implies the statement of the lemma. ∎

Now we claim the following.

Theorem 5.7.5 *If for every node x in p_{alt}, $p(x)$ does not contain any nodes on queue, then $w(p_{pac}) \leq w(p_{alt}) + L\epsilon$.*

Proof We will use the idea in 5.7.4. In this graph we would change enough parent pointers $\pi(.)$ so that $p(s)$ changes from p_{pac} to p_{alt}. Note that we need to change at most L pointers. For let p_{alt} be the path $z_L, z_{L-1}, z_{L-3} \ldots z_1, g$, where z_L is the source node, s and g is the goal node. Now change the pointers sequentially and one by one as follows: $\pi(z_1) = goal$; $\pi(z_2) = z_1$

; $\pi(z_3) = z_2 \ldots \pi(z_L) = z_{L-1}$. Note that by doing it in this sequence, we will never introduce a loop in the graph. Also note, that at the end of these changes, $p(s) = p_{alt}$.

Denote by $p_{new}(z_i)$, the path $p(z_i)$ after i updates to the $\pi(.)$ pointers in the sequence of updates. We now prove by induction that in the sequence above, when we change the pointer $\pi(z_n)$ to $\pi(z_{n-1})$, then $w(p_{old}(z_n)) \leq w(p_{new}(z_n)) + n\epsilon$.

The base case is proved in 5.7.4, when we make just a single change: $\pi(z_1) = goal$. Now suppose that the above statement is true after $n-1$ pointer updates. When PACBURM terminated, by corrolary 5.7.12, $w(p_{old}(z_n)) \leq w(z_{n-1}, z_n) + w(p_{old}(z_{n-1})) + \epsilon$. By the inductive hypothesis, after $n-1$ pointer updates we have $w(p_{old}(z_{n-1})) \leq w(p_{new}(z_{n-1})) + (n-1)\epsilon$. This easily implies that by changing $\pi(z_n) = z_{n-1}$, we would have $w(p_{old}(z_n)) \leq w(p_{new}(z_n)) + n\epsilon$

∎

For Case(A) we have assumed that for every node x in p_{alt}, $p(x)$ does not contain any nodes on queue. Before going on to the other cases, we note if PACBURM terminates only when queue becomes empty, then the hypothesis that $p(x)$ does not contain any nodes on queue, would be true for every node x. Hence the above would be the only case applicable.

We now go on to other cases when p_{alt} does *not* satisfy the stated hypothesis above. That is, for the remaining cases, p_{alt} contains a node x, such that $p(x)$ includes some nodes on queue. We recall the definitions of h and $k(x)$. We sort the nodes in the queue based on their key $k(x)$.

Definition : $h(x)$ Consider those iterations of the while loop (on line 5),

when the given node x was chosen for removal on line 6. There could be several such iterations. Let L_x be the most recent of such iterations. Let t be the time when we just finished executing line 7 in L_x, or $t = -\infty$ if there was no such iteration. We define $h(x)$ as the minimum value of $\overline{C}(x)$ since t.

Definition : k(x)

$$k(x) = \min\{h(x), \underline{C}(x)\} \tag{5.3}$$

Definition $\underline{k}$ This is defined as the minimum value of $k(x)$ amongst all nodes on queue.

CASE (B) Suppose we traverse from goal, backwards towards source, along p_{alt}. For the current case we are going to assume that the first node along this traversal that does not satisfy the hypotheses in (A), is in queue. Let this node be y. Let p_{alt} be the path $s \to \ldots y \to x \ldots g$. That is, x is the next node after y towards goal. Note that by definition of y, x satisfies the hypotheses of case (A).

Subcase I:

For this subcase, suppose when PACBURM terminated $\pi(y) = x$. As $p(x)$ does not include any nodes on queue, $p(y)$ includes no nodes (other than y) on queue. By lemma 5.7.6 we know $C(y) \supseteq W(p(y))$. Therefore, $\underline{C}(y) \leq \underline{W}(p(y))$. Secondly, by lemma 5.7.9 when the while loop terminates, $p(s)$ does not include any nodes on queue, and hence by lemma 5.7.6 $\overline{W}(p(s)) \leq \underline{k}$. So we get $\overline{W}(p(s)) \leq \underline{k} \leq \underline{C}(y) \leq \underline{W}(p(y))$.

Note that the nodes in $p_{new}(x)$ satisfy the hypotheses in (A). Therefore $w(p(x)) \leq w(p_{new}(x)) + (L-1)\epsilon$, because $p_{new}(x)$ contains at most $L-1$

edges. We now obtain

$$\begin{aligned}
\overline{W}(p(s)) &\leq \underline{W}(p_{old}(y)) \\
&\leq w(y,x) + w(p(x)) \ \textit{(because } \pi(y) = x\textit{)} \\
&\leq w(y,x) + w(p_{new}(x)) + (L-1)\epsilon \\
&\leq w(p_{alt}) + (L-1)\epsilon \\
\Rightarrow \ & w(p(s)) - w(p_{alt}) \leq L\epsilon
\end{aligned}$$

As the path that PACBURM returns is $w(p(s))$, the last line easily implies the result.

Subcase 2:

Now suppose $\pi(y) \neq x$. Let $p_2 = y \rightarrow p(x)$. Now by theorem 5.7.8, $h(y) \leq \underline{W}(p_2) + \epsilon$. As $y \in \textit{queue}$ this means, $\underline{k} \leq \underline{W}(p_2) + \epsilon$. From subcase I of case (B) we know $\overline{W}(p(s)) \leq \underline{k}$. So we get $\overline{W}(p(s)) \leq \underline{k} \leq \underline{W}(p_2) + \epsilon$. We now get,

$$\begin{aligned}
\overline{W}(p(s)) &\leq \underline{W}(p_{alt}) + \epsilon \\
&= w(p(x)) + w(y,x) + \epsilon \\
&\leq w(p_{new}(x)) + (L-1)\epsilon + w(y,x) + \epsilon \\
&\leq w(p_{alt}) + L\epsilon \\
\Rightarrow \ & w(p(s)) - w(p_{alt}) \leq L\epsilon
\end{aligned}$$

The second last line follows because $p_{new}(x) \cup (y, x) \subset p_{alt}$.

CASE (C) The only remaining possibility is as follows. Let y be the first node along p_{alt} as we traverse backwards from goal to src, that does not satisfy the hypotheses of case (A), *as well as* it is not on queue. Hence there must be a node along $p(y)$ that is on queue.

Again suppose that $p_{alt} = s \ldots y \rightarrow x \ldots goal$. Note that in this case $\pi(y) \neq x$ (Else x can not satisfy the hypotheses of case (A) - contradiction). Now let $p_2 = y \rightarrow p(x)$. We observe the following three facts:

1. By lemma 5.7.9, $\overline{C}(y) \geq \underline{k}$.

2. By lemma 5.7.8, $h(y) \leq \underline{W}(p_2) + \epsilon$.

3. As $y \notin queue$, by lemma 5.7.10 $\overline{C}(y) = h(y)$.

Hence we get, $\underline{k} \leq \overline{C}(y) = h(y) \leq \underline{W}(p_2) + \epsilon$. From subcase I of case (B) we know $\overline{W}(p(s)) \leq \underline{k}$. Hence just like the subcase II in case (B), we get $\overline{W}(p(s)) \leq \underline{k} \leq \underline{W}(p_2) + \epsilon$. The remaining analysis is identical to subcase II of (B).

The three cases (A), (B), (C) together, hence prove theorem 5.7.3, and hence theorem 5.7.1.

5.7.2 Lemmas Used

In this section we prove a few results that were quoted in the proof of theorem 5.7.1. Note that the statement of all lemmas and theorems assumes that the state of execution is not 'inside' the while loops. That is, the program is just about to start a new iteration on line 5. For definnitions of some terms refer

to 5.4.1. For definitions of $h(x)$, $k(x)$, $\underline{k}$, refer to section 5.7. Recall that an interval W is said to *modify*, if it changes in a way such that $W_{new} \not\subseteq W_{old}$.

We first observe that the proof of lemma 4.6.1 carries to PACBURM without change; i.e. it remains identical. Hence we can use lemma 4.6.1 for our results.

Lemma 5.7.6 *If $p(x)$ is a path to goal, and $p(\pi(x))$ does not include any nodes on queue, then $C(x) \supseteq W(p(x))$.*

Proof We prove by lemma 4.6.1 and induction along $p(x)$. The statement of the theorem is certainly true if $x = goal$, as $C(goal) = W(p(goal)) = [0, 0]$. Now assume that the statement is true for $\pi(x)$ and we would like to show that this implies that it is also true for x.

By inductive hypothesis $C(\pi(x)) \supseteq W(p(\pi(x))$. As $\pi(x) \notin queue$, by lemma 4.6.1, $C(x) \supseteq W(x, \pi(x)) + C(\pi(x))$. Hence $C(x) \supseteq W(x, \pi(x)) + W(p(\pi(x))) = W(p(x))$ ∎

In the proofs below when we say x *offers* a path to y we mean performing offer(x, y). When we say that y rejects the path from x, we mean that $\pi(y) \neq x$ at the end of offer(x, y). Or when $\pi(y) = x$, is updated to $\pi(y) \neq x$.

Also given a node x, by L_x we denote the most recent iteration of the while loop on line 5 when x was the node chosen for removal on line 6. By t_x we denote the time when we just finished executing line 7 in the iteration L_x.

Lemma 5.7.7 *If $x \notin queue$, and $\pi(y) \neq x$ then $h(y) \leq \underline{W}(y, x) + \underline{C}(x) + \epsilon$.*

Proof For each neighbor x of y, such that $\pi(y) \neq x$, either y rejected the path from x at some point, otherwise such a path was never offered.

Suppose x never offered a path to y, then x never left the queue. So either x never entered the queue, and hence $C(x) = \infty$. Or x is still in queue, which contradicts the hypothesis of the lemma. Hence we conclude that if y never rejected the path from x, then the statement of the lemma is satisfied as $C(x) = \infty$

Hence suppose that y rejected the path from x at some point. Define t_i as the most recent time when y rejected the path from x. Hence at time t_i we had:

$$\overline{C}_{t_i}(y) \leq \underline{W}_{t_i}(y, x) + \underline{C}_{t_i}(x) + \epsilon$$

Here ϵ is the maximum tolerance allowed in offer(), and the subscripts t_i denote the cost-intervals at time t_i.

Note that $t_i > t_y$, as immediately after t_y, y looks at all offers from its neighbors on line 8. As $h(y)$ is the minimum value of $\overline{C}(y)$ since t_y, and $t_i > t_y$, we have: $h(y) \leq \overline{C}_{t_i}(y)$. Therefore $h(y) \leq \underline{W}_{t_i}(y, x) + \underline{C}_{t_i}(x) + \epsilon$.

It suffices to prove now that $W(y, x)$ and $C(x)$ have not modified since t_i. But note that t_i must have occurred *after* line 8 in L_x, as x offers its path to all neighbors on line 9. Now $C(x)$ or $W(y, x)$ can't modify after line 8 of L_x, else x would be inserted back into the queue, contradicting the definition of L_x. This implies $W(y, x)$ and $C(x)$ have not modified since t_i. ∎

Lemma 5.7.8 *If $p(x)$ does not include any nodes on queue, and $\pi(y) \neq x$ then $h(y) \leq \underline{W}(y, x) + \underline{W}(p(x)) + \epsilon$.*

Proof Note that if $p(x)$ is not a valid path to goal, then we defined $W(p(x))$ as ∞, and the statement of the theorem is true. So we consider only the case when $p(x)$ is a valid path to goal.

We know by lemma 5.7.7 that $h(y) \leq \underline{W}(y,x) + \underline{C}(x) + \epsilon$. All that needs to be shown is that $\underline{C}(x) \leq \underline{W}(p(x))$. This is true by lemma 5.7.6 ∎

Lemma 5.7.9 *If $p(x)$ includes a node on queue, then $\overline{C}(x) \geq \underline{k}$*

Proof The conclusion is trivial if $x \in$ *queue*; or if $\pi(x) = \emptyset$ (and hence $C(x) = \infty$). So assume $x \notin$ *queue* and $\pi^n(x)$ is the first node from x to goal along $p(x)$, that is on queue. We claim:

$$\overline{C}(x) > \overline{C}(\pi(x)) \ldots > \overline{C}(\pi^{n-1}(x)) > \overline{C}_{old}(\pi^n(x)) \geq k(\pi^n(x)) \geq \underline{k}$$

Let t' be the time when we most recently performed the update $C(\pi^{n-1}(x)) = W(\pi^{n-1}(x), \pi^n(x)) + C(\pi^n(x))$. By C_{old} above we denote the cost-interval at the time t'.

Now to prove the above. Till we reach π^{n-1} none of the nodes are on queue. Hence we can use lemma 4.6.1 to keep on concluding the above inequalities, until (and including) the inequality $\overline{C}(\pi^{n-2}(x)) > \overline{C}(\pi^{n-1}(x))$. As no update of $C(\pi^{n-1})$ could have been performed since t', we also conclude the next inequality: $\overline{C}(\pi^{n-1}(x)) > \overline{C}_{old}(\pi^n(x))$.

Now it suffices to prove $\overline{C}_{old}(\pi^n(x)) \geq k(\pi^n(x))$. Recall that $h(\pi^n(x))$ is defined as the minimum value of $\overline{C}(\pi^n(x))$ since a certain time t. It is easy to verify that $t' > t$ (see line 10 of PACBURM). Therefore $h(\pi^n(x)) \leq \overline{C}_{old}(\pi^n(x))$. As $k(\pi^n(x)) \leq h(\pi^n(x))$ by definition, we get: $k(\pi^n(x)) \leq$

$h(\pi^n(x)) \leq \overline{C}_{old}(\pi^n(x))$. The last inequality follows from the definition of $\underline{k}$.

∎

Lemma 5.7.10 *If $y \notin$ queue, then $\overline{C}(y) = h(y)$.*

Proof $\overline{C}(y)$ has been non-increasing since time t_y, the end of line 7 of L_y. Else y would be inserted into queue again contradicting the definition of L_y. As $h(y)$ is defined as the minimum value of $\overline{C}(y)$ since t_y, we get $\overline{C}(y) = h(y)$.

∎

Lemma 5.7.11 *Suppose $p(x)$ and $p(y)$ do not contain any nodes on queue, and that $\pi(x) \neq y$. Then $\overline{C}(x) \leq \underline{W}(x, y) + \underline{C}(y) + \epsilon$.*

Proof If x and y were never inserted into the queue, then $C(x) = C(y) = \infty$ and the statement is true. So we can assume that at least one of the nodes was inserted into queue. As currently x and y are not in queue, the nodes must have been removed.

Let t be the most recent time when offer(y, x) ended. Note that y was a neighbor of x during this most recent offer(y, x). Else the edge weight $W(x, y)$ must have modified and x and y would be inserted into queue. This contradicts the definition of t. Our basic argument below rests on the fact that x or y could not have been inserted into queue after t, as that would contradict the definition of t.

First suppose that $\pi(x) \neq y$ at time t. Therefore, $\overline{C}(x) \leq \underline{W}(x, y) + \underline{C}(y) + \epsilon$ at time t. Note that t would have happened when x or y was being removed from the queue. If it was x, then it would have happened on line 8. If it was y then it was on line 9. In the latter case, neither $C(x)$ nor $C(y)$

have modified since t, else they will be inserted again into queue. Hence the inequality is still true. In the case when offer(y, x) happened on line 8, $\overline{C}(x)$ could not have increased since t, and $C(y)$ has not modified since t (else the node would be inserted again into queue). Again we see that the inequality must still be true.

Now suppose that at time t, $\pi(x) = y$. Then $\pi(x)$ must have been updated to $\pi(x) \neq y$ at a time $t' > t$. Without loss of generality, assume t' is the most recent such time. At t' the inequality $\overline{C}(x) \leq \underline{W}(x, y) + \underline{C}(y) + \epsilon$ would be true. Now at t', $C(x) \neq \infty$, else we must have updated $\overline{C}(y) = \infty$ at some time after t, and y would be inserted into queue again.

Therefore at time t', $\pi(x) = z$,where $z \neq y$, and this must have happened in offer(z, x). Now offer(z, x) can happen when either x is getting removed or when z is getting removed. In the latter case, $C(x)$ and $C(y)$ have not modified since t', and hence the inequality is still true. In the former case, $\overline{C}(x)$ can not increase after t', and $C(y)$ can not modify since t'. Hence again we see that the inequality must still be true. ∎

Note that in the statement of the above theorem, we can now easily change $C(x)$ and $C(y)$ to $W(p(x))$ and $W(p(y))$, which gives the following corollary.

Corollary 5.7.12 *Suppose $p(x)$ and $p(y)$ do not contain any nodes on queue and $\pi(x) \neq y$. Then $\overline{W}(p(x)) \leq \underline{W}(x, y) + \underline{W}(p(y)) + \epsilon$*

Proof By lemma 5.7.6, $C(x) \supseteq W(p(x))$ and $C(y) \supseteq W(p(y))$. Also, by lemma 5.7.11, $\overline{C}(x) \leq \underline{W}(x, y) + \underline{C}(y) + \epsilon$. Combining these two, gives the result. ∎

5.8 Termination and Correctness of DBURM

In chapter 4 we postponed the proof of the correctness of DBURM when the queue is prioritized by the key $k(x)$ (section 4.7). We are now in a position to give its proof.

One could verify that all proofs that we have given in this chapater carry to DBURM algorithm without change, as the algorithms are almost identical. We note the following:

1. Points 2 and 3 on line 11 of PACBURM are redundant in the DBURM setting, as these events can not happen in the while loop on line 5 of DBURM. This is because in this while loop, edge weights $W(e)$ can not *modify* . With points 2 and 3 on line 11 removed, it makes the while loops on lines 5 of DBURM and PACBURM, identical.

2. Edge weights in DBURM can indeed modify , but outside the while loop on line 5. Whenever $W(e)$ modifies we insert the nodes adjacent to e, into the queue. The fact that we are inserting these nodes into the queue in a different place in the code, does not change any of the proofs in this chapter. The proofs use the fact that we are inserting the nodes, only the following way: " ... $W(x, y)$ could not have modified since time t, else x would have been inserted into the queue, which contradicts ...".

Hence the theorems that we proved for PACBURM carry over to DBURM. However in DBURM, the tolerance $\epsilon = 0$. We hence have the following theorems:

Theorem 5.8.1 *DBURM terminates.*

Proof The intervals $W(e)$ in DBURM can only become narrower, and after a pre-specified number of samples, N, the width of $W(e)$ becomes 0. In other words, $W(e)$ becomes a value. Now we can use theorem 5.6.1 to conclude that DBURM terminates. ∎

Theorem 5.8.2 *When DBURM terminates, the path returned, $p(src)$ has the least expected cost amongst all paths from src to goal.*

Proof We know theorem 5.7.1 is also valid for DBURM. However for DBURM, in the statement of theorem 5.7.1, we should set $\alpha = 0$, and $\epsilon = 0$. This gives us the desired result. Now to explain why should we set them to 0.

Let $\hat{w}(e)$ denote the expected cost of traversing the edge e. We set $\alpha = 0$, as the intervals $W(e)$ in DBURM contain $\hat{w}(e)$, with certainty (probability 1). And we set $\epsilon = 0$, as the tolerance is 0 in DBURM. ∎

5.9 Experiments

For reference in the ensuing paragraphs, we first recall the statement of theorem 5.7.1:

Theorem 5.7.1: Suppose that $W(e)$ are the $(1 - \alpha)$ confidence intervals of the mean cost of edge e. Then when PACBURM terminates, with probability of at least $(1 - \alpha)^{|E|}$ we have: $\hat{w}(p(src)) - \hat{w}(p_{opt}) \leq L\epsilon$. ∎

Here $\hat{w}(p(src))$ is the expected cost of the path returned by PACBURM, and $\hat{w}(p_{opt})$ is the expected cost of the optimal path from source to goal - i.e.

the path with the least expected cost. Also $|E|$ is the total number of edges in the graph.

We quickly recall the basic working principle of PACBURM. The edge weights, $W(e)$ are $(1-\alpha)$ confidence intervals of the expected cost of the edge e. When comparing two paths, PACBURM narrows the two cost intervals until the intervals do not intersect in their interior (if $\epsilon = 0$). These intervals are narrowed by taking more samples. The algorithm can be made faster in two ways: (a) Allow the intervals to intersect by a length of ϵ. (b) Choose a larger value of α. In some sense, both of these options are shrinking the intervals, so that separating out the intervals becomes easier for PACBURM.

There are two aims of these experiments.

1. Firstly, to show that when we vary the tolerance level ϵ, the time required to search for the path reduces by around two orders of magnitude. Also, the error incurred, $\hat{w}(p(src)) - \hat{w}(p_{opt})$, remains small.

2. Secondly, to show that varying the confidence level $(1-\alpha)$, surprisingly, is not equally beneficial to running time. For example by varying $(1-\alpha)$ from $(1 - 10^{-6})$ to $(1 - 10^{-3})$, the running time does improve - it runs twice as fast. However this is still not comparable to what we observe when varying ϵ.

Note that $\alpha = 10^{-3}$ and $\alpha = 10^{-6}$ are *very* different scenarios. Recall that in theorem 5.7.1 we stated that we can be sure of our statement only with a probability of $(1 - \alpha)^{|E|}$, where $|E|$ is the number of edges in the graph. Now as an example, taking $|E|$ to be 5000 we get, $(1 - 10^{-3})^{|5000|} = 0.006$ and $(1 - 10^{-6})^{|5000|} = 0.995$.

We explain now why changing α is not that beneficial. Although the two values of α above lead to very different probabilities, but the confidence intervals are not impacted by as much. When $\alpha = 10^{-6}$, the confidence intervals are about 1.5 times wider than when $\alpha = 10^{-3}$. This difference is significant when the confidence intervals are wide. However if we have taken enough samples, and the confidence intervals are already narrow, then allowing a tolerance of ϵ is much more effective. Also, it is when the intervals are already narrow that narrowing them further takes a large number of samples - it is these tight comparisons that we want to make faster, and ϵ tolerance achieves that.

5.9.1 Computing an upper bound of $\hat{w}(p(src)) - \hat{w}(p_{opt})$

We will call $\hat{w}(p(src)) - \hat{w}(p_{opt})$ as 'regret'.

As we vary the ϵ in the experiments we will show that regret remains small. By theorem 5.7.1 one could bound it by $n\epsilon$ where n is the number of nodes, but as that is an over estimate, in this sub-section we will explain how we compute a much tighter bound.

When PACBURM terminates, each edge has an edge weight $W(e)$, which is a confidence interval of the expected cost of the edge. From the statement of theorem 5.7.1, we know we can assume that the expected cost of traversing every edge, $\hat{w}(e)$, is such that $\hat{w}(e) \in W(e)$.

With this assumption in place, we compute an upper bound of regret as follows. Denote for any interval W, its upper bound by $\overline{W}$, and its lower bound by $\underline{W}$. After PACBURM terminates, it returns the path $p(src)$ from

source to goal. We then assign a value $w'(e)$ to every edge, such that $w'(e) = \overline{W}(e)$, if $e \in p(src)$. And $w'(e) = \underline{W}(e)$ otherwise. We then run Dijkstra's algorithm to compute the shortest path p', from source to goal in this graph. And finally we compute $(w'(p(src)) - w'(p'))$. We claim this is an upper bound of the maximum regret because:

$$w'(p(src)) - w'(p') \geq w'(p(src)) - w'(p_{opt}) \geq \hat{w}(p(src)) - \hat{w}(p_{opt})$$

The first inequality is true as p' is the optimal path when we assign edge weights $w'(e)$. The second inequality is basically true because $\hat{w}(e) \in W(e)$. We give a more detailed reasoning below.

Lemma 5.9.1 $\hat{w}(p(src)) - \hat{w}(p_{opt}) \leq w'(p(src) - w'(p_{opt})$

Proof The inequality can be proved by contradiction. Suppose that $\hat{w}(p(src)) - \hat{w}(p_{opt}) > w'(p(src) - w'(p_{opt})$. Now $\hat{w}(p(src)) - \hat{w}(p_{opt}) = \sum_{s1} \hat{w}(e_i) - \sum_{s2} \hat{w}(e_i)$. Here $S1$ is the set of all edges such that $e \in p(src), e \notin p_{opt}$, and the $S2$ is the set of edges where $e \in p_{opt}, e \notin p(src)$. But $w'(e) \geq \hat{w}(e)$ for all $e \in S1$; and $w'(e) \leq \hat{w}(e)$ for all $e \in S2$, and we reach a contradiction. ∎

Hence $(w'(p(src)) - w'(p'))$ is an upper bound of maximum regret, as we earlier claimed.

5.9.2 Graph Generation

We generated nodes sampled at random. The source and the goal nodes were not fixed but were chosen randomly for each individual run, as we want

to show that PACBURM performs better under a variety of circumstances. The nodes were generated in an 'environment' shown below. Here the nodes and edges were accepted only if they were not in collision with the gray colored obstacles. We introduced the environmet to provide structure to the graph. We also ran experiments over 'graphs without environment' but are not including the results, as the results remained very similar.

The distribution of the edge costs, was a truncated Gaussian distribution. For each edge, the mean of this distribution was a random number between the Euclidean length of the edge, and 4 times the Euclidean length. The variance was a random number between 0 and $1/3 \times$ the mean of the distribution. As we assume positive edge costs, this Gaussian distribution was truncated to force positive samples at all times: if a sample from the gaussian distribution $< 10^{-6}$, then we assume that the sample had a value of 10^{-6}. This is to ensure that we have 'positive' edge weights at all times.

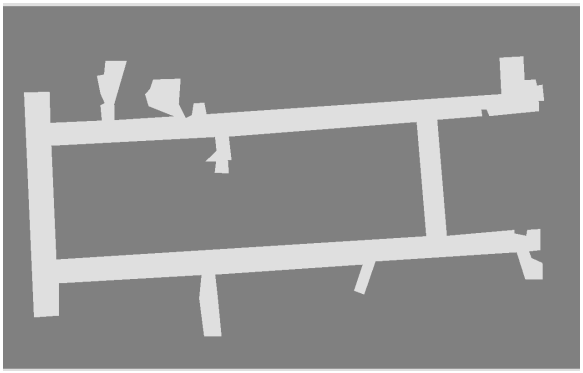


Figure 5.1: Environment over which the graph was generated

Table 5.1: Varying ϵ with 1000 nodes

ϵ	regret	frac. regret	T(s)
0.10	0.00	0.0000	191.70
1.00	0.10	0.0001	5.75
10.00	3.87	0.0039	0.76
20.00	13.01	0.0107	0.50
50.00	56.81	0.0637	0.29

5.9.3 Experimental Results

In this section, all numbers quoted in the tables are averages over 30 runs.

Varying ϵ

Table 5.1 is the result when we generated 1000 nodes in the graph. α was maintained at 10^{-5}. As the number of edges were always between 5000 and 6000, we get $(1-\alpha)^{|E|} \geq 0.94$ for the probabilistic part of theorem 5.7.1. By *regret* we mean the upper bound of $\hat{w}(p(src)) - \hat{w}(p_{opt})$, as discussed before. The value *frac. regret* is fractional regret and defined as: regret/$\underline{W}(p(src))$.The running time in seconds is in the column *T(s)*. As can be seen, the percentage error goes up from 0% to 6.4%, but the running time improves by more than two orders of magnitude.

Table 5.2 gives similar results but when the number of nodes in the graph = 4000. For these experiments we used $\alpha = 10^{-6}$. As the number of edges in the graphs always remained less than $90,000$, we get $(1-\alpha)^{|E|} \geq 0.91$. Again, we obtained similar results to before. Increasing ϵ allows to reduce time by around two orders of magnitude, while maintaining a small fractional

Table 5.2: Varying ϵ with 4000 nodes

ϵ	regret	frac. regret	T(s)
0.1	0.02	0.0001	270.14
1.00	0.34	0.0008	24.94
10.00	12.71	0.0226	9.85
20.00	30.55	0.0462	5.22
50.00	77.17	0.1140	2.81

regret.

Varying α

Here we show that when we vary α, the advantage to running time is not as much, as when we varied ϵ. For each of $\epsilon = 1.0, 10, 20, 50$, in the experiments above, we experimented with $\alpha = 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$. However, we will tabulate only with $\epsilon = 1.0$, as that is where changing α made the greatest impact. For the other values of ϵ there was almost no change in the running time, as we varied α.

Denote the number of edges in the graph by $|E|$. As before we ran on two graphs; one with 1000 nodes, and one with 4000 nodes. In the former case we noted that $|E| \geq 5000$, and in the latter $|E| \geq 85000$. Hence as we vary α from 10^{-3} to 10^{-6}, we have $(1-\alpha)^{|5000|}$ varying from 0.007 to 0.995. Also $(1-\alpha)^{|85000|}$ varies from 1.16×10^{-37} to 0.92. Hence there is a vast change in the probabilistic part of the statement in theorem 5.7.1. Yet the advantage to running times is not as much.

As the regret always remained fairly close to 0 (because we are using a small ϵ), we are not tabulating that value below. The column T(s)_{1000} is

Table 5.3: Varying α

$(1-\alpha)$	$\mathrm{T(s)}_{1000}$	$\mathrm{T(s)}_{4000}$
0.999	3.41	17.26
0.9999	4.52	19.48
0.99999	5.67	22.12
0.999999	6.81	24.63

the running time when we generated 1000 nodes, and $\mathrm{T(s)}_{4000}$ is with 4000 nodes.

As can be seen the time did improve but not as markedly as when we varied ϵ. The reason for this has been discussed at the beginning of this section. Namely that PACBURM compares two path costs by narrowing the cost intervals of the two paths, until the intervals no longer intersect (if $\epsilon = 0$). In order to make things more efficient we can 'shrink' these two intervals. When the confidence intervals are already narrow, choosing a smaller value of α would not shrink the intervals further by too much. However ϵ-tolerance, can be considered as always shrinking each of the two intervals by an amount of $\epsilon/2$.

5.10 Summary

In this chapter we considered the problem of finding an optimal path through a graph, when the expected edge costs are unknown, but costs can be sampled. Hence sampling the cost we can compute a $(1-\alpha)$ confidence interval of the expected cost of an edge. We gave the PACBURM algorithm, which

is a generalized version of the DBURM algorithm (chapter 4), and finds a probably approximately optimal path in this setting.

When comparing the costs of two paths, we would be comparing two *intervals*. DBURM keeps sampling until the two intervals do not overlap in their interiors, at which times it is straightforward to pick the path with the 'smaller' cost. These comparisons can be made quicker if the intervals are shrunk. There are two ways to do this (a) Use a smaller value of confidence, $(1-\alpha)$. (b) Allow an ϵ tolerance - i.e. allow the intervals to overlap over a subinterval of length $\leq \epsilon$. In some sense, both these options are 'shrinking' the two intervals, so that the intervals can be separated out more easily.

We worked along the latter option, ϵ-tolerance. We proved that PACBURM returns a path that is within a cost of $L\epsilon$ of the optimal path, with a probability of $(1-\alpha)^{|E|}$. Here $|E|$ is the number of edges in the graph, and L is the number of edges in the optimal path. We also proved that PACBURM terminates with probability 1, if the underlying cost distributions have finite variance.

In the experiments, we showed that by varying ϵ, we can keep the 'fractional regret' to 1% in one experiment, and 5% in the other, while the running time decreases by 50-100 times. Here by 'regret' we mean the difference in expected costs, of the returned path, and that of the (ground truth) optimal path. The same kind of impact however was not achieved when we varied α. The reason being that when the intervals are already narrow, a (reasonable) change in ϵ can shrink the intervals by a greater amount than a (reasonable) change in α. Hence a larger value of ϵ is more likely to shrink the two narrow intervals by enough, so that they are separated, and we don't need to sample

any more. As it takes a lot of additional samples to narrow the confidence intervals even further when they are already narrow, varying ϵ gives a much greater advantage to efficiency, than varying α.

The ideas in this chapter are applicable to any planner that uses Monte Carlo simulations to compute the expected cost of an edge. These simulations can be used to compute confidence intervals of the expected costs. The ϵ-tolerance can then be used to reduce the number of simulations we need, while the path returned remains of an acceptable quality. On the other hand, we can use the analysis of this chapter to assess the quality of the returned path as well.

Chapter 6

Conclusion

Robot motion planning is a PSPACE hard problem. Nevertheless sampling based planners, such as the PRM [44, 45], have managed to solve many planning problems involving robots with a large number of degrees of freedom, in a reasonable amount of time. PRMs build a roadmap (graph) in the configuration space of the robot, and then find an optimal path through this roadmap. However classical PRM assumes perfect information about the environment. This assumption is not usually true, as environments are usually not known with perfect precision. Also classical PRM assumes a static environment. In chapters 3 and 4 we gave the BURM and DBURM algorithms to find an optimal path through the roadmap when the environment is dynamic and the information about the environment is imprecise.

In chapter 3 we gave the BURM algorithm which finds an optimal path, when the environment map is imperfect: we do not know the exact position of the obstacles. In such a case, the expected cost of traversing a path in the roadmap depends on the *probabilities* of collision when the robot traverses the

path. Calculating these probabilities is computationally prohibitive. BURM tackles this by not computing the probabilities of collision exactly: it only computes bounds on the probability. The bounds on the probability of collision at a given configuration are refined hierarchically, using geometry to divide the domain of integration into subdomains, and then computing the probabilities of colliision over the subdomains. The refinments are performed conservatively, only if required to identify the optimal path. We compared BURM to MCURM that uses pure Monte Carlo Integration to compute the probabilities of collision; it does not refine the bounds hierarchically, and does not keep bounds but computes the probabilities exactly. BURM outperformed MCURM by more than 2 orders of magnitude.

In chapter 4 we extended BURM to the Dynamic BURM algorithm (DBURM), which can handle dynamic changes in the environment. Similar to BURM, we do not compute the exact expected cost of an edge, but bound it within an interval, which can be made tighter with more computation. Hence in DBURM, edge weights are intervals. DBURM is related to dynamic graph algorithms, which search for shortest path in a graph efficiently after some edge weights have altered – these algorithms are already hard to understand and prove [86, 56, 25]. As in our case edge weights are intervals, it was even more challenging to efficiently recompute the optimal path after a dynamic change. A second option is to re-run BURM after every dynamic change. In the experiments DBURM outperformed 're-running BURM from scratch' option in most of the experiments. DBURM used 1.5 to around 3 times lesser number of collision checks when refining probability bounds. Hence DBURM required lesser information before identifying the

optimal path. In most of the experiments, DBURM was 1.5 to 2 times more efficient in terms of running time as well.

In a variety of situations, we do not know the expected costs of edges exactly, but we can sample the costs. The samples are usually taken in the guise of Monte Carlo simulations. Accurately speaking these samples can help us calculate the confidence intervals of the expected costs, but not the exact expected cost. Hence we can only return a path that has the least expected cost with a *high probability*. That is the returned path is the optimal path with a high probability. Going a step further, we can return a path that is *probably approximately* optimal. That is, with a high probability, pr, the expected cost of the returned path, is within a small distance, δ, from the optimal path.

By decreasing pr or increasing δ, we can make the graph search algorithm run faster. In chapter 5 we generalized the DBURM algorithm to give the Probably Approximately Correct BURM (PACBURM). We showed that we can return a path *much* faster by just a slight increase in δ. That is, with a small sacrifice in the quality of the path returned, the running time can be reduced by a lot. This is potentially applicable to any graph search algorithm, where the exact costs are unknown - by agreeing to bear a small error, the number of samples that we require before identifying the 'probably approximately' path, can be reduced drastically. Also, if we use for the classical way of using the average of N simulations to calculate the expected cost of edges, we can use the reasoning in chapter 5 to quantify the quality of the path that is being returned, by bounding pr and δ in such a setting.

6.1 Summary of Contribution

The main running theme in this thesis is: How to efficiently return a (probably approximately) optimal path, when the edge costs in the graph are not known precisely, but the precision can be iteratively improved by paying additional computation cost. The above problem is induced by planning the motion of the robot in an environment which is non-static, as well as the positions of the obstacles are not know with perfect accuracy. The graph that is made in the configuration space of the robot, is known as the *roadmap* in motion planning literature. The contribution can be summarized as three points.

1. When the environment map is not precise, we need to calculate the *probabilities* of collision of the robot, in order to compute the expected cost of a path. The first contribution is the idea of not computing the exact probabilities, but to compute bounds on it. These bounds can be refined by more computations. However BURM refines a bound, only if it helps in identifying the optimal path. Part of this refinement was done by using geometry of the robot configuration alone (which is another contribution), and part of it was done by Monte Carlo Integration. This led to efficiently finding the optimal path through the roadmap.

2. Giving a graph search algorithm that efficiently recomputes the optimal path after a dynamic change to the environment, when edge weights are intervals. This is the DBURM algortihm. Similar algorithms have been given before, but not when edge weights are intervals. The reason

that they are intervals, is that we are not computing the exact expected costs of edges (see point (1) above). Rather, we compute bounds on the expected costs. These bounds can be refined by more computation. Hence the expected costs are within a refinable interval. The aim is to pay little computation cost on narrowing the intervals, while still being able to identify the optimal path after a dynamic change.

3. In a more general scenario, given an edge e in the graph, we may not know the expected cost, $\hat{w}(e)$, of the edge, however we can sample the costs of e. From statistics we know, that using these sample costs we can compute an interval that bounds $\hat{w}(e)$ with probability $(1 - \alpha)$, where $\alpha > 0$. Such an interval is called as the $(1 - \alpha)$ confidence interval of $\hat{w}(e)$. By generalizing our previous algorithms, we gave the PACBURM algorithm that efficiently returns a path, that is *probably approximately* optimal in this setting. To our knowledge, a graph search algorithm that efficiently returns a close-to-optimal path by computing confidence intervals of expected costs, has not been given before. This has two potential uses. Firstly by sacrificing the quality of the path returned by a small amount, we can return a path much faster. This is borne out by our experiments. Secondly we can assess the quality of the path returned using the analysis in that chapter. Note that the usual assumption that Monte Carlo simulations give us the exact expected cost of edges, is not true: it can only give us the confidence interval of the expected cost. These ideas are therefore applicable to all planners that use simulations, (i) to either reduce the number of samples we

need while keeping the returned path of an acceptable quality or (ii) assess the quality of the returned path.

6.2 Future directions

The methodology of BURM can be applicable to other problems in motion planning which were not considered in this thesis. For example, we could consider the case of moving obstacles whose trajectories in the near future, are known with some amount of imprecision. Here it would be important for BURM to be able to compute the results fast enough, so that the robot can quickly execute its motion. Another possibility is to mix BURM and MDP together: suppose we have the transition model for the MDP, but lack complete information about the cost of executing a given action. This cost could be dependent on the probability of collision while executing the action. Here again we could use the idea of hierarchical refinement of probability estimates as we did in BURM.

We did not pay much attention to adapting the sampling of nodes, during the construction of the roadmap. There are lots of possibilities. For example, the sampling could be guided by the cost function: if the cost of collision is prohibitive, then the samples should be far from the obstacles etc. It would be interesting to compare the quality of the path returned by BURM under different sampling techniques. For dynamic environments, such as for the DBURM algorithm, sampling for nodes after a dynamic change can be even more useful.

The PACBURM algorithm finds a probably approximately optimal path

through the graph, when expected costs of edges are unknown, but we can take samples to tighten the $(1-\alpha)$ confidence intervals of the expected costs. The path returned by PACBURM, has a cost within δ of the optimal path, with probability pr. We would like δ to be low, and pr to be high. In chapter 5 we focussed on proving a tight upper-bound on δ once PACBURM has terminated. However the bounds on pr were quite naive: pr $\geq (1-\alpha)^{|E|}$, where α is the confidence level used, and $|E|$ is the total number of edges in the graph. It should be possible to come up with better bounds for pr as well, which would allow us to use a smaller value of $(1-\alpha)$, and yet have a high bound on pr. Also, we only proved that PACBURM terminates with probability 1 : at worst it can run for ever, but such scenarios happen with a probability measure of 0. By making a few more assumptions, or introducing more variables, it should be possible to give the efficiency of PACBURM in terms of $O(f(n))$ notation, where n is the size of the graph.

Bibliography

[1] P. Agarwal, L. Guibas, S. Har-Peled, A. Rabinovitch, and M. Sharir, *Penetration depth of two convex polytopes in 3d*, Nordic J. of Computing **7** (2000), 227–240.

[2] R. Alterovitz, T.T. Simeon, and K. Goldberg, *The stochastic motion roadmap: A sampling framework for planning with markov motion uncertainty*, Robotics: Science and Systems, 2007.

[3] J. Barraquand and J.-C. Latombe, *A Monte-Carlo algorithm for path planning with many degrees of freedom*, IEEE International Conference of Robotics & Automation, 1990.

[4] R. Bellman, *Dynamic programming*, Princeton University Press, 1957.

[5] J.V.D. Berg, P. Abbeel, and K. Goldberg, *LQG-MP: Optimized path planning for robots with motion uncertainty and imperfect state information*, Robotics: Science and Systems, 2010.

[6] R. Bohlin and L.E Kavraki, *Path planning using lazy prm*, IEEE Int. Conf. on Robotics & Automation, 2000, pp. 521–528.

[7] V. Boor, M. Overmars, and F. Van der Stappen, *The gaussian sampling strategy for probabilistic roadmap planners*, IEEE International conference on Robotics and Automation, 1999, pp. 1018–1023.

[8] B. Bouilly, T. Simeon, and R. Alami, *A numerical technique for planning motion strategties of a mobile robot in presecne of uncertainty*, IEEE Int. Conf. on Robotics and Automation, 2007.

[9] R.A Brooks, *Symbolic error analysis and robot planning*, Int. J. of Robotics Research (1982), 29–68.

[10] B. Burns and O. Brock, *Sampling-based motion planning with sensing uncertainty*, IEEE International Conference on Robotics and Automation, 2007, pp. 3313–3318.

[11] J. Canny, *On computability of fine motion plans*, IEEE International Conference on Robotics and Automation, 1989.

[12] A.R. Cassandra, L.P. Kaelbling, and M.L. Littman, *Acting optimally in partially observable stochastic domains*, AAAI, 1994, pp. 1023–1028.

[13] H-T. Cheng, *Algorithms for partially observable markov decision processes*, PhD thesis, University of British Columbia, Vancouver, 1988.

[14] H. Choset, K.M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L.E. Kavraki, and S. Thrun, *Principles of robot motion: Theory, algorithms and implementations*, The MIT Press, 2005.

[15] T.H. Cormen, C.E Leiserson, R.L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, 2009.

[16] M. de Berg, V. Kreveld, M.H. Overmars, and O. Schwarzkopf, *Computational geometry: Algorithms and applications*, Springer, 2000.

[17] C. Demetrescu, D. Frigioni, A. Marchetti-spaccamela, and U. Nanni, *Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study*, In Proc. Workshop on Algorithm Engineering, 2000, pp. 218–229.

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

[19] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri, *Computing the intersection-depth of polyhedra*, Algorithmica (1993), 518–533.

[20] D. Dubois and H. Parade, *Fuzzy sets and systems: Theory and applications*, Academic Press, 1980.

[21] A. Elfes, *Using occupancy grids for mobile robot perception and navigation*, Computer **22** (1989), 46–57.

[22] Erdmann84, *On motion planning with uncertainty*, Tech. Report 810, AI Lab., MIT, 1984.

[23] D. Ferguson, N. Kalra, and A. Stentz, *Replanning with rrts*, IEEE Int. Conf. on Robotics and Automation, 2006, pp. 1243–1248.

[24] D. Ferguson and A. Stentz, *The delayed d* algorithm for efficient path replanning*, IEEE Int. Conf. on Robotics and Automation, 2005, pp. 2045–2050.

[25] Dave Ferguson and Anthony Stentz, *Delayed d*: The proofs*, Robotics Institute, Carnegie Mellon University, Tech. Report, 2004.

[26] T. Fraichard and R. Mermond, *Path planning with uncertainty for car-like robots*, IEEE Int. Conf. on Robotics and Automation, 1998.

[27] D. Frigioni, *Fully dynamic algorithms for maintaining shortest path trees*, J. Of Algorithms **34** (2000), 251–281.

[28] J.P. Gonzalez and A. Stentz, *Using linear landmarks for path planning with uncertainty in outdoor environments*, Int. Conf. on Intelligent Robots and Systems, 2009.

[29] L.J. Guibas, D. Hsu, H. Kurniawati, and E. Rehman, *Bounded uncertainty roadmaps for path planning*, Workshop on Algorithmic Foundations of Robotics, 2008.

[30] R.V. Hogg, A. Craig, and J.W. McKean, *Introduction to mathematical statistics*, 6 ed., Prentice Hall, 2004.

[31] R.A Howard, *Dynamic programming and markov processes*, MIT Press, Cambridge, Massachusetts, 1960.

[32] D. Hsu, T. Jiang, J. Reif, and Z. Sun, *The bridge test for sampling narrow passages with probabilistic roadmap planners*, IEEE International Conference on robotics and Automation, 2003, pp. 4420–4426.

[33] D. Hsu, R. Kinder, J.-C. Latombe, and Rock S., *Randomized kinodynamic motion planning with moving obstacles*, Int. J. of Robotics Research (2002), 233–256.

[34] D. Hsu, J.-C. Latombe, and H. Kurniawati, *On the probabilistic foundations of probabilistic roadmap planning*, Int J. Robotics Research **25** (2006), no. 7, 627–643.

[35] D. Hsu, J.-C. Latombe, and R. Motwani, *Path planning in expansive configuration spaces*, IEEE International Conference on Robotics and Automation, 1997, pp. 2719–2726.

[36] Y. Huang, *Motion planning with localization and mapping uncertainties for a mobile manipulator in exploration and inspection tasks*, Ph.D thesis, Engineering Science, Simon Fraser University, 2009.

[37] Y. Huang and K. Gupta, *Collision-probability constrained prm for a manipulator with base pose uncertainty*, ICRA, 2009.

[38] S. S. Isukapalli, *Uncertainty analysis of transport-transformation models*, Ph.D. Thesis, Rutgers University, 1999.

[39] T.A Johansen, *Computation of lyapunov functions for smooth nonlinear systems using convex optimization*, Automatica (2000), 1617–1626.

[40] L.P. Kaelbling, M.L. Littman, and A.R. Cassandra, *Planning and acting in partially observable stochastic domains*, Artificial Intelligence **101** (1998), no. 1-2, 99–134.

[41] M.H. Kalos and P.A. Whitlock, *Monte Carlo methods*, vol. 1, John Wiley and Sons, 1986.

[42] K. Kant and S.W. Zucker, *Toward efficient trajectory planning: Path velocity decomposition*, Int. J. of Robitics Research (1986), 72–89.

[43] O.E. Karasan, C.P. Mustafa, and H. Yaman, *The robust shortest path problem with interval data*, Bilkent University, 2001.

[44] L.E. Kavraki, M.N Kolountzakis, and J.-C. Latombe, *Analysis of probabilistic roadmaps for path planning*, IEEE Transactions on Robotics and Automation, 1996, pp. 3020–3025.

[45] L.E. Kavraki, P. Švetska, J.-C. Latombe, and M.H. Overmars, *Probabilistic roadmaps for path planning in high dimensional configuration spaces*, IEEE Transactions on Robotics and Automation, 1996, pp. 566–580.

[46] G. Kewlani, G. Ishigami, and K. Iagnemma, *Stochastic mobility-based path planning in uncertain environments*, IROS, 2009, pp. 1183–1189.

[47] C.M. Klein, *Fuzzy shortest paths*, Fuzzy Sets and Systems (1991), 27–41.

[48] S. Koenig and M. Likhachev, *Improved fast replanning for robot navigation in unknown terrain*, IEEE Int. Conf. on Robotics and Automation, 2002, pp. 968–975.

[49] P. Kouvelis and G. Yu, *Robust discrete optimization and its applications*, Kluwer Academic Publishers, 1997.

[50] J. Kuffner and S.M. LaValle, *An efficient approach to single query path planning*, Proceedings of the IEEE Conference on Robotics Research, 2000, pp. 995–1001.

[51] H. Kurniawati, D. Hsu, and W.S. Lee, *Sarsop: Efficient point-based pomdp planning by approximating optimally reachable belief spaces*, Robotics: Science and Systems, 2008.

[52] J.-C. Latombe, *Robot motion planning*, Kluwer Academic Publishers, 1991.

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

[54] A. Lazanas and J.-C. Latombe, *Motion planning with uncertainty: A landmark approach*, Artif. Intell. (1995), 287–317.

[55] T.S. Levitt, D.T. Lawton, D.M. Chelberg, and P.C. Nelson, *Qualitative landmark-based path planning and following*, AAAI, 1987, pp. 689–694.

[56] M. Likhachev and S. Koenig, *Lifelong planning a* and d* lite: The proofs,*, College of Computing, Georgia Institute of Technology, Tech. Report, 2004.

[57] M. Lin and D. Manocha, *Collision and proximity queries*, Handbook of Discrete and Computational Geometry (J.E. Goodman and J. O'Rourke, eds.), CRC Press, 2004.

[58] T. Lozano-Perez, *The design of a mechanical assembly system*, A.I Memo 397, MIT Artificial Intelligence Lab, 1976.

[59] T. Lozano-Perez, M. Mason, and R.H. Taylor, *Automatic synthesis of fine-motion strategies for robots*, International Journal of Robotics and Research (1984), 3–24.

[60] N.A. Melchior and R.G. Simmons, *Particle rrt for path planning with uncertainty*, ICRA, 2007, pp. 1617–1624.

[61] P.E. Missiuro and N. Roy, *Adapting probabilistic roadmaps to handle uncertain maps*, IEEE International Conference on Robotics and Automation, 2006.

[62] T. Mitchell, *Machine learning*, McGraw Hill, 1997.

[63] G.E. Monahan, *A survey of partially observable markov decision processes: Theory, models, and algorithms*, Management Science (1982), 1–16.

[64] R. Montemanni and L.M. Gamberdella, *An exact algorithm for the robust shortest path problem with interval data*, Computers and Operations Research (2004), 1667–1680.

[65] A. Nakhaei and F Lamiraux, *A framework for planning motions in stochastic maps*, Int. Conference on Control, Automation, Robotics and Vision, 2008.

[66] F. Noe, M. Oswald, G. Reinelt, S. Fischer, and J.C. Smith, *Computing best transition pathways in high-dimensional dynamical systems: Application to the $\alpha_l \rightleftharpoons \beta \rightleftharpoons \alpha_r$ transitions in octaalanine*, Multiscale Modeling and Simulation (2006), 393–419.

[67] S. Okada and M. Gen, *Fuzzy shortest path problem*, Comp. and Industrial Engr, 1994.

[68] C. Papadimitriou and J.N. Tsisiklis, *The complexity of Markov decision processes*, Mathematics of Operations Research **12** (1987), no. 3, 441–450.

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

[70] R. Pepy and A. Lambert, *Safe path planning in uncertain configuration space using rrt*, Int. Conf. on Intelligent Robots and Systems, 2006.

[71] T.L Perez, M.T. Mason, and R.H. Taylor, *Automatic synthesis of fine-motion strategies for robots*, Inr. J. Rob. Res. (1984), 3–24.

[72] J. Pineau, G. Gordon, and S. Thrun, *Point-based value iteration: Any anytime algorithm for pomdps*, Joint Conference on Artificial Intelligence, 2003.

[73] K.-M. Poon, *A fast heuristic algorithm for decision-theoretic plannig*, Master's thesis, The Hong Kong University of Science nd Technology, 2001.

[74] S. Prentice and N. Roy, *The belief roadmap: efficient planning in belief space by factoring the covariance*, Int. Jour. of Robotics Reserach (2009), 1448–1465.

[75] G. Ramalingam and T.W. Reps, *An incremental algorithm for a generalization of the shortest-path problerm*, Tech Report, Computer Sciences Department, University of Wisonsin, Madison, 1992.

[76] G. Ramalingam and T.W Reps, *An incremental algorithm for a generalization of the shortest-path problem*, J. of Algorithms (1996), 267–305.

[77] J.H. Reif, *Complexity of the mover's problem*, IEEE Symposium on Foundations of Computer Science, 1979, pp. 421–427.

[78] J.H. Reif and M. Sharir, *Motion Planning in the presence of Moving Obstacles*, IEEE Symposium on Foundations of Computer Science, 1985, pp. 144–154.

[79] N. Roy, W. Burgard, D. Fox, and S. Thrun, *Coastal navigation - mobile robot navigation with uncertainty in dynamic environments*, ICRA, 1999.

[80] N. Roy and G.S Thrun, *Finding approximate pomdp solutions through belief compression*, Artif. Intell. (2005), 1–40.

[81] S. Russell and P. Norvig, *Artificial intelligence: A modern approach*, Prentice Hall, 2003.

[82] G. Shani, R. Brafman, and S. Shimony, *Forward search value iteration for pomdps*, Int. Jnt. Conf. Artificial Intelligence, 2007.

[83] R.D. Smallwood and E.J. Sondik, *The optimal control of partially observable markov processes over a finite horizon*, Operations Research **21** (1973), 1071–1088.

[84] T. Smith and R. Simmons, *Heuristic search value iteration for pomdps*, Uncertainty in Artificial Intelligence, 2004.

[85] E.J. Sondik, *The optimal control of partially observable markov processes*, PhD thesis, Stanford, 1971.

[86] A. Stentz, *Optimal and efficient path planning for partially-known environments*, IEEE Int. Conf. on Robotics and Automation, 1994, pp. 3310–3317.

[87] Anthony Stentz, *The focussed d* algorithm for real-time replanning*, In Proceedings of the International Joint Conference on Artificial Intelligence, 1995, pp. 1652–1659.

[88] R. Tedrake, *Lqr-trees: Feedback motion planning on sparse randomized trees*, Robotics: Science and Systems, 2009.

[89] S. Thrun, W Burgand, and D. Fox, *Probabilistic robotics*, MIT Press, 2005.

[90] K.I. Tsianos and Kavraki L.E, *Replanning: A powerful planning strategy for hard kinodynamic problems*, IEEE Int. Conf. on Robotics and Automation, 2008.

[91] S. Urmson and R. Simmons, *Approaches for heuristically biasing rrt growth*, IROS, 2003.

[92] L. Valiant, *A theory of the learnable*, Communications of the ACM, vol. 27, 1984, pp. 1134–1142.

[93] J-S. Yao and F-T. Lin, *Fuzzy shortest-path network problems with uncertain edge weights*, Information Science and Engr. (2003), 329–351.

[94] Y. Yu and K. Gupta, *Sensor-based roadmaps for motion planning for articulated robots in unknown environments. some experiments with an*

eye-in-hand system, IEEE/RSJ Int. Conf. on Intelligent Robots and Systems, 1999.

[95] L.A. Zadeh, *Fuzzy sets*, Information and Control (1965), 338–353.

[96] L. Zhang, Y.J. Kim, G. Varadhan, and D. Manocha, *Fast c-obstacle query computation for motion planning*, IEEE International Conference on Robotics and Automation, 2006.

[97] N.L. Zhang and W. Liu, *Planning in stochastic domains: problem characteristics and approximation*, Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology, 1996.

[98] N.L. Zhang and W. Zhang, *Speeding up the convergence of value iteration in partially observable markov decision processes*, Artificial Intelligence Research (2001), 14:29–51.

[99] P. Zielinski, *The computational complexity of the relative robust shortest path problem with interval data*, European Journal of Operational Research (2004), 570–576.