Making Roadmaps Using Voronoi Diagrams

Michael L. Wager

16 Nov 2000

Abstract

The path planning problem has been proven both PSPACE complete and NP complete. All the traditional approaches have only had marginal results when trying to tackle this problem. In recent years a Probabilistic RoadMap planner (PRM) has been developed that has had impressive results. This planner has had success in high dimensions but because of its randomness it is not appropriate in low dimensions. The Voronoi diagram approach is exact and is thus appropriate in two dimensions. I will discuss my implementation of the Voronoi diagram approach and show that this method is not only faster than the standard visibility graph, but is more desirable as well.

Acknowledgements

I want to firstly give all the glory to God for providing me with a miracle to start and complete this within three weeks. This year was an especially difficult one for me and I want to thank some people who stuck by me in times of need. My supervisor Dr. Peter Kovesi helped me persevere and made sure I finished something that I started. I want to thank the incredibly young man, Voon-Li Chung for saying that ``You will finish Honours''. I also want to say a special thank you to Chew Yan Nurdin for staying up with me on all those late nights and cheering me on all the way. The rest of the gang, E-Ling Wong, Warren Teh and Chew Lee Nurdin all supported me and boy are we going to celebrate when we go around Australia. I want to thank all of these people for making the impossible, possible. All things are Possible!

List of Figures

2.1 Object Representation.
2.2 The Minkowski Set Difference.
2.3 Slihouette Curve.
2.4 A subgoal network.
2.5 Dijksta's algorithm.
4.1 Constructing a Voronoi Diagram.
4.2 The input given to the Voronoi diagram calculation.
4.3 The Voronoi diagram of some simple obstacles.
4.4 Two lines intersect in the above six ways.
4.5 The pruned Voronoi diagram.
4.6 Smoothing an intersection of three edges.
4.7 The final smoothed Voronoi diagram.
4.8 The maze used to test the Voronoi diagram and the visibility graph.

Contents

Abstract
Acknowledgements
1 Introduction
2 The Traditional Approach
2.1 Introduction
2.1.1 Definitions
2.1.2 Standard Procedure
2.2 Modeling Objects
2.2.1 Grid
2.2.2 Cell Tree
2.2.3 Polyhedra
2.2.4 CSG and B-Rep
2.3 Cspace Construction
2.3.1 Point Evaluation
2.3.2 Minkowski Set Difference
2.3.3 Boundary Equation
2.3.4 Needle
2.3.5 Sweep Volume
2.3.6 Template
2.3.7 Jacobian Based Approach
2.3.8 Fast Fourier Transform
2.3.9 Analysis
2.4 Motion Planning
2.4.1 Skeleton

2.4.2 Cell Decomposition
2.4.3 Potential Field
2.4.4 Mathematical Programming
2.5 Searching
2.5.1 Depth First and Breadth First
2.5.2 Best First
2.5.3 A^*
2.5.4 Bidirectional
2.5.5 Dijkstra
2.5.6 Random Search
2.6 Local Optimization
2.7 Conclusions
3 Roadmap Approach
3.1 Introduction
3.1.1 Definitions
3.2 Potential Field Planner
3.3 PRM Construction
3.3.1 Pre-processing Phase
3.3.2 Querying Phase
3.4 Performance
3.4.1 Summary
3.5 Conclusions
4 Voronoi Implementation
4.1 Introduction
4.2 Voronoi Diagrams
4.3 Implementation
4.3.1 Set up
4.3.2 Voronoi Construction
4.3.3 Pruning
4.3.4 Connecting to diagram
4.3.5 Smoothing and Dijkstra Search
4.4 Experiment
4.5 Suggestions
4.6 Conclusions
5 Conclusion
A Original Honours Proposal

Chapter 1

Introduction

Hwang and Ahuja [HA92] describe a highly automated factory where mobile robots pick up parts and deliver them to assembly robots. The robots must navigate the factory without colliding with other objects and robots. The paths that they take must be efficient and seamless; that is they must not take any delays in choosing a good path. This need for collision avoidance and efficient motion leads to the creation of robot motion path planning algorithms.

A path planner is an algorithm that seeks to find a path from a starting point to the finishing point. This planner is given the source and goal configuration and has to find a connecting path that avoids any intermediate obstacles.

There are an increasing number of practical problems that involve path planners. Koga et al. [KKKL94] show how computer graphics animators benefit from a path planner that automatically synthesizes video clips with graphically simulated human and robot characters. Kavraki [Kav97] illustrates how motion planning is used in molecular biology to compute motions of molecules docking against each other. Chang and Li [CL95] show that path planning has been used to check if parts can be removed from an airplane engine for inspection and maintenance. Efficient and reliable path planners are also required in medical surgery and space exploration.

Reif [Rei79] proved that the path planning problem is PSPACE-complete. This means that a path planner requires storage space (memory) exponential in the degrees of freedom. This exponential space requirement stems from the vast number of different configurations these planners have to cope with. For example, if a robot has 12 degrees of freedom and each of these can occupy only 100 discrete positions it would amount to 100^{12} configurations. To store all of these configurations would require a lot of space!

In addition to the storage problem, Reif also proved that the path planning problem is NP-complete. This means that a path planner's running time is exponential in the degrees of freedom. Together the exponential time and storage requirements create a challenging research problem.

Many different heuristics have been proposed to solve the motion planning problem. Firstly in Chapter two, I will look at the notion of Cspace, its construction and its place in the standard procedure. In Chapter three I will explain in depth the advantages of the Probabilistic RoadMap planner(PRM). In Chapter four I will explain my implementation of the road map approach using Voronoi Diagrams. Additionally, I will show that this approach is better than the standard Visibility Graph approach. In conclusion, I will show that even though the Voronoi diagrams planner and the PRM are two of the best planners, different applications require different solutions.

Chapter 2 The Traditional Approach

2.1 Introduction

Hwang and Ahuja [HA92] have looked in depth at all of the best attempts at solving the motion path planning problem. They have come to the conclusion that different approaches are required in different applications. Having said that there are some basic issues and steps that make up any motion planner.

2.1.1 Definitions

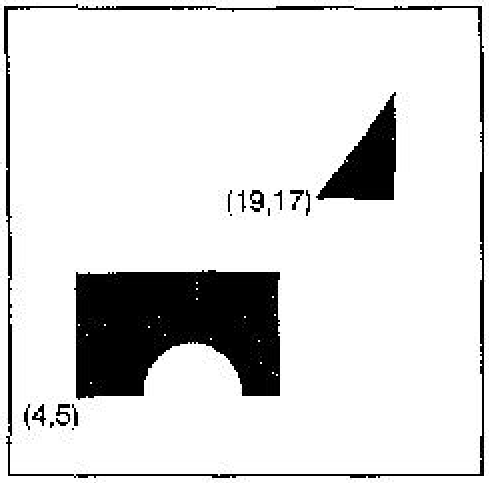
Before addressing these steps it is important that the terms used here on in are well defined. The *world space* refers to the physical space in which robots and obstacles exist. The *configuration* of an object is a set of independent parameters that characterizes the position of every point in the object. Six parameters are needed to specify the configuration of a rigid object in three dimensions (three for position and three for orientation). The *degrees of freedom (dof)* are the number of parameters required to specify the configuration of an object. The *configuration space (Cspace)* is the set of all configurations. The *free space* (C_{free}) is the part of Cspace in which the robot does not collide with any obstacles. The *path* of an object is a curve in the configuration space. *Feasible* simply means collision free. A *solution* is a feasible path from the start to the end.

2.1.2 Standard Procedure

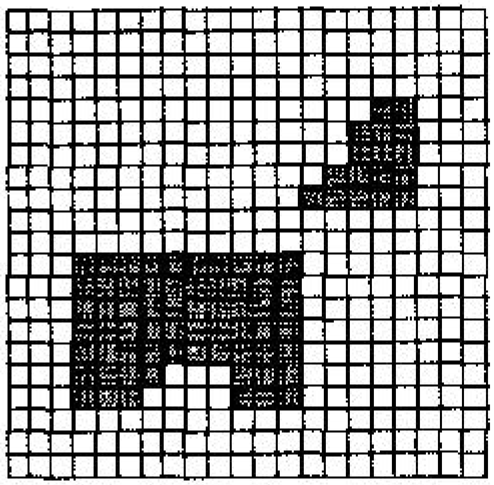
The standard procedure in motion planning begins with constructing a model of the physical robot and objects. Once this internal world has been built the configuration space is built using any one of many methods. A suitable motion planning approach is then applied on the Cspace. After this, a search method must be selected to find a solution path. Finally, the solution path may be locally optimized to yield a shorter and smoother path.

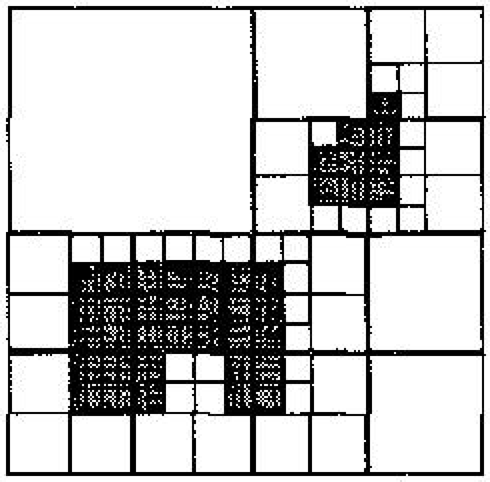
2.2 Modeling Objects

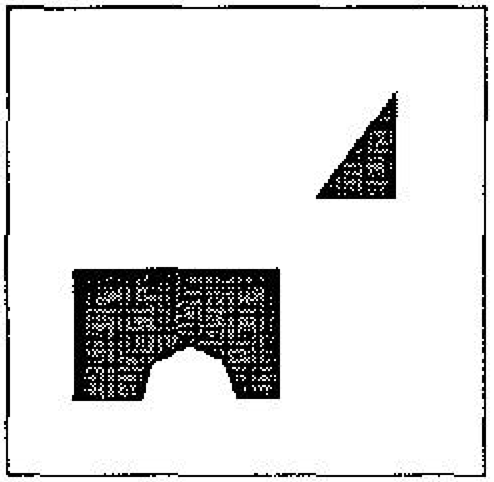
A robot has to have a model of objects in its environment before any motion planning can commence. This information can either be attained by visual sensors or by a human entering data. Hwang and Ahuja [HA92] list five common ways of representing the objects.

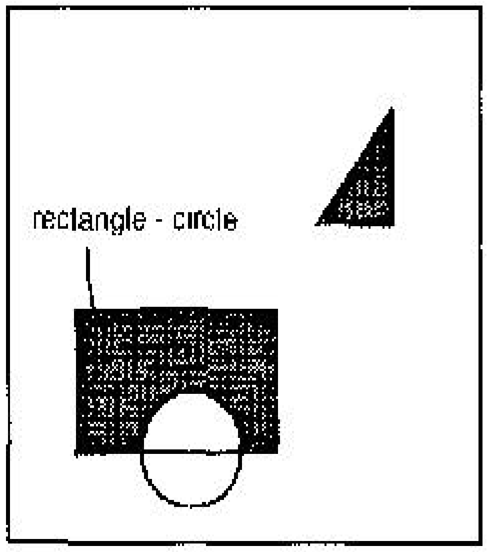


(a) original objects

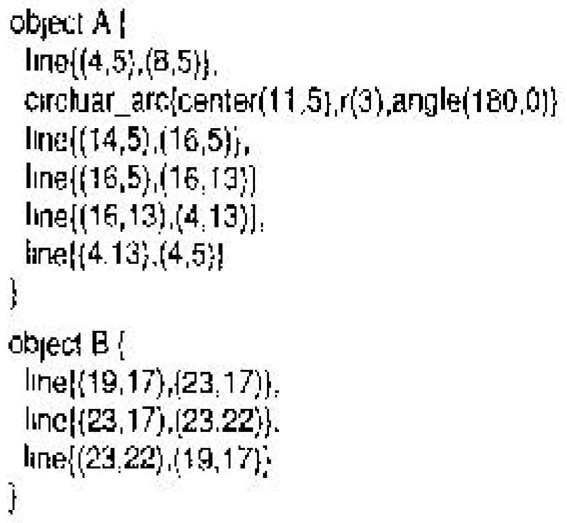

(b) grid


(c) cell tree (quadtree)


(d) polygonal approximation



(e) constructive solid geometry (CSG)



```
object A {
  line((4,5),(8,5)),
  circular_arc(center(11,5),r(3),angle(180,0))
  line((14,5),(16,5)),
  line((16,5),(16,13))
  line((16,13),(4,13)),
  line((4,13),(4,5))
}
object B {
  line((19,17),(23,17)),
  line((23,17),(23,22)),
  line((23,22),(19,17))
}
```

(f) boundary representation (B-rep)

Figure 2.1: Object Representation. Reprinted from Hwang and Ahuja [HA92].

2.2.1 Grid

A grid is an array of identical cells where each cell indicates the presence of an obstacle. A cell would typically be marked as 1 if an obstacle occupies it and be marked as 0 otherwise. See Figure 2.1 b for more details. Although this may seem inefficient its simplicity has many computational advantages. Calculating the volume for instance is best calculated using this representation. This is especially true when the object is irregularly shaped.

2.2.2 Cell Tree

The cell tree is an extension of the grid idea where the object space is divided into a small number of big cells. See Figure 2.1c. The cells that are completely inside or outside are marked as such and the other cells are further divided. This division continues until it reaches a resolution limit. The cell tree uses less space than the grid but the adjacency computation of cells takes longer.

2.2.3 Polyhedra

The commonly used polyhedral representation approximates all the obstacles as a series of straight lines or flat faces and simply stores their corner coordinates. This approximation is usually accurate enough due to the nature of objects having many straight surfaces. Spheres, circles and other irregular shapes are represented by a finite number of flat faces in such a way that the original shape is preserved as much as possible. See Figure 2.1d. Polyhedra are often used since many efficient algorithms exist for computing the intersection of, and distance between, two polyhedra. This is important since computation of intersection and distance is performed quite often in motion planning.

2.2.4 CSG and B-Rep

Constructive Solid Geometry (CSG) represents objects as the union, intersection and set differences of primitive shapes such as squares, triangles and circles. The Boundary Representation (B-Rep) explicitly lists boundary features of objects such as lines and circular arcs. The good thing about both of these methods is the ease at which circles are represented and small space that it needs. See Figure 2.1e-f.

2.3 Cspace Construction

Once the objects have been represented the Cspace can be constructed. Hwang and Ahuja [HA92] list seven basic ways of computing this and Kavraki [Kav95] gives an interesting alternative using the Fast Fourier Transform.

2.3.1 Point Evaluation

This simple method involves placing the robot in every configuration and determines whether it intersects any obstacles. This is the most inefficient method since all possible configurations are checked for intersection and this grows exponentially.

2.3.2 Minkowski Set Difference

See Figure 2.2. The Minkowski Set Difference (MSD) of two sets A and B is the set of points:

$$\mathrm{MSD}(A,B) = \{ a - b \text{ \textbackslash mid } a \subseteq A , b \subseteq B\} .$$

This results in a set containing A such that B can not intersect A providing the reference point of B does not go inside the boundary of this new set. In the construction of Cspace this method effectively grows the obstacles by the size of the robot. For a rigid robot that cannot change its orientation the Cspace is the union of the MSD between areas occupied by obstacles and the robot. However if the robot can rotate or change its shape the Cspace needs to have another dimension and be built up slice by slice. Every slice is the MSD of the obstacles and the robot with a fixed orientation. Since this is rather complicated this method is mostly used for rigid non-rotating robots such as a two dimensional polygon.

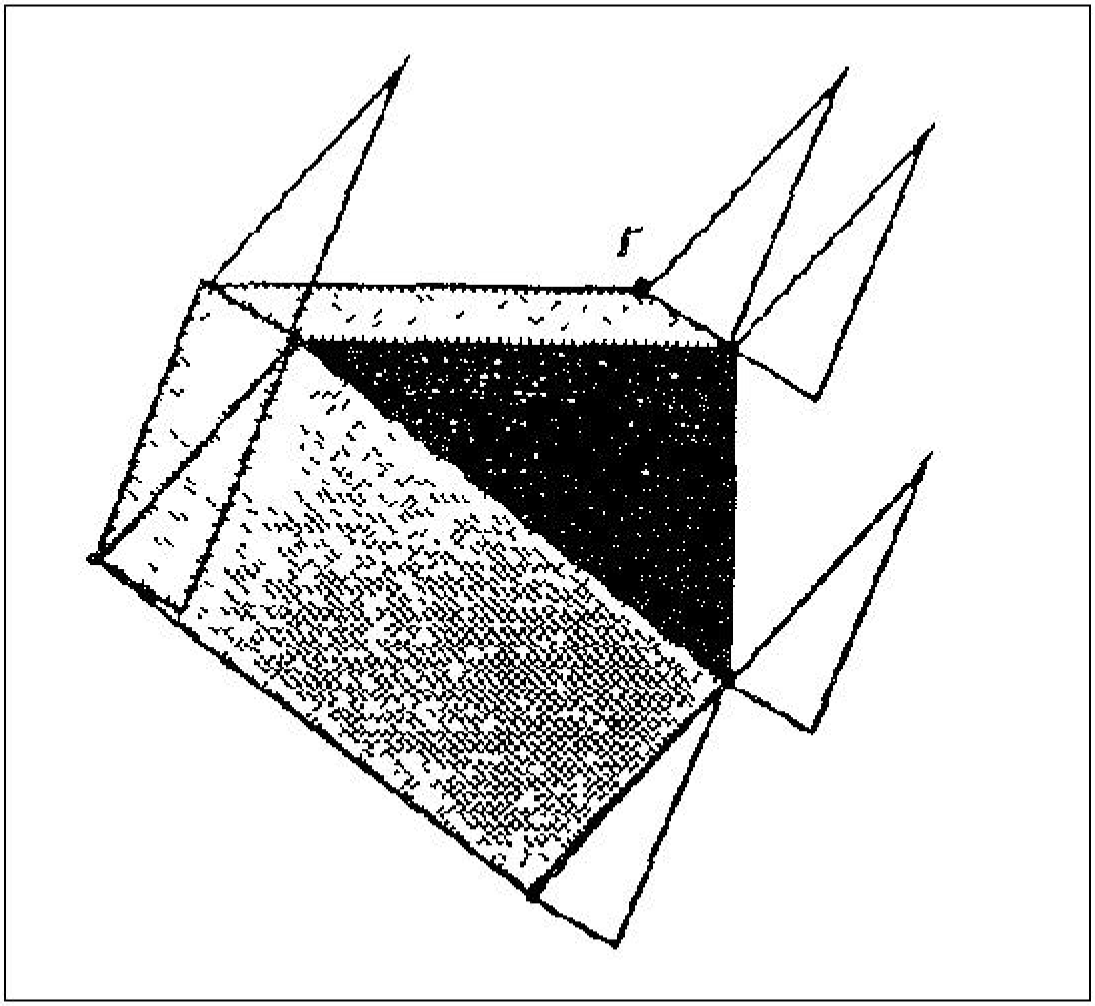


Figure 2.2: The black area is the original object and the lightly shaded region is the Minkowski Set Difference. The reference point of the robot, r, in this orientation cannot be placed in this region. Reprinted from Hwang and Ahuja [HA92].

2.3.3 Boundary Equation

This method involves deriving the constraints on the configuration variables that bring the robot in contact with obstacles. These constraints are expressed algebraically as equations. The equations are derived from the vertex to face and edge to edge contacts between the robot and obstacles. Finding these equations is very difficult especially for Cspace having dof (defined in section 2.1.1) greater than three.

2.3.4 Needle

The needle method involves fixing every configuration parameter except one. By varying this parameter the values that bring the robot into contact with an obstacle are computed. These values give intervals (or needles) of the Cspace. This method is often used to generate a two dimensional slice of the Cspace by fixing all but two parameters. One of these parameters is slowly varied as needles are constructed with the other parameter. This needle method is not used for higher dimensions because of the large number of needles required.

2.3.5 Sweep Volume

This method computes the volume in the world space swept by a robot as the configuration is varied over a set in Cspace. If the volume does not intersect any obstacles the set is in C_{free}. Sweep volume is effective in determining C_{free} but is hard to compute in higher dimensions.

2.3.6 Template

Branicky and Newman [BN90] developed a method that computes Cspace based on the features of the physical obstacles. The shapes of the obstacles have matching shapes called a template in Cspace. A line obstacle for instance has a matching ellipse inside Cspace. The size and position of the templates depend on the orientation of the obstacles. Complex obstacles are broken down into simpler shapes that have matching templates in Cspace. Finally the Cspace is made up of these templates ``stamped'' together. This method works well for dof less than five but for higher dof it suffers from huge memory requirements due to the way templates are stored.

2.3.7 Jacobian Based Approach

Paden et al. [PMF89] developed a very elegant method to compute a ``block'' of Cspace. The Jacobian J of a robot is a matrix that relates the displacement, dx, of a point, p on the robot to the change in the robot configuration, dq. The bound B(q) is the maximum of the absolute value of the Jacobian J at a configuration q over all points on the robot (max|J(q)|). If the minimum distance between a robot in a configuration q and all the obstacles is D then it follows that the sphere centred at q with radius D/B(q) is in C_{free} . It also follows that if the minimum distance between a configuration q that is inside an obstacle and the edge of the obstacle is D– then the sphere centred at q with radius D–/B(q) is an obstacle in Cspace. This approach can be adjusted by using different measures for distance. This will result in different shapes being included or excluded from C_{free}. Cuboids for example can be generated using the distance max\mid p_i - q_i\mid, where p_i and q_i are a point on the robot and a configuration of the robot.

2.3.8 Fast Fourier Transform

Kavraki [Kav95] developed this interesting method that relies on the observation that for a non rotating rigid body the Cspace is a convolution of the workspace and the robot. The running time depends only on the discretization used. It is independent of the complexity of the robot and obstacles. This method works best where no orientation changes are required. If they are needed the Cspace can still be built up slice by slice using a Fast Fourier Transform for every orientation but this takes time. In addition, this method benefits from the amount of research already done in this field and from the development of specific hardware to handle the Fast Fourier Transform. Another advantage is the fact that intersections can be computed in constant time since the Cspace is stored in a grid. The analysis done by Kavraki suggests that this method is best used when the shape of the obstacles and robot are really complex. Otherwise if the shapes are simple other methods such as needle, point and Minkowski Set Difference are more appropriate.

2.3.9 Analysis

All of the above methods work well in low dimensions. However as the dof increase, Cspace grows exponentially and thus becomes extremely difficult to compute. I hypothesize that they have a natural limit at around six dof. For higher dimensions other methods need to be used since the Cspace can no longer be explicitly stored. In the following Chapter I will discuss an approach that has been successfully used in dof greater than 75.

2.4 Motion Planning

Once Cspace has been constructed the real problem solving begins. Many different approaches have been proposed for Motion Planning (MP). Hwang and Ahuja [HA92] state that these methods are a variation of four key approaches, the skeleton, cell decomposition, potential field, and mathematical programming. I will outline all of these methods in this section. In the next two chapters I will discuss in more detail two of the skeleton approaches, the probabilistic roadmap and the Voronoi diagram.

2.4.1 Skeleton

This approach involves retracting or reducing C_{free} onto a network of one-dimen-sional lines. MP is then reduced to a graph-searching problem. Three steps are involved. A path is first found from the starting position to a point on the skeleton. A path is then found from the goal configuration to a point on the skeleton. Finally a path is constructed between the two points on the skeleton. The correctness of the solution strongly depends on the skeleton representing the entire Cspace. If the skeleton does not represent the entire Cspace a solution path may be missed. The standard skeletons are the visibility graph, the Voronoi diagram, the silhouette and the subgoal network. All of these will be discussed with the exception of the Voronoi diagram, which will be covered in Chapter four.

The visibility graph is the collection of lines in C_{free} that connects a feature of an object to that of another. A solution is found when the start and goal configurations are included as features in the graph. Asano et al. [AGHI85] developed a way to construct this graph in $O(n^2)$ time in two dimensions, where n is the number of features. This visibility graph will be compared with the Voronoi implementation in Chapter four.

Canny [Can87] presented a general method of constructing a skeleton in arbitrary dimensions. His method involves recursively projecting objects of higher dimensions into lower dimensions and tracing out the boundary curves of the projection, which is called the silhouette. This process terminates when it reaches the one-dimensional line. To ensure that the projections retain the features of the object, linking curves are placed where new silhouette curves appear or disappear. This method is mostly used for theoretical analysis since a path found using this method makes the robot slide along obstacle boundaries. See Figure 2.3.

w

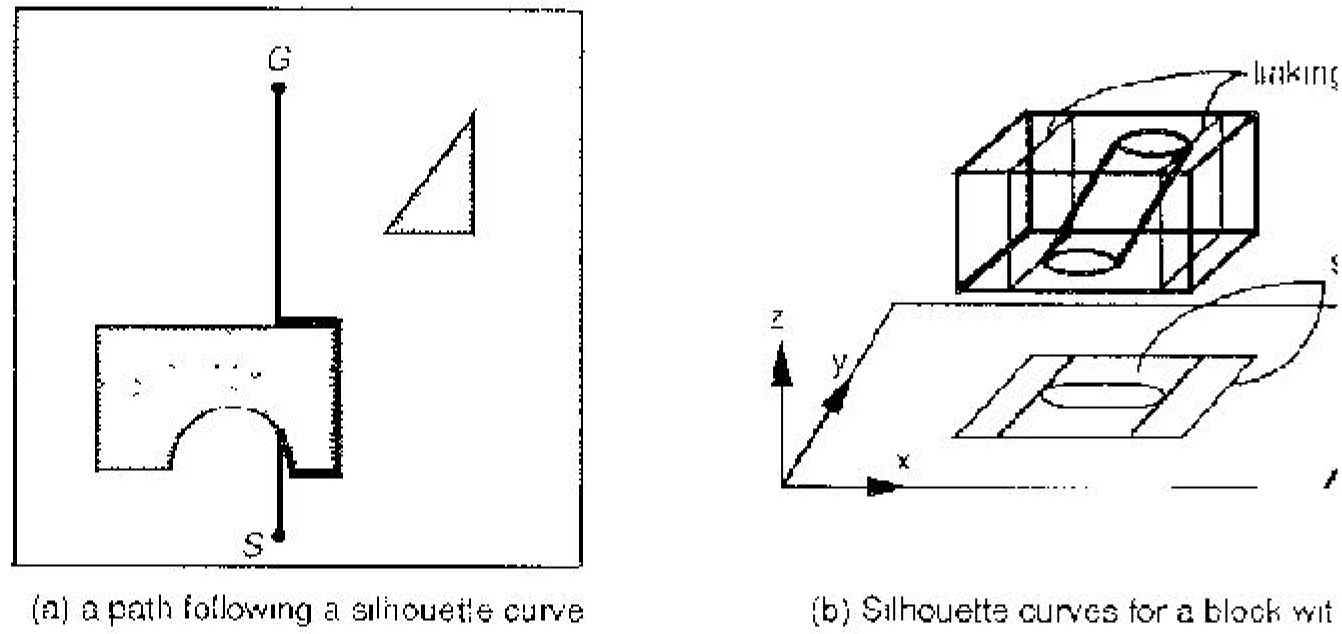

G
S
linking
z
y
x
(a) a path following a silhouette curve
(b) Silhouette curves for a block wit

Figure 2.3: Silhouette curve. Reprinted from Hwang and Ahuja [HA92].

The subgoal network does not need the Cspace to be constructed since it only checks a number of finite configurations. The method involves keeping a list of reachable configurations until it reaches the goal. A simple local MP algorithm called a local operator is used to determine reachability. Moving the robot in a straight line between the configurations is an example of a local operator. See Figure 2.4 . As its name suggests the subgoal network generates intermediate configurations called subgoals and uses the local operator to successively move the robot through the subgoals. If the end configuration is not reached these subgoals are stored and more subgoals are generated and passed to the local operator. This process continues until it reaches the goal. To summarize, this method moves towards the goal from the generated subgoals using a local operator.

The subgoal network is memory efficient since the Cspace does not need to be explicitly stored. The choice of the local operator is the most important consideration in this method since it determines its completeness. The simple go-straight local planner often fails to find solutions between two points that are far apart. Chen and Hwang [CH92] completed work on a local operator that is a global motion planner. It decomposes the MP problem into a number of simpler MP problems. Hwang and Ahuja [HA92] recommend using a potential field method for a local operator since it is found to be the most efficient. This is an example of where two MP techniques are used together to create a better solution.

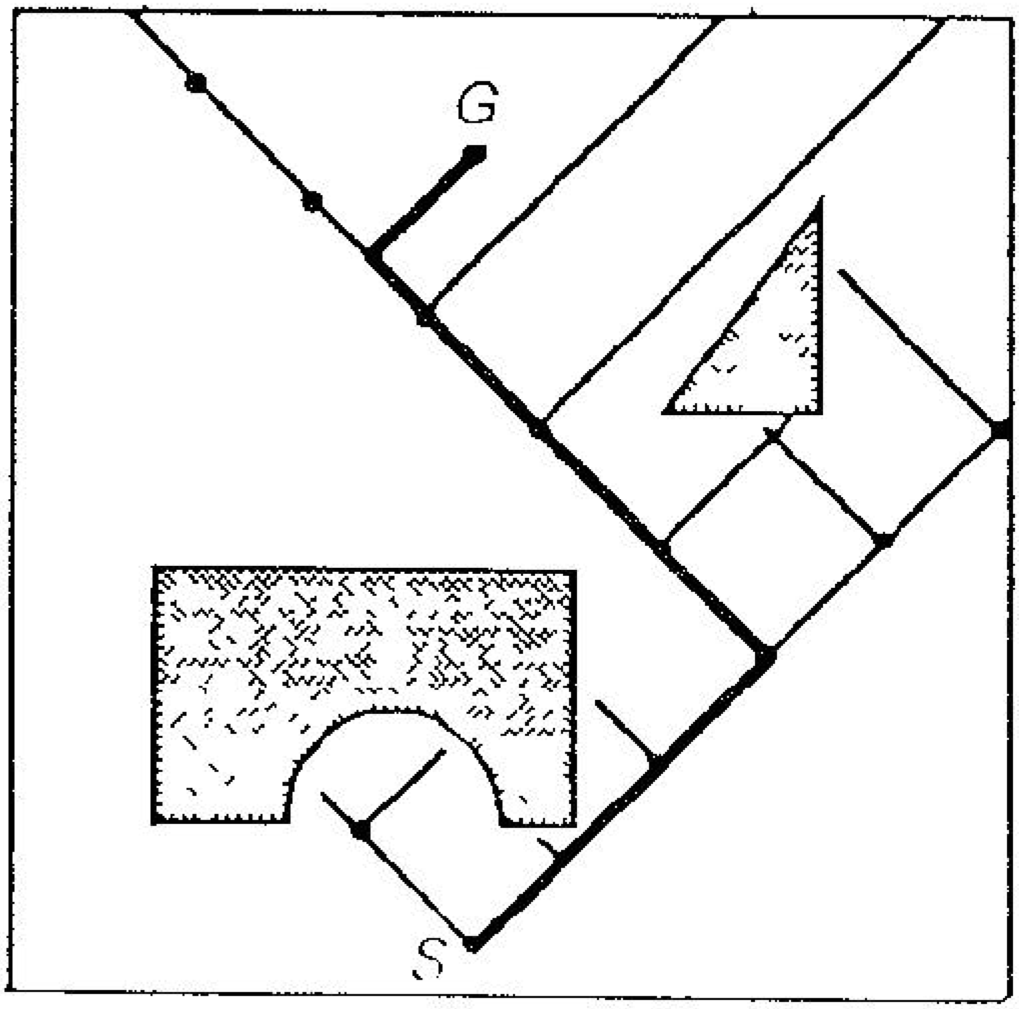


Figure 2.4: A subgoal network. Reprinted from Hwang and Ahuja [HA92].

2.4.2 Cell Decomposition

This MP algorithm divides C_{free} into a set of simple cells. The adjacency relationships between each cell are stored and then the collision free path can be computed. This solution is found by finding the cells that contain the initial and final configurations and then connecting them using a sequence of connected cells. The cells can be decomposed in a few different ways. The first method uses the object boundaries to generate the cell boundaries (object dependent). The second way partitions Cspace into cells of a simple shape and then computes if the cell is in C_{free} (object independent). The object dependent decomposition generates a lot fewer cells than the object independent method but the algorithm is also more complex. C_{free} is better represented by the object dependent method since this decomposition does not contain the gaps around obstacles that are present in the object independent method. Increasing the number of cells in the independent decomposition can reduce this effect but this consumes more memory.

2.4.3 Potential Field

Khatib and Mampey [KM78] first developed this idea of using a potential function for obstacle avoidance. The potential is a scalar function that has a minimum when the robot is at the goal configuration and a high value on obstacles. The function in C_{free} will be sloping downward towards the goal. In this way the solution will be found by following a path down towards the goal configurations.

Two steps are needed to construct the potential. Firstly, an obstacle potential is constructed that has a high value on the obstacle that decreases monotonically as the distance from the obstacles increases. Using the inverse of the distance to the obstacles is a suitable choice. Next, a goal potential is constructed that has a large negative value at the goal configuration that increases monotonically as the distance from the goal increases. The inverse of the distance to the goal is a good choice to make here. The obstacle potential and the goal potential are added together to give the final potential.

Although this approach seems simple, the potential function usually has several local minima at configurations other than the goal. These minima trap the robot and no solution is found. The potential field is therefore often used in conjunction with other motion planners like the subgoal network mentioned earlier. One other problem with the potential field is the problem it has creating an obstacle potential for concave objects. Despite its few problems the potential field is still attractive due to its low computational costs.

2.4.4 Mathematical Programming

This MP approach uses a numerical method to find the optimal solution. This is set up by representing obstacle avoidance as a set of inequalities on the configuration parameters. The solution is thus an optimization problem that finds a path between the start and goal configurations minimizing a certain scalar quantity. This quantity is typically the total distance of the path. Alternatively the optimization could be changed to maximize the distance from the obstacles to achieve a desirable path. This however is better done using Voronoi diagrams that are explained in Chapter four.

2.5 Searching

After C_{free} has been described using a motion planning algorithm the solution is found by searching for a path through the feasible configurations. After a skeleton approach has been used, for example, a search is still required to find the shortest solution. Many searches have been developed such as breadth first, best first, depth first, A^*, bidirectional [BF81], simulated annealing [BL90] and Dijkstra's shortest path algorithm [Dij59]. All of these searches can be used in MP but they each have their unique advantages.

2.5.1 Depth First and Breadth First

The depth first search and the breadth first search are both not very efficient. The depth first search always generates a child of the most recently reached configuration. In this way it travels in straight lines and only changes direction when a obstacle is reached. The solution is thus not going to be the shortest one and in many cases it will seemingly take the longest possible way. The breadth first search generates the children of the earliest reached configurations first. This is similar to a bush fire since it searches all the configurations closest to it before moving outward. This will find the shortest path but it takes a long time.

2.5.2 Best First

The best first search is an improvement since it generates the children of the current configuration and moves to the child nearest to the goal. This search requires some measure of distance to be known. The straight-line distance is a useful metric in this case. This search can also take a long time to calculate if a blind alley exists between configuration obstacles.

2.5.3 A*

The A* search is a good way to find the solution when the minimum cost is desired. This search requires an underestimate of the cost from the current configuration to the goal (cost to go) such as the straight-line distance. The sum of the actual cost so far and the cost to go gives a lower bound on the actual cost. This sum is called the total cost. During the search the child with the lowest total cost is visited first. The performance of A* is linked to the accuracy of the cost to go. The smaller the error in the underestimate the faster the search will be.

2.5.4 Bidirectional

A bidirectional search generates the children of both the start and the goal configuration. It progressively moves outward until one of the children connects and thus finds a solution. This search is efficient when the goal configuration is hard to reach. A good example will be the case when the goal is in a narrow channel between obstacles.

2.5.5 Dijkstra

Dijkstra's algorithm is the most efficient in finding the shortest solution. It finds this solution in $O(n^2)$ time. It works by partitioning the space into two sets of configurations. See Figure 2.5.

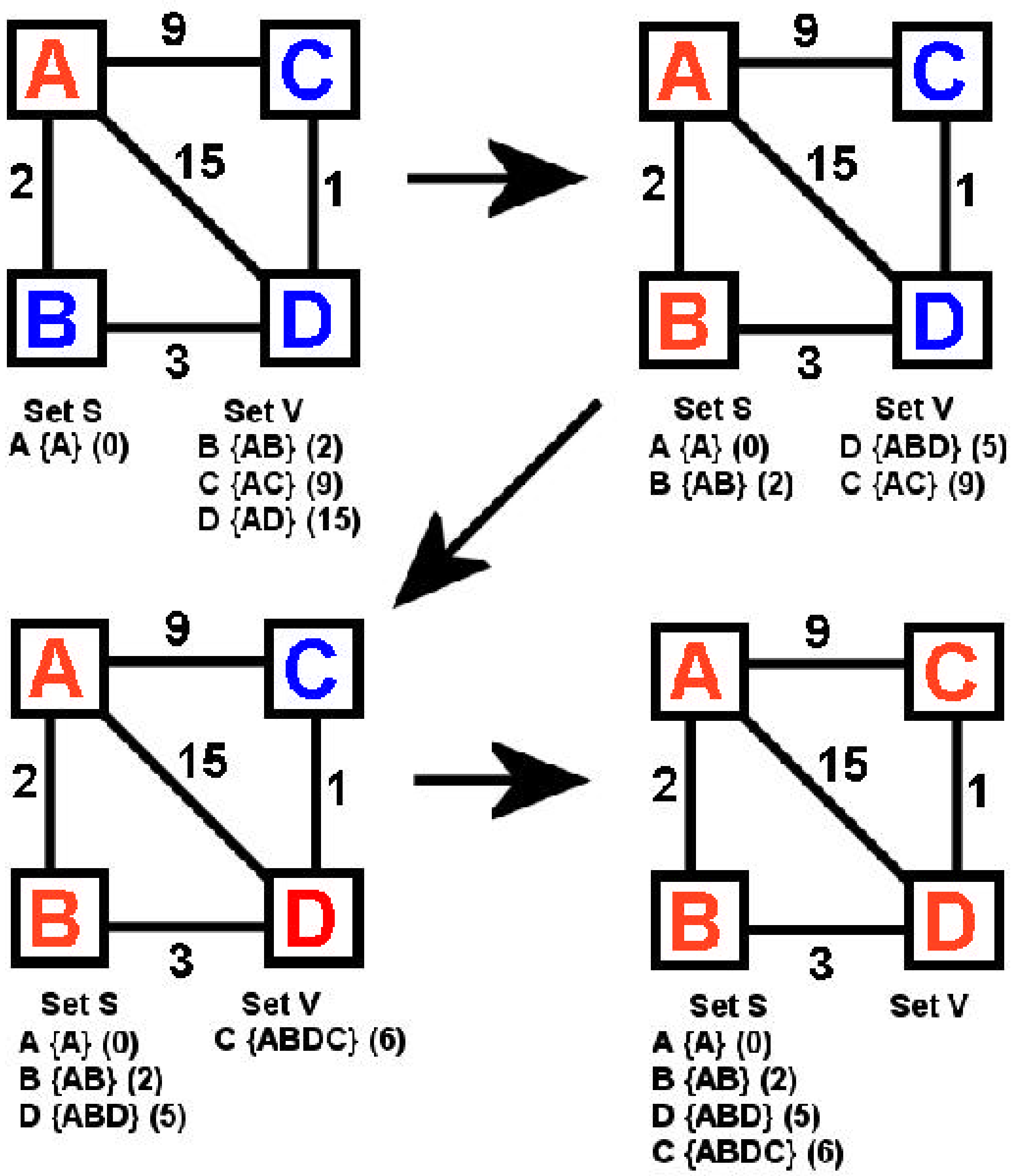


Figure 2.5: This is how Dijkstra's algorithm works. The set S contains the configurations whose shortest paths have already been determined. The set V is made up of all the other configurations and the best estimates of the shortest path to them. The curly braces show the estimated path in set V and the shortest path in set S. The brackets show the estimated distance in set V and the shortest distance in set S. In each iteration, the path with the smallest estimate in V is added to the set S. The estimates are recalculated and the process begins again.

The set S is the set of configurations whose shortest paths have already been determined and the set V is made up of all the other configurations. It also keeps track of the best estimates of the shortest paths (distance and direction) to every configuration. The algorithm begins with only the source configuration in S. It relaxes the neighbours of the set S by updating the shortest path if necessary. The configuration u in V with the shortest path is moved into S. This process continues until there are no more configurations inside the set V. In this way the algorithm not only finds the solution, it also finds the shortest path between the source and any other configuration. In my Voronoi implementation in Chapter four I used this search.

2.5.6 Random Search

If it is not possible to search the entire search space due to exponential time requirements a random search technique can be used. Barraquand and La-tombe [BL90] have successfully used simulated annealing, for example. This method will not be covered here since the next chapter will involve a random MP approach.

2.6 Local Optimization

After a solution has been found it can still be further optimized using a number of criteria. The length of the path, the safety clearance between the robot and obstacles, the total traveling time and the amount of energy spent are some of the typical measures used. Hwang and Ahuja [HA92] express this mathematically as a minimization of

$$\int_{q_{start}}^{q_{goal}} (1+\frac{w}{D(q)})dq$$

where D(q) is the distance between the robot and obstacles; w is the relative weighting factor, and the integral is over the path connecting q_{start} and q_{goal}. This minimization prevents the robot from colliding with obstacles. In Hwang and Ahuja [HA89] an obstacle potential function is used in place of D(q) and this is found to be simpler to compute and converges to an optimum in 20-30 iterations.

2.7 Conclusions

The above standard procedures can be applied to any motion-planning problem but the implementation varies with every different situation. Combinations of approaches can also be used to great effect. Hwang and Ahuja [HA92] discuss many of these combinations in great depth. In the next two Chapters I will only discuss two of these approaches in detail.

Chapter 3 Roadmap Approach

3.1 Introduction

With all of the previous approaches, the number of degrees of freedom of the robot and its environment was the major limiting factor. Barraquand et al. [BKL$^+$97] developed a random sampling scheme for path planning that has been successful in coping with larger dof. Kavraki and Latombe [KL98] extended this idea to solve multiple planning problems involving robots with 3 to 16 dof using a Probabilistic RoadMap planner (PRM). Koga et al. [KKKL94] used a similar approach to automatically synthesize video clips with graphically simulated human and robot characters involving 78 dof.

The roadmap is an extension of the skeleton approach. A roadmap is a collection of paths that allow for efficient navigation of Cspace. The idea behind the roadmap is as follows: If you wanted to travel between two places, you would need a map. Since this map does not exist, it has to be created. This map does not need to be too detailed since you only need the major highways to find a connecting route. Upon creation of the map, you are then able to travel between any two points in the region provided that you can see the highway from these two points. This simple idea summarizes the roadmap approach. It involves a pre-processing phase where the roadmap is constructed. After this, many different queries can be made to check if roads exist between two points.

Most other planners only allow one query. The fact that roadmap planners can do more is a considerable advantage. One of the other advantages of the PRM is that its performance is measurable. By this, I mean that the probability that it will find a path bounds its running time. A significant improvement in running time results from lowering the probability of correctness. This is the notion of *probabilistic completeness*.

In this Chapter, I will discuss two implementations, the potential field planner and the PRM. I will also assess their performance along with the associated visibility and path clearance assumptions.

3.1.1 Definitions

The *clearance* of a configuration q is the minimum distance from q to the boundary of C_{free}. In three dimensions, this corresponds to the radius of the largest sphere centred at q that is still inside C_{free}. If the configuration q is inside an obstacle, the *clearance* returns a negative number corresponding to the minimum distance from q to the boundary of C_{free}. Therefore, if a configuration is in C_{free} a positive distance is returned; if it is not in C_{free} a negative distance is returned; and if it is on the boundary of C_{free} zero is returned.

A *trap* is a basin of attraction, in a potential function that is almost completely surrounded by forbidden configurations. A configuration *sees* another configuration if a simple path such as a straight line can connect them. The *degree* of a configuration is the number of configurations that it is connected to. The *connectedness* of a region in Cspace is proportional to its degree. The *least connected* region in Cspace contains the configuration that has the lowest degree. A *component* is a set of configurations where a path is known to exist between every configuration. A *deterministic planner* is guaranteed to find a path between any two configurations but it takes a long time to do so. A *milestone* refers to a single configuration in a roadmap.

3.2 Potential Field Planner

Koga et al. [KKKL94] in their implementation extended the potential field planner that was first used by Barraquand and Latombe [BL91]. The potential function is a positive function with a global minimum of zero at the goal configuration; that is

$$U : C_{free} \rightarrow R^{+} \cup \{0\}\ .$$

Barraquand and Latombe [BKL+97] describe some techniques that automatically generate this function. These techniques are similar to the ones mentioned earlier in Chapter Two. A solution is found by alternating between down and escape motions until it reaches the global minimum.

The down motion works by taking a configuration q_s and checking its adjacent configurations for a lower potential until it cannot find one. This motion will gravitate downward until it reaches a local minimum. The outline for the down motion algorithm is as follows:

```
Down-Motion($q_s$)

1. Store $q_s$ in q.
2. While q is not labelled a local minimum:

    a. Pick at random up to h (arbitrary number) configurations
       adjacent to q until q′ is found such that $U(q') < U(q)$.
    b. If this is successful then reset q to q′ else label q as a local
       minimum.

3. Return q.
```

The adjacency computation in step 2(a) takes advantage of the clearance function. The creation of a set of adjacent configurations to q relies on the geometric properties of c, the *clearance* of q. Adjusting each configuration component q_i, in n dimensions by up to $c/\sqrt{n}$ remains inside C_{free}. The expression of the set of adjacent configurations to q is the product of n intervals:

$$\prod_{i = 1 \ldots n} [q_i + c/\sqrt{n} \,,\, q_i - c/\sqrt{n}] \;.$$

The random generation of adjacent configurations becomes a matter of choosing a random point in the adjacent set.

The escape motion involves a random walk of length L, hoping that this will end up out of the local minimum. Barraquand and Latombe have shown that on average a random walk of length L ends up $\sqrt{L}$ away from the starting point. They also suggest choosing $\sqrt{L}$ to be the radius of the Cspace. The outline for the escape motion is as follows:

Escape-Motion(q_l)

```
1. Pick at random the length, L, of the motion.
2. Store $q_l$ as q and set l to 0.
3. While l < L:

    a. Pick at random a free configuration q′ adjacent to q.
    b. Set l to be l plus the max. of $|q_i - q_i'|$, where i varies through
       each dof.
    c. Store q as q′.

4. Return q.
```

If the down motion falls into a trap, this escape motion has little chance of escaping the local minimum.

The potential field planner works by successively moving down and up until it reaches the goal. The algorithm is the following:

```
Potential field planner($q_{start}$,$q_{goal}$)

1. Store down-motion($q_{start}$) as $q_l$
2. While $q_l \neq q_{goal}$ do:

    a. Do:

        i. If total configurations generated so far exceed an
           arbitrary number n then return NO.
        ii. Store Down-Motion(Escape-Motion($q_l$)) as q′.

       until $U(q_l') < U(q_l)$.
    b. Store $q_l'$ as $q_l$.

3. Return YES.
```

This planner has been used with success in many situations but it still has a few niggling problems. Despite the presence of escape motions to deal with local minima, it can still be trapped indefinitely. This uncertainty means that the potential field planner's convergence speed remains unknown. In the next section, I will discuss the PRM that was written to overcome these problems.

3.3 PRM Construction

Kavraki et al. [KSLO96] developed this planner that overcomes the problems with the potential field planner. No problem specific heuristics are required with the PRM. Additionally, in the next section, I will show that its convergence speed is known and can be used to bound the running time of the algorithm.

3.3.1 Pre-processing Phase

The pre-processing phase involves choosing r random configurations that do not intersect any obstacles. After this every pair of chosen configurations that are closer than d are checked for connectedness using a simple and fast planner. Finally, the roadmap is re-sampled by generating s extra configurations in the least connected regions. The algorithm is as follows:

```
Pre-processing:

 1. Store 0 as i.
 2. While i < r do:

     a. Pick a random configuration q in Cspace.
     b. If the clearance of q is greater than 0:

          i. store q in Roadmap and
         ii. increment i.

 3. For every pair of milestones whose distance apart is less than d
    try to connect.
 4. Starting with the least connected region c and in order of
    increasing degree generate a configuration q in the
    neighbourhood of c and try to connect. If successful store q in
    Roadmap. Repeat until s extra configurations are added.
```

The above algorithm still leaves a lot of room for choice. The first choice to make is the design of the random configuration generator in step 1. The generated roadmap needs to represent accurately the connectedness of Cspace. Thus the random generator must generate a wide sample of configurations.

Another decision to make is the choice for connect in step 3. A fast and simple straight line planner is a good choice to make. A deterministic planner would not be a good choice because these planners take a long time to execute and accuracy is not necessary at this stage of the algorithm.

The distributions of r and s also need to be determined. Barraquand et al. [BKL+97] recommend first randomly generating 2/3 of the total number of milestones to be generated and then re-sampling to generate the other 1/3. This choice tends to decrease the number of connected components in Roadmap. Alternatively, the re-sampling step 4 could be replaced by trying to join the different components using a deterministic planner (permeate). This guarantees that all the components in Roadmap are distinct and would generate the best queries but this takes a long time and should only be used as a last resort.

One final choice is the determination of the total size of Roadmap. The greater the number of milestones the higher is the accuracy and speed of the queries. This is related to performance and will be discussed later.

3.3.2 Querying Phase

After the roadmap has been created the querying process can commence. This phase involves trying to connect both q_{start} and q_{goal} to the same component of the roadmap. If it is not initially successful it tries g times to find an intermediate configuration q that sees both a milestone and the start or goal configuration. If none of these are successful it returns NO and terminates. Otherwise, if it is successful in finding q it returns YES if both q_{start} and q_{goal} are connected to the same component. If however they aren't connected to the same component then NO is still returned. The algorithm is as follows:

Query(q_{start},q_{goal}):

1. For both q_{start} and q_{goal} do the following:

```
    a. If there exists a milestone m that sees $q_i$ then store m as $m_i$.
    b. Else repeat g times:
         i. Pick a random configuration q in the neighbourhood of $q_i$
            until q sees both $q_i$ and a milestone m.
        ii. If successful store m as $m_i$.
       iii. Else if all of g fail return NO.
2. If $m_{start}$ and $m_{goal}$ are in the same connected component return
   YES else return NO.
```

This algorithm varies according to the choices made in the pre-processing phase. If the re-sampling step were changed to permeate

then step 1(b)iii would have to be changed to return FAILURE. Choosing permeate guarantees that every YES or NO answer from the query is correct. Therefore, NO in step 1(b)iii must be changed to FAILURE.

In step 1(b)i the generation of a neighbour of q_i may require several guesses before one can be found. The random generation must take this into account to speed up the algorithm.

3.4 Performance

Both the potential field planer and the PRM provide solutions to many practical problems with high dof. The potential field planner, however has a few niggling problems related to the local minima that it cannot escape easily. The PRM is a better approach since it does not have any problem-specific heuristics and its convergence speed is known.

The PRM is probabilistically complete. This means that if a solution exists it will find one with high probability in bounded time. The running time grows slowly with the probability we are willing to tolerate. Barraquand et al. [BKL+97] have guaranteed the performance of the PRM if the following assumptions are true:

- Visibility Assumption The *Visibility Assumption* or *e-goodness* states that each configuration sees a significant portion of C_{free}. C_{free} is *e-good* if each configuration can see a ε portion of the other configurations. So for example, if C_{free} is a circle then every configuration can see every other configuration. Then C_{free} is said to satisfy the visibility assumption and would be 1-good.

  The value of ε is inversely proportional to the running time of PRM and to the number of milestones in the roadmap. However, even high values of ε cannot prevent C_{free} from containing narrow passages. Consider C_{free} made up of two circles overlapping slightly. The value of $\varepsilon \approx 0.5$ and yet C_{free} still contains a narrow passage. The visibility assumption yields high performance but it is not strong enough and another is required.

- Path Clearance Assumption The *Path Clearance Assumption* states that between the tw configurations given to the query there exists a collision free path T that achieves some clearance ε_{inf} between the robot and the obstacles. This is a powerful assumption and if true, no re-sampling step or deterministic planner is required in the pre-processing phase. The value ε_{inf} is also inversely proportional to the running time of PRM and to the number of milestones in the roadmap.

3.4.1 Summary

Barraquand et al. state that the number of milestones needed grows only as the absolute value of the log of the probability of an incorrect answer decreases. They prove this given the assumptions stated above. If the assumptions are true then the PRM produces extremely good results. Otherwise, it is still a good planner since it can deal with high dof. The potential field planner in my opinion is not as good since its convergence speed is unknown.

3.5 Conclusions

In this chapter, I have compared two planners that have been successful in practical motion planning. I have implemented a roadmap approach using Voronoi diagrams that I will discuss in the next Chapter.

Chapter 4 Voronoi Implementation

4.1 Introduction

The PRM is probabilistically complete; it sacrificed a percentage of its correctness for substantial gains in speed. In lower dimensions, however the complexity required is vastly smaller and such a sacrifice is not required. Constructing exact planners that always find a solution if it exists is fast in low dof. In two dimensions, I have implemented the Voronoi diagram approach and the Visibility graph approach. They are both roadmap planners since queries are only answered after a map of Cspace is built. These planners are also exact and efficient. In this Chapter, I will explain my implementation of these approaches and show that the Voronoi diagram is not only faster but it yields a more desirable path (where the paths do not touch the obstacles) as well.

4.2 Voronoi Diagrams

McKerrow [McK91] states that Voronoi diagrams can be used to divide the environment into regions. Lee and Drysdale [LD81] show how partitioning the plane into polygonal regions, each of which is associated with a given point, forms a Voronoi diagram. The region associated with a point is the locus of points closer to that point than any other point. Edges separating two regions are composed of points equidistant from the given points. Therefore, the Voronoi diagram is the set of lines equidistant from two or more points. This makes an excellent planner since the created paths will be desirable. The created paths lie in between the obstacles and in doing so, they will not touch any obstacles.

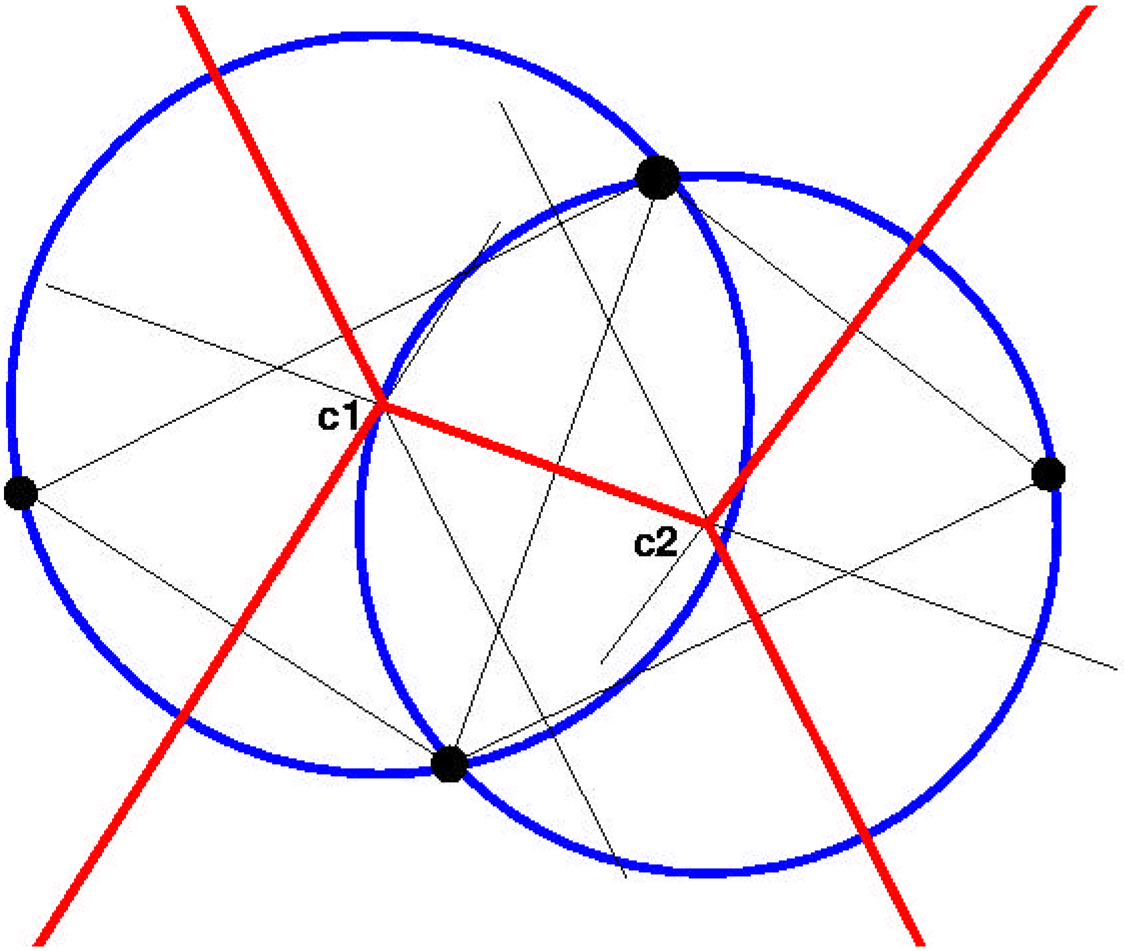


Figure 4.1: Constructing a Voronoi Diagram is as follows: The thick lines in red specify the Voronoi Diagram, the small black circles specify the points, the thin lines specify the lines equidistant from the points and the big circles prove the correctness of the Voronoi Diagram.

Figure 4.1 shows how a Voronoi diagram is constructed. The first step involves drawing lines that bisect every pair of points in the region. Figure 4.1 displays these bisections as the thin lines. The intersection of these lines determines the corners of the Voronoi diagram (c1 and c2 in Figure 4.1). A circle centred at this corner coordinate will intersect the three points closest to it (the big circles in Figure 4.1). The final Voronoi diagram is the collection of lines drawn between the corners (the thick lines in Figure 4.1). Yap [Yap87] devised a way to find this diagram in O(nlogn) time.

4.3 Implementation

I implemented the Voronoi diagram approach using Matlab version 5.2 in Unix. Matlab is a useful mathematical language and since a Voronoi function comes built in, it was the logical choice. Six steps are involved in the planner. Firstly, the environment is set up. The Voronoi diagram is then constructed based on the points of the obstacles and the boundary. The diagram is then pruned so that only the lines outside of the obstacles remain. The start and final configurations are then connected to the pruned diagram. Finally, the lines are smoothed and a Dijkstra search is performed to find the shortest path.

4.3.1 Set up

I set up the obstacles as polygons made up of the corner coordinates. The polygons can be any shape; they can be concave or convex. The obstacles are programmed manually in the simulation but sensors could easily attain this information in a practical environment. For simplicity, the robot is assumed to be a point. In a practical environment, the polygons must be grown to accommodate for the shape of the robot. A Minkowski Set Difference is a good choice to make here.

The Voronoi diagram generates lines between points and does not generate lines around points. It was necessary to create a boundary to the simulation so that the Voronoi diagram would generate paths around obstacles. This boundary is made up of a number of equally spaced points in the shape of a rectangle. The actual shape of the boundary does not matter as long as it goes around all the obstacles, the start and final configurations.

More points than just the corners represent the boundary because the Voronoi diagram generates better results. Using a spacing equal to the length of the shortest edge of the obstacles is sufficient. The same principle applies to the obstacles. Points are inserted around each obstacle with the same spacing that was used to generate the border. See Figure 4.2. When the path planner receives all of these points along with the start and goal points the computation begins.

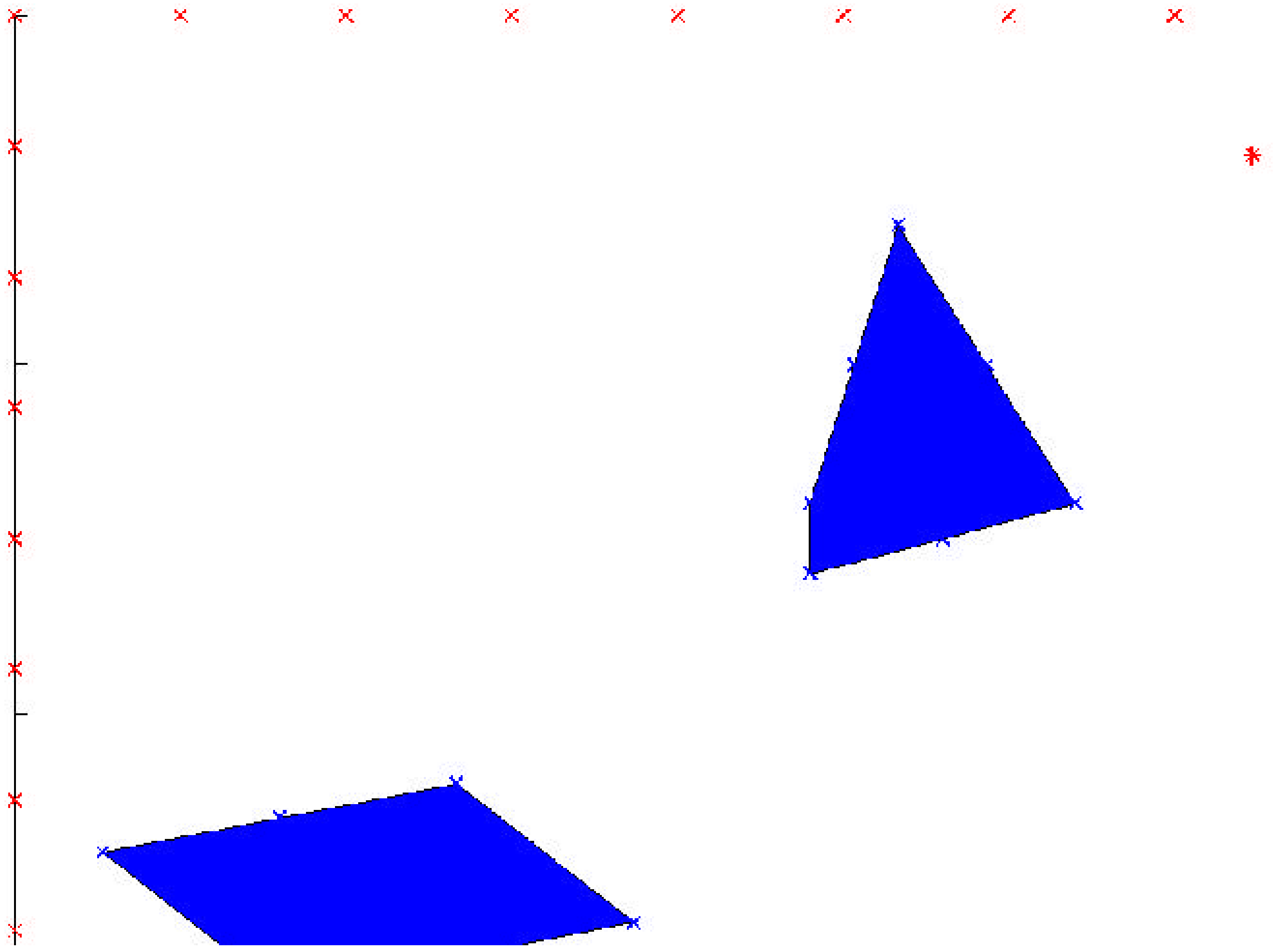


Figure 4.2: This is the input given to the Voronoi diagram calculation. The points in the outer rectangular box specify the boundary. The blue objects are the obstacles. The points surrounding the obstacle specify the obstacles. The two star shaped points specify the start and goal configuration.

4.3.2 Voronoi Construction

The Voronoi diagram takes O(nlogn) time, where n is the number of points. The function takes in a set of points representing the obstacles and the boundary and returns a set of lines specifying the diagram. Figure 4.3 shows the output from the Voronoi diagram computed from the input in Figure 4.2. The Figure shows that the Voronoi diagram intersects the obstacles. These points must be pruned before any path planning may commence.

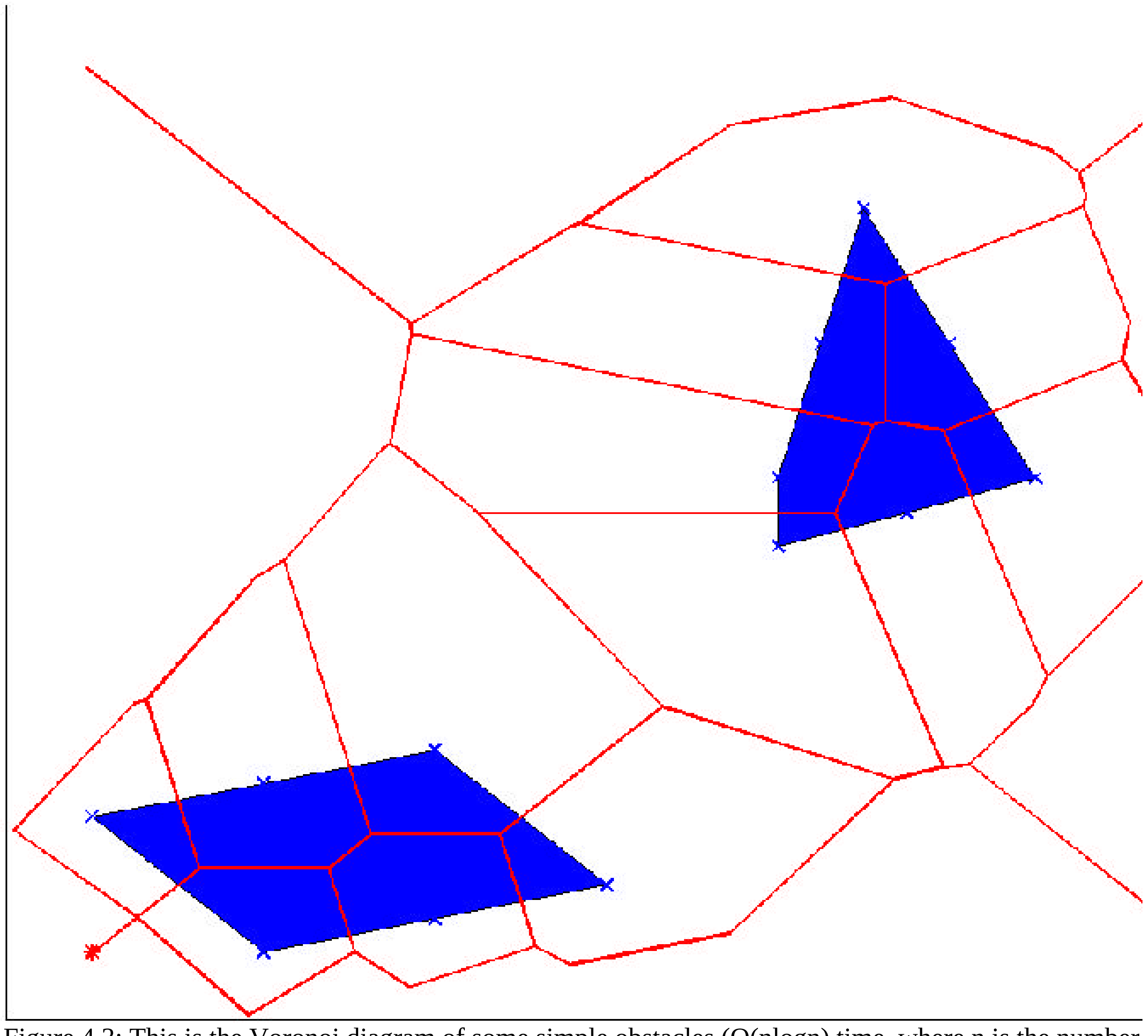


Figure 4.3: This is the Voronoi diagram of some simple obstacles (O(nlogn) time, where n is the number

of points).

4.3.3 Pruning

The lines of the Voronoi diagram that go through any obstacle edge are discarded. Every line in the Voronoi diagram is checked with every edge of the obstacles. This takes O(n^2) time since there are O(n) lines in the Voronoi diagram and O(n) obstacle edges, where n is the number of points. The lines that touch the outer edge of an obstacle are not pruned because it is conceivable that a robot will touch an obstacle. To distinguish between the lines that go through an edge and those that just touch is not as easy as it first looks. This requires in depth classification of every type of line intersection. For example, two lines may intersect (go through each other), they may just touch or not touch at all. If the lines just touch they may form a T-junction, an L-junction or they may be parallel to each other. If they just touch and are parallel, they may overlap or not overlap. See Figure 4.4 for more details.

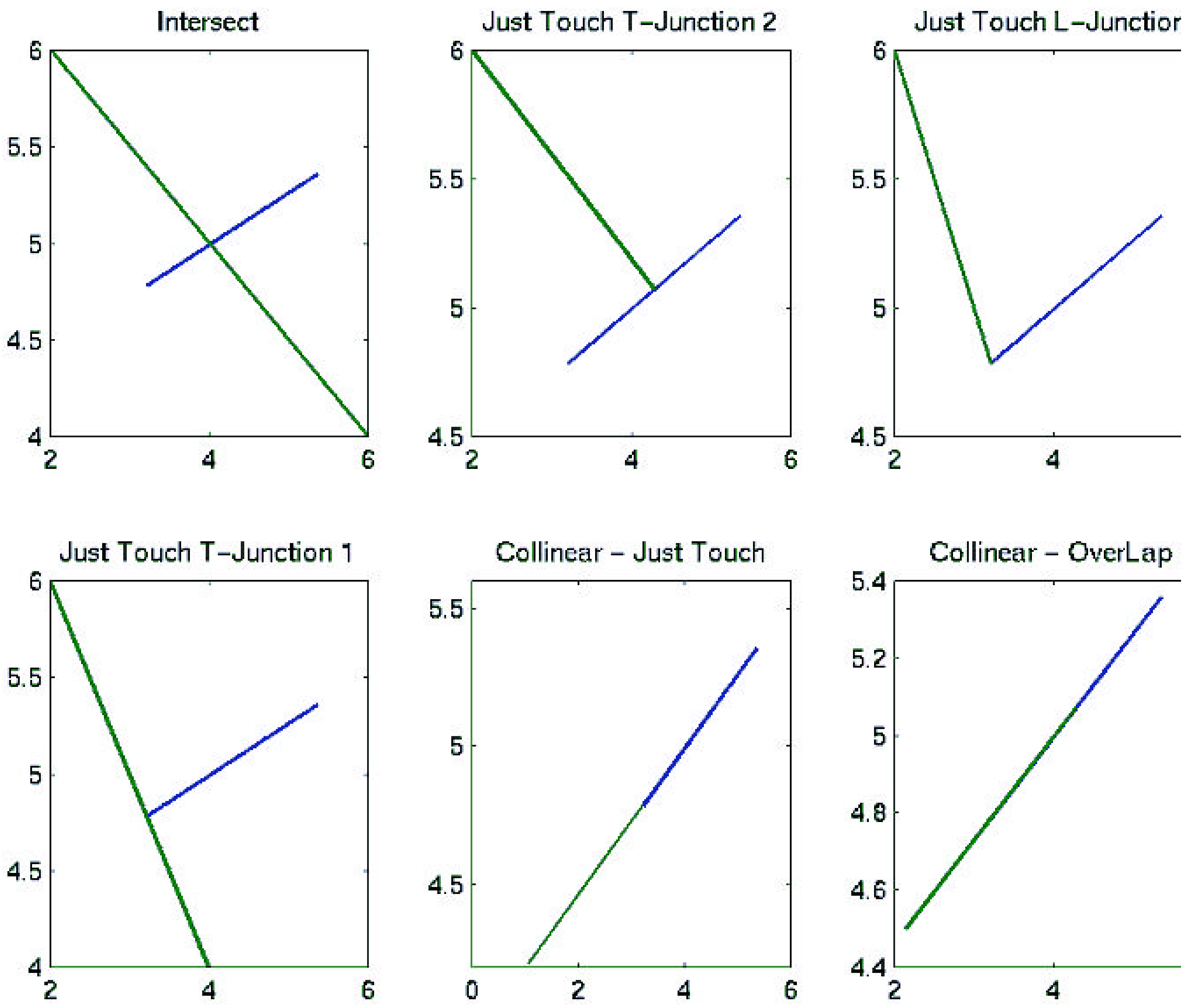


Figure 4.4: Two lines intersect in the above six ways.

My implementation classifies each intersection and uses this information to determine if the path intersects the polygon or not. After the entire pruning process has finished, the function returns the set of lines that are outside of all the obstacles but inside the boundary. Figure 4.5 shows the output of the pruning function when it received the Voronoi diagram from Figure 4.3.

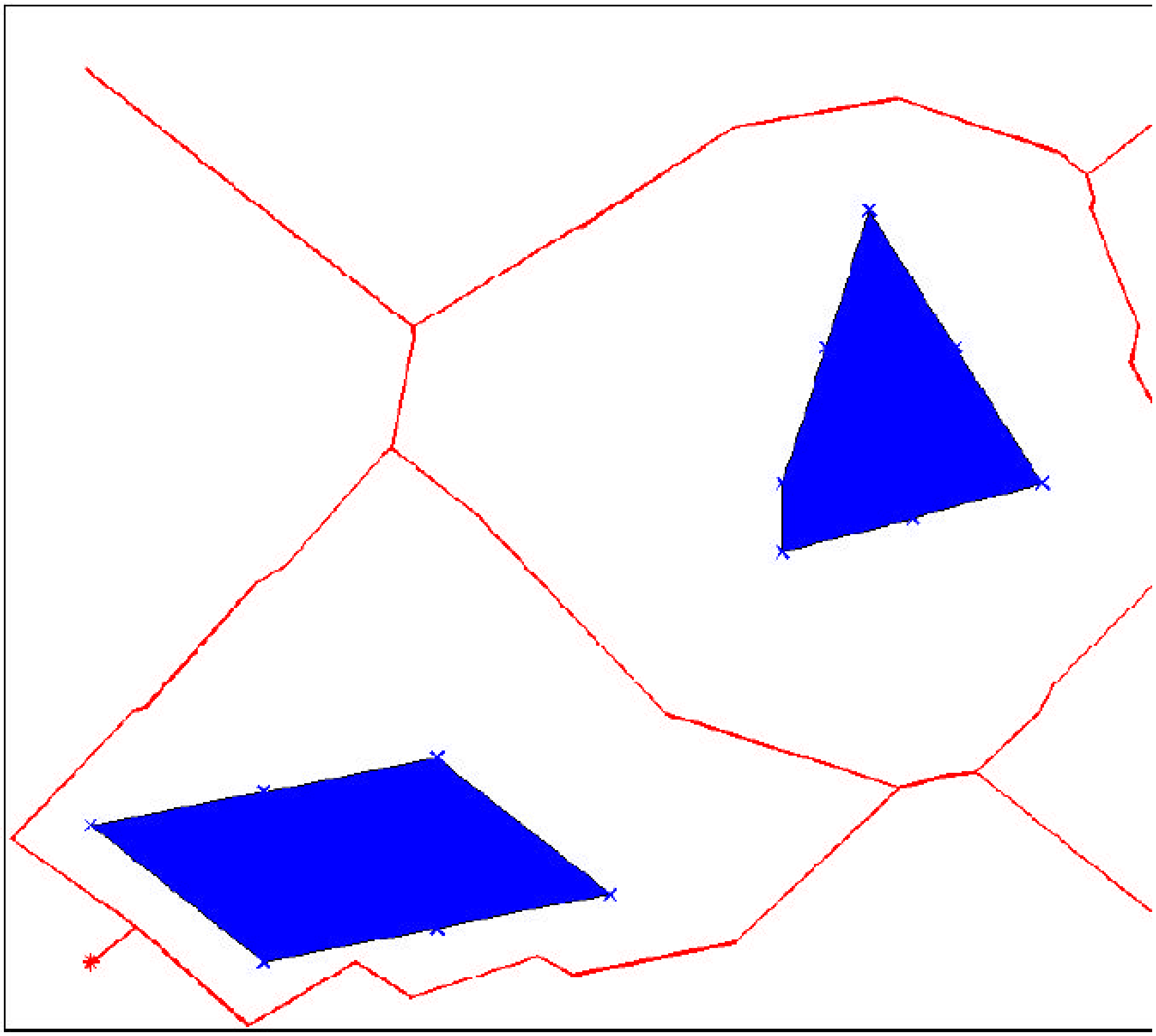


Figure 4.5: This is the pruned Voronoi diagram (O(n^2) time, where n is the number of points).

4.3.4 Connecting to diagram

The start and final configurations are both connected to the pruned Voronoi diagram by finding the nearest edge. This takes O(m) time, where m is the number of edges in the pruned diagram. This takes special consideration of any obstacles that may be in the way. If an obstacle is blocking the direct path, another edge must be chosen.

4.3.5 Smoothing and Dijkstra Search

After the configurations have been connected to the diagram, a path can now be found between the start and the goal. Before searching for the shortest path, the edges are all smoothed to remove the large corners that exist in the Voronoi diagram. Joining the midpoints of each connecting edge smooths each corner. Achieving this takes O(m^2) time, where m is the number of pruned edges. This takes so long because for each edge it has to search for all the edges that it is connected to. The pruned Voronoi diagram contains some corners where three edges meet. See Figure 4.6. Retaining the original corner and joining it to the mid points of the three edges resolves this situation.

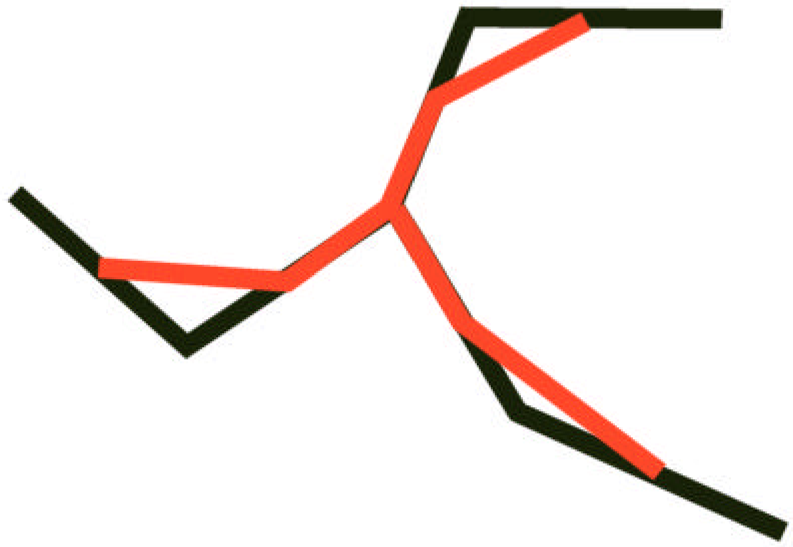


Figure 4.6: Smoothing an intersection of three edges. The black line is the original line. The midpoints of these line segments are joined to create the smoothed line.

Finally, a simple and efficient Dijkstra search finds the shortest path. This takes O(m^2) time, where m is the number of smoothed edges. Figure 4.7 shows an example of the shortest smoothed path. The order in which smoothing and searching for the shortest path is done does not greatly affect the solution. It is easier to search for the path and then smooth this path because every corner will only intersect one other edge. Smoothing first however, will ensure that the shortest paths of all the smoothed edges are returned. Either way the total path length will not change much, if at all.

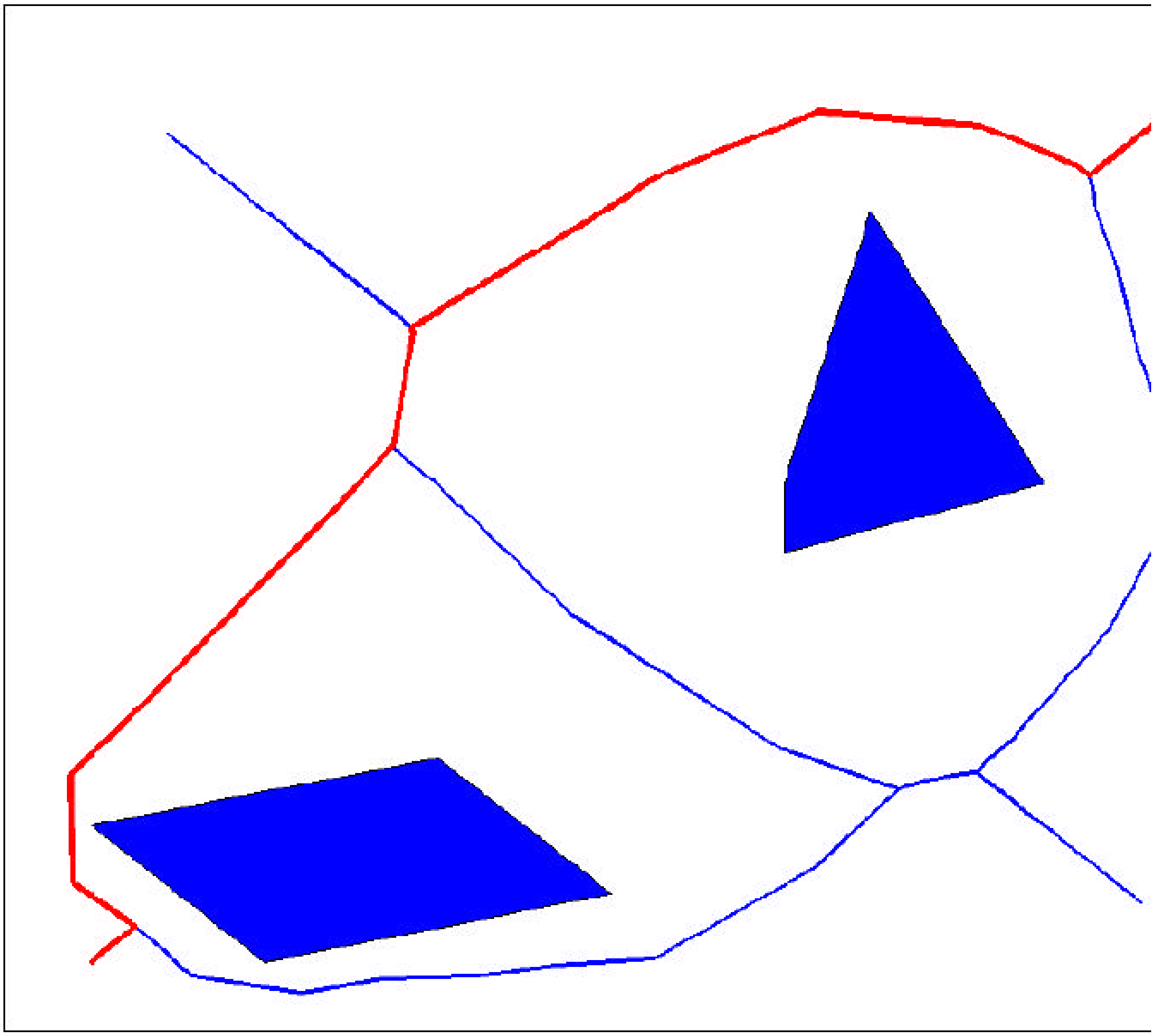
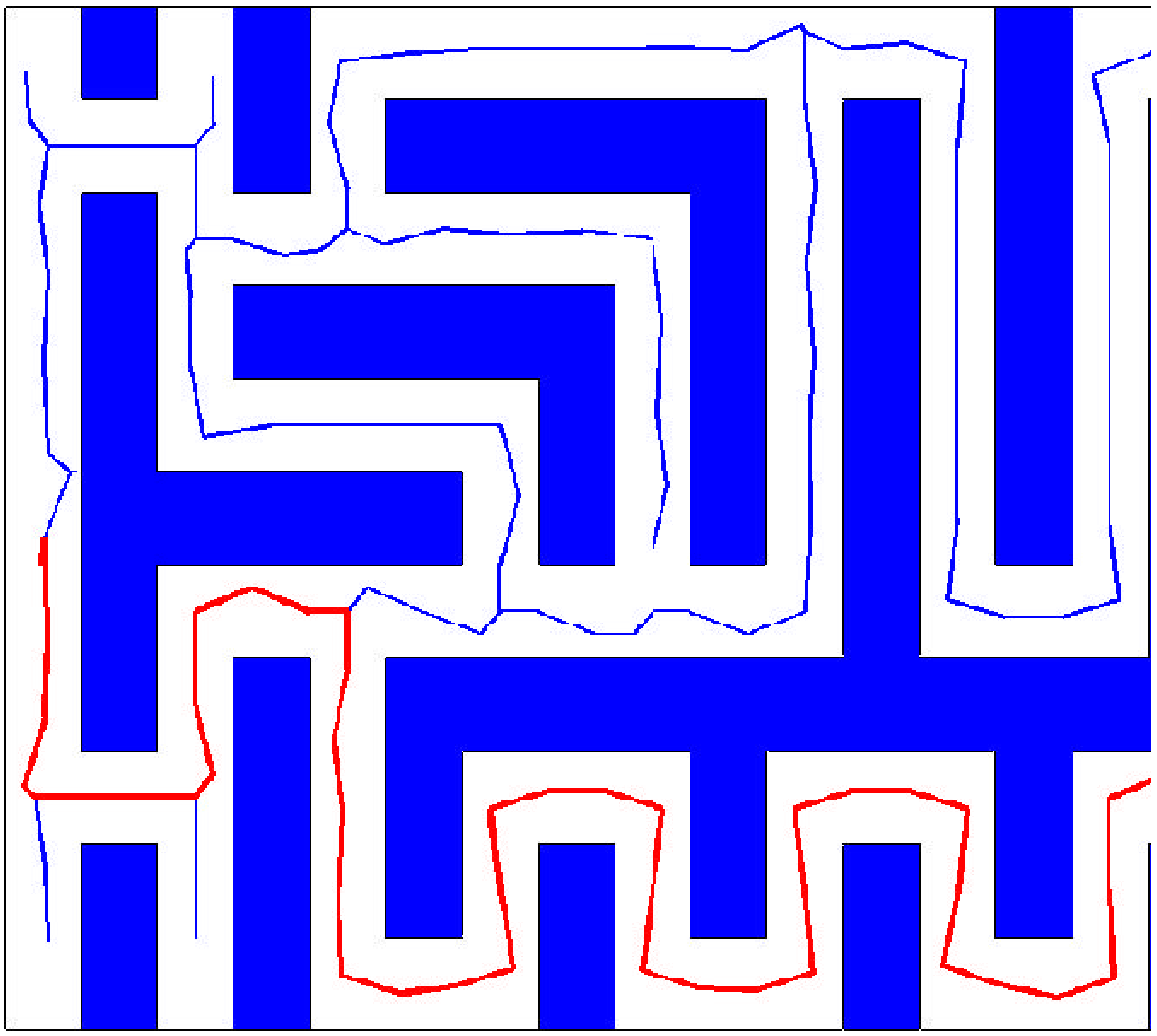


Figure 4.7: This is the final smoothed Voronoi diagram with the shortest path ((O(m^2) time, where m is the number of points).

4.4 Experiment

To test my implementation I built a maze and compared the results with the standard visibility graph (simply connecting all the points together). See Figure 4.8 for details. The visibility graph produces a shorter solution path but the solution is undesirable since it continually touches the obstacles. On the other hand, the path produced by the Voronoi method avoids the obstacles and takes less time to find as well. The Voronoi approach takes O(n^2) time whilst the Visibility graph takes O(n^3) time. The Visibility graph is a good approach in two dimensions but its tendency to touch the boundary of the obstacles suggests that the Voronoi diagram is a better approach to use.



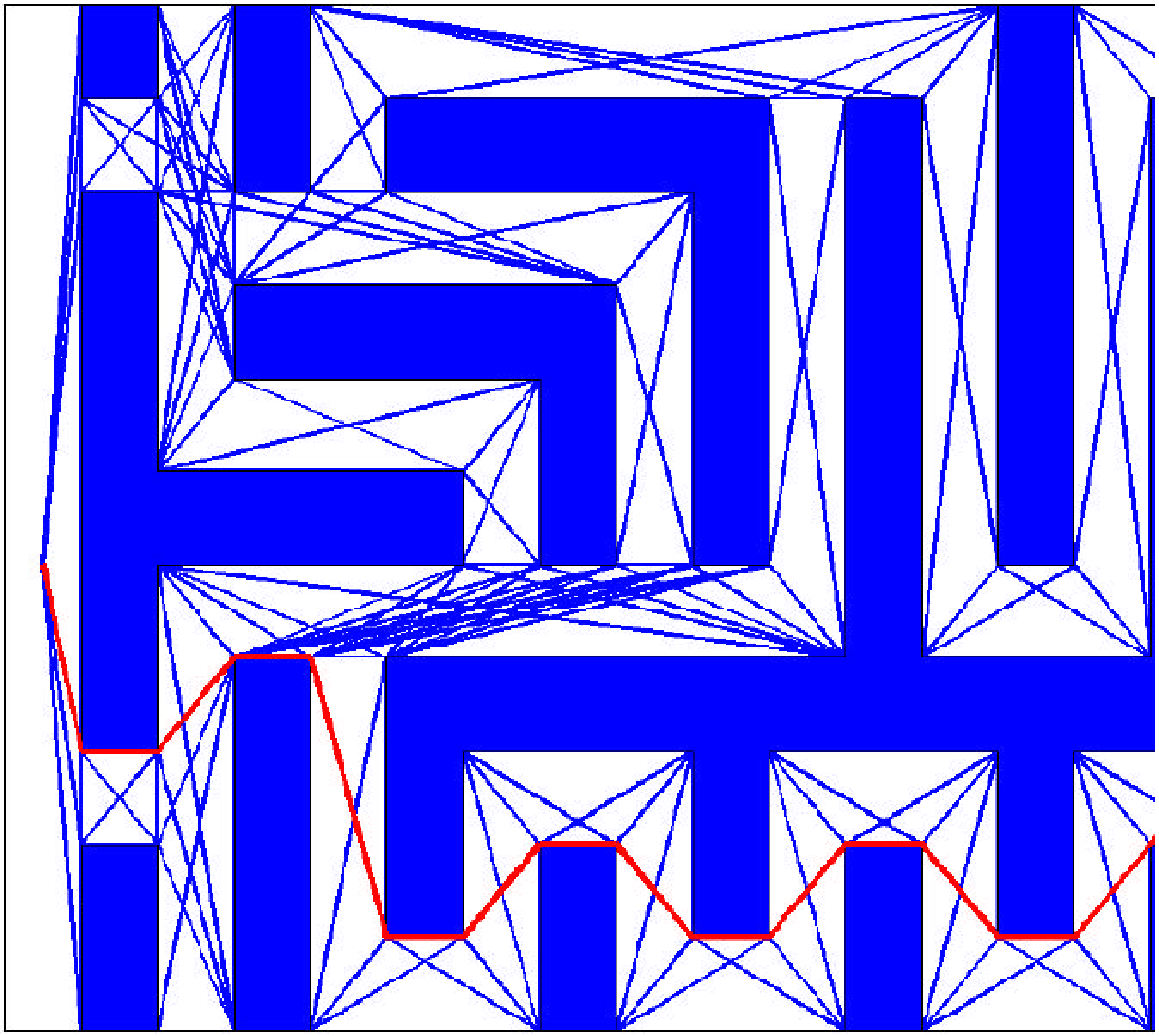


Figure 4.8: The maze (top) is the smoothed path using the Voronoi approach (O(n^2) time, where n is the number of points). The maze (bottom) is the smooth-ed path using the Visibility graph approach (O(n^3) time).

4.5 Suggestions

A real robot finds it hard to navigate around sharp corners. It would be better if the whole path were a single smooth curve. The current implementation smooths the paths to a certain degree but some sharp corners remain. This needs to be improved. This would require inserting more points into the path until a certain level of smoothness is attained.

Takashi and Schilling [TS89] developed the generalized Voronoi diagram that is the locus of points which are equidistant from object boundaries. This is better since the pruning step can be skipped. The boundary also does not need to be specified since the generalized Voronoi diagram returns the paths around the obstacles. Additionally for the same reason, only the corners of the polygon need to be entered. The advantages of this generalized Voronoi diagram need weighing up against the increased time taken to compute the diagram.

Extending the Voronoi approach to three dimensions would be useful. Hwang and Ahuja [HA92] state that this would be more complicated and it would not be obvious what to suggest as features. The Voronoi diagram among polyhedra is a collection of two-dimensional faces. The question to be answered is how to navigate these faces. In n dimensions the Voronoi diagram is a collection of n-1 dimensional faces. It becomes increasingly difficult to use these faces to find the shortest path. I suggest that it is pointless in trying to extend past three dimensions because proven planners such as the PRM are successful in navigating higher dof.

Another suggestion is to use the Voronoi approach as a local planner in PRM. This would have to be tested because whilst it yields better results than a simple straight-line planner does, it also takes more time. The trade off between speed and complexity needs pursuing.

4.6 Conclusions

The Voronoi diagram approach is a very attractive roadmap approach to use in low dimensions. It is accurate, fast and produces a desirable path. Using this approach in low dimensions and using the PRM in higher dimensions boosts performances in accuracy and speed. The Voronoi diagram is more accurate than the PRM in lower dimensions and the PRM is faster than the Voronoi diagram in higher dimensions. Together they make a useful partnership in combating a whole range of problems.

Chapter 5 Conclusion

Barraquand et al. [BKL$^+$97] state that no single planner is likely to be the most efficient for all possible problems. Every application requires a hand made solution. In high dimensions the PRM is likely to contribute to a good solution but it may not. If the workspace is not static for example, PRM cannot generate a reasonable roadmap and fails to give a solution. Likewise with the Voronoi diagram planner in two dimensions. If the obstacles are moving it will not be able to generate a solution. This example shows that even the best solutions for one environment fail in another environment. In static environments the PRM is able to generate solutions involving over 75 dof and the Voronoi diagram planner is able to generate accurate solutions quickly in two dimensions, but in moving environments they are both poor choices. Therefore, if an application involves a static environment either one of the Voronoi diagram planner or the PRM must be at least considered because these are two of the best planners available.

Appendix 1 Original Honours Proposal

Title:

Linux Driver for RTX Robot and high level motion representation

Author:
Michael Wager
Supervisor:
Dr Peter Kovesi

Background

In the early 1990's the Computer Science department attained an RTX robot. The RTX robot is simply an arm with 7 degrees of freedom. It was used extensively to perform many tasks such as the putting of a golf ball into a hole. However, the robot eventually became just a piece of equipment on the ground floor that was rarely touched.

The lack of usage of the robot is not due to the inability of the robot but the age of the software. The RTX robot was initially working perfectly when connected to a 386 running DOS. However, timing problems occurred when the software was transferred to a faster 486. This was because the RTX software was executing too quickly and thus could not synchronise its signals with the robot. The manufacturer compounded the problem since they did not provide any updates to the original DOS driver. Although an update would be nice, it is no longer necessary since most of the department computers are running linux. What is required is a linux driver. The driver would not be easy to manufacture since no linux driver exists and there has been no development in the area. The effort is worth it however, since development of a driver for linux would bring the robot up to date and facilitate further research and experiments.

One such experiment would be to discover and implement higher-level constructs for the motion of the robot. At present, the typical way of expressing motion to a robot is by specifying a starting point and an end point with possibly some acceleration parameter. This low-level approach is tedious since you would need to specify every point along a path to get the required arm movement. It would be extremely helpful to have higher-level motion constructs that would enable the user to concentrate on other significant problems.

McKerrow [3] and Trevelyan [4] have previously looked at high-level robot motion. They both consider error detection and recovery to be a vital part of motion. With Trevelyan it was extremely important that the robot does not behave chaotically. This is because he worked for many years on the sheep-shearing project where a sheep was sheared by a robot. A single mistake could easily lead to the death of a sheep. His work has made significant progress in high-level motion representation.

Aim

I will aim to develop an efficient and robust RTX robot driver for linux. The foundations of the driver would closely follow the original specifications outlined in the RTX manual [1]. The driver would also follow linux conventions in communicating via serial ports [5]. To build upon this I would look in depth at various ways of expressing high level motion and implementing at least one of them. Finally, I would look at ways of extending the application to allow it be used by a mathematical language like Matlab. This would allow the robot to be used to its potential and would enable further motion experimentation.

Method

Driver Research:
(Week 5 - Week 6) Discover the best ways to engineer a driver for the Robot. Do some experimentation to find out if Java would be feasible as compared to C. Decide on a structure for the driver.

Initial Experimentation:
(Week 7 - Week 9) This would involve sending and receiving basic bit patterns following the specification in the manual. [2] Start thesis.

Further Engineering and Research:
(Week 10 - Week 13) To build on the foundation by implementing some higher level functions. Concurrently look into ways of expressing motion effectively.

Exams and a Holiday:
(Study - Hol 1) Study for Exams, do brilliantly and then take a well-earned break.

Motion Research:
(Hol 2 - Hol 3) Research ways of expressing motion and then decide on one to implement. Continue thesis.

Motion Implementation:
(Week 1 - Week 2) Implement and test one way of expressing motion. Work more on the Thesis.

Matlab and Thesis:
(Week 3 - Week 4) Look at ways of using the driver inside external applications. Continue writing the thesis.

Draft Thesis:
(Week 5 - Week 8) Complete draft thesis and hand to supervisor.

Seminar Preparation Final Thesis:
(Week 9 - Week 11) Prepare Seminar and hand in final thesis early.

Seminar:
(Week 12) Give Seminar and then take a well-earned break before exams.

Software and Hardware Requirements

I would need a linux machine connected via a serial cable the RTX robot. I will need to have C and Java with the Java Communications 2.0 API that would enable me to communicate via the serial port.

References

[1] 1987, *Programming RTX using the library*, Universal Machine Intelligence Limited, London.
[2] 1987, *Using intelligent periphals communications*, Universal Machine Intelligence Limited, London.
[3] McKerrow, P. J. 1991, *Introduction to Robotics*, Addison-Wesley, Singapore.
[4] Trevelyan, J. P. 1992, *Robots for Shearing Sheep Shear Magic*, Oxford University Press, New York.
[5] *The Linux Serial Programming How To*, http://www.linuxhq.com/ldp/howto/Serial-Programming-HOWTO.html

Bibliography

[AGHI85]

T. Asano, L. Guibas, J. Hershberger, and H. Imai. Visibility polygon search and Euclidean shortest path. In *The 26th Symposium on Foundations of Computer Science*, pages 155-164, Portland, Oreg., 21-23 October 1985.

[BF81]
A. Barr and E. A. Feigenbaum. *The Handbook of Artificial Intelligence*. William Kaufmann, Los Altos, Calif., 1981.

[BKL+97]
J. Barraquand, L. Kavraki, J. Latombe, T.-Y. Li, R. Motwani, and P. Raghavan. A random sampling scheme for path planning. *International Journal of Robotics Research*, 16(6):759-774, 1997.

[BL90]
J. Barraquand and J. C. Latombe. A Monte-Carlo algorithm for path planning with many degrees of freedom. In *Proceedings of IEEE International Conference on Robotics and Automation* [IEE90], pages 1712-1717.

[BL91]
J. Barraquand and J. Latombe. Robot motion planning: A distributed approach. *International Journal of Robotics Research*, 10(6):628-649, 1991.

[BN90]
M. Branicky and W. Newman. Rapid computation of configuration obstacles. In *Proceedings of IEEE International Conference on Robotics and Automation* [IEE90], pages 304-310.

[Can87]
J. F. Canny. A new algebraic method for robot motion planning and real geometry. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 39-48, Los Angeles, 12-14 October 1987. IEEE.

[CH92]
P. C. Chen and Y. K. Hwang. Practical path planning among movable obstacles. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 444-449, Sacramento, 7-12 April 1992. ACM.

[CL95]
H. Chang and T.Y. Li. Assembly maintainability study with motion planning. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1012-1019, 1995.

[Dij59]
E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269-271, 1959.

[HA89]
Y. K. Hwang and N. Ahuja. Robot path planning using a potential field representation. In *The IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 569-575, Sand Diego, 4-8 June 1989.

[HA92]
Yong K. Hwang and Narenda Ahuja. Gross Motion Planning - A Survey. *ACM Computing Surveys*, 24(3):219-291, Sep 1992.

[IEE90]
IEEE. *Proceedings of IEEE International Conference on Robotics and Automation*, Cincinati, 13-18 May 1990.

[Kav95]
L. E. Kavraki. Computation of configuration-space obstacles using the fast fourier transform. *IEEE transactions on Robotics and Automation*, 11(3):408-413, 1995.

[Kav97]
L. E. Kavraki. *Algorithms for Robotic Motion and Manipulations*, chapter Geometry and the discovery of new ligands, pages 435-448. A. K. Peters, 1997.

[KKKL94]
Y. Koga, K. Kondo, J. Kuffner, and J.C. Latombe. Planning motion with intentions. In *Proceedings of SIGGRAPH'94*, pages 395-408, 1994.

[KL98]
L. Kavraki and J. C. Latombe. *Practical Motion Planning in Robotics: Current Approaches and Future Directions*, chapter Probabilistic Roadmaps for Robot Path Planning, pages 35-53. John Wiley, 1998.

[KM78]
O. Khatib and L. M. Mampey. *Fonction decision-commande d'un robot manipulateur*. DERA/CERT, Toulouse, France, 1978.

[KSLO96]
L. E. Kavraki, P. Svestka, J. C. Latombe, and M. Overmars. Probabilistic roadmaps for fast path planning in high dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566-580, 1996.

[LD81]
D. T. Lee and R. L. Drysdale. Generalization of Voronoi diagram in the plane. *SIAM Journal on Computing*, 10(1):73-83, 1981.

[McK91]
Phillip John McKerrow. *Introduction To Robotics*. Addison Wesley, 1991.

[PMF89]
B. Paden, A. Mees, and M. Fisher. Path planning using a Jacobian based freespace generation algorithm. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 1732-1737, Scottsdale Arizona, 14-19 May 1989.

[Rei79]
John H. Reif. Complexity of the mover's problem and generalizations (extended abstract). In *20th Annual Symposium on Foundations of Computer Science*, pages 421-427, San Juan, Puerto Rico,

29-31 October 1979. IEEE.

[TS89]
O. Takashi and R. J. Schilling. Motion Planning in a Plane using Generalized Voronoi Diagrams. *IEEE Transactions on Robotics and Automation*, 5(2):143-150, 1989.

[Yap87]
C. K. Yap. An O(nlogn) algorithm for the Voronoi diagram of a set of simple curve segments. *Discrete and Computational Geometry*, 30(2):365-393, 1987.

File translated from T$_E$X by T$_T$H, version 2.34.
On 16 Nov 2000, 21:37.