Guaranteeing Safety in the Presence of Moving Obstacles

Robert C. Kohout

ABSTRACT

Path planning is a fundamental problem in robotics research. Whether the robot is a manipulator arm in a factory floor, an unmanned all-terrain vehicle, a flying drone, or a household assistant serving coffee, the motions of the robot must be planned and executed in such a way that the robot can accomplish its goals. Motion planning must take into account the robot's inherent abilities to move and maneuver, its speed, and all of the various constraints imposed upon these abilities by the environment in which the robot is situated. Many real-world application domains are dynamic, in the sense that the plan-relevant parameters in the environment evolve over time. In such cases, motion planning must also take into account the time that it takes to plan. A perfect plan is useless if it cannot be produced in time to execute it in a changing world.

This technical report focuses upon the problem of avoiding moving obstacles in a 2-dimensional environment. Specifically, it addresses the problem of guaranteeing that a robot will never be hit by an obstacle in the environment. It establishes conditions for guaranteeing that a safety-preserving path will always exist in the most commonly studied problem in moving obstacle avoidance, known as the Asteroids Avoidance Problem. These results are then extended to less restricted, more realistic variants of the problem, including the important case where the locations and trajectories are only made known to the planning algorithm at runtime.

Once these conditions are established, they are used to develop an incremental algorithm that can solve the restricted Asteroids problem in low-order polynomial time. This algorithm takes its own observed worst-case running time into account, completes in a fraction of a second, and has been used to control Dodger, a simulated robot that avoids moving obstacles in hard real time. In over ten machine-weeks of testing, involving well over a million obstacles generated in a variety of ways, Dodger has not been hit by a single obstacle.

The research described in this report was conducted by the author in partial fulfillment of the doctoral degree requirements at the University of Maryland, College Park, under the direction of Prof.James A. Hendler. This research was supported in part by grants from the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), the ARPA I3 Initiative (N00014-94-10907), NSF (IRI-9306580), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), and ARPA contract DAST-95-C0037. Dr. Hendler is also affiliated with the UM Institute for Systems Research (NSF Grant NSF EEC 94-02384).

Chapter 1

Introduction

Many real-world applications require both intelligence and timeliness. Factory automation systems need to understand and manipulate complex operating environments, and quickly adjust when orders change, machines break or anticipated supplies are not delivered. Vehicle dispatching systems need to decide which vehicles to send where, and when, while adjusting to changing traffic conditions and customer demands. Military command and control systems need to acquire, integrate, and summarize huge amounts of complex, conflicting and often unreliable data in time to decide what actions should be taken on the battlefield. Highway vehicles must operate at high speeds in rapidly changing environments, and negotiate roadways fraught with hazards. In most cases, the bulk of information processing in each of these systems is currently done by human beings. In each case, people have long sought to automate large portions of the overall task. If these goals are ever to be achieved, the often conflicting requirements for intelligence and timeliness must be addressed.

This report brings together two major disciplines of Computer Science: Artificial Intelligence (AI), and Real-Time Computing. These two research areas conflict fundamentally. Real-Time Computing focuses upon the design of systems that perform with predictable precision, so that correct behaviors can be guaranteed. This sort of predictability is antithetical to the wide range of behavior that characterizes intelligence, and the classical methods of AI reflect this. Intuitively, intelligence involves deliberation, and deliberation takes time. More to the point, like people, a modern AI system will solve simple problems quickly, but more difficult problems take more time. It is possible to develop AI systems that return **some** result within a given time bound, but with very few exceptions, the quality of this result cannot be assured. This report takes a problem that has traditionally been addressed by conventional AI search techniques, and finds a solution to this problem that is **provably** adequate, both with respect to quality and timeliness.

1.1 Motivation

To be autonomous, a mobile robot must be able to plan its motions through time and space. Thus motion planning is a fundamental and well studied problem in mobile robotics. Whether the machine is a wheeled robot maneuvering through the corridors of a University, a manipulator arm on a factory floor, an unmanned vehicle negotiating rough outdoor terrain, or a deep space probe, billions of miles from earth, the problem of determining how to get from one place to another occurs over and over again in robotic applications.

Each application domain has its own unique geometry, and each robot has its own set of sensors and effectors. As a result, the field of motion planning is broad and many-faceted. This report focuses upon a particular kind of motion planning, namely, the problems associated with avoiding moving obstacles while maneuvering in a plane. Such problems arise naturally in a variety of real-world applications, and consequently, there has been considerable research in this area. The main contribution that this report makes to the existing body of research is its examination of the issues involved in **guaranteeing** obstacle avoidance in such domains.

It is easy to see why such guarantees are desirable. In most applications, getting hit by a car, a forklift, or other another robots, is a very costly and undesirable event. In addition, an important class of control systems, known as "mission-critical" real-time systems, is further complicated by the potential for catastrophic failure if the system does not produce acceptable behavior within strict deadlines. This makes it imperative that system operation can be proven correct before deployment. In particular, it is not sufficient that response times be "fast." When the cost of failure may be catastrophic, it is essential that we can **prove** that response time is "fast enough." Because of the potentially high cost of inadequate timeliness, **temporal adequacy** must be a fundamental component of the design and analysis of mission-critical systems. If mobile robots are ever going to be deployed in mission-critical applications, we need some way to guarantee that they will always be able to find and execute safety-preserving paths.

It also easy to see that, in practice, absolute guarantees of correct **system** behavior cannot be made. Sensors and effectors are never completely accurate, and software models will never completely capture all of the potentially relevant details of the world. The best that system designers can reasonably hope for is to minimize the overall possibility of failure. However, even though hardware and software are inherently limited, it is still useful to determine the theoretical limits and possibilities within a domain. In fact, if one desires to engineer a high-reliability system, it is necessary to first establish the theoretical bounds on system performance, and then endeavor to minimize the deviations from the theoretical optima in a realized system.

1.2 Contributions

The Asteroids Avoidance Problem can be stated as follows:

> Given a robot, at initial location R, that is capable of instantaneous, unbounded acceleration, up to some maximum speed V_r, and a set of moving obstacles $O_1, O_2, \ldots O_n$, that move at known, constant linear velocities, find a "safety-preserving" path that the robot can travel to avoid being hit by any of the obstacles.

This is the best known and most often studied problem in moving obstacle avoidance. This report establishes the theoretical conditions under which it is possible to guarantee safety in the Asteroids Problem. It then examines the ways in which the assumptions of the problem can be relaxed, while still guaranteeing avoidance, and establishes conditions for guaranteeing that a robot can always make progress towards its goals, while still avoiding all moving obstacles.

If a system has deadlines which absolutely must be met in order to assure system safety, the deadlines are known as "hard" deadlines, and a system designed to ensure that all such hard deadlines can be met is a "hard real time" system. In a hard real time system, overall system utility can go to zero when a deadline is missed. If system utility diminishes, but will still have some positive value when a deadline is missed, the deadline is said to be "soft." In order to ensure that all of the required resources are always available to a hard real time system, shared resources are typically **scheduled** , in such a way that even in the worst case, all of the required deadlines can be met. This generally means that some sort of calendar is created that maps shared resources to system processes as a function of time.

This work is groundbreaking because it is the first research in moving obstacle avoidance that takes the runtime of the algorithm into account when solving the avoidance problem. In order to guarantee avoidance, it is not enough to expect fast processors and good engineering to suffice to ensure timeliness. The running time of the algorithms must be accounted for in the control strategy that is used to avoid obstacles. To be useful, an avoidance algorithm must produce results before they become obsolete. As an algorithm is running, the world may change, and thus render any ongoing computation invalid. In all dynamic domains, the time required to find a solution is a fundamental part of the problem, but prior to this report this issue had never been addressed in the context of moving obstacle avoidance.

Dodger, a simulated robot that avoids moving obstacles in hard real time, was implemented based upon this report's theoretical results in obstacle avoidance. Dodger demonstrates that, in exactly the subclass of the Asteroids Problem where it is possible to ensure that safety will always exist, it is also possible to design and implement a reactive avoidance strategy that suffices to avoid obstacles in real time. Since its last bug was fixed, Dodger has avoided over a million obstacles in the course of over 60 days of real-time tests, conducted in a variety of different obstacle-generating scenarios, and **has not been hit by a single obstacle.** The algorithm employed by Dodger is complete, in the sense that a safety-preserving path to a goal is always found, and runs in a fraction of a second. This algorithm is $O(N \log N)$ worst-case, but empirical evidence suggests that actual performance is much better than this.

Most of the previous work in moving obstacle avoidance has focused upon a "static" problem. That is, the problem specifics are known to the algorithm when it begins, and they do not change afterwards. An implicit assumption in this approach is that such an algorithm can be easily modified to address the more realistic, "dynamic" case, where aspects of the problem become known to the control system over time, and are subject to change. This report establishes a fundamental difference between the static and dynamic variants of the Asteroids Problem. Necessary and sufficient conditions for guaranteeing safety in the static case are derived in Chapter 3 of this report. The proof of these properties is constructive, and suggests a straightforward algorithm for solving the problem when the requisite conditions are known to hold true. However, this algorithm is **not** sufficient to ensure safety in the dynamic version of the problem, where moving obstacles appear and disappear with the passage of time. While this report establishes sufficient conditions for ensuring obstacle avoidance, it does not show that these conditions are also necessary. In fact, it presents empirical evidence that strongly suggests that they are **not** necessary.

1.3 Brief Outline

This report contains seven additional chapters. Chapter 2 reviews the existing literature in robot motion planning. Chapter 3 develops the theoretical constructs that are required to ensure that a robot will always be able to avoid moving obstacles. Chapter 4 contains the algorithms required to implement these theoretical results in Dodger. Chapter 5 contains a description of a classical AI search algorithm that could be used to solve the static variant of the Asteroids Problem. Chapter 6 describes Dodger, a robot simulation that solves the Asteroids Avoidance problem in real time. Chapter 7 contains the empirical evaluations of Dodger and its algorithms. Chapter 8 summarizes this research effort, and discuss avenues for future work.

1.4 Detailed Outline

Chapter 2 reviews previous work in robot motion planning, including both time-invariant domains, where none of the obstacles change position over time, and time-varying domains, in which obstacles may move. Particular attention is paid to whether the problem is static or dynamic (i.e., whether or not all of the relevant information is known to the planning system at the time it begins to plan a path), and whether the algorithm is **complete**, in the sense that it will always find a solution when a solution exists. This chapter also examines the claims of real-time performance that are often associated with such path-planning systems.

Chapter 3 contains the main theoretical results of this report. I begin by establishing necessary and sufficient conditions for guaranteeing safety in the classical (static) version of the Asteroids Problem, and then I use these results to establish sufficient conditions for guaranteeing safety in the more natural, dynamic version of this problem. Chapter 3 then proceeds to examine ways in which the assumptions and restrictions of the Asteroids problem can be relaxed, while still allowing avoidance to be guaranteed. This chapter ends by establishing conditions for guaranteeing that the robot can always make progress towards its goal.

The proof for necessary and sufficient conditions in the Asteroids problem is constructive, so much of the avoidance algorithm can be derived directly from the proof. Chapter 4 addreses the one major algorithmic question raised by the proof, namely, how to find a "safe haven" location that the robot can move to, in two different way. First is an algorithm for deterministically finding a "safe haven", and then a Monte Carlo technique is also described. This chapter also contains a description of the goal achievement heuristics that are used in the Dodger system.

Chapter 5 contains a detailed description of a search algorithm for solving the static Asteroids Problem. Based upon earlier work, this chapter explicitly addresses many of the issues that had been treated as details in that work. It also develops the techniques and algorithms that are used for ensuring safety in the Dodger system.

Chapter 6 describes Dodger and the Maruti Hard Real-Time Operating System upon which it runs. It describes the particular variant of the Asteroids Problem that Dodger solves, including the various schemes for generating obstacles that were used to test Dodger. Chapter 6 then presents the system architecture, including the ways in which Dodger depends upon the services of Maruti.

In Chapter 7 presents the results of various empirical studies. It begin by presenting some results of the behavior of the complete search algorithm described in Chapter 5, proceeds to the empirical analyses of Dodger, beginning with a study of how Dodger behaves across a variety of obstacle generation strategies and goal achievement heuristics. This Chapter then analyzes both the worst-case and average-case runtime behavior of the avoidance algorithm. This chapter concludes with an empirical investigation of the necessary conditions for guaranteeing safety in the dynamic version of the Asteroids Problem.

Chapter 8 summarizes this report, and concludes with a discussion of possible ways in which this work can be extended in future research.

Chapter 2

Previous Work

2.1 Introduction

Motion planning has been a fundamental component of Robotics research from the time the first mobile robots were constructed in the late 1960's. 65 (65) describes what may be the earliest such robot, and this description includes a discussion of a graph-search algorithm used to determine the shortest collision-free path from the robot's location to a goal position. In the almost 30 years since that time, a large body of research has focused upon the problems associated with determining how to direct a mechanical robot from one location to another, subject to constraints imposed by the application environment and the mechanical constraints of the robot itself.

Broadly speaking, motion planning problems fall into two categories. **Gross motion planning** focuses on problems involving distances much greater than the size of the robot. **Fine motion planning** revolves around the problem of moving an object in a narrow space, where the required accuracy of motion approaches, or even exceeds, the positional accuracy of the robot. This report, and the rest of this chapter, is solely concerned with problems and issues implicit in gross motion planning. Throughout this report, the terms "motion planning" and "path planning" are used interchangeably, to refer to problems associated with planning the gross motions of mobile robots. There are a number of excellent surveys of gross motion planning research, including (39; 54; 82). The reader should refer to these for detailed descriptions of research that falls beyond the scope of this chapter and this report.

Adopting terminology from 39 (39), gross motion planning can be categorized as either **time-invariant** or **time-variant**, according to whether or not the obstacles in the domain can change position over time. This report focuses upon time-variant problems, which can be viewed as a proper superset of time-invariant problems, but I shall review the major results in time-invariant motion planning, since much of it is relevant to the time-variant case. Path-planning problems can also be either **static**, in which all of the obstacle information is known to the planner prior to planning, or **dynamic**, in which information about obstacles becomes known to the planner over time, and after the planning process in initiated. The vast majority of path-planning literature focuses upon the static case, but the vast majority of real-world path-planning applications fall into the dynamic case.

Most time-variant robot application environments are not of a nature which allows all of the relevant information to be known **a priori**, and yet the bulk of research into time-varying path-planning falls into the static category. Such work is based upon a generally unstated assumption, namely, that the time required to compute a solution is irrelevant to the problem. By definition, time-varying domains can change between the time that an algorithm begins, and the time that it finishes. However, this issue is rarely, if ever, addressed by researchers working within this paradigm.

The implicit assumption is that, if static solutions can be developed, then adapting them to more realistic, dynamic solutions will be relatively straightforward. However, as this report shows, there are fundamental differences between the static and the dynamic problems. In static problems, an algorithm has complete information, and any solution it finds to a given problem is guaranteed to suffice, due to the static nature of the domain. In contrast, dynamic problems must take into account not only the events known to have occurred, but also any events that might **possibly** occur. Thus the set of solutions is more constrained in the dynamic case.

Section 3.1.1 (page 15) develops necessary and sufficient conditions for guaranteeing safety in the best known and most commonly studied problem in time-varying path-planning. However, this result does **not** transfer directly to the dynamic case. Section 3.1.2 (page 19) establishes sufficient conditions for the dynamic version of this problem, which are qualitatively different from those in the static problem. The fact that these conditions have not been shown to be necessary does not contradict the contention that the static problem is fundamentally different from the dynamic problem. Even if it does eventually prove to be possible to guarantee safety under the same conditions in both the static and dynamic versions of the problem, it is definitely **not** the case that the algorithms developed in this report to ensure safety in the static version of this problem also suffice in the dynamic case.

2.1.1 Real-Time Motion Planning

There has been significant research in the area of dynamic motion planning. Typically, such studies are associated with the control of actual robots, and so the convenience of the static assumption cannot be adopted. Many of the papers in this area purport to describe real-time algorithms. With only a very few notable exceptions, these real-time claims stem directly from the dynamic nature of the problem. In other words, most work in "real-time path-planning" is more correctly categorized as research in dynamic path planning, since the real-time aspects of the work are confined to the fact that data is gathered "online", after system execution has begun. Such claims are not atypical, and stem generally from the looser definition of real time adopted outside of the real-time systems community. As 32 (32) put it

> A strict definition of real time is that the system guarantees that tasks will receive up to their worst-case runtime and resource requirements, which presumably means that tasks will produce the best possible results (highest quality results). In real-time AI the focus is usually on high-level goal achievement, rather than worst-case requirements. For this reason, often in real-time AI less strict definitions of real time are used. One common (and usually implicit) definition is that the system will statistically (e.g., on average) achieve the required quality value by the required time, but no guarantee is made about any particular task[1].

Inside of the real-time systems community, it is recognized that not all systems are **hard real time** Such systems are constrained by hard deadlines, which absolutely must be met to avoid catastrophe. These systems are also known as **mission-critical** systems. Systems that are shown to meet some explicitly stated statistical performance properties, where timeliness is not strictly guaranteed, are often referred to as **soft real time** systems. Systems where the statistical properties are implicit, unexamined and unstated, are known as **coincidentally real time** systems. This usage will be adopted in this report. According to this terminology, most real-time path-planning algorithms are coincidentally real time.

By contrast, the Dodger system described in this report, and the control algorithms it employs, are real time in the strong sense of the term. Dodger guarantees that it will avoid moving obstacles, it takes its own running time into account, and it solves a restricted version of the most commonly

---

[1] (32, page 318).

studied problem in time-variant obstacle avoidance. The restrictions Dodger imposes on the domain are natural, reasonable and feasible. In addition, they are based upon the restrictions necessary to ensure that a safety-preserving path exists in the static version of the problem. The rest of this chapter reviews many of the major problems and results in gross motion planning, for both time-variant and time-invariant problems.

2.2 Static Obstacles in the Plane

Time-invariant problems were the central focus of path-planning research until the mid-1980's, and this remains an active research area. While initially focused upon the problems associated with moving a robot in a 2-dimensional space, path-planning research has since been generalized to include problems in N-dimensional "configuration space." The primary benefit of this generalization is that it has allowed researchers to develop upper-bounds for problems as diverse as the control of robot manipulators in factory environments, and navigation by an off-road vehicle over rough terrain, in both time-varying and time-invariant domains. However, it turns out that no general-purpose solution to the abstract problem has been found that has allowed researchers to move away from more narrowly focused solutions to particular problems. Thus, while there is a large body of literature developed around the more abstract problem, it is dwarfed by the work done on various aspects of the problem of manipulating a mobile robot through a set of immobile obstacles situated in a 2-dimensional space.

The abstract problem of moving a robot through a set of stationary obstacles is known as the "Piano Movers Problem" (76), or just the "Mover's Problem". It is also refered to the "Findpath Problem". Work by Lozano-Pérez (in, e.g., 56 (56), 57 (57)), extended the scope of the problem from the planar path-finding problem addressed by 65 (65), to a more general class of path-finding problems in **configuration space**. The configuration of an object is a set of parameters which, when taken together, are sufficient to describe the position of every point on the object. The set of all combinations of such parameters is the configuration space of the object. Obstacles are modeled as constraints on a robot's position in configuration space, and the space not occupied by obstacles is referred to as the **free space** of the robot. In addition to physical objects, this formalism allows any constraints on a robot's position, orientation or other configuration parameter, to be modeled as an obstacle in configuration space.

The roughly 30 years of research in this area can be partitioned into three or four basic approaches. **Roadmap methods** map the uncountably large set of possible robot motions into a network of one-dimensional lines, and thus reduce the problem to some form of graph search. **Cell decomposition** partions space into small cells, and searches for a path from the cell containing the starting position to a cell containing the goal location, where adjacent cells in the path adhere to some set of adjacency constraints. **Potential field methods** create "potential functions" that are designed to repel the robot from obstacles and towards the goal. It is generally recognized that, under the standard view of potential fields, they are best suited for the local decision-making associated with obstacle avoidance, and not particularly good at helping a robot achieve longer term goals. As a consequence, there has been research into the utility of **hybrid planners**, that integrate fast, "reactive" avoidance behaviors with some sort of more deliberative goal-achievement capability.

2.2.1 Roadmap Methods

Most roadmap methods are descended from the visibility graph approach employed in (65). Roadmap methods tend to be **complete**, in the sense that they are guaranteed to find safety-preserving paths when such paths exist. The basic approach is to collapse the uncountably large set of possible paths into a much smaller, usually finite, network graph, such that a path from a fixed location to a second goal location will exist on the network if and only if such a path exists in the larger set of all possible paths. 58 (58) are generally credited with formulating the "standard" problem, which involves manipulating a translating, but non-rotating, point robot in the plane. Representing the robot as a point can be accomplished via a technique introduced by 80 (80), whereby the polygonal obstacles in the environment are expanded by computing their Minkowski set difference with the robot. After this translation, the unobstructed paths in the transformed version of the problem are exactly the same as those in the original.

The size of a visibility graph is $O(N^2)$ in the number of obstacle vertices, and 4 (4) have shown how to find it in $O(N^2)$ time. 34 (34) describe an algorithm that takes time $O(K + N \log N)$, where K is the number of edges in the graph, and is typically much smaller than the worst-case of N^2. Once such graphs are constructed, they can be searched using standard graph-search techniques, such as A*, that support the efficient determination of resource-optimal paths, such as shortest time, shortest distance, etc.

A drawback with visibility graphs is that the vertices of the networks upon which they are based are obstacle vertices. Even when the obstacles have been "grown" from the original obstacles using the method of 58 (58), the implication is that such paths move the robot arbitrarily close to collision. While is can be shown that time and distance-optimal paths are a subset of such a network, it is often desirable to keep the robot at a safe distance from the obstacles. 66 (66) introduced the **retraction** method for motion planning, which uses generalized Vornoi diagrams to define networks that maximize the clearance between the robot and obstacles it must pass in order to get to the goal. The network defined in this fashion can be computed in $O(N \log N)$ time, and searched in the same way as a visibility graph. 13 (13) developed an $O(N^3)$ algorithm, based upon what he calls "generalized cones", in which "The paths are good in the sense that the distance of closest approach to an obstacle over the path is usually far from minimal." While the asymptotic runtime of this algorithm is not as good as that of 66 (66), and the performance properties much more vague, its actual runtime in uncluttered environments is apparently quite good. This algorithm was subsequently improved in 52 (52), by using a union of complex cones and convex polygons to represent free-space.

18 (18) developed the **silhouette method**, that generalizes the roadmap method to arbitrary dimensions. Basically, obstacles of n dimensions are projected onto curves of $n - 1$ dimensions, recursively, until a searchable network of 1-dimensional lines is produced. While the primary utility of this approach was to support a theoretical analysis of the complexity of motion planning, a version of this algorithm was implemented in (55). 7 (7) have recently analyzed roadmap methods, and established an upper bound on the problem that is $O(d^{k^2})$, where d is the maximum degree , and k is the number of variables in the semi-algebraic polynomials defining the search space.

2.2.2 Cell Decomposition

The basic idea of cell decomposition is to partition unoccluded space into a set of non-overlapping **cells**. When the union of these cells is exactly equal to the unoccluded space, we say the cell decomposition is **exact**, otherwise, it is **approximate.** Exact cell decomposition was used by 73 (73), to determine feasible paths for a line segment ("ladder"), which was free to rotate in the plane. In (74), this technique was used to establish a general-purpose, albeit very inefficient, path-planning method. The basic technique is to use algebraic curves to represent a robot's free space, and the primary benefit of this approach has been to establish theoretical upper-bounds for general classes of problems. As 54 (54) puts it, the upper bound on complexity which is established by 74 (74) is

> A free path in a configuration space of any fixed dimension m, when the free space is a set defined by n polynomial constraints of maximal degree d, can be computed by an algorithm whose time complexity is exponential in m, and polynomial in both n

("geometrical complexity") and d ("algebraic complexity") [2].

75 (75) give an $O(n^{2+\epsilon})$ algorithm for static planning problems in 3-dimensions, for any $\epsilon > 0$, by using an exact cell decomposition approach which they call **vertical decomposition**. This approach can be used for time-variant motion planning in 2-dimensions, but only if the maximum velocity of the robot is assumed to be unbounded.

A more practical approach, in terms of absolute runtime performance, has been to decompose the plane into a set of regular geometric shapes (e.g.,squares, rectangles, or hexagons), which do not generally allow the unoccluded free space to be represented exactly, and hence the name, "approximate" cell decomposition. As with exact cell decomposition, an adjacency relation is used to construct a graph which can then be search for a path. Most often, this approach is done hierarchically, and based upon rectangloids. The workspace is partitioned into cells at some initial level of granularity, and then cells are marked as either unoccupied, occupied, or "mixed" (neither totally occupied nor totally unoccupied.) Depending upon the algorithm, mixed cells can then be further partitioned, and the process repeated down to some predefined maximum granularity, or the graph can be searched for a solution prior to refinement.

Standardizing the shape of the cells greatly simplifies their computation, and in practice these methods have proven quite efficient. However, unlike exact cell decomposition techniques, which are complete, approximate cell decomposition may fail to find feasible paths. Approximate cell decomposition has been used in 2-dimensional static environments by a number of different authors, including 56 (56), 12 (12), and 42 (42). 81 (81) and 37 (37) extend the concept to 3-dimensional workspaces, and 56 (56) uses it for an abstract manipulator in a static environment.

2.2.3 Potential Field Methods

Another general approach to path planning involves defining a heuristic function defined on every represented position in free space. This function simply maps a robot's current location into a direction which the robot should move in order to make progress towards the goal. The most commonly used heuristics are known as potential field functions. Potential fields have low values at the goal, and high values at obstacle boundaries. Ideally, at all other points, they have intermediate values such that all paths slope towards the goal. Under this model, the robot's path-planning choice at any given location is simple: examine the immediate vicinity, and move to the location with the lowest potential field value.

The primary advantage of potential field functions is that they are very fast to compute. In fact, early applications of this technique to path planning can be found in (45; 46), in which a potential field function is employed to achieve collision avoidance in real time. This is one of the few studies in path planning that describes an algorithm that is real time in the strict sense of the term. The runtime performance of the algorithm and the frequency at which it is run are shown to be sufficient to ensure that the delay between sensing and servo-activation is sufficiently small in the worst-case that the results of the algorithm can always be employed to avoid obstacles.

50 (50) describes a potential field function that takes the robot's velocity into account, along with positional information. 2 (2) employs a potential field function that can be adapted dynamically at run time to reflect the changing goals of the planning agent, and uses this onboard an experimental robot. 14 (14) also uses a potential field function onboard an experimental robot. 63 (63) present a principled approach to the synthesis of potential functions, so that actuator limits can be accounted for in a general fashion. 10 (10) describe a potential field function, based upon a data structure instantiated at run-time, that they call a "virtual force field" (VFF). The author's real-time claims about this approach are based solely upon the dynamic nature of the VFF. By the terminology adopted above, this system is coincidentally real time.

---

[2] (54, page 35).

Potential fields define a gradient over the workspace, and then gradient descent is used to find a path to the goal. 49 (49) identify four drawbacks to the general approach of potential field methods:

- Robots can become trapped in local minima.
- Potential field functions do not generally allow a robot to travel through narrow passages.
- Robots can oscillate between obstacles.
- Robots can oscillate in narrow passages.

11 (11) attempt to address these problems by extending the VFF approach to a new data structure called "vector field histograms"(VFH's), which addresses the oscillation problems, as well as the difficulties imposed by narrow passages. VFH's are coupled with a path monitor that detects traps, and a global path planner that directs the robot efficiently towards its goals. 51 (51) address the issue of local minima in a similar fashion, by integrating an artificial potential function with a cell decomposition-based technique which they call "path relaxation." 47 (47) provides a topological analysis of the conditions under which a potential field **cannot** be constructed so that no local minima exist. He then offers a class of "almost global" potential functions for a constrained version of the problem, for which all but a small subset of the points of the domain are on a gradient towards the goal.

2.2.4 Hybrid Path Planning

47 (47) has a more general perspective of potential fields than do most researchers. His work focused upon determining the limitations of potential field techniques. Thus, he assumes static problems, where the obstacle locations are known prior to runtime, and an unbounded computation that can be made "offline", prior to runtime, to determine the best possible potential field function to use at runtime. Configuration space is decomposed into cells of some fixed granularity, and then the set of all pairwise paths computed and stored in a data structure that enables a robot to quickly determine "where to go next", from any location, given any destination cell. This seems to be the basis of the gradient field approach suggested by 68 (68), but they provide insufficient detail or citations to be certain. However, these authors quite clearly distinguish between their gradient fields and potential field functions. As they put it,

> Another vector field type of representation, the artificial potential field, appears superficially very similar to the gradient field and it is also used for navigation and obstacle avoidance (2; 45; 50)...The computation of potential fields is generally based upon the superposition model, in which charges are distributed such that repulsive forces are generated near obstacles, and attractive forces are generated near goals...The resultant field, however, does not represent an optimal path, and may easily contain local minima and traps[3].

Under this prevailing view, potential functions are inherently local. Thus it is no surprise that they are much better at avoiding local obstacles than they are at steering a robot towards a distant goal. Incorporating global information takes time, and potential functions tend to be used precisely when time is of the essence. Research into the more strictly defined sort of potential fields has been largely abandoned, in favor of hybrid systems that use some sort of potential function for obstacle avoidance, and some other mechanism to steer the robot towards its goals. 59 (59) describes a promising decomposition approach that finds "channels" which are constructed so that gradient descent methods will take the robot to its goal. 3 (3) defines an architecture

---

[3] (68, page 1376).

that allows the global influences of a robot's behavior to be taken into account by the more time-critical,"reactive" behaviors, which can also be viewed as potential field functions. 6 (6) integrate potential field functions with randomized search techniques, and 54 (54) describes a technique developed by 25 (25), which uses statistical methods to "learn" to avoid local minima. 22 (22) describe a potential field approach that explicitly addresses uncertainties in position, orientation and control. 60 (60) develop a hybrid architecture for integrating symbolic, reactive planning with control theoretic approaches to motion control.

In recent years, a number of authors have focused explicitly on the problem of obstacle avoidance, implicitly recognizing the fact that there is a place for both rapid, local reaction and more deliberate, global planning in most real-world planning applications. Many of these, such as (64), (40), (38), and (21), make some sort of real-time claim. For example, 21 (21) describe an impressive system for using visual information to recognize and avoid obstacles while the robot travels at relatively high speeds. As evidence of the real-time capabilities of the system, they state

> The robot has wandered around the lab at 30 cm/s for as long as 20 minutes without collision. To our knowledge, this is the first such demonstration of real-time wandering using only image motion cues.

Typically, as is the case with each of the papers cited above, the real-time nature of the approach stems from the dynamic nature of the problem. In other words, when not all of the obstacle data is known prior to plan execution, and must be somehow sensed at run-time, especially when a robot is simultaneously moving and avoiding obstacles, researchers often claim that their system is real time. (Not always, of course. For example, 27 (27) describe a technique used aboard a robot that safely navigates at 95 cm/sec in cluttered environments, and they make no use of the term "real time.") As noted above, in the absence of any statement of conditions required to maintain safety, whether they be hard or soft constraints, and the performance of the system with respect to such constraints, this is real-time avoidance in the loosest "coincidentally real time" sense of the term. As 61 (61) put it

> Real-time computing is **not** about building "fast" systems; it is about building systems that are predictably "fast enough" to act on their environments in well-specified ways.

There is no doubt that the primary utility of potential field functions, as well as the avoidance components of hybrid path planners, is derived from the fact that they are relatively fast to compute. However, there are very few papers in the path-planning literature that directly address the question of "fast enough". This becomes a fundamental issue when one considers the problems associated with avoiding moving obstacles.

2.3 Moving Obstacles

Many of the path-planning approaches referred to in the previous section were developed to apply in multi-dimensional domains, since even in time-invariant applications, the dimensionality of configuration space is generally greater than the three dimensions of the physical environment. Since a moving obstacle in n-dimensions can be represented as a static obstacle in $(n+1)$-dimensions, one might expect that these earlier results will naturally apply to time-variant domains. However, the temporal dimension is fundamentally different from the other dimensions in most configuration spaces, since it can be traveled in only one direction. It may be possible to modify some of the algorithms developed in the previous section to account for the unidirectionality of time, but there are no overt examples of this in the literature.

2.3.1 Static Path Planning in Time Variant Domains

Most of the research into path planning in time-varying domains has been on the static version of the problem, where all of the information required to plan is known to the planner in advance. This is the form of the problem introduced by 69 (69) as the **Asteroids Avoidance Problem**. In this problem, a point robot (in 1, 2, and 3 dimensional space) must find a path from a start location S to a goal location G, while avoiding a set of n constant velocity, polygonal obstacles. 69 give an algorithm for solving this problem that is exponential in the number of obstacles. In Section 3.1.1 (page 15), the conditions under which it is possible to guarantee safety in this problem are established. Chapter 5 (page 46) describes the Reif and Sharir algorithm in depth, and explicates many of its sub-problems that are treated as details in the original paper. The algorithm of (69) is complete, in the sense that it is guaranteed to find a path if it exists, and this early paper established an upper bound on the problem. In this paper, it is also shown that the time-invariant path-planning problem in three dimensions is $\mathcal{PSPACE}$-hard when the robot's ability to change velocity is bounded, and $\mathcal{NP}$-hard when it is not bounded. 17 (17) established that the Asteroids problem is $\mathcal{NP}$-hard in cases where the robot's velocity is bounded.

43 (43) present a heuristic solution to a slightly more general problem, in which obstacle trajectories are not constrained to be constant velocity, but are still known, and presumably computable. They also assume that some non-trivial set of obstacles will have zero velocity, and then decompose the problem into a two step process where 1) they find a collision-free path through the static obstacles and 2) they then attempt to find a set of speed assignments for the robot such that it can travel this path and avoid all moving obstacles. This approach is inherently incomplete, and is not guaranteed to find a path even when it exists. 28 (28) extend this approach to consider a set of "adjacent" paths, to prepare for cases in which step 2) is not successful.

24 (24) describe another heuristic technique for planning the motions of several moving objects in the plane. To do so, they treat the problem of planning the motion of a single moving object in the presense of time-varying obstacles as a sub-problem. Their solution is to partition the 3-dimensional configuration time-space into a finite set of 2-dimensional planes, and search for time-increasing paths in this partitioning using a visibility graph technique. Other than the fact that they consider the problem of planning several motions (which they "solve" by simply commiting to some heuristic, "priority" ordering), this approach is little more than an inexact, and hence incomplete, version of Reif and Sharir's exponential search. While it may be faster than the complete algorithm, since it considers fewer possible paths, it is still exponential in the number of obstacles.

29 (29) use a version of Reif and Sharir's algorithm to solve what is essentially the Asteroids problem, except that obstacles are allowed to move along piecewise linear trajectories. This algorithm runs in $O(n^2 \log n)$ time, but assumes that the robot's velocity is fixed and known in advance. This is useful when the robot happens to be able to move faster than all of the obstacles. 31 (31) exploit the triangle inequality to show that, in cases where the robot moves faster than all obstacles, a safety-preserving path to a goal exists, if and only if a safety-preserving path to the goal exists in which the robot always travels at its maximum speed. However, in cases where even a single obstacle can move faster than the robot, this property does not hold, and the utility of Fujimura and Samet's "accessability graphs" is greatly diminished, since it is not at all clear how it would be used to search over the set of all possible robot velocity assignments. Again, Chapter 5 (page 46) describes an algorithm based upon Reif and Sharir's seminal work, which generalizes the accessibility graph concept of 29 (29) to account for the fact that a robot typically has uncountably many choices of velocities that it may travel to get between two locations. In the general case, each of the nodes in the accessibility graph must be expanded from points in time-space to intervals in time-space. However, this introduces complications. Section 5.3 (page 49) develops a solution for the problem of finding what 29 call a **collision front**, in the case where the robot originates not from a point, but from an interval in time-space.

30 (30) use a cell-decomposition technique to solve a time-varying path-planning problem. Like all approximate cell-decomposition techniques, their solution is incomplete, and may fail to find a path, even when it exists. In order to ensure that paths are found, this technique should use relatively small cells. However, the search algorithm is exponential in the size of the cell-decomposition, in the worst-case. Thus, this approach is both incomplete, and asymptotically worse than the upper-bound established by 69 (69).

78 (78) show that, when obstacles are polygons constrained to move along piecewise linear trajectories, then free space can be represented by a closed set of non-overlapping polytopes, and that the algebraic nature of these polytopes can be exploited to use linear optimization techniques to search the space of potential paths. The principle advantage attributed to this approach is that it allows search for near-optimal paths, where optimality can be represented by an arbitrary linear constant. However, 20 (20) have shown that this algorithm is not complete, and thus it is not clear how much of an advantage this actually is. In addition, the algorithm is worst-case exponential in the number of obstacles, and the runtime performance for even small numbers of obstacles appears to be quite poor.

2.3.2 Dynamic Path Planning in Time-Variant Domains

The dynamic nature of time-varying domains has very rarely been considered in the path-planning literature, despite that fact that it is usually not the case that all of the requisite information will be available to the planner prior to run-time. 44 (44) focus upon the problem of how to avoid moving obstacles when their trajectories are sensed and must be extrapolated from observations. This work was very preliminary, used only two moving obstacles to test the concept and was concerned only with avoidance, and not goal achievement. 53 (53) present a systematic study of the problems inherent in **collision detection**, i.e., sensing and estimating the position and trajectories of moving obstacles, which could conceivably be the basis of a real-world, online path-planning system. They do not address the issue of finding paths, or even of avoiding collisions once they are detected.

35 (35) also addresses the problem of avoiding moving obstacles with unknown trajectories, where the trajectories are assumed to be piecewise linear, and the number of segments finite. He presents an $O(N^5)$ algorithm for solving this problem, where N is the number of obstacle vertices. However, the problem he addresses is one of navigating a point robot through a set of (presumably Minkowski sum-expanded) polygonal obstacles, which are constrained to never collide. This constraint amounts to requiring that no two moving obstacles ever move so close together that the robot cannot fit between them. 79 (79) outline a hybrid cell-decomposition/potential field technique for dynamic plan-planning in the presence of moving obstacles. Their algorithm on $O(p)$, where p is the length of the plan, but it assumes that there is one SIMD processor for each cell in the decomposed space. Like other potential field approaches, this approach is not complete, and best suited for avoidance, as opposed to goal achievement. 77 (77) analyze the expected path-execution time of path deviations made in response to the unexpected appearance of moving obstacles, when the set of “shelter” locations is known in advance.

2.4 Relationship of this Thesis to the Existing Literature

This report, examines the problem of guaranteeing safety in a dynamic, time-varying domain. It establishes necessary and sufficient conditions for guaranteeing safety in the static Asteroids Avoidance Problem, as well as sufficient conditions for doing so in the dynamic version of the problem. In all of the literature reviewed above, the path-planning problem has focused upon finding a solution **if a solution exists.** Prior to this report research, no one had addressed the issue of guaranteeing that a solution always exists.

This thesis describes Dodger, a system that solves a restricted version the dynamic Asteroids

Problem in hard real time. The domain restriction is natural, and derived directly from the constraints required to ensure that a safety-preserving path will always exist. When N is the number of obstacles known to the robot at any given time, the algorithm employed in this system is $O(N \log N)$ in the theoretical worst-case, but the observed worst-case behavior is better than linear in N. The avoidance algorithm has many of the advantages of potential field functions: it runs quickly and frequently, making decisions based upon its current view of the environment. However, this view is not local, and while the implemented system does not explicitly search through time, temporal projections are implicit in its computations. Based upon my review of the existing literature, this is one of the very few systems in path planning that is real time in a strict sense: it takes its own running time into account, and it is structured to ensure that decisions are always made quickly enough to guarantee that obstacles are avoided. It is the **only** work that has produced hard real-time solutions to problems in dynamic, time-variant motion planning.

Section 3.4 (page 28) shows that it is possible to guarantee that the robot can always make progress to its goals. The results of this section are the basis for one of three different goal-achievement algorithms that Dodger has been tested with. When Dodger uses this "heuristic," the overall algorithm is complete, since Dodger is guaranteed to eventually reach all of its goals, and it is also guaranteed to avoid all moving obstacles.

Chapter 3

Theoretical Foundations

It is tempting to adopt the following argument: the Asteroids problem has been shown to be $\mathcal{NP}$-complete (17; 18), so for even moderately large problems, it is not feasible to find solutions at all, let alone in real time. This chapter will show that even though the Asteroids Problem has been shown to be hard in the general case, finding solutions in low-order polynomial time is possible when the problem is restricted to ensure that solutions will always exist. This restriction turns out to be both natural and practical. The theoretical results developed in this chapter provide the foundation for the real-time capabilities demonstrated by Dodger. Chapter 4, will describe the algorithms that build upon the results of this chapter to allow Dodger to guarantee obstacle avoidance in real time.

3.1 The Asteroids Problem

This section will begin by considering a variant of the Asteroids Problem that is essentially the classical Asteroids Problem of, e.g., (69),(17) and (31). Typically this problem is cast as one of finding a path to a goal from a starting location, while avoiding moving obstacles. This is not strictly necessary: since all of the obstacles are moving in a straight line, it suffices to show that the robot is capable of getting to some one safe location, at which it can stop indefinitely without being hit. If this is the case, then eventually there will exist a linear, constant velocity, collision-free path to **any** goal location in the plane.

Starting with this statement of the problem, necessary and sufficient conditions for guaranteeing that the robot can avoid the obstacles are established. In subsequent sections, the implications of relaxing the assumptions of this form of the problem are examined. The first relaxation addressed is the "dynamic" asteroids problem, where the total number of obstacles that appear over time is unbounded, but the number of moving obstacles that are within some known distance of the robot at any given time is fixed. Then the problems presented by static obstacles are examined, and the techniques developed in the earlier parts of this chapter are extended to account for obstacles that do not move. The problem of accounting for the time it takes the robot to turn and accelerate is also addressed, as is guaranteeing safety when the trajectory of the obstacles is uncertain. The chapter concludes the by showing that it is possible to avoid obstacles while at the same time ensuring that the robot can make progress towards some goal location in the plane. [1]

3.1.1 The Static Asteroids Problem

For the purposes of this section, the "static" Asteroids Problem can be stated as: Given a robot, at initial location R, that is capable of instantaneous, unbounded acceleration, up to some maximum speed V_r, and a set of moving obstacles $O_1, O_2, \ldots O_n$, that move at known, constant

---

[1] Major portions of Section 3.1.1 and Section 3.1.2 have appeared in (48).

linear velocities, find a "safety-preserving" path that the robot can travel to avoid being hit by any of the obstacles.

The fundamental question which addressed in this section is:

> Under what conditions can one guarantee that 1) a safety-preserving path exists which will allow the robot to avoid being hit by any obstacle and 2) we can compute the path in time to execute it?

It is not obvious that such restrictions exist. However, shall be demonstrated, once we fix the number, speed and sizes of the obstacles, and the maximum speed of the robot, the obstacles can always be avoided, so long as they are each initially some minimum distance away from the robot. This distance will be refered to as the **threat horizon**, H. Ideally, we would like to make H as small as possible. Furthermore, it is not enough to know a path exists, we must be able to find it, and find it quickly enough to be able to execute it before being hit. These issues are all addressed in the proof of Theorem 1 below.

For the purposes of this section, the robot will be modeled as a single point and the obstacles as circles with diameters $d_1, d_2, \ldots d_n$. Most path-planning literature assumes that the robot and obstacles can be bounded by polygons. Reducing the robot to a point is a standard technique developed by 58 (58). It turns out that a solution in the case where the robot is a convex polygon is equivalent to the case where the robot is a point and the sizes of the obstacles have been increased by the size of the robot. However, this technique can only be used when the obstacles do not rotate. Since a polygon that does not rotate can be bounded by a circle, and a polygon that does rotate can also be bounded by a (possibly larger) circle, centered at the center of rotation, circles are used to represent obstacles in order to simplify the presentation, and actually gain some generality.

Theorem 1: Let R be a point representing the location of a robot at time t_0 in a 2-dimensional Euclidean plane. Assume that the robot can rotate and accelerate instantaneously, but is limited by a maximum speed V_r. Let $O_1, \ldots, O_n$ be a set of n circular obstacles with diameters $d_1, \ldots, d_n$ which move at known, constant velocities $v_1, \ldots, v_n$. Let V_o be the largest of the v_i. Let W be the sum of the widths of the obstacles, i.e., $W = \sum_{i=1}^{n} d_i$. If each of the obstacles is initially a distance greater than $\frac{W(V_o+V_r)}{2V_r}$ from R, then there exists a "safe haven" point S such that none of the O_i will touch S at any time, and the robot can move at constant linear velocity from R to S without intersecting any of the O_i.

Proof: Let Φ_i be the space occupied by obstacle O_i, from time t_0 to t_∞. Since the obstacles move along linear paths, Φ_i is comprised of all of the space between two parallel rays, separated by a width of d_i. Since the space which lies between two parallel lines is called a **plank**, this region shall be called a **half-plank** of width d_i.

If each of the obstacles begins at a distance greater than $W(V_o + V_r)/2V_r$ from R, then for each obstacle O_i, there must exist a positive number $\hat{\epsilon}_i$, such that O_i begins **exactly** $W(V_o + V_r)/2V_r + \hat{\epsilon}_i$ from R. Let $\epsilon = 2V_r\hat{\epsilon}_i/(V_o + V_r)$. Then ϵ is positive, $\hat{\epsilon}_i = \epsilon(V_o + V_r)/2V_r$, and O_i begins a distance $(W + \epsilon)(V_o + V_r)/2V_r$ from R.

Since the shortest distance between two points in a Euclidean plane is a straight line, the earliest time that one of the obstacles could intersect the robot would be in the case that the closest obstacle is moving at speed V_o, and it and the robot move straight towards each others, as depicted in Figure 3.1. In other words, the worst case is when that obstacle O_n for which $\hat{\epsilon}_n$ is the smallest of the $\hat{\epsilon}_i$, travels at speed V_o and heads directly towards R, while the robot travels a straight-line path towards O_n, at its maximum speed V_r. In this case, O_n and the robot would collide at time $t_0 + ((W + \epsilon)(V_o + V_r)/2V_r(V_o + V_r))$, which is just $t_0 + (W + \epsilon)/2V_r$.

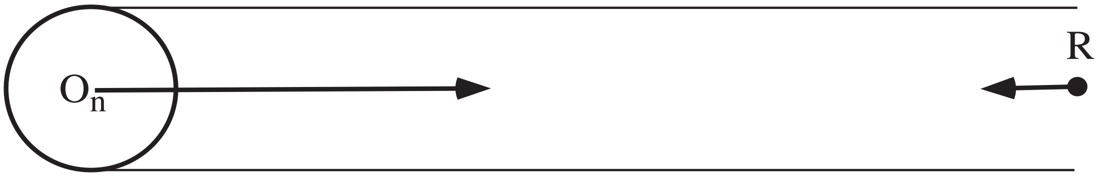


Figure 3.1: Earliest-Collision Scenario

Now consider the region which comprises all of the points to which the robot could move by time $t_j > t_0$, while never moving at a speed greater than V_r. This area is just a circle, with radius $V_r(t_j - t_0)$. It follows that the area which comprises the locations to which the robot could travel before it could possibly be hit is a circle centered at R, with radius $V_r(t_0 + (W+\epsilon)/2V_r - t_0)$, which simplifies to $(W+\epsilon)/2$. We shall call the distance $(W+\epsilon)/2$ the **safety radius**, and the circle of this radius centered at R the **safety region**.

Now it must be shown that the n half-planks, $\Phi_1, \ldots, \Phi_n$, cannot completely cover the safety region. This follows from the 2-dimensional version of Bang's solution to Tarski's "plank problem" (5), which states[2]:

Theorem 2 (Bang): Let L be a convex body of minimal width l in a Euclidean plane of 2 dimensions. If L is contained in the union of p planks of widths $h_1, \ldots, h_p$, then $h_1 + \cdots + h_p \geq l$.

Clearly, the set of objects which can be covered by planks is a superset of the set of objects which can covered by half-planks. Since the safety region is a convex body of width $W + 2\epsilon$, by Bang's theorem, in order for the n half-planks to cover the safety region, $\sum_{i=1}^{n} d_i$ must be greater than or equal to $W + 2\epsilon$. But, by definition, $W = \sum_{i=1}^{n} d_i$, and ϵ is positive, so there must be some area within the safety region which is not covered by the Φ_i. This proves the existence of S. To see that the robot can move from R to S without being hit, one only need remember that the safety radius was defined so that it is possible for the robot to move anywhere in the safety region by the time the first obstacle reaches its perimeter. **Q.E.D.**

This proof satisfies the existence criteria. The robot will also need to be able to determine **where** a safe haven is in time to get to it, without compromising safety. Chapter 4 will describe one way to do this. For now, it is sufficient to note that this can be done in $O(n^2 \log n)$ time, where n is the number of planks. Once the actual safe haven is determined, computing a path to it is trivial, since the theorem guarantees that there is a straight-line path to the safe haven, and the robot is capable of traveling this path before it could possibly be hit by an obstacle.

To ensure that the robot can compute a solution in time to safely use it, we need to know the **actual** worst-case time required to compute the solution. Call this time t_s. If we increase H by the maximum distance an obstacle can travel in t_s, then we ensure that the system will have sufficient time to compute and execute the solution. Thus, the threat horizon, should be

$$V_o * t_s + W(V_o + V_r)/2V_r$$

to guarantee timeliness. Note that a similar argument can be used to account for robot rotation and acceleration times, in the more realistic cases where acceleration and rotation are not instantaneous.

3.1.1.1 The Necessity of the Threat Horizon

The previous subsection showed that constraining obstacles to begin their travels outside of the threat horizon H was sufficient to ensure the safety of the robot. This subsection will show that

---

[2] This theorem generalizes to n-dimensions.

the bound is tight, i.e., for any $H' < H$, there exist choices of V_r and V_o, and a configuration of obstacles such that a collision cannot be avoided.

Note that the size of the safety region depends only upon the sizes of the obstacles, while H also depends upon the ratio of V_o to V_r. This leads to the following theorem:

Theorem 3: Let R be a point in a 2-dimensional Euclidean plane, which represents the location of a robot at time t_0. Assume that the robot can rotate and accelerate instantaneously, but is capable of only a maximum speed V_r. Let $O_1, \ldots, O_n$ be a set of n circular obstacles with diameters $d_1, \ldots, d_n$, which move at known, constant velocities $v_1, \ldots, v_n$. Let $W = \sum_{i=1}^{n} d_i$, and let δ be a positive constant, such that $\delta > 2WV_r/V_o$. If at time t_0 obstacles are allowed to start a distance of no more than

$$(W - \delta)(V_o + V_r)/2V_r$$

from the robot, then there exist configurations for which it is not possible for the robot to avoid collision.

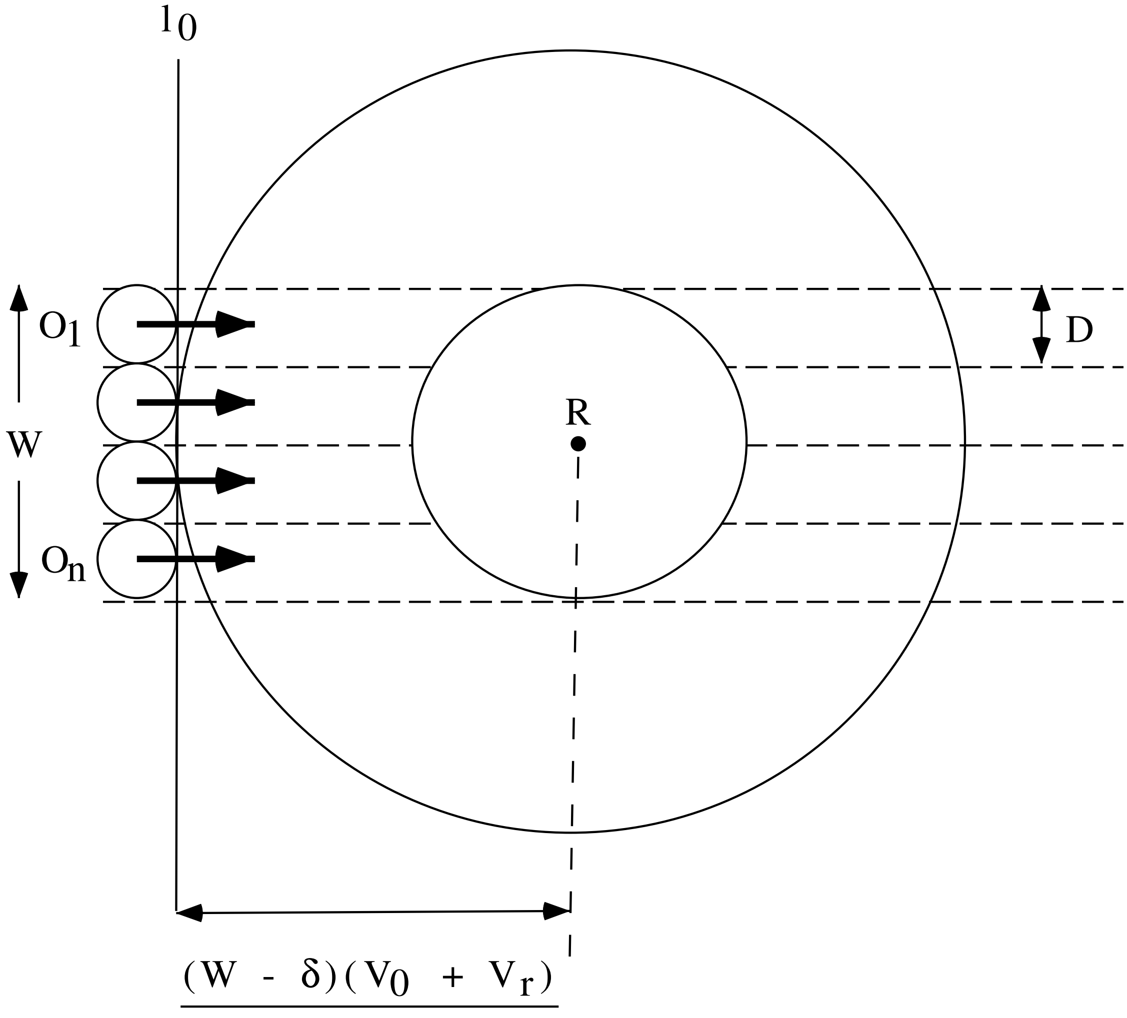


Figure 3.2: Necessity of the Threat Horizon

Proof: It suffices to show a single configuration for which it is not possible to avoid the obstacles. Let all the obstacles O_i have the same diameter D. Assume that the O_i start in the configuration depicted in Figure 3.2, where the obstacles are just touching (i.e. for $i < n$ the distance from the center of O_i to the center of O_{i+1} is D), and they are all traveling at speed V_o, along parallel courses as indicated in the figure. Let l_t be the line which is tangent to all of the obstacles at time t, and which is on the same side of the obstacles as it is at time t_0, when it is on the side of the obstacles nearest the robot. Let the distance from R to the line l_0 be $(W - \delta)(V_o + V_r)/2V_r$. Clearly, all of the obstacles are initially at least this distance from R at time t_0. The robot can travel the distance $(W - \delta)/2$ in time $(W - \delta)/2V_r$. When it has done so, the line $l_{(W-\delta)/2V_r}$ will be a distance $(W - \delta)/2$ away from the (original) point R. If the obstacles can travel the distance W before the robot can move $\delta/2$, then they will completely traverse the circle of radius $W/2$ centered at R before the robot is able to move outside of this circle. That is, if $W/V_o < \delta/2V_r$, the robot will be hit by at least one obstacle. Since, by definition $\delta > 2WV_r/V_o$, this completes the proof.

In cases where V_o is large relative to W and V_r, δ approaches arbitrarily close to 0. Consequently, $H = W(V_o + V_r)/2V_r$ is the minimum distance for which we can guarantee that a safety-preserving path exists, in the general case.

3.1.2 The Dynamic Asteroid Avoidance Problem

The previous section presented a necessary and sufficient criterion for guaranteeing safety in the classical "static" asteroid avoidance problem. In doing so, a functional relationship was established between the number, size, and speed of the obstacles, the maximum speed of the robot, and the distance which obstacles must initially be from a robot in order to ensure that the robot will never collide with any of the obstacles. The problem addressed thus far makes the same simplifying assumption as is made in, for example, (29; 30; 43; 69) : the position and the trajectories of the obstacles are known **prior** to execution time. While this formulation has proven challenging, it is overly optimistic. Normally, we can expect the existence, location, and trajectories of obstacles to become known only at execution time, perhaps while the robot is already in the process of avoiding previously detected obstacles. This section examines the problem of guaranteeing safety when the location of obstacles must be sensed at execution time, which is referred to as the "dynamic" Asteroid problem. Unlike the previous problem, under these conditions, it is not possible to compute a single safe haven and traverse a linear, fixed velocity path to it. Rather, a continuously evolving path must be determined as new information becomes available about the environment, and this must be done in such a way that safety is never compromised. Using Theorem 1, the next section develops a sufficient condition for ensuring the existence of a safety-preserving, piecewise-linear path in this problem.

3.1.2.1 Obstacles with Uniform Velocity

Consider the asteroids problem described above, where we know there are at most n circular obstacles $O_1, \ldots, O_n$, traveling along linear trajectories at constant speeds. For simplicity, assume that all of the obstacles are of the same diameter, D. In addition, assume that all of the O_i move at the same speed, V_o. Unlike the previous section, we do not assume that we know the location of the obstacles in advance. Instead, the obstacles are allowed to appear, one or more at a time, up to a maximum of n obstacles. We wish to determine a safety horizon, $\bar{H}$, such that we know that a safety-preserving path exists as long as all of the obstacles initially appear at a distance of at least $\bar{H}$ from the robot. The following is a corollary of Theorem 1 above:

Corollary 1: Let H be $W(V_o + V_r)/2V_r$, where $W = nD$, V_r is the maximum speed of the robot, and V_o is the (uniform) speed of the obstacles. If each of the obstacles O_i appears at some time $t_{a(i)} \geq t_0$, at a distance greater than $\bar{H} = nH + W$ from the position of the robot at that time, then there exists a piecewise linear, collision-free path from the starting point R to some point S such that none of the O_i will touch S at any time.

Proof (by induction on m, the number of obstacles that have already appeared[3]) The base case is handled by Theorem 1, since $nH + W \geq H$, and we can assume that $t_{a(1)} = t_0$.

Assume, by induction, that after the first $m < n$ of the obstacles have appeared, there exists a safety preserving path to some point S_m which is safe from all of the obstacles $O_1, \ldots, O_m$. Let R_x be the location of the robot at time t_x. If at time $t_{a(m+1)}$ all of the obstacles $O_1, \ldots, O_m$ are farther than H from $R_{a(m+1)}$, then once again Theorem 1 holds, and a safety preserving path exists from $R_{a(m+1)}$ to some point S_{m+1}. If one (or more) of the O_i is within H of the robot, then the robot can wait at S_m until those obstacles have traveled at least a distance H from S_m. In the worst case, this time is $(H + D)/V_o$. Of course, another obstacle could move to within H of S_m in this time. Since there are only $m < n$ obstacles, the longest we would have to wait would be $m(H + D)/V_o$ before we can be assured that all of the obstacles are at least H from S_m. In this worst-case, the most recent obstacle, O_{m+1} will still be a distance greater than $(nH + W) - m(H + D) = n(H + D) - m(H + D) = (n - m)(H + D)$ from S_m, and since $m < n$, we know that this distance is greater than H. Thus we know that at some time, all of the $m + 1$ obstacles will be outside of the safety region, and so Theorem 1 applies. Thus there exists a (linear) path from S_m to some new point S_{m+1}. **Q.E.D.**

Since by definition $W = nD$, it follows that $\bar{H} = n^2 D((V_o + V_r)/V_r + 1)$, and thus this corollary gives an $O(n^2)$ upper bound for the threat horizon in the dynamic asteroids problem. It is also easy to see that $\bar{H}$ is sufficient to guarantee safety so long as there are never more than n obstacles within $\bar{H}$ of the robot R at any single time, even if many more than n obstacles appear over time.

3.1.2.2 Allowing the Speed of Obstacles to Range from V_l to V_o

There is a straightforward generalization of the above theorem in cases where the obstacles are constrained to have constant positive velocities ranging from a lower bound of V_l to a maximum speed of V_o.

Corollary 2: Let $O_1, \ldots, O_n$ be n circular obstacles of fixed diameter D, each constrained to move at a constant linear velocity, V_i, such that $\forall i \; V_l \leq V_i \leq V_o$, where V_l and V_o are fixed positive constants. Let $W = nD$, and H be $W(V_o + V_r)/2V_r$, where V_r is the maximum speed of the robot. If each of the obstacles O_i appears at some time $t_{a(i)} \geq t_0$, at a distance greater than

$$\bar{H} = \frac{V_o(nH + W)}{V_l}$$

from the position of the robot at that time, then there exists a collision-free path from the starting point R to some point S such that none of the O_i will touch S at any time.

Proof (by induction on m, the number of obstacles that have already appeared): The base case is again handled by Theorem 1, since that theorem applies whenever obstacle velocities are constrained by some maximum, V_o, and obstacles occur outside the threat horizon H. Since $V_o \geq V_l$, it follows that, for all values of n, $V_o n / V_l \geq 1$, and thus $\bar{H} > H$.

The inductive step is very similar to that for Corollary 1: Assume, by induction, that after the first $m < n$ of the obstacles have appeared, there exists a safety preserving path to some point S_m which is safe from all of the obstacles $O_1, \ldots, O_m$. Let R_x be the location of the robot at time t_x. If at time $t_{a(m+1)}$ all of the obstacles $O_1, \ldots, O_m$ are farther than H from $R_{a(m+1)}$, then once again Theorem 1 holds, and a safety preserving path exists from $R_{a(m+1)}$ to some point S_{m+1}. If one (or more) of the O_i is within H of the robot, then the robot can wait at S_m until those obstacles have traveled at least a distance H from S_m.

---

[3] This is not induction on n, the maximum number of obstacles that can appear.

In the following, the earlier proof must be changed to account for the fact that slower moving obstacles may remain near the robot for longer periods of time: In the worst case, this time is $(H+D)/V_l$ (note that $(H+D)/V_l \leq (H+D)/V_o$, which was the worst case in the previous proof). Again, another obstacle could move to within H of S_m in this time. Since there are only $m < n$ obstacles, the longest we would have to wait would be $m(H+D)/V_l$ before we can be assured that all of the obstacles are at least H from S_m. In this worst-case, the most recent obstacle, O_{m+1} can travel a distance of up to $V_o * m(H+D)/V_l$ (if it happens to have the maximum speed, V_o). In any case, then, this will still be a distance greater than $V_o * (nH+W)/V_l - V_o * m(H+D)/V_l = (n-m)V_o(H+D)/V_l$ from S_m, and since $n > m$, and $V_o \geq V_l$, we know that this distance is greater than H, and Theorem 1 applies. Thus there exists a (linear) path from S_m to some new point S_{m+1}, which is safe with respect to all of the currently visible obstacles. **Q.E.D.**

3.1.3 Tighter Bounds

Note that this threat horizon grows with the ratio of the maximum speed to the minimum speed of obstacles, V_o/V_l. If this number is large, the size of the horizon becomes prohibitive. Intuitively, slower moving obstacles should be easier to avoid, but in the context of this result, allowing the speed of obstacles to approach zero will make the threat horizon approach infinity. This is a clear indication that the bound is not tight. Even in the case where $V_o = V_l$, H is chosen to account for the case when all n obstacles are a distance H from the obstacle. That is, the worst-case scenario for the static problem occurs when all of the obstacles will occupy the safety region at roughly the same times. In such cases, the robot must find a location that is a safe haven with respect to all obstacles. When obstacles are more spread out, it may be possible for the robot to find safety-preserving paths where no locations it moves through are safe havens with respect to all obstacles. Ultimately, a location that the robot passes through need only be safe at the time the robot occupies it. A safe haven is defined to be a location that is safe with respect to a given set of obstacles at **all** times.

$\bar{H} = nH + W$ is chosen to account for the times when the n obstacles are "evenly" spaced. That is to say, it was constructed to ensure that, no matter what the configuration of obstacles when a new obstacles approaches within $\bar{H}$ of the robot, there will eventually be a time that **no** obstacles will be within H of the robot. Then, Theorem 1 can be applied, and a linear, constant velocity path to a safe haven is guaranteed to exist. When all the O_i are going to occupy the safety region at roughly the same time, a threat horizon of H alone is sufficient. When the obstacles are evenly spaced, so that exactly one is within H of the robot at any given time, then a horizon of $H' = \bar{H}/n = H + D$ will suffice, since the robot will be able to dodge each obstacle one at a time, by simply moving (at most) half the width of that obstacle. This observation may allow us to reduce the threat horizon by a constant factor, but the resulting distance will still be quadratic in the number of obstacles. I believe that a threat horizon exists that is asymptotically better than $O(n^2)$ in the number of obstacles, but I have been unable to prove or disprove this conjecture. Section 7.4 (page 78) presents empirical evidence that supports this conjecture.

3.2 Stationary Obstacles

When the speed of obstacles is allowed to be zero, then the threat horizon $\bar{H}$ of Corollary 2 goes to infinity. A different approach must be used in order to account for the presence of stationary obstacles. This section, examines ways in which the concept of the threat horizon can be extended to account for the additional complexities caused by stationary obstacles. First, it shows how the safety radius can be expanded to account for the presence of stationary objects, while still guaranteeing the existence of a safe haven. Then this section examines how to adjust the threat horizon to reflect the worst-case time required to travel a possibly-obstructed path to safety. In both cases, it is easy to find upper bounds which are sufficient to guarantee safety. Finally, it is shown that while these bounds are not tight, domain analysis can often allow more reasonable worst-case bounds to be established.

3.2.1 Increasing the Size of the Safety Region

In the proof of Theorem 1, it was sufficient to show that an arbitrarily small safe haven existed. In order to guarantee the existence of a safe haven in the presence of both moving and stationary obstacles, we must adapt the approach to ensure that our potential safe haven is not going to be occupied by stationary obstacles. Assuming initially that all that is known about the stationary obstacles is their total area, α, how large must the safety radius be so that the existence of a safe haven can still be guaranteed? If all one knows is the area occupied by the stationary obstacles, then the safety region must be expanded so that its area is larger than the worst-case area occupied by the moving obstacles, plus α, the area occupied by stationary obstacles. This is not as simple as calculating the area, A, occupied by the safety region needed in the absence of stationary objects, and increasing the safety radius R, of this region to $\bar{R}$, such that $\pi\bar{R}^2 = A + \alpha$. This is because, as the safety radius increases, the worst-case area occupied by moving obstacles also increases. We want to find the radius of a circle so that, if all of the obstacle half-planks cover the circle, there will always be an area of size greater than α that is not covered.

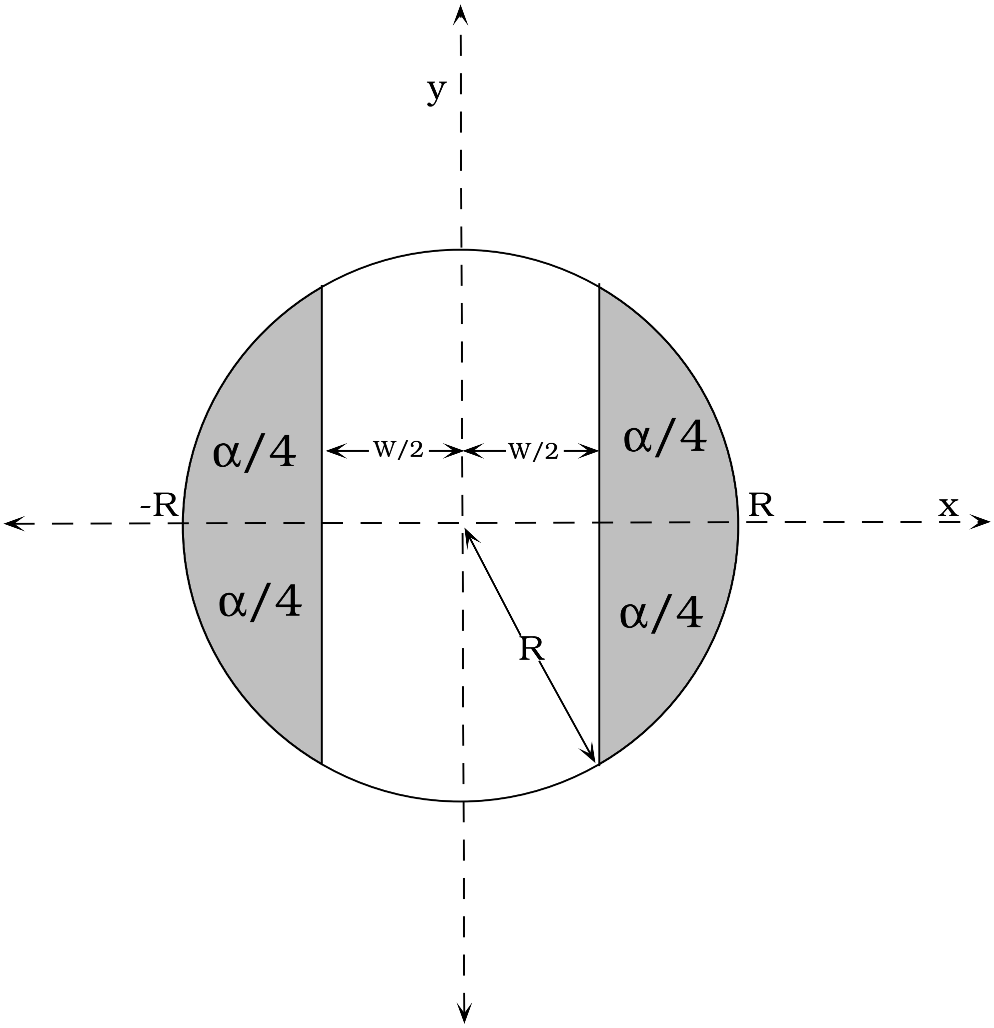


Figure 3.3: Determining the Safety Radius

Consider Figure 3.3. Recall that W is the sum of the widths of the obstacles. From Bang's Plank Theorem, it follows that the largest area of a circle that can overlap the obstacle half-planks is the area between two parallel lines each $W/2$ from the center of the circle, since this is the largest convex object of minimum width W that lies completely within the safety region. Let us call this area ω. In Figure 3.3, it is the part of the circle that is not shaded. We want to find a safety radius R such that $\pi R^2 > \alpha + \omega$. This is complicated by the fact that ω is itself a function of R. If its center is at the origin, the equation for a circle of radius R is just $R^2 = x^2 + y^2$. Equivalently, $y = \sqrt{R^2 - x^2}$. From Figure 3.3, it should be clear that

$$\alpha = 4 \int_{W/2}^{R} \sqrt{R^2 - x^2}\, dx$$

Integrating this yields

$$\alpha = \pi\, R^2 - \frac{W\,\sqrt{4\,R^2 - W^2}}{2} + 2\,R^2\,\arctan(\frac{W\,\sqrt{4\,R^2 - W^2}}{-4\,R^2 + W^2}) \tag{2.1}$$

Unfortunately, there is no obvious way to obtain a closed-form solution for R in terms of α and W. However, this can be solved iteratively, especially since we do not need an exact value for R : **any** value larger that the exact solution will guarantee the area is at least as large as required. A similar form of this equation will also be used later, in Section 4.2, which shows how Monte Carlo methods can be used to find the safe haven.

3.2.2 Finding a Path

A number of authors have examined the problem of determining a path through some set of stationary obstacles, e.g., (13; 66; 43; 42), and solutions have been found with time complexities as low as $O(n \log n)$, where n is the number of vertices of the obstacles. This section adopts the traditional assumptions that robots are holonomic and travel along piecewise-linear paths. One common method for solving the find-path problem is the visibility graph approach used by 58 (58), which will be briefly sketched.

Assume that all obstacles are modeled as convex polygons, and call the set of polygonal edges S. A visibility graph of S is a graph that has a node for each endpoint of every segment, and an edge between nodes v_0 and v_1 if

1. v_0 and v_1 are adjacent vertices on the same obstacle, or
2. v_0 and v_1 are on different obstacles, and the open segment (v_0, v_1) does not touch any segment in S.

In other words, an edge between v_0 and v_1 exists in the graph if the straight-line path between the two vertices does not pass through an obstacle. Figure 3.4 shows a simple example of a visibility graph. The edges corresponding to the edges of the polygons A,B, and C are in the graph because of the first condition above, and the edges

$$A_2B_1, A_4B_3, A_3B_1, A_3B_3, B_2C_1, B_2C_2, B_2C_3, B_4C_1, \text{ and } \; B_4C3 \tag{2.2}$$

are in the graph due to condition 2.

Several algorithms are known for finding the visibility graph on a set of line segments in time $O(n^2)$, where n is the number of vertices in the set S. 34 (34) describes an algorithm that takes time $O(k + n \log n)$, where k is the number of edges in the graph.

To use visibility graphs to solve the find-path problem, add the robot's location, R and a destination G into the set S of segments that delimit obstacle boundaries. Then compute the visibility graph for these vertices. A version of Dijkstra's algorithm (23) for single-source shortest

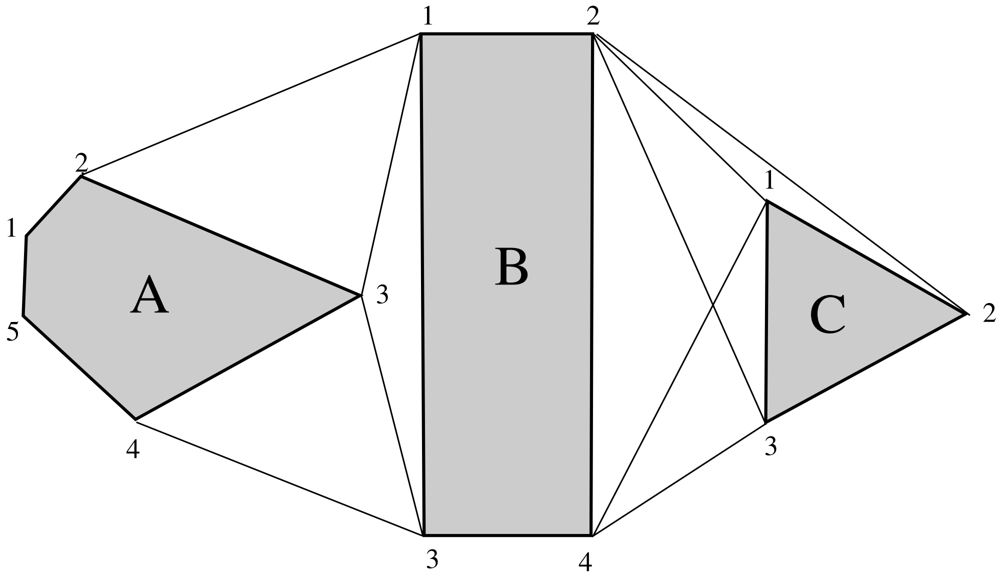


Figure 3.4: A Visibility Graph

paths can then be used to find the shortest path from R to G that does not pass through the interior of an obstacle. This computation can be performed in $O(n^2)$, so the overall path-planning problem can be solved in $O(n^2)$.

It is therefore a straightforward matter to bound the computation time required to find a path to a safe haven with some function quadratic in the number of obstacle vertices. But to guarantee that the robot can avoid all moving obstacles while negotiating a path through some set of stationary obstacles, we also need to be able to bound the path **execution** time, in order to ensure that the robot can sense incoming moving obstacles soon enough to move out of their way. This amounts to finding the longest minimum-length path between any two points within the safety radius. In general, even when the area occupied by the obstacles is known and fixed, this distance can be made arbitrarily long. For an intuition on why this is, consider Figure 3.5, which shows how such a long path could be constructed. If we assume that moving obstacles cannot penetrate into the interior of static obstacles, a robot would not have to travel arbitrarily far into the interior of a concave obstacle in order to avoid an obstacle traveling along a linear trajectory of fixed velocity. So it may be possible to bound the distance a robot would have to travel by limiting the set of origination and destination points to those that are either reachable by obstacles, or not reachable by a moving obstacle, but arbitrarily close to locations that are reachable.

Another way to simplify this problem is to assume that all stationary objects are convex. This is not as strong an assumption as it might seem: first, it merely requires us to eliminate from consideration as a safe haven any point inside a concavity of an obstacle. This, in turn implies that the effective area of the obstacles will increase.

Define the width of an obstacle to be the maximum distance between two points on its exterior. If all obstacles are concave, and we allow them to touch, then the most they can force the robot to deviate from a straight-line path between two points is at least half the sum of their maximum widths, as shown in Figure 3.6(a). Exactly how much more than this R may be diverted is unclear, but appears to be bounded by the width of the largest obstacle. In any case, the total diversion is certainly less than the sum of the widths of the obstacles.

Of course, allowing obstacles to touch is the same as combining them into a single obstacle. If we assume for this reason that all distinct obstacles are disjoint, determining the worst-case

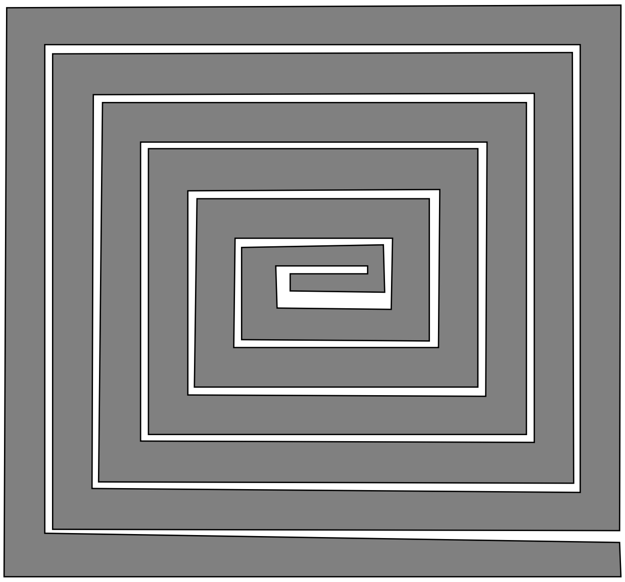
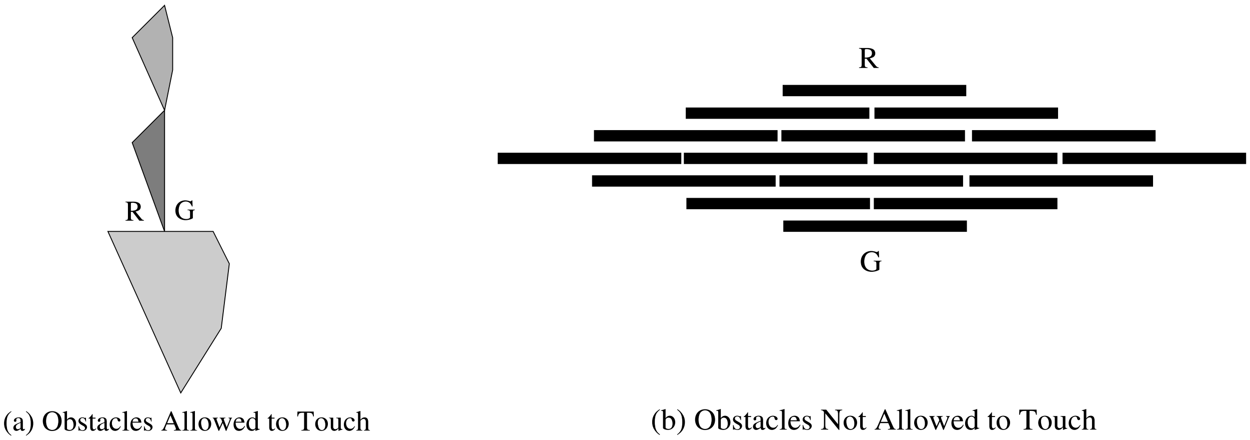


Figure 3.5: An Arbitrarily Long Path in the Interior of an Obstacle



(a) Obstacles Allowed to Touch

(b) Obstacles Not Allowed to Touch

Figure 3.6: Obstacles Forcing Long Diversions

obstruction is greatly complicated. First note that "thin" obstacles actually force more diversion than "rounder" obstacles. Figure 3.6(b) shows that the general solution is at least $W * \lfloor\sqrt{N}\rfloor$, where W is the width of the greatest obstacle, and N is the number of obstacles.

This assumes that all obstacles are of equal width, and that they are very much wider in one direction that in the other (perpendicular) direction. They can, in fact be made arbitrarily narrow, and arbitrarily close together. In this example, the robot is forced to travel a distance of roughly $4W$ in addition to the distance from R to G. Whether one can force a diversion of worse than $W\lfloor\sqrt{N}\rfloor$ is an open question, which involves reasoning about all the possible deployments of all the possible shapes and sizes of obstacles.

In practice, though, it is hard to imagine having information about the number, width and area of stationary obstacles without knowing something about their deployment. Assuming that we do have an upper bound on the time it will take the robot to travel the (worst-case) path to safety, it is straightforward to extend the threat horizon by a corresponding amount. If the worst-case extra distance that the robot can travel due to obstruction by moving obstacles is D, then the time the robot will take to travel this distance is D/V_r, and the threat horizon needs to be increased by the distance the fastest obstacle can travel in this time, i.e., $D * V_o/V_r$.

3.2.2.1 Turning and Accelerating

It is also interesting to ask what effect the obstacles can have on the number of turns that the robot is forced to make. This is relevant, because physical robots will need to account for the time required to make such turns, in addition to the time need to travel the path. Assuming that the obstacles are convex polygons, then clearly we need make no more than $\lceil (V+1)/2 \rceil * N$ turns, where V is the number of vertices in the obstacle with the most edges[4]. However, the actual bound appears to be much lower. If we adopt the conjecture suggested above that $\lfloor \sqrt{N} \rfloor$ bounds the number of non-touching obstacles that the robot can be forced to circumvent in the shortest path from R to G, then it follows that the robot need make no more than $\lceil V/2 \rceil * \lfloor \sqrt{N} \rfloor$ turns.

If the time required to turn is high, it may be worth "expanding" the obstacles by enclosing them in rectangles, and not allowing the robot's path to enter any rectangle (except perhaps to terminate). By doing this we can ensure that the robot need never take more than 3 turns per obstacle: one when it reaches the corner of the enclosing rectangle and turns towards one of the two adjacent vertices, another when it reaches that vertex and turns to traverse the next edge, and the third when it leaves the edge. Again, whether this is a worthwhile approach is highly domain-dependent.

Once the worst-case number of turns is established, adjusting the threat horizon is again simply a matter of extending it to account for the time it will take the robot to execute these turns.

3.3 Changing Velocities and Uncertainty

Up until this point, I've assumed that all obstacles travel at known, constant velocities. Allowing obstacles to change speeds does not alter the fundamental results of Theorem 1 (page 16) or Corollary 2 (page 20). However, accounting for obstacles which can deviate from a straight-line path transforms the problem considerably. If the trajectory of an obstacle is known but non-linear, the problem becomes one of finding the equivalent of Bang's Plank Theorem for whatever curves the various obstacles trace in the plane, i.e, we can use the same technique as used in Theorem 1 if we can accurately characterize the largest circle which the various obstacle paths can completely cover.

Allowing for the case where the actual path of the obstacle is not known to the robot is much more difficult. To begin with, if the obstacles can travel faster than the robot, and are able to change their direction without any restrictions, it is not possible to guarantee avoidance of even a single obstacle. So, we need to assume some bounds on the ability of the obstacles to change course.

One way to do this is to bound the change in direction as a function of time. If we limit an obstacle's ability to deviate from an initial direction θ_0 with a maximum angular velocity of ω, what can we say about the robot's ability to avoid obstacles? To begin with, at time $t = \pi/\omega$, an obstacle can make a turn of 180 degrees, or in other words, after this time, it could be heading in **any** direction, and for this reason, no point in the plane can be considered a permanent safe haven with respect to even a single obstacle. This does not mean that such obstacles cannot be avoided, it only means that, given enough time, any obstacle can get to any fixed point in the plane. Is it possible to adjust the threat horizon to allow for even slight changes in course?

Consider this approach: The set of points possibly occupied by the obstacle after time t is shown in Figure 3.7. The inner, dotted lines represent the boundaries of the obstacle's position at times between t_0 and a fixed point in time t. For simplicity, assume that θ_0 is 0, and that R is at the origin. Then the two dotted lines of Figure 3.7 are actually arcs of circles, which represent the path traveled by the robot if it moves at its maximum velocity while turning constantly at a rate of ω (or

---

[4] Not so clearly, this can be made $\lceil V/2 \rceil * N$ as long as we are careful to travel from vertex-to-vertex when moving from one obstacle to the next.

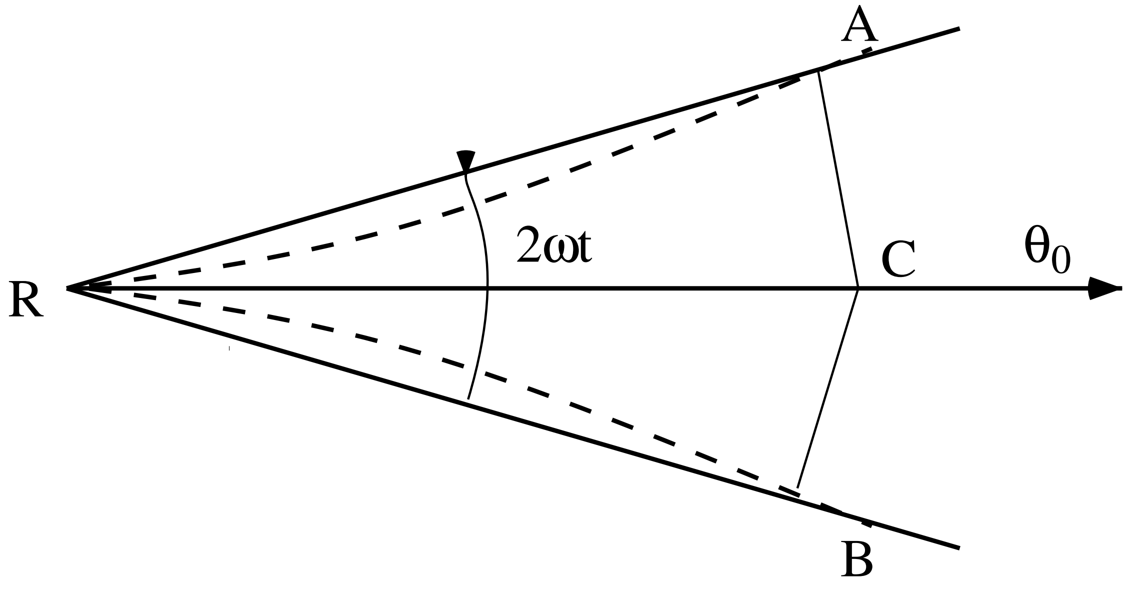


Figure 3.7: Approximating the Area Occupied by an Obstacle Whose True Trajectory is Not Known

$-\omega$). As noted above, the robot will turn 180 degrees in time π/ω, so it will complete the circuit in twice this time, i.e., $2\pi/\omega$. If the robot moves at maximum speed V_r, then the circumference of this circle is $2\pi V_r/\omega$, and the radius is therefore ω/V_r. Thus the equations for the two circles are

$$x^2 + (\omega/V_r - y)^2 = (\omega/V_r)^2 \tag{3.1}$$

$$x^2 + (-\omega/V_r - y)^2 = (\omega/V_r)^2 \tag{3.2}$$

We can use the outer lines as conservative estimates for the space occupied by an obstacle through time t. Using this (over)estimate, note that an entire half-plane can be covered by N obstacles, if the threat horizon $\bar{H}$ is greater than $V_o * \pi/N\omega$, as shown in Figure 3.8. This provides an upper bound on $\bar{H}$.

The outer lines are defined as follows

$$\begin{aligned} x &= x_0 + t * V_o * \cos\theta \pm \omega \\ y &= y_0 + t * V_o * \sin\theta \pm \omega \end{aligned}$$

After time t, the angle $\angle ARC = \angle BRC = \omega t$. The distance $RA = RB = RC$, and is just $V_o * t$. Thus $\triangle ARC$ and $\triangle BRC$ are both isosceles, and by the Law of Sines, their base is equal to $2tV_o \sin \omega t$. If we model the region occupied by each obstacle as a plank of width $4tV_o \sin t\omega$, the we can use the previous results of Theorem 1 (page 16), and Corollary 1 (page 19) to determine a safe harbor with respect to any obstacle no farther than tV_O away from the robot. In the static case, where we are given the initial position of a set of obstacles and asked to find a safe haven with respect to them, this approach cannot be used. Somewhat paradoxically, as we move the position of obstacles farther and farther back, even though they become easier for the robot to avoid, the set of points which they could possibly hit becomes larger and larger, and as noted above, a single obstacle a distance of $V_O\pi/\omega$ away will cover an entire half-plane. We were able to adopt this strategy in the dynamic case, since we needn't worry about obstacles arbitrarily far away. As long as the robot can sense the position of an obstacle as it enters the expanded threat horizon $\bar{H}$, it has a strategy for avoiding it.

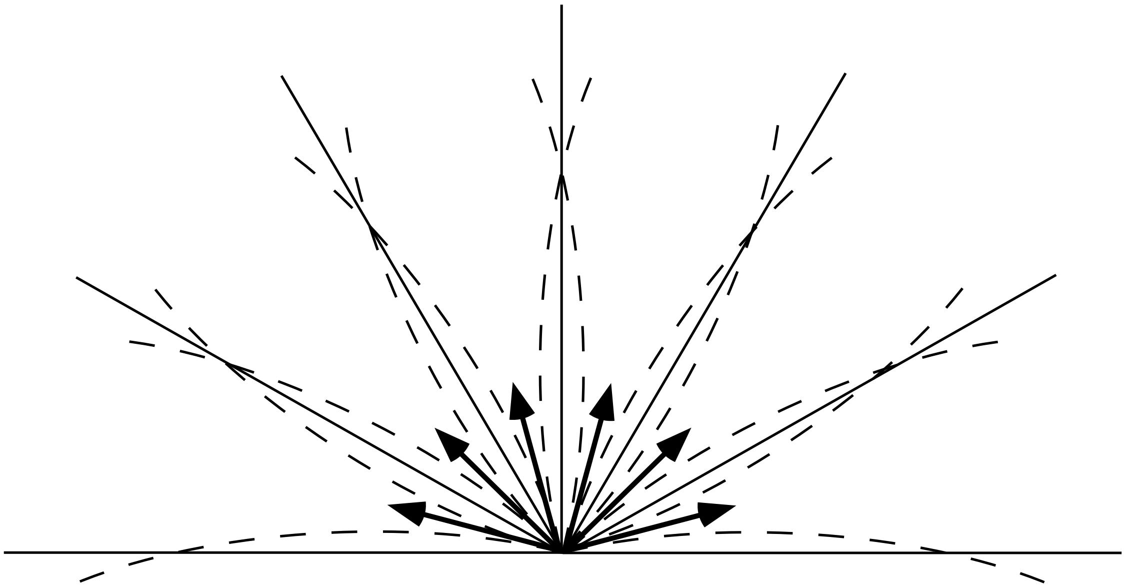


Figure 3.8: Plane Covering By 6 Obstacles Due To Uncertainty

Recall that the threat horizon, $\bar{H}$ was defined in Corollary 1 to be

$$n * W(V_o + V_r)/2V_r + W$$

In that formulation, W was the sum of the widths of the obstacles. We need to change this to reflect the "growth" caused by uncertainty, i.e,

$$W' = W + n4tV_o \sin t\omega$$

The resulting threat horizon is

$$\bar{H} = (W + n4tV_o \sin t\omega)(V_o + V_r)/2V_r + W + n4tV_o \sin t\omega$$

Unfortunately, t is a function of $\bar{H}$, i.e., $t = \bar{H}/V_o$. In other words, we need to solve for t, such that

$$t = ((W + n4tV_o \sin \omega t)(V_o + V_r)/2V_r + W + n4tV_o \sin \omega t)$$

and there is no closed-form solution to this equation. Again, we can use iterative methods to find t, if it exists. In practice, this technique will only work for small values of ω. Remember, however, that in cases where α is unbounded, we cannot guarantee to avoid even a single obstacle. These results show that it is possible to avoid a belligerent obstacle that can move faster than the robot, if the obstacle's ability to turn is limited. One should keep in mind that the **actual** width of the obstacles does not change, and this choice of threat horizon is very conservative. Consequently, if we are willing to sacrifice guarantees on the robot's ability to find a safe haven, in favor of accepting some arbitrarily small but non-zero risk, we can expect to reduce the threat horizon considerably.

3.4 Goals of Achievement

The Asteroids problem addressed in (69),17, and (18) is one of finding a path from a starting position to a single goal position while avoiding all moving obstacles. So far, only with the problem of avoidance has been considered. Recall, that these papers both addressed the "static" problem, where the positions of all the moving obstacles were known, and the robot was charged with finding a single safety-preserving path. In this formulation of the problem, it is sufficient to find a path to a single safe haven, since the robot can wait there indefinitely, until the obstacles move out of its way and a safe path presents itself. It follows from this and Canny's $\mathcal{NP}$-completeness result (18), which shows that the Asteroids Problem is $\mathcal{NP}$-complete, that the problem of finding a safe haven and a path to it must also be $\mathcal{NP}$-complete.

In the dynamic asteroids problem, where any number of obstacles are allowed to appear over time, subject to some limit on the number within a fixed distance of the robot, the situation is less clear. One can imagine scenarios where the obstacles somehow conspire to keep the robot away from the goal. Since the target is fixed, we may not be able to ensure that the robot can ever get to it. However, it turns out that it is possible to find a threat horizon that will guarantee the robot's ability to make progress to the goal, and this in turn allows us to ensure that the robot **is** eventually able to move safely to any target location, as long as all obstacles are constrained to appear outside the threat horizon.

Corollary 3: Assume a fixed goal position G. Let $H = W(V_o + V_r)/2V_r$ be the threat horizon determined in the **static** case of Theorem 1 above. Let $\bar{H} = nH + W$ be the threat horizon of the dynamic case in Corollary 1. Finally, let $\bar{H}_G = 2\bar{H}$. If all obstacles are constrained to appear a distance $\bar{H}_G$ from the robot, then whenever a new obstacles appears, there exists a collision-free path from the robot's initial position at R to some point R' such that the $distance(R', G) < distance(R, G)$.

Proof Recall that the set of points that the robot can get to in a fixed time t is a circle of radius $t * V_r$. Furthermore, recall that the static threat horizon H ensures that the robot is able to move a distance $(W + \epsilon)/2$ before an obstacle appearing outside of the threat horizon could possibly hit it. Clearly, doubling H will double this distance, so for a threat horizon of $2H$, the robot will be able to move to any point within a circle of radius $W + \epsilon'$ before possibly being hit by an obstacle which starts at a distance outside of this horizon.

Now, for fixed and distinct points R and G, consider the set of points that are closer to G than R. This is the set of points bounded by, but not including, the circle centered at G whose radius is the distance from R to G. It follows that the set of points which are closer to G than R, and to which we can guarantee that the robot can safely travel, is the intersection of this circle with the safety region.

If G is within the safety radius, then the robot can clearly make progress to G: in fact, it can safely travel to G without being hit (by the definition of the safety radius and Theorem 1). So we can assume, without loss of generality, that G is outside of the safety region. Let C_S be the circle of radius $W + \epsilon$ centered at R, and let C_G be the set of points that are closer to the goal G than the robot at point R. The set of points $C_G \cap C_S$ cannot be covered by the half-planks representing the location of the obstacles over time. To see this, first note that the area occupied by $C_G \cap C_S$ increases with the distance from R to G. Therefore, this area is smallest when G is as close to R as possible, i.e., just outside the safety radius of $W + \epsilon$. Now consider Figure 3.9. The parallelogram $\square RI_0GI_1$ can be inscribed within the region. $C_G \cap C_S$, and all of the edges, RI_0, I_0G, GI_1, and I_1, R. are of length $W + \epsilon$, as is the segment GR. From the Pythagorean Theorem, it follows that the height of the equilateral triangles $\triangle\, GI_0R$ and $\triangle\, GI_1R$ is $\sqrt{3}/2(W + \epsilon)$, and hence segment I_0I_1 is of length $\sqrt{3}(W + \epsilon)$. Thus $\square RI_0GI_1$ has width $W + \epsilon$, and so does the region $C_G \cap C_S$. From Bang's Plank Theorem, it follows that the half-planks representing the paths of the obstacles over time cannot cover this region. Consequently, some part of $C_G \cap C_S$ is a safe haven w.r.t. any of the obstacles which appear a distance of at least $2H$ from R.

Now consider the proof of Corollary 1. If we expand the threat horizon H to $2H$, then at any step in the induction where the robot chooses a safe haven, we can guarantee that the robot can choose a safe haven that is in the region $C_G \cap C_S$, since we have shown above that even when this region is smallest, the obstacles cannot cover all of it. The threat horizon $\bar{H}_G = 2\bar{H}$ was chosen

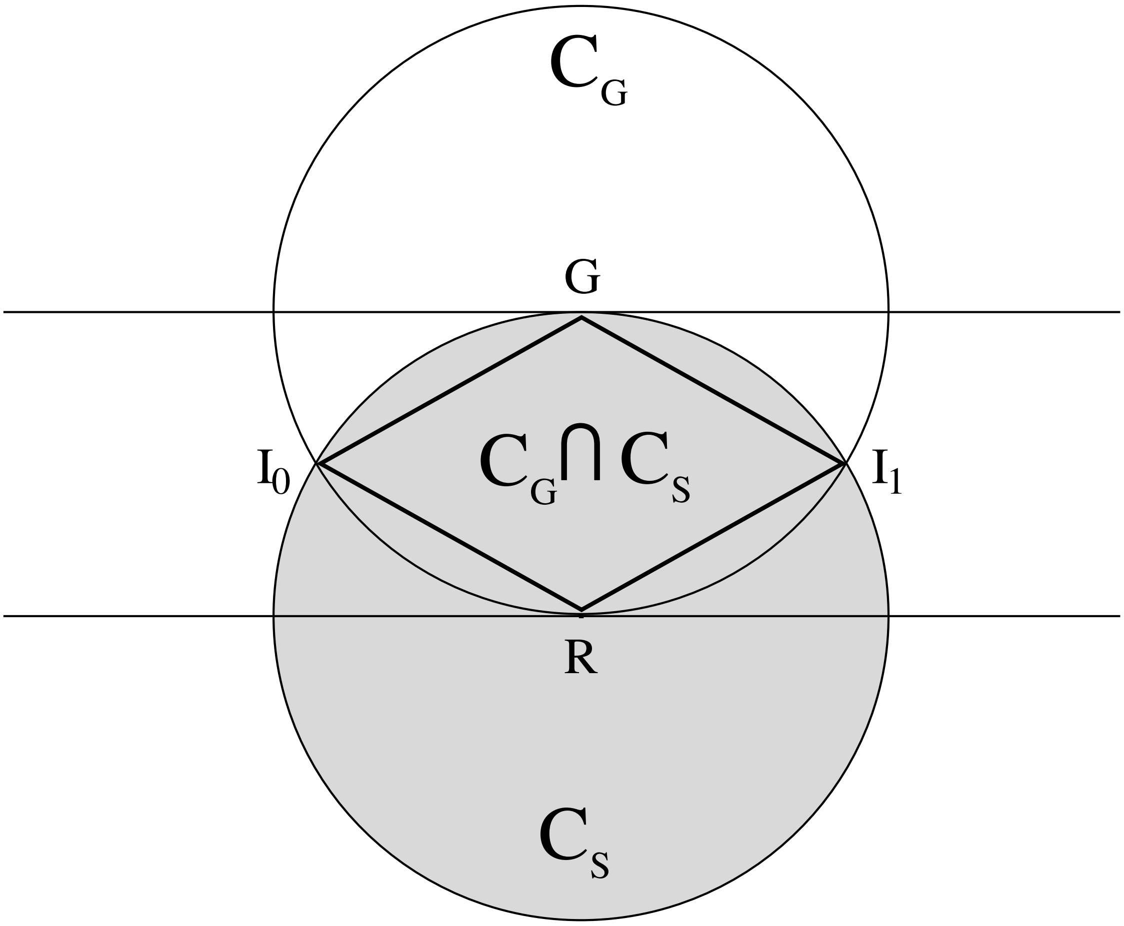


Figure 3.9: The Smallest Intersection of the Safety Region with the Set of Points Closer to G than R

to reflect the necessary increase from H to $2H$ in the proof. It follows that every time the robot chooses and moves to a safe haven, it reduces its distance to the goal.

One of the heuristics that Dodger uses to decide where to go if it cannot find a constant velocity path to its goal is based upon this result. Chapter 7 compares the results of using this approach to those achieved by employing other heuristics. The primary drawback to this approach is that the threat horizon must be extended in order to ensure that a safe haven which is closer to the goal than the robot's current position will always exist. Chapter 7, also examines the effects of shortening the threat horizon below this theoretical minimum. Shortening the horizon implies that safety can no longer be guaranteed, but it would be a straightforward matter to modify Dodger to look first for safety in the general direction of the goal, and only consider moving away from the goal if this is not feasible.

3.5 Summary

This chapter has established a theoretical foundation for real-time obstacle-avoidance. The concept of the threat horizon is central to assuring that a robot will be able to avoid all of the moving obstacles that it encounters, and it also provides the basis for a control regimen that can take the run-time requirements of an obstacle-avoidance algorithm into account. In essence, one can account for the time required to compute a reaction to a newly sensed obstacle simply by extending the threat horizon by the maximum distance that an obstacle could travel while the result is being calculated. Dodger has demonstrated the feasibility of this approach.

After establishing the existence of a threat horizon for the static Asteroids Problem, basic result was extended into cases where 1) the positions and trajectories of the obstacles become known to the robot over time, 2) where there are stationary obstacles as well as moving obstacles, and 3) where obstacles are not restricted to having fixed velocities. This chapter has also shown how it is possible to construct a threat horizon such that the robot is guaranteed to be able to make progress towards the goal. The next chapter will describe some of the algorithms required to put this theory into practice.

Chapter 4

Dodger Algorithms

The Dodger system demonstrates that it is possible for a computer program to be intelligent **in** hard real time: it solves the dynamic asteroids problem in real time, guaranteeing that moving obstacles will be avoided. To do this, the problem has been decomposed into two separate problems: a "critical" component, which ensures the safety of the system, and a non-critical component, which focuses on finding paths from the robot's current position to the location of a goal. Corollary 1 (page 19) establishes the minimum conditions necessary to ensure that the robot will always be safe, and it also provides a blueprint for how this should be done. Dodger employs an algorithm based upon this result to guarantee that the sensing and calculating needed to avoid obstacles will always be performed in time for the simulated Dodger robot to actually avoid them. This algorithm runs quickly at regular intervals, and even in the worst case, it completes in a fraction of a second.

This reactive, hard real time capability is all Dodger needs to avoid obstacles. However, Dodger is also expected to find and traverse paths to randomly generated goals. Consequently, Dodger must also solve the find-path problem. Currently, Dodger employs local decision-making routines to guide it towards its goals. Each of these routines produces acceptable behavior in real time, and one of them is complete, in the sense that it guarantees that paths will be eventually be found whenever they exist. While not complete, the other two heuristics have proven adequate in practice, and actually outperform the complete routine in the number of goals the Dodger is able to achieve over time.

In theory, is possible to use an exhaustive search routine that is guaranteed to eventually find a solution in all configurations. In many cases, this search will find a solution quickly. However, in the worst case, the time required to find a solution will be such that the search is obsolete before it has finished. In such cases, the search can be abandoned as soon as it becomes obsolete. If a new obstacle enters the threat horizon, or a known one leaves it, the search will be resumed from scratch. Eventually, a safety-preserving path to the goal will be found[1], and in the process, safety is never compromised. Such a search is described in Chapter 5.

The theoretical basis for safety-maintenance appears in Section 3.1.2.1 above. As long as objects are constrained to appear outside of the threat horizon, we know by Theorem 1 (page 16) that a safe haven exists. Corollary 1 extends this result to domains, such as Dodger, where the number of obstacles appearing over time is unbounded, and subject only to a maximum number that may be within some fixed distance of the robot at any given time. The proof of Corollary 1 also provides the fundamentals of an algorithm for avoidance. As each obstacle crosses the threat horizon, the Dodger system is faced with a choice: if no obstacle happens to be inside the safety region[2], then

---

[1] Recall that Corollary 3 of Section 3.4 establishes sufficient conditions for ensuring that the robot can always find a path that takes it closer to the goal than its current position.

[2] Recall that the safety region is simply a circle whose diameter is just larger than the sum of the maximum widths

Dodger computes a safe haven with respect to all of the obstacles it knows about, and it begins to head there immediately. If obstacles are inside of the safety region, then Dodger may not be able to find an unobstructed, constant-velocity path to safety. But the threat horizon was constructed to ensure that, eventually, such a path will exist. In such cases, Dodger can continue with whatever it is currently doing until the time that it finds a constant velocity path to a new safe haven. The theoretical results ensure that Dodger will be safe in any case.

This chapter describes two different methods for computing the safe haven: a deterministic one that finds the intersections of the safety region with the planks which represent the space occupied by the obstacles over time, and another which uses Monte Carlo methods to find a safe haven. The second method suffers from two defects: there is a very small but non-zero chance that it will fail to find a safe haven, even when one exists, and it requires the threat horizon to be extended somewhat. However, this technique is algorithmically simple and much more computationally efficient than the deterministic algorithm. It is the algorithm actually used in the Dodger system, and neither of its theoretical drawbacks has proven to be a difficulty in practice.

4.1 Finding a Safe Haven Deterministically

Recall that a **plank** is a pair of parallel lines in the plane. The extremes of an obstacle moving at a fixed velocity will trace two parallel rays over time. In Chapter 3 the region between these two rays was called a **half-plank**. The problem of finding a safe haven can then be posed as the problem of determining a part of the safety region that does not intersect any of the half-planks associated with the obstacles inside of the threat horizon. This section describes one way of doing this.

4.1.1 Preprocessing

In the current implementation, Dodger models obstacles in $\mathcal{C}$-space as a center point, a width (or radius), a speed, and an angle of trajectory. For the purposes of the algorithm described here, this representation needs to be converted to the two parallel lines which bound the obstacle's path in the 2-dimensional spatial plane. Secondly, the algorithm requires these lines to be segments, and not rays, so one must find an appropriate place to clip. It is possible to calculate the obstacle's location at a sufficiently distant point in time and use the segment between that point and the obstacle's initial location. However, the rest of this section assumes that we have tested each line for intersection with the safety region, and that all of the line segments which the algorithm processes lie within the safety region, with their endpoints on the circle which bounds it. While this will not improve processing in the worst-case, it both simplifies the algorithm and promises improved average-case performance in all domains where even a small number of obstacles do not intersect the safety region.

Since by definition the safety region is wider than the obstacles, if an obstacle intersects the safety region, then at least one of an obstacle's two bounding lines must intersect the circle which bounds the safety region. As long as one keeps information about which side of a given segment the obstacle is on (and thus which side it is not on), the algorithm does not need to keep track of the corresponding parallel line in the cases where only one of the two bounding rays of a half-plank intersects the safety region.

Next, obstacles must be "merged": the algorithm requires that the segments do not coincide. So, if two obstacles happen to be traveling parallel to each other, and happen to share a common outer boundary, they can be joined into a single obstacle. Similarly, if two or more obstacles happen to share a trajectory (though perhaps separated in time), or if they travel in exactly opposite directions, all but one of these obstacles can be thrown out. One way to do this would be to first

---

of n obstacles, where n is the largest number of obstacles that will be within the threat horizon of the robot at any given time.

"normalize" the segments so that all segments have trajectories between 0 and π as described above, and each segment is further represented by the point where its containing line crosses the y-axis (or, if it parallels the y-axis, the point where it crosses the x-axis.) These normalized segments can then be easily tested for duplicates. In the cases where coincident segments are the result of completely overlapping obstacle paths, we can throw out all but one. In cases where the coincidence is the result of two parallel obstacles that touch at a point, both of the segments can be thrown out, effectively merging the two obstacles into a single larger one. Again, these determinations can be made simply by maintaining information about which side of a segment is the "inside."

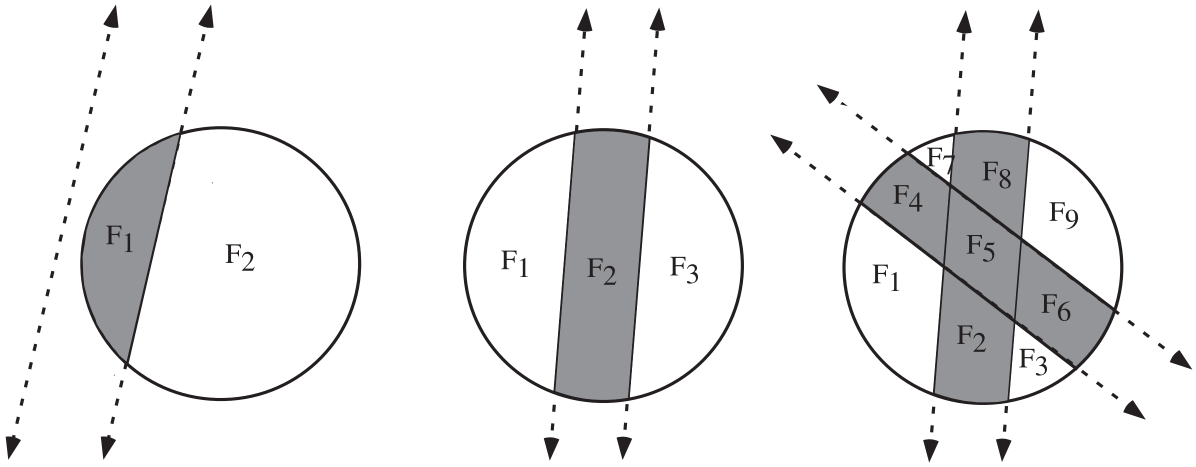
4.1.2 Data Structure

The algorithm uses what is known as a **winged-edge** data structure (8). This description closely follows 67 (67). There are three fundamental data types in this structure: vertices, edges and faces. Edges are the principle type in the following sense: each vertex points to (any) one of its incident edges. Each face also points to an arbitrary one of its edges. In contrast, each edge record e in the basic data structure contains eight pointers: one to each of its endpoints, v_0 and v_1, one to each of the two faces which contain it, f_0 and f_1, two to edges incident to e at v_0, and two to edges incident to e at v_1. Following O'Rourke, these latter edges are named ${e_0}^-, {e_0}^+, {e_1}^-$, and ${e_1}^+$, where ${e_0}^-$ is the edge incident to e at v_0 in the clockwise direction, ${e_0}^+$ is the edge incident to e in the counterclockwise direction, etc. f_0 and f_1 are such that, if one starts at e and follows the clockwise edges starting at ${e_0}^-$, the edges of f_0 will be encountered. Similarly, the edges of f_1 can be produced by following the counterclockwise edges starting with ${e_0}^+$.

This algorithm will require the use of two additional fields in the edge structure: every edge will be a part of either the circle representing the boundary of the safety region, or a ray representing the outside edge of a half-plank. Assume the half-planks will be numbered from 1 to n. The two rays of each half plank i will be assigned the numbers $2i - 1$ and $2i$, and the safety-region circle will be assigned the number 0. A field, R, is needed to hold this number, which will tell us whether an edge is a line segment ($R \neq 0$) or an arc ($R = 0$), as well as which ray the edge is associated with. Note that, after the preprocessing above, it is not possible for a single edge to be part of more than one ray (as in the case where two parallel planks touch, and their edges coincide.) In the event that an edge is part of the circle, a bit is also needed to indicate whether it is a major or minor arc of that circle. Face records are also extended to contain a field that will tell whether or not the face is on the inside or outside of a plank.

4.1.3 Algorithm

If a plank intersects a circle, it will create either 2 or 3 distinct regions, as depicted in Figure 4.1(a) and (b). Each maximally connected region of the plane, that contains no edges or vertices, is known as a **face**. The darkened area in this figure depicts the area of intersection, and the lighter area represents the part of the circle that would be a safety region with respect to this single threat. When a second plank intersects the circle, as shown in Figure 4.1(c), a number of new faces can be created. Note that while the **size** of the unshaded area decreases monotonically with each new plank, the **number** of unshaded faces can either increase or decrease when a new plank intersects some set of existing planks and the circle representing the safety region. The basic idea of the algorithm is to keep track of the areas inside the circle that are not intersected by any plank. We will "mark" all of the faces that are contained in the interior of one or more planks, so that, once the algorithm is completed, the unmarked faces will each constitute a region that is a safe haven with respect to all of the obstacles considered. Processing is done plank-by-plank, and for each plank, edge-by-edge. As each edge is added, existing edges and faces are bisected, and so new edges and faces must be created, and existing ones modified. Once both sides of the plank are processed, and the winged-edge data structure has been modified to reflect the new topology of the graph, the faces between these two plank edges are then marked, to indicate that they are



(a) (b) (c)

Figure 4.1: Plank Intersecting a Circle

intersected by at least one plank.

Let n be the number of clipped plank segments after preprocessing. Call these $P[1..n]$. Let the winged-edge data structure be as described above, and let $E[1..n^2]$ be the edges, $V[1..n^2]$ be the vertices and $F[1..n^2]$ be the faces. Initialize the winged-edge structure to have a single edge, representing the boundary of the safety region. For this edge, $v_0 = v_1 =$ an arbitrary point on the circle. Clearly, this is a major-arc edge. By convention, f_0 is the exterior of the circle and f_1 will be the interior. Proceeding plank-by-plank, for each plank, process each of its two bounding rays

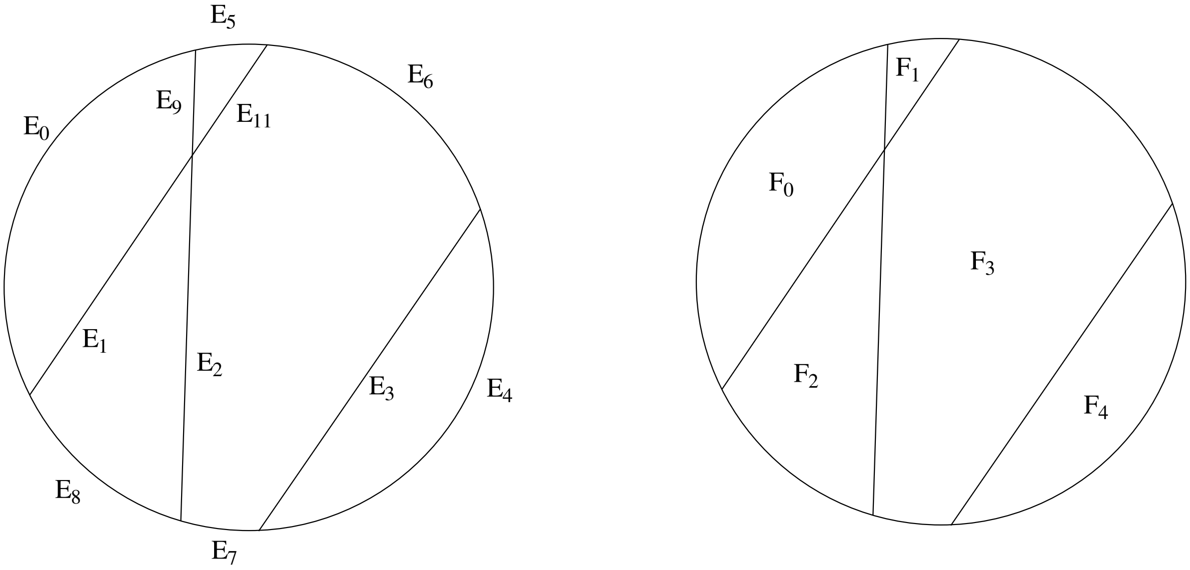


Figure 4.2: Finding a Safe Haven

separately. For a given ray R of the current plank, begin by first finding all of the edges currently in the edge set E that intersect R. Determine the subset of edges E' that intersect R (and the points of intersection), and sort them in the order that their intersections fall on the ray R. Figure 4.2 shows an intermediate state in the computation, the left half of the figure depicts the edges, and the right half depicts the faces. Figure 4.3 shows the effect of processing a single ray R. Five edges intersect R: $E_0, E_1, E_2, E_3,$ and E_4, and the faces $F_0, F_2, F_3,$ and F_4 are bisected by R.

Assume that we have the list of intersection $<$ point,edge $>$ pairs $< V, E >$, sorted in order as they proceed from the origin of the ray R. Note that, for each consecutive pair in the list, $< V_i, E_i >, < V_{i+1}, E_{i+1} >$, a new edge will need to be added, unless this edge corresponds to an existing edge. The endpoint vertices of the new edge are V_i and V_{i+1}. The edges E_i and E_{i+1} will have exactly one face in common. Let this face be F_c. This face is bisected by the new edge, so we will construct a new face, call it F'_c. Both F_c and F'_c should be made to point to the new edge. Some of the existing edges will have to be modified to reflect the fact that they now point to F'_c, as is now explained. Assuming that V_i and V_{i+1} do not coincide with an existing vertex, each of the edges E_i and E_{i+1} is split into two edges at their respective intersection points (i.e. either V_i or V_{i+1}). The edges E_i is split into E'_i and E''_i, which will become the wings e_0^+ and e_0^- of the new edge between V_i and V_{i+1}. Similarly, E_{i+1} will be split into E'_{i+1} and E''_{i+1}, which will become the wings e_1^+ and e_1^-. To summarize, the new edge will be

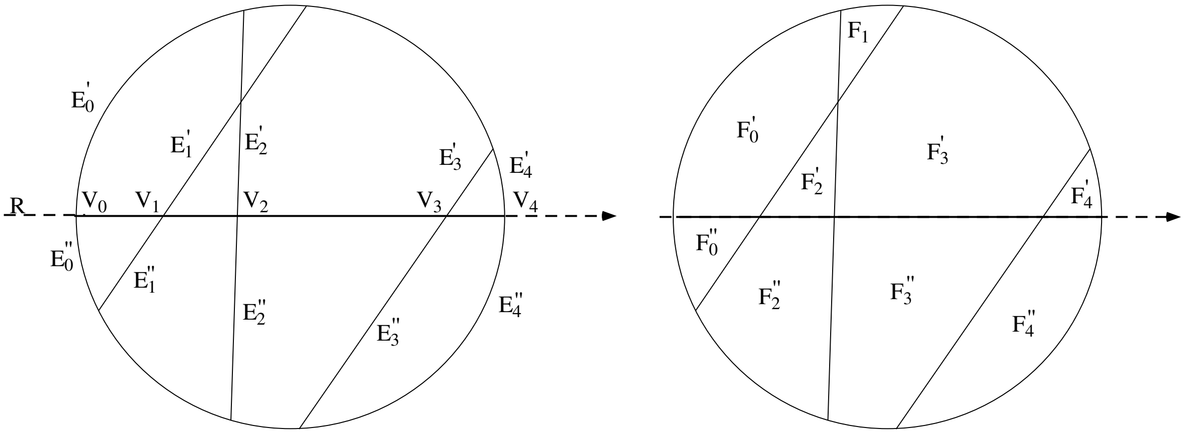


Figure 4.3: Processing a New Ray While Finding a Safe Haven

$$E_N \quad = \quad < V_i, V_{i+1}, F_c, F'_c, E'_i, E''_i, E'_{i+1}, E''_{i+1}, R, a >$$

Where a is used only to tell whether or not an arc edge is a minor or major arc. Two things need to be shown:

1. how to correctly construct E'_i and E''_i. (E'_{i+1} and E''_{i+1} can then be constructed in exactly the same fashion).
2. How to determine the wings. Only the first iteration of the algorithm will have to determine both pairs of wings. After that, the wings E'_i and E''_i will be the wings E''_{i+1} and E'_{i+1} determined in the previous iteration. Note that the clockwise wing in one step becomes the counter-clockwise wing in the next.

In order to determine the wings, first recall that, by definition,

$$E_{i+1} \quad = \quad < v_0, v_1, f_0, f_1, e_0^+, e_0^-, e_1^+, e_1^-, r, a >$$

The new edges E'_i and E''_i are then

$$\begin{aligned} E'_{i+1} &= < V_{i+1}, v_1, F_c, f_1, N_{i+1}, N_i, e_1^+, e_1^-, r, a > \text{ and} \\ E''_{i+1} &= < v_0, V_{i+1}, F'_c, f_1, e_0^+, e_0^-, N_i, N_{i+1}, r, a > . \end{aligned}$$

Starting with edge e_0^+ the faces of the edges surrounding F'_c need to be updated to reflect this fact. Recall that these can be found by simply following the e_0^+ links until the original edge is encountered. If the original E_i was a major arc, determine which if either of the altered E_i and E'_i is a major arc and update the a field accordingly.

Once we've processed all of the planks, we must still find a face which is not on the interior of any plank. To do this, we first mark all the faces which **are** on the interiors of planks, and then simply search the list of faces for one which is unmarked. To mark the faces, we first sort all of the edges into $< p, min(V_0, V_1) >$ order, where p is the number of the associated ray, and $min(V_0, V_1)$ is the endpoint with the smallest x-coordinate, and in case those are the same, the smallest y-coordinate. Proceeding through this sorted list allows us to effectively move along each ray, from intersection to intersection. At each intersection, we can tell the identity of the intersecting ray by examining the wings. We can then determine which sides of the ray are inside and outside of its corresponding plank, and mark the faces associated with the edge we are processing. As we enter the interior of a plank, we increase a counter that tells us how many planks we are currently "inside" of. As we encounter the corresponding edge, which indicates that we are exiting a plank, we decrement this counter. Unless the counter is 0 (thus indicating that the current edge is not on the interior of any plank), we mark both faces of every edge we process.

Once this process is completed, any point inside of any of the unmarked faces will suffice as a safe haven. From Chapter 3, we know that at least one such region will exist, and we can find it by a sequential scan through the faces.

4.1.4 Analysis

The time required by preprocessing is $O(n \log n)$, where n is the number of planks, since the removal of duplicates will require us to sort the normalized representation of the planks.

The maximum number of possible intersections with a given ray will be $2n$, since a ray could conceivably intersect both rays of all the other planks besides its own, plus the safety region boundary twice. Sorting these for the first pass will again take $O(n \log n)$ time. Marking edges to reflect new faces can take $O(n)$ time for a given intersection, but the total number of edges processed for a given ray will not exceed $2n$. So the time to process a single ray is dominated by the time to sort. Since this is done $2n$ times, the time required to find all of the intersection points will be $O(n^2 \log n)$.

Similarly, the second pass, during which faces on the interiors of planks are marked takes $O(n \log n)$ for each of the $2n$ bounding rays, for an asymptotic run-time of $O(n^2 \log n)$. It follows that the overall runtime is also $O(n^2 \log n)$.

It is possible to be smarter about finding the intersection points than is the case with this algorithm. The technique adopted here is essentially the "naive" approach, which takes $O(n^2)$ time, where n is the number of planks. 9 (9) described an algorithm for finding these intersections in $O((k+n) \log n)$ time, where k is the actual number of intersection points. 48 (48) suggest that this algorithm could be adapted to improve the run-time performance. However, in the worst-case, k is n^2, so the worst-case performance is $O(n^2 \log n)$, which is actually worse than the straight-forward algorithm. In the context of a hard real-time system, such as Dodger, it does not make sense to use an algorithm with inferior worst-case performance. Even though it may be possible to use this algorithm to identify faces and mark them in a single pass, the worst-case performance does not improve upon the straight-forward algorithm presented here. 19 (19) do describe an $O(n \log n + k)$ algorithm for finding intersections, and it should be possible to adapt that algorithm to identify and mark faces in a single pass, which would be an improvement over the algorithm presented here. However, even then, the processing overhead is large enough that it would be justified only for relatively large numbers of obstacles.

4.2 Monte Carlo Technique

The method described above for computing the safe haven is both complex and time-consuming. In cases where an extremely small but non-zero chance of failure is judged tolerable, and where it is reasonable to extend the threat horizon somewhat, there is a simple and fast technique for finding the safety-preserving destination for the robot. Assume, for example, that we are able choose the safety radius so that, even in the worst case, the obstacles can never threaten to occupy more than half of the safety region. Then simply picking a random point in the region has a 50% chance of finding a safe haven. The odds of failure are 1 in 2^n, where n is the number of chances we are allowed to pick. The odds of failure in 100 picks are, for all practical purposes, essentially zero. Even if we allow 20 picks before giving up, the system will be unable to find a safe haven only one time in over a million chances.

In order to use this approach, we must be able to determine whether some randomly chosen point is safe with respect to the moving obstacles. This is easy to do when the obstacles travel along linear paths, and we can check n obstacles in $O(n)$ time. We also need to be able to adjust the safety radius so that the obstacles will never be able to touch more than 50% of the safety region. This is not as easy as simply increasing the radius so that the area of the circle doubles, since the area threatened by the obstacles will increase as the size of the circle increases. Section 3.2.1 describes how to increase the safety region to account for the area occupied by stationary obstacles. Equation 2.1 described the relationship between the additional area α and the safety radius R. This can be adapted for our purposes as follows: recall that W is the sum of the widths of the obstacles (or, from another perspective, the diameter of the interior of the safety region.) If we want to ensure that, in the worst-case, a random point selected from an expanded safety region will be a safe haven with probability p, the Equation 2.1 can be adapted as follows:

$$(1-p)\Pi R^2 = \pi\, R^2 - 2\, W\, \sqrt{R^2 - W^2} + 2\, R^2 \arctan\!\left(\frac{W\,\sqrt{R^2 - W^2}}{-R^2 + W^2}\right) \tag{2.1}$$

Again, we cannot find a closed-form solution for R in terms of W and p based upon this equation. However, Dodger implements an interative algorithm that can produce an approximation to R with arbitrarily small precision. The reader should also note that this calculation is **offline**, i.e., it is performed prior to runtime and thus it is not imperative that we be able to compute it quickly. Nonetheless, it finds its results in a matter of milliseconds.

Besides the slight risk of failure, this procedure requires that the safety horizon be extended beyond the minimum necessary to ensure a safe haven exists. This in turn implies that the threat horizon will have to be increased correspondingly. This cost is somewhat ameliorated by the fact that for **any** algorithm, the threat horizon needs to be extended to account for the worst-case time required to find a safe haven. Normally, the worst-case time for the guaranteed algorithm will be much worse than that for the randomized technique. Still, the threat horizon will normally be larger for the Monte Carlo method. In addition, Dodger has been able to use a extended safety region which has only 10% more area than the minimum, and has never required anything close to the 100 throws the current implementation allows for. This is due to the relative rarity of worst-case and near-worst-case configurations in the current system. This will be discussed in greater detail in Chapter 7.

Compare the difference in the conceptual complexity between the technique described here and the deterministic technique of the previous section. In order to benefit from the use of a deterministic algorithm, an implementation of it must ensure that the probability of a bug is smaller than the probability of a failure in the Monte Carlo method. The cost of developing a provably correct algorithm would be justified by a true mission-critical application. Otherwise the likelihood of code failure in the guaranteed algorithm could easily be greater than the likelihood that the randomized algorithm will fail to find a safe haven.

4.3 Dodger Findpath Algorithm

This section describes and analyzes the high-level find-path algorithm the Dodger runs incrementally at regularly scheduled intervals. At the most fundamental level, the path-finding algorithm is designed to make sure that the following invariant remains true at all times:

> Whenever a new obstacle is sensed, Dodger knows a collision-free path to a location that is a safe haven with respect to all of the other, previously sensed obstacles that are in range of Dodger's sensor. In addition, there will exist a (possibly length 0) collision-free path from that location to a location that is a safe haven with respect to all of the obstacles in sensor range, including the newly sensed obstacle. Moreover, Dodger will be able to find and use this path.

A safe haven is a point on the plane that will not be hit by any member of some specified set of obstacles. Chapter 3, Theorem 1 (page 16) proves that no matter where a robot is in the plane, there will always be a safe haven within a distance of $\epsilon + \sum_{i=1}^{n} W_i/2$, where ϵ is an arbitrarily small, positive constant, and W_i is the width of obstacle i. Note that, in the most general case, it is not always necessarily true that a path to it exists, or that if a path does exist, that it can be found in polynomial time, let alone quickly. It is up to Dodger to ensure that, regardless of the deployment of obstacles, a safe haven will always be reachable. This requires that such a safe haven must exist, that a collision-free path from the robot's current location exists, and that Dodger can find both the location and the path quickly enough to reach the safe haven while avoiding obstacles.

The high-level algorithm can then be described as follows. First recall that, at all times, Dodger knows of a reachable safe haven. Assume that it is currently headed towards it. When a new obstacle is sensed, Dodger tries to find a constant velocity path to the goal. If it can, and the goal happens to be a safe haven, Dodger can simply start going to it. However, if the goal is not a safe haven, Dodger selects a safe haven that it can go to via a constant velocity path, after reaching the goal. If such a path to a safe haven via the goal cannot be found, Dodger tries to find a constant velocity path to a heuristically chosen target, that it hopes will take it closer to the goal. Such targets are safe havens by construction. If it cannot do this, Dodger simply continues along its current path: if all else fails, it can get to the safe haven it is currently head towards, and then wait until it finds a constant velocity path to a new location that is a safe haven w.r.t. to all obstacles, including the new one. The threat horizon, which defines the distance at which obstacles are sensed, was constructed to ensure that, even in the case where the robot must wait to find a path to safety, the robot will always be able to find such a path.

The actual implementation of Dodger uses this high-level algorithm as a foundation. The initial implementation, which implemented this algorithm, was markedly unintelligent. For one thing, it made decisions at only two times: when a new obstacle was detected, and when the robot reached a safe haven and needed to determine where to go next. This preserved safety, but it unnecessarily retarded progress to the goal. Since Dodger must be prepared to execute its worst-case code legs at any iteration of its find-path algorithm, and therefore time must be scheduled for the worst-case performance, there was no computational advantage to this policy. Therefore, Dodger now recomputes its path in every scheduled iteration of the find-path algorithm, which results in a considerable improvement in performance. In the original algorithm, Dodger would slow down in order to avoid an obstacle, but would not speed up once the obstacle had passed. In addition, the robot would halt after it picked up a goal, and wait for an obstacle to enter its threat horizon. Incrementally recalculating paths solved both of these problems, but introduced some subtle difficulties.

Here is the actual code of the top-level Dodger find-path algorithm.

```
service
ROBOTfind_path(ROBOT_in: ROBOTsense_msg,  msg )
{
int inside = msg->inside ;
Obstacle *obs = msg->obstacles ;
static Location safety_start ;

/*
** make sure we don't blindly go straight for a distant goal
** while the world changes around us
*/
if (GLOBALgoing_to_safety &&
   (distance( robot.loc, safety_start) > SAFETY_RADIUS))
    waiting = GLOBALgoing_to_safety = 0 ;

if((!GLOBALgoing_to_safety
    && (( distance(robot.loc, goal.loc)) > 0.0))) {
    Segment seg ;
    Location target, targ2 ;

    /*
    ** get a segment representing the goal's location
    ** in 3-space
    */
    seg = get_goal_seg( robot, &goal ) ;

    /*
    ** the if statement tries to get either
    ** 1)a straightline path to the goal, or
    ** 2) a path to some heuristic target as long as
    ** SAFETY_2_GOAL isn't turned on, in which case,
    ** heuristics aren't used                            */
    if ((find_unoccluded_interval(obs,seg,robot,&target ))
#ifndef SAFETY_2_GOAL
        ||
        ((GLOBALgoing_to_safety =
             heuristic_target(obs, goal, robot, &target)
#endif
        ) {
        /*
        ** got a target, but I have to worry about what the
        ** robot is going to do after he gets there. if he
        ** could be hit,he has to worry NOW about where
        ** he'll go after he's picked up the goal
        */
        if (GLOBALgoing_to_safety)
            safety_start = robot.loc ;

        if (!unsafe( target))  {
```

```
            /* first target was safe */
            waiting = 0 ;
            goto_target( &robot, target, 1 ) ;
            }
        else {
            /*
            ** here's where we look for a "next target", to
            ** ensure that there's a path to safety after
            ** the pickup
            */
            if (get_next_target(&robot, target,
                        SAFETY_RADIUS, &targ2 )) {

                waiting = 0 ;

                robot.next_target = targ2 ;
                goto_target(&robot, target, 2 ) ;
                }
            else {
                /*
                ** no safe target, so we have to find a
                ** place to go
                */
                if (inside ) {
                    /*
                    ** the theory says we can't be certain
                    ** of finding a safe haven when
                    ** obstacles are inside of the
                    ** THREAT_HORIZON, but we'll try to,
                    ** and then wait until nothing is inside
                    */
                    waiting =
                    prepare_to_wait(&robot,&target_source);

                else {
                    /*

                    */
                    waiting = 0 ;
                    GLOBALgoing_to_safety = 1 ;
                    safety_start = robot.loc ;
                    goto_target( &robot,
                    find_safety(robot.loc,SAFETY_RADIUS,
                        goal.loc),1 ) ;
                    }
                }
            }
        }
    else if (!inside && !GLOBALgoing_to_safety) {
        /*
```

```
[100]         ** we couldn't find either a straight-line path
[101]         ** to the goal, OR a heuristic target AND there
[102]         ** are no obstacles inside the threat horizon
[103]         */
[104]         waiting = 0 ;
[105]
[106]         GLOBALgoing_to_safety = 1 ;
[107]         /*
[108]         ** get the safety location
[109]         */
[110]         safety_start = robot.loc ;
[111]         goto_target( &robot,
[112]         find_safety(robot.loc,SAFETY_RADIUS,goal.loc ),1);
[113]         }
[114]     else {
[115]         /*
[116]         ** look for safety
[117]         */
[118]         waiting = prepare_to_wait(&robot,&target_source);
[119]         }
[120]     }
[121] }
```

The **GLOBALgoing_to_safety** variable was introduced to prevent Dodger from trying to find a new heuristic target every time it executed the find-path algorithm. The check on lines [012 – 014] was introduced to ensure that Dodger would not find a distant heuristic target and then blindly head towards it while new obstacles appeared to make the path infeasible.

When path updates can occur at any time, we cannot guarantee that the robot can find a safe haven that is a constant velocity trip from the robot's current location. The theory only assure us that such a path can be found eventually. In particular, if obstacles are inside of static-case threat horizon (see Section 3.1.1, page 15), it will not always be possible to find such a path. The **prepare_to_wait** function call on lines [084] and [123] addresses this issue. It first checks to determine whether or not the robot's current location is a safe haven, and if is, causes the robot to stop. Otherwise, it attempts to find another safe haven. Failing that, the function returns a 0, and the robot will continue to travel its current path.

The call to **heuristic_target** on line [038] is only relevant when either Heuristic 2 (Section 4.4.2), or Heuristic 3 (Section 4.4.3) are in use. When Heuristic 1 (Section 4.4.1) is employed, the algorithm substitutes a **find_safety** routine that only considers possible safe havens that are closer to the goal than the robot's current location. The calls to **find_safety**, on lines [094] and [116] use the Monte Carlo method described in Section 4.2 (page 38), or in the case where Heuristic 1 (Section 4.4.1) is used, a modified version that is designed to ensure progress towards the goal, as described in Section 3.4.

The algorithmic complexity of the algorithm is dominated by the call to **find_unoccluded_interval**, on line [034]. This routine takes the target goal segment, and tries to find some part of it that is reachable via a constant velocity path from the robot's current location. This routine finds the regions of occlusion imposed by each individual obstacle (using the algorithm of Section 5.5),and then merges the occlusions using heapsort. Assuming that each of the N obstacles happens to occlude some part of the goal, the cost of the merge is then $O(N \log N)$.

```
if (trajectory != get_direct_path_to_goal())
    trajectory = get_path_to_safe_haven() ;
execute trajectory
```

4.4 Heuristics

Chapter 5 describes an algorithm for computing paths from a robot's location to some goal location in two dimensions, while avoiding obstacles moving at fixed trajectories. While this algorithm may be suitable for use when the number of obstacles in relatively small, Maruti does not support the non-realtime computation required by a such search algorithm. In addition, the problems associated with integrating such a search with a hard real time executive are substantial. Instead of an exhaustive search, Dodger uses bounded, "reactive" routines to accomplish this task. Each of these routines perform reasonably well in practice, as discussed in greater detail in Chapter 7. This section simply describes the high-level algorithms employed.

4.4.1 Heuristic 1

The first heuristic is very simple, and builds upon the results of Section 3.4, which establishes conditions necessary to ensure that the robot can make progress to the goal. Since this routine actually guarantees that progress to the goal can be made, it is not technically a heuristic. By extending the threat horizon sufficiently, it is possible to ensure that the robot can find a safe haven that is always closer to the goal than its current position. Thus, the basic algorithm is simply In order to calculate the relevant trajectory, which is just a direction and a velocity that the robot should adopt, Dodger uses the algorithm described in Section 5.2 for finding the reachable parts of a ray in time-space, as well as that of Section 5.5 for finding the occluded intervals on such an interval, when the origin is known to be a point. All the heuristics used in Dodger make use of these routines. None of these heuristics needs to compute either contact motions or the set of occlusions on an interval when the origin can be an interval in time-space.

4.4.2 Heuristic 2

Like both of the other heuristics, Heuristic 2 periodically tries to find a constant velocity path to the current goal. If no such direct motion exists from the robot's current location, it tries to find a location to move to that is both safe and closer to the goal than the current location. Unlike Heuristic 1, the threat horizon has not been extended to ensure that such a path exists, so if no such point is found, then the robot invokes the Monte Carlo code described in Section 4.2.

This heuristic is not at all intelligent about its choice of possible targets: it simply follows a predescribed regimen for selecting points based upon their distance from the robot and their deviation from the straight-line path to the goal. This approach has two advantages over Heuristic 1: it biases the search for a trajectory towards points that are closer to the current straight-line path to the goal, and it does not require an extension of the threat horizon. If θ is the angle from the robot's current position to the goal, then the high-level algorithm is as follows:

```
(when a direct path to the goal does not exist)
for (j = 1 ; j ≤ F ; j++)
    for (i = 0 ; i ≤ (F-j) ; i++) {
        $target.x = robot.x + (j * D) * \cos(\theta + i * \pi/(2 * F))$
        $target.y = robot.y + (j * D) * \sin(\theta + i * \pi/(2 * F))$
        if ((target is a safe haven) and (trajectory = get_path_to_(target)) {
            execute trajectory
            return ;
            }
        if (i == 0) continue ;
        $target.x = robot.x + (j * D) * \cos(\theta - i * \pi/(2 * F))$
        $target.y = robot.y + (j * D) * \sin(\theta + i * \pi/(2 * F))$
        if ((target is a safe haven) and (trajectory = get_path_to_(target)) {
            execute trajectory
            return ;
            }
        }
```

D and F are both parameters. Basically, D tells the algorithm how far away from the robot's position it should look, and F tells how much a 90 degree angle should be cut to determine divergence from θ. If F is 4, the algorithm tries the positions shown in Figure 4.4, in the indicated order. The semi-circles drawn have radii which are multiples of D.

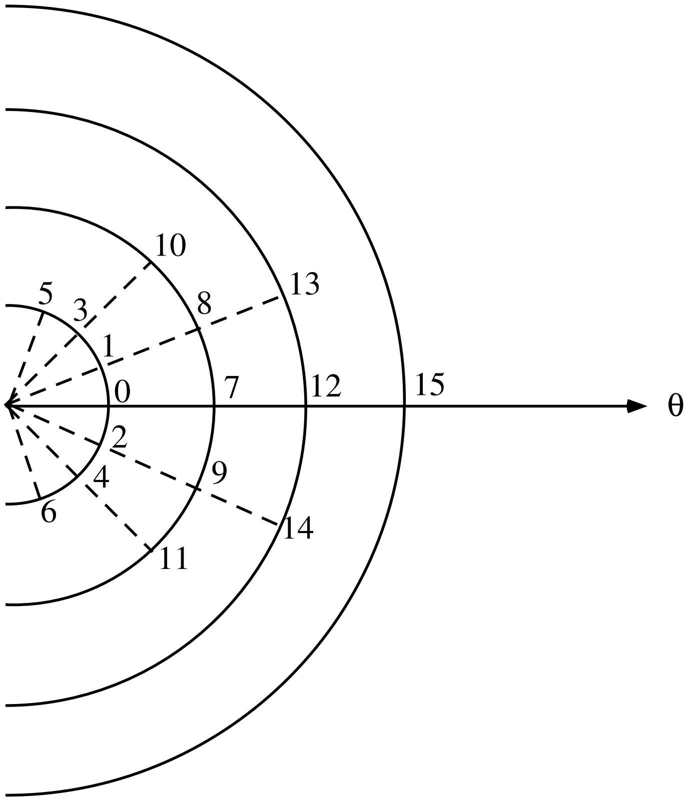


Figure 4.4: Heuristic Search Points

4.4.3 Heuristic 3

The final heuristic attempts to be a little bit smart about where the robot tries to go if there is no direct motion to the goal. It calculates the point at which each of the obstacles passes closest to the goal, and then considers the feasibility of going to a nearby point that is not threatened by the obstacle being considered. The robot considers such points in order of their increasing distance from the goal, and will not go to a point that is farther from the goal than its current position. The logic behind this heuristic is just that, if direct motions to the goals are blocked, the robot should go to some point which is both closer to the goal and outside of the path of the obstacle(s) occluding the direct path to the goal. Thus the algorithm is:

```
(when no direct path exists)
for (i = 0 ; i < number_of_obstacles ; i++)
    obs[i] = point at which obstacle centerpoint passes closest to goal
sort the obs[i] by their distance from the goal

for (i = 0 ; i < number_of_obstacles ; i++ ) {
    let t0, t1 be the two points on the exterior of the
        (expanded) obstacle that are tangent to lines passing
        through the robot's current location
    if either t0 or t1 is safe, and a direct motion to it exists,
        and it is closer than the robot's current distance from the
        goal, go to it (and break from loop)
    }
```

4.5 Summary

This chapter has presented the core algorithms that are at the heart of the Dodger system. It has described a deterministic routine for finding a safe haven with respect to a fixed set of obstacles. This chapter has also presented a Monte Carlo algorithm that is much simpler to implement, but which may occasionally fail to find a safe haven, even though one exists.

Chapter 5 describes an algorithm for finding a path from the robot's location to its goal, while guaranteeing obstacle avoidance. This algorithm is based on Reif and Sharir's, and is guaranteed to find a path if such a path exists. This chapter describes the reactive heuristics used in Dodger, which search for paths one direct motion at a time. Each of this heuristics suffices to produce satisfactory goal achievement in the Dodger system, as is documented in Section 7.2 (page 75).

Chapter 5

Exhaustive Search

In theory, is possible to use an exhaustive search routine that is guaranteed to eventually find a solution in all configurations. In many cases, this search will find a solution quickly. However, in the worst case, the time required to find a solution will be such that the search is obsolete before it has finished. In such cases, the search can be abandoned as soon as it becomes obsolete. If a new obstacle enters the threat horizon, or a known one leaves it, the search will be resumed from scratch. Eventually, a safety-preserving path to the goal will be found[1], and in the process, safety is never compromised.

This chapter develops the algorithm for exhaustively searching for a path to a goal. This algorithm is based upon work first presented in (69), in which the idea of searching through a space of time intervals is first presented. Reif and Sharir were primarily concerned with showing that an algorithm for finding paths in 2-space existed, hence they omit many details and were not concerned with the computational efficiency of their approach. This chapter will discuss the intuitions of the basic algorithm, present solutions to a number of problems which the original paper treats as details, and describe the ways in which this algorithm departs from Reif and Sharir's. Although both algorithms are worst-case exponential, they have the advantage that they will (eventually) always find a path[2]. However, in practice, deep searches will be interrupted by changes in the environment, so that only relatively shallow searches will ever succeed.

5.1 The Basic Algorithm

The fundamental insight of the Reif and Sharir paper is that it is possible to meaningfully reduce the set of all paths through time-space from uncountably many to a large but finite set of "canonical paths"[3]. They show that a safety-preserving path from point A to point B exists if and only if a canonical path exists. This in turns justifies a search in the space of canonical paths. Reif and Sharir characterize these paths as follows:

- A **static movement** is one in which the robot does not move.
- A **direct movement** is a constant velocity movement of the robot, such that obstacles may only be touched at the end points of the movement.
- A **contact movement** is a movement of the robot along the boundaries of an obstacle. In the 2-dimensional case, this movement is required to begin and end at (contact with) vertices of the obstacle.

---

[1] Recall that Corollary 3 of Section 3.4 establishes sufficient conditions for ensuring that the robot can always find a path that takes it closer to the goal than its current position.

[2] Since Dodger is guaranteeing that some path does always exist.

[3] This is my term, not Reif and Sharir's

- A **fundamental movement** is a direct movement, followed possibly by a contact movement. In other words, a fundamental movement is one in which the robot moves between two obstacles, and then optionally along the boundary of the second obstacle.

- A **normal movement** is a (possibly empty) sequence of fundamental movements, satisfying the following restrictions:
  - R1 : Between any two distinct direct movements, there must be a contact movement.
  - R2: No two distinct contact movements are allowed to visit the boundary of the same obstacle.

Reif and Sharir were able to prove that a safety-preserving path from the robot's starting position to its goal existed if and only if there existed a normal movement from the starting position to the goal. The **if** direction is trivial. That is, a normal motion is a safety-preserving path. The intuition of the other direction is just that any collision-free path through time-space which is not a normal motion can be systematically shortened to a collision-free path which is a normal motion. The path constructed in this way is not necessarily the shortest path to the goal, but it is guaranteed to be shorter than any of the uncountably many paths from which it could be constructed. For this reason, a normal motion from the robot's starting position to a goal is referred to as the "canonical" path.

Since a normal motion is a set of fundamental motions, and no such motion may visit any previously visited obstacle, the number of fundamental motions in a canonical path is bounded. This provides the basis for a solution of the problem, since a search for a collision-free path need only consider canonical paths containing $n+1$ or fewer fundamental motions, where n is the maximum number of obstacles. This in itself is not sufficient, since from any given point to the path traced out by any obstacle, there are still uncountably many paths. However, this problem is overcome via the observation that this set of paths can be represented by the origination point and a line segment, where the segment endpoints represent the earliest and latest possible junctures for a robot starting at the origination point and traveling to a given obstacle in a single direct motion.

Omitting a number of details which shall be addressed later, these facts are enough to describe the high-level algorithm, shown in Figure 5.1. This amounts to a breadth-first search through the set of obstacle-to-obstacle paths, none of which can be longer than n, the number of obstacles.

```
/*
** NOTE : || is a concatenation operator on paths
*/
enqueue(pathlist, robot_starting_point)

while ( P = dequeue(pathlist) != NIL)

    if a path P' exists from the end of P to the goal
        halt - we've found path P || P' from the start to the goal

    else
       for every path P' from the end of P to an obstacle
           not yet visited in the path P
               enqueue(pathlist, P || P')

if the process gets here, no path exists
```

Figure 5.1: High Level Search Algorithm

5.2 Reaching an Obstacle From a Point

Reif and Sharir model the robot's position at any given time as a point in space, and obstacles are modeled as convex polygons in the plane. Any given obstacle O_i has a fixed set of $V_i > 2$ vertices, and each vertex traces a line through the configuration space. For them, the problem of reaching an obstacle O_i from a fixed point R in $\mathcal{C}$-space is the problem of reaching any (or all) of the V_i lines traced by the vertices of O_i. Reif and Sharir were only interested in establishing the existence of a path-finding algorithm, so the inefficiencies of this approach were not a concern to them. In contrast, the cost of computing the paths to the $\sum_{i=1}^{n} V_i$ different vertex-paths is a fundamental concern to the Dodger system. This is the primary advantage to modeling objects as circles. Doing so allows the runtime to be reduced to roughly $n / \sum_{i=1}^{n} V_i$ that of the Reif and Sharir algorithm. Since each obstacle has at least 3 vertices, this implies a per-node speed-up of at least 67%. Instead of having to compute paths to each of a set of parallel lines, Dodger only computes the path to the center of the circle representing the obstacle. While this complicates the issue of finding contact-motions, it is justified by the savings in expected execution time. The problem of determining contact motions will be examined below. For the rest of this section and the next section, The problem of finding a path to a circle will be treated as equivalent to finding a path to the circle's center.

Let $[x_0, y_0, t_0]$ be the position of a robot in time-space at time t_0. The set of points reachable by the robot from this point is a cone with its apex at $[x_0, y_0, t_0]$. If the maximum speed of the robot is V_R, the boundary of the cone can be described by the equation

$$(x - x_0)^2 + (y - y_0)^2 = (V_R * (t - t_0))^2 \tag{2.1}$$

In order to find the set of positions of an obstacle reachable by the robot, we need only find the intersection of this boundary with the path traced by the obstacle. That is, an obstacle starting at point $S = [x_1, y_1, t_1]$ and traveling at speed V_O at some angle θ traces the line described by

$$\begin{aligned} x &= x_1 + R * V_O * \cos\theta && (2.2)\\ y &= y_1 + R * V_O * \sin\theta && (2.3)\\ t &= t_1 + R && (2.4) \end{aligned}$$

We want to find the intersection points of this line with the cone of Equation 2.1. It follows that $t_0 + t = t_1 + R$, so

$$t = t_1 + R - t_0 \tag{2.5}$$

Substituting Equations 2.2, 2.3 and 2.5 into 2.1 yields

$$(x_1 + RV_O \cos\theta - x_0)^2 + (y_1 + RV_O \sin\theta - y_0)^2 = (V_R(t_1 + R - t_0))^2 \tag{2.6}$$

This equation can be expanded and then arranged into the quadratic form $AS^2 + BS + C = 0$. Where

$$\begin{aligned} A &= V_O^2 - V_R^2\\ B &= 2(t_0 V_R^2 - t_1 V_R^2 - V_O x_0 \cos\theta + V_O x_1 \cos\theta - V_O y_0 \sin\theta + V_O y_1 \sin\theta)\\ C &= V_R^2(-t_1^2 - t_0^2 + 2t_0 t_1) - 2x_0 x_1 + x_0^2 + x_1^2 + y_0^2 + Y_1^2 - 2y_0 y_1 \end{aligned}$$

Finding the roots of this equation will tell us the time interval(s) that the robot can get to the obstacle. We are only interested in values of R greater than or equal to 0, since negative values imply intersections prior to t_0 in time. Examples of each type of interval are depicted in Figure 5.2. Depending upon the roots, the interval can be either

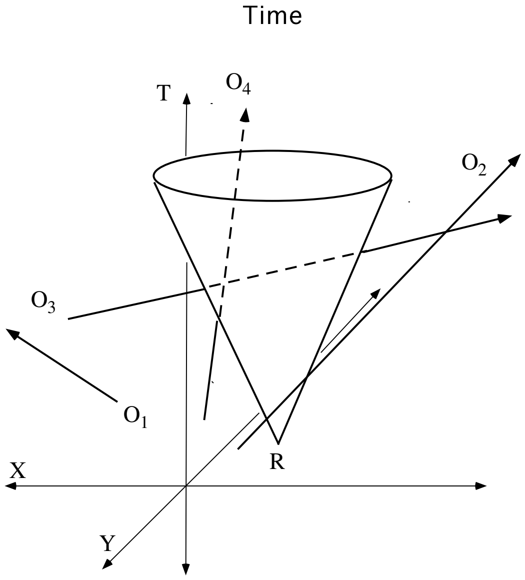


Figure 5.2: Reaching an Obstacle from a Point

- **empty** - implying that the robot can never get next to the obstacle (see the obstacle path O_1 in Figure 5.2).
- a **single point** - implying the very rare case when the obstacle path is tangent to the cone of points reachable from the robot (obstacle O_2).
- a **closed interval** - where one point is the earliest possible position in time that the robot can reach the obstacle, and the other is the latest. This can only happen when the $V_R \le V_O$ (obstacle O_3).
- an **open interval** - implying that it is possible for the robot to reach the obstacle at any time after some initial time t_n. This is only possible when $V_R > V_O$ (obstacle O_4).

5.3 Reaching an Obstacle from an Interval

It rarely happens that the set of points on an obstacle's path which is reachable from a given point is itself a point. Thus, except for the initial case, when the robot is known to be at a single position in time, the search process is usually more interested in finding the set of points (i.e., the intervals in time, or the line segment in $\mathcal{C}$-space) for which one obstacle can be reached from the set of points determined previously in the high-level algorithm presented in Figure 5.1. If the set of originating points is an interval, the procedure described above cannot be used directly: it will not suffice to test the end points. The set of points reachable from a closed interval is depicted in Figure 5.3. The algebraic description of this space is relatively straightforward: we can take Equation 2.1 and modify it to reflect the fact that the initial position may be an interval, rather than a cone, as shown in Equation 3.1.

$$(x - (x_0 + T(x_1 - x_0)))^2 + (y - (y_0 + T(y_1 - y_0)))^2 = (V_R(t - (t_0 + T(t_1 - t_0))))^2$$

$$0 \le T \le 1 \tag{3.1}$$

Where $[x_0, y_0, t_0]$ and $[x_1, y_1, t_1]$ are the end points of the segment. Note that in this equation, T defines the normalized position of the apex of a cone along the base segment. It is not possible to directly solve Equations 2.2,2.3,2.4, and 3.1, for the intersection points, as we did for the case

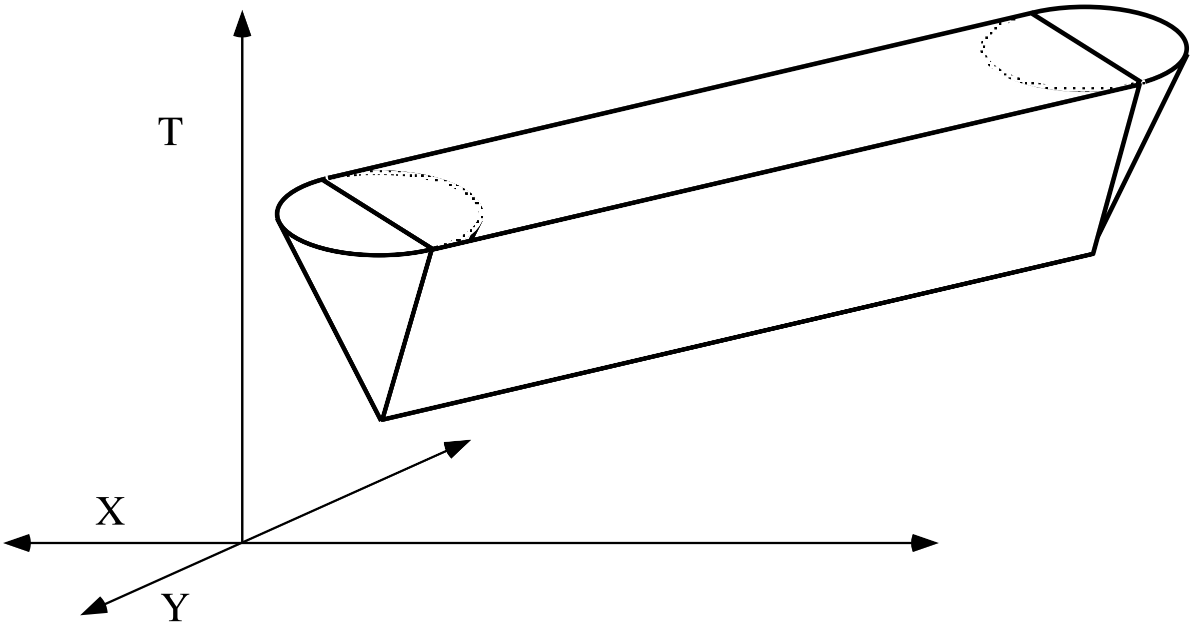


Figure 5.3: Reaching an Obstacle from an Interval

of a simple cone, because for a given intersection point P there can be arbitrarily many values for R and T which will satisfy these equations. The intuition for this is simple: for a given point P, the robot can get there by starting at the latest possible time, and moving at its maximum speed, or it can start earlier, and arrive at the same point by making an appropriate adjustment in its velocity. Thus there is another degree of freedom in the problem that has not been captured by these equations. To solve this problem, the following solution is adopted. First, observe that the space shown in Figure 5.3 can be partitioned into three distinct sections, as depicted in Figure 5.4. These sections are the two "half cones" with apexes on the endpoints, and the darkened area in the figure, which is known as a "wedge", following 18 (18). The formulae which describe the boundaries of this wedge are

$$x = x_0 + R(x_1 - x0) \pm S\,V_R\,sin(\theta) \tag{3.2}$$

$$y = y_0 + R(y_1 - y0) \mp S\,V_R\,cos(\theta) \tag{3.3}$$

$$t = t_0 + R(t_1 - t0) + S \tag{3.4}$$

$$0 \leq R \leq (t_1 - t_0) \tag{3.5}$$

Where $[x_0, y_0, t_0]$ and $[x_1, y_1, t_1]$ are the endpoints of the interval, $t_0 \leq t_1$, θ is the angle traveling from $[x_0, y_0, t_0]$ to $[x_1, y_1, t_1]$, and V_R is the maximum velocity of the robot. It is straightforward to determine the intersection of this boundary with a line, and Section 5.2 describes how to determine the intersection of a line with a cone. It is easy to modify that algorithm to determine the intersection with a half-cone: we simply need to translated the x and y values of any intersection point to a cone rooted at $[0, 0]$ (by subtracting the x and y values of the cone apex), and then rotate the resulting x value through the angle θ (where θ points towards the wedge), using the rotation identity

$$x' = x\,cos(\theta) + y\,sin(\theta) \tag{3.6}$$

If the resulting value of x' is negative, then the original point is on the exterior semi-cone, and hence an endpoint of the intersection with the obstacle's path.

Figure 5.4: Space Partitioned into Easily Computed Sections

Note that the interval at the base of this "extended cone" is itself the result of a previous iteration of the algorithm, and is therefore a segment of the path of one of the obstacles, O_n. It represents all of the various positions in $\mathcal{C}$-space that the robot could be at after following some normal movement which ended at O_n. If the maximum speed of the robot is greater than that of O_n, this segment will be included inside of the cone rooted at $p = [x0, y0, t0]$ which represents the set of points reachable by a robot starting at P. In such cases, the simpler problem of determining the intersection of a line and cone, described in Section 5.2, will suffice.

5.4 Occlusions

The routines above describe how to find the intervals at which an obstacle can be reached starting from a given linear interval in three dimensions. However, these routines assume that there are no obstructions between the starting locations and the destination obstacle. This section discusses the method Dodger employs to determine which, if any, portion of an interval cannot be reached by a direct motion because an obstacle is in the way. If an obstacle does occlude an interval, it may do so in one of three ways:

- It may block out the interval completely.
- It may block out one end of the interval.
- It may block out an interior interval.

The search algorithm proceeds by finding the set of intervals the robot could be occupying after one fundamental movement, then the set it could occupy after two, etc. In the first case listed above, the number of intervals associated with the path through the current interval is actually reduced. In the second case, it remains unchanged. In the third case, two (smaller) reachable intervals are created from the original. If there are n obstacles, and the path between any two of them happens to be occluded by each of the other $n - 2$ obstacles, then there will be at most $n - 1$ sub-intervals added to the search queue. In a worst-case analysis, this has the effect of increasing the branch factor of the search tree from n to $n^2 - n$. In tests of this part of the algorithm, however, there were rarely more than 1 or 2 occlusions, and it is difficult to see what arrangement of obstacles might cause the average number of occlusions to approach the "theoretical" worst-case.

5.5 Occlusions from a Point

This section examines the problem of determining the occlusion created by an obstacle when the position of the robot is a known point in space. The next section uses these results to address the more general (and more common) problem of finding the occlusions created by an obstacle when the exact position of the robot is not fixed, and may be any point on a linear interval.

Assume that we wish to determine whether a particular obstacle O_{occ}, modeled as a circle of width W_O, prevents a robot starting at point R in 3-space, from reaching the center of a second obstacle O_{targ} by a direct motion (i.e., a constant velocity path) over **any** interval of time in the future. In practice, of course, the robot will never attempt to reach the center of an obstacle, but this assumption greatly simplifies the problem, and it is easy to adjust a solution based upon this simplification to one in which the robot will only approach within some distance ϵ of the obstacle's exterior.

Reif and Sharir make the fundamental observation that all of the straight-line paths from R to O_{targ} lie in the plane containing R and the ray traced by the center of O_{targ} over time. Since we can assume that R does not lie on O_{targ}, this plane is well-defined. One can therefore find the obstructions caused by obstacle O_{occ} by first finding the intersection of O_{occ} with this plane, and then finding the set of straight-line paths from R to O_{targ} that pass through this intersection as depicted in Figure 5.5. In general, the intersection of the cylinder occupied by an obstacle in 3-space with a plane is an ellipse, with a minor axis of length equal to the width of the obstacle, W_O, and a major axis whose length is $W_O/\sin\theta$, where θ is the angle between the plane and the

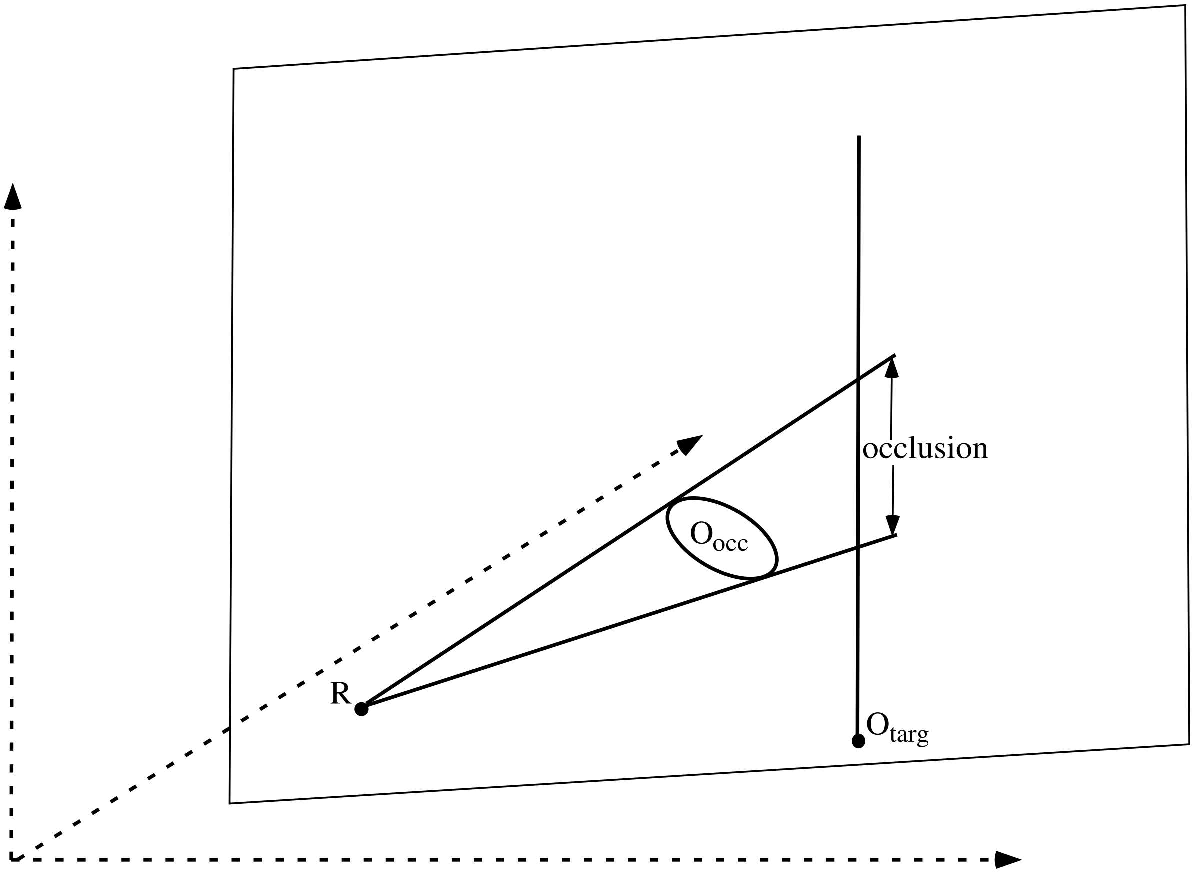


Figure 5.5: Find Occlusions in 3-Space

line traced by the center of the obstacle in 3-space. It is a simple matter to translate coordinates in the plane so that the center of this ellipse is at the origin, and the ellipse is oriented so that its major axis runs along the x-axis. In this case, the equation for the ellipse is

$$\frac{x^2}{W_O^2\ \csc(\theta)^2} + \frac{y^2}{W_O^2} = 1 \tag{5.1}$$

Let the translated x and y coordinates of R be x_0' and y_0', respectively. We want to find the two points on the ellipse, such that the lines passing through each of these two points and (x_0', y_0') are tangent to the ellipse. In other words, we want to find x and y which satisfy Equation 5.1, and which also satisfy

$$\frac{y_0' - y}{x_0' - x} = -\frac{x\ \csc^2\theta}{\sqrt{W_O^2 - x^2\csc(\theta)^2}} \tag{5.2}$$

where the right hand side is just the derivative of Equation 5.1, taken with respect to x. Solving these for x yields the following:

$$x = \frac{W_O^2\, x_0' \pm W_O\, y_0'\ \sin(\theta)\sqrt{{x_0'}^2\ \csc(\theta)^2 - W_O^2 + {y_0'}^2}}{{y_0'}^2 + {x_0'}^2\ \csc(\theta)^2} \tag{5.3}$$

Once the two tangent points have been determined, finding the occluded part of the segment is a matter of determining if and where the rays from (x_0', y_0') to each of these points intersect O_{targ}[4]. Finding the occluded region of an interval is then just a matter of finding the overlap of that interval with the occluded interval found by this process. There are, however, a few complications.

5.6 Degenerate Cases

There is a degenerate case: if the obstacle O_{targ} happens to be traveling parallel to O_{occ}, the two tangent lines will not intersect O_{targ}, regardless of whether or not O_{targ} is occluded, unless by coincidence O_{targ} happens to intersect one of the lines. Rather than follow 69 (69) and impose an artificial constraint on the obstacles, requiring them to travel at unique speeds or trajectories, this can be recognized as a special case and treated separately. Recognizing the special case is a matter of determining whether or not O_{targ} and O_{occ} are parallel to each other in 3-space, which can be the case iff they are traveling with identical speeds along the same angle (modulo π). Treating it is also fairly straightforward. Dodger simply tests whether or not the segment from R to one of the endpoints of the destination interval happens to intersect the cylinder representing O_{occ} in $\mathcal{C}$-space. If it does, then all of O_{targ} is occluded from R, including the destination interval.

5.7 Open Occlusions

It can also be the case that only one of the two tangent lines will intersect O_{targ}. This can happen in the two cases depicted in Figure 5.6. The view is of the plane which contains the robot's location R, and the target ray T. The ovals represent the intersection of some one obstacle's $\mathcal{C}$-space cylinder with this plane. The rays drawn from R and tangent to these ovals delimit the part of this plane which is occluded by the obstacle. In both cases, all but a finite segment of the target is occluded. How is such a situation to be interpreted? Recall that we are only considering those parts of the target that are reachable in a **single, constant velocity motion**. Thus, this situation reflects the fact that the robot must move at some minimum velocity in order to ensure avoidance

---

[4] One needs to be careful about the directions. The lines passing through R' and the tangent points may intersect O_{targ} without O_{occ} actually being between R' and the target obstacle.

of the obstacle. In other words, Figure 5.6(A) depicts the case where an obstacle is moving towards the robot, and Figure 5.6(B) depicts the case where the obstacle is moving away from the robot. In the latter case, none of the target obstacle is occluded, but it is noteworthy, since this situation needs to be distinguished from the previous one, i.e., one cannot simply assume that all open-ended intervals are of one type. It turns out that the two situations can be distinguished by examining the time coordinate of the center of the ellipse in the (untranslated) plane which contains all of the straight-line paths form R to the goal. If this coordinate is greater than the time coordinate of R, then the situation of Figure 5.6(A) is implied, otherwise, the obstacle is moving away from the robot, and we can assume that it doesn't occlude any part of the target.

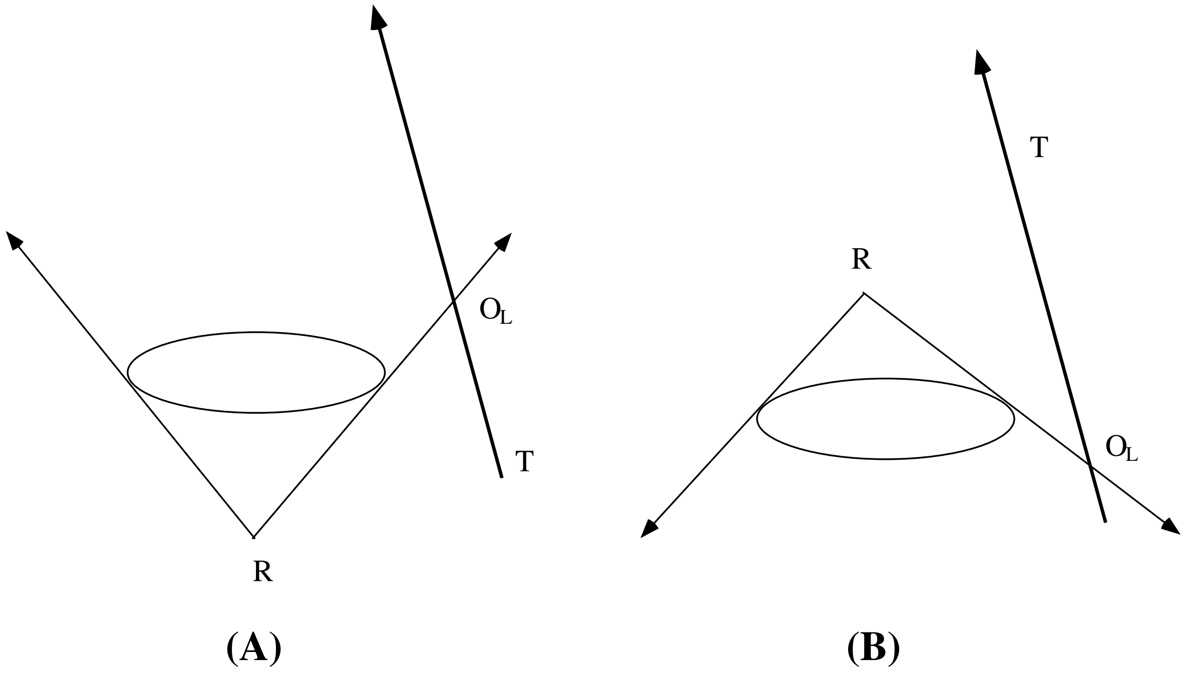


Figure 5.6: Open Ended Occlusions

5.8 Occlusions from an Interval

The preceding section showed how to determine the occlusion of an interval created by a single obstacle when the location of the robot is a fixed point in space. However, the search proceeds through a space of intervals in 3-space, and except for the root of the search, the intervals encountered are only very rarely single points. This section discusses the more general problem of finding an occlusion created by a single obstacle on some target interval, where the origin is allowed to be an interval on a line in 3-space.

First, some terminology is in order. Instead of originating at a single point R, the origin of the robot is now an interval, from R_L to R_H, where the time value of R_L is less than or equal to the time coordinate of R_H. (L is being used to designate "low" values, H for "high" values). R_H may equal infinity, in cases where the robot moves faster than the obstacle. The destination interval is again the path of the center of some target obstacle through 3-space, but in this section it will be referred to as $[T_L, T_H]$, again where T_L is prior to T_H in time. In the context of the search algorithm, the interval $[R_L, R_H]$ represents the time the robot could be "near" some obstacle after some number of fundamental motions. $[T_L, T_H]$ is the part of an obstacle path reachable by a single, direct motion from $[R_L, R_H]$, corresponding to the next fundamental motion in the search.

If R is a fixed point in the interval $[R_L, R_H]$, then in general, only a subset of the points $[T_L, T_H]$ is reachable from R, and this interval will be called $[V_L^R, V_H^R]$ or in cases where there is no ambiguity, simply $[V_L, V_H]$. Thus the relationship between $[R_L, R_H]$ and $[T_L, T_H]$ can be expressed as

$$[T_L, T_H] = \bigcup_{R \in [R_L, R_H]} [V_L^R, V_H^R]$$

For a given occluding obstacle O, if we fix $R \in [R_L, R_H]$, then the occlusion on $[T_L, T_H]$ from R and caused by O is $[O_L^R, O_H^R]$, subject to the same naming conventions, i.e., O_L^R precedes O_H^R in time, and where the superscript can be dropped when there is no ambiguity. Figure 5.7 depicts a typical occlusion and the locations of $[R_L, R_H]$ and $[T_L, T_H]$. Two points on the interval $[R_L, R_H]$ have been fixed: R_1 and R_2. For each of these, the corresponding $[V_L, V_H]$, and $[O_L, O_H]$ have been shown. The two-dimensional plane shown is the plane containing R and the interval $[T_L, T_H]$. All of the other lines are projections into this plane. In general, $[R_L, R_H]$ and $[T_L, T_H]$ are not co-planar. Since we are free to rotate and even “flip” the plane, it is possible to normalize the picture so that $[R_L, R_H]$ always appears below $[T_L, T_H]$, and the values T_L, O_L and V_L always appear to the left of their corresponding endpoints, T_H, O_H and V_H. However, the fact that R_L always appears to the left of R_H in the figures of this section is merely an adopted convention, so that time can be thought of as increasing as one moves from the lower left of the page to the upper right.

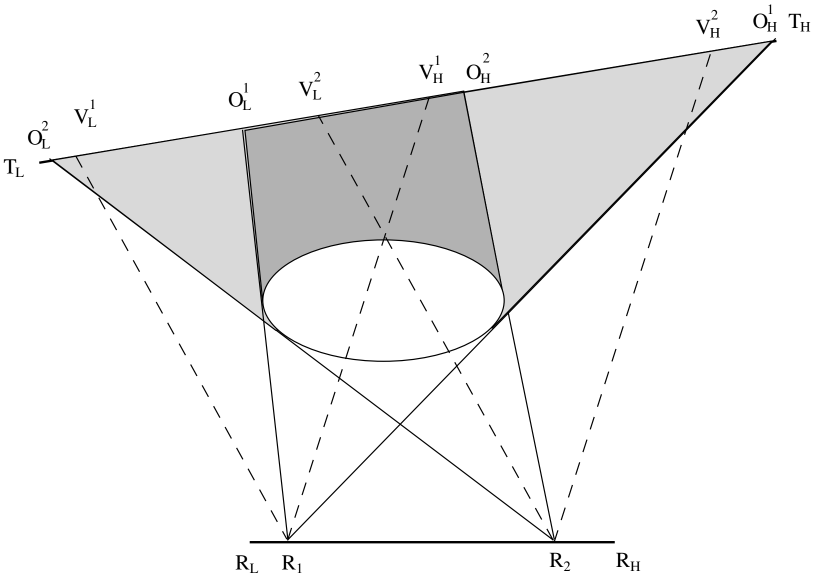


Figure 5.7: Occlusion in a Plane

For the purposes of the search, we want to find those points on $[T_L, T_H]$ which are not reachable from any point $R \in [R_L, R_H]$, due to an occlusion caused by obstacle O. Formally, we wish to find

$$\bigcap_{R \in [R_L, R_H]} (([O_L^R, O_H^R] \cap [V_L^R, V_H^R]) \cup ([T_L, T_H] - [V_L^R, V_H^R])) \tag{8.1}$$

There is no obvious way to obtain a closed form solution to this problem. Reif and Sharir suggest a plane-sweep algorithm in the two-dimensional space defined by the cross product of $[R_L, R_H]$ and $[T_L, T_H]$, where all of the n obstacles are processed simultaneously in $O(n^2 \log n)$ time. However, they are content to note that the relevant curves are “algebraic functions in t of constant degree”. Even in their case, where the use of polygonal obstacles will simplify the computation, these promise to be complex functions of many terms. In Dodger’s case, where Equation 5.1 would be but one of several simultaneous equations, the description of a single curve would likely contain scores, or even hundreds of terms, and computing the intersections of these curves would be even more complex. The solution suggested here is an iterative approximation algorithm, which allows us to find approximate solutions to Equation 8.1 that are arbitrarily close to the actual values. Using it allows the per-destination runtime to be reduced from $O(n^2 \log n)$ to $O(n)$, with substantially smaller constant coefficients. The solution is based upon the following observations:

1. The occluded area is continuous.
2. Visible segments move "with" origination points, i.e.,
$$\begin{aligned} time(R') > time(R) &\rightarrow time(V_L^{R'}) \geq time(V_L^{R}) \quad \text{and} \\ time(R') < time(R) &\rightarrow time(V_L^{R'}) \leq time(V_L^{R}). \end{aligned}$$
3. Occlusions move in the opposite direction of origination points, i.e.,
$$\begin{aligned} time(R') > time(R) &\rightarrow time(O_L^{R'}) \leq time(O_L^{R}) \quad \text{and} \\ time(R') < time(R) &\rightarrow time(O_L^{R'}) \geq time(O_L^{R}). \end{aligned}$$

To see what is meant by the second two points, consider Figure 5.8. Part (A) shows a typical occlusion, and part (B) shows how the figure is changed for some time(R') < $time(R)$, and part (C) shows the change for time(R'') > $time(R)$. This phenomenon can be exploited to implement a binary search that can be used to find arbitrarily close approximations to the actual occluded interval. Let $[I_L, I_H]$ be the occlusion on $[T_L, T_H]$ from $[R_L, R_H]$ caused by some obstacle O. It

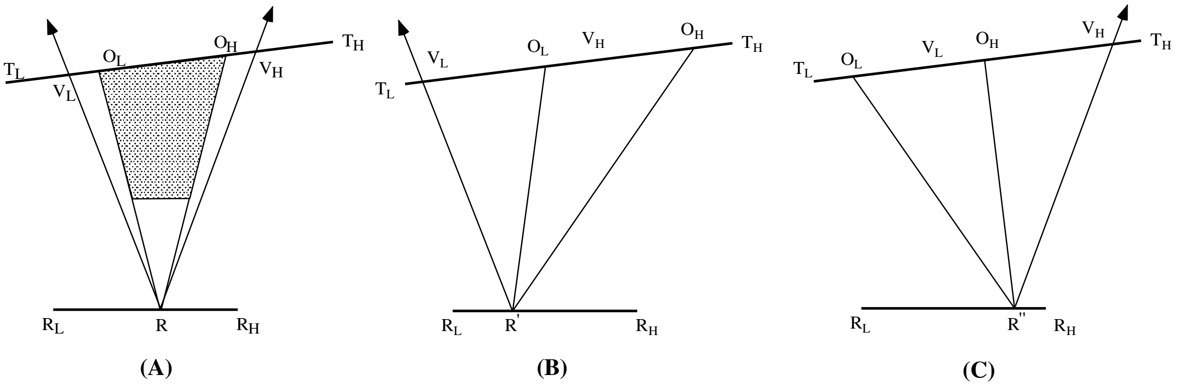


Figure 5.8: Occlusions and Reachable Segments Move in Opposite Directions

follows from the observation above that I_H is the point at which $O_H = V_L$, for some $R \in [R_L, R_H]$. Similarly, I_L is the point at which $O_L = V_H$ for some (different) $R \in [R_L, R_H]$. We can search for each of these points using two separate binary searches. To search for the value of I_H the algorithm of Figure 5.9 suffices. The value of ϵ in line 3 is intended to be an arbitrary constant, specifying the desired precision of the computation. In a hard real-time application, one may instead wish to bound the number of times the **while** loop may execute. Line 14 is designed to ensure that the occluded region is alway overestimated. The algorithm to determine the occlusion in line 5 was described in Section 5.5 (page 52). The algorithm for the computation in line 6 was described in Section 5.2 (page 48). To modify this to find I_L, one need only change the test in line 7 to

```
7:  if ($O_L == V_H$)
```

and the test of line 9 to

```
9:  else if ($O_L > V_H$)
```

and finally, the safety-ensuring test of line 14 needs to be

```
14: return the minimum of $O_L$ and $V_H$
```

```
low ← $R_L$
high ← $R_H$
while (absolute_value(high - low) > $\epsilon$ ) {
    $R$ ← median(high, low) ;
    $[O_L, O_H]$ ← the occlusion on $[T_L, T_H]$ from $R$ caused by $O$ ;
    $[V_L, V_H]$ ← the points on $[T_L, T_H]$ visible from $R$
    if ($O_H$ == $V_L$)
        this point is the value of $I_H$, halt the search
    else if ($O_H < V_L$)
        high ← $R$
    else
        low ← $R$
    }
return the maximum of $O_H$ and $V_L$
```

Figure 5.9: Algorithm for Finding I_H

5.9 Merging Occlusions

When Dodger searches for occlusions, it checks $n-2$ obstacles to see if they happen to occlude the path from an interval on one obstacle's path to an interval on anothers. In theory, it is possible to end up with as many as $n-2$ occluded regions of the target interval, and each of these may overlap with others. The merging process is straightforward, and is only mentioned here because, in the worst case, this sorting dominates the per-node computation time.

It is also worth noting a useful pruning heuristic: once the robot has "visited" and left an obstacle that moves faster than the robot's maximum speed, that obstacle can no longer occlude a feasible path. Once a fast-moving obstacle has passed the robot, we no longer need to be concerned about whether or not it can hit the robot. Nothing the robot can do will put it in the path of that obstacle. Therefore, one only need to consider as possibly-occluding obstacles those which are either 1) slower than the robot or 2) not just visited in the partially-computed path. This will have the effect of reducing the average number of potentially-occluding obstacles, and thus reduce the average per-node computation time.

5.10 Contact Motions

Recall that Reif and Sharir modeled obstacles as polygons in (69). One of their basic results was that, if a path from the robot's original location to a goal exists, then a path exists that consists of no more than n fundamental motions, where a fundamental motion is a direct motion, followed by a set of contact motions. So far, we have considered the problem of finding direct motions. Contact motions are movements between adjacent vertices of an obstacle polygon. Throughout this section, it will be assumed that the obstacle has been enlarged by some small constant size ϵ, so that tracing the edge of the obstacle does not imply that contact is actually being made, but rather that the robot is maintaining a constant distance of ϵ between itself and the obstacle.

In their paper, Reif and Sharir, implicitly assume that contact motions cannot be impeded by other obstacles, or in other words, that we needn't consider the possibility of colliding asteroids. And again, they are content simply to establish that it is possible to compute the times the robot could get to a vertex b' given that we know the time interval it could be at a vertex b on the same obstacle. While Reif and Sharir ignore some of the difficulties in doing this [5], there is no doubt that it can be done. Even if one allows the possibility of colliding obstacles, algorithms have already be given for computing the times a robot could be at some destination vertex p' if we know the times it

---

[5] They assume that finding the intervals that the robot can be at a given obstacle vertex is a simple matter of finding the time it takes the robot to traverse the edges between them, without taking into account that such a path may not in fact be feasible in time-space. Generally, they seem to forget that, even during contact motions, obstacles are moving.

can start traveling from an origination point p, and the robot is constrained to move at a constant velocity. Computing the earliest and latest times the robot could travel to the various points of a polygon, given the earliest and latest times it could be at some initial point p on that polygon, is simply a matter of using the already-developed algorithms to find the relevant intervals for the two adjacent points, and then continue around the perimeter until bounds have been established for all of the points of the polygon.

This approach suffers from two deficiencies: first, it requires $O(v)$ computations per node, where v is the number of points in an obstacle polygon. More importantly, each of these intervals then becomes a basis for the remaining search. Furthermore, rather than being a search from obstacle to obstacle, it is a search from obstacle vertex to obstacle vertex. In other words, the branching factor at a node, before considering occlusions, is $O(nv^2)$ using Reif and Sharir's approach, where n is the number of obstacles, and v is the number of vertices in an obstacle.

The approach advocated here has been to compute paths from centers of obstacles to centers of obstacles, while remaining cognizant of the fact that these paths cannot actually be traversed in their entirety. Consider the bottom part of Figure 5.10. The search proceeds as if the robot visits the indicated obstacle O_V (represented as a cylinder in time-space), on the incoming path labeled I and travels to a second obstacle, O_T along outgoing path O. Clearly, the part of its projected path from v_I to the center of the obstacle to v_O is not feasible. But if this path turns out to be a part of a path to a goal, then an actual, feasible path will be computed. Doing this reduces the branching factor of the search from $O(nv^2)$ to $O(n)$. The top half shows the "view from the top",

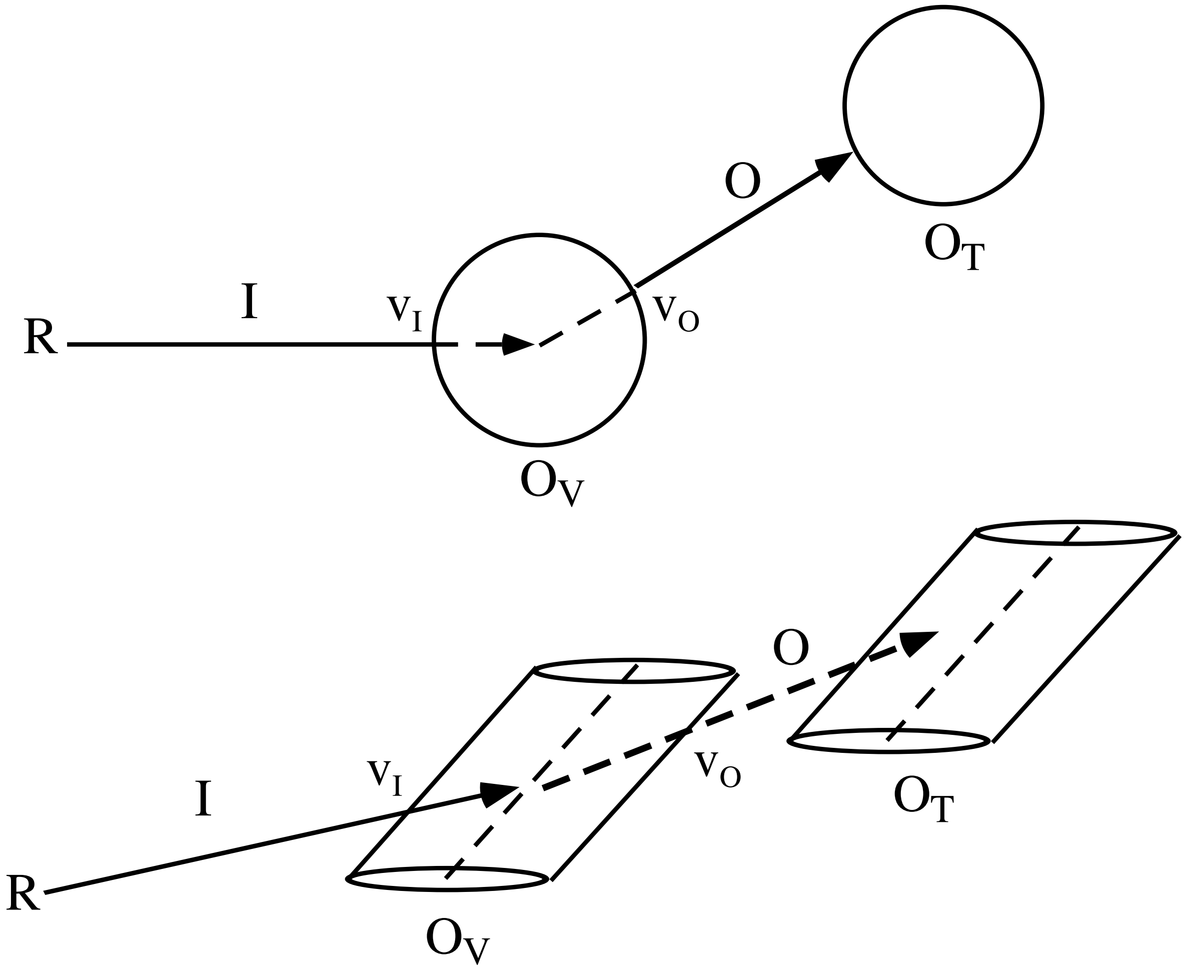


Figure 5.10: Centerline-to-Centerline Search

i.e., it is a depiction of the same situation in the 2-dimensional plane.

When obstacles are slower than the robot's maximum speed, and the incoming path is as illustrated, (i.e., a straight line between two points), the curve between v_I and v_O (in bold on the figure), can be found by first finding the median point of this curve, call it v_M, and then finding the intersection between the obstacle cylinder and the plane containing v_I, v_O and v_M. This intersection will be an ellipse which contains v_I, v_O and v_M, and the desired path will be the minor arc between v_I and v_O. v_M can be found as follows: First, normalize points so that the center of the obstacle cylinder is the z-axis in 3-space. Assume that, after translation, $v_I = (x_I, y_I, z_I)$, $v_O = (x_O, y_O, z_O)$, and $v_M = (x_M, y_M, z_M)$. The z-coordinate of v_M will be intermediate between the z-coordinates of v_I and v_O, i.e.,

$$z_M = \frac{z_I + z_O}{2}$$

To find x_M and y_M consider Figure 5.11, which is the projection of the normalized obstacle cylinder into the (x,y) plane. Clearly, v_M will be intermediate on the minor arc between v_I and v_O. In other words, if W is the width of the obstacle

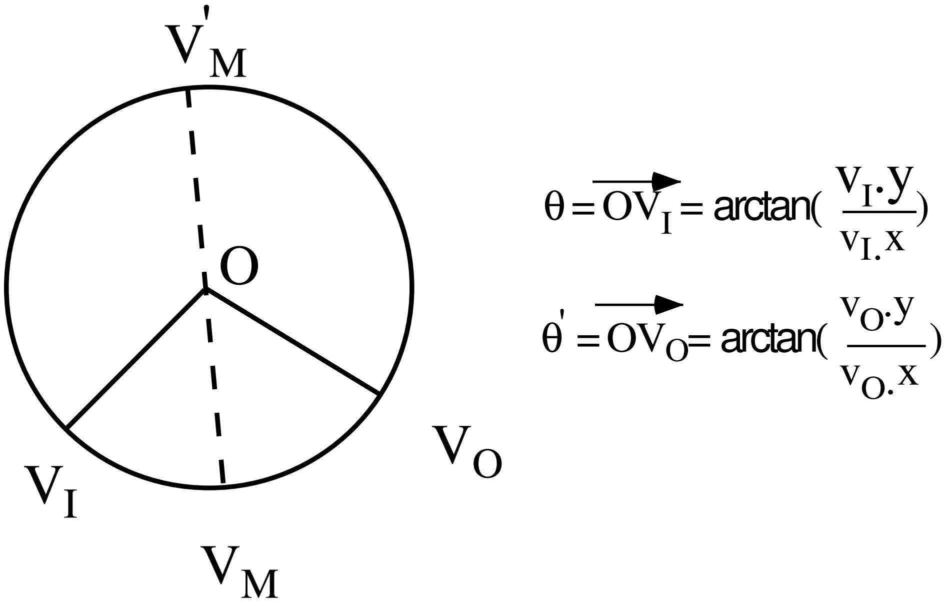


Figure 5.11: Finding the Midpoint of a Contact Motion

$$x_M = W * \cos\left(\frac{\arctan(y_I/x_I) + \arctan(y_O/x_O)}{2}\right)$$
$$y_M = W * \sin\left(\frac{\arctan(y_I/x_I) + \arctan(y_O/x_O)}{2}\right) \tag{10.1}$$

if $|\arctan(y_I/x_I) - \arctan(y_O/x_O)| \leq \pi$. Otherwise, in order to ensure that v_M is on the minor arc between v_I and v_O,

$$x_M = W * \cos\left(\pi + \frac{\arctan(y_I/x_I) + \arctan(y_O/x_O)}{2}\right)$$
$$y_M = W * \sin\left(\pi + \frac{\arctan(y_I/x_I) + \arctan(y_O/x_O)}{2}\right) \tag{10.2}$$

Feasibility can be tested by checking whether or not the computed path intersects the cone which bounds the points reachable by the robot from v_I. Except in very rare cases where the entire arc **is** on the boundary, or where it touches the cone at a point, an intersection implies that the path is not feasible, i.e, that the robot cannot travel quickly enough to traverse the desired path. In addition, we need to test some single point to make certain that it is inside this cone, since it is possible that the entire arc (besides the endpoint v_I) will be outside of the cone, and will therefore not intersect it, even though the path is not feasible. If it turns out that this arc is not a feasible path, and if the obstacle is slower than the robot, the major arc of Figure 5.11 may be feasible, and the corresponding path can be found by repeating the above process, but inverting the use of Equations 10.1 and 10.2.

Unfortunately, the paths found in this way will, in general, not be feasible for obstacles that can travel faster than the robot. This is because traversing the indicated path along the obstacle cylinder is basically requiring the robot to maintain contact with the obstacle while moving around it. Consider the top part of Figure 5.12.The obstacle O_V is moving in the direction indicated by the bold arrow, i.e., directly towards the robot. The target obstacle O_T is moving in the same direction, though at a slower speed. The robot's goal is at G. A 3-D depiction is in on the bottom of Figure 5.12. Assume that both obstacles are faster than the robot. Clearly, no point v_I on a path through the centerline of O_V is feasible, because such a path implies a collision with the obstacle O_V. But there is a feasible path that travels from R to v'_I (or v''_I), then to v'_O (resp. v''_O) and then to O_T. Computing the set of **all** such points is complicated, but we can instead use the (infeasible) time that the robot would be at the centerline of O_V, and easily find the 2 points

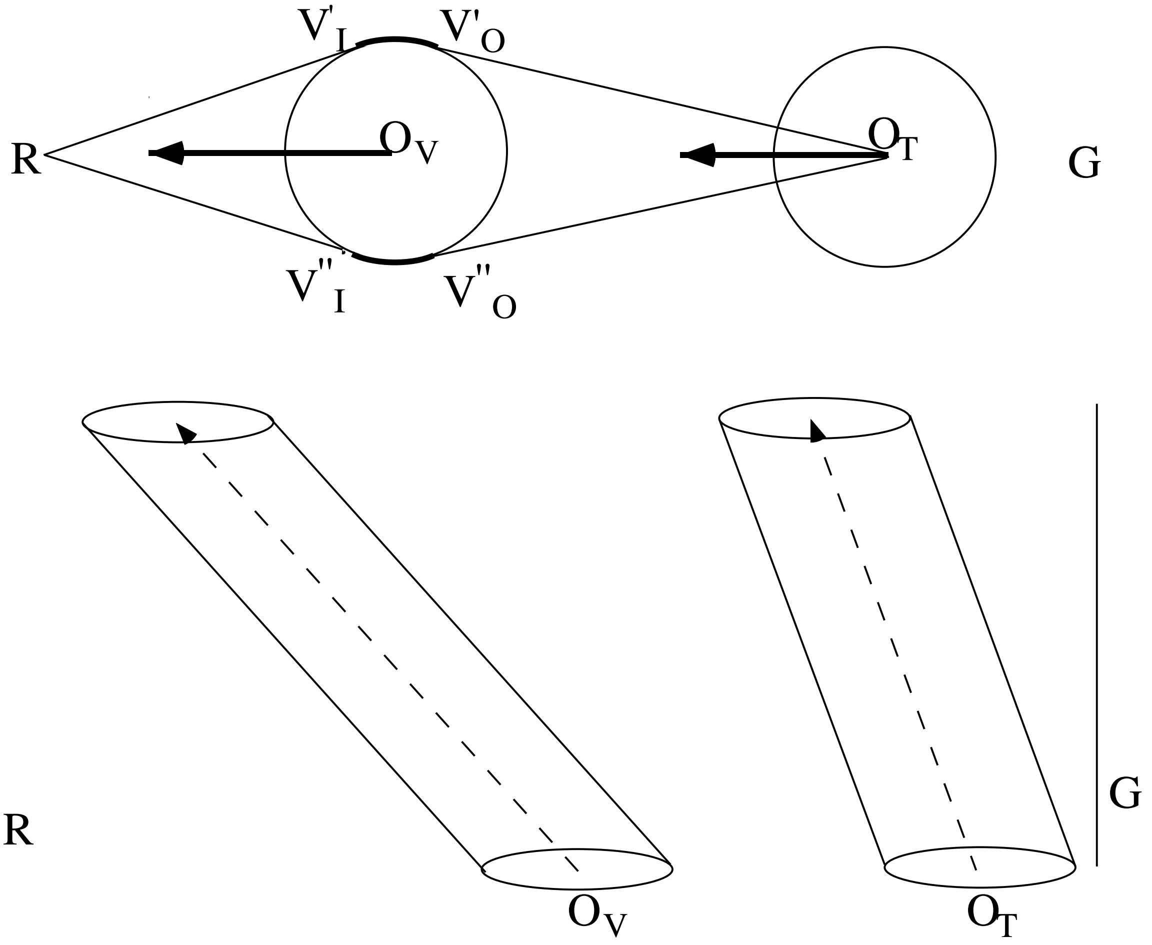


Figure 5.12: An Example of a Feasible Contact Motion

v'_I and v''_I, on the exterior of O_V such that the line containing v'_I and R is tangent to the circle ounding O_V, as is the line containing v''_I and R. Similarly, we can find points v'_O and v''_O using v_O to provide a time estimate, and we can then use the technique described above for finding paths for v'_I to v'_O and from v''_I to v''_O. At least one of these paths will be feasible, regardless of the initial estimates v_I and v_O.

5.11 Path Instantiation

The search process just described (in Sections 5.1-5.10) finds a set of centerline-to-centerline direct motions that take the robot from its initial location to its goal. Normally, each of the centerline-to-centerline motions found will not be a single point-to-point motion, but rather a set of interval-to-interval motions. Thus the search has not found a single path, but an uncountably large set of paths, each one viable modulo the need to find appropriate contact motions described above. After the search has found a potential path to a goal in this fashion, the **actual** path to travel needs to be determined. This amounts to choosing specific points on the various time intervals to visit the obstacles, which in turn translates to choosing the speeds and trajectories at which the robot will travel for its direct motions, as well as the actual contact motions to be used.

If, as in the case of Dodger, a hard real-time operating system like Maruti is being used to provide the necessary timing support, there is another issue that needs to be resolved during the instantiation phase. Maruti is discussed in more detail in Chapter 6 (page 67). It provides the system-level support necessary to ensure that Dodger will **always** have the time required to compute and execute the movements required to ensure obstacle avoidance. To do this, Maruti precomputes a cyclic schedule of exactly when every process in the system will execute. Thus, a path instantiation algorithm will not be free to choose **any** path found during the path search phase. Instead, it must take into account the times Maruti has set aside to execute stops and turns. Even in a simulation like Dodger, where stops and turns can be assumed to be instantaneous, this is still an issue. The simplest way to resolve it is to tell Maruti to set aside time for executing turns and stops at fixed intervals in its schedule. The path instantiation algorithm can then choose paths between points whose time coordinates fall at the beginning of one of these intervals. This is problematic from the scheduling perspective, in the following sense: the farther apart the intervals, the more viable paths that have to be thrown out because appropriate locations cannot be found during instantiation. But the closer the intervals, the more difficult it will be for Maruti to find a schedule.

5.12 Path Search Complexity Summary

In the worst-case, the path from one interval to another on object center-lines can be occluded by $n-2$ obstacles, and thus the branching factor in the worst case can be $n-1$ just to account for the number of intervals that must be considered to visit a single obstacle. However, the average case must be much smaller than this. Experience with a number of test cases suggests that it is rare for more than two obstacles to occlude such intervals, and also that most commonly there are no occlusions. Using this experience to make a conservative estimate, assume that there are on average two occlusions for each direct motion. Then each direct motion would result in a (worst-case) addition of 3 intervals into the search queue. Since this must be done for each of the target obstacles to be visited, the average-case branching factor will be bounded above by $3(n-1)$, where n is the number of obstacles. Since the number of obstacles to be visited decreases as the algorithm gets deeper into the search, and since we can assume that fast moving, previously visited obstacles cannot occlude paths, this number gets smaller at each level of the search.

The per-node execution time is dominated by the time required to merge occlusions, which is $O(n \log n)$ in the worst case. Again, this worst case only occurs when the number of occluding obstacles approaches n. Assuming again that the average number of such occlusions is actually a small constant, then the per-node computation time is simply $O(n-d)$, where d is the current depth of the search tree. This reflects the fact that a bounded amount of computation needs to take place to find the reachable intervals on $n-d$ different obstacle paths. So the amount of per-node computation also decreases with the depth of the search. Nonetheless, the overall time to find a path in the worst-case will be $O((3n)^n)$, assuming an average of two occlusions per direct motion.

Chapter 6

Dodger

Dodger solves the dynamic Asteroids problem in hard real time. Using the theoretical results of Chapter 3, and the algorithms of Chapter 4, Dodger has avoided over 1.5 millions obstacles in over ten CPU-weeks of testing in a simulated domain. This chapter contains a description of the Dodger program, the particular problem it solves, its high-level architecture, the system-level support it requires to operate, and the way the Maruti Hard Real-Time Operating System provides this support.

Chapter 3, develops the theoretical results upon which Dodger is based. In particular, it determines the **threat horizon**, which is the distance at which the robot must sense a new obstacle in order to guarantee that the obstacle can be avoided. This distance is a function of the robot's maximum speed, sensor delay times, the number of obstacles, their speeds and sizes, and the time taken to compute a path. Chapter 4 describes the algorithms Dodger employs to implement the theoretical results. Section 4.3 (page 39) describes the core path-planning algorithm.

The Dodger problem space has been constrained to the set of problems for which solutions are guaranteed to exist. This constraint is strong, in the sense that it reduces the complexity from $\mathcal{NP}$-complete to low order polynomial, but it is natural. Basically, problems are restricted to those in which moving obstacles must originate some bounded and pre-determined distance away from the robot. Determining this distance is the main topic of Chapter 3. Although the maximum number of obstacle with which Dodger must be concerned at any given time is fixed, the number of obstacles that Dodger avoids over time is unbounded. Similarly, Dodger is charged with moving from an initial location through an unbounded series of goal locations, where the goal locations are made known to Dodger one at a time.

A robot situated in the world will not have the luxury of taking as long as it needs to compute a solution. The obstacles are moving, and if timely action is not taken to avoid them, the robot will get hit. Path finding must occur quickly. But "quickly" is a relative term. In order to guarantee avoidance, paths must be found **quickly enough** to avoid collisions. Dodger directly addresses and accounts for the issue of timeliness. For example, there is a delay from the time an obstacle is sensed until the time that the control system "knows" of its existence. This delay is bounded, and Dodger knows the bound. It knows how long it takes to compute solutions, in the worst case, and it knows how to ensure that it never gets itself into a position where it will not be able to find a solution before it is hit.

In its current form, Dodger is a reactive system. All computations are made according to a fixed, pre-computed periodic schedule, and are made in bounded clock time. In principle, there can be two main components to the system: the reactive, hard real-time competences which are responsible for guaranteeing safety, and a non-realtime problem solving component that finds safety-preserving paths to goals. Chapter 5 describes an algorithm that searches for a path from the robot's location to a goal, and that is guaranteed to find such a path if it exists. Even though a complete search is not always time-feasible in a dynamic environment, this algorithm is feasible for relatively shallow searches. The results of Section 7.1 (page 71) suggest that it will also be useful in many domains, where the worst-case behavior will be unacceptable, but the average case performance is adequate. As long as avoidance is guaranteed, the primary problem is quickly finding and executing paths to the goal.

However, since Maruti does not support the sort of processing model that this algorithm requires, Dodger relies upon heuristic routines to find paths to goals. These heuristics are described in Section 4.4, starting on page 43. Each of them runs in bounded clock time, but they are all heuristic in the sense that they do not explicitly search for safety-preserving paths to the goal, even though such paths are guaranteed to exist. In fact, one of the "heuristics" has actually been shown to be sufficient to ensure that progress is always made towards a goal, and thus is theoretically complete. In practice, each of these heuristics is sufficient to support very effective goal achievement, without compromising safety.

The first section of this chapter will describe the particular class of problems that Dodger is designed to solve. The second section contains an overview of the high-level system architecture, which is based upon the Dynamic Reaction model of 71 (71). To guarantee that obstacles can be avoided in real time, one must establish that all of the necessary data collection and computation can be performed soon enough that the system can execute the solution. This requires low-level systems support, which the Maruti operating system provides. Section 6.3 briefly describes the Maruti system, and shows how it is used to ensure the timeliness of all Dodger computations. This chapter ends with a discussion of ways that the current Dodger program could be extended to solve more general problems. Chapter 3 treats these issues in greater detail.

6.1 The Dodger Problem Space

Dodger solves the "Dynamic" Asteroid Avoidance Problem, where the positions of the obstacles are not known at the beginning of program execution, but rather become known to the control algorithm as time passes. All obstacles are constrained to move at a single, uniform speed. There are no static obstacles, and sensing is perfect when obstacles move within the range of Dodger's sensors. The number of obstacles that may appear over time is unbounded, but the maximum number of obstacles that can appear on the screen at any given time is bounded and known to Dodger in advance. All obstacles move at fixed velocities, which are known to Dodger as soon as the obstacles are sensed. The number of obstacles, the maximum speed of the robot, and the (uniform) speed of the obstacles are all parameters to the program, but they do not change after execution begins. All obstacles have the same width, which is also a program parameter. Many of these constraints can be relaxed. Section 6.4 briefly examines ways in which the current program could be extended to solve more general problem classes, and Chapter 3 treats many of the related theoretical issues in much greater detail.

Corollary 1 (page 19) provides the theoretical basis for the obstacle avoidance competences. To summarize, it says that a safety-preserving path to a location that is not threatened by any obstacle exists, and can be found, as long as all obstacles are sensed at a distance of $H = W + nW(V_o + V_r)/2V_r$ away from the robot, where n is the number of obstacles, W is the sum of the widths of the obstacles, V_o is the speed of the obstacles, and V_r is the maximum speed of the robot. This theoretical result assumes that Dodger has perfect, instantaneous sensing, and does not consider the time required to compute safety-preserving paths.

To account for the more realistic case, in which both sensing and computation take time, Dodger has a single (simulated) obstacle sensor, that rotates at a frequency f where f is also a program parameter. This sensor sweeps the plane and reports all obstacles that it senses within a predetermined distance of the robot. This distance is $H + V_o/f + \lambda V_o$, where λ is an upper bound on the time required to sense and compute a solution to the problem. This distance is both necessary and sufficient to ensure that Dodger can guarantee avoidance. If an obstacle is moving directly at the robot, and is just outside of the sensor's range on one sweep of the sensor, there will still be enough time for Dodger to compute a solution when the obstacle is detected on the next sweep, and assure that Dodger has sufficient time to execute the solution. Figure 6.1 depicts a typical Dodger display screen. The Dodger display keeps the robot centered on the screen, and shows the sensor as it sweeps the plane. The circle surrounding the Dodger is merely for reference purposes. It shows the **safety radius** being used in the current simulation. The meaning of this value is discussed in detail in Chapter 3, but in essence, it shows the area in which Dodger must search for a safe haven with respect to the obstacles it has sensed. The safety radius is constructed to guarantee that such a safe haven always will exist inside of it. The grid of squares on the display serves no algorithmic purpose. It is merely a visual aid that allows the viewer to easily gauge the motion of the robot and the obstacles.

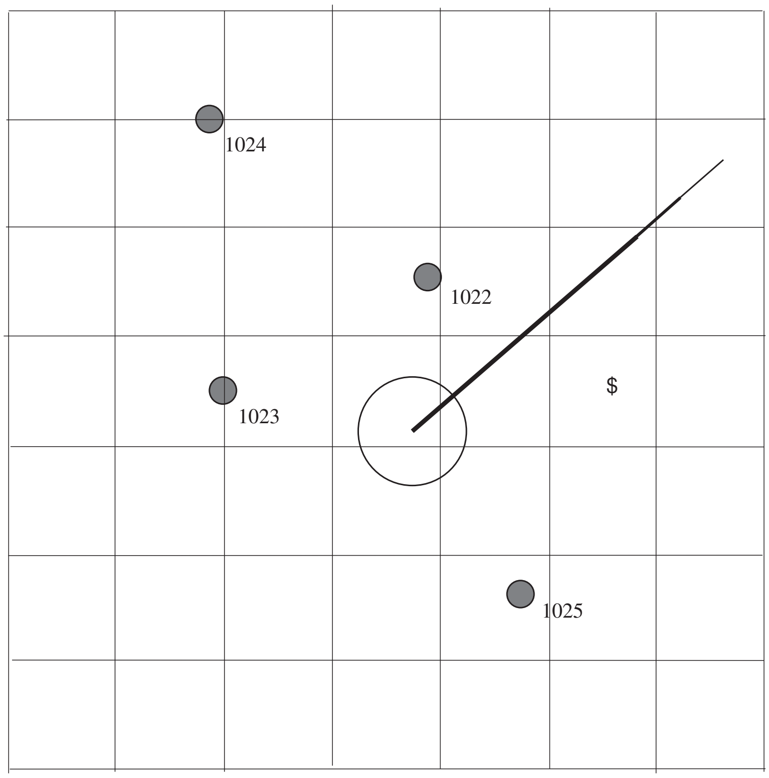


Figure 6.1: The Dodger Simulation Display

The Dodger environment is scaled, so that the sensor is always a fixed size on the display. Obstacles are modeled as circles of uniform width, and are assigned unique identification numbers on the display. They always appear on the boundaries of the display, but they are not sensed until they are "touched" by the sensor sweep. Dodger has been tested on a wide variety of obstacle generation strategies, which will be described below. Goals are generated pseudo-randomly, and appear one at a time, as each previous goal is "picked up." They are represented as a "$" on the display screen. As a goal location is reached, a new one is generated, and Dodger is notified via a "goal sensor." New goals are constrained to appear on the display screen, but otherwise, there is no strategy addressing how they are produced.

6.1.1 Obstacle Generation

Dodger is designed to avoid obstacles regardless of how they are arranged on the plane. To test this, a number of obstacle generation strategies have been implemented. Initially, new obstacles were created randomly, i.e.,

> Obstacles are generated at random start locations at the edge of the screen, and their trajectory is towards some random point on a different edge.

This is complete, in the sense that eventually, it would generate all possible combinations of obstacles on the plane (modulo the precision of the random number generator), but in practice, it very rarely results in scenarios that are difficult for Dodger. A second random strategy performed slightly better, since it constrained all obstacles to move either the entire height or the entire width of the display, i.e.,

> Obstacles are generated at random start locations at the edge of the screen, and their trajectory is towards some random point on the opposite edge.

This strategy was also quickly abandoned in favor of a suite of strategies that generate obstacles in ways that have a much higher density of difficult, threatening situations. Dodger has been developed and tested with the following obstacle generation strategies:

1. Obstacles are generated at random start locations at the edge of the screen. Their trajectory is then towards either a) the robot's current location or b) the goal's current location. The choice between these two is also random.
2. Obstacles start at a fixed "battery" position, and the trajectories are determined as in case 1.
3. Obstacles start at a fixed "battery position", and their trajectories are towards the robot's current goal, plus or minus a certain amount of random noise.
4. Obstacles start at random locations on the edge of the display, and their trajectories are towards the robot's current goal, plus or minus a certain amount of random noise.
5. Obstacles are generated as in case 4 with a probability of 2/3. In the rest of the cases, the starting locations are selected so the obstacle and the robot are headed directly towards each other, and the goal is between them.
6. Obstacles are generated in groups, traveling along parallel trajectories, such that the center of the group and the robot are headed directly towards each other, and the goal is between the robot and the obstacles.
7. Obstacles start at the screen edge position closest to the goal, and travel on a course that will intercept the robot if it travels at its maximum velocity on a straight-line path towards the goal.
8. Obstacles start at a fixed, "battery" position, and travel on a course that will intercept the robot if it travels at its maximum velocity on a straight-line path towards the goal.

The find-path algorithm does not know which of these strategies is being employed, and it does not attempt to determine what it is. Chapter 7 presents a summary of how Dodger performs with respect to the eight different goal generation techniques enumerated above.

Dodger allows the goal generation method to be specified at compile time. It would be simple to do this at startup time (e.g., via command line arguments), except that Maruti does not currently have any way to accept such parameters. Maruti was designed to support embedded, standalone applications, and does not currently provide a conventional, time-sharing interface. The application to be run is identified at boot time, at which point Maruti begins to execute it. While it might be convenient to allow runtime parameterization, it is not necessary.

Other parameters that can be varied at compile time include: the number of obstacles, the size of obstacles, the speed of obstacles, the maximum speed of the robot, the sensor sweep rate (i.e., the time it take the sensor to complete one full revolution), and the heuristic that determines where Dodger should go if it can't find a straight-line trajectory to its goal.

6.2 Dynamic Reaction

The concept of Dynamic Reaction (DR) was introduced in (71). DR was designed to solve problems in dynamic worlds by separating the need to respond appropriately to frequent and potentially threatening change in the world from the rest of the planning process. The DR model is founded on the concept of monitors: a set of them can be engineered to react to unexpected events, while a conventional search-based planner can be used to established long and intermediate-term goals. Sanborn and Hendler showed that it was possible to structure a set of monitors so that the global behavior of the system was "intelligent," in the sense that it was possible to pursue long-term goals while avoiding short-term, dynamic hazards in the environment.

Sanborn and Hendler distinguished between the problem of satisfying longer-term goals of achievement (e.g.,"get something to eat" or "get across the street"), that can be usefully addressed by classical planning techniques, and shorter-term goals of avoidance (e.g., "don't get hit by a car") that are typically associated with maintaining safety in a rapidly changing world. In order to accomplish these short-term goals, they proposed the use of a set of periodic processes, referred to as "monitors," that perform local computations in a bounded, and typically very short, amount of time.

As originally envisioned, Dodger was going to be an implementation of this architecture. However, Maruti does not yet support non-realtime, "offline" planning, so plan generation has been embedded in reactive, real-time heuristics, which have proven themselves adequate for this application. So, Dodger uses monitors both to guarantee obstacle avoidance and to direct it towards lower-priority goals of achievement. The architecture of the Dodger system is shown in Figure 6.2. Each of the ovals represents a periodic process that must be scheduled in the Maruti calendar (see Section 6.3). In the current implementation, the environmental simulator and the robot execution module are combined into a single software process, that is also periodically scheduled. However, these are conceptually physical entities. The arrows represent the flow of information between processes. It is important to note that Dodger does not give the planning algorithm complete and immediate access to the simulated world. There is a delay between the time an event happens in the world, and the time that it is sensed, as well as between the time that it is sensed, and the time that the planner has access to this information. In this implementation, Dodger behaves as if the robot responds directly to the control of the path-generating planning module. There is no message delay. Rather, robot behaviors are modified directly and immediately via function calls.

6.3 Maruti

The Maruti Operating System (70) supports hard real-time applications, while providing fault tolerant operation. Maruti provides guaranteed-service scheduling, in which jobs that are accepted by the system are guaranteed to meet the constraints of the computational requests. Viewed from the bottom-up, a Maruti program is organized around a set of periodic functions, known as **entry points**, and a set of message-invoked functions known as **services**. Programs are written

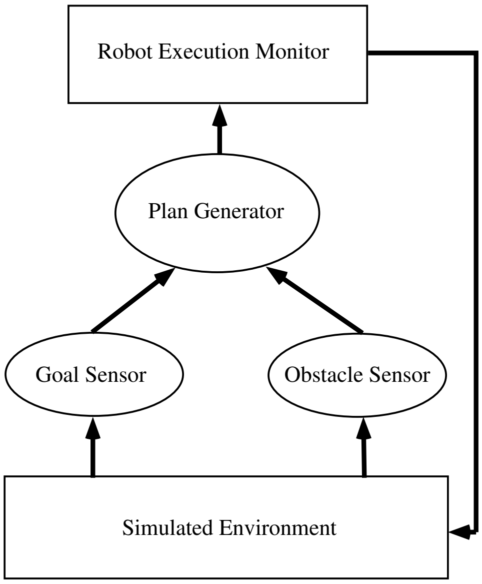


Figure 6.2: The Dodger System Architecture

in an extension of ANSI C, known as MPL. Given the worst-case running times of entry points and services, and a specification of the frequency that each entry point should run, the Maruti scheduler constructs a cyclic schedule, or **calendar**, that guarantees that the entry point function, and every service it invokes either directly or indirectly, will be able to run to completion within the period of the invoking entry point function.

The Maruti Operating System supports guaranteed, periodic scheduling capabilities that allow Dodger to guarantee obstacle avoidance. Maruti ensures that all computations will complete within preestablished periods. Thus, Dodger is able to establish the worst-case delay between the time an obstacle is sensed and the time the reactive response to the new threat is computed. This, is turn, allows a corresponding extension of the threat horizon, so that the existence of a computable, safety-preserving path can be guaranteed.

Whenever a new obstacle appears, Dodger must determine whether or not it poses a threat, and if so, how to avoid it. It must do so in time to allow the actual execution of the path it computes. The time to sense, compute and execute must therefore be bounded, and we must be able to guarantee that, whenever a new obstacle appears, Dodger can begin to execute a safety-preserving path in response, before the obstacle enters the threat horizon. In particular, the delay between the time an obstacle appears within the range of the sensor, is sensed, becomes known to the planner, and a reaction is computed must be both strictly bounded and known to Dodger. Maruti provides the system level support required to ensure that this is possible.

The close correspondence between the DR model and the Maruti software paradigm allows a straightforward implementation of monitors. There are three entry points in Dodger, corresponding to 3 sets of periodic processes, or more correctly "computation graphs" that are scheduled by Maruti. The environmental simulator/robot executive comprises one software module, intended to represent the physical environment. The robot find-path algorithm happens to be in this module, so that it can have direct access to the robot effectors, without having to communicate with them via message passing. Currently, the path-planning system has perfect, instantaneous knowledge of the location of the robot, but only uses information about the goal and the various obstacles after it is notified by its sensors. To ensure safety, the threat horizon must be extended to account for the lag between the time an obstacle is sensed and the time a reaction to this information is determined.

In the three obstacle case upon which most of the empirical tests were run, the simulated environment is updated every 50 milliseconds. The obstacle sensors receive updates about the location of the obstacles via a message that is processed by a service in the obstacle sensor module. Goal messages are only sent when the status of a goal changes (i.e. a goal is achieved, or "picked up", or a new goal is created.) Despite the fact that goal messages are delivered relatively rarely, Maruti schedules time for the service that receives the update message to run every 100 millisecond period. The Dodger sensors (obstacle and robot sensors) each run once in every 100 millisecond period. Thus every obstacle report will be received no later than 150 milliseconds after it is sent from the simulator. The path planner also runs in a 100 millisecond period, so in the worst-case, Dodger will have a response to a new threat within 250 milliseconds of the time it comes within range of the sensor. The "theoretical" threat horizon described in Section 3.1.2.1 (page 19) needs to be extended by the distance that an obstacle could travel in this amount of time. In addition, it is extended by the distance that an obstacle can travel in one sweep of the sensor, which is a system parameter that was set at one sweep every 1.12 seconds for most of the experiments described in Chapter 7.

6.4 Extensions

The current Dodger implementation is parameterized to allow for variations in the number of obstacles, the speed of obstacles, the maximum speed of the robot, the (uniform) width of obstacles, and the sweep rate of the obstacle sensor. Chapter 3 analyzes a number of other scenarios that fall within the same problem class. These are listed below, along with a brief discussion of what would be required to incorporate them into Dodger.

- The static Asteroids Problem should be considered a subproblem of the current implementation, which solves a version of it every time a new obstacle appears on the screen.

- The current system could handle variable width obstacles with only minor changes. The only difficulty lies in deciding if and how to modify the computation of the threat horizon. When some number of small obstacles are in the proximity of the robot, it should be possible to reduce the value of H to reflect the reduction in the maximum value of W, the width of the obstacle. If we assume some overall bound W on the size of any given obstacle, and generate obstacles randomly distributed below this bound, then the problems generated will be no more difficult than the problems currently being generated, modulo the question of adjusting the threat horizon. If we allow obstacles of any width, subject to the constraint that their widths sum to no more than some value $\mathbf{W}$, then there is no need to adjust the threat horizon, and the current program could solve the problem with only minor modifications.

- Allowing obstacles to have different, constant speeds, subject to some minimum and maximum values, would not require any substantive change to the current control system. However, as noted in the discussion of Corollary 2, on page 20, the threat horizon, H, increases with the ratio of the fastest allowable speed to the slowest allowable speed. Even when this ratio is 1 (i.e. when all obstacles travel with the same speed), the threat horizon grows with the square of the number of obstacles, so that the scaled obstacles become small points on the screen once there are more than 4 or 5 obstacles. Allowing obstacles to have different speeds would imply that their display size would be divided by the ratio of the maximum to the minimum obstacle velocity.

- The find-path algorithms would work if the obstacles were allowed to travel along piecewise linear paths, but safety could not be guaranteed, even if the piecewise linear trajectory could somehow be determined as soon as the obstacle was sensed. Chapter 3, shows how it is possible to guarantee safety if the ability of the obstacles to change its course is bounded.

In the case where there is no such bound, it is not possible to guarantee that the robot can avoid even a single obstacle, if that obstacle can travel faster than the robot.

- Assuming that we can bound the total area of (convex) static obstacles that could ever occupy the safety region (see Section 3.2.1, page 22), the primary additions needed to solve the Asteroids Problem with static obstacles is a routine for solving the static find-path problem, and bounds upon both the path computation and execution times. The details of this are discussed in Section 3.2.2, beginning on page 23. In cases where the goal achievement heuristics Dodger currently employs are adequate, static obstacles could be added with no modifications to the current algorithm.
- Incorporating imperfect sensing cannot be done without major revisions to the current system.

6.5 Conclusion

Dodger is essentially a proof of concept. It demonstrates that the theoretical results of Chapter 3, and the algorithmic results of Chapter 4 are sufficient to address the primary issues of obstacle avoidance in real time. Chapter 7, presents empirical data that shows Dodger is extremely adept at obstacle avoidance, but in the final analysis, Dodger is a prototype. None of its algorithms have been proven correct, even if there are proofs of the main principles upon which its avoidance behaviors are based. Dodger operates in a simulated environment, and relies heavily upon assumptions that will rarely hold in the real world. Still, this work represents a significant step towards realtime obstacle avoidance. Prior to this report, the issue of timeliness was never addressed in work on obstacle avoidance. Despite its strong assumptions, the Asteroids Problem has been extensively studied for the same reasons that it is the focus of this report. Dodger demonstrates that the concepts are sound, and this report establishes a foundation for further research into realtime obstacle avoidance. Chapters 3 examines many of the issues associated with relaxing the constraints of the problem.

Thus the Dodger program, and this report as a whole, represent an important step towards deploying obstacle avoidance algorithms as part of embedded systems that will be situated in the real world. If avoidance is to be guaranteed, it is not enough to expect fast processors and good engineering to suffice to ensure timeliness. The running time of the algorithms must be accounted for in the control strategy that is used to avoid obstacles. Dodger demonstrates that, in exactly the subclass of the Asteroids Problem where it is possible to ensure safety will always exist, it is also possible to design and implement a reactive avoidance strategy that suffices to avoid obstacles in real time.

Chapter 7

Empirical Results

7.1 Non-Realtime Search

Chapter 5 (page 46) describes an algorithm for an exhaustive search for paths from a robot's position to its goal. In a dynamic problem domain, such a search must be able to run to completion quickly, if its use is going to be feasible. As part of this report research, a version of the search described in Chapter 5 has been implemented, and this section contains an empirical analysis of its performance.

The high-level search algorithm is described in Figure 5.1, on page 47. It is basically a breadth-first search through the space of direct motions between obstacles, where only direct motions from obstacle centerline to obstacle centerline [1] are considered. As explained in Chapter 5, centerline-to-centerline search greatly improves the branching factor of the search suggested by 69 (69), which searches through the space of all direct motions between vertices of polygonal obstacles. Thus, the algorithm described here has a worst-case branching factor of N, the number of obstacles, whereas that of (69) has one of V, the total number of obstacle vertices, which is never less than $3N$.

The search begins with a single interval on its search queue: the position of the robot at time t_0. In actual practice, the search would begin by determining whether there is any unobstructed, constant velocity path to the goal, at which point the search would terminate. For the purposes of this analysis, the goal check was skipped, and so the search was forced to exhaustively examine all of the possible safety-preserving paths to a goal.

The search then begins by determining the time intervals at which the robot can be at the centerlines of the N moving obstacles, using the procedure described in Section 5.2 (page 48). Each such interval must be pruned of its occluded segments, as described in Section 5.5 (page 52). The resulting, reachable segments are then added to the search queue.

After all of the straightline paths from the robot's goal location to the various reachable intervals of each of the N obstacles has been determined, the first ply of the search is finished. Again, the second ply would normally begin by checking each of the intervals in the search queue for a direct, unobstructed path to the goal, but this has been eliminated for the purposes of worst-case analysis. Note that, rather than being a single point, the entries in the queue are (in general) line segments, and thus reachability must be determined using the algorithm described in Section 5.3 (page 49), and occlusions eliminated via the algorithm of Section 5.8 (page 54).

In an actual application, a path planner would have to account for contact motions, as described in Section 5.10 (page 57), as well as path instantiation, as described in Section 5.11 (page 61), but neither of those procedures were implemented for the purposes of these experiments.

---

[1] Recall that an obstacle centerline is just the line traced by the center of the circular obstacle in time-space.

The randomly generated initial obstacles locations were uniformly distributed on a square region with the robot at its center, and their random trajectories were uniformly distributed in the range $-\pi$ to π. The independent variables were the ratio of the robot and the obstacle's speed, and the number of obstacles in the domain. The main dependent variable of this study was CPU time. Information was collected about the number of nodes processed in each ply. All of these experiments were run on Sun Ultra 1's.

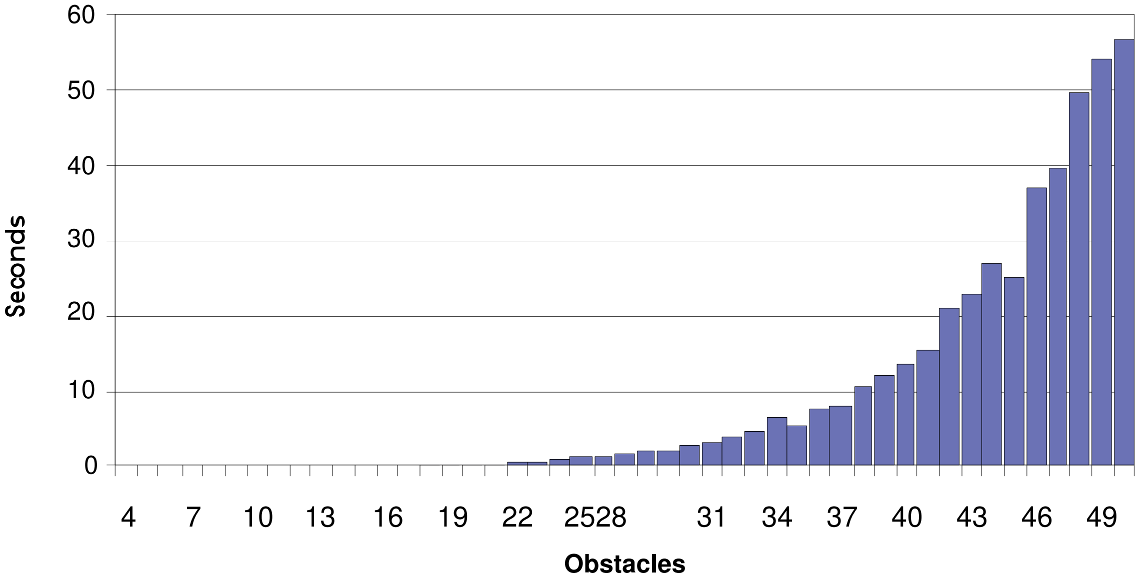


Figure 7.1: Average Search Time with a Robot/Obstacle Speed Ratio of 1/10

Figure 7.1 plots the runtime behavior of the search as a function of the number of randomly generated obstacles, when the robot/obstacle speed ratio is 1/10. Each data point represents the average of 100 test cases run using the number of obstacles indicated on the x-axis. The worst-case behavior for each number of obstacles is shown in Figure 7.2.

The main thing to note is that the average case performance is quite good for small number of obstacles. The average case for an exhaustive search does not exceed 10 CPU seconds until there are 36 obstacles in the domain, and even in the worst case, this threshold is not exceeded until 30 obstacles. And one should keep in mind that this is for the exhaustive case, where all possible sequences of centerline-to-centerline direct motions are considered. In application, searches would stop as soon as a path to the goal is found.

In addition to runtime performance as a function of the number of obstacles, the effects of increasing the robot/obstacle speed ratio were investigated. As Figures 7.3, 7.4, and 7.5 show, the worst-case search times get substantially worse as the robot's speed approaches that of the obstacles. This makes sense, since the robot has to consider a larger set of possibilities. More to the point, as the maximum speed of the robot becomes larger with respect to the obstacles, more and more centerline segments will become accessible from any given point.

Somewhat paradoxically, when the robot's speed exceeds that of all the obstacles', this search will evaluate all $N!$ possible sets of direct motion paths which visit each centerline only once. This is essentially because, regardless of the robot's location in the plane, it will always be able to eventually overcome any moving obstacle which it is faster than. The paradox is derived from the results of 31 (31), which show that in exactly those cases where the robot's speed exceeds that of all

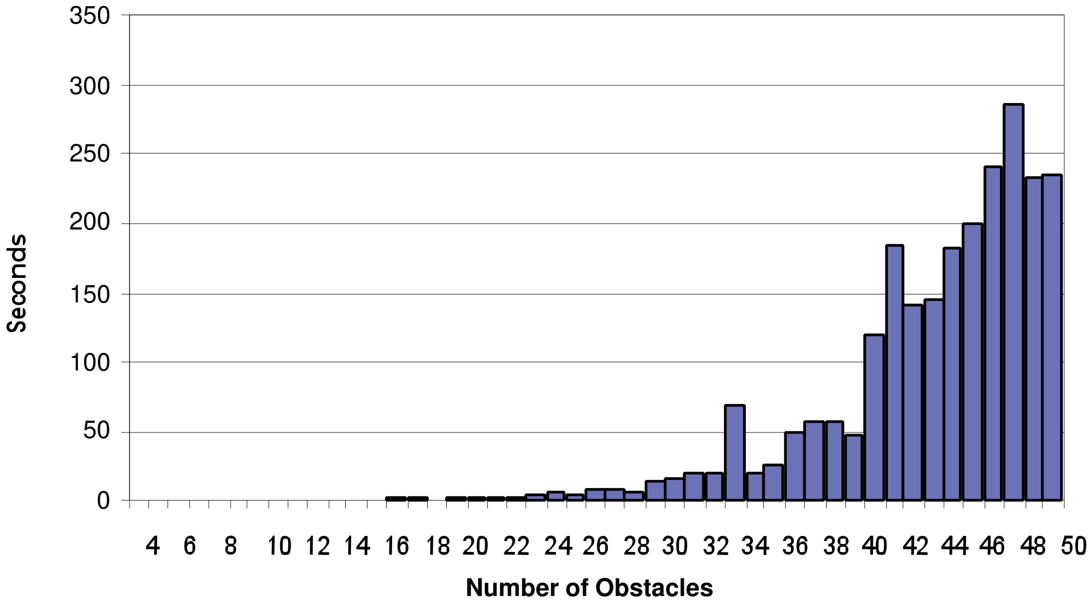


Figure 7.2: Worst-Case Search with Robot/Obstacle Speed Ratio of 1/10

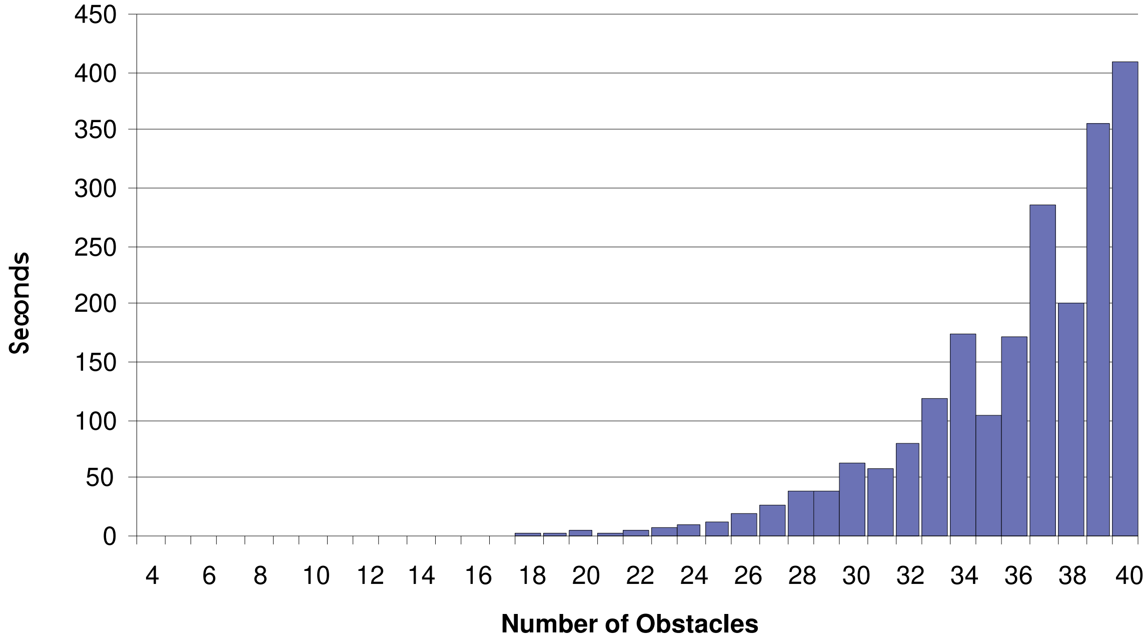


Figure 7.3: Search with an Robot/Obstacle Speed Ratio of 1/2

the obstacles, there exists a low-order polynomial time path-finding algorithm, which also happens to find the time-minimal motion from the robot to its goal.

All of the curves shown in Figures 7.3, 7.4, and 7.5 represent the results of 100 searches for each number of obstacles. The maximum number of obstacles (40 in Figures 7.3 and 7.4, 20 in Figure 7.5)

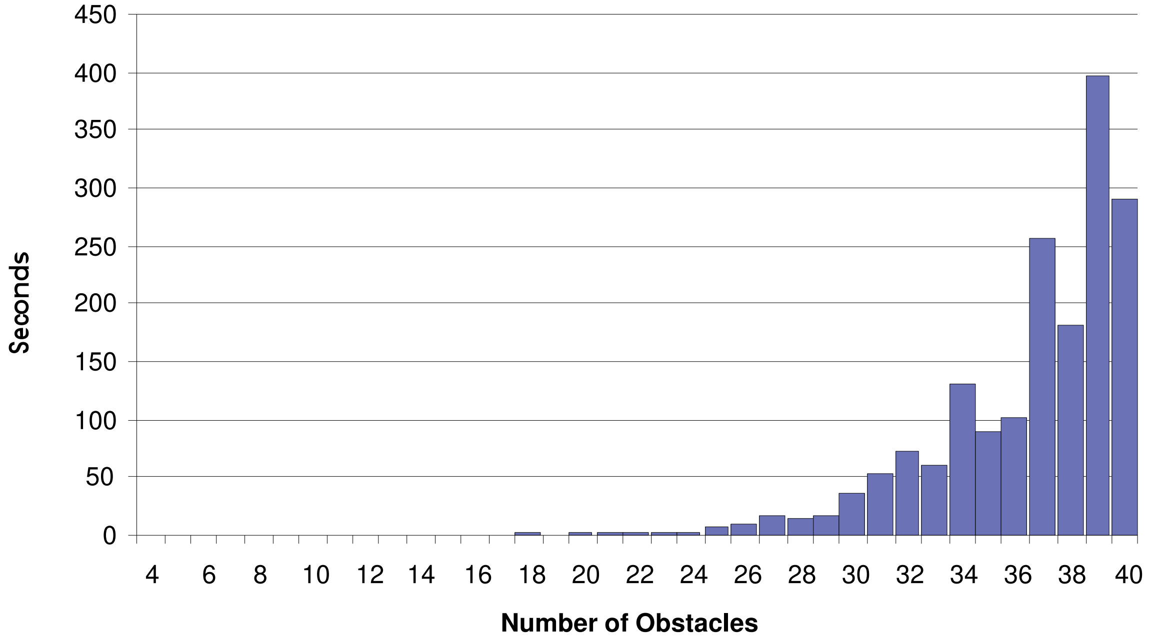


Figure 7.4: Search with an Robot/Obstacle Speed Ratio of 2/3

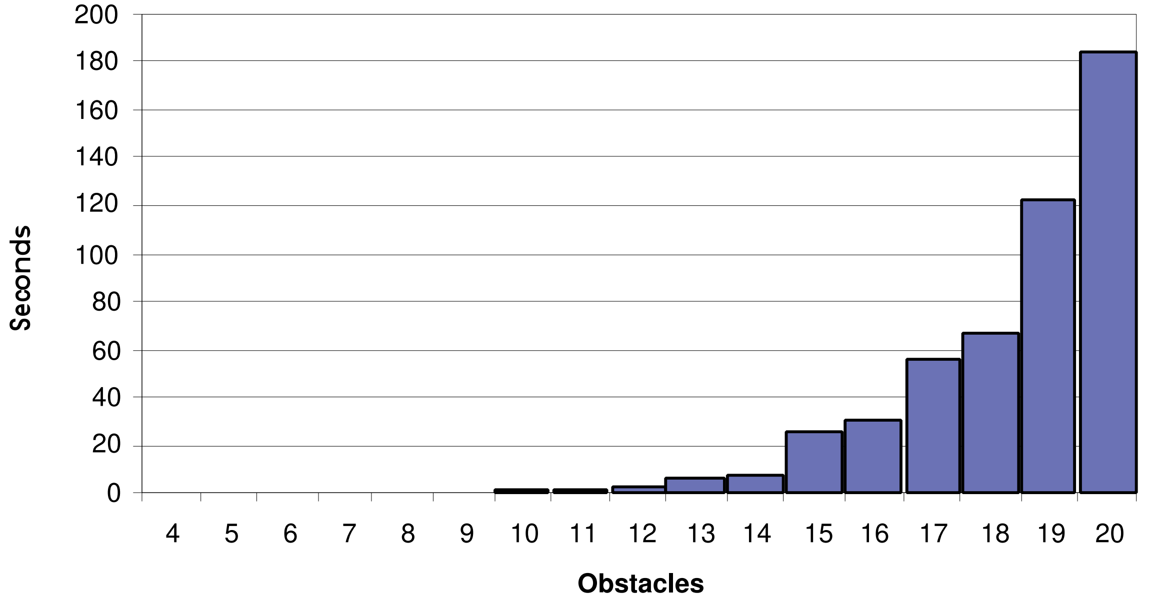


Figure 7.5: Search with an Robot/Obstacle Speed Ratio of 9/10

reflect the approximate level at which the searches began to exhaust virtual memory. Wherever possible, the search program uses space to save time, but as a result, memory was exhausted whenever there were 800K-1M nodes on the search queue, depending on the virtual memory of the machine and the maximum number of obstacles permitted. These levels were chosen to ensure that few, if any out-of-queue space events occurred. There was only one such failure, at the 40 obstacle level in the search summarized by Figure 7.4.

The variance of the runtimes increases dramatically as the number of obstacles increases. For example, in the 39 obstacle case of Figure 7.4, runtimes ranged from 10.6 to 5618 seconds, with an average of 396 seconds, and a standard deviation of 783 seconds. In the 40 obstacle case, the average was 290 seconds, values ranged from 18 to 1944 seconds, and the standard deviation was 356 seconds. A one-tailed t-test for the hypothesis that the 39 obstacle case is harder than the 40 obstacle case, where the variance of the two samples is not assumed to be the same, indicates that the difference is not significant ($t = 0.17$). The value of the average performance on 38 obstacles was 182 seconds. A similar t-test on using the values from the 39 and 40 obstacles cases, showed that values of 396 seconds could be measured in roughly 6.5% of all 100 experiment trials, even when the mean was 225.

7.2 Heuristics and Obstacle Generation

Section 4.4 (page 43) describes the three different heuristic strategies that Dodger employs to make progress towards its goals. Section 6.1.1 (page 66) describes the eight different obstacle generation strategies that were employed to test Dodger's avoidance abilities. To test the relative merits of the different heuristic strategies, each heuristic was tested for roughly a day in each of the obstacle generation scenarios. All tests were run on a 66MHz PC, running the Maruti HRT Operating System. Figure 7.6 shows the average number of seconds it took for the robot to reach a goal in each of these scenarios. This summarizes over 2.5 million seconds of testing, in which over two hundred thousand goals were collected by the Dodger. The obstacle/robot speed ratio used in this tests was 1.5, and the find-path algorithm was run once in every 100 millisecond period.

Obstacle Generation	Heuristic 1	Heuristic 2	Heuristic 3	Avg
1	12.0124	11.7227	13.0430	12.2594
2	12.3990	11.6364	11.7241	11.9198
3	11.6727	11.6912	11.6377	11.6672
4	11.7315	11.7164	11.6863	11.7114
5	13.9108	13.1632	11.7187	12.9309
6	13.7546	14.8327	12.3023	13.6299
7	13.0014	13.0249	12.0940	12.7067
8	12.9875	13.0430	13.0135	13.0146
Avg	12.6837	12.6038	12.1524	12.4800

Figure 7.6: Seconds per Goal

This data shows Heuristic 3 to be slightly better than the other two heuristics in the average case. The variance of the rate of goal achievement has not been empirically established, but the large sample size ensures that most of these differences are meaningful. For example, assume that the standard deviation of the rate of goal achievement is 6 seconds. The total number of goals generated for heuristics 2 and 3 in this experiment was 137,120. By the central limit theorem, the variance of the sample, $\sigma_{\bar{x}}$, is then $6^2/\sqrt{137210}$, which is roughly 0.0972. The Z-statistic for the hypothesis that the mean rate of goal achievement is equal for Heuristics 2 and 3 is then $(12.6038 - 12.1523)/0.0972 = 4.64$, which implies that the two means are different at a confidence level better than .00001. Even if heuristic 3 is significantly better than the other two heuristics, it is not considerably different, i.e., the first heuristic is roughly 4% worse than the third.

It appears that obstacle generation strategy 6 causes the greatest delays to the Dodger. In this strategy, obstacles begin on the edge location that is closest to the robot's location at the time that they are generated, and are targeted to intercept the robot as it moves at its maximum speed along a straight line path to its goal. It is important to note that, even though this strategy most impedes progress to the goal, it is relatively easy to avoid: the robot needs only to slow down. The generation strategies were designed to test the Dodger under a variety of conditions, and the relative abilities of the various strategies to retard progress to the goal should not be considered an indication of the difficulty Dodger may have avoiding obstacles generated under these regimens. In fact, during debugging, even though it is one of the least obstructive, obstacle strategy 3 tended to generate failures more frequently than the others. Naturally, this was somewhat due to the nature of the code errors, but it nonetheless illustrates the point that this data should not be over interpreted.

The total duration of these 24 trials was 2,501,129 seconds, or approximately 28.95 days, during which time the robot was not hit by a single one out of a total of 481,904 obstacles. The worst-case execution time of the find-path algorithm over this period was 39.0 milliseconds, and the average case was 1.92 milliseconds. The worst-case processing, which includes screen I/O, and environmental simulation, in addition to the find-path times, was 88.98 milliseconds, and the average case was 41.65 milliseconds.

7.3 Worst-Case Execution Times

It is difficult to characterize the worst-case arrangement of obstacles, in the sense that they will force the Dodger find-path algorithm to do the most possible processing to decide where to go next. The problem is not algorithmic, but rather geometric. The worst cases can be generally characterized as those in which large numbers of obstacles occlude parts of many other obstacles trajectories. It is important to ensure that enough time has been allowed in the schedule for the algorithm to **always** complete within its allocated quantum.

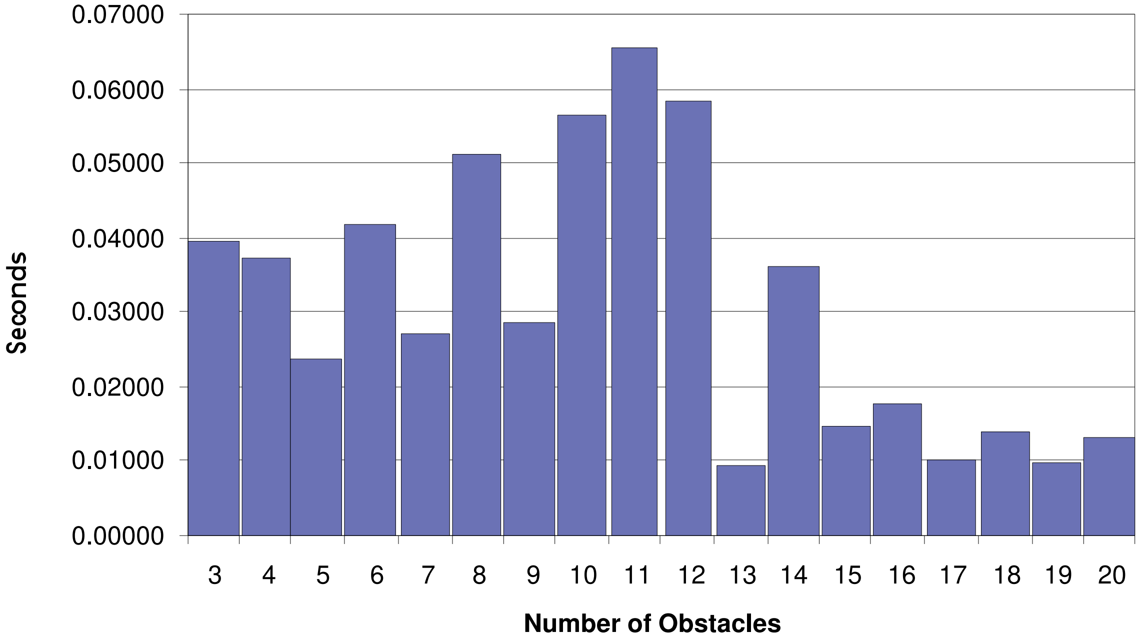


Figure 7.7: Worst-Case Performance

To test this hypothesis, a set of 18 tests were run, varying the number of obstacles from 3 to 20.

Obstacles were generated using generation strategy 3 (obstacles start at a fixed "battery position", and their trajectories are towards the robot's current goal, plus or minus some random noise), and the robot used Heuristic 3. Again, the obstacle/robot speed ratio was 1.5. The worst-case running time of the find-path algorithm is plotted versus the number of obstacles in Figure 7.7.

These 18 experiments totaled 1,671,319 seconds of runtime (19.34 days), 132,418 goals and 131,220 obstacles. During these trials, the Dodger was not hit once. In order to accommodate the increased overhead of simulating and displaying larger numbers of obstacles, the frequency of the find-path computation was decreased from once per 100 milliseconds to once per 150 milliseconds.

The variance in the plot suggest that nineteen days was not enough runtime to produce a smooth curve, but the trend is nonetheless clear. The observed worst-case performance peaked at 65.48 milliseconds in the 11 obstacle case, and quickly dropped off thereafter. This behavior is consistent with our expectation. The threat horizon in this experiment grows quadratically in the number of obstacles. As speculated in Chapter 3, this is more quickly than is strictly required. Section 7.4 offers empirical evidence that, even for the strategy employed by Dodger, which has not been show to be optimal, the threat horizon could be substantially reduced without compromising safety. As a consequence, Dodger has far more time than it need to detect and avoid trouble in the many-obstacle cases.

We should not conclude, on the basis of this graph, that the worst-case behavior can be bounded by some constant value. Consider Figure 7.8, which plots the average runtime of the find-path algorithm. This curve is quite clearly linear, with a least-squares fit of $0.000266537 + 0.000153479x$.

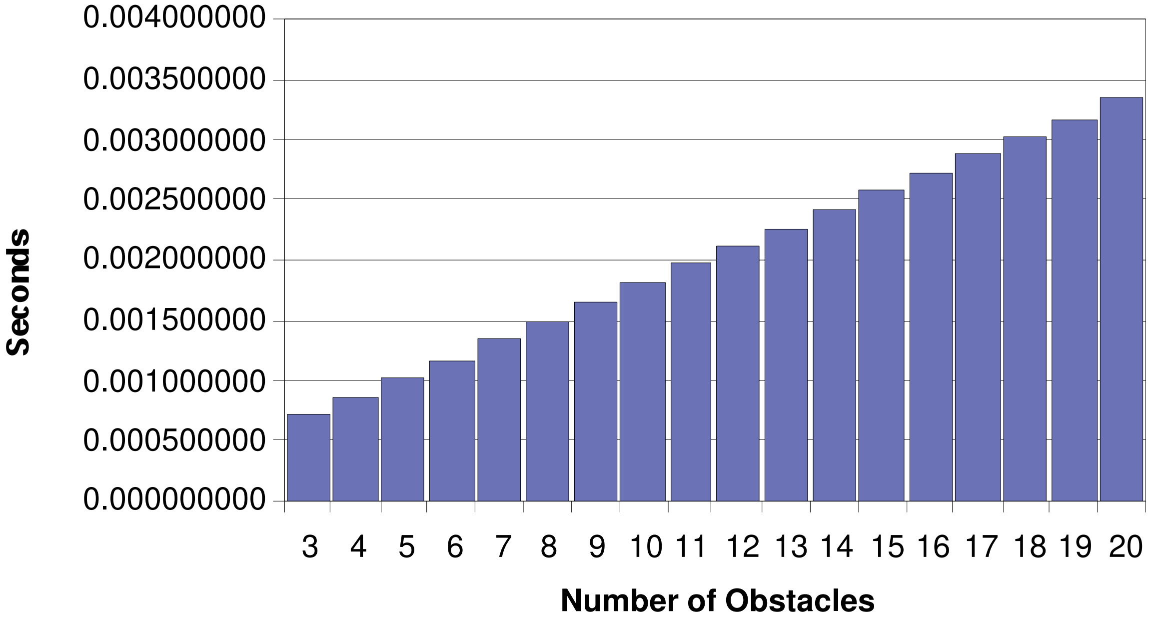


Figure 7.8: Average Case Performance

Thus the average case performance increases by approximately 1530 microseconds per obstacle. In the worst-case numbers plotted in Figure 7.7, 1530 microseconds is roughly 15% of the smallest observed worst-case behavior. However, the shape of the two curves suggests that eventually the baseline, average case performance will begin to dominate the total. Projecting from the least-squares fit, the average case performance will exceed the observed worst-case performance of 65.48 milliseconds after 425 obstacles.

7.4 Empirical Lower Bounds on the Threat Horizon

Section 3.1.3 (page 21) discusses the fact that the threat horizon for the dynamic Asteroids problem, which was derived in Section 3.1.2 (page 19), is only an upper bound on the horizon. The threat horizon for the static case is clearly a lower bound, but it is $O(N)$, where N is the number of obstacles, while the upper bound is $O(N^2)$. Establishing tight lower bounds is often difficult, and this case has proven no exception. In order to establish the minimum necessary threat horizon, one must somehow reason over all possible obstacle configurations **and** all possible avoidance strategies. It turned out that this was possible in the static version of the problem, but I have been unable to do so for the dynamic case. As mentioned in Section 3.1.3, my intuition is that the threat horizon is asymptotically better than $O(N^2)$. This section presents empirical evidence for that conjecture.

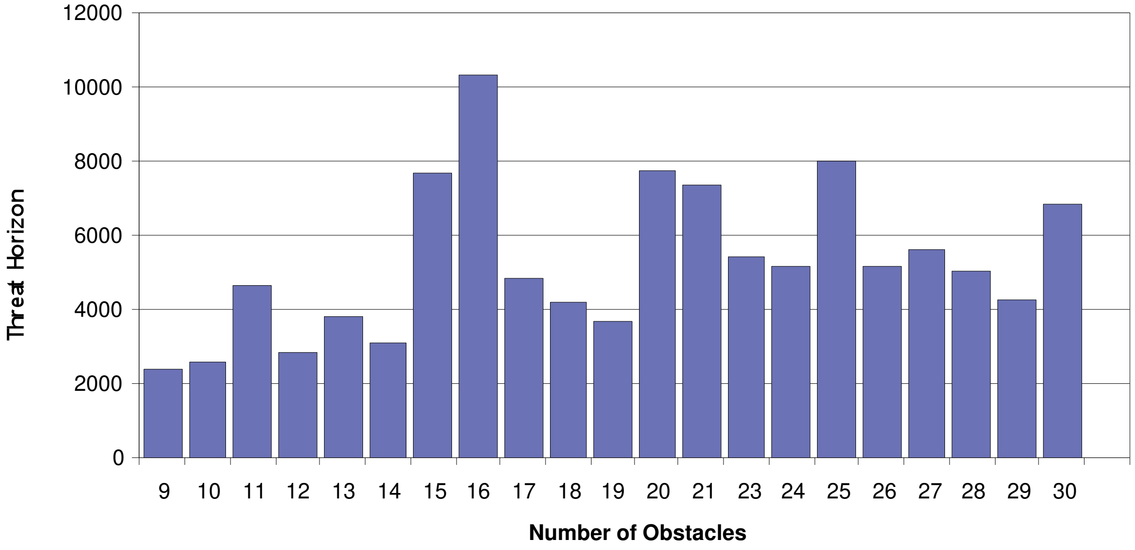


Figure 7.9: Empirical Threat Horizons

The basic idea of this experiment was to reduce the threat horizon to the point where the robot starts to get hit, and to thus establish a rough idea of where the threat horizon lies. There are two problems with this approach: first, Dodger employs a well-defined avoidance strategy, but the threat horizon is defined over **all** possible strategies. Second, just because the Dodger runs without collision for a fixed number of seconds, N, does not imply that it would not be hit at some later time. As N increases, our confidence grows, but in the absence of a mathematically grounded proof, we can never be one hundred percent certain that a threat horizon empirically determined in this way will suffice in all situations. Thus the point of this investigation is not to empirically establish a threat horizon, so much as it is to get an idea of how it grows with the number of obstacles.

The experimental design was as follows: obstacle generation strategy 1 was used, and the Dodger employer goal-achievement heuristic 3. For each number of obstacles from 9 to 30, the threat horizon was reduced, until the difference between the threat horizon that sufficed to allow the Dodger to avoid obstacles for 2 or more days, and a threat horizon at which the Dodger was observed to be hit, was with 1% of the theoretical horizon established in Section 3.1.2. The diameter of all obstacles was 80 units. The graph of Figure 7.9 plots the number of obstacles vs. the empirically determined threat horizon, in units of width. The scatter of this curve is a clear indication that a two day threshold is not sufficiently long to produce a smooth indication of the trend. Bear in mind, however, that each data point plotted in this figure represents roughly a week to 10 days of machine time. It took over 283 machine-days, not time lost to power outages and

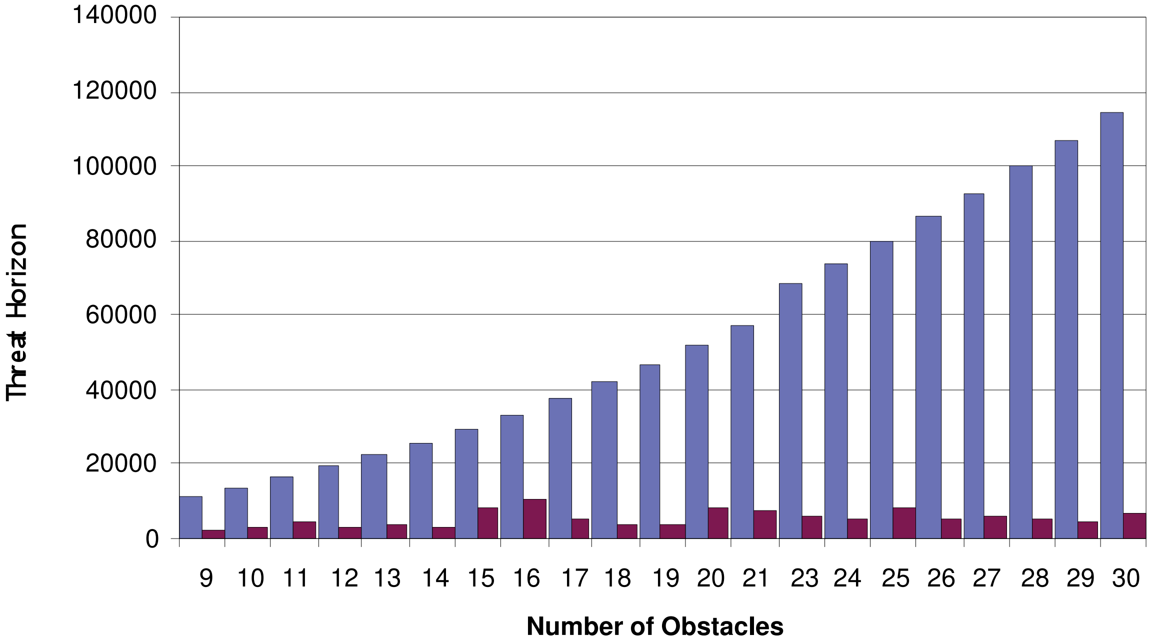


Figure 7.10: Empirical Threat Horizon and Theoretical Horizon

other minor disasters, to collect this data. Extending the required period of collision-free running time to a week would have taken several machine-years. Despite the scatter, there does appear to be an upward trend to the data. More importantly, consider Figure 7.10, which plots both the curve shown in Figure 7.9, and the theoretical horizon.

It is clear from this curve that the theoretical horizon greatly over estimated the actual horizon. Finally, consider Figure 7.11, which plots the ratio of the observed threat horizon to the theoretical lower bound, established for the static case in Section 3.1.1 (page 15). Among other things, this chart shows that, in over 9 machine-months of testing, no failures were observed for a threat horizon greater than 5 times the horizon required for the static case. Moreover, this graph does not appear to be trending upwards, and as a consequence, supports the hypothesis that the true theoretical lower bound is linear in the number of obstacles. Keep in mind that Dodger's avoidance strategy was based upon the proof of the theoretical bound. There is therefore every reason to believe that better strategies exist for the general problem.

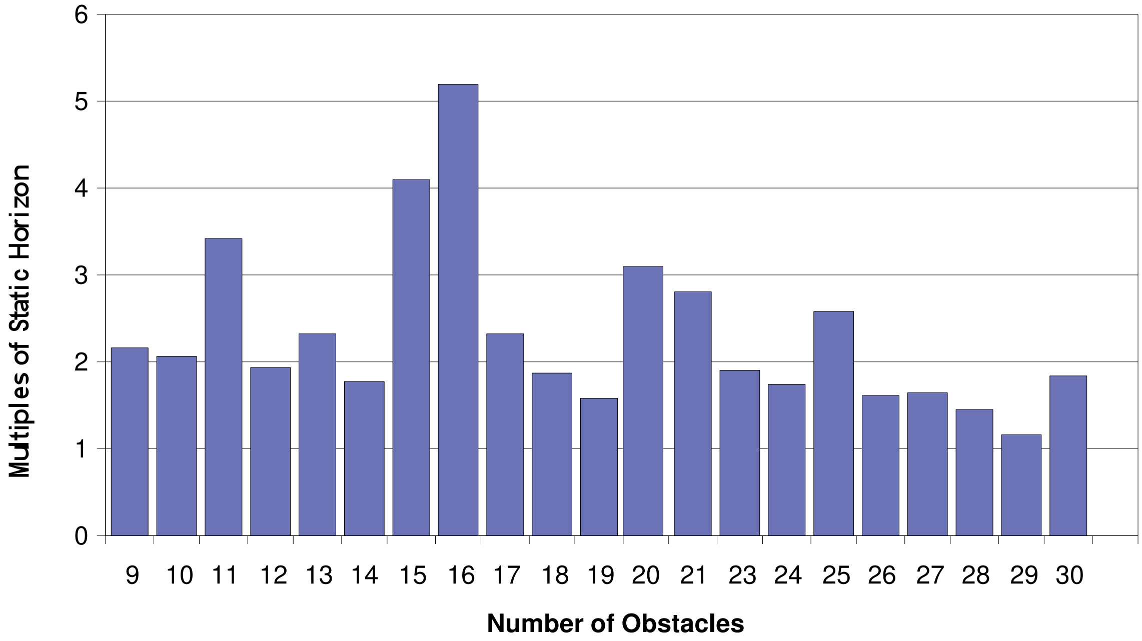


Figure 7.11: Observed Horizon/Theoretical Horizon for the Static Problem

Chapter 8

Conclusion

Over the last ten years, many AI Planning researchers have focused their efforts on so-called "reactive" systems, primarily in an effort to cope with the dynamic nature of real world planning problems. While not obviating the need for conventional, search-based planners, many researchers have demonstrated that reactive architectures are capable of generating robust behavior in dynamic, unpredictable environments. Moreover, such systems have several other advantages over classical planners: they generally do not require strong world models, and thus can function in the presence of uncertainty and incomplete knowledge of the problem domain; they do not make unrealistic static world assumptions to simplify a search space; and perhaps most importantly, they are capable of making timely responses in a changing world.

Timeliness is the essence of reaction, and it is clear that traditional, state-space search paradigms are ill-suited to addressing the needs of agents responding to a rapidly changing environment. Exhaustive search of state-space is fundamentally time-bound, and systems that need to respond quickly to changes in the world do not have the time to perform deep and/or complex searches for appropriate responses. Thus, research into reactive systems has been focused upon creating mechanisms that will enable an agent to make correct **and** timely responses in a changing world.

One approach has been to essentially "precompile" traditional deliberative systems, in an effort to calculate the appropriate response to a broad set of environmental conditions ahead of time (72; 33; 41). Such systems are founded upon the presumed correctness of conventional planners, and focus upon extending this paradigm to achieve timeliness. The advantages of this approach include its exhaustiveness and its generality, but it suffers from all the problems attendant upon the proper construction of adequate world models, including the need to control the combinatorial explosion of possible world states.

Others have chosen to abandon strong models of the environment. Several researchers (e.g. (1; 71; 26)) have demonstrated that the regularities of the world can be exploited to allow a relatively small set of behaviors to produce desirable global effects. Brooks 16 (16) has taken the extreme position that the world models of traditional systems are unnecessary, and that intelligent behavior is possible in systems that minimize symbolic representation of the external world. Without exception, these systems rely upon the intuitions and experiences of an engineer to discover and exploit the relevant regularities in the environment. This eliminates the need for world models, but at the expense of **demonstrable** correctness. It is by no means clear in which environments these systems might adequately function, beyond those in which they have been developed and tested. In addition, this approach sacrifices generality. Since they rely fundamentally upon their developers for their proper functioning in the world, these **ad hoc** systems offer little insight into how other reactive systems can be built.

A third approach has been to construct "hybrid" systems, comprised of separate reactive and deliberative components. While such systems have the potential for combining the timeliness of purely reactive behaviors with the correctness achievable through state-space search, current implementations suffer from all the weaknesses attributed to **ad hoc** systems above, as well as many of those of conventional planners. In particular, correctness is determined empirically, the "intelligence" of the reactive component is engineered, and the deliberative planner requires an adequate world model.

None of these approaches has directly addressed the issue of time. Almost no research has been done to **ensure** that reactions occur in a timely fashion, or even to establish the conditions under which appropriate responses can be made. With the exception of 62 (62), and (36) most claims of real-time performance are based upon either empirical testing or the restricted nature of the computations allowed in the computing model.[1] This may be sufficient for systems in which there is little or no penalty for delay, but in mission-critical systems, where timely response is critical, bounded response time alone does not imply adequate timeliness. In such applications, which range from airplane guidance and control systems, to hospital care and monitoring systems, to nuclear power plant control, it is not sufficient that response times be "fast." When the cost of failure may be catastrophic, it is essential that we can **prove** that response time is "fast enough." Since the essence of reaction lies in its timeliness, a fundamental component of the design and analysis of mission-critical reactive systems must be their temporal adequacy.

8.1 Accomplishments

This report describes the first research to ever address the problem of guaranteeing obstacle avoidance in dynamic domains. It establishes the theoretical conditions under which it is possible to guarantee that a safety-preserving path exists in the static version of the Asteroids Avoidance problem, and then extends these results to the more realistic, but less studied dynamic version of the problem. These results are then used to construct a path planner with provably correct behavior, where correctness is defined both in terms of behavior **and** the timeliness of computation.

Prior to this report, all of the work on the Asteroids problem addressed the static problem of finding a single collision-free path. In other words, all past research assumes that the positions and trajectories of the robot, the single goal, and all of the obstacles are known at the start of the path-finding computation. Such conditions rarely exist in the world. More typically, the locations and trajectories of obstacles become known to the planning agent over time, and there may be a series of goal locations to get to in the course of achieving some other higher-level result. In the static case, it is possible to ignore the problems implicit in computing a solution soon enough to actually use it. In the more realistic, dynamic case, a correct implementation must account for the time used to compute a solution.

Section 3.1.1 (page 15) establishes necessary and sufficient conditions for guaranteeing safety in the static version of the Asteroids Problem. Section 3.1.2 (page 19) extends these results to establish sufficient conditions for guaranteeing safety in the dynamic version of the problem. However, these conditions are not necessary. As is typical with many algorithmic problems, a tight lower bound has proven difficult to establish. Doing so requires reasoning over all of the possible configurations of obstacles, **and** all of the possible avoidance algorithms. The tight bound of Section 3.1.1 does this, but there are few techniques for doing this generally. This much is certain: the algorithm and conditions sufficient to ensure safety in the static problem are **not** the same as those applicable in the dynamic case. This challenges the assumption, implicit in all previous work on the problem, that a solution to the static case will somehow prove adequate for the more realistic dynamic version.

---

[1] For example 41 (41) points to the bounded depth of the GAPPS circuits, and 15 (15) relies upon the nature of the hardware he employs to make claims of adequate "real-time" response.

This report also describes Dodger, a system designed and implemented to solve the dynamic Asteroids Avoidance problem in hard real time. Dodger is a reactive system: it not only finds paths quickly, the path-planning algorithm is designed to take its own worst-case running time into account. Thus solutions are actually **guaranteed** to be useful once they are computed. The experiments described in Section 7.2 and Section 7.3 document over 47 days in which the Dodger was not hit by a single one of over 600,000 obstacles. Including tests run to determine the worst-case running times, and to establish the correctness of its performance, Dodger has successfully avoided all obstacles in over 7 machine-weeks of testing.

The Dodger algorithm is fast, taking no more than 68 milliseconds on a 66 MHz PC, in the observed worst-case, and far less time in the average case. With relatively minor modifications, it can be employed to heuristically solve a number of similar problems in obstacle avoidance, including cases where the paths of the obstacles are not perfectly known, and/or move along piecewise linear trajectories. "Heuristically", means that, in these cases, the current algorithm cannot guarantee avoidance. Given the running time of the algorithm, and the fact that no other known algorithm guarantees avoidance under such conditions, it should nonetheless be considered a viable candidate for a path-planning system, even when guarantees are not required.

Chapter 5 describes an exhaustive search that is guaranteed to find a safety-preserving path in the Asteroids problem, whenever such a solution exists. While based directly upon an algorithm suggested by 69 (69), the algorithm of Chapter 5 makes a number of performance-oriented modifications that were not of importance to its original authors. It should be considered a generalization of the "accessibility graphs" of 29 (29) in which **all** possible speeds of the robot are accounted for simultaneously. Accessibility graphs are of considerable utility in the atypical case where the robot's speed exceeds that of all moving obstacles, since 31 (31) have shown that, in such cases, a safety-preserving path exists **iff** a safety-preserving path at which the robot always travels its maximum velocity exists. However, in the normal case, deciding what the velocity of the robot should be is a fundamental and integral part of the problem.

The algorithm of Section 5.8 (page 54), which is required to determine the occluded (and hence unoccluded) parts of a line segment, with respect to a single obstacle, when the position of the robot is constrained to lie on a line segment, has general applicability to path planning, both with respect to its result, and to the technique used to achieve it. In real-world applications, the position of a robot is most typically known only imprecisely. If the location can be established to fall within some bounds, this algorithm, or some variation of it, provides a way to efficiently determine whether or not some set of locations is reachable via a constant velocity motion.

This research has also contributed indirectly to the future development of Maruti. Plans for integrating scheduled, hard real-time code with less predictable, non-realtime routines, such as the search algorithm of Chapter 5, are currently in place, and development is scheduled to begin soon.

8.2 Future Work

This research represents an important first step towards being able to guarantee the correct performance of agents situated in dynamic, spatial domains. As might be expected, it raises a number of questions, each of which is important and interesting in its own right. In my opinion, the most significant of these is the problem of determining a tight lower bound on the threat horizon necessary to ensure safety in the dynamic Asteroids problem, which is discussed in Section 3.1.3 (page 21), and examine empirically in Section 7.4 (page 78). Tightening this bound would greatly enhance the utility of this work, both with respect to the canonical Asteroids problems, as well as with respect to more realistic variants of the problem, such as those discussed in Section 3.1.2.2, Section 3.2, and Section 3.3.

The results of Section 7.1 suggest that integrating a search with Dodger's reactive avoidance capabilities may be feasible in many cases. Doing so would require some mechanism for integrating the plan with a pre-computed real-time schedule. This issue is briefly addressed in Section 5.11 (page 61). Basically, in its current form, Maruti computes a schedule which determines when every software component in the system will be run, on a periodic basis. If a find-path search does not take the constraints imposed by this schedule into account, it will not be possible to correctly execute the planned path. On the other hand, taking such information into account greatly complicates the search, and may result in an otherwise feasible path being rejected.

The search of Chapter 5, summarized in Figure 5.1 on page 47, checks each node for a constant velocity path to the goal before expanding it. Since such a path is guaranteed to exist for any safe haven, it would be interesting to see the effects of changing or expanding this to a check for a constant velocity path to the goal, followed by a search for paths to a single point in each of the distinct sets of safe harbors that can be computed as described in Section 4.1 (page 33).

It would also be interesting to deploy the Dodger algorithm on a working mobile robot, not so much to attempt to guarantee avoidance, as to determine whether or not it can be adapted for real-world application. As noted above, the performance characteristics of the algorithm suggest that, so long as the boundaries of moving obstacles can be established to a known level of precision, and their trajectories can be approximated as piecewise-linear paths, it should perform satisfactorily in domains such as parking lots. The real research would involve integrating this capability with an ability to search through a space of static obstacles.

References

[1] P. E. Agre and D. Chapman, "Pengi: An Implementation of a Theory of Activity," in *Proc. National Conf. on Artificial Intelligence*, pp. 268–272. Morgan Kaufmann, 1987.

[2] R. Arkin, "Motor Schema-Based Mobile Robot Navigation," *The International Journal of Robotics Research*, vol. 8, no. 4, pp. 92–112, August 1989.

[3] R. C. Arkin, "Integrating Behavioral, Perceptual, and World knowledge in Reactive Navigation," *Robotics and Autonomous Systems*, vol. 6, pp. 105–122, 1990.

[4] T. Asano, T. Asano, L. Guibas, J. Hershberger, and H. Imai, "Visibility of Disjoint Polygons," *Algorithmica*, vol. 1, no. 1, pp. 49–63, 1986.

[5] T. Bang, "A Solution of the "Plank Problem"," in *Proceedings of the American Mathematical Society*, volume 2, pp. 990–993, 1951.

[6] J. Barraquand and J.-C. Latombe, "A Monte-Carlo Algorithm for Path Planning with Many Degrees of Freedom," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1712–1717, 1990.

[7] S. Basu, R. Pollack, and M.-F. Roy, "Computing Roadmaps of Semi-Algebraic Sets," in *Proceedings of the 28th Annual ACM Symposium on the Theory of Computation*, pp. 168–173, 1996.

[8] B. G. Baumgart, "A polyhedron representation for computer vision," in *AFIPS National Computer Conference*, volume 44, pp. 589–596, 1975.

[9] J. L. Bentley and T. A. Ottmann, "Algorithms for Reporting and Counting Geometric Intersections," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 643–647, September 1979.

[10] J. Borenstein and Y. Koren, "Real-Time Obstacle Avoidance for Fast Mobile Robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1179–1187, 1989.

[11] J. Borenstein and Y. Koren, "The Vector Field Histogram – Fast Obstacle Avoidance for Mobile Robots," *IEEE Transactions on Robots and Automation*, vol. 7, no. 3, pp. 278–288, June 1991.

[12] R. Brooks and T. Lozano-Pérez, "A Subdivision Algorithm in Configuration Space for Findpath with Rotation," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 15, no. 2, pp. 224–233, 1985.

[13] R. A. Brooks, "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 13, no. 3, pp. 190–197, 1983.

[14] R. A. Brooks, "A Robust Layered Control System for a Mobile Robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, no. 1, pp. 14–22, March 1986.

[15] R. A. Brooks, "Elephants Don't Play Chess," in *Designing Autonomous Agents*, P. Maes, editor, MIT/Elsevier, 1990.

[16] R. A. Brooks, "Intelligence Without Representation," *Artificial Intelligence*, vol. 47, pp. 139–159, 1991.

[17] J. Canny and J. Reif, "New Lower Bound Techniques for Robot Motion Planning Problems," in *Proceedings of the 28th IEEE Symposium of Foundations of Computer Science*, pp. 49–60, 1987.

[18] J. F. Canny, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge,Mass., 1988.

[19] B. Chazelle and H. Edelsbrunner, "An Optimal Algorithm for Intersecting Line Segments in the Plane," *Journal of the Association for Computing Machinery*, vol. 39, no. 1, pp. 1 – 54, January 1992.

[20] R. A. Conn and M. Kam, "On the Moving-Obstacle Path-Planning Algorithms of Shih, Lee, and Gruver," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 27, no. 1, pp. 136–138, 1997.

[21] D. Coombs, M. Herman, T. Hong, and M. Nashman, "Real-Time Obstacle Avoidance Using Central Flow Divergence and Peripheral Flow," in *5th International Conference on Computer Vision*, 1995.

[22] F. De la Rosa, C. Laugier, and J. Nájera, "Robust Path Planning in the Plane," *IEEE Transactions on Robots and Automation*, vol. 12, no. 2, pp. 347–352, April 1996.

[23] E. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, 1959.

[24] M. Erdman and T. Lozano-Pérez, "On Multiple Moving Objects," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1419–1424, 1986.

[25] B. Faverjon and P. Tournassoud, "A Practical Approach to Motion Planning for Manioulators with Many Degrees of Freedom," *Preprints of the Fifth International Symposium of Robotics Research*, pp. 65–73, 1989.

[26] R. J. Firby, "An Investigation into Reactive Planning in Complex Domains," in *Proc. National Conf. on Artificial Intelligence*, pp. 202–206, 1987.

[27] D. Fox, W.Burgard, and S. Thrun, "The Dynamic Window Approach to Collision Avoidance," *IEEE Robotics and Automation Magazine*, vol. 4, no. 1, pp. 23–33, March 1997.

[28] T. Fraichard and C. Laugier, "Dynamic Trajectory Planning, Path-Velocity Decomposition and Adjacent Paths," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 1592–1597, 1993.

[29] K. Fujimura and H. Samet, "Accessibility: A New Approach to Path Planning among Moving Obstacles," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 803–807, 1988.

[30] K. Fujimura and H. Samet, "A Hierarchical Strategy for Path Planning Among Moving Obstacles," *IEEE Transactions on Robots and Automation*, vol. 5, no. 1, pp. 61–69, 1989.

[31] K. Fujimura and H. Samet, "Time-Minimal Paths Among Moving Obstacles," in *Proceedings of IEEE International Conference on Robotics an d Automation*, pp. 1110–1115, 1989.

[32] A. Garvey and V. Lesser, "A Survey of Research in Deliberative Real-Time Artificial Intelligence," *Real-Time Systems*, vol. 6, pp. 317–347, 1994.

[33] M. P. Georgeff and A. L. Lansky, "Reactive Reasoning and Planning," in *Proc. National Conf. on Artificial Intelligence*, pp. 677–682, July 1987.

[34] S. Ghosh and D. Mount, "An Output Sensitive Algorithm for Computing Visibility Graphs," in *Proceedings of the 28th IEEE Symposium of Foundations of Computer Science*, pp. 11–19, 1987.

[35] J. Gil de Lamadrid, "Avoidance of Obstacles with Unknown Trajectories: Locally Optimal Paths and Periodic Sensor Readings," *International Journal of Robotics Research*, vol. 13, no. 6, pp. 496–507, December 1994.

[36] L. Greenwald and T. Dean, "A Conditional Scheduling Approach to Designing Real-Time Systems," in *Artificial Intelligence Planning Systems*, June 1998.

[37] M. Herman, "Fast, Three-Dimensional, Collision-Free Motion Planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1056–1063, 1986.

[38] H. Hu and M. Brady, "A Bayesian Approach to Real-Time Obstacle Avoidance for a mobile Robot," *Autonomous Robots*, vol. 1, pp. 69–92, 1994.

[39] Y. K. Hwang and N. Ahuja, "Gross Motion Planning – A Survey," *ACM Computing Surveys*, vol. 24, no. 3, pp. 219–291, September 1992.

[40] M. Ianigro, T.D'Orazio, F.P.Lovergine, E. Stella, and A.Distante, "Real-Time Obstacle Avoidance Based on Sensory Information," *SPIE Volume 1831 Mobile Robots VII*, pp. 62–73, 1993.

[41] L. P. Kaelbling, "Goals as Parallel Program Specifications," in *Proc. National Conf. on Artificial Intelligence*, pp. 60–65, 1988.

[42] S. Kambhampati and L. Davis, "Multiresolution Path Planning for Mobile Robots," *IEEE Transactions on Robots and Automation*, vol. 2, no. 3, pp. 135–145, 1986.

[43] K. Kant and S. Zucker, "Towards Efficient Trajectory Planning: Path Velocity Decomposition," *International Journal of Robotics Research*, vol. 5, pp. 72–89, 1986.

[44] N. Kentarnavez and S. Li, "A Collision-Free Navigation Scheme in the Presence of Moving Obstacles," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 808–813, 1988.

[45] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 500–505, 1985.

[46] O. Khatib, "Real-Time Obstacle Avoidance for Manipulators and Mobile Robots," *International Journal of Robotics Research*, vol. 5, no. 1, pp. 90–98, 1986.

[47] D. Koditschek, "Exact Robot Navigation by Means of Potential Functions: Some Topological Considerations," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 14–24, 1987.

[48] R. C. Kohout, J. A. Hendler, and D. J. Musliner, "Guaranteeing Safety in Spatially Situated Agents," in *Proc. National Conf. on Artificial Intelligence*, pp. 909–914, August 1996.

[49] Y. Koren and J. Borenstein, "Potential Field Methods and Their Inherent Limitations for Mobile Robot Navigation," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1398–1404, 1991.

[50] B. Krogh, "A Generalized Potential Field Approach to Obstacle Avoidance Control," in *Robotics International Robotics Research Conference*, August 1984.

[51] B. Krogh and C. Thorpe, "Integrated Path Planning and Dynamic Sterring Control for Autonomous Vehicles," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1664–1669, 1986.

[52] D. Kuan, J. Zamiska, and R. Brooks, "Natural Decomposition of Free Space for Path Planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 168–173, 1985.

[53] K. Kyriakopoulos and G. N. Saridis, "Distance Estimation and collision Prediction for On-Line Robotic Motion Planning," *Center for Intelligent Robotic Systems for Space Exploration, Renselar Polytechnic Institute*, vol. NASA-CR-192737, no. 25, , January 1991.

[54] J.-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, Boston/Dordrecht/London, 1991.

[55] M. Lin and J. Canny, "A Fast Algorithm for Incremental Distance Calculations," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 1008–1014, 1990.

[56] T. Lozano-Pérez, "Automatic Planning of Manipulator Transfer Movements," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 11, no. 10, pp. 681–698, 1981.

[57] T. Lozano-Pérez, "Spatial Planning: A Configuration Space Approach," *IEEE Trans. Computers*, vol. C-32, no. 2, pp. 108–120, February 1983.

[58] T. Lozano-Pérez and M.. Wesley, "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Communications of the ACM*, vol. 22, no. 10, pp. 560–570, 1979.

[59] A. Maciejewski and J. Fox, "Path Planning and the Topology of Configuration Space," *IEEE Transactions on Robots and Automation*, vol. 9, no. 4, , August 1993.

[60] V. Manikonda, P. Krishnaprasad, and J. Hendler, "Languages, Behaviors, Hybrid Architectures and Motion Control," Technical Report ISR TR 97-59, Institute for Systems Research, University of Maryland, January 1997.

[61] D. J. Musliner, J. A. Hendler, A. K. Agrawala, E. H. Durfee, J. K. Strosnider, and C. J. Paul, "The Challenges of Real-Time AI," *IEEE Computer*, vol. 28, no. 1, , January 1995.

[62] D. J. Musliner, E. H. Durfee, and K. G. Shin, "CIRCA: A Cooperative Intelligent Real-Time Control Architecture," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 23, no. 6, pp. 1561–1574, 1993.

[63] W. Newman and N. Hogan, "High Speed Robot Control and Obstacle Avoidance Using Dynamic Potential Functions," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 14–24, 1987.

[64] W. S. Newman, "Real-Time Configuration Space Transforms for Obstacle Avoidance," *The International Journal of Robotics Research*, vol. 10, no. 6, pp. 650–666, 1991.

[65] N. J. Nilsson, "A Mobile Automaton: An Application of Artificial Intelligence Techniques," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 509–518, 1969.

[66] C. Ó'Dúnlaing, M. Sharir, and C. Yap, "Retraction: A New Approach to Motion Planning," in *Proceedings of the 15th ACM Symposium on the Theory of Computing*, pp. 207–220, 1983.

[67] J. O'Rourke, *Computational Geometry in C*, Cambridge University Press, 1994.

[68] D. W. Payton, J. K. Rosenblatt, and D. M. Keirsey, "Plan Guided Reaction," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 20, no. 6, , 1990.

[69] J. Reif and M. Sharir, "Motion Planning in the Presence of Moving Obstacles," in *Proceedings of the 25th IEEE Symposium of Foundations of Computer Science*, pp. 144–154, 1985.

[70] M. Saksena, J. da Silva, and A. Agrawala, "Design and Implementation of Maruti," Technical Report CS-TR-3181, University of Maryland Department of Computer Science, December 1993.

[71] J. C. Sanborn and J. A. Hendler, "A Model of Reaction for Planning in Dynamic Environments," *Int'l Journal for Artificial Intelligence in Engineering*, vol. 3, no. 2, pp. 95–102, April 1988.

[72] M. Schoppers, "Universal Plans for Reactive Robots in Unpredictable Environments," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 1039–1046, 1987.

[73] J. Schwartz and M. Sharir, "On the Piano Movers' Problem: I. The Case of a Two-Dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers," *Communications on Pure and Applied Mathematics*, vol. 36, pp. 345–398, 1983.

[74] J. Schwartz and M. Sharir, "On the Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds," *Advances in Applied Mathematics*, vol. 4, pp. 298–351, 1983.

[75] O. Schwarzkopf and M. Sharir, "Vertical Decomposition of a Single Cell in a Three-Dimensional Arrangement of Surfaces and its Applications," *Discrete Computational Geometry*, vol. 18, pp. 269–288, 1997.

[76] M. Sharir, "Algorithmic Motion Planning in Robotics," *IEEE Computer*, vol. 22, no. 3, pp. 9–19, March 1989.

[77] R. Sharma, Y. Aloimonos, and D. M. Mount, "Navigation in a hazardous environment with distributed shelters," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, pp. 883–898, Charlottesville, Virginia, 1991.

[78] C. Shih, T. Lee, and W. Gruver, "A Unified Approach for Robot Motion Planning Among Moving Polyhedral Obstacles," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 20, no. 4, pp. 903–915, July 1990.

[79] M. Slack and D. Miller, "Path Planning through Time and Space in Dynamic Domains," in *Proc. Int'l Joint Conf. on Artificial Intelligence*, pp. 1067–1070, 1987.

[80] S. Udupa, "Collision Detection and Avoidance in Computer Controlled Manipulators," in *Proc. National Conf. on Artificial Intelligence*, pp. 737–748, 1977.

[81] E. Wong and K. Fu, "A Hierarchical-Orthogonal-Space Approach to Collision-Free Path Planning," in *Proceedings of IEEE International Conference on Robotics and Automation*, pp. 506–511, 1985.

[82] C.-K. Yap, "Algorithm Motion Planning," in *Algorithmic and Geometric Aspects of Robotics*, J. T. Schwartz and C.-K. Yap, editors, pp. 95–143, Lawrence Erlbaum Associates, London, 1987.