University of Alberta

Library Release Form

Name of Author: Douglas Jon Demyen

Title of Thesis: Efficient Triangulation-Based Pathfinding

Degree: Master of Science

Year this Degree Granted: 2007

Douglas Jon Demyen
10820 - 78 Avenue, Apt. 109
Edmonton, Alberta
Canada, T6E 1P8

Date: _____

If you never fall,
you are not really trying

University of Alberta

Efficient Triangulation-Based Pathfinding

by

Douglas Jon Demyen

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Efficient Triangulation-Based Pathfinding** submitted by Douglas Jon Demyen in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Dr. Michael Buro

_____

Dr. Hong Zhang

_____

Dr. Farbod Fahimi

Date: _____

To my parents and my brother Jeff,
and also to my girlfriend Christina

Abstract

Pathfinding for commercial games is a challenging problem, and many existing methods use abstractions which lose details of the environment and compromise path quality. Conversely, humans can ignore irrelevant details of an environment that modern search techniques still consider, while maintaining its topography.

This thesis describes a technique for extracting features such as dead ends, corridors, and decision points from an environment represented using a constrained Delaunay triangulation. The result is that the pathfinding task is simplified to the point where the search algorithm need only decide to which side of each obstacle to go, while all features of the environment are retained.

We present algorithms which search the triangles of the environment (Triangulation A*) and the decision points identified (Triangulation Reduction A*). We also explore a number of techniques which deal with finding paths for circular objects of nonzero radius and enhancements to various aspects of the search.

Acknowledgements

I think it is a great privilege to be supervised by someone with as much passion and enthusiasm for his work as Michael Buro. Often seeming torn between his interest in games and his duties as a professor, the twin desires of producing work with academic merit and that which is "cool", help his students achieve their goals while working on something they enjoy and of which they can be proud.

During my time at the University of Alberta I have had much valuable input from my professors, who were always willing to lend their help and advice, and fellow students through many an exchange of ideas. I have found the environment here to be very much a community, and the many events an excellent forum for both interaction and hearing about many other exciting fields.

I am fortunate to have such great friends in Regina who have made the effort to visit me when possible and see me when I am in town. I am glad that we could stay close despite that our busy schedules do not allow us to talk as much as we would like.

To those of my friends who have moved to Edmonton, it was always nice to have a familiar face to see, despite being so far from my hometown. It was good to know others were going through the same change. And those who I have met since I came have helped make this city my home.

Of course the largest portion of my thanks goes to my parents for not only influencing who I am, but more directly by encouraging me always to better myself whether through my studies or otherwise, and even helping me out when my education or other responsibilities claimed most of my time. My brother Jeff also had a large part of my being where I am today, whose respect means as much to me as that of anyone.

Finally, the support of my girlfriend Christina was likely most responsible for my move to Edmonton which allowed me to follow my dreams. Her care has helped me though what would otherwise have been a difficult transition and her love has encouraged me every step of the way.

Contents

List of Figures

Chapter 1

Introduction

1.1 Artificial Intelligence

Artificial Intelligence is a vast field of research, both in breadth and depth, so any definition encompassing its many approaches and techniques would have to be suitably general. One interpretation, however, is that the aim of Artificial Intelligence is to emulate human-like behaviour when applied to various tasks. This is because we as humans view ourselves, by and large, to be intelligent.

An auxiliary, if not just as important, goal of Artificial Intelligence is to use the above process to gain further understanding into the way the human brain operates. Because many applications studied by Artificial Intelligence are performed well by humans, successfully directing a computer toward the same task can be an indication that the method by which it is achieved is similar in some way to the human approach to the problem.

Games have emerged as a particularly interesting application area within Artificial Intelligence. Here was an area where humans could perform well in the presence of a large number of possibilities for play, they could look at a few details of a state of a game and produce a summary of it, they could develop complex strategies and quickly learn insight into that of their opponent. To even approach these abilities, a modern computer would have to perform lengthy techniques despite continuing advances in technology.

This was an indication that despite their sequential speed and accurate storage abilities, computers were not intelligent in the same ways as humans. Where computers would consider a huge number of possibilities using only brute force, humans could easily discern which had the most potential. While computers learned about their opponent over the course of many games, humans could do it in few, if more than one.

Certainly this was an application area where learning about and applying methods employed by humans could only improve the play by computers. Indeed, it has been an active area within Artificial Intelligence, with much progress being made in classical games like checkers [43, 44], Othello [7, 8], card games like poker [4, 3], and more. More recently, with great increases in computing power and the rise in popularity of computer and console games, *commercial games* has become a popular area within games itself.

Originally, restrictions in both time and memory confined the "Artificial Intelligence" used in commercial games to being predefined by the programmers during development. Any perceived intelligence was simply a matter of these predefined (or *scripted*) behavious being appropriate for the situation. Besides not being dynamic, much less truly intelligent, this process could take a considerable amount of work on behalf of the developers.

Only recently has the hardware available to run such games had sufficient processing speed and memory capacity to allow application of Artificial Intelligence. However, because

commercial games are constantly pushing the boundaries of modern computing, updates to the game's state are done many times each second and the vast majority of time and memory are allotted to other areas such as graphics, sound, physics, networking, and so on, and so Artificial Intelligence still receives very small amounts of resources within which to work.

For this reason, Artificial Intelligence in commercial games must work within strict time and memory requirements. This is often a point of contention between academia and the commercial games industry, with academic solutions often requiring resources not available in practice, and commercial approaches not having sufficient academic merit (often still being based on scripted solutions).

1.2 Pathfinding

When implementing Artificial Intelligence in a commercial game, the first task is often pathfinding. Pathfinding is the process of determining a set of movements for an object at a particular position and orientation (collectively, *configuration*), which when applied, result in the object being in another configuration.

This sequence of movements, or *path*, between the initial configuration (the *start*) and the final configuration (the *goal*) must also have the property that at no point when the object is on the path, should its configuration be invalid. In pathfinding, this often means that the object must not collide with any obstacles in the environment in which the object exists.

This is an intuitive requirement, since when deciding on a route to take to the store, a human would not consider a path which goes through a building to be valid. This meets well with the goal of simulating a human's decisions given a similar situation. Obviously if an object in a commercial game chose a path which brought it into collision with a wall, one would not deem its actions to be very intelligent.

As a consequence of this, pathfinding is often considered the most fundamental Artificial Intelligence task in commercial games. No matter what techniques are used to govern its behaviour, a character in a commercial game cannot seem intelligent if it cannot move around its environment reasonably and without colliding with obstacles.

This requirement for "reasonable" motion belies the complexity of another property that is often desirable for pathfinding—*optimality*. What it means for a path to be optimal varies by the application, however two of the most common definitions are that the path be as short as possible, and that it take the least amount of time to travel.

Using a distance metric, an optimal path is the valid path between the start and goal configurations whose combined distances between adjacent intermediate configurations is no greater than any other. This is the most common measure of optimality, and is intuitive as such. If someone is going from New York to Los Angeles, we would not consider a route that passes though Miami to be intelligent, unless this person had some reason for going there first.

Another common measure of optimality is the time required for the path to be traversed. This type of optimality need be considered in situations where parts of the environment take more time to traverse than others, if the object being moved takes time to turn or accelerate, and so on. In most other cases, the shortest path is also the one which takes the least amount of time to follow. One can imagine a situation where this is not the case: if travelling between cities the fastest route is often along connecting highways, even if following dirt roads might result in less distance, as one can only drive along dirt roads at certain speeds.

There are other possibilities for defining an optimal path, such as the path which involves the fewest number of rotations, and some that consider additional constraints based on the

environment, like avoiding an enemy's line of sight whenever possible. That said, not all situations require finding an optimal path, and one can sometimes find a path that is nearly optimal with a fraction of the resources required to find one that is exactly optimal.

In a commercial game, a character can often select a path that is not quite optimal while still appearing intelligent. As long as the path selected does not move in a very unintuitive way, the character can usually be forgiven for a suboptimal path, as humans very seldom concern themselves with taking the absolute best path, so long as it is rational.

Now, in order for an object to appear to move around its environment intelligently, we have the requirement of finding a collision-free path between the start and goal configurations that is either optimal according to some measure, or at least very close to it. Again, the application of commercial games imposes strict requirements on both time and memory usage.

This problem has been studied extensively, but nonetheless this thesis presents a novel approach, explained in Section 1.4, that meets all these requirements, as well as having properties conducive to further applications and enhancements.

1.3 Example

In the spirit of Artificial Intelligence, we look at the pathfinding problem from the perspective of a human to see what we can learn from the way a human goes about determining a path between locations. Consider a man attempting to find the best path from the door of the restaurant shown in Figure 1.1 to the table at which his friend is sitting.

The man will not plan his individual steps, he is instead likely to decompose the problem to that of moving between small areas of the restaurant that are free of obstacles such as tables and walls. On top of this, he will not consider dead ends. While it seems trivial that he would not ponder going into corners, this also applies to going into the kitchen or down the hallway leading to the bathrooms—if these do not lead to the table at which his friend is sitting, he will not spend time thinking about them as possibilities. Figure 1.2 shows the areas of the restaurant eliminated by this technique alone. Current pathfinding techniques do not avoid searching these places any more than they would any other part of the restaurant.

Another problem common to current pathfinding techniques is that if the goal could be in a small area or require the object to go through a small area, then the search will have to consider moving between all such areas, and thus the restaurant will be made up of more,

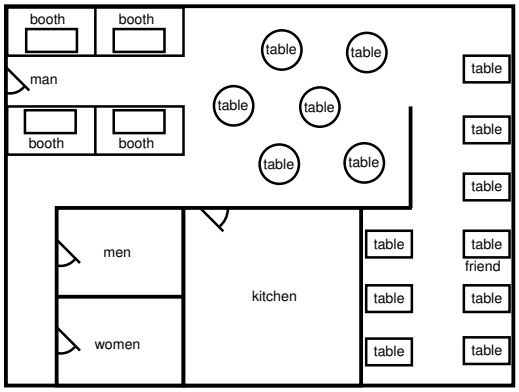

Figure 1.1: Environment for pathfinding example

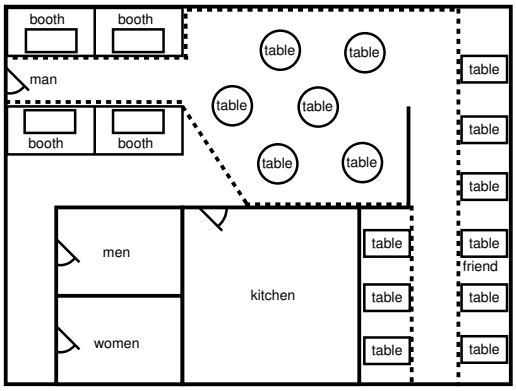

Figure 1.2: Dead ends of the restaurant cut off by dotted lines

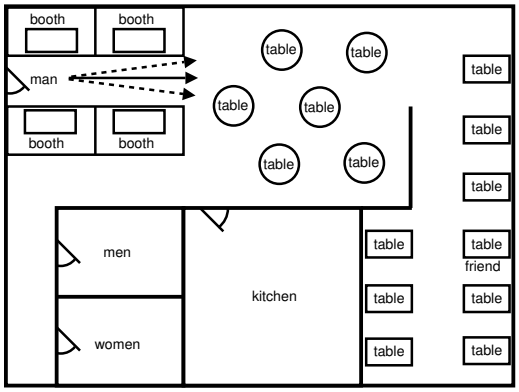

Figure 1.3: Proceeding down a corridor

smaller areas, forcing the search to make more decisions and thus use more memory and time. In our example, however, the man will not take any more time to decide how to get to his friend in a restaurant where the walls were zigzagged (creating several small nooks) than if the walls were straight, and he can still represent the walls precisely in either case.

Let us say that upon entering the restaurant, there are booths to both sides of the man. We know he will not sit down at either, since he will not consider dead ends. This again seems trivial, but it reduces his decisions to either moving forward or moving backward. Since backward would move him to where he has already been, that is not a very useful option either. Thus, his options until reaching the other side of the booths are simply a series of forward steps. Obviously he will not separate these conceptually, and consider his first "move" to be going to the end of the booths. In other words, he will consider this to be a *corridor*. This is shown in Figure 1.3.

Again, most pathfinding techniques will consider all these as moves, which result in more decisions than realistically have to be made. Also, if the area between the booths was wider than he man, some pathfinding techniques would consider lateral movement as well as forward movement as shown by the multiple arrows in Figure 1.3. This is also unnecessary, since the man will not consider at which side of this corridor he will walk when determining a path—he only knows he will traverse it, and his exact path within it will be determined by how he enters and exits the corridor.

At the other side of the booths is an area with tables sitting out from the walls, to which

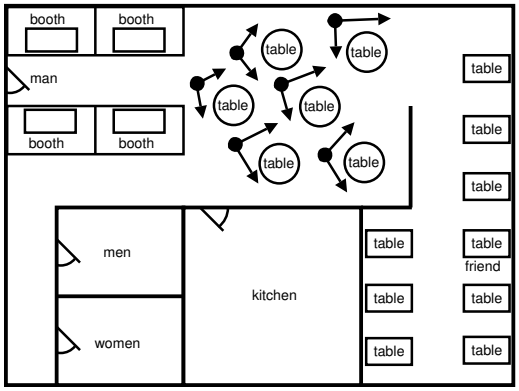

Figure 1.4: Points at which the man must make a decision

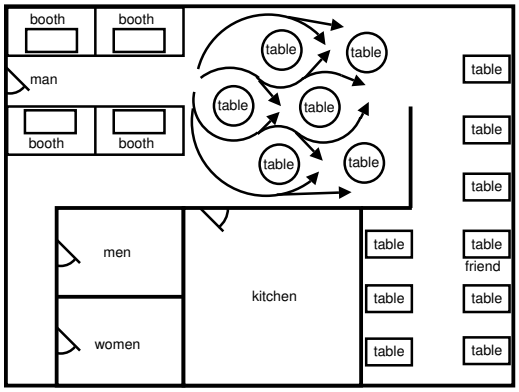

Figure 1.5: Possible paths through the table area

the man can walk on either side. This is the first point at which the man must make an actual decision. He may consider at this point different combinations of going around the tables. Figure 1.4 shows the points where such decisions must be made. The number of such combinations is exponential in the number of obstacles, however many resulting paths are obviously too long, and he will quickly rule them out. When looking at the best such combinations, he will finally consider the actual steps that would lead him to the chosen side of each table. Some of the possible paths through the tables are illustrated in Figure 1.5.

With this information he can determine which of these combinations of decisions leads to the shortest path through the tables. He will not consider *how* he will walk around each table, because once the series of decisions is made, the best path adhering to this combination of decisions can be determined quickly. Again, this shows how the man will not consider individual moves, lateral moves in larger areas, or dead ends not leading to the goal, as decisions.

At the opposite side of this open area of tables is a row of tables on each side of another corridor. At one of these tables is the man's friend. This time the man will not consider going all the way through the corridor as this would take him past his goal, but will instead stop at the table at which his friend is sitting. We see that while he has considered going through earlier corridors as single moves, it has not hindered his ability to move partway through this one. Figure 1.6 shows how the path does not traverse the whole corridor.

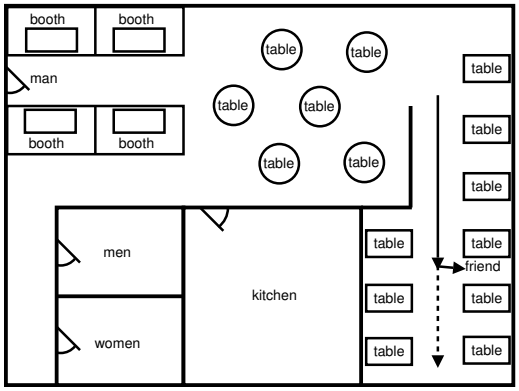

Figure 1.6: Walking partway down a corridor and having a seat in a dead end

Similarly, when reaching his friend's table, the man can sit down even though he did not consider such a "dead-end" option at any other table in the restaurant. The man can then follow his chosen path by moving to the end of the booths, going around the tables in the open area in a manner which results in the shortest overall path, going partway down the corridor of tables, stopping at his friend's table and sitting down, also shown in Figure 1.6.

Finally at this point the man will consider his individual steps; once making the high-level decisions about his route, he can easily plan these steps minimizing the distance travelled while avoiding obstacles and adhering to the chosen route.

1.4 Approach

The approach taken to pathfinding in this thesis will be described in the context of the above example, in this section. When looking at the decision process of a human, one can see the improvements that are possible with the proper abstractions and representation.

First, we use a triangulation to represent the environment, which is described in terms of line segment barriers (polygonal representation). This addresses the issue that a human will consider areas of the environment (triangles) instead of individual steps when approaching the pathfinding problem.

This also has the advantage that the number of triangles in the environment is not sensitive to its size. A large area has no more triangles than a smaller area with the same features; this number is only affected by the complexity of the barriers in the environment.

In addition, the triangles added to deal with more detailed areas of the environment do not affect other areas. This is akin to how the man could consider small and complex areas of the restaurant without taking longer to consider larger open spaces.

The approach taken in this thesis uses the properties of a constrained triangulation (one formed around the aforementioned line segment barriers) to identify dead ends, corridors, and decision points in a connected area of the environment. The result of this process is a graph where the nodes represent where a decision must be made as to which side of an obstacle to go. The implications of this technique are many, and fit well with the method by which a human solves the pathfinding problem: only considering how to go around obstacles, and considering every path which adheres to these decisions as one.

The pathfinding task then consists of reaching the adjacent such nodes from both the start and goal points by moving to the adjacent corridor if the point is in a dead end, and then considering the decision points at each end of the corridor if there. Only one decision point need be considered for the start or goal if the point is on one already.

Now on the graph, the search need only consider decision points since the paths by which the start and goal points connect to this graph have already been identified. This is similar to how a human can ignore dead ends and move through corridors in one pass throughout the search, even if the start or goal is in such a dead end or corridor.

Because these are the only points at which a decision must be made, it implies what was described above how the man will only consider which corridors he will traverse, and not the details of how he will traverse them. Once a succession of corridors leading from the start to the goal (or between the decision points adjacent to them) is found, the best path through these corridors can be found.

The process of finding such a path can be thought of conceptually as weaving a string through the selected corridors and then pulling it at the start and goal points so that it tightens around the obstacles and forms the shortest path through those corridors. Once the length of this path is determined, it would be adopted as the best path if no shorter paths have been found.

When one path is determined, we can determine which routes around obstacles may provide a shorter path than the best found so far, and consider only those. This process can be continued until an optimal path is found (this is guaranteed when no other routes could yield a shorter path than the best one found), or at least one of acceptable quality.

The above assertion about "pulling the string" is not entirely accurate, since it is often the case that the object for which the path is being found, is not a point object. In the case of the man in the restaurant, he would obviously not appreciate a path which touched tables or walls, since being centered on the path would cause him to collide with them. Again, a main aim of pathfinding is to avoid such obstacles.

For the purposes of this thesis, objects are considered to be circular with some radius. Conceptually, to keep an object of some radius from colliding with obstacles, we center circles of this same radius around the corners of the obstacles around this route, and then pull the string around them. This ensures that the resulting path stays at least the object's radius away from these obstacles, so that when the object's center follows this path, it will not collide with anything, as desired.

When creating the triangulation, we also determine what size of objects can fit through the triangles. When dead ends are identified, we determine for each point in them how large of objects can reach the connecting corridor. When identifying corridors, we determine for each point the maximum object radius that can reach each connecting decision points.

We finally determine the sizes of the largest objects which can travel through entire corridors connecting decision points, so that once searching between decision points, we can determine valid paths for an object of a given size without considering any other structures. While these size considerations are somewhat tangent to the rest of the work in the thesis, they were necessary for applying the technique in the chosen application areas.

1.5 Contributions

Here we will quickly overview the parts of the work described above, as well as some others, which are novel contributions. We will follow that with an outline of the structure of the rest of the thesis.

The aforementioned methods for dealing with objects with nonzero area by measuring the throughput of triangles, dead ends, and corridors has many benefits over existing techniques, which will be described in its respective section.

Also the description above about "pulling the string" with circles around the corners of obstacles was actually accomplished by creating a new version of an existing algorithm called the *funnel algorithm* to deal with circular objects of nonzero radius. The original

and modified versions of this algorithm both run in time linear in the number of triangles through which the path moves.

We then determined an anytime search algorithm for searching through a triangulation which deals with the inherent uncertainty within a triangulation in order to find an optimal path, while being as efficient as possible.

As mentioned earlier, the main work of this thesis is the technique by which the triangulation is examined to find features such as dead ends, corridors, and decision points. This has many advantages as will be described later.

Another anytime algorithm was created to deal with searching solely between decision points after the above technique is performed. This algorithm must make the same considerations as the first, but also deal with a number of possibilities that might exist due to the placement of the start and goal points in the environment.

Other techniques employed include faster point location in order to find the triangle in which the start and goal points reside, and a simple technique for finding the first path quickly, thereby increasing the window of the anytime algorithms in which a solution is available.

1.6 Thesis Outline

Chapter 2 This chapter will review work previously done in Artificial Intelligence in general search techniques, pathfinding-specific search techniques, basic abstractions, and abstractions applied to pathfinding.

Chapter 3 The thesis will here discuss the issues involved with environment representation in pathfinding, the common grid-world representation, general triangulation representations, and the Dynamic Constrained Delaunay Triangulation representation on which this work was based.

Chapter 4 Next we will describe considerations for pathfinding using objects which have nonzero size. This includes a method for determining the maximum radius for a circular object which can move through a particular triangle and an algorithm for calculating it, and then the extension of an existing method for finding the shortest path through a group of triangles (the funnel algorithm) from a point object to a circular one of nonzero radius.

Chapter 5 This chapter describes searching within a triangulation starting with a simple search which can find suboptimal paths, a discussion on the effects of approximated path lengths in the search for an optimal path, and a search incorporating these principles.

Chapter 6 Here the thesis covers the abstraction of the triangulation starting with the different classifications for the triangles (analogous to dead ends, corridors and decision points above), the consequences of these on the resulting graph, an algorithm for creating this abstraction given a Constrained Delaunay Triangulation, and the information that the abstraction itself contains.

Chapter 7 The thesis goes on to describe the procedure for searching the abstraction, the special cases that can occur, the effects of approximated path lengths in the search for an optimal path as it differs between the triangulation and the abstraction, and finally the search algorithm resulting from these concerns.

Chapter 8 An explanation of other enhancements done for the search is given in this chapter. These include point location based on sectors and other approaches, and

a minor modification to the earlier searches meant to produce an initial (possibly suboptimal) solution quickly without affecting the ability for the search to find an optimal solution.

Chapter 9 This chapter contains information from experiments which were performed with the search techniques described earlier. This will include a discussion of the test environments and other application areas, the experimental setup, its results and an analysis thereof.

Chapter 10 The thesis will then conclude with a recap of what was covered and an explanation of the many possible extensions and applications of the work covered. References follow.

Chapter 2

Previous Work

The work presented in this thesis was originally intended for use in the Real-Time Strategy (RTS) commercial game genre. This type of game puts the player in the place of a general waging a battle. Objectives include gathering and managing resources, building and defending one or more bases, constructing vehicles and training troops for attacking the forces of another army or accomplishing some other objective.

RTS games pose many challenges described in more detail in [10]. Of primary interest here, however, is that in addition to the ordinary constraints of a commercial game to perform computations many times per second, there are many objects (buildings, vehicles, troops, etc.) to consider. The environment also tends to be large, with both wide open areas such as fields, and small details such as a jagged line forming the base of a cliff.

The above requirements simply reinforce the constraints under which a pathfinding system must run for such a game: it must find a path even faster than is normally required because several paths may be requested at a time, and the representation should handle both large areas and detailed regions efficiently.

The technique developed in this thesis also has other properties which could be useful in RTS games, described in Section 10.2. The majority of the work, however, is applicable to all pathfinding.

The goal of the techniques done here was to apply them in the Open Real-Time Strategy (ORTS) game engine [11]. This is an ideal test bed for algorithms such as pathfinders because it is an open source engine providing easy integration, is very general in its implementation, and is not susceptible to compromise by its clients, among other advantages [9].

2.1 General Search

A domain in which some search is to take place can have one or several attributes associated with it at any one time. An assignment of these attributes to particular values yields a *state*, which represents a situation in this domain at some instance, possibly resulting from a series of *actions*. Performing an action on some state usually results in some other state. All possible assignments of values to the attributes associated with a domain produces all possible states in that domain, also called a *state space*.

A search is performed to determine the series of actions which, when applied to a given initial state, or *start*, result in a desired final state, or *goal*. A sequence of intermediate states resulting from these actions is often another result which a search can yield. Either of these is referred to as a *path*. A search is done by performing each possible action on the start state, which yields a corresponding subsequent state for each action. If none of these

states produced is equivalent to the goal state, we then perform this set of actions on one of these resulting states, and continue in this manner.

There are several methods for performing this general task. One such example is breadth-first search. As indicated by the name, this algorithm first searches all states resulting from a single action performed on the start state before searching those which require two, and so on. This technique is usually implemented by means of a queue, and whenever the goal state is found, it is found by the shortest possible sequence of actions. A drawback of breadth-first search is that it is memory-intensive: by the time it reaches the goal state, all states that result from fewer actions than the goal are stored in memory. Obviously, as the state space becomes larger and the goal is farther from the start, the memory requirement of this approach can quickly surpass the capacity of a computer.

In such cases, a technique called depth-first search can be used. These will search into the space along one path at a time to find a solution. This type of search uses less memory, but take longer to complete, hence their use when sufficient memory is not available for breath-first techniques. However, often it was not desirable to search along each path as far as possible before trying another, since this could yield a path much longer than the shortest possible sequence. Also in many cases such a search is not even possible.

For these reasons, depth-first iterative-deepening (DFID) can be used instead, which expands each path a certain distance, and if no solution is found, it will search each path slightly further, and continue in this manner until a solution is found. While this technique will return the shortest path to the goal, understandably it requires even more time to complete than depth-first search due to the repeated search involved.

Originally, these search techniques were used mainly to traverse graphs, however eventually, larger state spaces began to be used, and more efficient techniques were required. The techniques above effectively traverse the search space but do not search it intelligently. In many cases knowledge about the structure of the search space could be applied to the benefit of the search.

Thus, an algorithm called A* [23] was devised to incorporate such knowledge to search the state space more efficiently. A* is based on a breadth-first search, but uses a *heuristic*, which is a measure of how close any state is to the goal of the search. Its approach is not only intuitive, but also has several properties desirable in many domains. These are explored in more detail in Section 5.1.

Because of this, A* has enjoyed widespread use for decades. Despite its efficiency, however, one problem with A* is that as state spaces grow even greater in size and domains become even more complex, the memory required by this approach stretches beyond that available in computers.

So similarly to how A* introduces heuristics into breadth-first search, IDA* [33] introduces them into DFID search. IDA* shares many of the same properties as A*, but also works in settings where sufficient memory is not available for techniques such as A*, at the cost of requiring more time because of repeated search. As with A* and breadth-first search, heuristics allow IDA* to proceed in a much more intelligent fashion than DFID search.

Other search techniques followed which allow the use of such heuristics to intelligently guide their progress, however those mentioned above are the most prominent.

2.2 General Abstractions

One question that might come to mind is the origin of the heuristics used by such algorithms as A* and IDA*. Often, they are the result of human knowledge of the domain being searched. In most cases, people develop explicit techniques to determine from a state in a given domain, approximately how far it is to the goal.

However, there is another, more general method to provide these values: abstraction. This involves producing from a given state space, a parallel space comprised of fewer states, known as an *abstract space*. For any state in the search, a corresponding state can be determined in this abstract space, where it is easier to get an estimate for the distance to the goal. The actual distance between this abstract state and that corresponding to the goal in the abstract space often forms a fairly accurate heuristic in the original space.

There are a number of ways to go about constructing an abstraction of the original space. The most common is to form a partition over the original state space where all states in a single partition correspond to the same resulting state in the abstract space. This creates a significant reduction in the state space and often allows the exact distance between the abstract states corresponding to the current state being searched and the goal, to be calculated exactly. Once again, there are numerous approaches to obtaining these values. Two such approaches are pattern databases [16] and Hierarchical A* [26].

Pattern databases associate each state in an abstract space with a value indicating the distance they are from the corresponding goal. These are populated by determining the state corresponding to the goal in the abstract space, and from this state, exhaustively visiting the entire abstract space using breadth-first search, and recording for each abstract state, the distance from the goal.

This way for each state in the original space, the corresponding state in the abstract space can be found and the distance of this state from the abstract goal determined immediately and used as a heuristic in the original space. This technique represents the end of the spectrum where all values are calculated during preprocessing, and is useful if a number of such searches are likely to be done.

Hierarchical A*, by contrast, calculates such values as the search is being performed. To find a heuristic for a state in the original space, the corresponding state is found in an abstract space. The distance between this abstract state and the goal in this space is determined by a search, and this value is used for the heuristic of the original state. A heuristic is also needed for the search in the abstract space, so each state here is mapped to one in an even more abstract space, where the distance between this state and the goal again becomes the heuristic for the state in the less abstract space.

This process continues searching more and more abstract spaces, until reaching a space so abstract that it contains only one state, or some suitably abstract space where a domain-dependent heuristic is used. These values are saved, and other useful figures are cached, so that subsequent searches can benefit from these calculations. Searches using this method take longer than those using a pattern database, but require no preprocessing, so it is more suited to situations where too few searches will be performed to warrant constructing a pattern database.

2.3 Pathfinding Search

In addition to the general search techniques that are commonly used for pathfinding tasks, there are many, more specialized searches which exploit the properties inherent to this problem. Indeed, pathfinding is so important to application areas such as commercial games and robotics, that there are methods which address not just pathfinding as a whole, but even subtasks therein. Unfortunately a complete treatise of pathfinding techniques is well beyond the scope of this thesis, but we will give a synopsis of a few important results below.

The problem of finding a shortest path in a plane is well-studied [39]. The optimal solution is given in [25], where execution time and memory usage is $O(n \log n)$, respective of the vertices in the environment. In most cases, a more practical approach is more suitable. Probably the most common type of solution uses a grid-based environment representation

[32], described further in Section 3.2.

However, other techniques such as visibility graphs [34] can be used as well. This involves connecting the vertices of the obstacles to each other so long as an unobstructed straight line can be formed between them. A search is then performed on this graph, which is guaranteed to contain an optimal path for an object of zero size. The drawbacks to this technique lie in the fact that the number of edges in the graph can be quadratic in the number of vertices on the obstacles, which is detrimental in respect to both time and memory, and that in the presence of a change in the environment, this graph can require extensive repairs, and finally that it only provides paths for objects of a single size.

In addition to such deterministic methods, there are random techniques that have been used to some degree of success. In higher-dimensional spaces, complete algorithms are sometimes unwieldy and so algorithms such as Rapidly-exploring Random Trees (RRTs) were developed [35]. Such trees quickly explore the space at random in an attempt, in this case, to find a path between a start and goal. While sometimes useful in the case of an unsearchably large or complex environment, these techniques are not guaranteed to find a path except in their limit, and typically do not find an optimal path. In the application areas for which the work of this thesis was targeted, both of these abilities are important and so such techniques are not suitable.

Nevertheless, it was a primary goal to produce a technique that functioned effectively in large and complex environments, so information specific to pathfinding in Euclidean space was required. In particular, the techniques presented in this thesis are based on the triangulation environment representation, or more precisely, the Constrained Delaunay Triangulation representation presented in [28]. Because this work is so central to this thesis, it is presented in more detail in Section 3.4. It should be noted that there are other popular polygonal representations available such as trapezoidal decomposition [30, 29], however the triangulation representation provided mechanisms for the abstraction procedure discussed in Chapter 6.

2.4 Pathfinding Abstractions

A work with which that in this thesis is compared is the PRA* algorithm presented in [47]. While there are many pathfinding techniques which make efficient use of abstractions such as HPA* [5], which decomposes an environment into "rooms" and caches the best paths between the entrance and exit points of each to construct a complete path between points quickly, PRA* is the most competitive algorithm at the time of writing.

PRA* (short for Partial-Refinement A*), performs pathfinding search in an abstracted representation of the environment and then converts the solution from this space to one on the original environment. It works by building layers of increasing abstraction, forming a layer by abstracting neighbouring states of a layer into a single state in the more abstract layer above. States at the top, most abstract level represent groups of states in the original space which are reachable from each other. For pathfinding, these states equate to areas in the environment, specifically, the cells in the grid, which is described further in Section 3.2. When a path is requested, the start and goal are projected up through layers of abstraction by determining at each layer, the state of the layer above which corresponds to that state. This is continued until either they become the same state in some abstract layer, indicating that a path exists between them, or they reach two distinct states at the most abstract level, indicating such a path is impossible. This is similar to checking the component indices in TRA* as described in Subsection 7.1.1.

If a path can be found, a suitably abstract layer is chosen on which to perform the search, for example the layer halfway between the original graph and that at which the start and

goal met in a single state. The search is performed at this layer, and once complete, the states forming the path are projected onto the layer below by determining which states on that layer correspond to those in the solution path on the layer above. Then another search is performed on this level, but only in the states corresponding to those in the solution on the level above. This is continued downward through less abstract layers until a path on the original environment results.

There are several advantages to this approach. The first and most obvious is the increased speed which results from the search being performed on an abstract layer with a much smaller state space than the original environment. Another advantage is that the existence of a path between any two points in the environment can be quickly determined instead of requiring a lengthy exhaustive search.

Finally, because a path is found on some abstract layer and the actual path must be within the corresponding states in the original environment, this "concrete" path need only be determined a portion at a time. For example, the actual path for the first few states in the solution path on the abstract layer need only be determined for the object to start along a path which we know will lead to a goal. This is useful in commercial games where a delay before an object starts moving toward the desired location is should be avoided. Subsequent portions of the path can then be determined at any time while the object moves along the portion of the path already known.

Also, as seen in Section 9.2, the presence of the abstraction also makes this algorithm less susceptible to increases in execution time as the distance between the start and goal increases, and also more predictable. This is because the actual distance between the start and goal can be predicted by the height of the layer at which they meet at the same state. Thus, the resulting search is performed on a suitably abstract layer, meaning that both complex and long paths are dealt with on a more abstract layer, reducing their negative impact on execution times.

A disadvantage of this method is that creating the abstraction of the environment in such a manner inherently loses details therein. Therefore, as long as the search is performed on any abstracted layer, this method cannot guarantee the shortest path will be found. Obviously if the initial search is done on a very abstract level, the resulting suboptimality will be more likely and more pronounced than if it was performed on one which is closer to the original environment. Luckily, performing the search on the layer at half the height of that at which the start and goal become the same abstract state, results in both a very efficient search, and a path which, with a high degree of probability, is very close to optimal.

Chapter 3

Environment and Representation

The first question that one must ask when approaching a pathfinding problem is that of how to represent the environment. This decision can depend on a number of factors such as the nature of the environment and the pathfinding techniques that are applicable to each representation.

In this section, we will discuss such concerns and give advantages and disadvantages of a couple of the most common environment representations. The method selected for use in this thesis will be described and justified.

3.1 Environment Description

In essence, an environment in pathfinding dictates which configurations between which an object can and cannot transition in a single motion or time step. Typically, this is given in terms of obstacles, with which the object must not intersect at any time, either on a time step or in between them.

While this is the most common case in pathfinding and indeed in commercial games, it is worthy of note that more complex situations are possible, such as ledges that the object can move down but not up. Technically, situations such as "teleportation" where an object moves farther than is normally possible in a period of time, are possible in this framework. Indeed, such cases exist in some commercial games.

Such possibilities, however, can complicate the pathfinding task and distract from its fundamental properties. We therefore adopt the convention that an object can move at a constant rate in any direction of its current position, providing that neither the final configuration from this motion, nor any intermediate configuration results in the object overlapping with an obstacle in the environment (or another object).

To further simplify matters, this thesis will consider all objects to be radially symmetric (circles in 2 dimensions). This implies that if an object in a particular configuration does not overlap an obstacle, it cannot be made to through rotation. Similarly, an object intersecting an obstacle always will unless its position is changed. Thus, an object's configuration is simply considered to be its position for our purposes. From this point on in the thesis, the terms *configuration* and *position* are used interchangeably.

3.2 Grid-World Representation

The most common environment representation used for pathfinding is known as the *grid-world* representation. This is where a grid (most often, of squares) is overlaid on the environment and each cell of the grid is considered either *traversable* if no part of an obstacle overlaps the cell, or *obstructed* otherwise. Figure 3.1 shows an environment which has some obstacles. Its grid-world representation is shown in Figure 3.2, with the obstructed cells shaded and traversable cells left white.

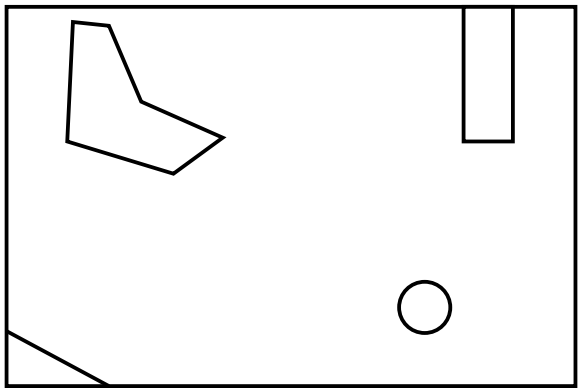

Figure 3.1: Environment for representation example

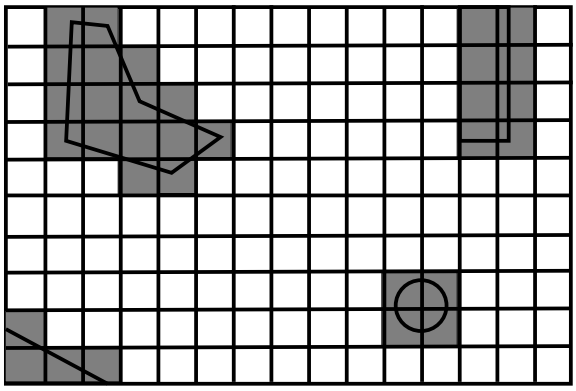

Figure 3.2: Grid-world representation of environment

The position of the object for which pathfinding is to be done is discretized to a (traversable) cell, and each possible move for the object results in its position changing to one of that cell's neighbours (either 4- or 8-neighbours in the case of square cells) which are traversable. Figure 3.3 shows possible moves for an object (depicted by a circle) into traversable 4-neighbours (solid arrows) and 8-neighbours (solid and dotted arrows).

This representation is the most common in commercial games as it is easy to define an environment as a number of cells on a grid, and because many pathfinding techniques are made for this representation. A disadvantage of this is that if obstacles such as walls are not axis-aligned, this results in imprecise representation. An example of this is shown in Figure 3.4.

If the resolution is not sufficient, this imprecision can lead to the pathfinding algorithm finding suboptimal paths or none at all. When the grid has sufficient resolution, especially

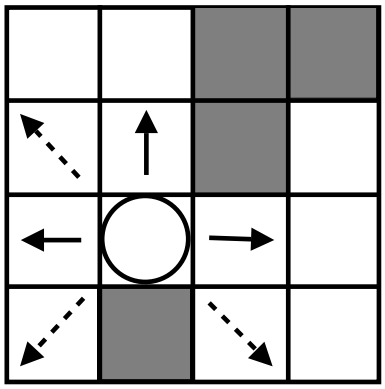

Figure 3.3: Possible moves for an object in a grid world

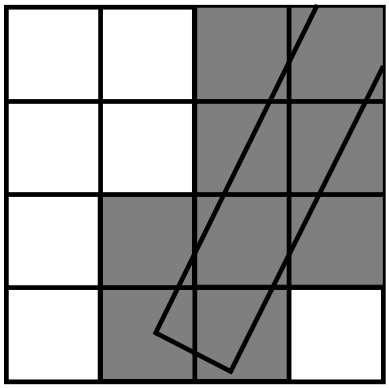

Figure 3.4: Imprecise representation of a non-axis-aligned obstacle

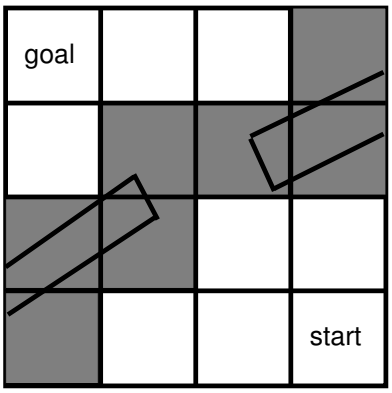

Figure 3.5: No path due to insufficient grid resolution

in this situation, the number of cells can severely complicate the pathfinding process, multiplying the number of moves required for a path. In Figure 3.5, the imprecision causes no path to be found between the start and goal cells. The resolution of the grid must be increased to reduce this imprecision to the point where such a path can be found, as in Figure 3.6.

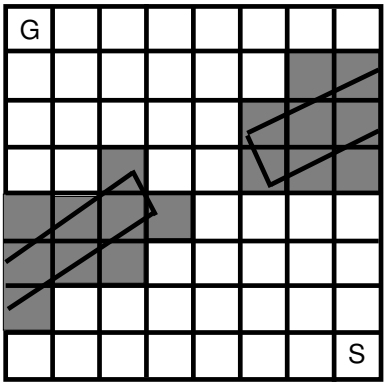

Figure 3.6: Grid resolution increased to produce path

Similarly, the number of cells in an area is dependent on its size; a large area has a greater number of cells than a small one, even if they are geometrically similar. For these reasons, grid-world-represented environments, particularly more accurate or non-axis-aligned ones, tend to have many more cells than other representations.

3.3 Triangulation Representations

A slightly less common representation is the *constrained triangulation*, a variation of which is used in the work in this thesis. A constrained triangulation represents the borders around the obstacles in the environment as line segments or *constrained edges*. *Unconstrained* edges are then added between the end points (and points of intersection) of these constrained edges, without such edges crossing, until no more such edges can be added, at which point the environment is made up entirely of triangles.

Figure 3.7 shows the same environment as in Figure 3.1, but with curves approximated by line segments (the reason for which is explained below). Figure 3.8 shows that environment represented by a constrained triangulation with solid lines for constrained edges and dotted lines for unconstrained edges (this is the convention adopted in diagrams of constrained triangulations from this point forward). Traversable and obstructed spaces in the environment are implicitly defined in that objects always originate in traversable space and cannot move across constrained edges into obstructed space.

Triangulations have a number of advantages compared to the grid-world representation, in that they can precisely represent any environment which contains straight obstacles (axis-aligned or not), the number of triangles in any area is determined by the properties of that area and not its size, and the necessity to include more detail in any one area does not increase the number of triangles in others.

To illustrate, consider adding an obstacle to a grid-world-represented environment which is smaller than any of the grid cells. In this case, either the obstacle would have to be imprecisely represented, or the entire grid would have to be made up of more, smaller cells. If a triangulation was used, this obstacle could be precisely represented (provided its shape

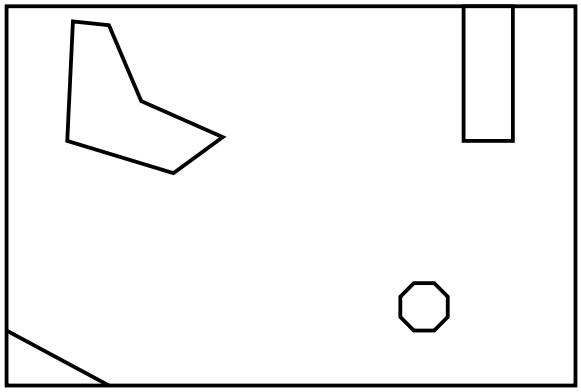

Figure 3.7: Same environment with curves approximated by line segments

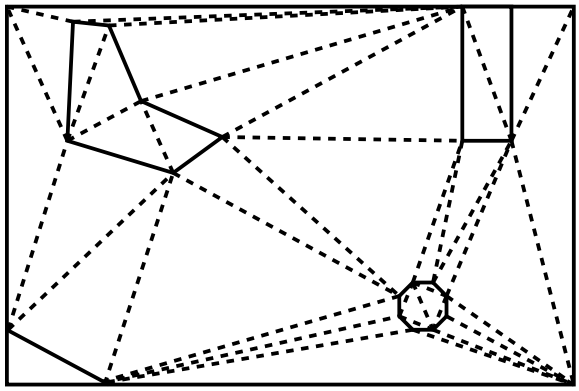

Figure 3.8: Environment represented by a constrained triangulation

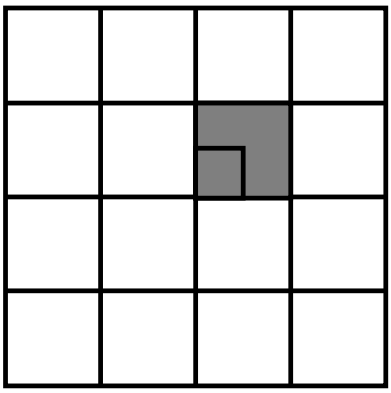

Figure 3.9: Small obstacle imprecisely represented in a low-resolution grid

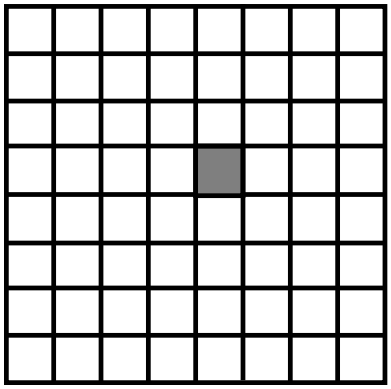

Figure 3.10: Grid resolution increased to better approximate small obstacle

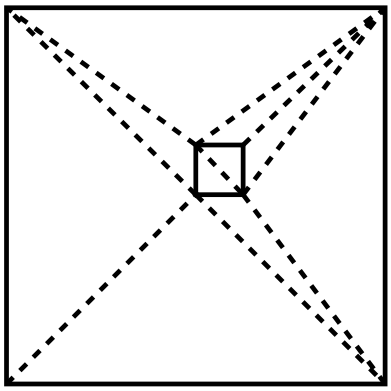

Figure 3.11: Small obstacle added to (empty) triangulation

is polygonal) while only creating smaller triangles in the vicinity of the obstacle, avoiding modifications to and an increase in the number of triangles in other areas.

Figure 3.9 shows such an obstacle being added to a grid and imprecisely represented due to the grid's relatively low resolution. In Figure 3.10, the obstacle is more accurately represented at the cost of increasing the resolution of the grid everywhere. This same obstacle is shown added to an otherwise empty triangulation in Figure 3.11.

When borders of obstacles in a triangulated environment are curved, they can be approximated by line segments, using more for increased accuracy. This is why the circle from the environment in Figure 3.1 is approximated by an octagon in Figure 3.7 prior to triangulation. This only increases the number of triangles in the area of the curve.

For the above reasons, there are generally much fewer triangles in a triangulated polygonal environment than there are cells in a grid-world representation with any reasonable resolution. Also, there are a number of existing algorithms which use triangulations and their inherent simplicity is helpful in the abstraction discussed in Chapter 6.

3.4 Dynamic Constrained Delaunay Triangulations

The particular implementation of a triangulation used in this work is the Dynamic Constrained Delaunay Triangulation (DCDT) developed by Marcelo Kallmann [28]. Below we will describe different types of triangulations to familiarize the reader before describing this

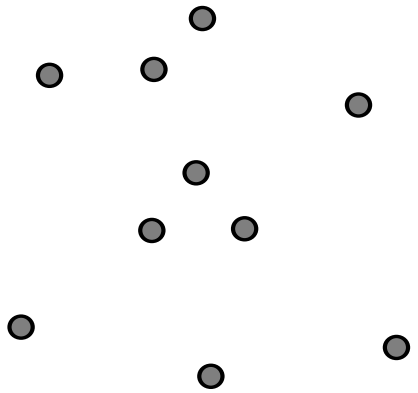

Figure 3.12: A collection of points to be triangulated

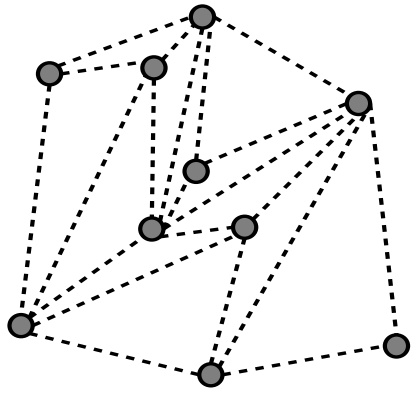

Figure 3.13: A triangulation of this set of points

work. A more complete treatment of these and other geometric structures is given in [41].

A basic triangulation takes a collection of points like that in Figure 3.12, between pairs of which edges are added such that these edges do not cross, until no more such edges can be added. At this point, within the convex hull of these points, all areas are triangular, as illustrated in Figure 3.13. A *Delaunay* Triangulation (DT) of these points adds the edges in such a way that the minimum interior angle of all triangles in the triangulation, is maximized. This is equivalent to other requirements described further in [28]. This property implies that these triangulations tend to avoid "sliver" triangles wherever possible, which is beneficial to the triangulation as a whole. An efficient algorithm for the computation of a Delaunay Triangulation is given in [1]. As an example, compare the Delaunay Triangulation in Figure 3.14 to the regular triangulation of the same points in Figure 3.13.

As described earlier, pathfinding in triangulations is normally done using some form of Constrained Triangulation (CT), where constrained edges represent borders between traversable and obstructed space. Often the convex hull is made using constrained edges to specify that objects are not to exit the triangulated area. For instance in many games this area might be a rectangle. A rectangular area enclosed by constrained edges containing barrier segments is shown in Figure 3.15, and a Constrained Triangulation of these is shown in Figure 3.16. Specifying that a Constrained Triangulation be *Delaunay* (now Constrained Delaunay Triangulation or CDT) again adds the requirement that the unconstrained edges be added in a way maximizing the minimum internal angle of any triangle

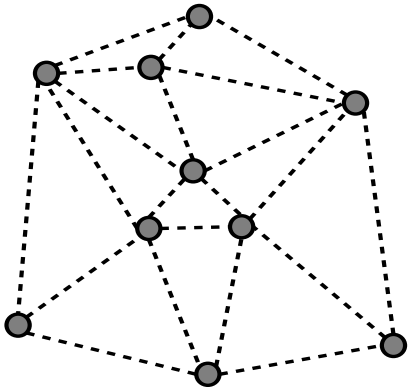

Figure 3.14: A Delaunay Triangulation of the same set of points

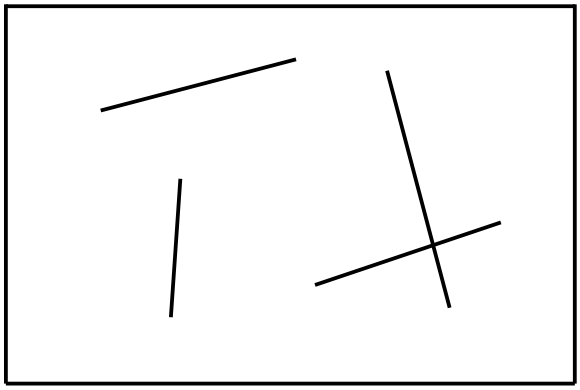

Figure 3.15: A collection of segments including a bounding box

in the triangulation, as long as none of the constrained edges are affected. A Constrained Delaunay Triangulation of this same environment is shown in Figure 3.17; again, compare the unconstrained edges to those in Figure 3.16.

A Constrained Triangulation with the Delaunay property better represents the structure of the environment, especially for abstraction, as shown in Chapter 6. More specifically, however, this property is used to ensure that whenever a valid path exists between two points in the triangulation, one exists such that it does not pass through any triangle more than once, which is no longer. The corresponding proof is given in Chapter 4. See [13] for an optimal $O(n \log n)$ algorithm for computing a Constrained Delaunay Triangulation, and [17] for one that works on-line.

The further specification that the triangulation is *dynamic* simply implies that in the presence of a change in the triangulation (one or more constraints is added, deleted, or moved), the triangulation can be repaired locally [28]. While this is not a requirement for the research in this thesis, it is advantageous in the application areas.

For example, in an RTS game, it is often the case that the user will encounter previously unknown terrain. In terms of the triangulation, this will result in constraints being added (or removed). In order to perform pathfinding properly given this new information, these constraints will have to be incorporated into the triangulation.

Obviously in the presence of the strict constraints of a commercial game as discussed earlier, it would be more beneficial to only modify the triangulation in the region of the change

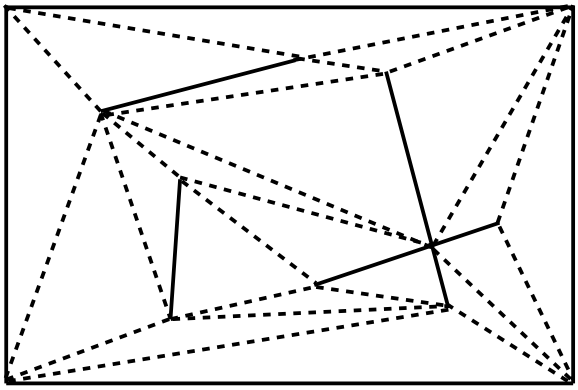

Figure 3.16: A Constrained Triangulation on this collection of segments

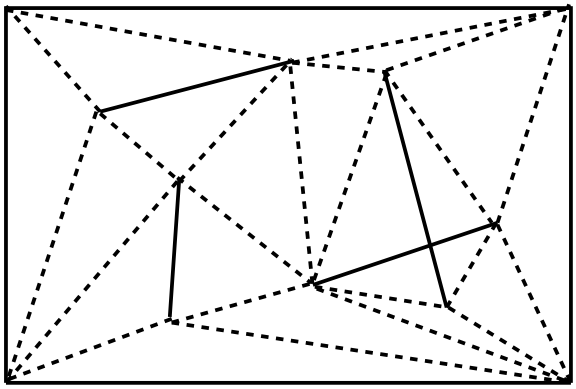

Figure 3.17: A Constrained Delaunay Triangulation on the same collection of segments

than to completely rebuild it. Kallmann's DCDT provides a mechanism for modifying these constraints with minimum changes to the triangulation.

Changes to the triangulation consist of the addition and removal of vertices and constrained edges; changes in location are done by removing the affected constraints and subsequently adding them in their new position. Since for pathfinding, we consider vertices to be obstacles, for the most part, these are added only as endpoints and intersection points of constrained edges, however, they could form "light post"-like obstacles.

Adding a vertex is done by first locating it within the triangulation (this point location process is described in further detail in Section 8.1), to see if it lies on another vertex, an edge, or in a triangle face. If the vertex is incident with another, it does not need to be added. If it lies on an edge, this edge is split into two edges consisting of the original endpoints of the edge each leading to the vertex just added. Then the triangles incident with the original edge are split by adding unconstrained edges between the new vertex and the vertex of each triangle opposite the original edge. This situation is shown in Figure 3.18.

If the edge lies in a triangle face, unconstrained edges are added between the vertices of this triangle and the newly added vertex. Figure 3.19 shows a vertex being added inside a triangle face. When a vertex is added to either an edge or a face, the triangles surrounding this are may have lost the Delaunay property.

Therefore we must check the edges surrounding the triangles involved with the insertion

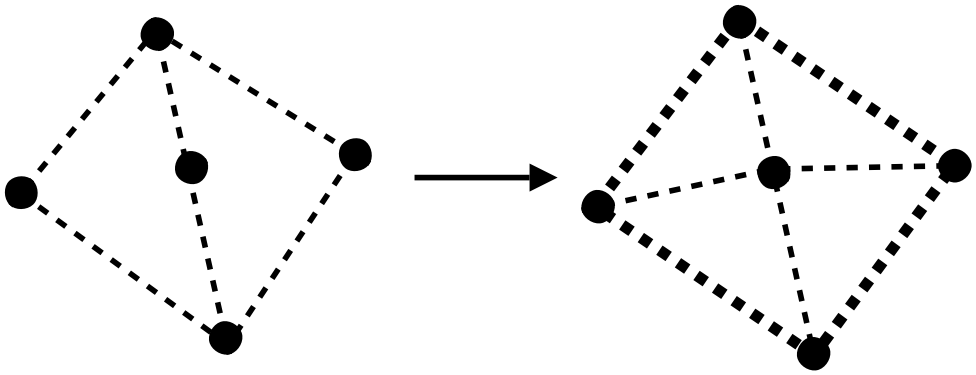

Figure 3.18: Adding a vertex on an existing edge

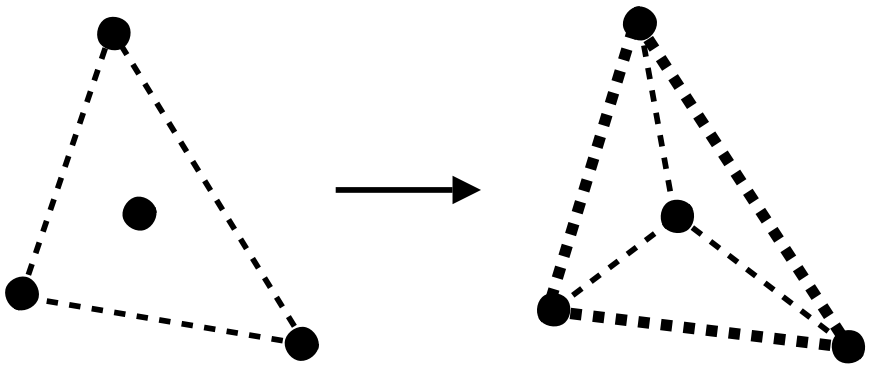

Figure 3.19: Adding a vertex in a triangle face

(shown in bold on the right side of both figures) to see if they need to be "flipped". We check if the triangles forming the quadrilateral of which the edge forms the diagonal has lost the Delaunay property. If so, the diagonal is instead made to join the other pair of opposite vertices in this quadrilateral, restoring this property in the region.

However, doing this might cause other triangles to lose this property, and so the other two edges of the original triangle not in the direction of the inserted point, are similarly checked. In this way, the area affected by the insertion of the vertex expands as a star-shaped polygon from this region. Although in the worst case, this could lead to flipping every edge in the triangulation, the expected number of flips, no matter how the vertices are distributed in the triangulation, is constant [21].

Inserting a constraint is done in several steps. A constraint often represents a single obstacle, and could be comprised of several line segments, for example. Each such segment is added as described below, and can correspond to multiple constrained edges in the resulting triangulation. The process for adding a segment of a constraint will be illustrated here using an example similar to that in [28]. First, point location is performed to find both endpoints of the segment. These endpoints are added as vertices to the triangulation. Figure 3.20 shows the triangulation after this step; the solid horizontal line is the segment being inserted.

Next, the intersection points between the new segment and existing constrained edges are calculated and inserted. For each intersection point with a constrained edge, the constrained edge and the new segment are split at the intersection point into two edges or segments each.

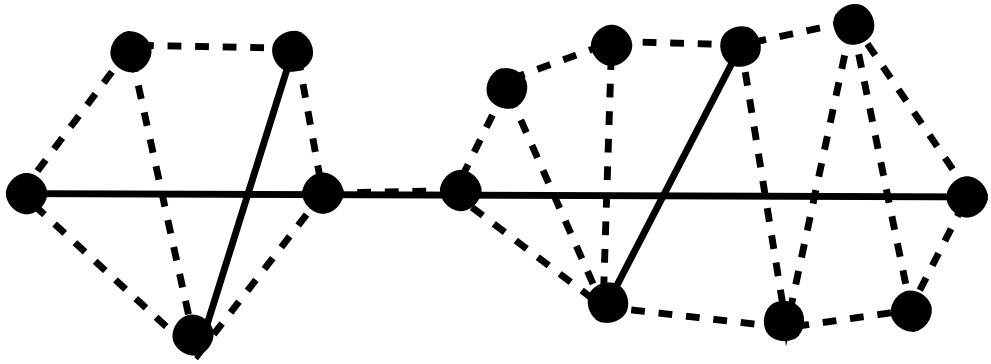

Figure 3.20: A segment being inserted into the triangulation

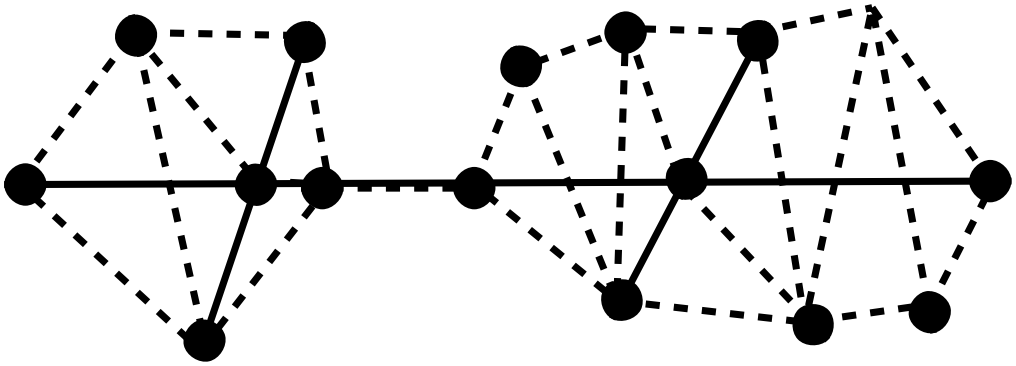

Figure 3.21: Intersection points between the new segment and constrained edges are inserted

This step is shown in Figure 3.21. After this, all unconstrained edges which cross the new constraint are calculated, and removed from the triangulation, as shown in Figure 3.22.

The final step is the insertion of the segments of this new constraint as constrained edges. Note that such a segment intersecting with another constraint on a vertex also splits the segment into smaller segments, but the other constraint does not change. If a segment of this constraint overlaps an existing edge in the triangulation, this constraint is simply added to the edge. In this way, an unconstrained edge can become a constrained edge, or a constrained edge could represent multiple constraints. Finally, the non-triangular regions left by the previous step are triangulated, yielding the final result shown in Figure 3.23.

Removing a particular vertex from the triangulation is done by first removing all edges for which the given vertex is an endpoint, and then the vertex itself. This leaves a non-triangular area around where the vertex was removed, which must be triangulated. It is assumed there are no constrained edges for which this vertex is an endpoint, and that the vertex itself is not a constraint, otherwise, such a procedure would not be allowed.

Removal of a constraint is done by first locating the constrained edges which correspond to this constraint. Once those are located, this constraint is removed from these edges. Each constrained edge for which this represents the only associated constraint, becomes unconstrained.

Finally, the endpoints of all edges which formerly comprised this constraint are checked for whether they form an endpoint for any constrained edges, or themselves represent a

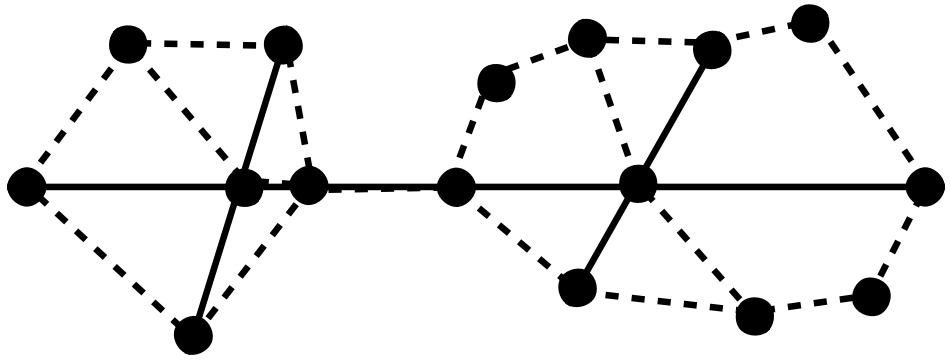

Figure 3.22: Unconstrained edges crossing the new segment are removed

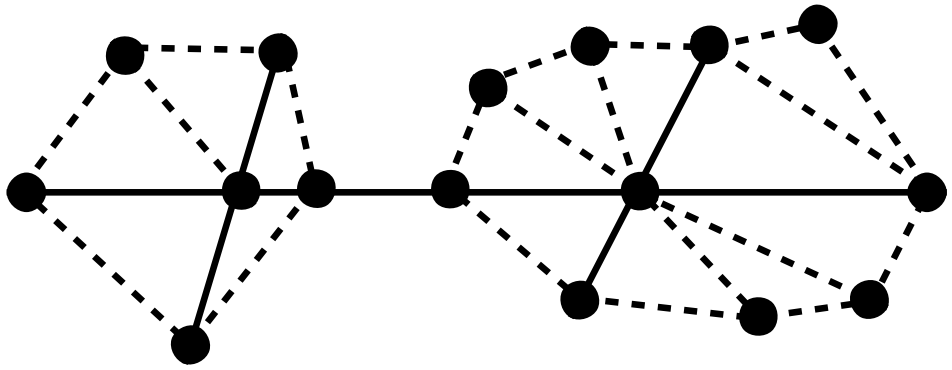

Figure 3.23: Final triangulation after new segment has been inserted

constraint. The vertices that do not are removed from the triangulation using the procedure above. Moreover, if a vertex which formed the intersection point between the removed constraint and another constraint (as evidenced by exactly two constrained edges incident with this vertex, representing the same constraint and being colinear), this vertex and the incident constrained edges are removed from the triangulation and then a single segment is inserted to replace the two smaller segments that were removed.

Chapter 4

Nonpoint Objects

In this section, we explore the implications of performing pathfinding for objects larger than the simple point object case. As described earlier, we consider objects to be radially symmetric so as to remove the orientation component of the configuration and are left only with position (in two dimensions).

There are a number of ways to approach this problem, the most common one being the Minkowski Sum operation. This consists of adding every element of the object's shape to every element of the obstacles in the environment. This results in the obstacles in the environment being "grown" so that the object can be treated as a point object and pathfinding done in this environment [36], then when the object follows this path in the original environment, it will not collide with the obstacles.

Figure 4.1 shows an environment and nonpoint object and Figure 4.2 shows that same environment with the object's shape added to the obstacles via the Minkowski Sum operation. Figure 4.3 shows a path found for a point object in this environment, and finally, Figure 4.4 shows the path for the nonpoint object in the original environment.

There are, however, several disadvantages to the use of Minkowski Sum for pathfinding. The first is that there would have to be separate representations of the environment corresponding to each size of object, all of which would require time to calculate and memory to store. Also, we wish to perform pathfinding for objects of arbitrary size, and using this approach would either necessitate performing this calculation online—which may debilitating in a real-time setting—or using a precalculated environment representation for an object of

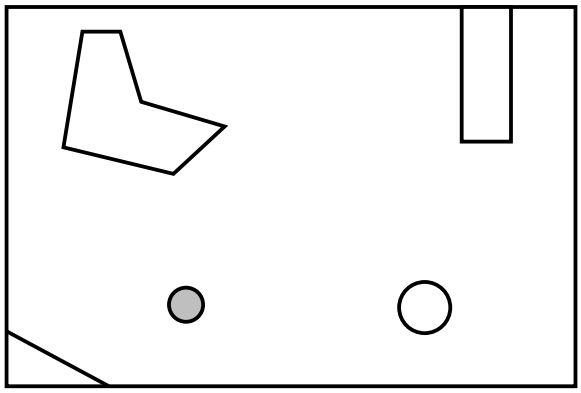

Figure 4.1: Nonpoint object in original environment

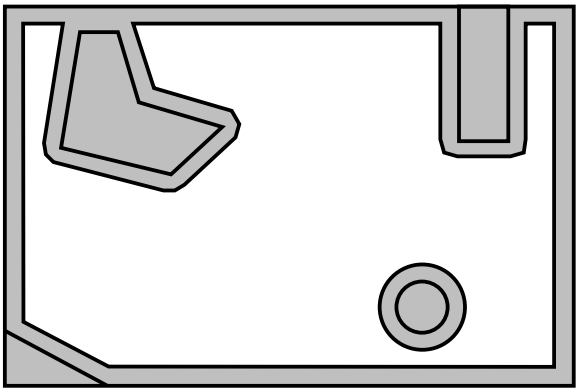

Figure 4.2: Minkowski Sum of object on environment

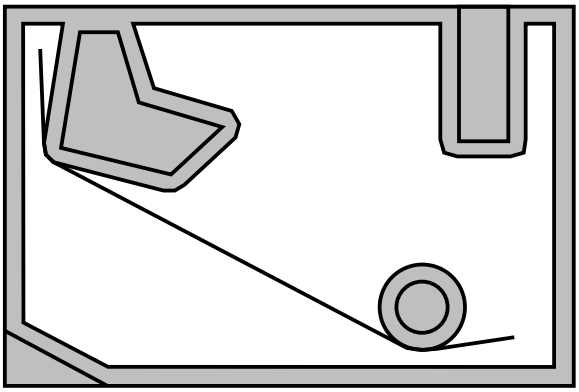

Figure 4.3: Path for point object among "grown" obstacles

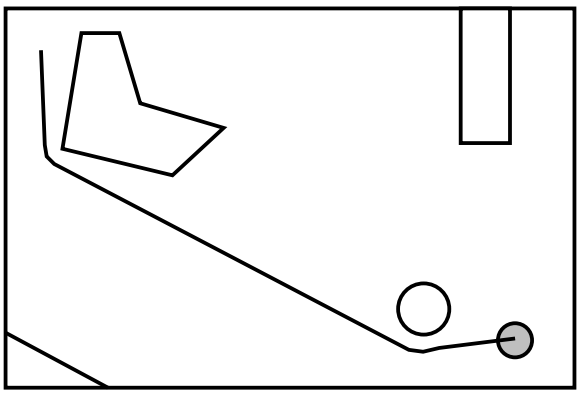

Figure 4.4: Same path for nonpoint object in original environment

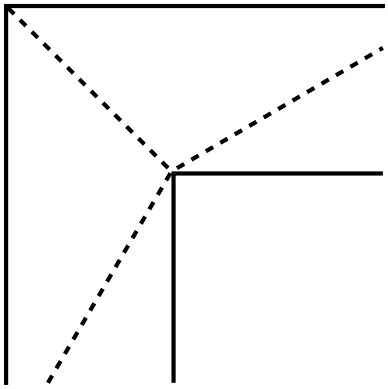

Figure 4.5: Part of a Constrained Triangulation

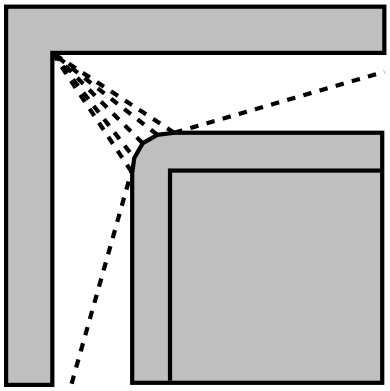

Figure 4.6: Obstacles grown by Minkowski sum for a circular object, approximated by a regular polygon

different size (risking finding invalid paths if the object is larger than that for which the representation was constructed, and risking missing valid paths if the object is smaller).

Finally, because we are dealing with the situation involving radially symmetric objects, representation of an environment grown with a Minkowski Sum becomes overly complex. As discussed in Section 3.2, a grid representation of such circular detail would either be inaccurate or require many cells, which increases the memory required to store the environment, and slows the pathfinding process.

Triangulations are negatively affected by this operation as well. Refer to Figure 4.5 for part of a triangulation of an environment. A circular object is approximated by a regular polygon and added via Minkowski Sum to the obstacles resulting in the environment shown in Figure 4.5. Again, there is a trade-off between accuracy and complexity in representation, as adding more sides to the regular polygon to better approximate the object results in more triangles in the resulting environment. The method of growing obstacles used in the work on which this thesis is based [27] is presented in [37]. For further details regarding Minkowski sums, such as how they are calculated and their properties, see [2, 22].

For these reasons we seek to avoid such an operation and instead wish to simply use the original representation of the environment in order to calculate both the portions of the environment that can be traversed by an object of arbitrary size, as well as a path for that object which avoids obstacles and travels through a sequence of these areas. Using

triangulations, both of these can be exactly determined for circular objects.

We will begin with a description of a method used to calculate the maximum possible size an object can be and still pass through a particular triangle in Section 4.1. Then we will prove that this indeed provides the correct value in Section 4.2. Next, we discuss a technique for determining the shortest path adhering to a sequence of triangles found through which the desired object can fit. An existing algorithm for this problem using point objects is shown in Section 4.4, and the modified version for use with nonzero-radius circular objects is given in Section 4.5.

4.1 Width Calculation

As described earlier, we desire to find the diameter of the largest circular object that can move between two (unconstrained) edges of a triangle in a Constrained (Delaunay) Triangulation, for example, between edges a and b in Figure 4.7 (Subsection 4.2.1 describes more formally what is meant by being able to move between two edges). Luckily this is equivalent to finding the closest obstacle to the vertex joining these two edges (vertex C in the diagram) in the region extending between the edges as shown. An obstacle, in this respect, can be a vertex or a point on a constrained edge.

There are three cases possible within a triangle which can determine the closest obstacle in this region. The first such case is that either $\angle CAB$ or $\angle CBA$ is a right angle or obtuse (Subsection 4.1.1), the second arises when these angles are acute and edge c is constrained (Subsection 4.1.2), and finally the last possibility is when $\angle CAB$ and $\angle CBA$ are acute and edge c is unconstrained (Subsection 4.1.3).

In each case, the path for the object of maximum diameter is determined as an arc hugging vertex C. While the object need not always follow this path to traverse the triangle, it is true that an object could not successfully traverse some other path and not this one, as is proven in Theorem 4.2.11.

In these proofs, we assume circular objects, but this could be extended to other shapes as well. The maximum allowable size of a circular object through a series of adjacent triangles could be used to determine the throughput of smaller objects, or the maximum allowable size of a rectangular object, for example.

The techniques used assume that at the very least, each vertex in a triangulation represents a constraint. That is, if one was representing an environment, one would only add a vertex to the triangulation either because it was an endpoint for a constrained edge, or

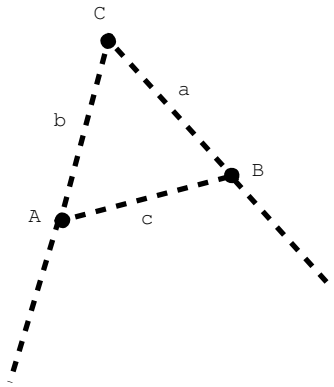

Figure 4.7: Part of a triangulation

if it were a point obstacle. This is intuitive because adding unneeded vertices would only complicate the triangulation and slow down subsequent algorithms. If such vertices were added, however, these methods might incorrectly determine the maximum diameter of an object through a triangle in the case where a path for the true largest possible object would pass through such an unconstrained vertex.

A brief discussion on the complexity of this algorithm is given in Subsection 4.1.4, and the proofs that this technique is equivalent to finding the maximum radius of an object with a valid path through this triangle are given in Section 4.2.

4.1.1 Case 1: Angle *CAB* or angle *CBA* is Right or Obtuse

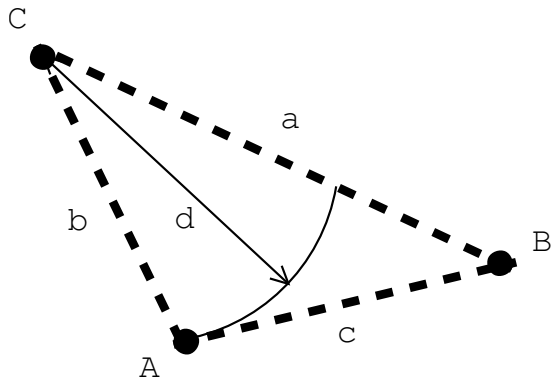

Figure 4.8: Case 1: angle at vertex A is obtuse

The first, and simplest case to consider is that which occurs when either $\angle CAB$ or $\angle CBA$ is right or obtuse. Assume, without loss of generality, that $\angle CAB$ is right or obtuse. It follows that edge b is shorter than edge a. Thus, the maximum allowable diameter d of a circular object between edges a and b in this triangle is the length of edge b. See Figure 4.8 for a visual explanation. This follows from Lemma 4.1.1 below.

Lemma 4.1.1 *The closest point on a line to another point is where a line passing through the point intersects the line at a right angle.*

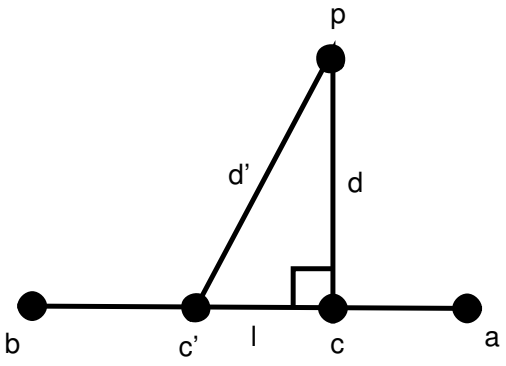

Figure 4.9: The closest distance of a line to a point

Proof In Figure 4.9, the point c is the point where a line passing through point p intersects with line ab at a right angle. For any other point c' on the line ab, it will be some positive distance l away from c. Thus, if the distance from p to c is d, then the distance (d') from p to c' is $\sqrt{d^2 + l^2} > d$. Thus, c is the closest point to p on line ab, as desired. ∎

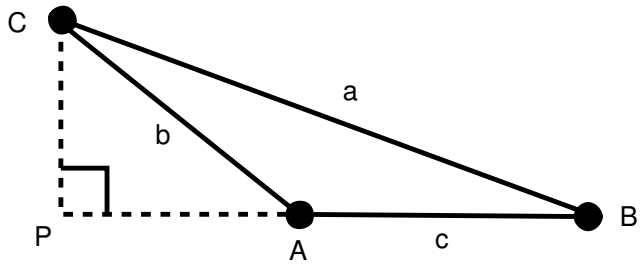

Figure 4.10: Triangle with one obtuse angle

Similarly, consider Figure 4.10. The length of segment b is $\sqrt{|CP|^2 + |PA|^2}$. For any point $A' \neq A$ along the segment between vertex A and vertex B, the length of the segment between C and A' would be $\sqrt{|CP|^2 + (|PA| + |AA'|)^2} > \sqrt{|CP|^2 + |PA|^2}$ since $|AA'| > 0$. Thus, vertex A is the closest point on segment AB to vertex C, and since there can be no obstacles in this triangle (any obstacles would have been incorporated in the triangulation), it follows that the closest obstacle is $|b|$ away from vertex C and this is the maximum diameter of an object that can move between edges a and b in this triangle.

4.1.2 Case 2: Edge *c* is Constrained

In the case that both $\angle CAB$ and $\angle CBA$ are acute, the point on the line passing through the vertices A and B that is closest to vertex C lies between A and B as shown in Subsection 4.1.1 above. In the case that edge c is constrained, this point is an obstacle. This situation is shown in Figure 4.11.

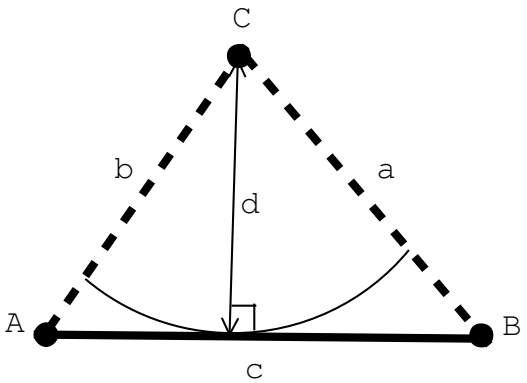

Figure 4.11: Case 2: angles at vertices A and B are acute and edge c is constrained

As described in Subsection 4.1.1, since there can be no obstacles within the triangle, the closest point on edge c to vertex C (when edge c is constrained) represents the closest obstacle to vertex C in the triangle. Assuming the distance between vertex C and the point P on edge c which makes the segment CP perpendicular to c, is d, the diameter of the largest circular object that can traverse the triangle from edge a to edge b is d, as desired.

4.1.3 Case 3: Edge *c* is Unconstrained

In the case where edge c is not constrained and both $\angle CAB$ and $\angle CBA$ are acute, the situation gets slightly more complex, as the closest point on edge c to vertex C no longer represents an obstacle.

Vertices A and B are still obstacles, and thus the maximum object diameter that can traverse this triangle from edge a to edge b is bounded above by both $|a|$ and $|b|$. Figure 4.12 shows a case where the shorter of edges a and b is the distance to the closest obstacle. However, since there may still be obstacles on the opposite side of edge c from vertex C closer to C than either A or B, we must consider these possibilities.

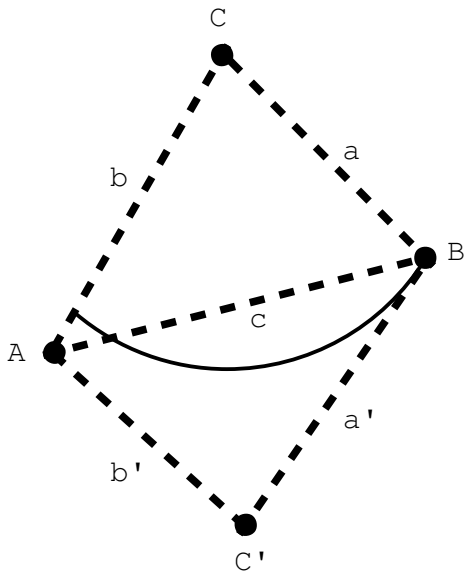

Figure 4.12: Vertex B is the closest obstacle to vertex C

What must occur then is a search that is bounded by the closest obstacle found so far. Thus, this search begins by searching across edge c to the triangle opposite this edge, bounded above by $min\{|a|, |b|\}$ and continues as described below.

When the search enters a triangle via an edge, it checks the other two edges as follows. We will say each edge is the segment between two vertices U and V. First of all, an edge will only be considered if both angles $\angle CVU$ and $\angle CUV$ are acute, that is, if the closest point on the line passing through U and V to vertex C lies between these two points. If this criterion is not met, search along this sequence of edges ends.

Figure 4.13 shows how considering a segment that does not fit this requirement could incorrectly determine the closest obstacle to vertex C. Here, if edge b' were considered an obstacle, the distance from vertex C to an obstacle would be incorrectly considered to be its distance from the dotted arc. However, there is no obstacle at this distance, since b' does not extend past vertex A, and further it would not be in the area between edges a and b.

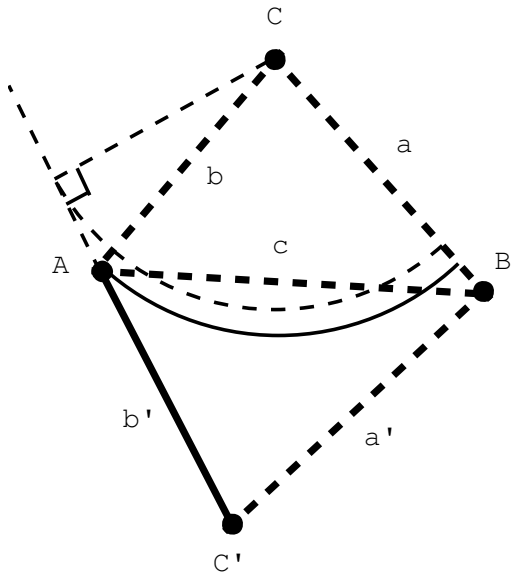

Figure 4.13: Edge b' should not be considered because $C'AC$ is obtuse

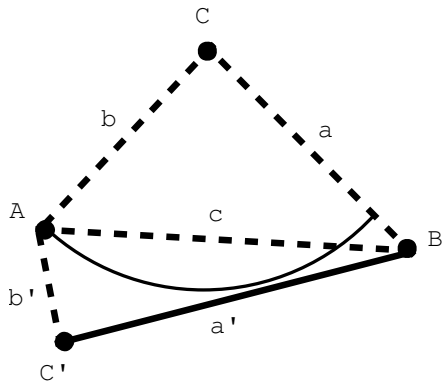

Figure 4.14: Edge a is farther from vertex C than A, so it is not considered

The actual closest obstacle to vertex C in this region is vertex A, and this distance is shown by the solid arc.

Next, we consider the distance from vertex C to the closest point on segment UV. If this distance is greater than the current upper bound, search across this series of edges returns because further search will not yield a closer obstacle than the closest already found. This situation is depicted in Figure 4.14.

If this distance is less than the current upper bound and segment UV is constrained, then the current upper bound is updated to reflect this new distance, and search returns from series of edges. A case where a constrained edge determines the closest obstacle to vertex C is shown in Figure 4.15. Finally if the distance is less than the current upper bound and UV is unconstrained, search continues across this edge into the adjacent triangle.

Pseudocode for the algorithm for determination of the width of some triangle T when moving between two edges a and b is given in listings 1, 2, and 3. Next, we discuss the complexity of the algorithm.

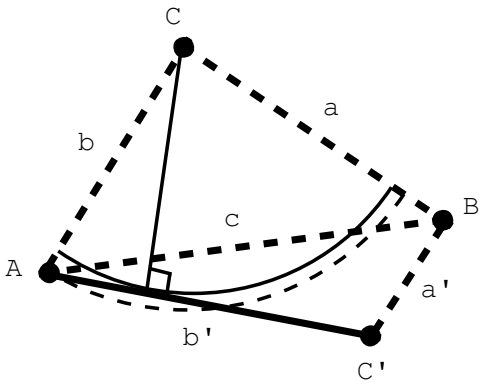

Figure 4.15: Edge b' becomes the closest obstacle to vertex C

Algorithm 1 DistanceBetween(Vertex C, Edge e) : Distance

```
1:  A , B ← EndpointsOf(e)
2:  if A_x = B_x then
3:      return |A_x − C_x|
4:  else
5:      rise ← B_y − A_y
6:      run ← B_x − A_x
7:      intercept ← A_y − (rise/run)A_x
8:      a ← rise
9:      b ← −run
10:     c ← run × intercept
11:     return |a·C_x + b·C_y + c| / √(a² + b²)
12: end if
```

Algorithm 2 SearchWidth(Vertex C, Triangle T, Edge e, Distance d) : Distance

```
1:  U, V ← EndpointsOf(e)
2:  if IsObtuse(C, U, V) ∨ IsObtuse(C, V, U) then
3:      return d
4:  end if
5:  d' ← DistanceBetween(C, e)
6:  if d' > d then
7:      return d
8:  else if IsConstrained(e) then
9:      return d'
10: else
11:     T' ← TriangleOpposite(T, e)
12:     e', e'' ← OtherEdges(T', e)
13:     d ← SearchWidth(C, T', e', d)
14:     return SearchWidth(C, T', e'', d)
15: end if
```

Algorithm 3 CalculateWidth(Triangle T, Edge a, Edge b) : Distance

```
1: C ← VertexBetween(a, b)
2: c ← EdgeOpposite(C, T)
3: A ← VertexOpposite(a, T)
4: B ← VertexOpposite(b, T)
5: d ← min{Length(a), Length(b)}
6: if IsObtuse(C, A, B) ∨ IsObtuse(C, B, A) then
7:    return d {Case: 1}
8: else if IsConstrained(c) then
9:    return DistanceBetween(C, c) {Case: 2}
10: else
11:    return SearchWidth(C, T, c, d) {Case: 3}
12: end if
```

4.1.4 Complexity

Of course, we desire to know that this algorithm's complexity will be reasonable if we wish to use it in certain domains. One can note that in the worst case, determining the width of a single triangle could require searching on order of all the triangles in the triangulation. Such a search could not be any worse, because this would require searching triangles multiple times when determining the width for a single triangle, which is not possible given our algorithm.

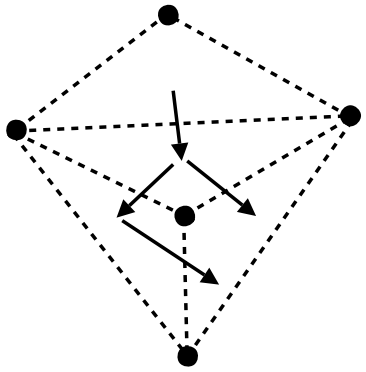

Figure 4.16: Search for the width of a triangle overlaps on a triangle

Refer to Figure 4.16. Here we show the search to find the width of the triangle at the top of the figure, using arrows. This search goes into the triangle at the bottom right from two different directions. However, one can note that entering this triangle from the left as shown is impossible using the technique presented because the angle created by this segment and the vertex at the top of the figure, is obtuse. Thus, this triangle would not be searched from this direction.

This is true in general, as shown in Figure 4.17. Here we see that search for the closest obstacle to vertex C could not enter triangle T through both edges f and d, because the exterior angles of a triangle are necessarily greater than π, meaning either $\angle DEC$ or $\angle FEC$ (or both) must be obtuse, so they would not be considered. If both these angles were acute, then edge e would be closest to vertex C, and search would enter triangle T through it.

While this shows the worst case for searching to determine the width of a single triangle, there are several reasons that in most cases, this value can be determined in much less time. First of all, both cases 1 and 2 require constant running time. Case 1 occurs quite frequently

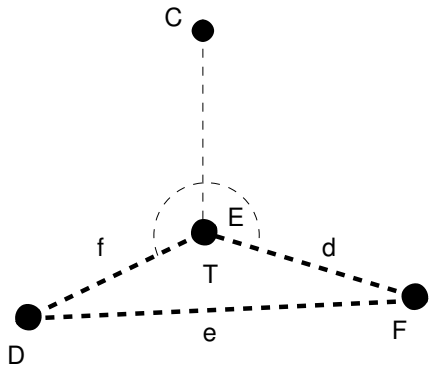

Figure 4.17: Proof that at most one exterior edge of a triangle can form two acute angles with a point outside that triangle

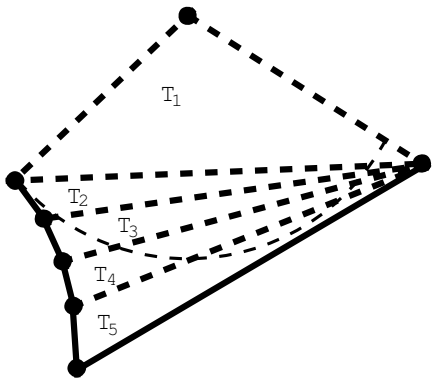

Figure 4.18: Searching across all triangles to find the width of triangle T_1

in triangulations, and when there are few or no point obstacles (triangulated vertices not representing endpoints of constrained edges) case 2 will occur even more often.

Also one can notice that unless $\angle ACB$ is close to π, case 3 cannot result in a very long search. In most cases, the bound of the search ($min\{|a|, |b|\}$) will not exceed the value at the start of the search (the distance of edge c from vertex C) by very much. For a search to traverse many triangles, particularly when this difference is small, the triangles across edge c would have to be very thin. This is uncommon to see in most triangulations and a very rare case in Delaunay Triangulations. Such a case is illustrated in Figure 4.18.

Although searching a triangle can conceivably expand search across two other edges resulting in an exponential search, we must remember that in the worst case, the search is still limited by the number of triangles in the triangulation and thus could not be worse than linear. Next one must note the conditions under which search could branch in such a way. These conditions are rare to find in a triangulation, and impossible in a Delaunay Triangulation, as proven later in Theorem 4.3.6.

As an example, observe Figure 4.19. In this case the algorithm for finding the width between edges a and b in triangle T will result in a search across edge c and into triangle T', and then across *both* edges a' and b'. However, in a Constrained *Delaunay* Triangulation, edge c would have been replaced by the edge shown in grey, and thus this effect would not have occurred.

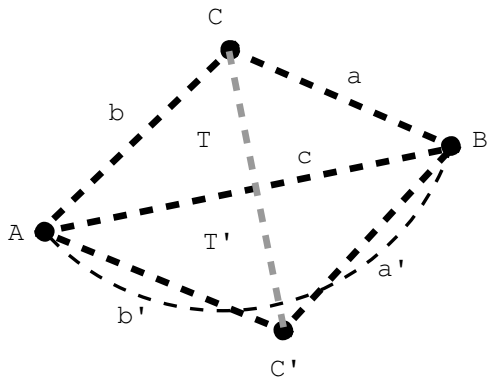

Figure 4.19: Finding the width of triangle T results in a branching search

One could also conceivably bound the search of all the triangles if the maximum object size is known *a priori*. If there are no objects with a diameter greater than a certain value, one could bound the searches of all the triangles by this distance. That way, search would not be performed on triangles that are wide enough to permit passage of all possible objects.

Finally, while a proof of the upper bound of this algorithm's running time for all triangles in a triangulation would be quite involved, we can make a number of observations to see that it will not be unmanageable. Combining the above considerations affecting the length of the search to determine the width of a single triangle, we see that when determining the width for one triangle results in a search across several others, the properties of the other triangles searched will are often such that they will almost or always fall under case 1 or 2 when their width is determined. For example, in Figure 4.18, finding the width of triangle T_1 results in a search through triangles T_2, T_3, T_4, and T_5. However, the width of all the other triangles can be determined without any search at all. Thus, the cost of finding the widths of all these triangles is linear in the number of triangles. Because of this, the complexity of running this algorithm on the entire triangulation is likely linear in the number of triangles and not quadratic.

4.2 Arc Paths

In this section, we show that the distance of the closest obstacle to vertex C between edges a and b is equivalent to the diameter of the largest circular object which can move between these edges. In Subsection 4.2.1, we define what it means for an object to perform such a motion, and go through some preliminary proofs that we will use in the following section.

In Subsection 4.2.2, we prove that whenever the closest obstacle between edges a and b is at distance d from vertex C then the a circular object of diameter d can move between these edges (Theorem 4.2.10) and that if there exists a circular object of a some diameter that can move between these edges, there will be no obstacles within this distance of vertex C between edges a and b, (Theorem 4.2.11). These combined prove the equivalence of the distance between vertex C and the closest obstacle between edges a and b and the diameter of the largest circular object that can move between these edges, thus that the width calculation presented in Section 4.1 correctly finds the diameter of the largest circular object that can move between these edges. Finally, Theorem 4.2.12 shows how to determine the motion for an object of given radius between these edges, which is shortest. This result will be used later on.

4.2.1 Definitions

The motion for an object takes place over time, and at any time the object is in exactly one place in two-dimensional space. Therefore, we define an object's motion to be a *curve* in mathematical terms. A curve is a function γ, which maps a parameter indicating its progress to a point in two-dimensional space, in particular, $\gamma : [0, 1] \mapsto \Re^2$.

A *path* is a curve which can be followed by an object. Because an object will have a certain speed and is unable to jump between locations suddenly, we require that a path be *continuous*.

For the purposes of this section, we need only consider the path of an object as it travels through a single triangle at a time. First, we will add the conditions required of such a path moving through a triangle T, and in particular, between two edges a and b.

Definition 4.2.1 *A curve γ forms a* **path from edge a to edge b** *if it is a path, and $\gamma(0)$ is on edge a and $\gamma(1)$ is on edge b.*

A path *between* edges a and b is either a path from edge a to edge b or from edge b to edge a.

Furthermore, we wish to constrain the path not to stray too far from the triangle through which the object wishes to pass. Hence we define a path which travels through a triangle T below.

Definition 4.2.2 *A curve γ forms a* **path through triangle T for a circular object of radius r** *if it is a path, and $\forall x \in [0, 1], \gamma(x)$ is within distance r of some point in T.*

Using this definition, if an object following a path between edges a and b in triangle T completely leaves this triangle, for example by crossing edge c, we consider it instead as separate paths going from edge a to edge c, continuing somehow through other triangles, and then returning to triangle T going from edge c to edge b.

The reason it is not required for the path to stay entirely within T is because sometimes a path might exist where the object is always partially within T but where the path itself might cross edge c. Suppose the triangle opposite edge c is T'. If we required the path to be entirely within T, such a path would require finding a path from a to c in T, from c to c in T', and then from c to b in T. This would complicate the problem unnecessarily and so such a path is only considered to be going through triangle T.

In Section 4.3, we will prove the fact that not requiring the path itself to stay entirely within T allows us to rule out paths which traverse any triangle more than once. This is a useful result both in searching for a valid path, and avoiding complications in the funnel algorithm described in Section 4.4. In addition, Section 4.5 illustrates how this does not affect the calculation of a valid path for a nonpoint object through a series of adjacent triangles whose widths are sufficient for the object.

An example of a path γ such that $\exists x \in [0, 1]$ where $\gamma(x)$ is not in T but $\nexists x \in [0, 1]$ where $\gamma(x)$ is more than distance r away from any point in T, and thus is a path through triangle T, is shown in Figure 4.20. An example of a path γ such that $\exists x \in [0, 1]$ such that $\gamma(x)$ is not within distance r of some point in T, and thus is not a path through triangle T, is shown in Figure 4.21.

Another obvious requirement for a path is that it does not bring the object following it, into collision with an obstacle. Here we will define again what exactly comprise the obstacles in the environment for the purposes of further definitions.

Definition 4.2.3 *Let $O \subset \Re^2$ be a set of* **obstacles** *in the environment. In particular, $\forall o \in O, o$ is either a vertex in the triangulation, or a point on some constrained edge.*

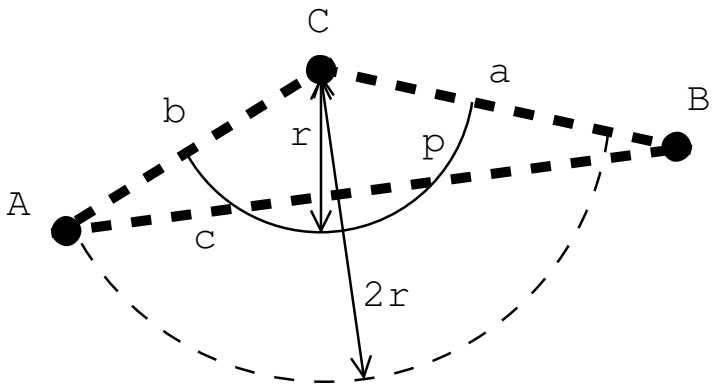

Figure 4.20: A path that is always within r of T

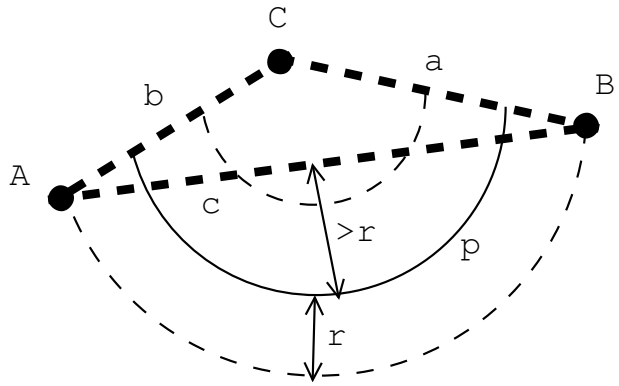

Figure 4.21: A path that is $> r$ away from T at some points
u

With this defined, we can now describe exactly what corresponds to a collision, and the characteristics of a path which will not result in one for the object.

Definition 4.2.4 *A curve γ forms an* **unobstructed path** *for a circular object of radius r if it is a path, and $\forall x \in [0,1]$, $\nexists o \in O$ such that $dist(\gamma(x), o) < r$.*

Furthermore, we can eliminate considerations of obstacles across the edges the path connects, in this case, edges a and b. For example, when the object is crossing the edge a—that is, for points on the path for which some segment of length r extending from it intersects edge a—we do not consider obstacles in the region opposite edge a from triangle T. Similarly, when the object crosses b, we do not consider obstacles in the region opposite edge b from T. Thus, here we provide a definition for a path which is not obstructed by obstacles that do not lie across those edges.

Definition 4.2.5 *A curve γ forms an unobstructed path for a circular object of radius r* **between edges** *a* **and** *b if it is a path between edges a and b, and $\forall x \in [0,1]$ and $o \in O$ such that $dist(\gamma(x), o) < r$, the segment between $\gamma(x)$ and o crosses either edge a or edge b.*

The reason for this requirement is because the ultimate goal of finding these paths through individual triangles is to combine the paths through adjacent triangles together to

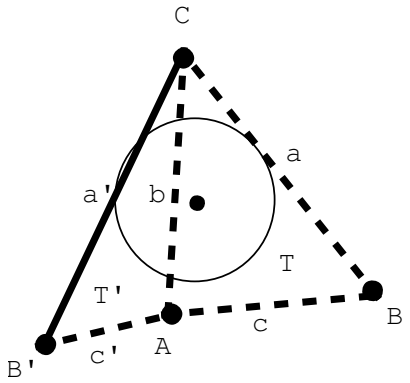

Figure 4.22: Obstacle outside of triangle T interfering with a path inside of it

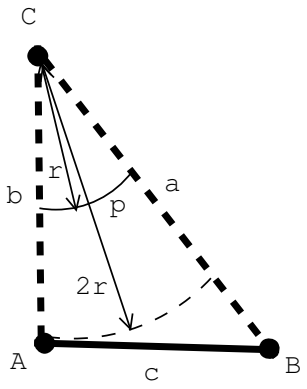

Figure 4.23: An example of an arc path

form a single path through a triangulation. Thus, obstacles on the opposite side of a will be the considered when finding a path through the triangle which shares edge a, and similarly for b. An example of such an obstacle across edge b interfering with a path in triangle T is given in Figure 4.22.

We now have sufficient conditions that we can formulate what is required for a path to be *valid*.

Definition 4.2.6 *A curve γ forms a* **valid path** *through triangle T between edges a and b if it is an unobstructed path between edges a and b through triangle T. When the context is clear, we simply refer to such a path as a* **valid path**.

Now, for the purpose of the proofs that follow, we will define a specific type of path called an *arc path*.

Definition 4.2.7 *A curve γ is an* **arc path** *for a circular object of radius r between edges a and b in a triangle T if it is a path between those edges in that triangle, and $\forall x \in [0,1], dist(\gamma(x), C) = r$, where C is the vertex at which edges a and b meet.*

An arc path is so called because it forms an arc between these edges a and b of radius r. Equivalently, we may refer to such a path as one which "hugs" vertex C. Figure 4.23 shows an example of such a path. Arc paths are used because by their very definition, they have

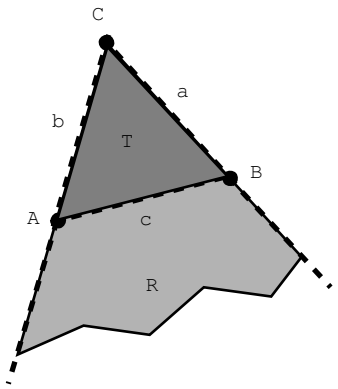

Figure 4.24: Region R for triangle T when moving between edges a and b

only to be shown to be unobstructed to qualify as a valid path. For this final requirement to be met for an object of radius r, the closest obstacle to vertex C must be at distance $\geq 2r$ from vertex C. We prove that this condition holds if and only if there exists a valid path in the sections that follow.

We wish to further narrow the possibilities for obstacles that can cause an arc path to become obstructed. Consider the region which exists between the rays extending from vertex C toward A and from C toward B. This forms what is called a *cone* in two-dimensional space, referred to here as region R. Figure 4.24 illustrates this region for a triangle.

Sometimes an obstacle outside of region R can interfere with a path through T between edges a and b. If, at some point along a path, the object is crossing the boundary of R outside either edge a or edge b, then those obstacles are not being considered by the paths through the triangles sharing those edges and thus must be considered by the path through T.

However, such obstacles will not interfere with an arc path, because an object following an arc path will never cross the boundaries of the region R other than at edges a and b, as we show here.

Lemma 4.2.8 *A circular object of radius r following an arc path will not cross the boundary of R other than through edges a and b.*

Proof We will prove this by contradiction, by assuming we have a valid path which crosses this boundary and showing it is not an arc path.

Consider an arc path γ for an object of radius r between edges a and b in a triangle T that is valid. Without loss of generality, assume γ goes from edge a to edge b. Because γ is valid and thus unobstructed, $\forall x \in [0,1]$, $\nexists o \in O$ such that $dist(\gamma(x), o) < r$. And since $\gamma(0)$ is on edge a and $\gamma(1)$ is on edge b, obstacles must be at least distance r from these points. Now, $\forall x \in [0,1], dist(\gamma(x), C) = r)$, so since vertices are considered obstacles, this means that $dist(C, A) > 2r \wedge dist(C, B) > 2r$.

Without loss of generality, we will simply consider the boundary of R by edge b. The proof extends identically to the boundary by edge a. Assume, for some $x \in (0,1), \gamma(x)$ is distance $< r$ from the closest point to it w on the boundary of region R, that is, an object of radius r on $\gamma(x)$ will overlap this boundary. Furthermore, $dist(\gamma(x), A) \geq r$, because vertex A is an obstacle, and the segment joining $\gamma(x)$ and w does not intersect edge b.

Thus, since the segment from w to $\gamma(x)$ must be perpendicular to the boundary of R, then $dist(\gamma(x), C) > dist(w, C)$. Also we know that $dist(w, C) > dist(A, C)$, and $dist(A, C) \geq$

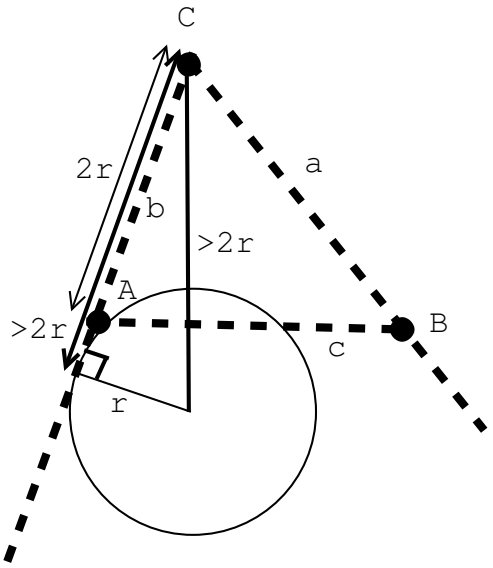

Figure 4.25: An object crossing the boundary of region R below edge b

$2r$. Thus, $dist(\gamma(x), C) > 2r$, which violates our definition that $\forall x \in [0,1], dist(\gamma(x), C) = r$ for an arc path γ. A diagram of this proof is found in Figure 4.25.

Thus, an arc path will not cross a boundary of region R outside of edges a and b, as desired. ∎

Corollary 4.2.9 *To verify that an arc path is unobstructed, we need only consider obstacles within region R.*

4.2.2 Proofs

Using the definitions from Subsection 4.2.1 above, we will prove the equivalence of the closest obstacle in region R of a triangle being at a distance of $2r$ from vertex C and the existence of a valid path between edges a and b of the triangle T for a circular object of radius r.

Theorem 4.2.10 *If there is no obstacle within $2r$ of vertex C in region R, then there is a valid path through triangle T from edge a to edge b for a circular object with radius r. In particular, there is such a path hugging vertex C.*

Proof Consider an arc path γ. By its very definition, an arc path is already guaranteed to be a path in triangle T between edges a and b. It remains to prove that if $\nexists o \in O$ such that o is in R and $dist(o, C) < 2r$, in which case the arc path is unobstructed and thus a valid path.

By the definition of an arc path, $\forall x \in [0,1], dist(\gamma(x), C) = r$. Thus, for this path to be obstructed, it must be that $\exists o \in O$ such that $dist(o, \gamma(x)) < r$ for some $x \in [0,1]$.

However, because of the triangle inequality, we know that $\nexists o \in O, x \in [0,1]$ such that $dist(o, C) \geq 2r \wedge dist(o, \gamma(x)) < r \wedge dist(\gamma(x), C) = r$. That is, an obstacle that is $\geq 2r$ from vertex C cannot be $< r$ from any point that is at distance r from that same point. This is depicted in Figure 4.26 for clarity.

Thus, if there are no obstacles within distance $2r$ of vertex C in region R, there are none within distance r of any point along the path from edge a to b in triangle T hugging vertex C. Therefore, γ is unobstructed and qualifies as a valid path. ∎

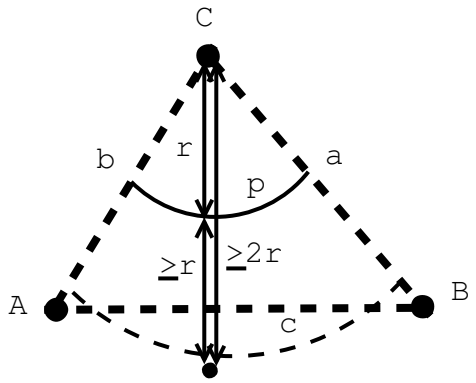

Figure 4.26: Using the triangle inequality to prove soundness

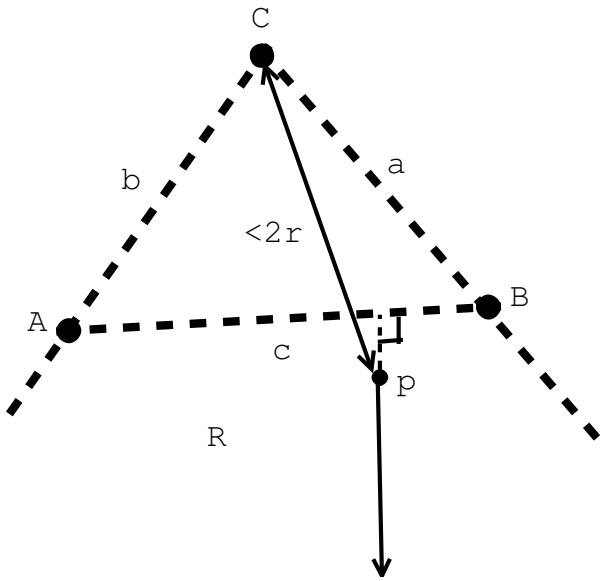

Figure 4.27: Partitioning of region R into 2 sub-regions

Theorem 4.2.11 *If there exists a valid path γ between edges a and b in triangle T for an object of radius r then $\nexists o \in O$ such that o is in R and $dist(o, C) < 2r$.*

Proof Assume $\exists o \in O$ such that o is in R and $dist(o, C) < 2r$. Consider the point p where p is in R and $dist(p, C) \leq dist(o, C) \forall o \in O$ such that o is in R. That is, p is the closest obstacle to vertex C in region R. Such a point must exist in region R because by Lemma 4.2.8, only obstacles in region R can interfere with an arc path from edge a to edge b through triangle T.

For the remainder of the proof, we will consider this point to be the only obstacle in region R. This is a relaxed constraint, and certainly, if no unobstructed path exists with this single point obstacle, no path unobstructed exists with any set of obstacles including p. We know that $\nexists o \in O$ such that o is in T, because an obstacle would be included in the triangulation as a vertex or constrained edge; all triangles are free of obstacles by the triangulation's construction.

Now consider the line segment going from vertex C to this point p. We know that the

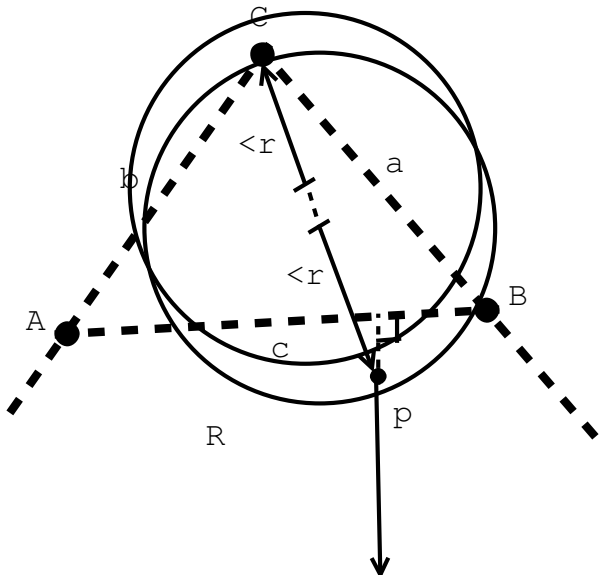

Figure 4.28: An object trying to pass between C and p

length of this line segment is $< 2r$. Similarly, consider a ray extending from p perpendicular to and away from edge c of the triangle. One can see that these partition R into two sub-regions. This partitioning can be observed in Figure 4.27. Thus, we can see that any valid path travelling between edges a and b in region R must cross either the segment between C and p, or the ray extending from p, since edge a and edge b are in different sub-regions of R, and a path cannot leave R without leaving T and thus becoming invalid. We will cover both of these cases below.

We will now show that any path γ between edges a and b is necessarily obstructed, or leaves triangle T, in both cases, becoming invalid. First, assume γ crosses the segment between C and p at some point u. That is, $\gamma(x) = u$ for some $x \in [0, 1]$. Since the segment between C and p has length $< 2r$, we know u must be at distance $< r$ from either vertex C or p (or both). Thus, if γ crosses any point on this segment, it comes within distance r of some obstacle, and thus becomes obstructed. This can be seen in Figure 4.28 where the above object is interfering with vertex C (its centre is distance $< r$ from C) and the lower object with point p.

Similarly, if γ crosses the ray extending from p perpendicular to and away from edge c, it must cross at some point v on the ray. That is, $\gamma(x) = v$ for some $x \in [0, 1]$. Furthermore, $dist(v, p) \geq r$, otherwise γ would be obstructed and become an invalid path. Figure 4.29 shows that since p is not in T, and the ray is perpendicular to c—the closest edge of T to p—the point that is distance r away from v that is closest to triangle T is along that ray. However, since v is at least r away from p, which itself is outside of T, it follows that at point v, there is no point within distance r which is inside triangle T. Thus, γ is not a path through T, and not valid.

Finally we conclude that since there is nowhere that a path from edge a can cross the partition to edge b, or vice versa, and remain valid, no valid path is possible. Thus, if a valid path exists for moving a circular object of radius r from edge a to edge b in triangle T, there must be no obstacles within distance $2r$ of vertex C in region R, as desired. ∎

Here we will prove that an arc path is the least distance valid path through a triangle. This result will become useful when we begin searching for paths in Constrained (Delaunay)

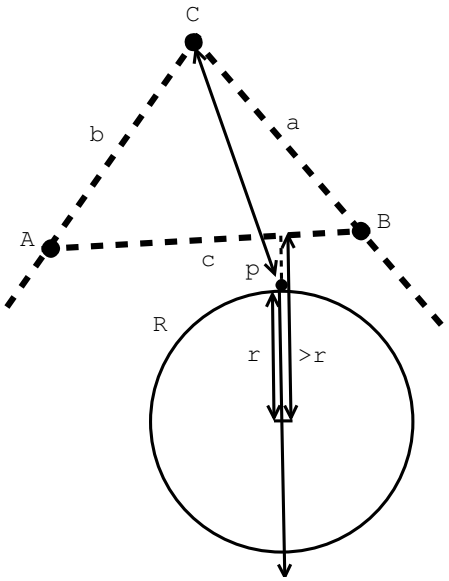

Figure 4.29: An object trying to pass below point p

Triangulations.

Theorem 4.2.12 *The arc path γ is the shortest valid path for a circular object with radius r between edges a and b through triangle T.*

Proof We know that for any valid path γ for a circular object with radius r between edges a and b through triangle T, $\forall x \in [0,1], dist(\gamma(x), C) \geq r$, where C is the vertex joining edge a and b, otherwise, γ would be obstructed and thus invalid. Again, we assume all vertices in a triangulation are obstacles.

The path such that $\forall x \in [0,1], dist(\gamma(x), C) = r$ is indeed the arc path for a circular object with radius r, so it remains to prove this path is shorter than valid paths φ such that $\exists x \in [0,1], dist(\varphi(x), C) > r$.

There are two cases to consider for this proof: one for paths φ where $dist(\varphi(0), C) = r \wedge dist(\varphi(1), C) = r$, that is, it meets edges a and b at the same points as the arc path, and one for paths ψ where $dist(\psi(0), C) > r \vee dist(\psi(1), C) > r$, that is, it meets these edges at different points.

We will consider the former possibility first. We know that $dist(\varphi(0), C) = r$ and $dist(\varphi(1), C) = r$ and $\exists x \in (0,1), dist(\varphi(x), C) > r$, that is, φ departs from the arc path at some point in between edges a and b. Let $k = \varphi(x)$. Also, $\exists w \in [0, x), dist(\varphi(w), C) = r$, that is, φ departs from the arc path at some point. Let $i = \varphi(w)$. Also, $\exists y \in (x, 1], dist(\varphi(y), C) = r$, that is, φ rejoins the arc path at some point. Let $j = \varphi(y)$. This configuration is shown in Figure 4.30.

Consider a path φ' that follows the arc path to point i, goes in a straight line to point k, and then to point j, and continues following the arc path through the triangle. Because the shortest distance between any two points is a straight line, we know that the length of φ' is no greater than that of φ. It remains to prove that the arc path between points i and j is shorter than the path from i to k to j, since this is where φ' differs from the arc path.

Take first the triangle formed by vertex C, and points i and k as shown in Figure 4.31. Consider $\angle iCk$ to be θ. We know the arc in this triangle has length $r\theta$ so we must show $dist(i, k) > r\theta$.

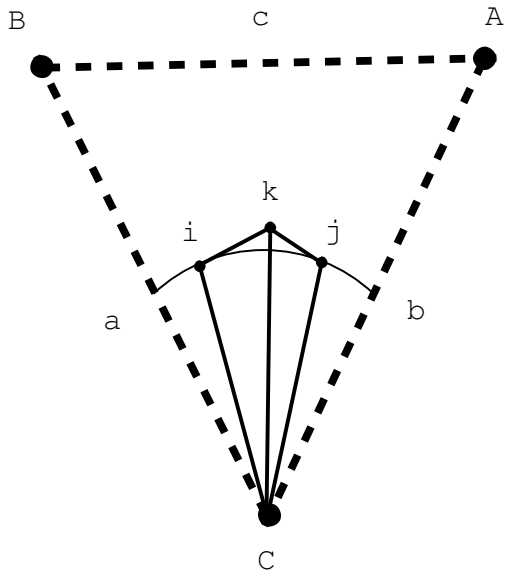

Figure 4.30: An alternate path departing from the arc path in the middle

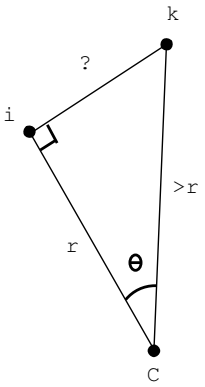

Figure 4.31: One section of the alternate path

We know that $dist(C, i) = r$ and $dist(C, k) > r$, so we must find $dist(i, k)$. We also know that $\angle Cik$ is either a right angle or obtuse, because otherwise segment ik would be closer than r to vertex C at some points, making φ' obstructed and thus invalid. Now, as θ increases, $dist(i, k)$ increases given we know $dist(C, i)$ and angle θ. Thus, we will assume $\angle Cik$ is a right angle, and the real length of segment ik will be at least that which we find using this assumption.

This way, we can say that $dist(i, k) \geq r \cdot tan(\theta)$, so then we have $r \cdot tan(\theta) \geq r\theta \Rightarrow tan(\theta) \geq \theta$, which is true for $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$, which is all we require. Thus, segment ik is longer than the corresponding portion of the arc, and this applies identically to segment kj as well. We have shown, then, that the arc path is the shortest of valid paths φ such that $dist(\varphi(0), C) = r \wedge dist(\varphi(1), C) = r$, that is, all paths that pass through points at distance r away from vertex C on edges a and b.

The proof concerning paths that do not meet both edges a and b at distance r from vertex C, follows similarly. Consider one side where the alternate path ψ and the arc path

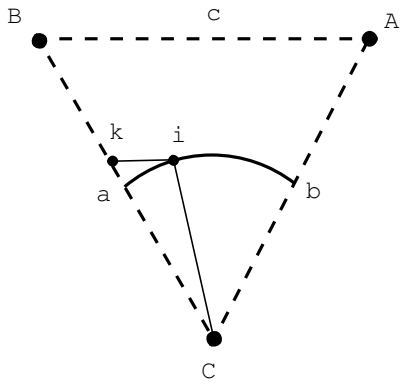

Figure 4.32: Alternate path departing from the arc path on one side

do not meet an edge at the same place. Without loss of generality, assume this is edge a and that ψ goes from edge a to edge b, that is, $dist(\psi(0), C) > r$. We will also assume that the arc path and ψ have at least one common point, that is, $\exists x \in (0, 1], dist(\psi(x), C) = r$. The extension of this proof to alternate paths that do not meet the arc path at all will be given afterward.

Let i be the closest such common point to edge a, or in other words, $i = \psi(w)$ where $dist(i, C) = r, w \in (0, 1]$, and $\forall x \in (0, 1]$ such that $dist(\psi(x), C) = r, x > w$. Also, let $k = \psi(0)$. This is shown in Figure 4.32. Now, consider the path ψ' consisting of a straight segment between i and k and then following the arc path to edge b. We know that the length of ψ' will not exceed that of ψ, because the straight line path between i and k cannot be longer than any other path between those points, and the arc path is no longer than any other path between i and edge b, as shown above. The proof that segment ik is longer than the corresponding section of the arc path is identical to the proof above. This extends equivalently to edge b and paths going from edge b to edge a.

Now we will show that this extends to any path ψ that does not share any common points with the arc path. Consider $w \in [0, 1]$ such that $dist(\psi(w), C) \leq dist(\psi(x), C) \forall x \in [0, 1]$, that is, $\psi(w)$ is the closest point to vertex C on path ψ. We know $dist(\psi(w), C) > r$, because if $dist(\psi(w), C) = r$ ψ would have a common point with the arc path and the proof above would apply, and if $dist(\psi(w), C) < r$, ψ would be obstructed by vertex C and thus not be valid.

From the proof above, the length of ψ is greater than that of the arc path for a circular object with radius $dist(\psi(w), C)$. Since this in turn is longer than the arc path of radius r ($dist(\psi(w), C) > r \Rightarrow dist(\psi(w), C) \cdot \theta > r\theta$ where $\theta = \angle ACB$), we have that ψ is longer than the arc path for a circular object with radius r.

Thus, we have shown that the arc path of radius r between two edges of a triangle is the shortest valid path for a circular object of radius r between those edges of that triangle, as desired. ∎

4.3 The Delaunay Property

The requirement that our triangulation be Delaunay not only ensures that "sliver" triangles are avoided as much as possible, providing triangular areas that better describe the environment, but also provides some results useful for searching the triangulation for a path and assuring that the algorithm for determining a triangle's width is manageable.

Theorem 4.3.4 states that when searching for a path through a Delaunay Triangulation,

those that pass through any one triangle multiple times need not be considered since shorter paths are preferred to longer ones. Theorem 4.3.6 says that in a CDT, the search to provide a triangle's width will never cross both opposite edges of a triangle. Finally, we briefly discuss how that the widths through different parts of the triangulation are identical regardless of the placement of the unconstrained edges.

The ability to eliminate paths crossing a triangle multiple times is useful not only in decreasing possibilities during search, but also avoids complication in the funnel algorithm described in Section 4.4 and the alternate version described in Section 4.5. Determining the shortest path through a group of adjacent triangles in time linear in the number of those triangles, can be problematic if a triangle is included twice, effectively creating a loop in this area.

Definition 4.3.1 *For an object of radius r and triangles t_1, t_2, and t_3 where t_1 and t_3 are adjacent to t_2 (via unconstrained edges), and $t_1 \neq t_3$, define $valid_r(t_1, t_2, t_3)$ if and only if the width of triangle t_2 when moving between the edges shared with triangles t_1 and t_3, is $\geq r$, that is, the object has a valid path from triangle t_1 through t_2 to t_3 (and also in the other direction).*

Definition 4.3.2 *Now, $path_r(t_1, t_2, ..., t_{n-1}, t_n) \Leftrightarrow \forall i, 1 \leq i < n-1, valid_r(t_i, t_{i+1}, t_{i+2})$, that is, an object or radius r has a valid path from triangle t_1 through t_2 and so on, in sequence, arriving at triangle t_n.*

Definition 4.3.3 *Let $|path_r(t_1, t_2, ..., t_{n-1}, t_n)|$ be the length of the shortest path through the triangles $t_1, t_2, ..., t_{n-1}, t_n$.*

If two intermediate triangles were the same, say $t_i = t_j$ for some $1 \leq i < j \leq n$ then removing the triangles in between them from this sequence would necessarily shorten the path, that is:

$$|path_r(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_{j-1}, t_j, t_{j+1}, ..., t_n)| > |path_r(t_1, ..., t_{i-1}, t_i, t_{j+1}, ..., t_n)|$$

Obviously since we have a desire to find the shortest path possible, we would not consider a path which visits the same triangle multiple times so long as this still results in a valid path. However this is only possible if $valid_r(t_{i-1}, t_i, t_{j+1})$, and depending on the triangulation, this may not be true. Luckily, it is for CDTs.

Theorem 4.3.4 *For a circular unit of radius r moving through a Constrained Delaunay Triangulation with triangles t_k,*

$$valid_r(t_{i-1}, t_i, t_{i+1}) \wedge valid_r(t_{j-1}, t_j, t_{j+1}) \wedge t_i = t_j \Rightarrow valid(t_{i-1}, t_i, t_{j+1})$$

Proof Figure 4.33 shows the case where an object could travel between two of the three pairs of edges of triangle T, but not the third pair. Here, if an object like that shown at edge a wants to reach edge b, it would have to pass through edge c, then edge a', somehow return through edges b' and c, then finally move to edge b.

In a Delaunay Triangulation, the diagonal, edge c in this case, should instead be between vertices C and C'. We will use the property of a Delaunay Triangulation that there cannot be a point inside the circumcircle of any triangle [18]. Since there are no constrained edges in this portion of the triangulation for such a situation, we can simply consider it a Delaunay Triangulation, and disregard the exceptions to the above rule required for a Constrained Delaunay Triangulation.

We will show that if this portion of the triangulation is Delaunay, that is, if there is no vertex in the circumcircle of triangle T, then there will be a valid path between edges

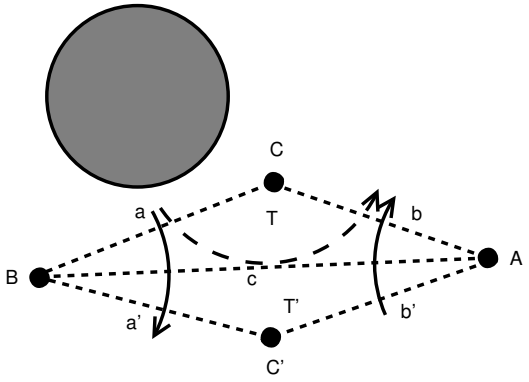

Figure 4.33: Objects of certain radii can move between edges a and c, and edges b and c, but not between edges a and b, of triangle T

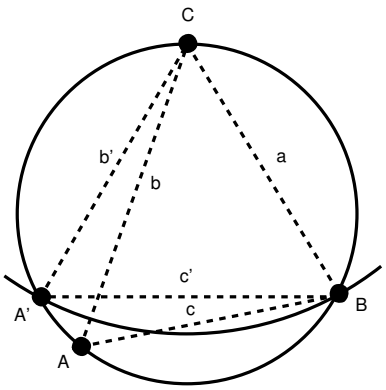

Figure 4.34: Circumcircle around triangle T and arc around vertex C

a and b for any object with a path between edges a and c, and edges b and c (an object with radius $r \leq min\{|a|, |b|, |c|\}$). Equivalently we wish to find that the arc around vertex C with radius $r = min\{|a|, |b|\}$ is unobstructed in region R between rays $\overrightarrow{CB}$ and $\overrightarrow{CA}$.

Refer to Figure 4.34. Here we have an arbitrary triangle, and a circumcircle around it. Assume, without loss of generality, that we wish to travel between edges a and b, and that $|a| \leq |b|$. So then we have an arc centered at vertex C, extending to vertex B (the arc has radius $|a|$). Since we know no vertices can be in the circumcircle, and we wish that there are no obstacles in the arc, we will show that the arc cannot extend past the circumcircle in the region R.

Obviously since the arc and the circumcircle intersect at vertex B and what we show as vertex A', which is the other point on the circumcircle at distance $|a|$ from vertex C, any extension of the arc past the circumcircle in region R (or as a relaxed requirement, the region between rays $\overrightarrow{CB}$ and $\overrightarrow{CA'}$) would happen at the opposite side of the circumcircle as vertex C (the bottom of Figure 4.34).

However, any extension of the arc past the circumcircle would require the radius of the arc to be twice that of the circumcircle, but the radius of the arc is known to be the length of edge a, so if this were true, vertices C and B could not both be on the circumcircle, which is a contradiction with the definition of the circumcircle. Thus, no vertex in a Delaunay Triangulation can cause an arc path to be invalid.

Therefore in Figure 4.33, the diagonal would be between vertices C and C' instead of A and B, and thus the path that the object would have to take again would only traverse each triangle once. We now have that

$$valid_r(t_{i-1}, t_i, t_{i+1}) \wedge valid_r(t_{j-1}, t_j, t_{j+1}) \wedge t_i = t_j \Rightarrow valid(t_{i-1}, t_i, t_{j+1})$$

as desired. ∎

Corollary 4.3.5 *Thus, for any object of radius r such that*

$$path_r(t_1, ..., t_{i-1}, t_i, t_{i+1}, ..., t_{j-1}, t_j, t_{j+1}, ..., t_n)$$

and $t_i = t_j$ for some $1 \leq i < j \leq n$, we can take the shorter $path_r(t_1, ..., t_{i-1}, t_i, t_{j+1}, ..., t_n)$ equivalently, and so we do not have to consider a path which traverses any triangle multiple times during search since it could just be replaced by a shorter path, as desired.

It is useful to know when determining the width of a triangle by searching across unconstrained edges, that the search will not cross two unconstrained edges of the same triangle, since it is another indication that the search will not traverse an unwieldy number of triangles. We prove that in a CDT, this situation is impossible.

Theorem 4.3.6 *In a Constrained Delaunay Triangulation, determining the width of a triangle will never result in a search across two unconstrained edges of the same triangle.*

Proof Figure 4.35 shows a case in which search across multiple edges of the same triangle is possible when determining the width between edges a and b.

If either edge a or b were constrained, one would not have to check if an object could pass between them, and if any other edge were constrained, the search could not reach the branching point since it would stop at the obstacle. So again we can use the properties of a regular Delaunay Triangulation to show that this situation cannot happen.

Again we use that there cannot be a vertex in the circumcircle of any triangle in a Delaunay Triangulation. For the width search to cross edge a', $\angle CBC'$ would have to be acute, and similarly $\angle CAC'$ must be acute for the search to cross edge b'. Without loss of generality, we will consider half of the quadrilateral $CBC'A$ and the circumcircle of triangle CBA, as shown in Figure 4.36.

Assume vertex C's position is fixed to the "top" of the circumcircle, we must show that when $\angle CBA$ is acute, then vertex A cannot intersect the circumcircle on the same side

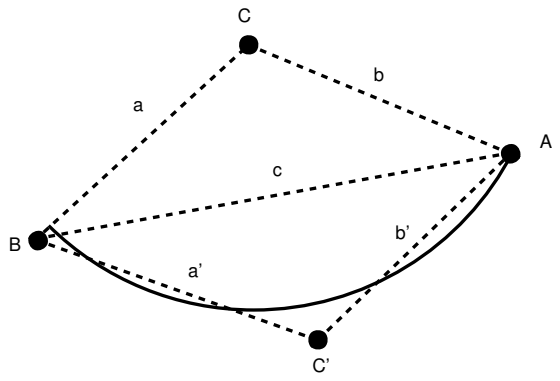

Figure 4.35: Situation in which determining the width between edges a and b would result in a search across multiple edges

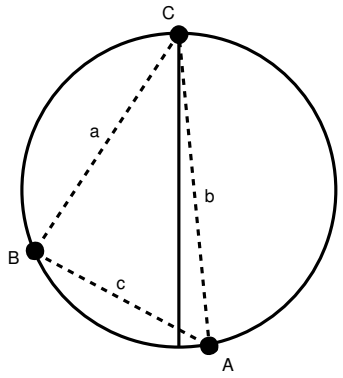

Figure 4.36: Half of the quadrilateral and triangle's circumcircle

of the circle as the bisector going through vertex C. This is because we know that since triangle CBA is acute, the circumcenter must be inside the triangle [6].

Since this applies to both sides, it will prove that where these segments cross, forming the fourth vertex of the quadrilateral, must be inside the circumcircle, making the triangle illegal in a Delaunay Triangulation. Therefore in a Delaunay triangulation, the search to determine the width of a triangle cannot cross two unconstrained edges of the same triangle, as desired. ∎

With all the results requiring the triangulation be Delaunay, it poses the question as to whether the triangle widths are sensitive to the placement of the unconstrained edges. Luckily this is not the case, which is useful due to the fact that there can be multiple Constrained Delaunay Triangulations for a set of points and constraints (if four points lie on the same circle).

To verify this, one can simply observe that the determination of each triangle's width between two edges is the distance between one of its vertices (considered to be an obstacle), and the closest obstacle between those two edges. Obstacles, both vertices and edges, as we know are required to be in the triangulation unchanged.

So it only remains to see that the regions formed between the pairs of edges of all triangles cover the area of the triangulation, and that it suffices to measure the distances between vertices and other obstacles. The former is trivial since the triangulation is made of triangles and the region between each triangle's edges at least covers that triangle. The latter follows from the fact that obstacles are vertices and line segments, and the minimum distance between two line segments is always equivalent to the distance between one of the endpoints (a vertex in the triangulation) and somewhere on the other segment.

4.4 Funnel Algorithm

The object of pathfinding in a triangulation is to find a series of adjacent triangles inside the first of which is the start point and inside the last of which is the goal. As stated in Theorem 4.3.4, whenever such a series of triangles contains any duplicates, the triangles between such matching pairs can be removed to shorten the path, providing the triangulation is Delaunay. Thus, we consider such a series of triangles to never contain a triangle more than once.

Obviously, as described throughout this chapter, we wish that the object have a valid path between each of the edges of each triangle in this series. If the object has a valid path between the edges of each triangle in such an adjacent series, it follows that there exists

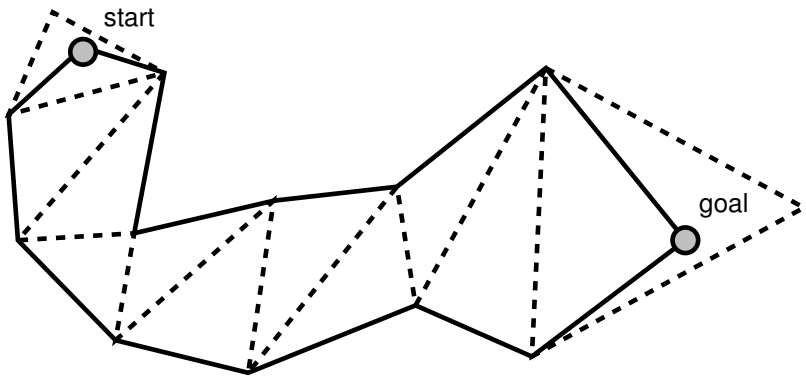

Figure 4.37: A series of triangles, start and goal points, and the resulting channel

a valid path that traverses all of these triangles. The most obvious such path consists of the arc path within each triangle, with these paths through adjacent triangles connected by straight segments.

Obviously for a path to be valid, the object must not overlap any obstacles when centered on either the start or goal positions. We can usually assume that the start point is valid since the object should already be positioned in an unobstructed position, however if the start or goal point needs to be checked for validity, one can use a test similar to that for determining the closest obstacle to a vertex.

We have left to ensure that there are unobstructed paths between the start and goal point and the edge connecting the triangles containing them, to the rest of the series. Since we know the positions themselves are unobstructed, obviously moving to a less restrictive area would not result in further obstruction. And the most obstructed area in a triangle is that between a vertex and its closest obstacle.

Therefore if the object must pass between a vertex and its closest obstacle, this width must be at least the diameter of the object for this path to be valid. If the object is already closer to the next triangle than this area, this test is unnecessary since the object is only moving to a less obstructed area than its starting point, which was already checked.

Now that we know such a path exists, we look at a technique for providing the shortest such path within this series [24]. We will define an *interior edge* to be unconstrained edges in the triangulation which the object will cross when traversing a path through this series of triangles. That is, it is one shared by two triangles adjacent in this series. Also, define an *interior triangle* to be a triangle in the series not containing the start goal point. That is, all triangles between and not including the first and last triangles in the series.

A *channel* is the simple polygon inside which we wish to find a valid path for the object. The vertices of the channel consist of the start and goal points as vertices, along with the vertices of all the interior triangles in the series. The edges of this polygon are those of the interior triangles other than the interior edges. Figure 4.37 shows a start and goal point, a series of triangles, and the resulting channel.

We use this channel and the interior edges in order to find the shortest path between the start and goal points, within this channel. This can be done in time linear in the number of triangles in the series, using what is called the *funnel algorithm* [12, 38]. This technique works for point objects; we will discuss the extension to circular objects of nonzero radius in Section 4.5.

The funnel algorithm considers three structures: the *path*, the *apex*, and the *funnel*. The path is the series of line segments forming the portion of the shortest path known at

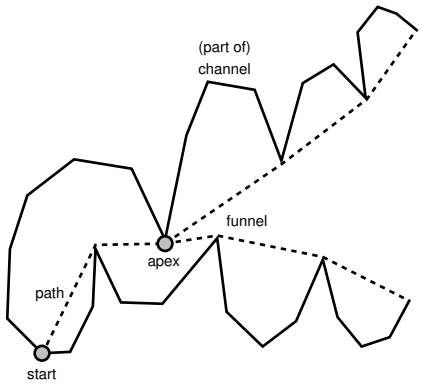

Figure 4.38: The path, apex, and funnel during a run of the funnel algorithm

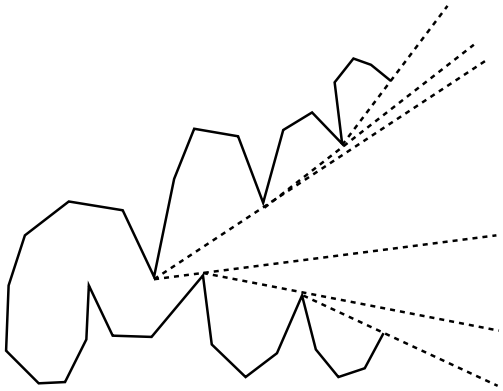

Figure 4.39: Wedges corresponding to the vertices in a funnel

the current point in the algorithm. The funnel consists of two series of line segments, one turning clockwise and one counterclockwise, which represent the area in which all shortest paths to the area not yet processed, must be. Finally, the apex is the point which joins the path and the funnel. Figure 4.38 shows these structures.

At the start of the algorithm the path is empty, the apex is set to the start point, and the funnel begins as the segments connecting the start point to the first interior edge. The funnel is stored in a deque structure. Each interior edge is processed in turn, with the vertex not already processed being added to the corresponding side of the funnel deque. Vertices are popped from that side of the funnel until the wedge in which the current vertex lies is discovered, at which point that vertex is added to the end of the funnel deque on that side. These wedges are illustrated in Figure 4.39.

If the apex is popped off the deque in this way, the next vertex to be considered becomes the new apex, and a segment connecting the old apex to the new one gets added to the path. Once the final interior edge of the channel is added to the apex, we add the goal point to the funnel on either side, say the right. Once this is done, we combine the path with the right side of the funnel to form the entire path between the start and the goal. This process yields the shortest path within the channel, as shown in [24]. Pseudocode for the funnel algorithm is given in code listings 4 and 5.

Here we consider an example of adding the same vertex to each side of the deque. If the object would cross the topmost edge of the new triangle, the vertex would be added to the

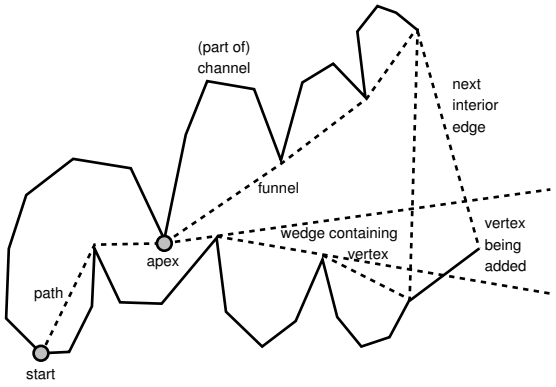

Figure 4.40: Adding a vertex to the right side of the funnel

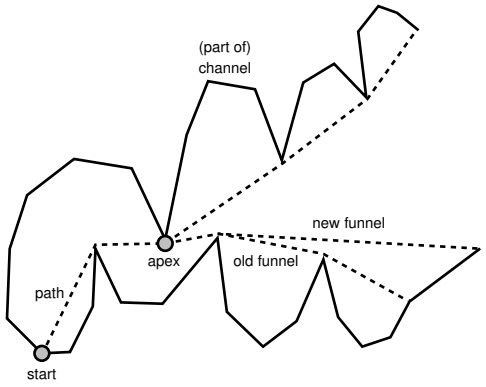

Figure 4.41: The new funnel after a vertex is added on the right

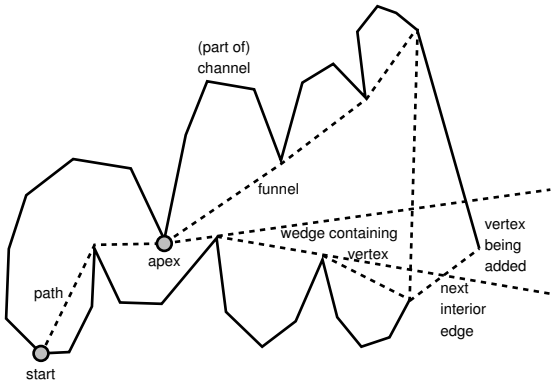

Figure 4.42: Adding a vertex to the left side of the funnel

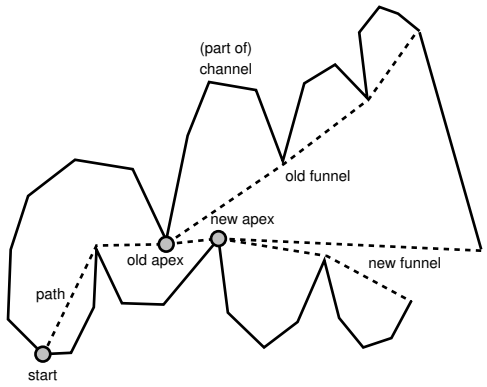

Figure 4.43: The new funnel and apex and path after a vertex is added on the left

right side as in Figure 4.40. Here all vertices to the right of the one in whose wedge the new vertex lies, are removed from the deque, and the funnel on that side changes to include the segment between those two vertices. This is shown in Figure 4.41.

If the object would cross the bottom edge of the new triangle, the vertex would be added to the left side of the deque, resulting in the situation in Figure 4.42. Here all the vertices on the left side of the deque are popped off, and then the apex is popped, moving the apex one vertex to the right and extending the path to include the segment between the old apex and the new one. Then we reach the vertex whose wedge contains the new vertex, so we stop popping vertices and replace the left side of the funnel with the segment between the apex and the new vertex as in Figure 4.43.

Algorithm 4 Funnel(Channel c, Point s, Point g) : Path

```
 1: p.Clear()
 2: if NumEdges(c) < 1 then
 3:    p.Add(s);   p.Add(g)
 4:    return p
 5: end if
 6: AddVertex(s, p)
 7: f ← FunnelDeque(s)
 8: v_l ← LeftEndpoint(c_0);   Add(f, v_l, Left, p)
 9: v_r ← RightEndpoint(c_0);   Add(f, v_r, Right, p)
10: for i ← 1 to NumEdges(c) do
11:    v'_l ← LeftEndpoint(c_i);   v'_r ← RightEndpoint(c_i)
12:    if v'_l = v_l then
13:       v_r ← v'_r;   Add(f, v_r, Right, p)
14:    else
15:       v_l ← v'_l;   Add(f, v_l, Left, p)
16:    end if
17: end for
18: Add(f, g, Point, p)
19: return p
```

Algorithm 5 Add(FunnelDeque f, Vertex v, Type t, Path p)

```
1:  if t = Left then
2:    loop
3:      if f_Left = f_Right then
4:        f.AddLeft(v);   break
5:      else if f_Left = f_Apex then
6:        θ ← Angle(f_Left, f_Left+1)
7:        φ ← Angle(f_Left, v)
8:      else
9:        θ ← Angle(f_Left+1, f_Left)
10:       φ ← Angle(f_Left, v)
11:     end if
12:     if CounterclockwiseTo(θ, φ) then
13:       f.AddLeft(v);   break
14:     end if
15:     if f_Left = f_Apex then
16:       ψ ← Angle(f_Apex, f_ApexType, f_Apex+1, Right)
17:       AddVertex(f_Apex, p)
18:       f.PopApexLeft()
19:     end if
20:     f.PopLeft()
21:   end loop
22: else if t = Right then
23:   {same procedure, with directions reversed}
24: else if t = Point then
25:   i ← 0
26:   while f_Apex+i ≠ f_Right do
27:     i ← i + 1
28:     AddVertex(f_Apex+i, p)
29:   end while
30: end if
```

4.5 Modified Funnel Algorithm

Now that we have a way to obtain the shortest path between the start and goal points inside the channel for a point object, we desire to find such a path for a circular object of nonzero radius r. Fortunately this can be done by conceptually attaching circles of equal radius around each of the vertices in the channel (except the start and goal points) and performing the funnel algorithm on those.

Both versions of the algorithm can be visualized as a rubber band being pulled through the channel from start to goal. The original funnel algorithm snaps to vertices in between, whereas this modified funnel algorithm bends around these circles. While the funnel algorithm yields line segments between the start and goal points and vertices in between, the modified funnel algorithm results in arcs of radius r around these vertices, and line segments tangent to them.

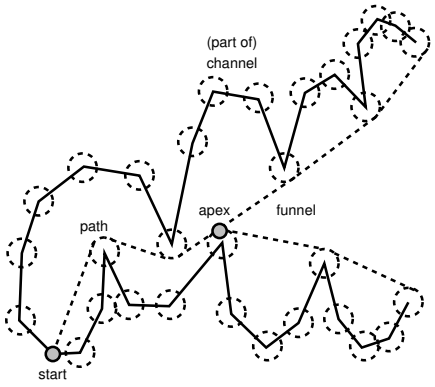

Figure 4.44: The modified funnel algorithm for an object of nonzero radius r

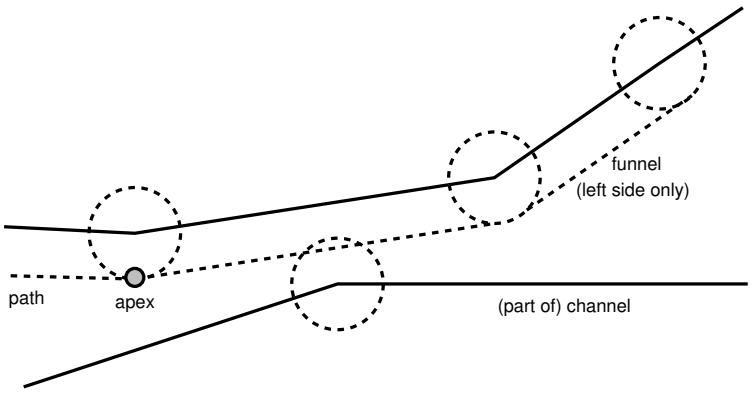

Figure 4.45: Case requiring another adjustment in the modified funnel algorithm

Figure 4.44 shows the modified funnel algorithm being run on the same channel as before, but for an object of some nonzero radius r. There is another modification that must be made to the algorithm to deal with the fact that adding these circles around the vertices of the channel changes the wedges in a way that would not occur in the original funnel algorithm. This case is illustrated in Figure 4.45, where a vertex must be added to one side which interferes with the opposite side of the funnel already constructed.

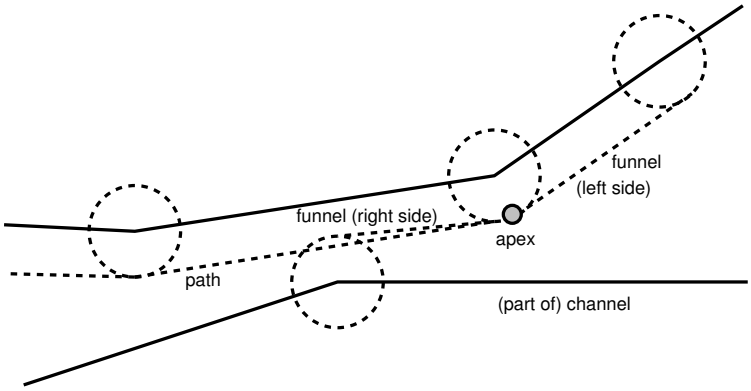

Figure 4.46: Unadjusted modified funnel algorithm run on the degenerate case

This does not occur in the original funnel algorithm because this would require the boundaries of the channel to cross, forming an invalid channel that would not be considered for such an algorithm. This situation is uncommon in Constrained Delaunay Triangulations, but is included here both for completeness and because the algorithm can then also be used in regular Constrained Triangulations. When looking at Figure 4.45, one might notice that part of the channel is too narrow for the object in question (the same size as the circles around the vertices). However, one must remember that the edges bordering the channel could be unconstrained, and part of the object could venture outside this area so long as part always remains inside.

If the modified funnel algorithm were to add the next vertex without the adjustment required to deal with this situation, the result would look similar to Figure 4.46, with the final path backtracking through the channel and probably intersecting an obstacle. Certainly this is not the desired behavior. In order to fit with our expectations of this algorithm, we would like the result of adding this vertex to look like Figure 4.47. The adjustment that must be made for this situation, then, is that when comparing the angle of the funnel moving to the new vertex to the angles of the funnel adjoining a vertex opposite the apex being considered (equivalent comparing the new vertex to the wedge of the other vertex in the original funnel algorithm), if the new vertex is not in this range but is closer to the apex than the other, the apex should be moved to the new vertex (adding the corresponding section to the path). This adjusts the funnel appropriately to produce the result seen in Figure 4.47.

A path that follows such arcs requires that the object be capable of second-order, or curved, motion. Sometimes, this may not be the case and it may have to approximate this motion. For example, the object may only be able to turn in place and move in straight lines. In this instance, the curved part of the path would have to be made up of straight segments similar to the approximation of curved barriers in the initial triangulation.

If the object is navigated by use of waypoints, a similar approximation is necessary. If the object is capable of steered motion such that its turning radius is at most its own radius, the exact paths can be used, but if its turning radius is more than this, a form of "out-in-out" cornering—where the object must approach the turn from farther away, touch the vertex in the middle of the turn, and exit away from the vertex—may be required. Obviously objects with holonomic motion could follow these paths exactly.

While these paths will be valid for a circular object, this technique should also be useful for objects with other shapes. The simplest way to extend this to other object shapes is to find a path using the bounding circle of the object. These paths would be guaranteed

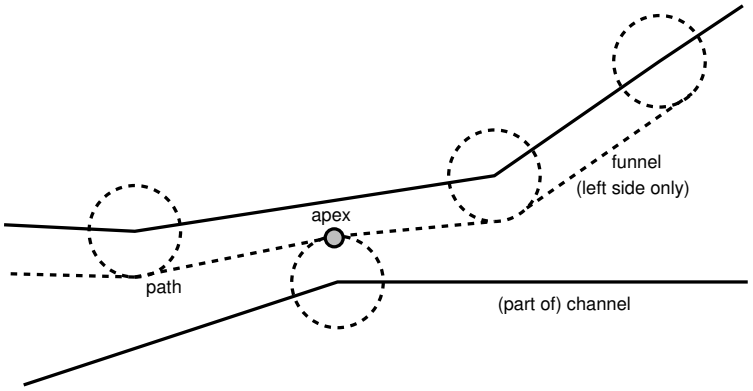

Figure 4.47: The desired result after dealing with the degenerate case

valid, however may miss some paths. For example, a rectangular object could fit through some corridors that its bounding circle could not, however only considering the width of the rectangle would result in collisions when the object would have to turn. Therefore more information about the motion of the object would be required. While the scope of this thesis is limited to radially symmetric objects, it may still be possible to test traversability of the environment and find paths for objects of other shapes.

Code listings 6 and 7 contain pseudocode for adding a vertex conceptually encompassed by a circle of radius r to a funnel deque, and the modified funnel algorithm described above, respectively.

Algorithm 6 Funnel(Channel c, Radius r, Point s, Point g) : Path

```
1:  p.Clear()
2:  if NumEdges(c) < 1 then
3:     p.Add(s);   p.Add(g)
4:     return p
5:  end if
6:  f ← FunnelDeque(s, r)
7:  v_l ← LeftEndpoint(c_0);   Add(f, v_l, Left, p)
8:  v_r ← RightEndpoint(c_0);   Add(f, v_r, Right, p)
9:  for i ← 1 to NumEdges(c) do
10:    v'_l ← LeftEndpoint(c_i);   v'_r ← RightEndpoint(c_i)
11:    if v'_l = v_l then
12:       v_r ← v'_r;   Add(f, v_r, Right, p)
13:    else
14:       v_l ← v'_l;   Add(f, v_l, Left, p)
15:    end if
16: end for
17: Add(f, g, Point, p)
18: return p
```

Algorithm 7 Add(FunnelDeque f, Vertex v, Type t, Path p)

if $t =$ Left **then**
  loop
    if $f_{\text{Left}} = f_{\text{Right}}$ **then**
      f.AddLeft(v); **break**
    else if $f_{\text{Left}} = f_{\text{Apex}}$ **then**
      $\theta \leftarrow$ Angle(f_{Left}, f_{ApexType}, $f_{\text{Left}+1}$, Right); $l_2 \leftarrow$ Distance(f_{Left}, v)
    else if $f_{\text{Left}+1} = f_{\text{Apex}}$ **then**
      $\theta \leftarrow$ Angle($f_{\text{Left}+1}$, f_{ApexType}, f_{Left}, Left); $l_2 \leftarrow$ Distance($f_{\text{Left}+1}$, v)
    else
      $\theta \leftarrow$ Angle($f_{\text{Left}+1}$, $Left$, f_{Left}, Left); $l_2 \leftarrow$ Distance($f_{\text{Left}+1}$, v)
    end if
    $l_1 \leftarrow$ Distance(f_{Left}, $f_{\text{Left}+1}$); $v_1 \leftarrow f_{\text{Left}}$
    if $f_{\text{Left}} = f_{\text{Apex}}$ **then**
      $\phi \leftarrow$ Angle(f_{Left}, f_{ApexType}, v, t)
    else
      $\phi \leftarrow$ Angle(f_{Left}, Left, v, t)
    end if
    if CounterclockwiseTo(θ, ϕ) **then**
      f.AddLeft(v); **break**
    end if
    if $f_{\text{Left}} = f_{\text{Apex}} \wedge l_2 < l_1$ **then**
      $\psi \leftarrow$ Angle(f_{Apex}, f_{ApexType}, v, t)
      AddVertex(f_{Apex}, f_{ApexType}, ψ, p)
      AddVertex(v, t, ψ, p)
      f.AddApexLeft(v, t)
    else if $f_{\text{Left}} = f_{\text{Apex}}$ **then**
      $\psi \leftarrow$ Angle(f_{Apex}, f_{ApexType}, $f_{\text{Apex}+1}$, Right)
      AddVertex(f_{Apex}, f_{ApexType}, ψ, p)
      AddVertex($f_{\text{Apex}+1}$, Right, ψ, p)
      f.PopApexLeft()
    end if
    f.PopLeft()
  end loop
else if $t =$ Right **then**
  {same procedure, with directions reversed}
else if $t =$ Point **then**
  $i \leftarrow 0$
  $t_1 \leftarrow f_{\text{ApexType}}$; $t_2 \leftarrow$ Right
  while $f_{\text{Apex}+i} \neq f_{\text{Right}}$ **do**
    if $f_{\text{Apex}+i+1} = f_{\text{Right}}$ **then**
      $t_2 \leftarrow$ Point
    end if
    $\beta \leftarrow$ Angle($f_{\text{Apex}+i}$, t_1, $f_{\text{Apex}+i+1}$, t_2)
    AddVertex($f_{\text{Apex}+i}$, t_1, β, p)
    AddVertex($f_{\text{Apex}+i+1}$, t_2, β, p)
    $t_1 \leftarrow$ Right
    $i \leftarrow i + 1$
  end while
end if

Chapter 5

Triangulation Search

In this chapter, we will first review some basic properties of A* search in Section 5.1 and then discuss what implications it has on searching in a triangulation in Section 5.2. Then we will look at the advantages and disadvantages of a simple approach used in previous work in Section 5.3, followed by considerations required for finding optimal paths in Section 5.4, and finally we will discuss the search used in this work which incorporates these concerns in Section 5.5.

5.1 Introduction to A* Search

Here we introduce some definitions used in search. We start with a state space S. For each state $s \in S$, we have a set $Succ(s) \subset S$ such that $\forall s' \in Succ(s)$, s' can be reached by a single action a from s. These states are called the *children* of s. Additionally, there is a cost associated with this action $c(s, a, s') \geq 0$.

We will also define a state s as the *parent* state of another state s' if and only if s' is a child of s. Also, an *ancestor* of a state s' is any state s such that s is either the parent state of s' or an ancestor to the parent state of s'.

The g-value of a search state, also referenced here as the distance travelled so far, is the cost associated with reaching the current state from the starting state via the states which preceded it. For pathfinding in a triangulation, it is the length of the path from the starting position, reaching some point on the triangle associated with the current search state. We see already some of the uncertainty inherent in searching a triangulation; it is not clear *which* point in the triangle to which this value should correspond. Here we assume it to be the closest point to the start on the edge by which search entered the current triangle. Implications of this are explored further in Section 5.2.

The h-value, or heuristic value of a state, also referenced here as the distance remaining, is an estimate of the distance between the current state and the goal state via the least cost path. We consider this to be the least distance between anywhere on the edge by which the current search state entered the triangle, and the goal position.

The f-value is calculated for a state s as $f(s) = g(s) + h(s)$. This is equivalent to the cost of the entire path between the start and the goal, given that the path between the start state and the current state is given by the states preceding the current state, and the path between the current state and the goal is optimal. It is an underestimate of the distance between the start and the goal positions along the current path.

A* is a widely-used search algorithm which works by using a priority queue which orders search states by f-value. First the start state is put on the queue, and then at each step the

search state on the queue with the least f-value is removed from the queue (or *expanded*), and its children are *generated* by the successor function, and put on the queue.

Since the h-value is an estimate, its value will not be exact. However if this value has certain properties, one can make some claims about the behaviour of A* search when using them. First, it is important that the heuristic is *admissible*, meaning no state's h-value ever overestimates the distance from that state to the goal. That is, $\forall s \in S, h(s) \leq h^*(s)$, where $h^*(s)$ is the true shortest distance between state s and the goal. This guarantees that A* search is complete, and that the first time the goal is expanded, the path along which it was expanded, is optimal [23].

A heuristic may also be *consistent*. This is a version of the *triangle inequality* which says that for any state $s \in S$ and its successor $s' \in Succ(s) \subset S$, $h(s) \leq h(s') + c(s, a, s')$. This means that by moving from one state to another, one cannot get closer to the goal by more than the cost of moving between those two states. A heuristic with this property is both intuitive, and guarantees that the corresponding A* search is optimal, and that whenever any state $s \in S$ is expanded for the first time, the path along which it was expanded, is an optimal path to this state. In addition, all heuristics which are consistent are also admissible [23].

5.2 A* in a Triangulation

The main advantage to pathfinding in a grid world over a triangulation is the fact that in a grid world, as the pathfinding search is being done, the distance travelled to the point corresponding to each search state is known exactly. This is because the object's path is assumed to travel between midpoints of adjacent cells, in straight lines. Since the cells are small relative to the size of the object, this path can be very accurate.

By contrast, when pathfinding in a triangulation, assuming that the object will travel along a simply-defined path, in straight lines between triangle midpoints for instance, is problematic for two reasons. First, in several cases, this path may intersect an obstacle or even cross one or more triangles other than those whose midpoints are being connected, even though they are adjacent. Figure 5.1 illustrates this for such a straight-line path between triangle midpoints.

Second, even when they do not produce invalid paths, usually such simple approximations can form poor estimates in path length as shown in Figure 5.2. This is because the triangles

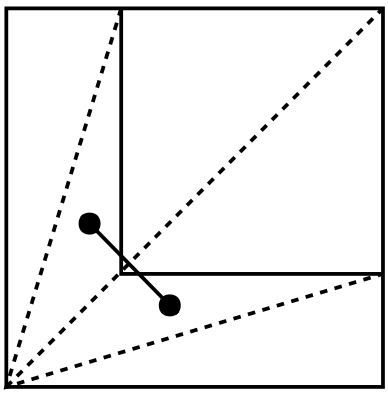

Figure 5.1: A path between midpoints of two adjacent triangles crossing other triangles and being obstructed

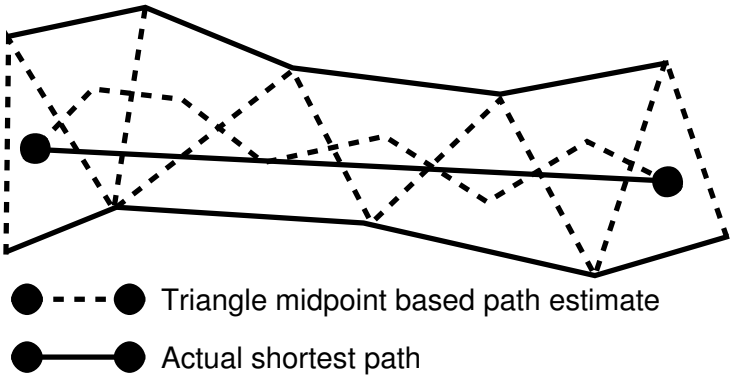

Figure 5.2: A path between triangle midpoints poorly estimating the length of the shortest path through them

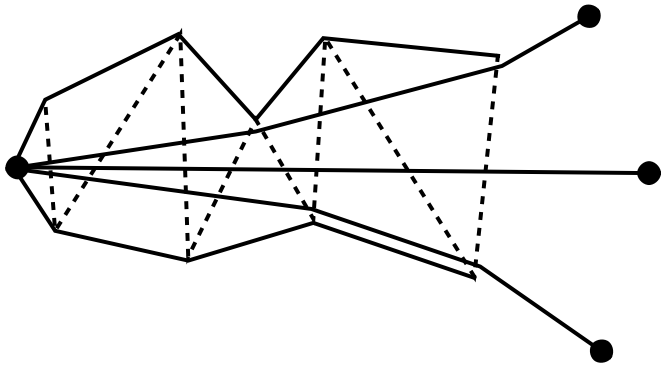

Figure 5.3: The path to a particular triangle depends on where the path continues

are often large in comparison to the size of the object, and because their placement is a result of the environment, generally forming a path between them in such an uninformed manner results in an estimate which does not reflect the actual path of the object.

As shown in Figure 5.3, even if during the search we take advantage of a complex technique such as the funnel algorithm (Section 4.4) or modified funnel algorithm (Section 4.5), we cannot be sure of the exact distance travelled to any particular triangle without knowing where the search will continue. Obviously once we have enough knowledge of the final path to determine this value, it is of little use.

Therefore, we need to perform search in the face of this uncertainty. Next, in Section 5.3, we explore the advantages and disadvantages of methods which deal with this simply by assuming exact knowledge of the path length during search.

5.3 Naïve Search

Despite the drawbacks discussed in Section 5.2, assuming during the search that the exact path length to the triangle associated with the current search state is known does have merit. Such an approach is taken in [27]. Here the path is estimated to be straight segments between the midpoints of each edge through which it travels. The midpoint of the (unconstrained) edge by which the search enters a triangle is considered to be the point by which the

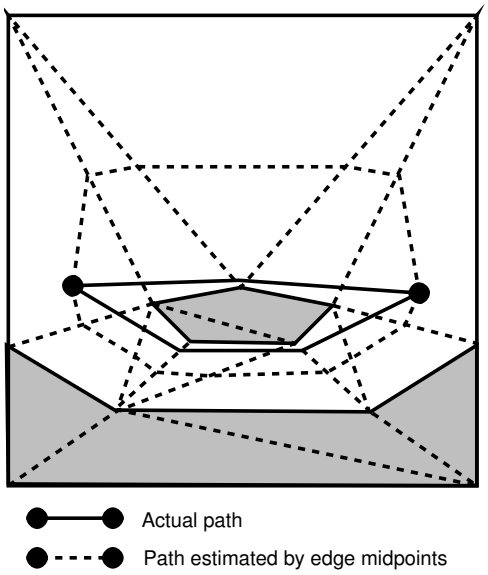

Figure 5.4: A case where the distance estimate results in a suboptimal channel being chosen

measurements are taken for the search state corresponding to that triangle.

The g-value for a search state is, as discussed above, the sum of the lengths of the segments between the midpoints of the edges crossed by that path so far. The h-value is calculated as the Euclidean distance between this point and the goal. In 2 dimensions, the Euclidean distance d between two points p and q is calculated as $d = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}$. d is also the length of a line segment with endpoints p and q.

For pathfinding, the Euclidean distance is known to be both admissible and consistent [36]. Thus, A* search is guaranteed that the path by which the current search state reached its triangle is optimal whenever any triangle is expanded during search, assuming we know the exact g-values of each state. Therefore, each triangle is only expanded once.

Of course since the g-values of each triangle are not known during the search, this isn't necessarily true. Thus, the search will never expand a triangle twice since under normal conditions this could only result in suboptimal paths. Under these conditions, the search may neglect to expand a triangle by a search state whose actual path is optimal but whose estimated distance (here the length of the edge-midpoint path) was greater than that of another search state.

Therefore assuming that the g-values for each search state are exactly known during the search can result in suboptimal paths overall. Such a situation is illustrated in Figure 5.4. Here, the edge-midpoint distance between the start and goal points going through the top of the environment is longer than that going through the bottom below the obstacle. Thus, the channel corresponding to the bottom of the environment is chosen and the funnel algorithm finds the shortest path traversing this channel. However, the globally shortest path goes through the upper portion of the environment, but was not explored due to the search expanding any triangle at most once.

5.4 Accumulated and Approximated Costs

The solution to the suboptimality problem of this approach is first not to eliminate expansion of a triangle during search whether or not a potentially better path to this triangle was found

through another search state. Because we will not know exactly when we have found the best path to a triangle during search, we cannot eliminate any of these paths since it could potentially make up part of an optimal path.

Similarly, we are no longer guaranteed that the path by which the search first expands the goal is optimal. Therefore, finding an optimal path between the start and goal in a triangulation requires that the search continues after the goal is expanded, to consider shorter paths whose estimated distances may have been greater than those of some longer paths.

In this way we wish to search using the *anytime algorithm* paradigm, where an initial, possibly suboptimal solution is found, and search continues finding better and better paths until it converges on an optimal solution. Anytime algorithms are very flexible in that they often find an initial solution quite quickly, and will yield a solution if stopped any time between this point and when it completes the search, which is better the longer the search has been running.

Each time the goal is expanded, the exact length of the path by which is was found can be determined, in this case by a funnel algorithm. This path is retained if it is either the first path found, or if it has a lower cost (less distance) associated with it than the best path it had previously found. Search then continues in this way, finding progressively lower cost solutions, until an optimal path is found, or more often, when it determines that its best path is indeed optimal.

This approach is quite advantageous when working within the constraints of real-time, because for various reasons, a given search may have more or less resources assigned to it, but should still provide a solution. Also, in many cases, such a search actually finds an optimal path long before it determines that the best path it has found is indeed optimal. Therefore, stopping an anytime algorithm before it finishes can still yield an optimal solution.

This raises another concern: how the search can determine when it has found an optimal solution. Basically we desire to know when no path corresponding to the states the search has yet to expand could possibly yield a lower cost solution (shorter path) than the best already found. The cost of the best path found is known exactly, so the search must determine when the other search states correspond to longer paths.

The solution to this problem is to force the g-value to be a lower bound, or underestimate, of its true value, during the search. That is, $\forall s \in S, g(s) \leq g^*(s)$, where $g^*(s)$ is the true distance between the start and state s via the path determined by that state. Then, we have that $\forall s \in S, g(s) \leq g^*(s) \wedge h(s) \leq h^*(s) \Rightarrow f(s) \leq f^*(s)$, where $f^*(s)$ is the shortest path between the start and goal going through the path dictated by the state s.

Now let $s^* \in S$ be the search state corresponding to an optimal path. That is, $\forall s \in S, f^*(s^*) \leq f^*(s)$. Also, let $s' \in Q$ be the state on the front of the search queue $Q \subset S$, or in other words $\forall s \in Q, f(s') \leq f(s)$. Now, if $f(s') \geq f^*(s^*)$, we have that $\forall s \in Q, f^*(s) \geq f(s) \geq f(s') \geq f^*(s^*)$, or equivalently that the actual cost of the best path through all states on the queue are at least as costly as that already found once the f-value of the state at the front of the queue exceeds the actual cost of the best path found.

This extends from the states on the queue to the entire state space since all children of the start state were put on the queue and all paths originating from the start state must pass through its children and so there are no possible alternative paths to be considered. This then gives us the criterion for stopping search with the knowledge that an optimal solution has been found.

Therefore for the search to converge on an optimal solution in spite of such inexact g-values, we must consider multiple paths to the same triangle and continue search after the initial solution is found by way of the anytime algorithm described above. Also, to achieve an effective stopping condition for this search, these g-values must be estimated as a lower bound of the true values. Next we discuss a search algorithm which incorporates

these modifications to search for an optimal path in a triangulation.

5.5 Triangulation A* (TA*)

We develop a search algorithm Triangulation A*, or TA* for short, for pathfinding in a triangulation. It works by first finding in which triangle the start point is located by a technique further detailed in Section 8.1. A search state corresponding to this triangle s_{start} is put on the search queue with values $g(s_{start}) = 0$ and $h(s_{start})$ being the Euclidean distance between the start and goal points.

At each step of the search, the search state with the smallest f-value is taken off the queue and expanded. The successor function generates a child state corresponding to each triangle adjacent to the current triangle across an unconstrained edge. This edge is used in calculation of the g- and h-values for these new states. The h-value is the Euclidean distance between the goal point and the closest point to it on this edge. This heuristic is known to be both admissible and consistent, and these properties are used in calculating the g-value.

The accuracy of the g-value has a considerable impact on the number of extraneous states searched, therefore it is important to estimate it as well as possible, while avoiding an overestimate. Therefore we calculate this estimate as the maximum of a number of known lower bounds, resulting in another lower bound. The lower bound estimates for a state s' with parent state s for an object of radius r, are described below:

- The first and simplest is the distance between the start point and the closest point to it on the entry edge of the corresponding triangle. As with the h-value, this does not overestimate the true value, and it satisfies the triangle inequality in that $g(s) \leq g(s') + c(s, a, s')$.
- The second is $g(s)$ plus the distance between the triangles associated with s and s'. We assume that the g-value of s is a lower bound, and so we wish to add the shortest such distance to achieve another lower bound. Again, we take this measurement using the edges by which the triangles were first reached by search. Since the triangles are adjacent, this is the distance of moving through the triangle associated with s. In Theorem 4.2.12, we proved that the shortest distance between two edges in a triangle was an arc path around the vertex shared by these edges. Thus, if the entry edges of the triangles corresponding to s' and s form an angle θ, this estimate is calculated as $g(s) + r\theta$.
- Another lower bound value for $g(s')$ is $g(s)+(h(s)-h(s'))$, or the parent state's g-value plus the difference between its h-value and that of the child state. This is an underestimate because the Euclidean distance metric used for the heuristic is consistent. To prove that with $g(s') = g(s)+h(s)-h(s')$, $g(s') \leq g^*(s')$, we take that the parent state's g-value is an underestimate, or $g(s) \leq g^*(s)$ to get $g(s') \leq g^*(s)+h(s)-h(s')$, then that the Euclidean distances used for the heuristic is consistent, or $h(s) \leq h(s') + c(s, a, s')$ to get $g(s') \leq g^*(s) + h(s') + c(s, a, s') - h(s')$, or $g(s') \leq g^*(s) + c(s, a, s')$, and since $g^*(s') = g^*(s) + c(s, a, s')$ by definition of the true g-values, we have $g(s') \leq g^*(s')$, as desired.

The maximum of these values often provides a fairly accurate g-value for each state, without overestimating the true value.

As a side note, a child of a search state will not be generated for a particular adjacent triangle if a state corresponding to that triangle is already an ancestor of that state. This exclusion can be done because it will never eliminate an optimal path, only one that could become shorter by removing part of it, as stated in Theorem 4.3.4.

This is important since the fact that we can consider multiple states corresponding to a triangle could otherwise lead to an infinite search space with the search continuing in cycles. This elimination reduces the search space and speeds up the search.

These modifications to the method presented in Section 5.3 provide an algorithm for finding optimal paths in a triangulation. They are fairly minor but allow the comparison between algorithms for finding an optimal path in a triangulation against those for finding one in a grid world, for example A*.

For this reason, TA* provides a base-line for comparison of triangulations to grid worlds as an environment representation. Our predictions in Chapter 3 are confirmed by the results in Chapter 9, where we see that TA* working on the base triangulation finds paths faster on average than PRA*, which works on an abstraction of a grid world based on the same environment.

Chapter 6

Abstraction

While using triangulations as an environment representation provides advantages in pathfinding over grid-world-based methods in speed (as seen in Chapter 9) and accuracy, they also afford possibilities for efficient abstraction. The fact that in a Constrained Triangulation, triangles always extend to touch obstacles in the environment, is useful in reducing an environment to a simplified graph reflecting its topology. A process for achieving such a reduction is described in this chapter.

As introduced in Section 1.4, we desire to partition the environment into a set of useful structures, in this case: decision points, corridors, dead ends, and to a lesser extent, islands. We do this by classifying each triangle as a *node* whose defining characteristic is its *level*, which is a number between 0 and 3 inclusive, indicating the number of structures on the resulting graph to which this node is conceptually adjacent.

6.1 Types of Nodes

An environment represented by a Constrained Triangulation has a graph inherently associated with it, referred to here as the *base-level graph*. The vertices of this graph correspond to the triangles in the triangulation and the edges join vertices whose corresponding triangles are adjacent across an unconstrained edge. Such a graph reflects the topological structure of the environment.

Another graph which we call the *most abstract graph* will be created as a result of the abstraction process. To illustrate the structure of these abstract graphs and to introduce the conventions used in the diagrams in this chapter, see Figure 6.1. This figure has the triangles removed for clarity, these will be added to other diagrams with the established convention of solid lines indicating constrained edges and dotted lines indicating unconstrained edges.

Here one can see various nodes and their classifications. There are some patterns of which to take note: level-0 nodes appear by themselves—these are equivalent to the "islands" mentioned earlier, level-1 nodes form trees and become the "dead ends" of the environment, level-2 nodes appear in chains, forming the "corridors" in the environment, and level-3 nodes appear where three such corridors meet, at a "decision point".

For use with pathfinding, this most abstract graph will be considered the minimal representation of these structures. On this graph, level-3 nodes form the vertices and chains of level-2 nodes the edges. In this way, pathfinding decisions must only be made at points in the environment we have determined are important and moving between them has been reduced to a single step. Level-1 and level-0 nodes disappear as they are only become necessary to the pathfinding task when the start or goal resides in such nodes.

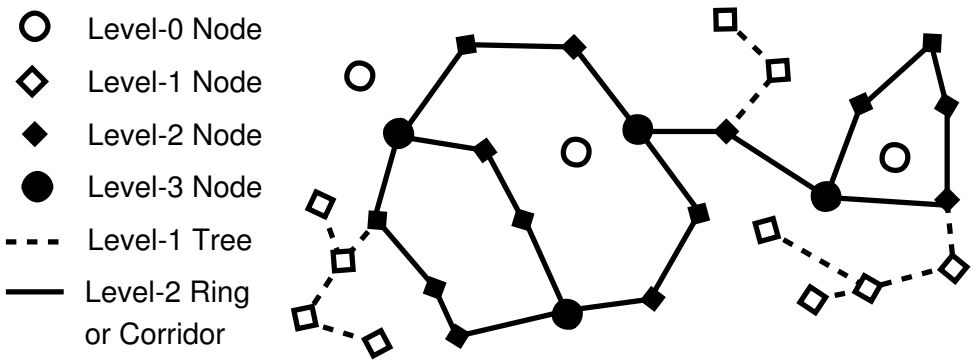

Figure 6.1: An example of an abstract graph

Below, we will give more formal definitions and illustrative descriptions of each kind of node and its role in the environment, pathfinding, and the graph resulting from this process.

6.1.1 Level-0 Nodes

Triangles which have all three edges constrained are classified as level-0 nodes. These are equivalent to the "islands" mentioned earlier. They do not connect to any larger graph structure. If either the start or goal position is within a triangle classified as this type of node, the only possibility for a valid path between them is if they both are in this same triangle.

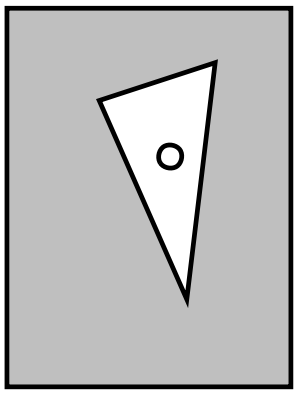

Figure 6.2: A triangle classified as a level-0 node

See Figure 6.2 for an example of a triangle abstracted as a level-0 node. The shaded areas, which indicate obstructed areas, will not be classified here for clarity, although in practice these areas are implicit and would be treated the same as traversable space.

6.1.2 Level-1 Nodes

Level-1 nodes form conceptual "dead ends". The base case for such a node is a triangle with two constrained edges. This is obviously such a dead end because there is only one triangle to which an object can move from it: the one across the single unconstrained edge. However, this only classifies the very corners of the environment.

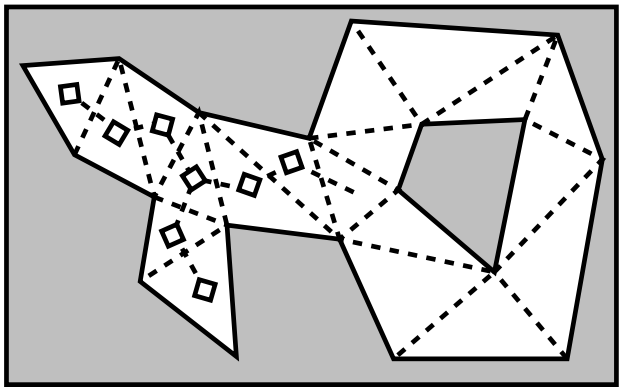

Figure 6.3: A dead-end area classified as level-1 nodes

Therefore we also define level-1 nodes as those which have two or more edges such that either the edge is constrained, or across that edge is a triangle classified as a level-1 node. In this way, the effects of the dead ends propagate to a series of level-1 nodes. Keep in mind that only two edges of a triangle classified as a level-1 node could be constrained, if all three were constrained, the triangle would be classified as a level-0 node. Graphs of adjacent level-1 nodes form trees, which can have at most one "root", which is a level-2 node adjacent to one level-1 node.

If the start or goal is in a triangle classified as this type of node, the goal can either be in another level-1 node in the same tree, or elsewhere. In the former case we can search for a path within this tree, and in the latter, we can move from the triangle containing the start point to the root of this tree and begin searching from there, since we know the goal is not in this tree.

In Figure 6.3, the leftmost and bottom-left triangles are immediate dead ends because they have two constrained edges. These are classified as level-1 nodes. Now the triangles adjacent to them have one constrained edge, and one level-1 node adjacent across an unconstrained edge, and so these become classified as level-1 nodes. This process continues until all the indicated triangles in the diagram are classified as level-1 nodes.

Figure 6.4 shows a traversable region without "floating" obstacles, which are completely inside and not touching the barrier which surrounds the region, and will hence be referred

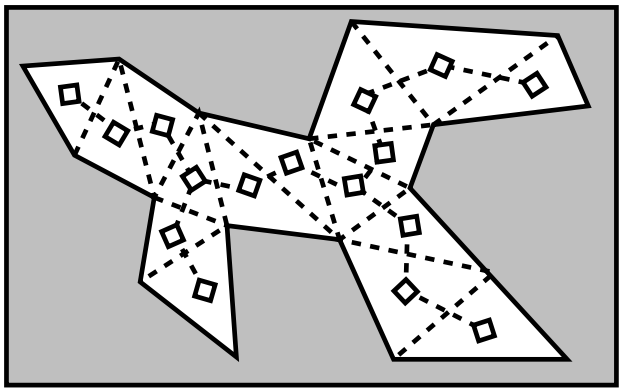

Figure 6.4: An unrooted tree of level-1 nodes

to simply as obstacles. Every triangle in this kind of area will be classified as a level-1 node since the dead ends will propagate through the entire region.

6.1.3 Level-2 Nodes

Triangles classified as level-2 nodes have exactly one edge that is either constrained, or across which is a triangle classified as a level-1 node. Groups of adjacent triangles corresponding to level-2 nodes indicate "corridors" in the environment and form conceptual edges between vertices formed by level-3 "decision point" nodes, or alternately, can form "rings" without level-3 nodes. They can serve as roots for level-1 trees and contain information about the level-3 nodes (if any) at the endpoints on the corridor.

If the start and goal points are on level-2 nodes (or in trees rooted at level-2 nodes), we check if the nodes are on the same corridor. If so, one path can be formed along this corridor, and then further searching outside this corridor must be performed to see if it is the shortest. If they are not on the same edge, search begins on the abstract graph using the level-3 nodes connected to the start and goal.

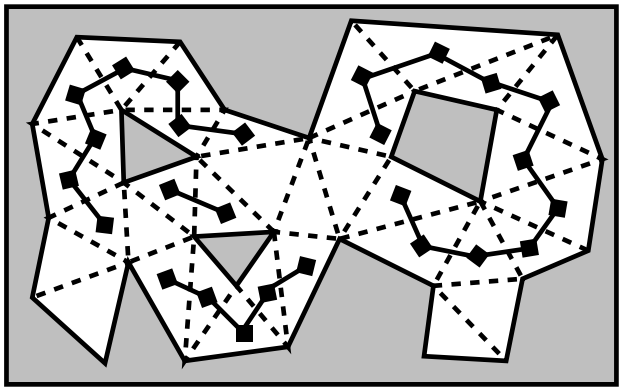

Figure 6.5: Corridors of triangles classified as level-2 nodes

Figure 6.5 shows a triangulated environment with the triangles classified as level-2 nodes indicated. Note that the blank triangles in the dead ends at the bottom left and right of the diagram would be classified as level-1 nodes. The other blank triangles towards the middle of the figure would be level-3 nodes.

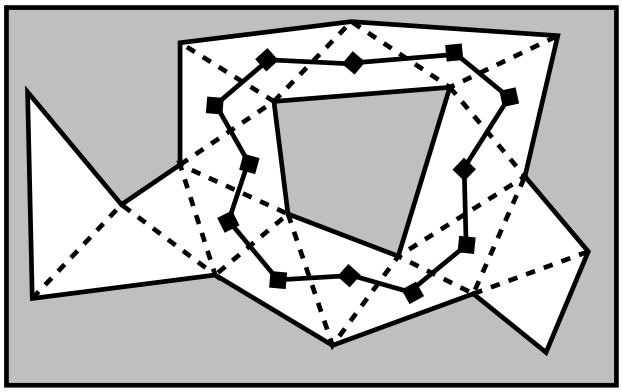

Figure 6.6: A ring of triangles classified as level-2 nodes

While level-2 nodes often form conceptual corridors between two areas of the environment, sometimes a series of adjacent level-2 nodes have no start or end, as is the case in Figure 6.6. This situation occurs when a traversable (or obstructed) area of the environment contains exactly one obstacle. In this case, level-2 nodes form a "ring" around it. Pathfinding in such a component is reduced to deciding which direction around the single obstacle the path should go.

6.1.4 Level-3 Nodes

Triangles classified as level-3 nodes have neither constrained edges, nor adjacent level-1 nodes. Triangles with this important designation represent the "decision points" in the environment, and the vertices of the most abstract graph.

During pathfinding, if either the start or goal point is in a level-3 node or search reaches one as described above, regular search can be done simply between level-3 nodes on the most abstract graph.

The number of level-3 nodes in a connected component of a graph is linearly proportional to the number of obstacles in that component, as we prove below in Theorem 6.1.1.

Theorem 6.1.1 *There are $2n-2$ level-3 nodes in a component with n obstacles.*

Proof Consider the most abstract graph of which the level-3 nodes are the vertices and the level-2 corridors are the edges. The obstacles in the component are surrounded by level-2 corridors, and thus form the faces of the most abstract graph, along with the additional face surrounding the entire graph. In other words, if n is the number of obstacles, $F = n + 1$. Since we know that each level-3 node is incident with three level-2 corridors, and the number of edges is half the sum of degrees of all the vertices, we can calculate the number of edges in this component of the most abstract graph as

$$E = \frac{1}{2}\sum_{i=0}^{V} Degree(V_i) = \frac{1}{2}\sum_{i=0}^{V} 3 = \frac{3V}{2}$$

Then, by Euler's Formula,

$$V - E + F = 2$$
$$V - E = 2 - F$$
$$V - (\frac{3V}{2}) = 2 - F$$
$$2V - 3V = 4 - 2F$$
$$V = 2F - 4$$
$$V = 2(n+1) - 4$$
$$V = 2n - 2$$

Therefore, the number of level-3 nodes in a component of the most abstract graph with n obstacles is $2n-2$. ∎

Figure 6.7 shows which triangles in an environment are classified as level-3 nodes. Notice that there are 4 for this component with 3 obstacles.

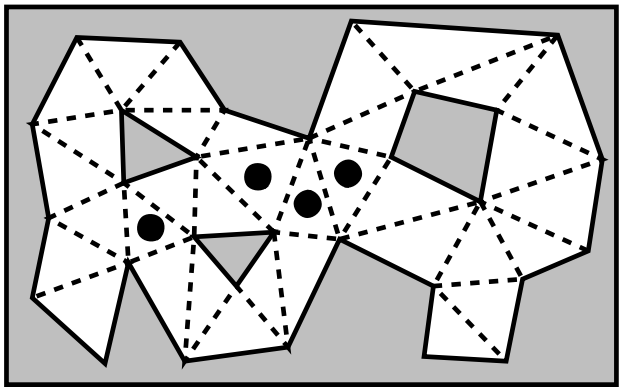

Figure 6.7: Triangles classified as level-3 nodes

6.2 Different Graph Structures

Unlike the base-level graph, the most abstract graph only has vertices with 3 attached edges, hence being formed by level-3 nodes. This eliminates the dead ends formed by level-1 nodes and collapses entire chains of level-2 nodes by simply considering the level-3 nodes across these chains to be adjacent.

The main factor in the size of the search space—the number of vertices in the graph—are significantly reduced as shown in Theorem 6.1.1. Thus, the search is no longer complicated by the nature of the obstacles in the environment, such as shapes of the barriers or concavities therein, only their number.

However, this definition of a graph warrants some discussion, since some structures can result which are atypical in some domains. In this section, we briefly discuss these different structures and their impact on the resulting pathfinding search.

6.2.1 Level-0 Islands

Level-0 nodes form a component of the base-level graph consisting of a single vertex. Technically, they do not exist in the most abstract graph. Obviously if the start or goal of a pathfinding search occurs in a triangle classified as such a node, though, they will nevertheless have to be dealt with. This is covered by one of the special cases of searching the most abstract graph, described in Section 7.1.

6.2.2 Level-1 Trees

If a traversable (or obstructed) component of the environment (also called a *connected component*) contains no obstacles, then the abstracted version of this component will contain only level-1 nodes in a tree, with no level-2 root. Such a situation is shown in Figure 6.4. While this does not provide as much information as having level-2 and level-3 nodes, we can note that in a tree, only one path between any two points exists, excluding those with cycles, which unnecessarily lengthen the path, as stated in Theorem 4.3.4. Therefore, if both the start and goal positions are within this tree, we can use a simple search to find any channel between the two, and it will contain the shortest path. Also one can note that because of the existence of a single acyclic path, the concern for searching any triangle multiple times is not an issue.

6.2.3 Level-2 Rings

If a connected component contains exactly one obstacle, the abstracted graph will contain level-2 nodes forming an edge as a ring with no level-3 endpoints. The result on the most abstract graph is an edge with no start or end point, a case not seen in most graphs. In this instance, when finding a path between start and goal points in different level-2 nodes (or in level-1 trees rooted in different level-2 nodes), the pathfinding task is simplified to checking whether it is shorter to join the points by travelling clockwise or counterclockwise around the ring. Figure 6.6 depicts an environment where the traversable space has a single obstacle around which a level-2 ring forms.

6.2.4 Loops

On the most abstract graph, there can be edges going from a vertex to itself. Such a situation is caused by a single obstacle in one portion of the graph, for example on the right side of Figure 6.8. When searching between level-3 nodes on the most abstract graph, following such an edge would increase the resulting path length and is unnecessary (again, as stated in Theorem 4.3.4). However, it is important when the start or goal is on this type of edge, to consider paths going both directions to get to the level-3 node which forms both endpoints, to reach the most abstract graph and continue searching.

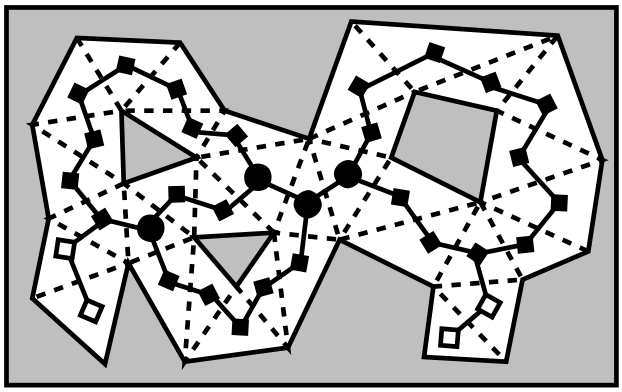

Figure 6.8: A corridor of level-2 nodes both starting and ending at the same level-3 node

6.2.5 Multiply-Connected Nodes

Another possibility is there being two (or even three) edges between the same two vertices. The case with three edges sharing the same endpoints is illustrated in Figure 6.9, while the case with two can be seen in Figure 6.8 with the two leftmost level-3 nodes. The consequences of this arrangement is that search might have to explore paths using both (or all three) edges as part of a channel because the actual path lengths cannot be known until the funnel algorithm is run on the complete channels. In the worst case, this could result in the number of channels to be considered being exponential in the number of obstacles, although a clever search algorithm should be able to prune most of these and still find the shortest.

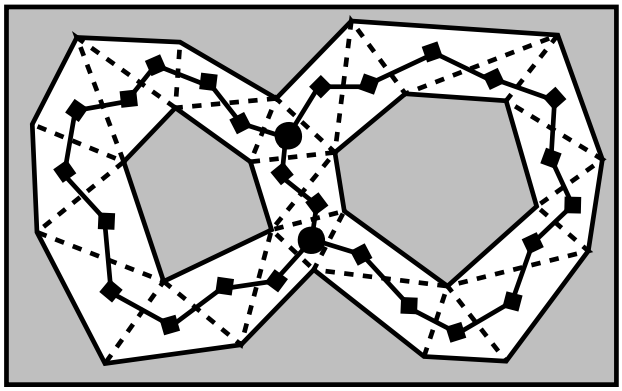

Figure 6.9: Three level-2 corridors sharing the same two level-3 endpoints

6.3 Information Contained

Obviously pathfinding cannot be done very well on the most abstract graph if the only information stored is the level designation for each triangle in the triangulation. This section discusses the information stored for each triangle, and the role it plays in the pathfinding search described in Chapter 7.

6.3.1 Level

The level of the node to which a triangle is classified is the most fundamental characteristic of the triangle. It relates whether the triangle forms an island, resides in a dead end, makes up part of a corridor, or is a decision point. The level of the node determines the first steps of the search, further described in Chapter 7.

6.3.2 Connected Component

Each node contains an index for the connected component of the environment to which it belongs. This is used during search to check, in constant time, if a path can possibly exist between the start and goal points. Obviously if the start and goal reside in different traversable components of the environment, no path between them is possible.

Apart from this convenience, the connected component is necessary to determine whether the start and goal are in the same unrooted level-1 tree, since there is no root to compare, without having to do a complete search. Similar cases exist for level-2 rings and the level-3 search in general.

6.3.3 Adjacent Structures

The purpose of the abstraction is for the search to be able to move out of dead ends without having to find its way out, get from anywhere in a corridor to the connected decision points without needing to take several steps, and skip over corridors between decision points without exploring them. For this to be possible, each node must store the nodes to which it is conceptually adjacent.

For nodes in rooted level-1 trees, this is the level-2 node forming the root of the tree, for those in a level-2 corridor, they are the level-3 nodes at both ends of the corridor, and for level-3 decision point nodes, they are the 3 decision points at the other ends of each level-2

corridor to which it forms one endpoint. Level-0 islands, nodes forming unrooted level-1 trees, and those in level-2 rings do not have any adjacent structures.

Each node stores 3 adjacent structures corresponding to the edges of the triangle. If the path to an adjacent structure is formed by crossing that edge, the value for that edge is set to that structure. This is used in retrieving the triangles to form the channels for calculating the actual length of each path found with the search algorithm.

Because of the possible graph structure discussed in Subsection 6.2.5, two (or even three) of the adjacent structures for a level-3 node could be the same decision point. Also, because of the structure discussed in Subsection 6.2.4, one structure adjacent to a level-3 node could be itself, or both adjacent to a level-2 node could be the same decision point.

6.3.4 Choke Points

On the base-level graph, the search can check for each triangle it traverses, if that triangle's width allows for the object's size to pass through. However, while searching the most abstract graph, such information would not be available while still passing over triangles when moving out of dead ends, corridors, and between decision points in a single step.

Therefore, what is needed is to record, for each structure adjacent to a node, the least width between the triangle associated with this node, and those with the adjacent nodes. This way, before deciding to move to that next structure, the search can know if that entire distance can be traversed by an object of given size. This allows the search to find valid paths for any size of object while still avoiding dealing with individual triangles.

Similarly to adjacent structures, values are stored for nodes in rooted level-1 trees that indicate the diameter of the largest object that can reach the root of the tree. For nodes in a level-2 corridor, these are the size of the largest object that can reach the respective level-3 endpoints, and for level-3 nodes, they are the diameter upper bounds for reaching the decision points that lie at opposite ends of the 3 corridors of which this node is an endpoint. Again, level-0 islands, unrooted level-1 trees, and level-2 rings need no such values.

6.3.5 Triangle Widths

For convenience and to avoid having to calculate them multiple times, each node also stores the widths of its associated triangle as decribed in Section 4.1. One width is stored for each of the three pairs of edges, and corresponds to the diameter of the largest object which can pass between these two edges.

This is calculated and stored even if one (or both) of these edges is constrained, because even though and object cannot pass between these two edges, it may start or end on this triangle, having to pass through its narrowest point (as described in Section 4.4), in which case this value is needed.

6.3.6 Lower Bound Distances

As described in Chapter 5, search requires an estimate of the distance travelled between the start point and the current triangle being expanded, and in our case we want this to be a lower bound. For the search of the base-level graph, we used the maximum of a number of lower bounds. Some were based on global information such as the position of the current triangle with relation to the start and goal points. However, we also measured the shortest distance to get through each triangle.

Similarly to the choke points, we want access to this value for all the triangles in between pairs of adjacent structures, however calculating this information from the individual triangles themselves would eliminate the benefit of moving directly to the next adjacent

structure. Therefore we similarly measure the distance between the current triangle and those to which it is adjacent during the abstraction process, and store them with the nodes.

As with the base-level graph, this is a measurement of the interior angle of all the triangles between which the distance is to be measured. The lower bound distance given by this value, then, is calculated as this total angle multiplied by the radius of the object currently being considered.

6.4 Abstraction Algorithm

In this section, we present an algorithm which can convert a Constrained Triangulation into the most abstract graph while providing the information necessary to perform efficient and accurate pathfinding at that level of abstraction. Furthermore, this algorithm is linear in the number of triangles in the triangulation, which is quite reasonable, especially when coupled with how many fewer triangles there are in a triangulated environment compared to cells in a reasonably-accurate grid. Below, we will walk through the application of this algorithm in general terms, applied to an example environment. Pseudocode for the implementation of this algorithm is given after, in code listings 9, 8, 10, 11, and 12.

Consider the environment in Figure 6.10. There are four traversable components—three on the left consisting of a triangle which will be classified as a Level-0 node, one with no obstacles which will result in an unrooted level-1 tree, and one with a single obstacle which forms a level-2 ring—and one taking up the majority of space and more closely resembling a typical component of an environment.

In the first step of the algorithm, we identify the level-0 nodes as the triangles with all three edges being constrained. In the diagram, this is the traversable component at the bottom left, but would also be the triangular obstacles in the component on the right, however for now we are ignoring these. The component attribute of these triangles are all assigned different values.

Here we also identify the triangles with two constrained edges. These are the most obvious kind of level-1 dead ends. For each of these that are identified, we put the triangle across the unconstrained edge on a queue, since these might now be classified as level-1 nodes.

All other triangles are put on another queue for processing as possible level-2 and level-3 nodes after the level-1 nodes are identified. Figure 6.11 shows the environment after this step. Triangles marked with a “q” are those that are on the first queue awaiting evaluation

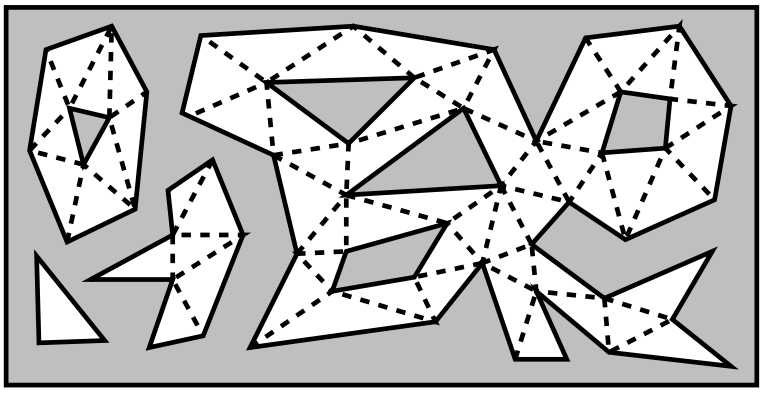

Figure 6.10: An example environment for the abstraction algorithm

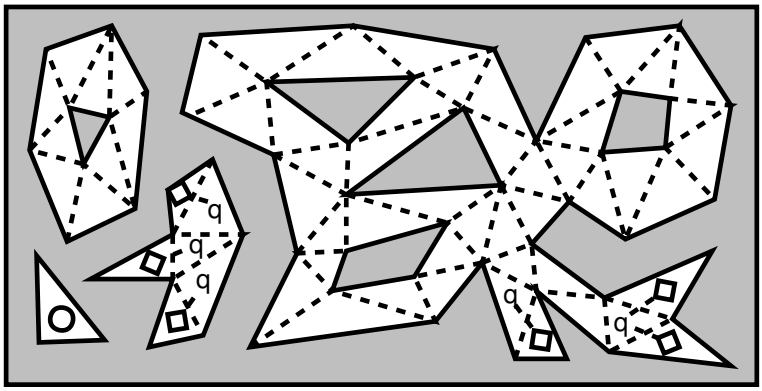

Figure 6.11: The environment after the first step of the algorithm

as possible level-1 nodes.

Then the triangles on the first queue are examined to determine if each is now a level-1 node. If the triangle being checked has two edges such that either the edge is constrained, or across that edge is a level-1 node, then this triangle is a level-1 node and the triangle across from the edge other than the two above is put on the queue for processing. In this way, the effect of the immediate dead ends with two constrained edges, propagates outward, filling the dead end with level-1 nodes as seen in Figure 6.12.

If a triangle being checked has three edges being either constrained or having a level-1 node adjacent, then the whole component is an unrooted level-1 tree. See the traversable component to the right of the level-0 node in Figure 6.12 for an illustration. This component should then be "collapsed", which mainly involves setting the component attribute of all the triangles therein to a unique value, to indicate it is not connected to any other traversable regions.

At this point the first queue is empty and we turn our attention to the second, which we filled with those triangles which had fewer than two constrained edges. These are candidates for level-2 and level-3 nodes. We examine each triangle from this queue which has not yet been classified as a type of node, and determine which are level-3 nodes. With all level-1 nodes identified, the level-3 nodes are those that have neither constrained edges nor adjacent level-1 nodes.

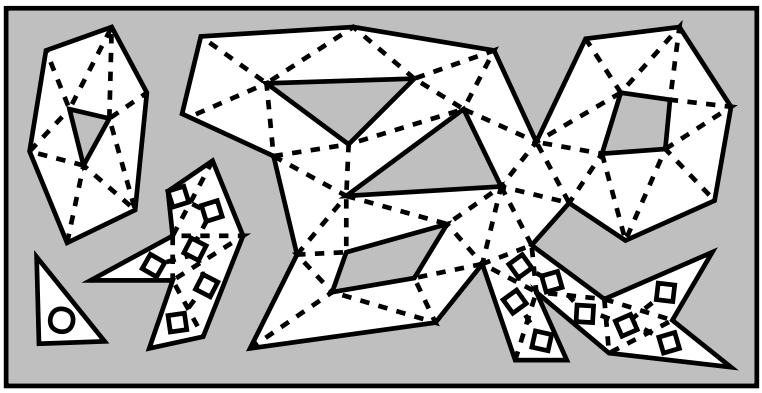

Figure 6.12: The environment after the second step of the algorithm

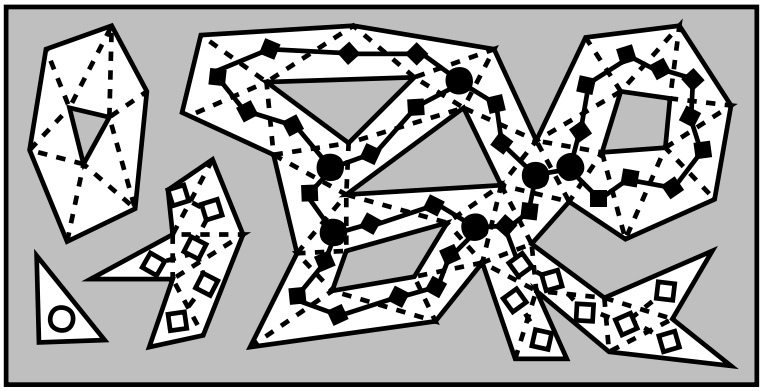

Figure 6.13: The environment after the third step of the algorithm

When a level-3 node is identified, it is put on a stack for processing. Until this stack is empty, we process the level-3 node on top of it by following each of the three corridors for which the current triangle is an endpoint. Until this process reaches another triangle which qualifies as a level-3 node, those in between are classified as level-2 nodes.

The distance along this corridor is accumulated as it is followed, and the choke point is maintained as the narrowest triangle width in the direction of the original level-3 node. These values for the level-2 nodes along the way, and their adjacent structure as the original level-3 node, are set in the appropriate direction. Additionally, all triangles are marked with the identifier for the current component.

Any level-2 nodes which have a level-1 node adjacent (across an unconstrained edge), have this level-1 tree "collapsed" into the corridor. This means performing a stack-based traversal of the tree, setting the component attribute of each triangle to that of the corridor, the choke point to the narrowest point between the current triangle and the root, and the angles to the sum of the interior angles of all the triangles between this one and the root.

Once the next level-3 node is reached along each corridor, it is put on the stack for processing next. In this way, each corridor gets traversed once in each direction, setting the values in the directions of the level-3 nodes at each end. Once the stack is empty, the entire component has been traversed, and a new component identifier can be selected. The processing of the queue then continues, finding and previously unclassified level-3 nodes. Figure 6.13 shows the environment after this technique has been performed on one of the level-3 nodes in the main traversable component.

When this process has gone through all the triangles in the queue, all components with level-3 nodes have been identified. Together with identifying level-0 islands and unrooted level-1 trees, this leaves only one kind of component yet to be processed: level-2 rings.

The queue is processed once more, this time removing triangles as they are visited. Any remaining unclassified triangles must be part of a level-2 ring. Each time one of these is found, a new component identifier is selected and assigned to all triangles in the ring as they are visited one by one. Other information is not necessary as distances and choke points become irrelevant without end points. However, if there are level-1 nodes adjacent to any triangles in a ring, their respective trees are collapsed into the root node on the ring as would be done on a corridor.

After this final process has completed, the result will look like in Figure 6.14. One can see the island, ring, and tree identified on the left side while in the main component, dead ends, corridors, and decision points have been correctly classified. More detailed pseudocode for this algorithm is presented below, with AbstractLevel2 in listing 12 being the main algorithm

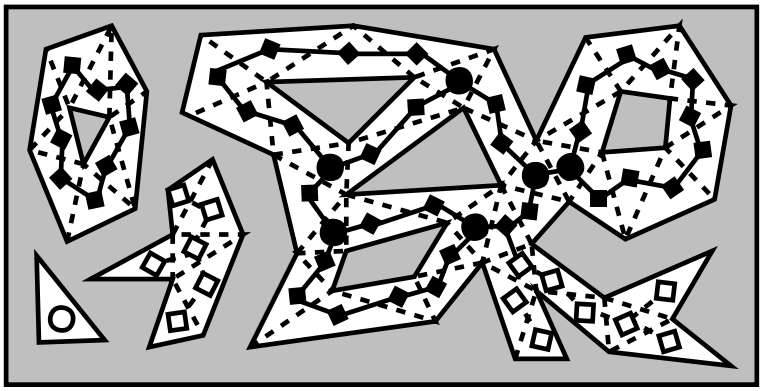

Figure 6.14: The environment after the fourth step of the algorithm

which one would run on a triangulation, which uses the others. Here subscripts refer to the attributes of a particular variable and superscripts are used as part of the variables' names, for readability. Chapter 7 examines how the resulting structure of this most abstract graph is exploited to create a search that is faster and more efficient than one on the base-level graph.

Algorithm 8 CollapseUnrootedTree(Triangle t, Component c)

```
1:  Stack s
2:  s.Push(t)
3:  while ¬s.Empty() do
4:     Triangle t^current ← s.Pop()
5:     t^current_component ← c
6:     for i = 1 to 3 do
7:        Edge e ← GetEdge(t^current, i)
8:        t^current_adjacent_i ← NULL
9:        if e_constrained then
10:          t^current_angle_i ← 0
11:          t^current_choke_i ← 0
12:       else
13:          t^current_angle_i ← ∞
14:          t^current_choke_i ← ∞
15:          Triangle t^next ← GetTriangleAcross(t^current, e)
16:          if t^next_component = NULL then
17:             s.Push(t^next)
18:          end if
19:       end if
20:    end for
21: end while
```

Algorithm 9 CollapseRootedTree(Triangle r, Triangle t)

```
1: c ← r_component
2: Stack s
3: s.Push(t)
4: Stack a
5: a.Push(0)
6: while ¬s.Empty() do
7:    Triangle t^current ← s.Pop()
8:    t^current_component ← c
9:    for i = 1 to 3 do
10:      Edge e ← GetEdge(t^current, i)
11:      Triangle t^last ← TriangleAcross(t^current, e)
12:      if t^last_component = NULL then
13:         t^current_adjacent_i ← r
14:         t^current_angle_i ← a.Pop()
15:         if t_last = r then
16:            t^current_choke_i ← Length(e)
17:         else
18:            for j = 1 to 3 do
19:               if t^last_adjacent_i ≠ NULL then
20:                  Edge e^back ← GetEdge(t^last, j)
21:                  t^current_choke_i ← Minimum(t^last_choke_j, WidthBetween(e, e^back)
22:                  break
23:               end if
24:            end for
25:         end if
26:         Edge e^right ← GetEdge(t^current, (i + 1)%3)
27:         s.Push(TriangleAcross(t^current, e^right))
28:         a.Push(AngleBetween(e, e^right))
29:         Edge e^left ← GetEdge(t^current, (i + 2)%3)
30:         s.Push(TriangleAcross(t^current, e^left))
31:         a.Push(AngleBetween(e, e^left))
32:      else
33:         t^current_adjacent_i ← NULL
34:         if e_constrained then
35:            t^current_choke_i ← 0
36:            t^current_angle_i ← 0
37:         else
38:            t^current_choke_i ← ∞
39:            t^current_angle_i ← ∞
40:         end if
41:      end if
42:   end for
43: end while
```

Algorithm 10 AbstractLevel0and1(Triangulation T, Component c) : Queue

```
1: Queue q, r
2: for all Triangles t ∈ T do
3:    t_level ← NULL
4:    t_component ← NULL
5:    CalculateWidths(t)
6:    n ← NumConstrainedEdges(t)
7:    if n = 3 then
8:       t_level ← 0
9:       t_component ← c
10:      c ← c + 1
11:      for i = 1 to 3 do
12:         t_angle_i ← 0;  t_choke_i ← 0;  t_adjacent_i ← NULL
13:      end for
14:   else if n = 2 then
15:      t_level ← 1
16:      for i = 1 to 3 do
17:         Edge e ← GetEdge(t, i)
18:         if ¬e_constrained then
19:            q.Enqueue(TriangleAcross(t, e))
20:            break
21:         end if
22:      end for
23:   else
24:      r.Enqueue(t)
25:   end if
26: end for
27: while ¬q.Empty() do
28:   Triangle t ← q.Dequeue()
29:   if t_level ≠ NULL then
30:      n ← NumConstrainedEdges(t)
31:      m ← NumAdjacentLevel(t, 1)
32:      if n + m ≥ 2 then
33:         t_level ← 1
34:         for i = 1 to 3 do
35:            Edge e ← GetEdge(t, i)
36:            Triangle t^next ← TriangleAcross(t, e)
37:            if ¬e_constrained ∧ t^next_level = NULL then
38:               q.Enqueue(t^next)
39:            end if
40:         end for
41:      end if
42:      if n + m = 3 then
43:         CollapseUnrootedTree(t, c)
44:         c ← c + 1
45:      end if
46:   end if
47: end while
48: return r
```

Algorithm 11 AbstractLevel3(Triangle t, Component c)

```
1: Queue q
2: q.Enqueue(t)
3: while ¬q.Empty() do
4:    Triangle t^base ← q.Dequeue()
5:    t^base_level ← 3
6:    t^base_component ← c
7:    for i = 1 to 3 do
8:       Edge e ← GetEdge(t^base, i)
9:       ω ← Length(e)
10:      θ ← 0
11:      Triangle t^current ← GetTriangleAcross(t^base, e)
12:      Triangle t^last ← t^base
13:      loop
14:         Triangle t^next ← NULL
15:         n ← NumConstrainedEdges(t^current)
16:         m ← NumAdjacentLevel(t^current, 1)
17:         if n + m = 0 then
18:            if t^current_level = NULL then
19:               q.Enqueue(t^current)
20:            end if
21:            t^base_choke_i ← ω;  t^base_angle_i ← θ;  t^base_adjacent_i ← t^current
22:            break
23:         else if n + m = 1 then
24:            if t^current_level = NULL then
25:               t^current_level ← 2
26:            end if
27:            Edge e^next ← NULL
28:            Edge e^last ← NULL
29:            for j = 1 to 3 do
30:               e ← GetEdge(t^current, j)
31:               Triangle t^temp ← GetTriangleAcross(t^current, e)
32:               if t^temp = t^last then
33:                  t^current_choke_j ← ω;  t^current_angle_j ← θ;  t^current_adjacent_j ← t^base
34:                  e^last ← e
35:               else if ¬e_constrained ∧ t^temp_level ≠ 1 then
36:                  t^next ← t^temp
37:                  e^next ← e
38:               else if ¬e_constrained ∧ t^temp_level = 1 then
39:                  CollapseRootedTree(t^current, t^next)
40:               end if
41:            end for
42:            ω ← Minimum(ω, WidthBetween(e^last, e^next))
43:            θ ← θ+ AngleBetween(e^last, e^next)
44:         end if
45:         t^current ← t^next
46:      end loop
47:   end for
48: end while
```

Algorithm 12 AbstractLevel2(Triangulation T)

```
1: c ← 1
2: Queue q ← AbstractLevel0and1(T, c)
3: for all Triangles t ∈ q do
4:    n ← NumConstrainedEdges(t)
5:    m ← NumAdjacentLevel(t, 1)
6:    if n + m = 0 ∧ t_level = NULL then
7:       AbstractLevel3(t, c)
8:       c ← c + 1
9:    end if
10: end for
11: while ¬q.Empty() do
12:    Triangle t ← q.Dequeue()
13:    if t_level = NULL then
14:       Triangle t^current ← t
15:       while t^current ≠ NULL do
16:          t^current_level ← 2
17:          Triangle t^next ← NULL
18:          for i = 1 to 3 do
19:             Edge e ← GetEdge(t^current, i)
20:             Triangle t^temp ← GetTriangleAcross(u, e)
21:             if e_constrained ∨ t^temp_level = 1 then
22:                if ¬e_constrained then
23:                   CollapseRootedTree(t^current, t^temp)
24:                end if
25:                t^current_angle_i ← 0
26:                t^current_choke_i ← 0
27:                t^current_adjacent_i ← NULL
28:             else
29:                if t^temp_level = NULL then
30:                   t^next ← t^temp
31:                end if
32:                t^current_angle_i ← ∞
33:                t^current_choke_i ← ∞
34:                t^current_adjacent_i ← NULL
35:             end if
36:          end for
37:          t^current ← t^next
38:       end while
39:    end if
40: end while
```

Chapter 7

Abstraction Search

In Chapter 5, we searched the base-level graph, and now, we wish to search the most abstract graph to take advantage of the reduction afforded by the abstraction of the environment from Chapter 6, while still finding an optimal path. This chapter explores the adjustments that must be made to our original TA* algorithm in order to effectively utilize the additional information afforded by the abstracted representation of the environment.

Before even performing a search of the most abstract graph, it is important to first check for a number of special situations that exist because of the structure of the graph and the placement of the start and goal positions. These are explored in Section 7.1 below. If none of these cases are present, then the regular search of the most abstract graph can begin. Section 7.2 deals with the actual search of the most abstract graph and how it differs from our original TA* algorithm.

7.1 Special Cases

The first step of the search is to find in which triangle both the start and the goal points reside. This is in contrast to only finding the surrounding triangle for the start point, which was done in TA* (Section 5.5). Certainly if this process takes excessively long, any advantage gained by searching the most abstract graph would be lost due to the extra point location search. Thus, in Section 8.1 we discuss an improved method for performing this process which increases its speed sufficiently to allow the efficiency of searching the most abstract graph to materialize.

After the triangles in which the start and goal points are encompassed are found, the information from their respective nodes is examined to determine the existence of one of the following special cases. If such a situation exists, often no search is needed, and the shortest path (or lack thereof) can be found even more quickly. These cases, how they are found, and their resulting resolution, are described below.

7.1.1 In Separate Components

When examining the information contained by the nodes corresponding to the start and goal points, the most fundamental requirement for a path to exist between them is that they must be in the same component. If the component indices of the start and goal nodes do not match, there can be no possible path between them, and the search can be halted immediately. If these indices do match, the only possibility for the lack of a path between the start and the goal would be due to the size of the object not being able to fit through certain areas, as a point object could move anywhere in a component.

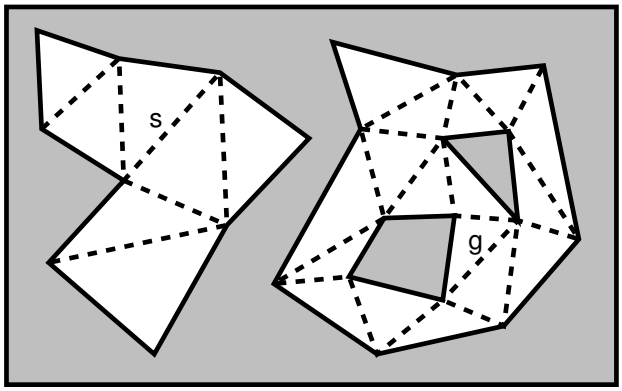

Figure 7.1: The start and goal are in two different connected components

Figure 7.1 shows a case where the start s and goal g are in different components and thus no path exists between them.

7.1.2 On a Level-0 Island

Once we have established that the start and goal are in the same connected component of the environment, finding that one is on a level-0 island implies that the other is as well. Furthermore, all pairs of points in a triangle can be joined by a straight line entirely within the triangle (since triangles are necessarily convex), and so such a path is valid, and since the shortest path between two points is a straight line, it is also optimal.

The only possibility for a path not to exist would be for either the start or goal position to be invalid due to being too close to the triangle's constrained edges. However, this situation should be found at the onset of the search using a technique similar to that of finding the closest obstacle to a vertex in a triangle (Section 4.1) for both the start and goal points to make sure they are valid. However, often we can assume that these points are valid simply from the application. In many cases, this check could be omitted at least for the start point, as the object is usually centered on this point, indicating that it is indeed valid.

This same logic applies when both the start and goal positions are located within the same triangle—unless one or both points are invalid, an optimal path is given simply by a

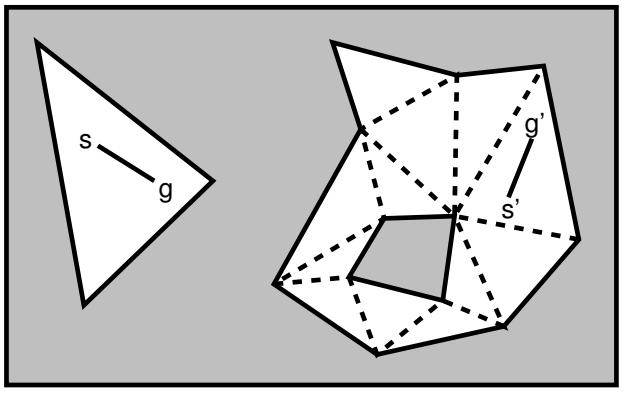

Figure 7.2: The start and goal are in the same level-0 island or otherwise the same triangle

straight line between them. Figure 7.2 shows the start and goal s and g in the same level-0 island, and s' and g' are both in the same triangle. In each case, if the radius of the object was large enough, the goal (and if it was even larger, the start as well) would be invalid and the path would not exist.

7.1.3 From a Tree to the Root

If either the start or the goal is inside a rooted level-1 tree and the other point is in the level-2 node forming the root of the tree, the shortest path between the two can be found immediately. Since the group of level-1 nodes adjacent to the level-2 root form a tree, there is a single channel of triangles between this root and any triangle in the tree, excluding those with cycles, which was stated in Theorem 4.3.4 to unnecessarily lengthen the resulting path.

First the choke point value of the node in the tree is checked to determine if the single channel linking it to the root is wide enough to accommodate the object in question. If it is not, no path exists and the algorithm can be stopped. Otherwise, the channel is constructed out of the triangle in the tree, the one opposite the edge indicated as the one adjacent to the root (see Subsection 6.3.3), and continuing in this manner until reaching the root. A funnel algorithm is then run on this channel, which yields an optimal path between the start and goal points.

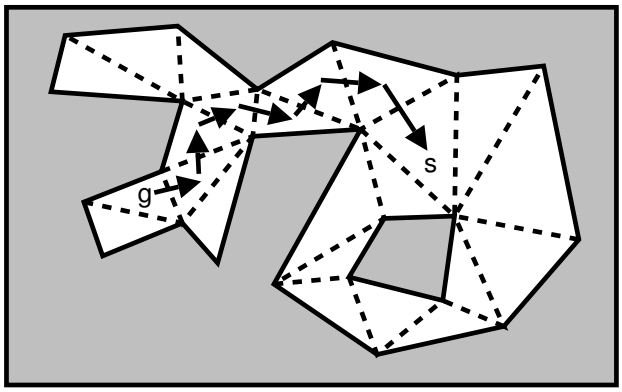

Figure 7.3: The start is the root of a tree containing the goal

Figure 7.3 shows a situation where the goal g is in a level-1 tree of which the level-2 node containing the start s is the root. The channel is constructed by "walking" from the goal to the start through the tree. It is then reversed and used by the funnel algorithm to yield the optimal path between them.

7.1.4 In a Level-1 Tree

If the start and goal point are both on triangles classified as level-1 nodes in an unrooted tree (we know they are in the same tree by the earlier check of the component index), or in a rooted tree with the same level-2 root, another situation exists. As mentioned in Subsection 6.2.2, there is a single channel in this tree which connects the triangles containing the start and goal points and has no cycles, and this channel yields an optimal path between them.

This shortest path between the start and goal can be found by a simple search within this tree, starting at one point and generating adjacent level-1 nodes at each step to avoid leaving the tree unnecessarily. Because of the inherent simplicity of this space, we can take some liberties in the search such as taking the distance measures from the triangle midpoints

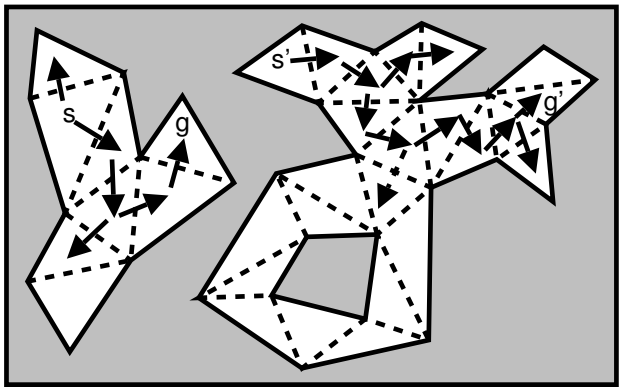

Figure 7.4: The start and goal are in the same level-1 tree

and assuming the distance travelled is the length of the straight segments between them. We can do this because the existence of the single path means that the g- and h-values used in the search are used solely as guides to find the only channel. Again, consideration of any triangle multiple times is not an issue in this simplified search.

Once the channel is found connecting the triangles surrounding the start and goal points, it is checked to make sure the object in question can indeed traverse it. If it cannot, no other channel could, as stated in Theorem 4.3.4, and the algorithm reports the lack of a valid path. If the object can indeed traverse the channel, a funnel algorithm is run on it to determine the actual path for the particular object, which is guaranteed to be optimal.

If the start and goal are not in the same level-1 tree but one or both is in a rooted level-1 tree, then we know there must be a path between the start or goal to the root of their respective trees if a path between them is possible. For either the start or goal, if the node associated with the surrounding triangle is in such a rooted level-1 tree, the choke point indicating the diameter of the largest object which can reach the root, is checked. If this value for either the start or goal is less than the diameter of the object in question, then no valid path exists for this object between the start and goal points since the object could either not get out of the level-1 tree in which the start point resides, or not get into the goal's tree (or both).

Figure 7.4 shows an unrooted level-1 tree on the left, with s and g, as well as the progression of such a search in that component. On the right is a rooted level-1 tree containing s' and g' where this search occurs; notice the dotted arrow indicating that search is not performed outside the tree.

7.1.5 In a Level-2 Loop or Ring

Another situation exists when both the start and goal are each on level-2 nodes or in level-1 trees rooted in level-2 nodes, that are on the same level-2 ring or loop. If they are on a ring, both corresponding level-2 nodes will have no endpoints. They must be on the same ring since they are in the same component, and there can only be a single ring in any component. If they are in the same level-2 loop, both endpoints of each corresponding level-2 node will be the same level-3 node.

In either case, as mentioned in Subsection 6.2.3, there exists two possible channels: going around the obstacle in the center of the ring clockwise, and counterclockwise. Channels are constructed for each, and the funnel algorithm is run on both, yielding an optimal path as the shorter resulting path.

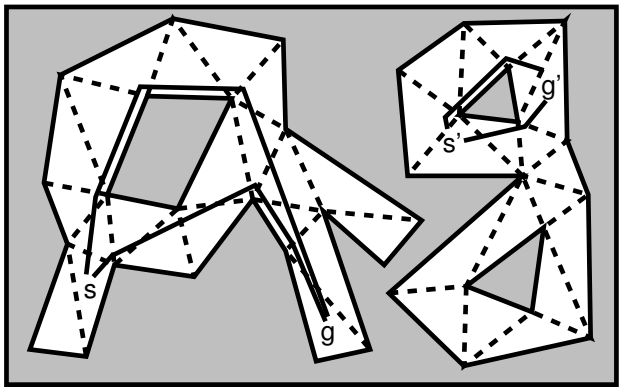

Figure 7.5: The start and goal are on a level-2 ring or loop

If the start point is in a level-1 tree, the method used in Subsection 7.1.3 is used to construct the portion of the channel moving from the start point to the root of the tree. We know from the test performed earlier (Subsection 7.1.4) that this portion of the channel is valid.

The two possibilities for the center portion of the channel are constructed by following the level-2 nodes forming the ring both clockwise and counterclockwise between the level-2 nodes associated with the start and goal points. If, while one of these portions is being constructed, a triangle is encountered through which the object cannot pass, that portion is invalid. If one portion is invalid, the other is the only candidate for a valid path between the start and goal, and if they are both invalid, no valid path exists between them.

Finally, if the goal point is in a level-1 tree, part of the channel is constructed between the triangle in which it resides, and the root of the tree. Again, we know this portion of the channel to be valid for the current object. If both center paths were valid, complete channels are constructed between the start and goal points, going each way around the channel, and the funnel algorithm run on both to determine which is optimal. If one center path was invalid, the other complete channel is constructed and the path calculated by running the funnel algorithm on this channel is optimal.

Figure 7.5 shows a component that forms a level-2 ring on the left, and one that forms two level-2 loops on the right. With s and g, both channels between the start and goal triangles leave the level-1 tree in which the start lies and enter that containing the goal. However, each goes a different direction around the obstacle in the center. The result is that the path calculated from the channel going below the obstacle is shorter than from that going above and thus it is optimal (provided the given object can fit through this channel). Similarly, between s' and g' on the right, the path passing beneath the obstacle at the center of the loop is shorter than that passing above.

7.1.6 On a Level-2 Corridor

If the start and goal are on level-2 nodes, or level-1 nodes in trees whose roots are level-2 nodes, such that they have the same two distinct endpoints, these corresponding level-2 nodes could be on the same level-2 corridor. This is still not certain because multiple level-2 corridors can have the same two endpoints as discussed in Subsection 6.2.5, so this possibility must first be checked.

First the distances from each corresponding level-2 node to one of the level-3 endpoints are checked against each other. If the two are indeed on the same edge, then when moving

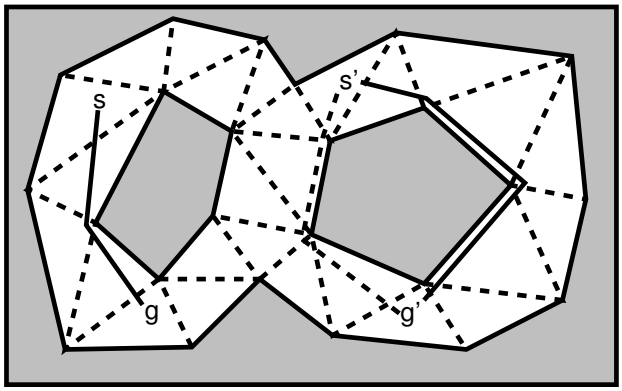

Figure 7.6: The start and goal are on a level-2 corridor

from the one with the larger such distance to this level-3 node, we will pass other other node. This is because the distance to this level-3 node will necessarily decrease along this corridor by the way the abstraction is built.

Thus, we begin constructing a channel starting at the farther level-2 node and working its way to the chosen level-3 endpoint. If this reaches the level-3 endpoint (or the distance of the current triangle being processed from this level-3 node is less than that of the other level-2 node), then the level-2 nodes corresponding to the start and goal points are not on the same level-2 corridor, and the search should proceed as described in Section 7.2.

Otherwise, once this process reaches the other level-2 node, this channel is combined, if necessary, with those which connect the triangles in which the start and goal points reside to the roots of their respective level-1 trees, to form a complete channel between the start and goal points along this corridor.

The funnel algorithm is run on this channel to determine the length of the shortest path through this channel, however this may not be a globally optimal path. Therefore the path found is stored and the anytime algorithm described later in Section 7.2 is run in an attempt to find a shorter path. For this search, the level-3 endpoint of this corridor which is closer to the level-2 node corresponding to the start than that corresponding to the goal is considered the start of the search of the most abstract graph, and the other is considered its goal.

However, as always, if the channel linking the two edges along the corridor is not wide enough to accommodate the object, this path is not considered. Similarly, if the corridor between either of the level-2 nodes associated with the start or goal and the level-3 endpoint in the opposite direction from the other level-2 node is too narrow, then the abstract graph search cannot be done.

Figure 7.6 shows two cases where the start and goal are on the same level-2 corridor. An optimal path between s and g on the left goes through the channel which traverses that corridor, whereas on the right the search of the most abstract graph must be done in order to find an optimal path between s' and g', shown here as a dotted line.

7.2 Triangulation Reduction A* (TRA*)

In this section, we introduce the algorithm Triangulation Reduction A*, or TRA* for short, for searching the most abstract graph using the techniques developed thus far. After this algorithm has started, and none of the special cases listed in Section 7.1 have been found to exist (except that in Subsection 7.1.6, which still necessitates this search, although with

slightly different starting conditions), the actual search must be performed on the abstraction.

While incorporating the same basic principles as the search of the base-level graph described in Chapter 5, there are still a few details to be covered. Here we describe the remaining considerations for this search of the most abstract graph.

7.2.1 Moving onto the Most Abstract Graph

Since we wish to search on the most abstract graph, whose vertices are the level-3 nodes identified by the abstraction, the search must first reach these vertices from the start and goal point. For both the start and the goal, there are three cases which could result in a search of the most abstract graph: the point could be on a level-3 decision point, along a level-2 corridor, or in a rooted level-1 tree. These possibilities are covered below. All other situations are handled in the special cases covered in Section 7.1 and would not result in this search.

The possibilities for both the start and goal can result in either one or two vertices on the most abstract graph. For the start position, this corresponds to the one or two level-3 nodes which would initially be put on the priority queue used by the search. For the goal position, there would be accordingly one or two level-3 nodes at which the search would be considered to have found the goal.

If the point in question resides on a triangle classified as a level-3 node, then there is one start or goal vertex for the search of the most abstract graph, corresponding to that node. If it is on a level-2 node, there are two start or goal vertices corresponding to the level-3 nodes which form the endpoints of the corridor of which this level-2 node is part. Finally, if the point is on a level-1 node, there are again two resulting vertices, corresponding to the level-3 endpoints of the corridor containing the level-2 root of the tree in which the level-1 node lies. Basically this equates to the vertex or vertices on the most abstract graph which are conceptually adjacent to the triangle containing the start or goal point.

For an example of this process, see Figure 7.7. Here, the start point s lies on a level-3 node, so the search queue is initialized with a single state corresponding to this node. The goal point g, on the other hand, is in a level-1 tree. The arrows indicate the "walk" from the triangle surrounding the goal point to the root of its level-1 tree, and then to the level-3 nodes g_1 and g_2 forming the ends of the corridor containing this root.

Another possible situation is shown in Figure 7.8. Here, the start point s is on a level-2 node, so states corresponding to the level-3 endpoints of the corridor (s_1 and s_2) containing

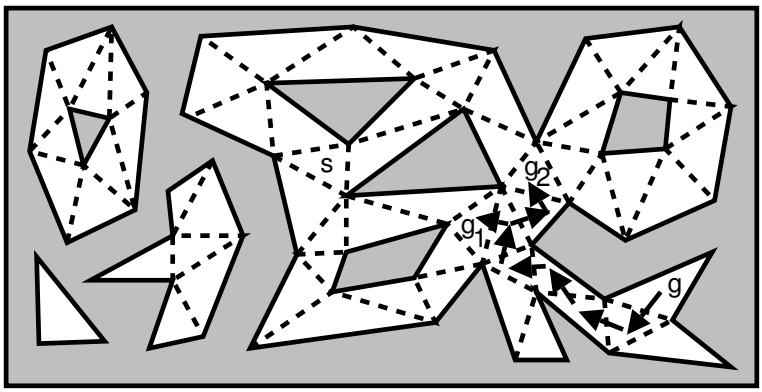

Figure 7.7: Abstract search starts with one state and has two goals

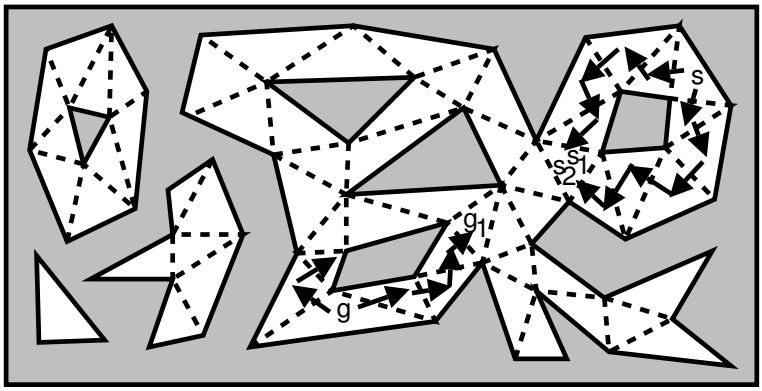

Figure 7.8: Abstract search starts with two states and has one goal

this node are put on the search queue. Even though both endpoints are the same node, they are both added. This is because depending on the rest of the path found, an optimal path might go either direction around the obstacle in the middle of the loop. The goal point g is also on a level-2 node, but only one of its level-3 endpoints (g_1) becomes a possible goal for the search of the most abstract graph. This is because we assume the object to be of such size that it cannot pass through one of the triangles to the left.

7.2.2 Accumulated Distance Measures

In addition, when the starting point is inside a triangle classified as a level-1 or level-2 node, the distance between the start point and the level-3 nodes which initialize the search is included in the g-values of the search states corresponding to those nodes.

For a start point on a level-3 node, there is no additional associated distance. For one on a level-2 node, it is the distance between the corresponding triangle and the ends of the corridor, as stored in the node itself. For one on a level-1 node, it is the distance associated with reaching the root of the tree as stored in the node, together with the distance from the root to the adjacent level-3 nodes similar to above, plus the distance of travelling through the root node, which is not incorporated in either measurement.

For simplicity, this was not done for the paths between the goal point and its adjacent level-3 nodes, thus the measurements for the heuristic were still taken between the triangles corresponding to each state, and the goal point itself. Alternately if this distance was considered for the level-3 nodes adjacent to the goal point, the heuristic for each state could be that to the closer of the two goal vertices, plus the distance from that vertex to the goal.

7.2.3 Checking Channel Widths

Similar to how the distances are considered, the widths must also be checked when the start or goal point is in a level-2 node. This is done by checking the choke point value for the section of the corridor between the current node and the level-3 node at each end. If the corridor is too narrow for the object to get to one end, the abstract graph search only considers the vertex associated with the node at the other. If the corridor leading to both ends is too narrow, the search fails (or if the case in Subsection 7.1.6 found an initial path, that one is returned).

We know from checking before that if the start or goal point is in a level-1 tree, the width of the channel connecting the triangle around that point to the root of its tree, is enough to accommodate the given object. The width between the level-2 root of this tree

and its adjacent level-3 nodes is checked in the same way as above, except that the width must also be checked through the root node between the edge connecting the root to the attached tree, and that connecting it to the rest of the corridor across which is the level-3 node in question.

7.2.4 Corridor Lengths and Choke Points

Searching the most abstract graph differs from searching the base-level graph in that information regarding the g- and h-values and triangle width cannot be taken directly from the individual triangles without nullifying the benefit of the reduced search space. Therefore this value must be extrapolated from the information calculated during the abstraction process.

For any level-3 node generated by TRA*, its heuristic is calculated as before, as with the distance travelled measurements relying on global information such as the placement of the start and goal positions, but the one that increments the value of its parent state must take advantage of the abstraction information. This value is determined by adding that of the parent state, to a measure of the distance to get from the entry edge of its triangle to that associated with the current state.

As with TA*, this is done by multiplying the interior angle by the radius of the object in question, however, because we are now traversing an entire corridor, we use the accumulated interior angle of the corridor as stored in the abstraction. On top of this, we add the interior angle of the parent state's triangle, to add the distance for moving from it's entry edge to the start of the corridor. This combined angle is then multiplied by the radius of the object and summed with the distance value of the parent state.

Similarly the search must take advantage of the information provided in the abstraction when determining which channels can yield a valid path for the object. Since we have already determined whether there are valid paths for this object between the start and goal points and their adjacent level-3 nodes, it remains to determine if the object can move through and between the level-3 nodes searched.

The abstraction provides the width of the narrowest point along each corridor, so as long as this value is at least the diameter of the object for which we are finding a path, the channel traversing that corridor will yield a valid path. However, we must also check that the object can move between the edges of each level-3 node which connect it to the corridors being traversed. For this, the appropriate width of the level-3 node is checked.

If either of these checks fail, the search will not consider searching the level-3 node at the opposite end of that corridor from this node. This guarantees that for any state in the search, there is a valid path between the start point and the triangle associated with that state, passing through the level-3 nodes specified by that state's ancestors.

7.2.5 Searching the Most Abstract Graph

In other respects, the search proceeds similarly to TA* described in Chapter 5 by considering any triangle multiple times, having the same anytime algorithm paradigm, and using the same g- and h-values, with slight modification as described above. The key difference is that the states in TRA* correspond to level-3 nodes, or vertices of the most abstract graph, whereas the states in TA* correspond to individual triangles, or vertices of the base-level graph.

In Chapter 9, we explore the speed of TRA* as well as TA* when compared with the standard A* and PRA* searches performed on a grid-world representation of the same environment. We also explore the behaviour of the anytime algorithms in how they converge on optimal paths, and the times required for the preprocessing of the environments tested,

including the initial triangulation, the abstraction thereof, and the computation associated with the improved point location described in Chapter 8. This chapter also describes a technique we applied to both TA* and TRA* to make the performance more predictable and suitable for the anytime algorithm paradigm and the application areas for which they were designed.

Chapter 8

Other Enhancements

This chapter briefly describes two enhancements added to both TA* and TRA* which are tangent to the main work of this thesis. The first such enhancement is a faster method of point location in the triangulation introduced to keep the benefits of the triangulation-based searches from being lost to a slower version of this task, and is described in Section 8.1. The second is a minor modification made to the anytime algorithm associated with both searches which decreases the time between the start of the search and when the first solution is found, while at the same time retaining the convergence on an optimal solution, presented in Section 8.2.

8.1 Sector-Based Point Location

To search for a path between two points in a triangulation, the first task is to find the triangle in which one of the points is contained, and then search for a path to the other. This first process is called point location. Point location is typically done in a triangulation by "walking" from some start triangle towards the desired point [19]. See Figure 8.1 for an example; here, the grey point is found by a walk indicated by the arrows, starting at the triangle in the upper-left corner of the triangulation.

This can be done, for example, by checking each edge of a triangle if the point lies to the left, when traversing these edges in counterclockwise order. If it lies to this side of all three edges (or along one edge), then the point is inside the current triangle. If the point is outside one of the edges, the triangle opposite that edge can be checked next. In this way, the algorithm gets progressively closer to the point in question.

However, in [27], the point location was done starting at a fixed triangle, similar to in Figure 8.1. On average, then, this process had to "walk" halfway across the triangulation to find each point for which this process was done. Not surprisingly, this was determined to take a large portion, if not the majority of the time required by the pathfinding algorithm.

This delay threatened to overshadow the benefits TA* received from using the triangulation, and also those that TRA* received from using the most abstract graph constructed from it, since this second algorithm requires that both the start and goal points be located in this way, instead of just one. Because we wanted our triangulation-based methods to be competitive with grid-based methods, an improved point location technique was in order.

There are a number of methods which offer improved performance: the triangulation refinement method [31], the chain method [20], persistent search trees [42], persistency using similar lists [15], and a randomized incremental method [40]. These techniques all find points in $O(n \log n)$ time and require $O(n)$ memory for the structures associated with

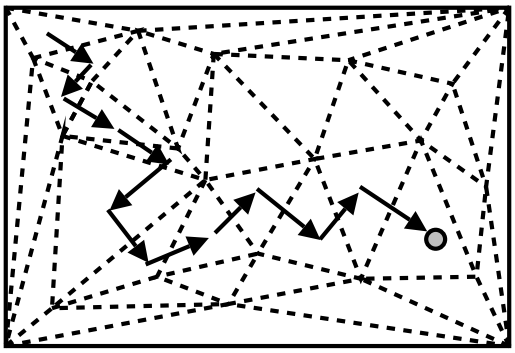

Figure 8.1: Point location from a fixed triangle

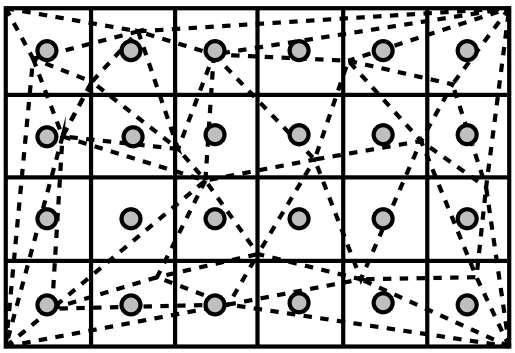

Figure 8.2: Decomposition of the environment into sectors, and their midpoints

them. However, these techniques would be difficult to update in the presence of changes in the triangulation, and thus were not desirable.

While methods are available that deal with the changes possible in a representation such as a DCDT, for example those summarized in [14], we desired a simple solution which was both easily updated in the event of repairs to the environment representation, and did not require complex structures to maintain. For this reason, we developed another method for improved point location, which we detail below.

Our redesigned method consists of conceptually decomposing the environment into *sectors*, in our case, forming a grid of rectangles over the environment, because points are defined by their horizontal and vertical coordinates, thus allowing the corresponding sector for a point to be calculated easily in constant time. Figure 8.2 shows the same environment as in Figure 8.1 with such sectors overlaid on it. Midpoints for these sectors can be calculated easily as well, and appear in Figure 8.2 as grey circles.

When preprocessing of an environment, such as the abstraction process described in Chapter 6, is being done, each triangle visited is checked to see if it overlaps any sector midpoints. This is done by taking the bounding rectangle for the triangle and checking each of the sector midpoints it overlaps, if they are within this triangle by using the same method as described earlier. When the triangle enclosing each sector midpoint is determined, the pointer corresponding to that sector is set to this triangle. These pointers are stored in a two-dimensional rectangular array to allow constant time access.

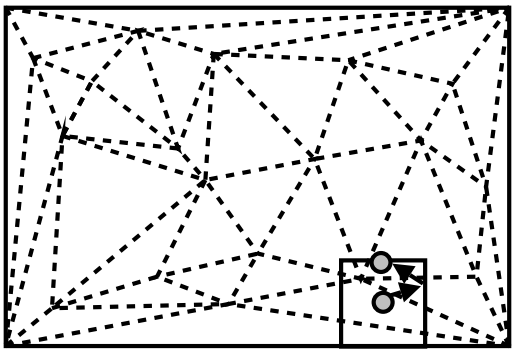

Figure 8.3: Sector-based point location

Now, for each point for which the location process is to be performed, the sector in which it is contained is calculated by its coordinates. With this information, the triangle enclosing the sector midpoint is retrieved from the array of these values. Then, the same "walk" process [19] is started, but from this triangle, resulting in a much shorter distance to the point and fewer triangles visited. An example of this is shown in Figure 8.3, with the active sector outlined and the midpoint shown, and the point being located indicated by another grey circle. Notice how many fewer steps are required to find such a point using this process.

Indeed, the "walk" is reduced in expected length to half the dimension of a sector. This results in enough of a performance gain to allow the benefits of TA* and TRA* to become apparent. For the experiments presented in Chapter 9, a modest 10×10 grid of sectors was used on each environment. As shown in that chapter, the preprocessing time associated with this technique is almost negligible, and the execution times resulting from TA* and TRA* are impressive, despite performing point location once and twice, respectively, for each path tested.

One can note that the repair of this array of pointers to triangles surrounding sector midpoints can be easily repaired if the triangulation changes using the mechanisms described in Section 3.4. Each triangle that was changed or added is simply checked, as all others, if it overlaps any sector midpoints, and the pointers are set accordingly. The time required for such a repair is minimal, as with the initial preprocessing. If, for some reason, there is no triangle corresponding to the midpoint of the sector containing the point for which this process is being performed, the point location "walk" can simply be started from some fixed triangle as before.

8.2 Finding a First Solution Quickly

During an initial run of the experiments presented in Chapter 9, some cases appeared that would be unacceptable for application in real time, despite that the vast majority of solutions were found very quickly. In certain situations such as that shown in Figure 8.4, the fact that any triangle in the case of TA*, or level-3 node in the case of TRA*, can be considered multiple times during the search, creates a weakness for these algorithms which extends their runtimes beyond even those of the standard A* algorithm.

The Euclidean distance heuristic draws these searches into a kind of "canyon", at which point it gets caught up in the many triangles or level-3 nodes therein. The many small obstacles in the canyon pose a problem to both TA* and TRA*. For TA*, these obstacles

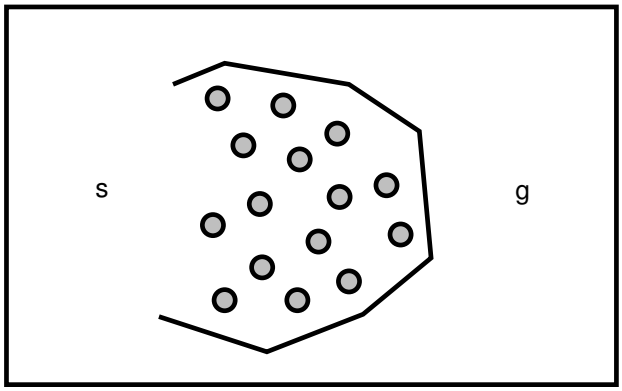

Figure 8.4: Situation which hinders TA* and TRA* searches

increase the number of triangles in this region, which increase the state space for this algorithm considerably. For both TA* and TRA*, these obstacles increase the number of level-3 nodes in the most abstract graph. TRA* searches these nodes directly, so this increases the size of its state space, but also for TA*, these points allowed the search to overlap, so each one increased the number of times the triangles in the canyon could be considered.

Because both TA* and TRA* will attempt paths going each direction around these obstacles, the number of states searched will increase considerably. In the case of such small obstacles, the distance measures, in avoiding an overestimate, will not increase by much with each state searched. This causes many combinations of paths to be attempted inside the canyon before these values finally increase enough for the search to venture out and find a correct path.

In general, small obstacles pose a problem for these techniques. Their consideration of triangles or level-3 nodes multiple times results in an exponential number of potential paths, and the distance measures are required to prune the majority of these. However, when the obstacles are small these distance measures can only increase slightly at each step, causing many more combinations of these paths to be followed than would be with larger obstacles.

Interestingly, the abstraction technique harnessed by TRA* would simply disregard such an enclosed area as a dead end if it did not contain obstacles. In this case, only abstracting the environment to the point where it does not lose information about its topology, is not enough to provide an efficient mechanism for dealing with such situations.

There were several possibilities for dealing with this drawback, however many simply would reduce the effect of very specific such situations instead of addressing the overall problem. A pathfinding system which completely solves this problem remains an open question, however there are several possibilities for further abstraction of the environment beyond the most abstract graph (losing some information about the topological structure of the environment and thus unable to guarantee optimal solutions) discussed in Section 10.2.

For now, the focus was put on the anytime algorithm aspect of these searches, since the spirit of such an algorithm is to return an initial path quickly and then converge on an optimal solution. However, in doing so, we did not want to sacrifice the ability of these algorithms to guarantee such convergence to optimal like many approaches would. Therefore a simple adjustment was made to prevent them from searching any triangle or level-3 node multiple times until the first solution is found.

This was done simply by checking, until the first solution is found, if the triangle associated with each state about to be expanded by the search has been marked as previously

expanded. If so, the search state is put on a secondary search queue instead of being expanded, and if not, the state is expanded as usual and the associated triangle is marked as having been expanded. When the first solution is found, all the states in the secondary queue are put back in the primary queue and search continues as normal without this extra check at each state expansion.

This modification does not change the fundamental aspects of the search, only decreases the time required by both algorithms to find the initial solution. In this respect, TA* and TRA* become more useful for real-time applications, lengthening the window of time in which a solution is available, producing an initial solution in even the degenerate cases mentioned above, early enough for such time requirements. Of equal importance is the fact that this technique does not affect the production of subsequent solutions, and thus still guarantees the convergence of both searches to an optimal solution.

In Chapter 9, the paths for which these algorithms could not find the optimal path in the time allotted, can be attributed to this type of exceptional situation. Since the searches were stopped after a multiple of the time required to find their first solution, these cases will not yield an optimal path since the first solution is found so quickly, and an optimal one takes so long. These mostly occurred in the environments taken from WarCraft III maps, which had many trees consisting of single tiles, and often resulted in the small obstacle pathology mentioned above.

While only a small fraction of the paths tested resulted in such a situation, and TRA* still being able to find such optimal paths eventually, it is important to develop solutions for such cases when the target application works in real time. An occasional suboptimal solution is far more desirable, in these conditions, than one that takes excessively long or returns no solution at all.

Chapter 9

Experiments

This chapter covers the experiments done to validate and measure the benefit of searching both the base-level graph (TA*) and the most abstract graph (TRA*) against both standard (A*) and abstracted (PRA*) grid-based techniques.

Section 9.1 discusses the details of the experiments, including both how they were done in the previous paper and how they were modified so that the techniques presented in this thesis could be meaningfully compared to the others. The results given by the experiments performed on our techniques are given in Section 9.2, and compared to the previous results. The immediate implications of these results are also discussed in that section. Following this, Section 9.3 further explores what these results could mean for further work.

9.1 Experimental Setup

For the experiments, we desired to test both TA* and TRA* against the benchmark A* and recent PRA* presented in [47]. Thus, the setup for these experiments is identical to that in this paper to allow a direct comparison of these methods. 116 environments were included in the experiment: 75 maps from the game Baldur's Gate and 41 from WarCraft III, which were selected as being interesting for having sufficient size and complexity.

These environments were particularly useful in that they come from successful commercial games, and therefore provided a unique opportunity to test these methods for the very application for which they were designed. Providing sufficient performance to allow pathfinding effectively in real-time in such environments would certainly indicate that the methods developed in this thesis successfully accomplished their goal.

The Baldur's Gate maps consisted of a grid of tiles, each marked either traversable or obstructed. Each of these maps were scaled to 512×512 tiles without changing the connectivity of the resulting graph. To convert these into a polygonal representation which can be triangulated, each traversable region is traced by placing line segments between pairs of traversable and obstructed tiles, in sequence to form a polygon around them. Similarly, obstacles are added to the representation. In this way, constrained edges of the triangulation equate to barriers between traversable and obstructed space in the environment. All paths were found between two tiles in the same component of traversable space.

The WarCraft III maps were slightly different in that each tile had both a type of terrain and a height value associated with it. Any tile could have a type indicating it was grass, swamp, water, a tree, or outside the play area. Furthermore, a tile could be a ramp which allowed paths to cross a height difference. These maps were also scaled to 512×512 tiles while maintaining their structure. Similarly to the Baldur's Gate maps, each component of a different terrain type was traced by line segments forming a polygon around it. For this

purpose, grass and swamp tiles were considered the same type so paths could cross between their respective tiles. After that, line segments were added between tiles of different height values where the associated tiles were not ramps. In this way, tile type boundaries and "cliffs", or abrupt height differences with no ramps, could not be crossed by paths. Paths in these maps were always within components of the same tile type (paths between two positions in water were allowed, for example), with at least one valid path between the start and goal (not separated by a cliff).

It is interesting to note that while the borders of the tiles were followed exactly during the conversion from tile-based to polygonal environments, often it was apparent that the tiles formed an inaccurate representation of the actual structure of the environment. For example, there were several "staircase" patterns in the tiles, which if originating in a polygonal representations could be represented by a single diagonal line. This would significantly reduce the number of triangles in the triangulation while at the same time improving its accuracy. Similarly if the tiles were arranged to approximate a curved barrier in the environment, this could be approximated more accurately in a polygonal representation and result in fewer triangles in the resulting triangulation.

If extrapolating these structures was done, TA* would receive a significant performance increase due to the reduction of the number of triangles in the triangulation, which in itself is already low compared to the number of cells in the corresponding grid. TRA* would remain unchanged for the most part, since such detail would not affect the number of vertices in the most abstract graph which it searches, only the distance measures slightly.

On each map, pairs of points are used where the optimal path between them on the grid is between 0 and 511 tile lengths, inclusive. Each pair is put into one of 128 buckets numbered 0 to 127 based on this length. In particular, a pair of points whose optimal path on a grid is l tile lengths, is put into bucket i, where $i = \lfloor l/4 \rfloor$. 10 pairs of points were generated for each bucket on each map, resulting in 1280 paths per map, or 1160 paths per bucket, and 148480 paths overall.

For the results showing percentiles, the data point at any path length relates the data in the corresponding bucket. A data point for the n^{th} percentile is the value of the $\lfloor n/100 \times 1160 \rfloor^{th}$ value, typically ordered from best to worst. Standard deviations were not used because it was not certain that the data was normally distributed.

For the experiments involving TA* and TRA*, the radius of the object was taken to be just less than half the length of a tile, so that the object would have to stay in the centers of tiles unless moving through free space, keeping it from having too much advantage over the grid-based methods in terms of path length. It was kept just short of half the width to avoid floating-point calculation errors in the modified funnel algorithm, and the determination of the object's traversability through triangles or corridors.

In the experiments using the grid-based methods A* and PRA*, an object could move to any of its 8-neighbours which were unmarked. However, cutting corners was disallowed, that is, moving diagonally across a 2×2 block of times was only permitted if none of the four tiles were obstructed.

The experiments were done on a computer with a Socket 754 AMD Athlon 64 3200+ processor and 2 sticks of 512 MB PC3200 CAS2-3-3-6 RAM and a motherboard with an NVIDIA nForce3 250 chipset. The code was created and compiled in Microsoft Visual Studio .NET 2003, and single-threaded.

To avoid having to re-implement the techniques used in [47], we simply took the exact paths tested there for these experiments to compare the results directly. These results were also single-threaded, and run on a dual-CPU Power-Mac running at 2 GHz with 1 GB of RAM and compiled in gcc 3.3. To compensate for the difference in system speeds between the two result sets, the running times for the original results were halved to facilitate more accurate comparison.

9.2 Results

Here we will show and discuss the results found in the experiments described earlier. The first thing to notice is the size and complexity of the environments in question. Since all environments have been scaled to 512 × 512 tiles, there are 262,144 cells in the grid used by both A* and PRA* for all 116 maps tested.

However, when the tracing operation is performed on the components of the environment to convert it to a polygonal representation, it is significantly simplified. Consider the "constraints" line in Figure 9.1. Here the maps were ordered by the number of triangles in the corresponding triangulation and given indices to arrange them from left to right.

There are more than 20 times more cells in a grid than there are constraints in the polygonal representation of even the most complex environment, and for many environments, this factor increases to above 100. This is a testament to the efficiency of polygonal representations over grid worlds, especially when one considers that often there would be far fewer still had the original environment been represented this way, instead of forcing diagonal and curved barriers to be "rasterized" to be represented by a grid.

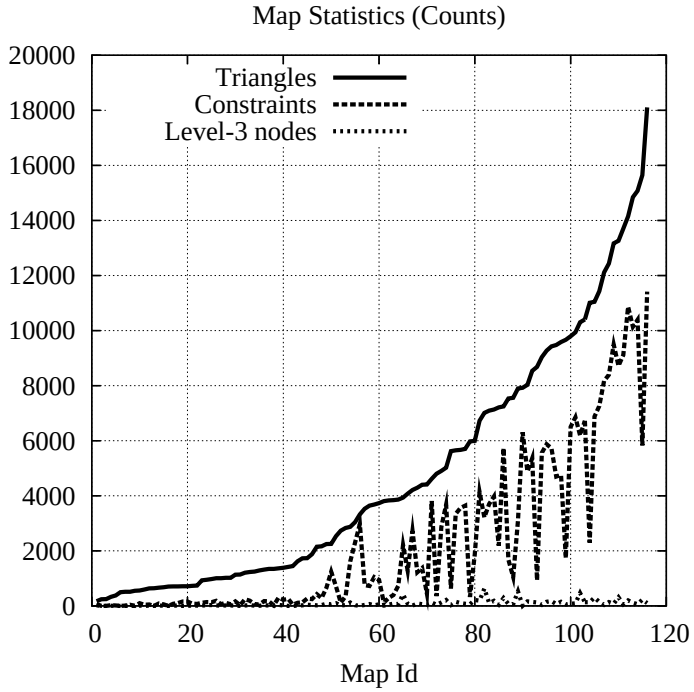

Figure 9.1: Triangles, contraints, and level-3 nodes in environments tested

Now, consider the "triangles" line in the figure. Again, there are far fewer cells in a triangulation than a grid, with the grid-world representation having over 10 times more cells than the most complex triangulation, and over 65 times more than the median. Since this reflects the base-level graph searched by TA*, we can see already that it offers a significant

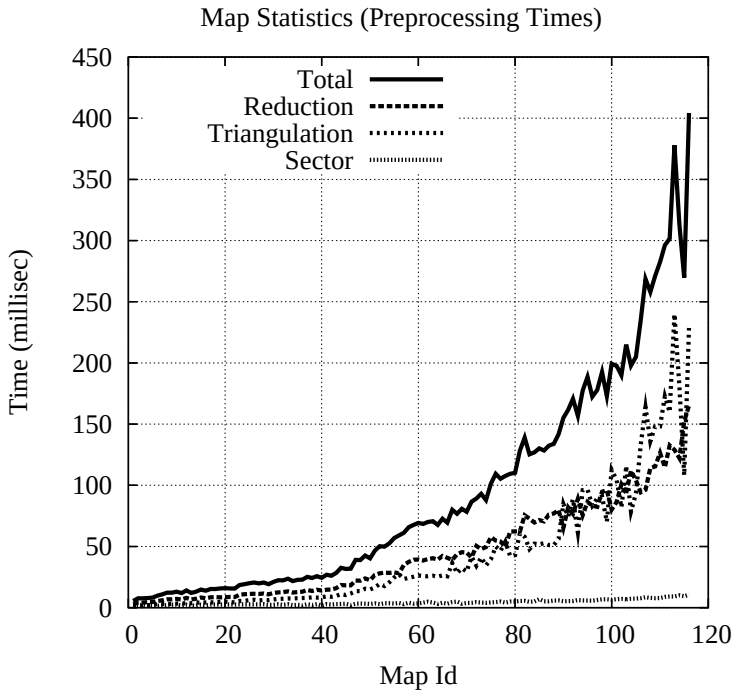

Figure 9.2: Preprocessing times divided into triangulation, reduction, and sector processing

advantage over the grid-based techniques.

Finally, and most importantly, notice the "level-3 nodes" line at the bottom of the graph. Not only are there significantly fewer of these than there are grid cells, constraints, or triangles, but their number hardly changes as the complexity of the environment forces the numbers of triangles and constraints up. This is the benefit of the independence of the most abstract graph from the nature of the obstacles in the environment and only their number. Even the maps with the most level-3 nodes have only a few hundred of them, far below the figures for other representations. Most maps have far fewer level-3 nodes still. The fact that TRA* searches the most abstract graph of which these are the vertices, gives it an even greater advantage.

Figure 9.2 shows the times required to perform the preprocessing of each environment. Again, this uses the same system of indexing the maps by the number of triangles in the resulting triangulation from Figure 9.1. While the times are long enough that this could not likely be done in real time, it might be able to be performed if the computation is spread out somewhat. This means it could be used if changes in the environment happen occasionally, however it would not be suited to incorporating constantly moving objects, such as other objects, in the environment. Possibilities for dealing with such situations are given in Section 10.2. This process could easily be done as a map is loaded at the onset of a level, for example, with the resulting wait being almost imperceptible.

It is important to notice that these times are indeed linear in the number of triangles

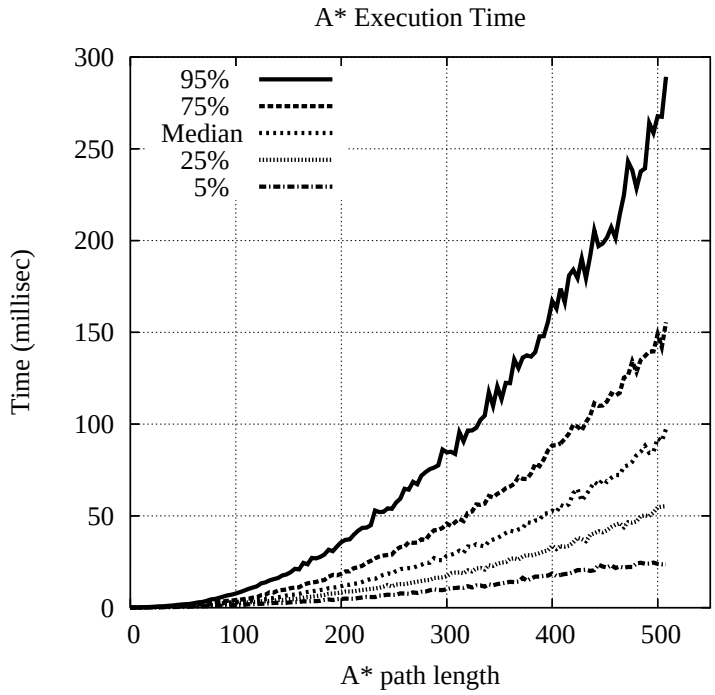

Figure 9.3: Percentiles of A* running times by path length

in the resulting triangulation, so as the environments become more complex, these times will not increase excessively. Given the results here, processing of a single triangle takes approximately 20 microseconds. The total preprocessing time is fairly evenly split between creating the Constrained Triangulation from the constraints using the implementation described in Section 3.4, and creating the most abstract graph using the algorithm given in Chapter 6. An almost negligible amount of time is spent doing the preprocessing for the point-location technique described in Section 8.1.

Figure 9.3 shows the execution times for the standard A* algorithm on the grid representation of the environment. Such times are certainly prohibitive to a real time setting with some of the longer paths taking over a quarter of a second. Something else to notice is the way the time increases more and more the longer the path gets. Certainly this solution does not scale well to longer paths, and on top of its long execution times, its behaviour can be hard to predict, as evidenced by the spread of the percentile lines, which can be important for application areas such as commercial games.

The execution times for PRA* (described in more detail in Section 2.4) are given in Figure 9.4. Looking at the time scale on the left and comparing it to that in Figure 9.3 indicates how much of a significant performance increase PRA* has over the standard A* algorithm. Considering that it works on a grid like A*, this algorithm performs quite well. This can be attributed to the selection of a layer of abstraction on which to perform the search, which reduces the search space significantly, but still provides sufficient details about

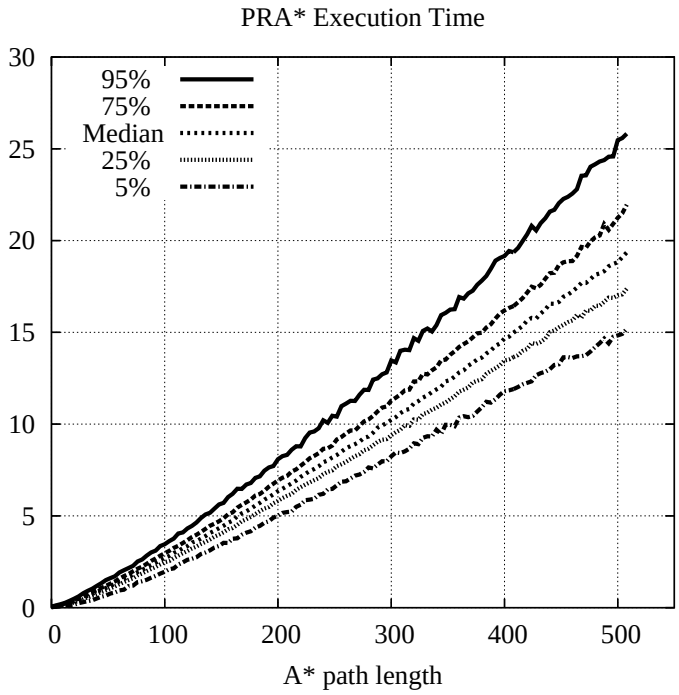

Figure 9.4: Percentiles of PRA* running times by path length

the environment to provide a high quality path.

In addition to the raw speed of this algorithm, the execution times increase in a more linear fashion than those of A*, when presented with longer paths. This is because the further the start and goal points are away from each other in terms of path length, the more abstract the layer where they meet at the same state, and subsequently, the more abstract the layer where the initial search is done.

Furthermore, the lines representing the percentiles are significantly closer together, indicating that the running time is much more predictable than A*, which makes it more suited to real time applications such as commercial games. Again, this is caused by many of the details of an environment that can complicate the execution of a pathfinding search being removed during the abstraction process.

This loss of information also has a negative effect on the lengths of the paths found using this algorithm; since details of the environment are lost the further the layer being searched is from the original graph, an optimal path on this layer can sometimes translate into suboptimal paths on the environment. Luckily in practice, the paths returned by PRA* are very likely either optimal, or very close.

Before looking at the execution times for the TA* and TRA* algorithms, some terminology must be introduced. For both of these algorithms, we designate a parameter F to denote the progress of the anytime algorithm. The value of this attribute designates the multiple of the time required by the algorithm to find the first path.

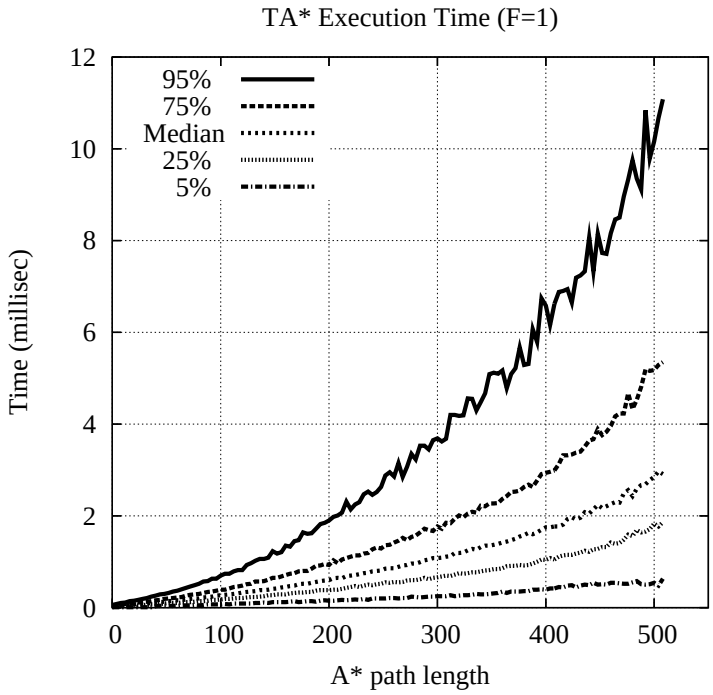

Figure 9.5: Percentiles of TA* running times by path length

For example, in Figures 9.5 and 9.6, the execution time with $F = 1$ indicates it is the time required by the algorithm to find its first solution. Later, in Figures 9.10, 9.11, 9.12, and 9.13, this value represents the multiple of this time at which the length of the current best path is taken. We also use TA*(f) to denote TA* with $F = f$ and similarly TRA*(f) to denote TRA* with $F = f$.

The execution times for the TA* algorithm described in Chapter 5 with $F = 1$ appear in Figure 9.5. Again, a glance at the time scale indicates the improvement this technique makes over A* and even PRA*. Even though this algorithm only searches the base-level graph, the benefit of the reduced number of triangles required to represent the environment over the number of cells required by the grid-world techniques, results in shorter execution times. This indicates how efficient a triangulation is for environment representation.

Despite its reduced execution times, however, the execution times of TA*(1) resemble those of A* in how they increase as the corresponding A* paths get longer, and how much the times for the different percentiles differs. Unlike PRA* and TRA*, TA* does not take advantage of an abstraction, and as such, the number of search states expanded increases significantly the farther the search must extend, especially considering that it is required to consider multiple states corresponding to any triangle. Furthermore, this lack of an abstract representation means that variations in the environment affect TA*'s performance more so than that of PRA* or TRA*, which are not affected by such details, resulting in more varied execution times than for those algorithms. Below we will explore the behaviour of

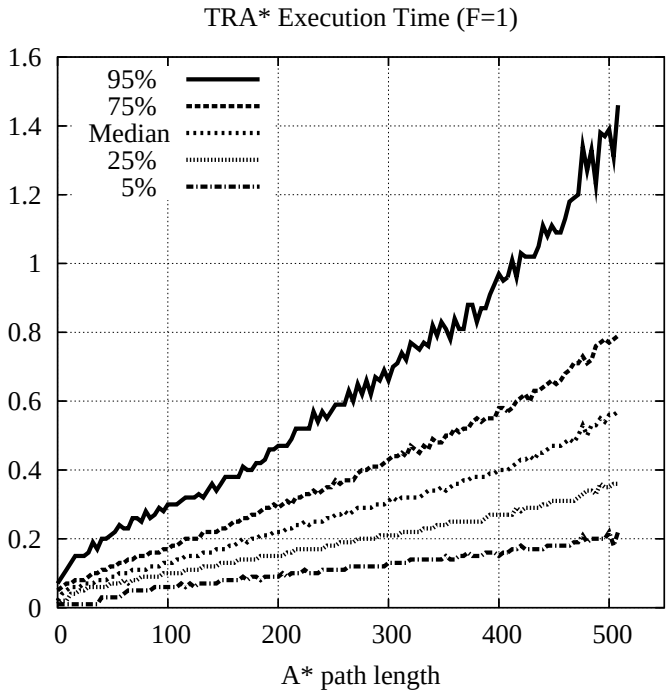

Figure 9.6: Percentiles of TRA* running times by path length

the anytime algorithm and how it converges on an optimal solution.

Figure 9.6 displays the execution times of the TRA* algorithm described in Chapter 7, with $F = 1$. The execution times of this algorithm are less than the other algorithms tested, because it benefits from both the improved representation efficiency afforded by the triangulation, and the reduced search space resulting from the abstraction process.

In addition to these short execution times, both the variation in these times and the way they increase as the lengths of the corresponding A* path lengths change are more akin to those features of the PRA* algorithm. Again, the abstraction reduces the details which cause the variation in the execution times. There is more variation here than with PRA* because the abstraction mechanism used by TRA* described in Chapter 6 reduces the environment to the minimal representation which still contains the topological structure of the environment, whereas the abstraction mechanism used in PRA* can reduce the environment further, at the cost of the ability to guarantee finding an optimal path.

Similarly, the execution times for TRA*(1) increase in a slightly less linear fashion than PRA* as the corresponding A* path lengths change, because of this inherent limit in the abstraction used by TRA*. As the distance between the start and goal increases, PRA* can simply perform the search on a more abstract representation of the environment, resulting in a smaller increase in the resulting execution time, however TRA* has only a single layer of abstraction at its disposal. The behaviour of the anytime algorithm associated with TRA* and its convergence of an optimal solution is explored below.

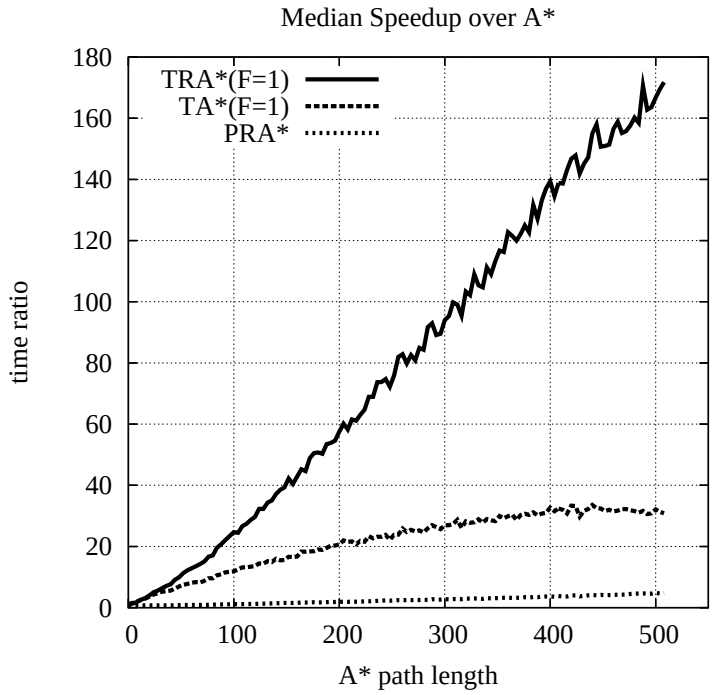

Figure 9.7: Median ratio of execution times of PRA*, TA*(1), and TRA*(1) to A*

We present a more direct comparison between the algorithms tested in terms of speed and the number of node expansions in Figures 9.7 and 9.8, respectively. Figure 9.7 shows the median speedup of PRA*, TA*(1), and TRA*(1) over A*. The speedup of PRA*, while less than that of TA*(1) and TRA*(1), increases steadily with the A* path lengths, as its abstraction allows its execution times to avoid the performance hit taken by A* with this increase.

TA*, while offering a greater speedup than PRA*, shows a decrease in this benefit as paths increase in length. Indeed, while the initial benefit of the reduced search space offered by the triangulation of the environment creates a large initial speedup, the lack of an abstraction to help it deal effectively with longer paths, makes it level off, eventually becoming parallel to the horizontal axis, indicating only a linear increase in performance.

While such an increase is certainly beneficial, TRA* performs even better. The even further reduced search space afforded by the abstraction process allows TRA* to gain a large initial speedup over A*, moreso than the other algorithms. On top of this, however, this speedup steadily increases as the corresponding A* paths get longer because, as mentioned before, the abstraction removes many details which make searches such as A* expand with this distance.

The results shown in Figure 9.8 echo these patterns in the 90^{th} percentile of search state expansions. As with its times, the number of states expanded by A* is significantly more than the other methods, and only gets worse as the paths increase in length. PRA* again

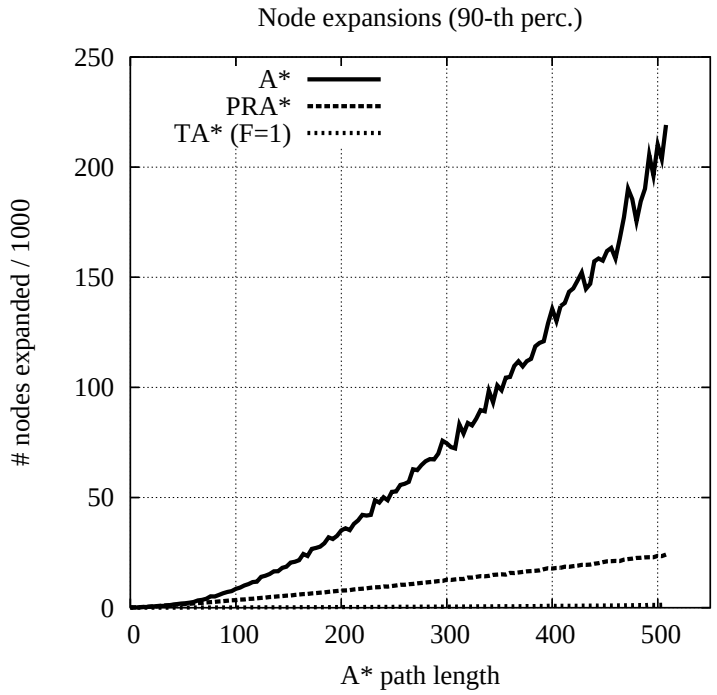

Figure 9.8: 90^{th} percentile of number of search states expanded by A*, PRA*, and TA*

does not exhibit this behaviour, and shows much improvement over the number of states expanded by A*.

TA* expands even fewer search states, being barely visible at the bottom of the graph, showing again the benefit of triangulations as an environment representation. TRA*, combining this benefit with that of its abstraction, expands even fewer nodes than TA* and was not included in this graph because it would not be visible.

Another trade-off between pathfinding techniques such as PRA* which use abstractions that can lose information about the environment, and those in this thesis, is the solution quality. Of the algorithms compared in this section, A* returns a single path which is guaranteed to be optimal, but which takes significant time and memory to find.

In contrast, PRA* returns a single path which, with a high degree of probability, is very close to optimal. This algorithm itself provides the opportunity to trade-off execution time and path quality by selecting the abstraction layer on which the initial search is given. The fact that the first (and only) solution provided by this technique is likely very close to optimal is an advantage, however, if the abstraction is used to any benefit (the search is performed on any but the original representation), one cannot be guaranteed an optimal solution.

The anytime algorithm format of the TA* and TRA* algorithms provides a different trade-off between execution time and path quality. Neither the base-level graph utilized by TA* nor the most abstract graph for TRA* provide options for differing levels of detail.

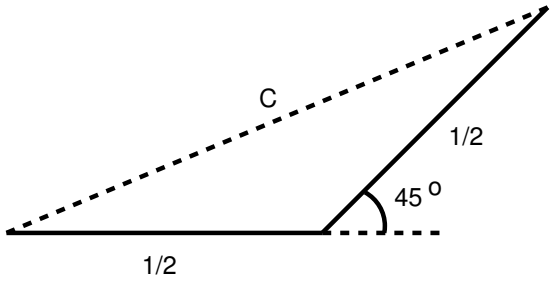

Figure 9.9: Calculation of the constant C to determine the "bound" line

The base-level graph presents the environment in complete form and the most abstract graph provides as few details as possible while still retaining the topological structure of the environment.

Thus, the time for these algorithms to find the initial solution and the quality of this solution cannot be easily controlled, as with the time required to converge on an optimal solution. However, at any time between these, the searches can be stopped and will yield a solution whose quality is between these two. In this way, giving more time to TA* and TRA* results in a better quality solution, which is a useful compromise, especially in a real-time setting when such computation might have to be halted if resources are needed elsewhere.

Figures 9.10, 9.11, 9.12, and 9.13 display the ratio of the length of the paths found by TA* and TRA* with various values for the F parameter, to the length of TA*(10). While most of the time TA* will converge on an optimal solution given 10-fold the time required to find the first solution, in some instances this was not the case. To cover such possibilities, we wanted to add a factor indicating how short an optimal solution could potentially be.

We know that the paths returned by A* are optimal, but because it is a grid-based method, they are constrained to moving between centers of tiles and the paths returned by the triangulation-based methods TA* and TRA* are not. Therefore, even suboptimal paths given by these last two methods are capable of being shorter than the optimal path while constrained to a grid. So we determined the maximum ratio of a grid-constrained path to its arbitrary-motion equivalent.

This is shown in Figure 9.9, where half of the grid path (shown by a solid line) is axis-aligned and the other half at a diagonal, and the arbitrary-motion path (shown by a dotted line) moves between the same endpoints. One can check that modifying this configuration necessarily increases the ratio of the length of the arbitrary-motion path to the grid one.

If we take the length of the grid path to be 1 like in the figure, we need to determine the length of the corresponding path in free space. First we must calculate the vertical and horizontal measurements of the diagonal section of the path (these will be equal because the line is on a 45° angle from the horizontal and vertical axes). Using Pythagoras' Theorem, these dimensions are $\frac{1}{2} = \sqrt{x^2 + x^2} \Rightarrow \frac{1}{2} = \sqrt{2x^2} \Rightarrow (\frac{1}{2})^2 = (\sqrt{2x^2})^2 \Rightarrow \frac{1^2}{2^2} = 2x^2 \Rightarrow \frac{1}{4} = 2x^2 \Rightarrow \frac{1}{8} = x^2 \Rightarrow \sqrt{\frac{1}{8}} = \sqrt{x^2} \Rightarrow \frac{1}{\sqrt{8}} = x \Rightarrow x \approx 0.3536$. Now, the arbitrary path is $\frac{1}{2} + x$ in length along one axis and x in length along the other, yielding a length of $C = \sqrt{(\frac{1}{2} + x)^2 + (x)^2} = \sqrt{(\frac{1}{4} + x + x^2) + (x^2)} = \sqrt{\frac{1}{4} + x + 2x^2} \approx \sqrt{(0.25) + (0.3536) + 2(0.3536)^2} = \sqrt{0.25 + 0.3536 + 0.25} = \sqrt{0.8536} \approx 0.9239$.

With this knowledge, a "bound" line was added to Figures 9.10, 9.11, 9.12, and 9.13 corresponding to the minimum length the optimal path with arbitrary motion could be. For the graph of the n^{th} percentile path length, this bound was calculated to be the $(100 - n)^{th}$

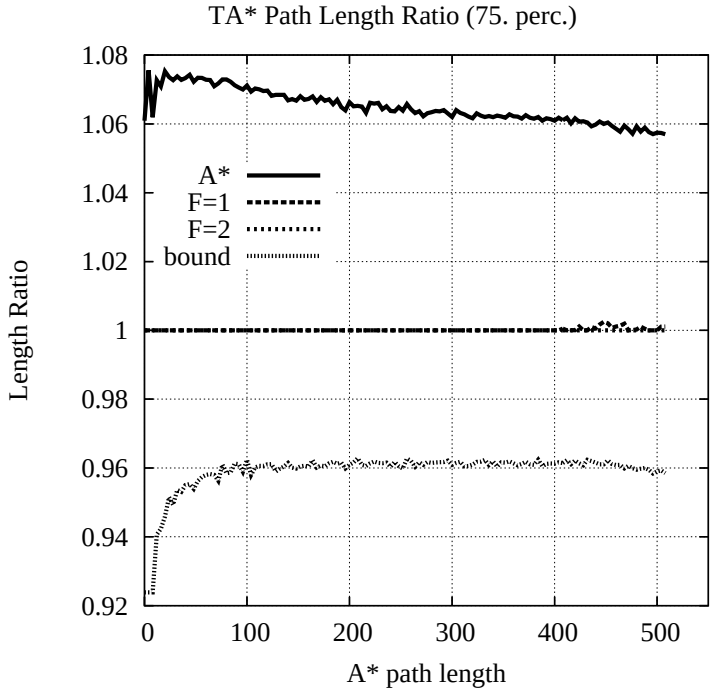

Figure 9.10: Ratio of 75^{th} percentile of TA* path length to TA*(10)

percentile of the A* path length, multiplied by this constant C.

We see in Figure 9.10 the 75^{th} percentile length of the initial paths returned by TA* ($F = 1$) and those returned given twice this amount of time ($F = 2$). Again, for the most part, and especially on this particular graph, TA*(10) is optimal, and thus the line at ratio 1 represents roughly the optimal path length. This is less true for the longer paths on the higher percentile graphs, where the optimal path length might be slightly less, moving this line down slightly towards the right of the graph.

Here we see that most of the time, TA* finds an optimal path first (indicated by the fact that most of the time, the line for TA*(1) lies on the ratio 1, meaning it is the same value the algorithm would return given 10 times as long). For some paths whose A* length is greater than 400 tile widths, the initial path returned is slightly longer than optimal, although by small fractions of a percent. In these cases, the final path is reached before $F = 2$, as this line lies entirely at the ratio 1.

Notice that the length of the path returned by TA*(10) is still within 4% of the minimum length possible for the optimal path for the majority of these paths. Also, even when the initial solution is longer than the final one returned, it is still much shorter than the optimal path on the grid returned by A*.

Figure 9.11 shows the 95^{th} percentile for the TA* path lengths. We see that in these rare cases, the first path returned by TA* ($F = 1$) is longer than that returned by TA*(10) on paths whose A* length is as short as 150 tile widths, and reaches roughly 5% longer than

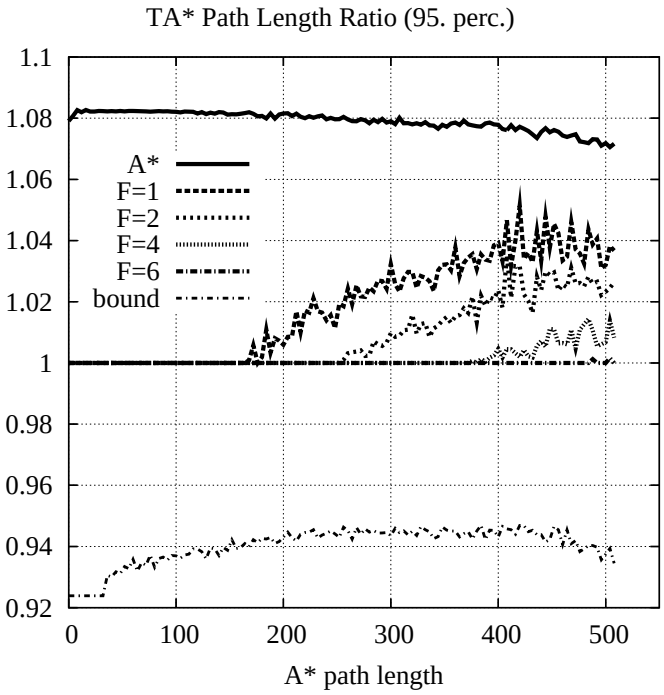

Figure 9.11: Ratio of 95^{th} percentile of TA* path length to TA*(10)

TA*(10). Looking at the line for TA*(2), we see it yields the same solution as TA*(10) until the A* length of the paths reach about 250 tile widths, and the path at this point can be up to 3% greater.

Similarly we see how on paths with longer A* lengths, a greater multiple of the time required to find the first path (or F parameter) is needed to provide an optimal path. In this case we see that TA*(6) finds paths of the same length as TA*(10) for 95% of the paths tested, for all but a few of the longest, where it deviates from this value slightly.

Now turning to TRA* for Figure 9.12, we see a slightly different situation. The initial path returned by TRA* is only equivalent in length to TA*(10) 75% of the time for paths whose A* length is less than 200 tile widths, whereas with TA* almost all tested paths found initially were equivalent to this figure. Increasing F to 2 finds paths of the same length as TA*(10), 75% of the time, up to an A* length of about 400 tile lengths, and a higher value for this parameter (and thus more time) is required to yield this value for all paths.

This is because TA* has inherently more accurate g-values, which is possible because visiting each triangle individually allows a better estimate, whereas TRA* skips entire corridors of triangles, only having the opportunity to adjust these values a fraction as often. This means that the initial solution returned by TRA* is often longer than that for TA*, and TRA* also takes longer to converge on an optimal solution, in relation to the time required to find its first solution. Note, however, that while TRA* requires a greater F parameter value to find an optimal solution, its first solution is so fast that converging on an optimal

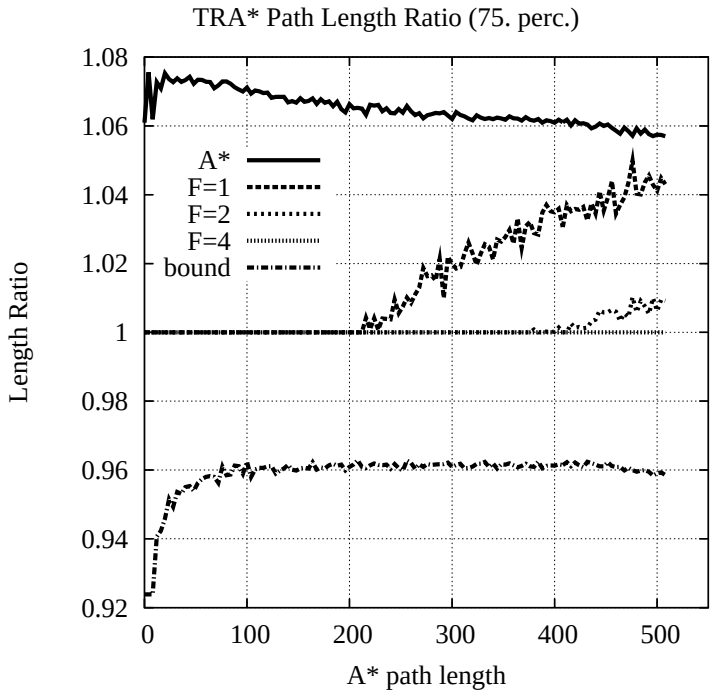

Figure 9.12: Ratio of 75^{th} percentile of TRA* path length to TA*(10)

solution with TRA* still takes a fraction of the time that TA* requires to find arrive at the same solution.

In Figure 9.13, we see this effect becomes even more pronounced at the 95^{th} percentile. Occasionally, the length of the first path found by TRA* is significantly longer even than that of an optimal path constrained to the grid (A* path). The lengths of the paths returned by TRA*(2) are fortunately less, with this figure decreasing as more time is allotted to the algorithm.

Although in these rare cases, it takes TRA* several times longer to yield a path close to that of TA*(10), than it took to find the first solution, these are likely the cases where TA* requires a higher F parameter to find such a value as well, and the initial solution is returned by TRA* several times faster than by TA*, so TRA* will still, in most cases, converge on an optimal solution first. Thus, although the initial path returned by TRA* is often longer than that returned by TA* and it takes longer in relation to the time to find this first path, to converge on an optimal path, it still most often reaches this value long before TA* does. The benefit of the smaller state space afforded by the most abstract graph, then, outweighs the drawback created by the less accurate distance measures resulting from it.

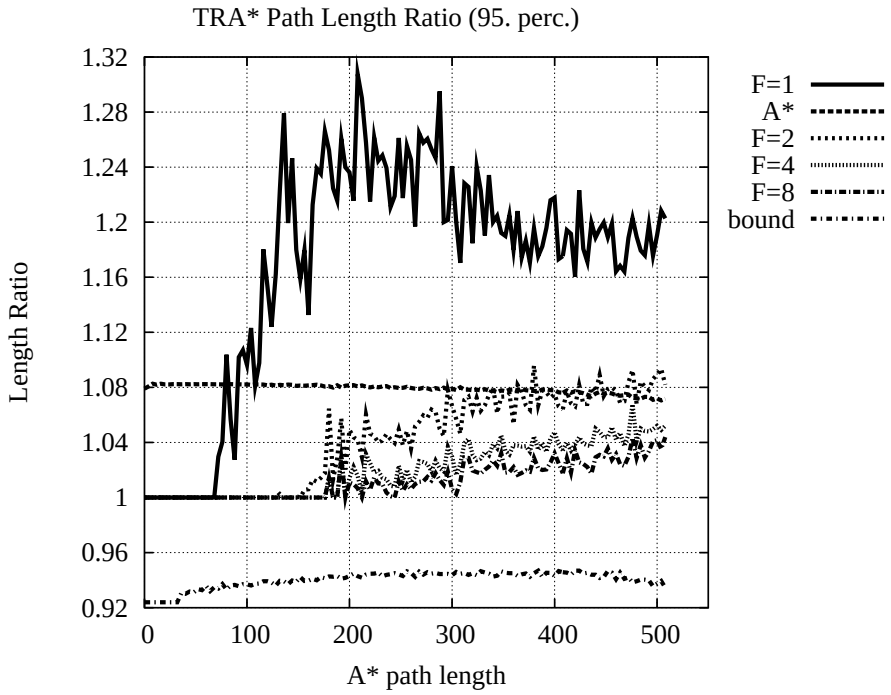

Figure 9.13: Ratio of 95^{th} percentile of TRA* path length to TA*(10)

9.3 Discussion

The behaviour seen in the anytime algorithms associated with TA* and TRA* could be exploited using a clever algorithm. One can see that in most cases, when the first path found by these algorithms is short or found in a very short amount of time, this path is very close to optimal, if any longer at all. Conversely, when the first path found is quite long or requires more time to find, this path is often farther from optimal and the algorithm requires longer to converge.

Using this knowledge, one could devise an algorithm which takes the length of the first path found or the time required to find it, and determines a multiple of this time at which a solution very close to optimal is likely. Therefore in the case of short or quickly-found paths, the algorithm could be stopped immediately and in the case of paths which are longer or require more time to find, it could be given more time to converge.

This type of setup would be useful in cases where there is a certain, fixed amount of time in which some number of paths must be found. Such a situation is common for example in Real-Time Strategy games where a number of objects could be ordered to move somewhere with a single command. This way less time is wasted on simpler paths allowing more for more difficult ones, and providing better quality paths on average.

This also alleviates another problem common to anytime algorithms: the fact that an optimal solution is often found long before the algorithm can determine that it has found one.

This approach halts the algorithm when it is likely that an optimal solution has been found, which can sometimes result in a suboptimal path being returned. However, in commercial games, a slightly suboptimal path is often more desirable than one which requires more resources to find.

When comparing our algorithms TA* and TRA* to A* and PRA*, a number of patterns emerged. First, the use of a triangulation to represent the environment gave TA* and TRA* an advantage over the other methods, which used a grid. Even TA* found solutions faster than PRA*, despite that the latter benefitted from an abstraction mechanism and the former worked on the environment itself. TRA* performed the best of all the algorithms tested, by having both the superior environment representation, and an efficient abstraction thereof.

Second, an abstraction mechanism not only allowed the methods PRA* and TRA* to find solutions faster than their counterparts with no abstraction, but also made these methods more predictable and less hampered by the length of the path. PRA*, which searched an abstraction of the grid-world environment, performed better than A*, which searched the grid world directly. Similarly, TRA* performed better searching the most abstract graph of the triangulated environment than TA* did searching its base-level graph.

The fact that PRA* was not as affected by the length of the path can be attributed to the fact that the longer the path between these points, the more abstract the layer at which they meet and therefore the more abstract the layer at which the search is performed, thus keeping the number of states searched from increasing as much with this distance. These layers of abstraction make PRA* the better of the two abstraction-based methods in dealing with longer paths effectively. While the most abstract graph used by TRA* has a single layer, its vertices again depend only on the number of obstacles in the environment, so an algorithm searching this reduced representation of the environment is affected less by the length of the resulting path by virtue of paths requiring fewer vertices.

Similarly, the abstractions allow these two methods to be more predictable (with less variation in the times to find solutions) than their counterparts. In both cases, this is a result of the details which complicate pathfinding tasks being lost to the abstracted version of the environment. In the case of TRA*, such details as dead ends, long corridors, and complexities in component barriers, are identified explicitly during the abstraction process. For PRA*, these are removed as increasingly abstract representations of the environment lose more information about the original environment.

To conclude, PRA* was the most predictable and least affected by path length, of all the methods tested. Considering the grid-world environment on which it was based, it showed vast improvement over A*, the other algorithm based on this representation. TRA* also showed that it handled such increasing distances well, and its running times did not vary as much as the methods which did not benefit from abstractions. The main strengths of TRA* are first its raw speed, which was greater than any other method, and second, the ability of its anytime algorithm to converge on an optimal solution.

TA* also offered significant speed, being faster than all but TRA*, and similarly afforded convergence to an optimal path. While it did not handle distance as well as the abstraction-based methods, it was still faster than both grid-based methods. TA* could still be useful over TRA* in situations where the environment changes often enough that updating the most abstract graph for each modification becomes prohibitive. In this case, the triangulation could be repaired by the mechanisms provided by the DCDT technique described in Section 3.4, and TA* can search on this environment directly, while still returning paths quite quickly.

Chapter 10

Conclusion and Extensions

In Section 10.1, a number of the advantages apparent from the work done to this point are discussed, while the many possible directions for further work and their expected benefits are explored in Section 10.2.

10.1 Conclusion

In this section, we will discuss the findings and contributions of the work in this thesis. Subsection 10.1.1 will summarize the findings regarding using triangulations as an environment representation and Subsection 10.1.2 will discuss the further work with regard to using triangulations to efficiently find paths for nonpoint (specifically, circular) objects. Then Subsection 10.1.3 will conclude the main contribution of this work—the reduction technique applied to the triangulation graph. Finally, Subsection 10.1.4 revisits the work done in terms of faster point location.

10.1.1 Triangulations

First we have seen the advantages of using polygonal representations for pathfinding. This reduces the state space significantly, especially in cases of line segment obstacles, and more so when these are not axis-aligned, forcing grid representations to either greatly increase their resolution or risk missing paths. Even in tiled environments which are designed to work with grid-based pathfinding solutions, polygonal representations have a smaller state space, representing any area similarly regardless of size.

The benefits of this representation alone can be seen in the results of our experiments with TA*. Triangulations in particular are an ideal polygonal representation for environments for a number of reasons. First, they provide a simple, uniform interface for pathfinding, second, there are fast algorithms for representing the environment in this way, third, they are conducive to repair in the presence of changes in the environment, and finally, they have useful properties for defining enhancements, which we will discuss next.

10.1.2 Base-level Enhancements

Because of the simple structure of a triangle, we have been able to introduce enhancements to the triangulation to improve base-level search. Such an improvement was the elimination of multiple representations of the graph and undesirable sliver-like triangles resultant from applying the Minkowski Sum operation to "grow" obstacles. This was achieved by calculating maximum radius for a circular object to travel between any two edges of a triangle.

This idea was extended to allow calculation of the shortest path for such an object through a channel of triangles in time linear in the number of triangles in that channel. This was important because in searching for an optimal path for an object between two points, it is often necessary to determine the lengths of the shortest paths through several channels.

Another enhancement to which triangulations lend themselves was the main contribution of this work: abstraction of the graph induced on the base triangulation. The results of this are concluded next.

10.1.3 Graph Abstraction

The homogeneous manner in which triangulations represent the environment allowed for the abstraction process to be performed on it. By identifying dead ends, corridors, decision points, trees, rings, and other graph structures, the pathfinding task was simplified to determining on which side of each obstacle to go. This is the most abstract graph one can search when looking for optimal paths.

This abstraction allowed us to identify several situations where this path can be found without using search, and when search was required, greatly decreasing the search space. The main advantage of this approach was that pathfinding not only did not depend on the size and orientation of areas, but it also did not depend on the properties of the individual constraints, only the number of obstacles.

As we saw in our experiments with TRA*, this abstraction was very successful in providing an efficient representation for pathfinding. It also provided an excellent basis for further research. This is discussed in more detail in Section 10.2.

10.1.4 Point Location

Finally, since the most time-consuming part of pathfinding was often found to be finding the triangle in which the start or end point is contained, it was necessary to implement an improved point location technique. This was doubly true due to the fact that pathfinding on the abstract graph involved finding the containing triangles for both the start and goal points and slow point location would threaten to overshadow the benefits of TA* and TRA*.

Point location was previously done by taking the "first" triangle in the triangulation and crossing edges toward the desired point until arriving at the triangle containing the point. This was prohibitively slow, taking time proportionate to the number of triangles in the environment.

To combat this, a number of rectangular "sectors" were defined covering the environment and whose midpoints were determined. When dealing with a triangle while either creating or repairing the triangulation, the algorithm would check if it overlapped any sector midpoints, and if so, would associate that triangle with the corresponding sector.

When point location was performed, the closest sector midpoint to the desired point was calculated and the point location "walk" was started from the triangle associated with that sector, which could be accessed in constant time. Even with a relatively sparse grid of sectors, this technique improved point location considerably, effectively eliminating the process as a bottleneck to pathfinding.

10.2 Extensions

There remain numerous possibilities for application of both triangulations and abstractions thereof, to pathfinding, motion, and other areas. Nonstatic environments are an area of particular interest. The three main situations in this area involve mobile obstacles, group

pathfinding, and multiple objects, which are discussed in Subsections 10.2.1, 10.2.2, and 10.2.3, respectively. Then we discuss possible extensions to this work other representations for the graph in Subsection 10.2.4, concerns regarding maintaining multiple, size-dependent graphs in Subsection 10.2.5, and finally offer some concluding remarks on the work in Subsection 10.2.6.

10.2.1 Mobile Obstacles

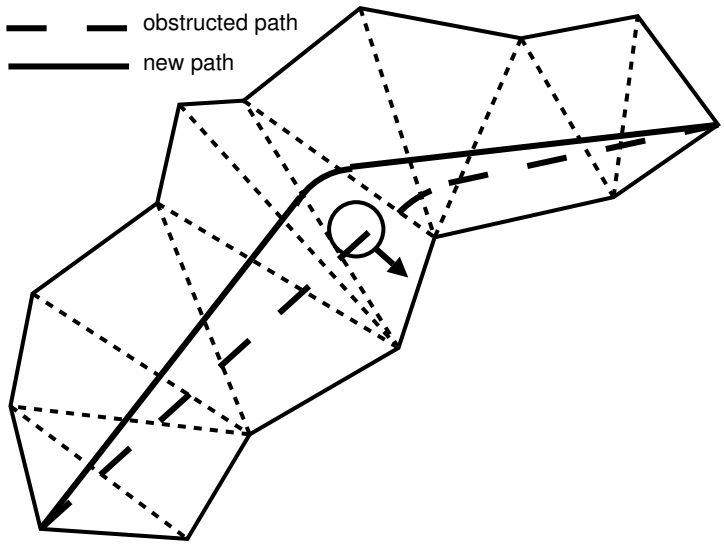

Figure 10.1: Object steering around a mobile obstacle within its channel

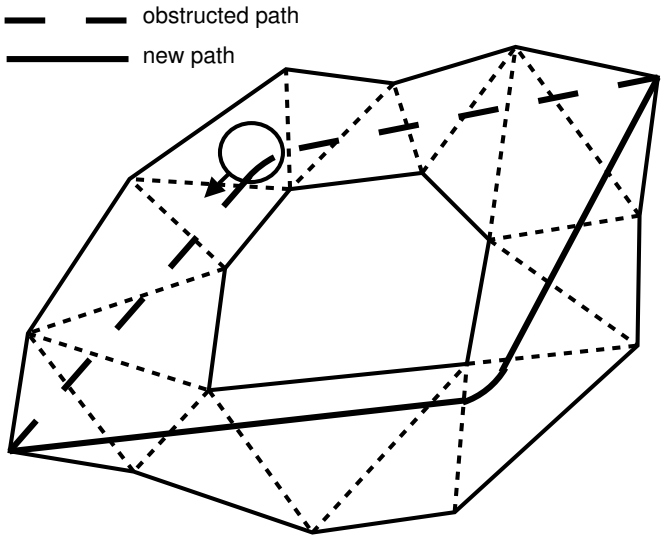

Figure 10.2: Object finding another channel when blocked by a mobile obstacle

Mobile obstacles is the situation involving either obstacles or other objects which may move into the path of ours, but whose motion we can neither control nor know beforehand.

In this case, the fact that the result of our pathfinding efforts is a channel of triangles as opposed to a single path can work to our advantage. If we predict that an obstacle will enter our object's channel about the time our object will reach that point, we have a few options for what to do to resolve such a collision.

The first possibility is that the collision occurs in a very wide triangle, and our object can simply move around the obstacle easily within the channel. Such a situation is shown in Figure 10.1. If it occurs in a narrow triangle and such a trivial fix is not possible, it may be that the obstacle will soon leave the channel or move to a wider triangle in the channel so our object can continue within the channel without waiting long. Finally, if the obstacle will block our channel and significantly hinder our object's progress, we can search the triangulation for a new channel and continue from there. A situation involving a mobile obstacle blocking most of a channel, and an alternative path being found is illustrated in Figure 10.2.

10.2.2 Group Pathfinding

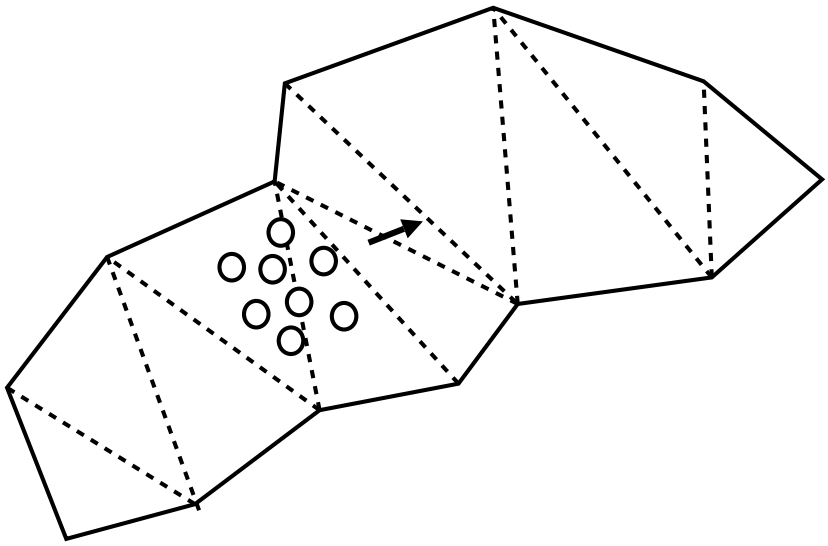

Figure 10.3: Group of objects travelling through a wide channel

Group pathfinding is a situation common in Real-Time Strategy (RTS) games and involves multiple objects all starting roughly in the same area and searching for a path to some other area. In this case, our determination of choke points can greatly aid us in pathfinding.

Once a channel is found between the start and goal areas, we can easily determine the narrowest point along the path. If this point is wide enough that our objects will not be significantly slowed, we can allow the objects to travel between the start and goal areas within the channel using some form of local control. Figure 10.3 shows such a case.

If the objects have varying top speeds, it would even be possible to send faster objects ahead of the group before narrower points of the channel so fewer have to travel through it once, as is shown in Figure 10.4, reducing congestion and speeding up the group as a whole.

If the narrowest point of the channel would slow the group of objects too much, one could search for other channels, and split up the objects between the channels based on the narrowest point of each. This way one could send more objects through wider channels, and in the case of varying object speeds, faster objects through longer channels as shown

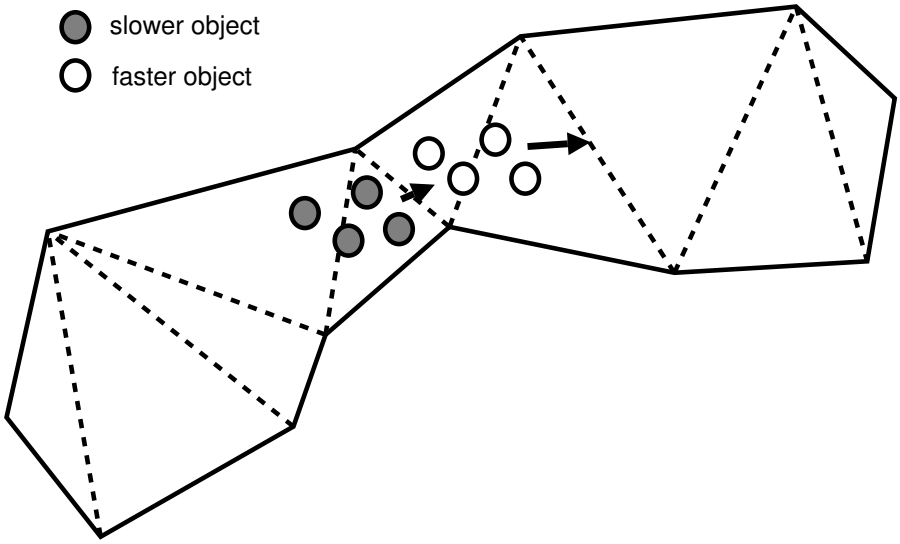

Figure 10.4: Group of objects spreading out to go through a choke point

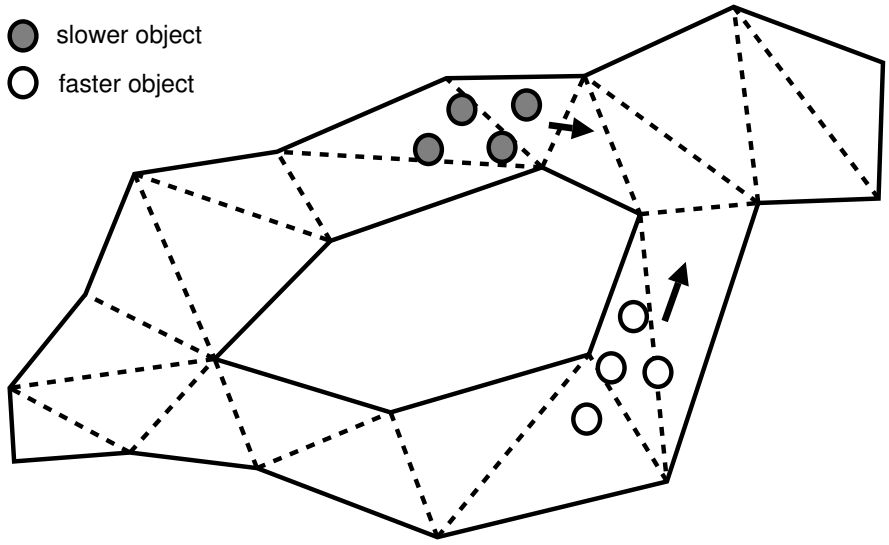

Figure 10.5: Group of objects splitting up into different channels

in Figure 10.5. One could view this problem of the fastest path for a particular group as a type of network flow problem.

Another consideration lies in the possibility of a group of objects being attacked when in an RTS game. In such an occurrence, it is more advantageous that a group of objects be close together so as to repel the attack more effectively. Thus, early in the game if the terrain and enemy positions are not known, it might be better to sacrifice group speed and keep the objects close together within one channel in case of an attack, whereas later in the game when terrain and enemy positions are known, one could split up a group travelling through unhostile terrain to get the objects to the goal sooner.

10.2.3 Multiple Objects

Finally, the problem of multiple objects is when we control a number of objects which all want to move from their unique starting positions to different goals. An approach to this problem in a discrete-time grid world [45, 46] can be applied to triangulations with minor adjustments. Instead of reserving cells completely for a time step, objects could partially reserve triangles for time intervals. This way a small object would only reserve a small portion of a large triangle.

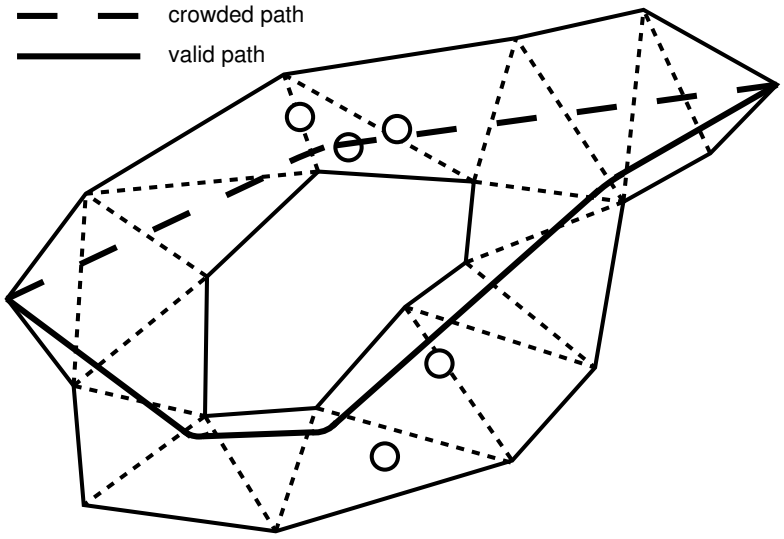

Figure 10.6: Object selecting a less crowded channel

When an object is searching for a path, it would disregard triangles that were reserved to a certain threshold based on that object's size, at the time when the object would pass through that triangle given the path. Figure 10.6 shows an object rejecting a channel containing a crowded triangle in favour of one which is less congested. As each object chooses a path, it reserves a portion of the triangles in the path (proportionate to the size of the object and that of the triangle) for the time intervals that it would pass through them.

This would prevent triangles from being too congested at any one time, and some local control should be sufficient to create valid paths for each object within its channel. Because this is done in continuous space, one would have to adjust a threshold dictating when an object would regard a triangle as too crowded for traversal, in order to retain as many paths as possible while avoiding over-congestion.

10.2.4 Further Abstraction

There are other extensions that are possible for the graph and abstraction layers. Further abstraction is possible if one collapses doubly-connected components of the abstract graph into single nodes of a more abstract graph. Where on the abstract graph the nodes represent decision points for which way to go around an obstacle, in this new graph, they would represent "rooms" in the environment which contain multiple paths between their entry and exit points.

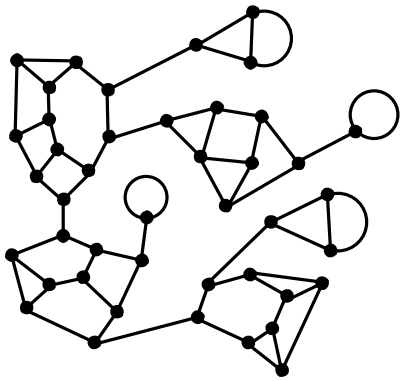

Figure 10.7: Graph of level-3 nodes

This graph would then be a tree of doubly-connected components. Because it is a tree, the highest level of the pathfinding problem would become trivial as there is only one path between any two points in a tree. The best paths between each pair of entry points of each doubly-connected component could even be cached, and then the search function would only have to get from the start and goal points on to this new abstracted graph, after which the pathfinding task would be simple. An abstract graph is shown in Figure 10.7 and its corresponding tree of doubly-connected components in Figure 10.8.

This approach may find suboptimal paths since the pathfinding is being done in continuous space and the shortest path between points in two triangles can change depending on where in those two triangles the points are. However, if the triangulation has certain properties, this suboptimality may be negligible, and in large environments it may be a welcome trade-off for faster searches. If this is not the case, the paths can always be refined

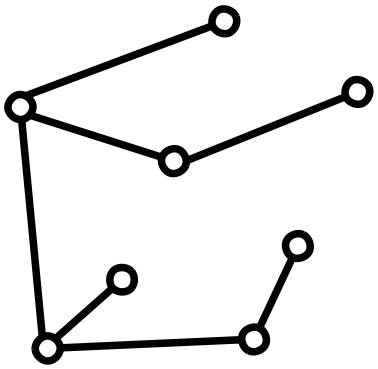

Figure 10.8: Tree of doubly-connected components

when needed on a less abstract version of the graph.

10.2.5 Size-Dependent Graphs

Another possible extension of this technique is apparent when considering that with certain object sizes, some edges of the graph will not be traversable. This means that during the search, some resources can be wasted searching dead ends, and the width must always be checked when moving between nodes. A solution to this problem would be to have multiple abstract graphs for different sized objects.

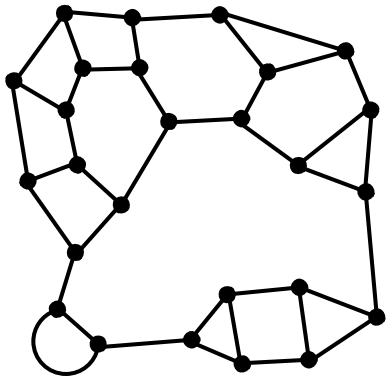

Figure 10.9: Abstract graph for a small object

Figure 10.9 shows an abstract graph for a small object, whereas Figure 10.10 shows the abstract graph for a larger object in the same environment. This second graph is missing edges corresponding to narrower corridors through which the larger object could not fit, and therefore has fewer edges. Although this is not as general as simply recording the widths through triangles, it would use less memory than creating multiple copies of the base-level graph, and allows for more efficient searching at the cost of more preprocessing, which is often desirable in commercial games.

If there are few different sizes of objects, the abstract graphs could simply be calculated and stored for each size of object. However, if there are more sizes of objects than there are different sizes of choke points in the environment, one could let the environment determine

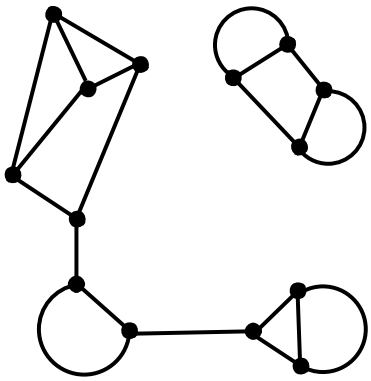

Figure 10.10: Abstract graph for a large object

the abstract graphs. Thus, if all objects are within a certain size range, each time there is a choke point of a new width in that range, we could create an abstract graph to reflect this connectivity. Thus, only objects smaller than the width of this choke point would see an edge crossing it. Then for any object, one would check into which range of sizes it fits and use the appropriate abstract graph in the ensuing search.

10.2.6 Final Thoughts

Indeed, there are many possible extensions of both polygonal pathfinding using Constrained (Delaunay) Triangulations and topological abstractions thereof. Hopefully these and more techniques will find further merit in academia and eventually application in commercial games and robotics.

Bibliography

[1] M. V. Anglada. An improved incremental algorithm for constructing restricted Delaunay triangulations. In *Computer & Graphics*, volume 21, chapter 2, pages 215–223. 1997.

[2] R. V. Benson. *Euclidean Geometry and Convexity.* McGraw Hill, 1966.

[3] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. *Proceedings of IJCAI-03, (Eighteenth International Joint Conference on Artificial Intelligence)*, 2003.

[4] D. Billings, D. Papp, J. Schaeffer, and D. Szafron. Opponent modeling in poker. *Proceedings of AAAI-98 (15th National AAAI Conference)*, 1998.

[5] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):1–22, 2004.

[6] C. B. Boyer. *A History of Mathematics, 2nd ed.* New York: Wiley, 1968.

[7] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997.

[8] M. Buro. How machines have learned to play Othello. *IEEE Intelligent Systems J.*, 14(6):12–14, 1999.

[9] M. Buro. ORTS: A hack-free RTS game environment. In *Proceedings of the International Computers and Games Conference*, pages 280–291, 2002.

[10] M. Buro. Real-Time Strategy games: A new AI research challenge. In *Proceedings of the International Joint Conference on AI*, pages 1534–1535, 2003.

[11] M. Buro and T. Furtak. On the development of a free RTS game engine. *GameOn'NA Conference*, pages 23–27, 2005.

[12] B. Chazelle. A theorem on polygon cutting with applications. In *In Proceedings of the 23rd IEEE Symposium on Foundations of Computer Science*, pages 339–349, 1982.

[13] L. P. Chew. Constrained Delaunay Triangulations. In *Proceedings of the Annual Symposium on Computational Geometry ACM*, pages 215–222, 1987.

[14] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proc. IEEE*, 80:1412–1434, 1992.

[15] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.

[16] J. Culberson and J. Schaeffer. Searching with pattern databases. *CSCSI '96 (Canadian AI Conference)*, pages 402–416, 1996.

[17] L. de Floriani and A. Puppo. An on-line algorithm for Constrained Delaunay Triangulation. In *Computer Vision, Graphics and Image Processing*, volume 54, pages 290–300. 1992.

[18] B. Delaunay. Sur la sphère vide. In *Izvestia Akademii Nauk SSSR*, volume 7, pages 793–800. Otdelenie Matematicheskikh i Estestvennykh Nauk, 1934.

[19] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *ACM Symposium on Computational Geometry*, 2001.

[20] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[21] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. In *Algorithmica*, chapter 7, pages 381–413. 1992.

[22] L. J. Guibas, L. Ramshaw, and J. Stolfi. A kinetic framework for computational geometry. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 100–111, 1983.

[23] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybern.*, 4:100–107, 1968.

[24] J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry Theory and Application*, 4:63–98, 1994.

[25] J. Hershberger and S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Comput.*, 28:2215–2256, 1999.

[26] R.C. Holte, M.B. Perez, R.M. Zimmer, and A.J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI Vol. 1*, pages 530–535, 1996.

[27] M. Kallmann. Path planning in triangulations. In *Proceedings of the IJCAI Workshop on Reasoning, Representation, and Learning in Computer Games*, pages 49–54, July 31 2005.

[28] M. Kallmann, H. Bieri, and D. Thalmann. Fully Dynamic Constrained Delaunay Triangulations. In *Geometric Modelling for Scientific Visualization*, pages 241–257. Springer-Verlag, 2003.

[29] K. Kedem, R. Livne J. Pach, and M. Sharir. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. In *Discrete Comput. Geom*, volume 1, pages 59–71. 1986.

[30] K. Kedem and M. Sharir. An efficient algorithm for planning collision-free translational movement of a convex polygonal object in 2-dimensional space amidst polygonal obstacles. In *Proc. 1st Annu. ACM Sympos. Comput. Geom*, pages 75–80, 1985.

[31] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[32] S. Koenig. A comparison of fast search methods for real-time situated agents. *AAMAS 04*, July 19-23 2004.

[33] R.E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artif. Intelligence*, 27(1):97–109, 1985.

[34] M. V. Kreveld, M. Overmars, O. Schwarzkopf, and M. de Berg. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[35] J. J. Kuffner and S. M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *Proceedings IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[36] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[37] J.-P. Laumond. Obstacle growing in a nonpolygonal world. In *Information Processing Letters*, volume 25, pages 41–50. 1987.

[38] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. In *Networks*, volume 14, chapter 3, pages 393–410. 1984.

[39] J. S. B. Mitchell. Geometric shortest paths and network optimization. In *Handbook of Computational Geometry*. Elsevier Science, 1998.

[40] K. Mulmulney. A fast planar partition algorithm. *Journal of Symbolic Computation*, 10:253–280, 1990.

[41] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[42] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29:669–679, 1986.

[43] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[44] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen. Solving checkers. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 292–297, 2005.

[45] D. Silver. Cooperative pathfinding. In *Proceedings of the 1st Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2005.

[46] D. Silver. Cooperative pathfinding. In *AI Programming Wisdom*. 2006.

[47] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. *AAAI*, pages 1392–1397, July 2005.