Real-Time Motion Planning and Safe Navigation in Dynamic Multi-Robot Environments

James Robert Bruce

CMU-CS-06-181

December 15, 2006

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Manuela Veloso, Chair
Avrim Blum
James Kuffner
Tony Stentz
Lydia Kavraki (Rice University)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored in part by United States Grants Nos. DABT63-99-1-0013, F30602-98-2-0135, F30602-97-2-0250, and NBCH-1040007. It was also supported by Rockwell Scientific Co., LLC under subcontract No. B4U528968 and prime contract No. W911W6-04-C-0058 with the US Army. The views and conclusions contained herein are those of the author, and do not necessarily reflect the position or policy of the sponsoring institutions, and no official endorsement should be inferred.

Keywords: Robotics, Robot Navigation, Motion Planning, Path Planning, Multi-Robot Systems, Robotic Soccer

Abstract

All mobile robots share the need to navigate, creating the problem of motion planning. In multi-robot domains with agents acting in parallel, highly complex and unpredictable dynamics can arise. This leads to the need for navigation calculations to be carried out within tight time constraints, so that they can be applied before the dynamics of the environment make the calculated answer obsolete. At the same time, we want the robots to navigate robustly and operate safely without collisions. While motion planning has been used for high level robot navigation, or limited to semi-static or single-robot domains, it has often been dismissed for the real-time low-level control of agents due to the limited computational time and the unpredictable dynamics. Many robots now rely on local reactive methods for immediate control of the robot, but if the reason for avoiding motion planning is execution speed, the answer is to find planners that can meet this requirement. Recent advances in traditional path planning algorithms may offer hope in resolving this type of scalability, if they can be adapted to deal with the specific problems and constraints mobile robots face. Also, in order to maintain safety, new scalable methods for maintaining collision avoidance among multiple robots are needed in order to free motion planners from the "curse of dimensionality" when considering the safety of multiple robots with realistic physical dynamics constraints. This thesis contributes the pairing of real-time motion planning which builds on existing modern path planners, and a novel cooperative dynamics safety algorithm for high speed navigation of multiple agents in dynamic domains. It also explores near real-time kinematically limited motion planning for more complex environments. The thesis algorithms have been fully implemented and tested with success on multiple real robot platforms.

Acknowledgements

First, I would like to thank my parents for supporting and guiding me as a student and as a son. I would like to thank my advisor, Manuela Veloso for her support of my research since 1998, picking up where my undergraduate education left off. Through much of my time at Carnegie Mellon, I would like to thank Scott Lenser, for always lending an ear for a problem research related or otherwise. The lab hasn't been the same since you left. I am indebted as well to all the members of our RoboCup teams past and present, for coming together to define a collective domain for pushing research which is much greater than the sum of its parts. Particular members who have helped me since I began my work include Mike Bowling, Brett Browning, Tucker Balch, Jennifer Lin, Dinesh Govindaraju, Mike Licitra, and Stefan Zickler. Without your work the teams never would have made it. I would like to thank other fellow lab members for our other collaborations, as well as the willingness to listen to me while I stood in front of a whiteboard to expound on a problem. Douglas Vail, Sonia Chernova, and Colin McMillen have best helped fill that role.

I am also indebted to particular research groups for supporting various aspects of my work. I would like to thank Masahiro Fujita for supporting an internship at Sony Corporation which allowed to explore planning on the QRIO platform. I would like to thank Tadashi Naruse, all his students (in particular Shinya Hibino) of Aichi Prefectural University for keeping our RoboCup project alive with their offer of joint collaboration and hosting me for RoboCup 2005. Most recently, I would like the thank the members of the ACO project from Rockwell Scientific for supporting my research work on complex kinematic planners.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Navigation

Navigation is a problem as old as autonomous mobile robotics itself. As soon as a robot can move on its own, it must figure out how to move to carry out its objectives or goals. More generally, a robotic or virtual agent will face the problems of motion control and constrained navigation in order to move successfully in an environment. These problems involve finding trajectories through a state-space from one world configuration to another configuration which satisfies some goal criteria. Finding such trajectories is referred to as *motion planning*. Two examples from robotics would be navigation from a robot's current location to some specified goal, or starting from the current robot state and reaching a configuration which is safe with respect to moving obstacles. A critical aspect of motion planning is that it can use simulated actions to evaluate possible trajectories in trying to solve a given problem. It can use this ability to find global solutions which fully solve a problem, whereas an approach using only immediate state information (without action simulation) may get locally "stuck" and fail to find a global solution. The motion planning approach depends on two important properties of the agent and its planner. The first is that the agent has some model of its environment or world state, while the second is that the agent has a model of what its actions do insofar as they affect the world state. Taken together, this allows an agent to predict the result of an action within the environment without actually executing that action, and use this as a primitive to build a motion planner for solving navigational problems.

An important subcategory of autonomous agents are mobile robots, which are free to move around an environment using a locomotive device such as wheels or legs. While an important

ability, this mobility significantly complicates the modelling of the environment and the effects of actions. Most stationary articulated robots can have precomputed models of their working area given to them, while mobile robots normally have to rely on perception to create such a model. Thus mobile robots traditionally were relegated to immediate one-step planning or reactive methods due to the minimal observability offered by sensors. Recent work, particularly with networked sensors and map-building, has allowed mobile robots to gain much better models of their environment, beyond what the sensors can detect at any given time. However, many mobile robots still rely on a reactive layer for local navigation of obstacles, relying only on the current state of the world and mapping it directly to an action. Improved world and action models mean there is an opportunity to use motion planning directly on top of robot control layers. A navigation system employing planning will, however, have to deal with the biggest issues facing mobile robots: The environment can change over time.

1.2 Dynamics

Dynamics is a key issue in navigation for mobile robots in partially structured environments, or more generally for navigation in a multi-agent system lacking explicit coordination. As considered in this thesis, dynamics can be split in agent dynamics and domain dynamics:

- **Agent dynamics** involve the effects of classical physics on the agent itself, resulting in the kinematic and dynamic limitations on the agent which may have to be considered for solving navigation problems. Other remaining differential processes on the agent apply as well, such as electrical and energy storage properties, although the example domains used in this thesis are primarily constrained by classical physics properties. Dynamics constraints limit the acceptable values for derivatives of an agent's position over time, while Kinematic constraints which limit motion along submanifolds of the configuration space. The combined set of constraints are referred to as *kinodynamic* constraints. Kinematic limitations apply at any speed, while dynamics constraints become steadily more important as an agent operates at higher speeds. Robot design cannot escape all agent dynamics issues, as even a holonomic robot lacking any kinematic constraints will face some form of dynamics limitations, and in particular bounds on acceleration and velocity. Thus dynamics limitations are a nearly universal issue for mobile agents.

- **Domain dynamics** involve changes in the problem instance as an agent operates. One cause of such dynamics are changes to the environment. An environment where

an agent operates alone, and obstacles do not change position can be said to be static. An environment where other agents operate, or where obstacles change over time, or even both, can thus be said to be dynamic. Environmental changes can result from classical physics, such as with a moving obstacle acting under known forces. However they can also be driven by other factors, such as discovery of previously unknown obstacles, or updated positions for existing obstacles caused by unmodelled outside influences. Additionally, Domain dynamics can also result from changes in the goal specification over time. This could be due to a higher layer (such as a task-oriented behavior using navigation as a primitive), or because the goal is specified in a dynamic fashion (such as a robot tasked with following another agent).

Another way of classifying overall dynamics into categories is to split based on predictable properties versus unpredictable properties:

- **Predictable dynamics** involve aspects that can be accurately modelled, such as classical physics involving acceleration, velocity, and forces. It can also describe domain changes which evolve in a known fashion over time, such as an obstacle which follows a known trajectory (even if the forces which drive it are unknown.)
- **Unpredictable dynamics** involve aspects that are not modelled, such as alteration of the problem instance by other agents, humans, or exceptional events. Unpredictable dynamics can also describe the gathering of additional information which alters and updates the agent's environment model, even if there was no change to the domain itself. Finally, unmodelled errors in the robot's actions or sensors can also contribute unpredictable dynamics since they alter the problem instance in an unknown way.

A final way of classifying dynamics is by their area of influence:

- **Local dynamics** can be used to describe changes which affect the local area near the agent. An example of local dynamics is the detection of new obstacles nearby a robot using an onboard sensor.
- **Global dynamics** describe changes which affect non-local areas. An example of global dynamics is a robot modifying its world model with map updates sent by other agents.

While the distinction between local and global dynamics may not be particularly interesting in forming a taxonomy or understanding the concept of dynamics, it is important from an

implementational view, as the area which involves dynamics affects the running time of many motion planning algorithms. As a result, the local and global distinction is one way to compare different algorithms when evaluating their suitability for a particular domain.

Now that we have identified several ways with which to classify dynamics, we can describe why such a distinction is useful. Much research has concerned navigation with dynamics, however it has mostly been concerned with predictable dynamics (see Chapter 6 for a detailed description of related work.) As for unpredictable dynamics, it has mainly been addressed in existing research as local unpredictable dynamics, involving changes which occur near the agent. However, a mobile robot may easily face unpredictable global dynamics coupled with predictable local dynamics. Several of the robot platforms considered as targets for this work match that template. Thus, within this thesis we will assume a design goal of operation within a globally unpredictable domain, where changes are made to the environment completely outside of the control of the agent driven by our algorithms. In addition, strong local predictable dynamics are present, affecting the robot's immediate actions. A typical case of such a dynamic workspace is the case of multiple agents operate within the same environment at high speeds.

1.3 Problem Definitions

This section defines the problems of motion planning and cooperative safety as addressed in this thesis. The major notation used for planning and safety are given, along with the definitions of general terms used throughout the document. It concludes with remarks about the complicating factors for motion planning and cooperative safety algorithms.

1.3.1 Motion Planning

In order to form a specification for the motion planning problem as it is applied in this thesis, a more formal definition is adopted. While the planning algorithms described in subsequent chapters are expressed less formally using pseudo-code, each still follows the definition given here. Thus the definition can be thought of as both a specification and a limit on the scope of the problem addressed by this thesis. We now derive a notation similar to Latombe [62], but with some additional generalization.

Definitions:

- W: The robot operates in an environment called a workspace, referred to as W. W has N dimensions (normally $N \in \{2, 3\}$) and is a subset of $\mathbf{R}^N$.
- B: Objects in the workspace that the robot cannot interpenetrate (obstacles) are denoted individually as B_i for some i, and the entire set denoted as B.
- A: A single robot is referred to as A. In the multi-robot case, there are n robots, and each robot is referred to as A_i.
- q: The position of a robot includes values for linear dimensions (such as x and y position), as well as values for orientation (θ) or the angles of various joints. Each of these degrees of freedom (dof) is bounded, and may be modular (such as for orientation). A vector q including all degrees of freedom is called a configuration.
- C: The set of all q values within range defines a subset of $\mathbf{R}^m$ called the configuration space, often abbreviated as *C-space* and denoted as C.
- $C_{free}, C_{contact}, C_{nonfree}$: At each q, the robot A may be penetrating an obstacle, in contact with an obstacle, or not in contact with any obstacle. The set of all configurations where A is inside an obstacle is called $C_{nonfree}$, while the set of obstacle contact configurations without interpenetration is called $C_{contact}$. The configurations where A touches no obstacles is C_{free}. The three sets are disjoint, and their union is C.
- $q_{init}, Qgoal$: The robot starts at position q_{init}, and if an explicit goal position exists it is called q_{goal}.
- G: A goal can more generally be expressed as an evaluation function $G : q \rightarrow \mathbf{R}$. In this case the q_{goal} is set to be the maximum of G over C.
- $\tau(s)$: A path is denoted as $\tau(s)$, and is a continuous function mapping $s \in [0, 1]$ to a configuration in C. A path is constrained by $\tau(0) = q_{init}$.
- *valid*: A path $\tau(s)$ is said to be *valid* if $\forall s.s \in [0, 1] \Rightarrow \tau(s) \in C_{free}$, i.e., the path does not touch any obstacles.
- *solution*: A path $\tau(s)$ is a *solution* if $\tau(0) = q_{goal}$.
- *feasible*: A path may be valid, but due to constraints on the robot's constraints (such as the inability to fly) it may not be executable by the robot. Such a path is *infeasible*, while a path that is within the robot's capabilities it is said to be *feasible*.

The general path planning problem for an explicit goal is:
Given: A, C_{free}, q_{init}, q_{goal};
Find a path $\tau(s)$ which is *valid*, *feasible*, and a *solution*.

We can also distinguish between two variants of the general path planning problem:

Path planning refers to algorithms where s is a pure parameter, and a path-tracking motion controller of some sort will be applied to execute the path.

Motion planning refers to algorithms where s can be mapped to time by some mapping function (i.e. $t = f(s)$). Thus τ defines a trajectory through the state space, based on time. Motion planning is thus a subset of path planning.

Although there are differences between these two types of planning, holonomic robots often blur the distinction with their ability to execute any path at a sufficiently low speed. This thesis deals with both holonomic and non-holonomic robots; The distinction between path and motion planning is mainly reserved for the latter, where the difference is more significant.

1.3.2 Cooperative Safety

In addition to motion planning, we can define a pure version of what it means for a group of robot agents to maintain safety. Note that this is independent of reaching any particular goal, so it is not directly expressible as a kinodynamic [1] motion planning problem with added dynamics dimensions. The definition adopted here is by no means the only way of expressing the concept of safety, and alternate formulations exist (see [32], [46],and [36,37,61]). However, the author is not aware of any alternate formulation that has become dominant, so we adopt our single formulation of cooperative safety for the remainder of this thesis. We have found our formulation to yield a practical algorithms that can be applied to currently existing robots.

We define safety as: Given n agents, where each agent i has a trajectory defined by $q_i(t)$ through state space ($q_i(t) \in C$), and occupies some space $r(q) \in C$ when at position q, then safety at time t is:

[1] *Kinodynamic* refers to combined kinematic (positional) and classical dynamic (position derivatives) properties

$$R_j(t) = \bigcup_{i \in [1,n], i \neq j} r(q_i(t)) \quad (1.1)$$

$$S'(t) = \forall_{i \in [1,n]}\, q_i(t) \in (C_{free} - R_i(t)) \quad (1.2)$$

Where $R_j(t)$ refers to the area covered by all robots *except* j, and $S'(t)$ is the boolean safety function which is true iff all robots are in the remaining free configuration space after removing the areas covered by the other robots. Thus if $S'(t)$ is true, no two robots will collide, and no robot will pass outside of C_{free}. Furthermore, if there are constraints on $\dot{q}(t)$ or $\ddot{q}(t)$ (i.e., velocity and acceleration) in addition to $S'(t)$, we have the problem of *cooperative dynamic safety*. The goal is that agents are able to navigate while maintaining $S'(t)$ at all times, thus ensuring collisions do not take place between agents or with environment obstacles.

1.3.3 Complicating Factors

The difficulty of a particular motion planning or safety problem as we have defined it depends on many parameters, including the complexity of the environment or workspace, constraints on robots' actions, and the ability of the robot to observe and accurately model its environment. The workspace complexity is based on factors such as dimensionality, the number of obstacles, and the complexity of obstacle geometry. Constraints on actions are physical limitations of the robot, such as non-holonomic motion, bounded velocity, or bounded acceleration. Throughout this work, there is a focus on dynamics, which can contribute to motion planning complexity both through predictive difficulty (unpredictable dynamics) and as constraints (predictable dynamics).

1.4 Existing Approaches to Motion Planning

Many design decisions have been made in research on classical navigation problems, and several of those will be adopted in this work The first is the concept of *replanning* to deal with dynamics. The simplest approach is called *unconditional replanning*, where the agent replans each time a new action is to be decided, usually at some regular interval. Alternatively, the agent can plan once, and then monitor the environment and its execution of the plan to determine if it succeeds or fails. If the plan fails during execution, the agent can replan at that time, and then continue its execution. This is called *conditional replanning*. Both

1. Map initial and goal locations to C-space representation
2. Update environment model with new information
3. Update C-space representation graph, or roadmap
4. Search roadmap for a path between initial and goal locations
5. Extract path vertices and edges as plan

Table 1.1: A generic motion planning and replanning algorithm

of this methods alternate planning with execution, and thus can be classified as *interleaved planning and execution*. While some algorithms treat replanning the same as planning from scratch on a new problem instance, other algorithms attempt to carry past information to aid in replanning.

Navigation problems for a mobile agent are often divided into several distinct distance scales, making the overall problem more tractable. Each level feeds a sequence of target locations to lower levels, which lead the robot to the goal. High level planning (such as driving directions on a road network), can largely ignore local motion constraints of the robot, and are global, meaning that a plan is generated all the way from the initial position to the goal position. Mid-level planning refers to methods for navigating a medium size environment (such as within a parking lot), where kinematic constraints on a robot may need to be respected to find a solution (such as parallel parking), but agent dynamics can mostly be ignored. Finally, local planning involves avoiding immediate obstacles while respecting all motion constraints, but is only concerned with reaching goals in a small area, such as out the braking distance. Also, it is common for lower layers to experience more domain dynamics, although this is dependent on the particular environments the agent is operating within. An example of this property is that significant changes in a road network occur less frequently than small obstacles impede the immediate path of a vehicle.

After determining appropriate distance scales and a replanning method, a planning algorithm must be selected. Many options exist, and a detailed overview is provided in Chapter 6, while a short summary will be provided here. First, a generic planning/replanning algorithm is listed in Table 1.1. For a replanning algorithm, the roadmap update step can make use of an existing roadmap from previous queries, while a non-replanning planner will use a new model created from scratch each query. Some algorithms also interleave the roadmap update step (3) with the roadmap search step (4) to gain efficiency for queries covering a small part of the total configuration space.

The planners listed in the related work can be split into two major categories based on the method for generating a roadmap from the configuration space. The two categories are grid-based approaches and randomized sampling approaches. Grid based planners use a regular grid over the environment, with one vertex per cell, and implicit edges connecting each cell to its immediate neighbors. Each cell can have an associated traversal cost, defining it either as free, an obstacle, or assigning some intermediate traversability. If the data source for the environment model is itself a grid, this representation is quite direct, whereas if the data source is not in grid form, a grid must be overlaid on the actual domain, and the cost values of cells populated. The initial and goal locations are mapped to the nearest cells. Edge weights for the implicit roadmap graph are set based on the cell traversal costs, and the resulting graph can be searched using a minimum-cost graph searching algorithm such as Dijkstra's or A^* [72]. The D^* [80] extension of A^* allows for efficient replanning with local updates of the environment around the initial position, as well as updating the initial position itself. D^* updates only the affected portion of the A^* search tree when changes are made to the environment, allowing more efficient update steps when compared to A^*. The popular Field D^* [35] algorithm relaxes the assumption of paths at fixed angles between cells, resulting in shorter paths on average compared to D^*, while sharing its replanning efficiency properties with D^*.

In contrast to grid-based approaches, randomized approaches use random sampling of the C-space in order to construct a roadmap. The Probabilistic Roadmap (PRM) family of planners first samples configurations from C-space at random drawn from some distribution over C-space [53, 54]. First, vertices are drawn at random from the distribution, and the initial and goal configurations are also added as vertices. Then, a local search algorithm attempts to connect nearby pairs of vertices, and if a free path is found, an edge is added between the two vertices representing the local planner's path between the two configurations. This forms a graph of C_{free} which can be searched using A^*. Classical PRM can answer many queries by attempting to connect the initial and goal positions to an existing graph, making it a multi-shot planning algorithm. However, domain dynamics can change the environment, and thus invalidates the roadmap. Some variants of PRM, such as Lazy-PRM, mitigate the construction time by deferring collision checks to the graph search stage, making it more appropriate for dynamic environments. Another family of randomized planners are based on the Rapidly Exploring Random Tree (RRT) concept [64, 65]. This interleaves roadmap construction with search by growing a tree incrementally from the initial position. Configurations are sampled randomly from C-space, and the nearest node in the current roadmap is extended a fixed distance toward the sampled point. The new edge and vertex are added only if the local path is free of obstacles. The sampling and extension steps are repeated to grow a search tree out from the initial position until it reaches a goal state. Many variants exist to improve the search efficiency of RRT, but it is a pure planner without improved efficiency replanning capabilities. The recent DRT [33] planner adds a replanning

mode where environment updates are used to invalidate branches of the RRT tree, and thus it supports efficient replanning for changes near the initial position, such as newly gathered sensor data about obstacles. This is analogous to D^* replanning for grids.

Handling domain dynamics is an important criteria for choosing a motion planning algorithm for a mobile robot, thus it is important to consider the type of dynamics each algorithm can handle. Figure 1.1 conceptualizes the appropriateness of various planning algorithms under conditions of increasing domain dynamics. At the left, there are no dynamics at all (high coherence), while at the right, there are maximum global domain dynamics - each replan is totally unrelated to the one preceding it (no coherence). With little or no dynamics, a static planner without support for replanning is sufficient. In a static domain, the solution need not be updated. Explicit replanners may not be appropriate in this case compared to their static counterparts, as they usually involve some additional overhead (such as with D^* versus A^*). With limited dynamics, such as shifting the initial or goal positions, explicit replanners such as PRM, D^* and DRT gain an advantage, and become the most appropriate planners. Once dynamics changes involve the local environment however, the PRM variants no longer work best, leaving only D^* and DRT. As domain dynamics starts to incorporate global changes however, all of the aforementioned algorithms are no longer appropriate. An example of such a situation would be moving both the initial and goal locations by some amount, as well as modifying the obstacles in the environment without a particular locality of change. At extreme amounts of change however, fast static planners once again become the most appropriate; The change is so great between replans that it is effectively a new planning problem each time. This leaves a large gap between moderate global dynamics and extreme dynamics, into which some mobile robotics applications fall. This thesis introduces the Execution-Extended RRT planner, or ERRT, to address this gap. It makes no assumptions about the locality or type of domain dynamics, and offers a few parameters to tune the planner for the level of dynamics present.

In the case of a multi-agent domain, a navigation system may need to control multiple robots in tandem. Planning for multiple agents can be handled with several different approaches. One conceptually simple method is to concatenate all the agents into a single representation of a joint state space and a joint action space. Unfortunately, the joint space approach leads to poor scalability due to the exponential dimensional scaling of most planning algorithms. This has led to popular decoupled approaches such as *prioritized planning* [87], where the agents plan in a priority order, and later agents avoid the paths planned by earlier agents. While the prioritized planning approach is incomplete, it scales well with the number of robots.

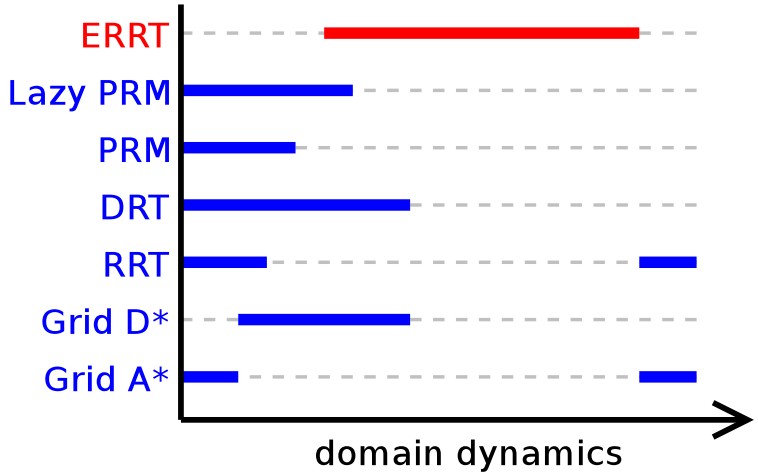


Figure 1.1: A qualitative comparison of the level of domain dynamics targeted by the design of various motion planning algorithms.

1.5 Approach and Thesis

Thesis: Safe multi-robot navigation operating within a real-time constraint is feasible using a combination of randomized motion planning and a cooperative safety algorithm. Furthermore, the approach can yield a practical navigation system for multiple robots operating in unpredictably dynamic environments.

Our approach to motion planning differs from much of the traditional research on motion planning, in that it involves an emphasis on constrained timing requirements as opposed to the solution of increasingly complex environments with calculation time used only as a benchmark. In general, a given problem domain will have some minimum difficulty imposed by the complexity of problem instances, and some maximum time in which a solution must be returned in order for a candidate algorithm to be practical. Such a hypothetical situation is shown in Figure 1.2. An algorithm's performance represents a curve plotting problem difficulty against the calculation time used, which increases with problem difficulty. The time and difficulty constraints can be represented as lines on the plot. The timing constraint imposes a maximum time allowed to solve a problem, thus the algorithm must lie below this line to solve problems of a particular difficulty within the alloted time. The difficulty constraint is a vertical line showing the minimum difficulty that a problem from that domain will entail; While simplifications can be made in problem domains, at some point no further simplification can take place due to the inherent difficulty. An algorithm can be said

to "solve" a problem domain if part of its performance curve lies both below the timing constraint and to the right of the complexity constraint. This is the case for the algorithm shown in Figure 1.2. If one considers the extremes of timing and difficulty constraints, the hypothetical situation shown in Figure 1.3 can arise. In situation (A), the timing constraint dominates, leading to the research question of how difficult a problem can be solved under the timing constraint. In situation (B), the difficulty constraint dominates, leading to the question of how fast a problem of that difficulty can be solved. Note that in each case, a different algorithm is better for each task.

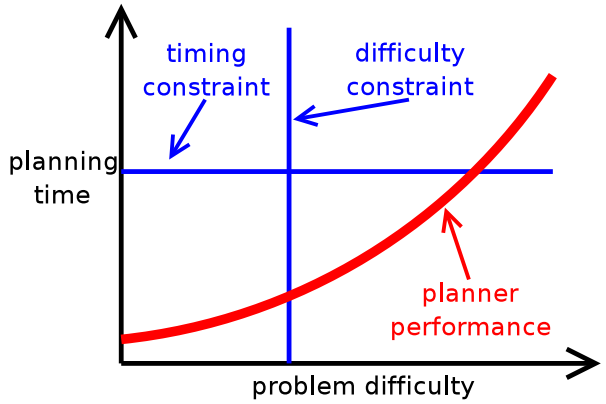


Figure 1.2: A qualitative explanation of planning challenges, plotting problem difficulty against the time used for a hypothetical planning algorithm.

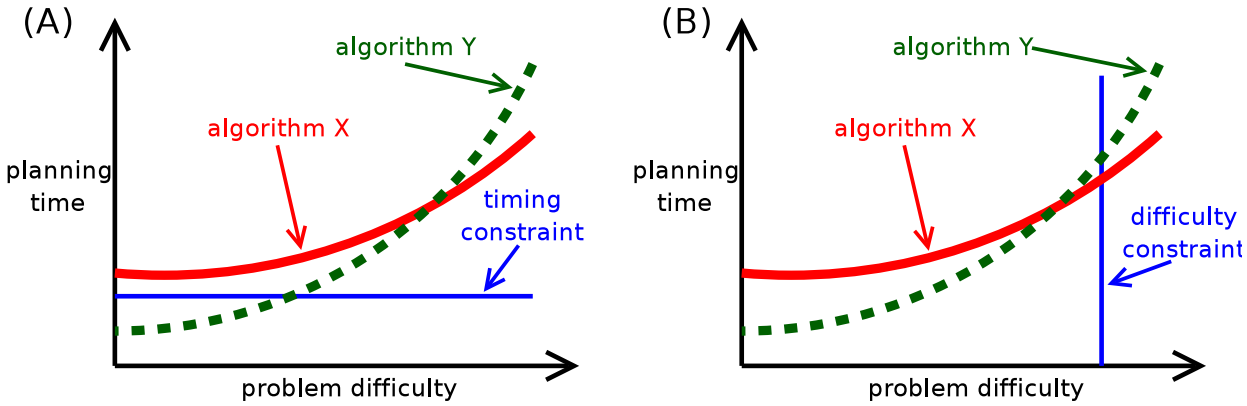


Figure 1.3: A hypothetical comparison of two algorithms under a given timing constraint (A), and a difficulty constraint (B).

This work is very much like the situation shown in Figure 1.3 (A), whereas much of the traditional research in motion planning attacks the problem shown in (B). This is because much

of the traditional work in motion planning is concerned with one-shot queries or multi-shot planning in static environments, and solving such problems in increasingly complex environments characterized by high dimensionality and complex free configuration spaces (See [65] as a typical example). Mobile robots are instead characterized by lower dimensional dynamic environments and nearly continuous re-queries. As a dynamic mobile robot environment can be unpredictable, any path found by a motion planning algorithm has a short period of applicability before it becomes obsolete. This short period leads to an implied iterative control cycle, and imposes a timing constraint on any planner used. In the robot domains considered in this thesis, these timing constraints end up the dominant factor in choosing a planning algorithm. Attempting to solve more complex problems then becomes a dependent factor for comparison, rather than the independent factor as in traditional work.

Thus at its core, this work seeks to extend the boundaries of motion planning, but not in terms of absolute difficulty of problem instances. Rather, it seeks to improve the solution speed of problems, allowing applications where navigation must be carried out within tight timing constraints to solve more complex navigation problems, while remaining within the timing constraint. Additionally, the acceptance of all types of dynamics allows significant changes in the environment and the goal specification can be made between replans, allowing the navigation system to be treated of as a primitive by task-oriented behaviors, rather than treating navigation as an end in itself.

The timing requirements for semi-structured robot domains is dictated by the presence of dynamics, and in particular unpredictable dynamics. A solution must be found and partially executed before the information becomes irrelevant due to unpredicted changes. In robot navigation systems where incomplete or heuristic planners have been used, a common reason for such a choice has been the tight timing constraints imposed by either the domain itself or limited computational resources present on the agent. While some existing mobile robot systems use planning for low-level control, many more do not use motion planning at all, instead relying on local reactive methods for immediate control of the robot, or limited one-step enumeration and evaluation of steering commands. The use of motion planning is restricted to higher levels of a navigation system, or is forgone completely.

The different requirements of mobile robot systems do not mean, however, that recent developments in motion planning for highly complex domains have no place in mobile robotics. What it does mean is that these algorithms must be adapted, and their capabilities improved for these domains to be practical for mobile robot control. Particular progress has been made on traditional planning problems using randomized sampling approaches, such as Probabilistic Roadmaps (PRM) and Rapidly exploring Random Trees (RRT). While the focus of these techniques has been solving difficult problems in cluttered environments and high-dimensional spaces, in this thesis I will demonstrate how they can be adapted for inter-

leaved planning and execution at high control rates in highly dynamic domains. In addition, local dynamics constraints can be handled by a real-time cooperative safety method, complimenting the motion planner's ability to deal with global unpredictable dynamics. The safety method also allows the robots to be planned for individually, ignoring the other robots for the purposes of the path planner, and using the safety algorithm prevent collisions rather than relying on a joint state space or path obstacles from prioritized planning. Specific contributions follow.

1.6 Thesis Contributions

The major contributions of this thesis are the following:

- **The DSS Cooperative Multi-Robot Safety Algorithm:** The novel multi-agent *Dynamics Safety Search* (DSS) algorithm is described. It is based on Fox et al.'s single-agent *Dynamic Window* approach [38]. DSS offers guaranteed safety for single-agent and coordinated multi-agent systems. The *Dynamic Window* approach cannot guarantee exact safety even in the single agent case. DSS is shown to have polynomial complexity (as opposed to exponential complexity for joint planning), and near linear complexity in simulation testing. Finally, DSS is demonstrated to aid in collision avoidance on real robots from the RoboCup small size domain.

- **The Waypoint Cache for Biased Replanning:** The Waypoint cache is a method of using previous plans to alter the sampling distribution for randomized replanners. It allows more efficient replanning in dynamic domains, and is used to develop the ERRT planner.

- **The ERRT Randomized Motion Planner:** The Execution-Extended RRT (ERRT) algorithm is a novel extension of the RRT family of planners. ERRT can be tuned to work for varying levels of domain dynamics faced in a particular application. Bidirectional Multi-Bridge ERRT introduces additional parameters to the Kuffner's RRT-Connect algorithm [51]. These new parameters allow a tradeoff between planner efficiency and plan length optimality.

- **Survey of Collision Detection for Motion Planning:** Existing work on collision detection is evaluated for the task of motion planning, which differs in emphasis from traditional comparisons of collision detection for other applications. The survey of work contributes observations of similarity between several algorithms when applied to motion planning.

- **Navigation for a Competitive Multi-Robot Soccer System:** Navigation lies at the heart of a system with multiple robots working as a team in a soccer domain. In addition to exploring the algorithms individually, robot soccer provides a way to combine path planning and safety methods in the context of a larger system with goals more complex than reaching fixed configurations. Using the algorithm as a primitive for an autonomous soccer system helped drive requirements for robustness and efficiency, while the competition offers a metric to compare implementations from many research groups.

Additional contributions of this thesis are the following:

- The *Extent Masks* collision detection algorithm is a novel contribution of this thesis research. It offers a tradeoff compared to other algorithms, scaling linearly in the number of obstacles rather than sub-linearly, but offers pure linear scaling to any number of dimensions. It it fast in practice for any domain with a small number of obstacles relative to the space occupied or the dimensionality of the environment. (Chapter 3)
- The novel Dynamic PRM algorithm is introduced, which extends Kavraki et al.'s Probabilistic Roadmap (PRM) algorithm [54]. It is tested on the QRIO humanoid robot. (Chapter 4)
- The mathematical approach to iterative swept-sphere collision detection developed in this thesis, although simple, appears to be novel. Quinlan's [75] related approach does not use the explicit iterative formulation. (Chapter 3)
- Requirements for robust motion planning for robotics applications are introduced, and in particular robustness to location error. The author is not aware of these requirements or their offered solutions being identified in existing work on randomized or graph-based motion planners. (Chapter 4)
- The heuristic of an *active goal* is introduced to solve kinematically constrained planning problems where the goal is defined as a fixed-radius orbit around a point. It is demonstrated as part of a planner for fixed-wing unmanned aerial vehicles (UAVs). (Chapter 4)

1.7 Guide to the Thesis

Table 1.7 indicates the chapters of particular relevance to understanding a given contribution.

Contribution	Ch.1	Ch.2	Ch.3	Ch.4	Ch.5	Ch.6	Ap.A	Ap.B
Multi-Robot Safety	•	○	○		★	○	○	
Waypoint Cache	○	○		★				
ERRT Motion Planner	•	○	○	★		•		
Col. Detection Survey		○	★	○				
Robot Soccer System	○	★	•	★	★	○	★	★

Key: ○=Somewhat relevant, •=Relevant, ★=Essential

Table 1.2: Guide to the thesis

Chapter 1 introduces the general problem statements for motion planning and safety and defines the scope of the work.

Chapter 2 describes the robot domains used for motivation and testing of the approaches in this work

Chapter 3 describes the collision detection problem for path planning and several approaches. Very high efficiency is required for practical real-time path planning. A novel approach is introduced for collision detection with sets of obstacles

Chapter 4 describes the motion planning algorithms which extend the existing randomized path planning approaches. Pseudocode and experimental timing data is given. Also describes application extensions.

Chapter 5 introduces the concept of multi-agent cooperative dynamics safety with a novel algorithm for providing it at real-time rates for moderately sized teams.

Chapter 6 explores related work for motion planning, and the most relevant safety method.

Appendix A describes the vision system used to gather positional data for the small-size domain. This chapter supports sensor error models used elsewhere in the document with experimental data

Appendix B explains the software system for the CMDragons small-size team in detail.

Chapter 2

Domains

Figure 2.1: Several platforms used in navigational research. These include RoboCup small-size league soccer (left), a fixed wing UAV (center), and the QRIO humanoid robot (right).

In research has been motivated by several platforms spanning a wide variety of the mobile robot parameter space. For each domain, a navigational system was developed using randomized planning with an iterated approach to replanning. The three primary domains which are the focus in this work are the following:

- RoboCup small-size league multi-robot soccer
- A kinematic theater navigation system for fixed-wing UAVs
- A 2.5D (heightfield) path planner for the QRIO humanoid robot

The properties of each of these domains and their navigational requirements are described in detail the subsequent sections.

2.1 RoboCup Small-Size Multi-Robot Soccer

The primary motivating domain for our navigational research since 2001 has been the the RoboCup F180 "small-size" league [55]. This league involves teams of five small robots, each up to 18cm in diameter and up to 15cm height. The robot teams are entered into a competition to play a type of soccer against opponent teams fielded by other research groups. Two halves of 15 minutes each are played, and during a game no human input is allowed. The match is refereed by a human to ensure fair play, and the referee's signals encoded and sent via a serial link to provided to each team. Thus the two robot teams must compete using full autonomy in every aspect of the gameplay.

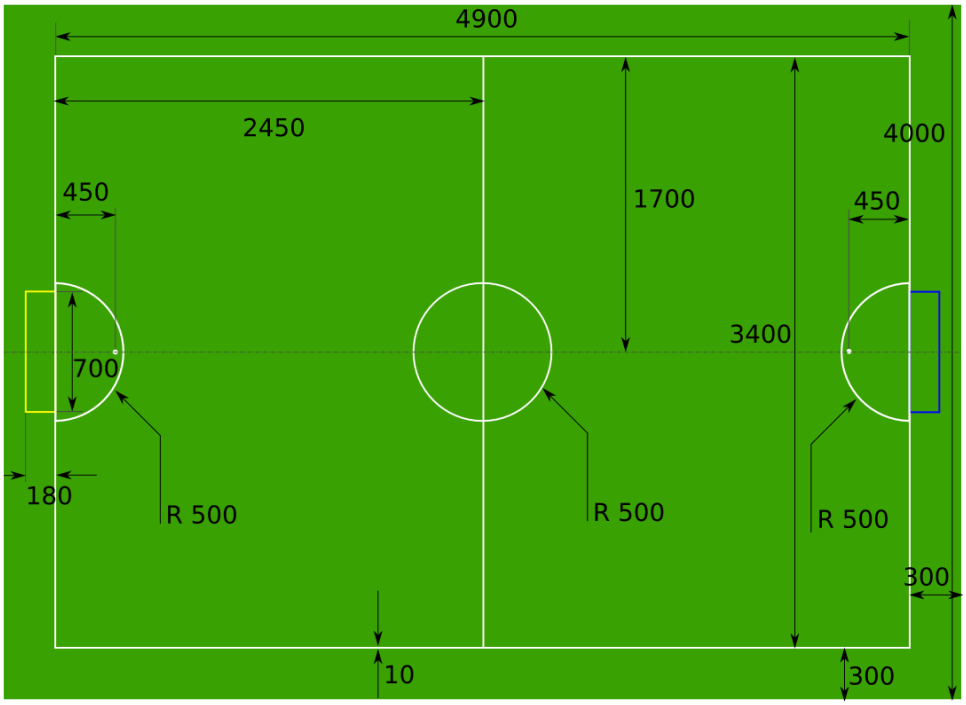


Figure 2.2: The dimensions of the current RoboCup small-size field.

The field of play is a carpet measuring 4.9m by 3.8m, as shown in Figure 2.2. The robots, in addition to fitting within a 18cm cylinder, are also forbidden from covering more than 20% of the ball, defined by the area of the ball falling within the convex hull of the robot when

projected onto the ground plane. Robots are not allowed to gain full control of the ball and remove all of its degrees of freedom, promoting team play [79]. An example of a small-size game (on an older, half-size field) can be seen in Figure 2.3

Figure 2.3: Two teams are shown playing soccer in the RoboCup small size league.

For the team control system, offboard sensing, computation, and communication are allowed. This has lead nearly every team to adopt a centralized approach for most of the robot control [19, 45, 78] Sensing in a typical system is provided by two or more overhead cameras mounted 4m above the field. The camera signals then feed into a central computer to process the image and locate the 10 robots and the ball on the field 30-60 times per second. The systems implement soccer strategies, behaviors, and control on the centralized computer. Finally, velocity commands are broadcast via a radio link the the individual robots, which implement velocity control on the individual robots.

Due to its competitive nature, teams in the small-size league have rapidly advanced the boundaries of robotic technology applicable to the domain. Since its start in 1997, teams have improved dramatically in capability. Top teams now have holonomic robots which can travel at speeds of over $2m/s$, with maximum accelerations between $3 - 6m/s^2$. Four generations of small-size hardware from the CMUnited and CMDragons teams are shown in Figure 2.4.

Figure 2.4: Five generations of Carnegie Mellon robots: (from left) 1997, 1998-99, 2001, 2002-03, 2006

Kicking devices have been implemented with can propel the ball at up to $15m/s$. Ball control devices composed of spinning rubber bars have been introduced to partially control the ball by imparting backspin, allowing robots to receive passes and also drive short distances [1]. In the last two years, special "chip kicking" devices have been introduced which can impart vertical as well as horizontal velocity to the ball, while remaining within the 20% holding rule. The most advanced of these kickers can propel the ball a distance of $4.5m$ in the air with the ball reaching a maximum height of $1.5m$ from the ground plane.

The speeds involved require every module to run in at real-time rates to minimize latency. Each component of the system must be efficient enough to leave sufficient computing resources for all the other modules and tasks. Since five robots must be controlled at up to 60Hz, this leaves a realistic planning time budget of only $1ms - 2ms$ for each robot. The approach taken with these robots is to adopt a path planner which ignores dynamics, and pair it with a dynamics safety system which maintains safety while incorporating a more complete velocity and acceleration model of the robot. An example of a planning search tree generated in a small-size like environment (with additional obstacles added) can be found in Figure 2.5.

2.2 Fixed Wing UAV

The second platform is an autonomous unmanned air vehicle (UAV), and in particular an autopilot designed for the small UAV shown in Figure 2.1. The navigation system is part of a much larger software system for autonomous collaborative control of multiple UAVs. The

[1]The current rules limit a robot to dribbling only 50cm before it must lost contact with the ball. This was introduced in 2004 to promote team play

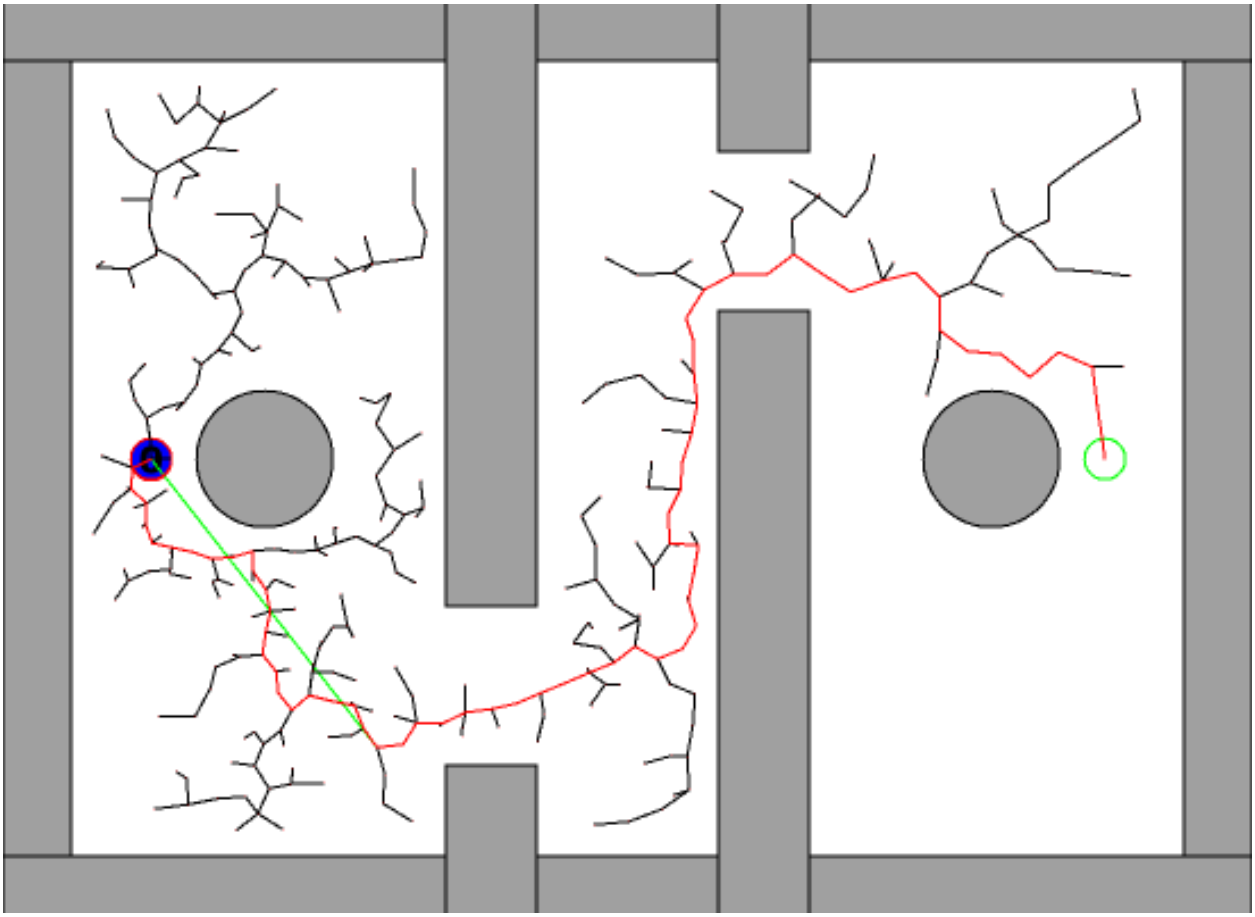


Figure 2.5: A robot on the left finds a path to a goal on the right using the ERRT algorithm.

overall software system consisted of an operator interface, numerous behaviors, a market-based task assignment system, an area coverage planner, and a route planning module. This was then fed to the hardware system which consisted of a radio link to a UAV autopilot. In this project, this stage could be simulated in software or using a hardware-in-the-loop (HIL) simulator which paired real radio links and autopilot hardware modules with a software aircraft simulator. The system was designed to easily incorporate real UAVs in a possible follow-on to the project.

The author was responsible for developing the route planning module which planned from a given UAV state to either a final destination point or a destination orbit. The motion planner had to support terrain data as well as primitive obstacles (no-fly zones), and model the kinematic constraints of the fixed-wing UAV. The aircraft, and thus the planner must operate in 3D at low to intermediate altitude, avoiding both the terrain and the user specified obstacles. The UAV's kinodynamics are highly constraint compared to a ground vehicle or rotorcraft; Despite the UAV's relatively small size ($3.5m$ wingspan), the minimum turning radius is $300m$. In addition, climb and descent rates are limited to $5m/s$. The output of the planner was a set of of feasible waypoints. The autopilot can accept small sets of waypoints every few seconds, but the planner interfaced with a path buffer which would manage the waypoints so that the number of waypoints was not a practical limitation. The route planning module was called both for actual navigation problems as well as for deriving cost estimates for the market based task allocation of multiple UAVs. Thus, while the timing was relaxed, answers would have to be returned in less than 10 seconds to allow the operator interface to operate smoothly, with shorter paths expected to require less planning time. Although the timing budget is much greater than the small-size environment, due to the 3D nature of the problem and the constrained kinodynamics, the problem to be solved is much more difficult. An example of a kinematically constrained search tree is shown in Figure 2.6.

2.3 QRIO Humanoid Robot

The Sony QRIO robot is a small humanoid robot with both complex actuation and significant sensory capabilities [41]. The robot, shown in Figure 2.7, is able to use a stereo pair located in the robot's head to generate a 3D occupancy grid model of the environment. This data can be used to generate a 2D grid of height values, or a *heightmap*, which can be used for local path planning. Existing robot models are able to generate the 3d occupancy grid relative to the robots torso location projected onto the ground. A locomotion system was also provided to ensure balance and generate walking steps to reach ego-relative positions [41]. Finally, a module exists for traversing step obstacles if the robot is positioned approximately in front of the transition. Such a module helps compensate for the finite resolution of the occupancy

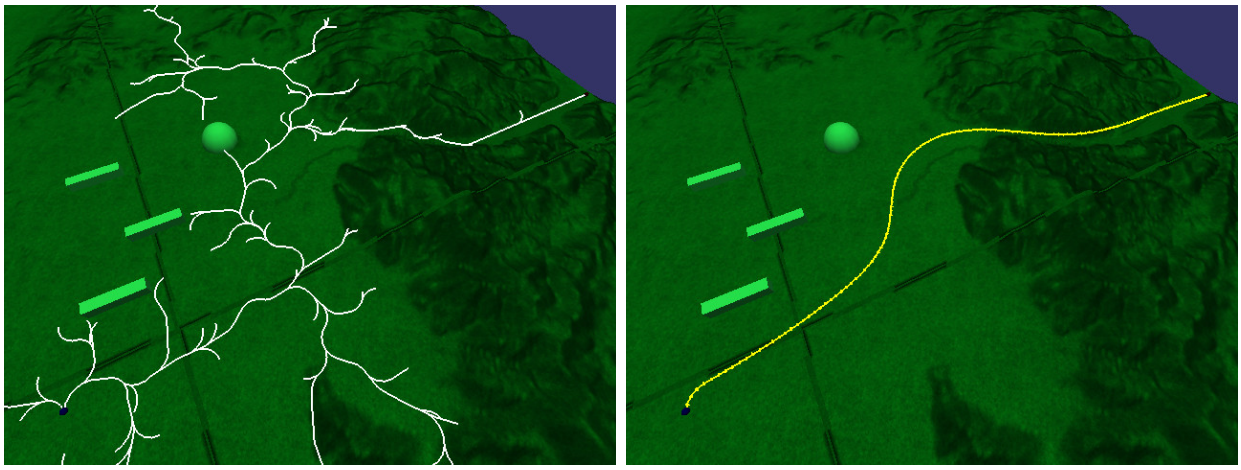


Figure 2.6: A kinodynamicly-limited search tree (left), and the corresponding simplified plan (right) for the UAV. The plan length is approximately 13km long.

grid representation, which is not sufficient for the high accuracy needed to traverse a step obstacle. This is because QRIO, like many humanoids, is limited in its step length, so positioning at the transition must be accurate for traversal to succeed.

A navigation module for QRIO is responsible for generating paths on a plane which avoid obstacles, as well as finding plane transitions that the robot can traverse. The the robot operates in a "2.5D" segmented heightfield, and the planner must generate both regular ego-motion goals for the walking controller as well as appropriate calls to the climbing/descent module when transitions are reached. Due to the Open-R architecture [40], planning can either run onboard the robot, or on an offboard computer via a wireless ethernet link.

The robot has a slower relative navigation speed when compared to the small-size robots, and thus a more relaxed timing constraint. Planning can take up to a few seconds, and batches of step commands can be executed which take from 2-10 seconds, at approximately 2 seconds per full step. The environment is complex in terms of obstacle geometry as it is derived from an occupancy grid model which is updated incrementally with stereo data. An example of a domain is shown in Figure 2.8, where a robot was commanded to navigate to the top of an obstacle after starting from the floor in the office environment.

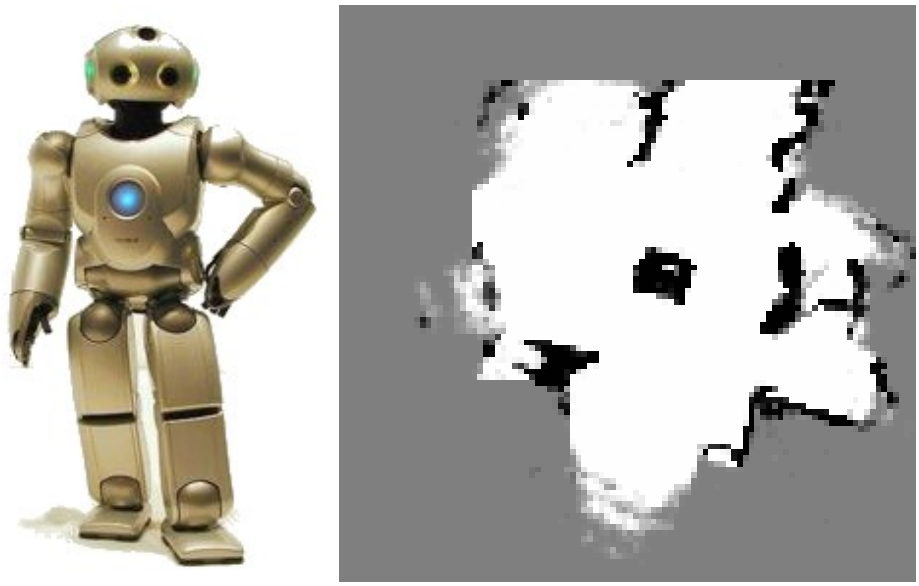


Figure 2.7: The Sony QRIO robot (left), and a stereo generated occupancy grid environment map (right).

2.4 Summary of Domains

The main properties of the domains can be summarized in Table 2.4. The small-size domain is used throughout this document for testing and development of algorithms, while the solutions developed for the UAV and humanoid domains are described further in Chapter 4.7.

Domain	Environment d.o.f.	Planning d.o.f	Planning Time	Control Period
RoboCup F180 (static)	2	2	2 ms	1/60 s
RoboCup F180 (dynamic)	2	6	2 ms	1/60 s
QRIO Humanoid	3	2.5	1-2 s	10 s
Fixed-Wing UAV	3	5	5-10 s	30-60 s

Table 2.1: A summary of the planning properties of several real robot domains

Figure 2.8: QRIO on top of an autonomously reached step obstacle.

Chapter 3

Collision Detection

Collision detection is an area of research in its own right, and has been extensively studied (for a survey see [68]). Despite the large body of existing work, further study of collision detection in the context of motion planning is still important for two reasons. The first is that path planning time in many domains is dominated by collision checks, meaning that optimization of this aspect provides an overall increase in efficiency. Randomized planners in particular tend to make a very large numbers of obstacle checks in solving a planning problem. This is an unavoidable fundamental property of current algorithms, because they discover the structure of the world through queries rather than some global analysis of the geometry. The second reason is that much of the collision detection research has focused on computer graphics and physical simulation, while the types of queries made for motion planning differ. In particular graphics and simulation is primarily concerned with "all-pairs" collision detection algorithms, as every object in the scene can undergo a transformation over time. In path planning, collision tests are geared toward incremental "what-if" tests of small changes in a state, and are structured as a tree or graph instead of a single evolving timeline. An exploration of collision detection is motivated by these differences, as they can potentially guide us to alternative solutions that work better for motion planning than an approach tailored for rendering or simulation.

The most general of the common approaches to collision detection support queries between complex polyhedra. For mobile robots however, it is often acceptable to make a more simplified model of the agent that consists of a small number of circles, spheres, or cylinders [75]. Many real applications desire the extra safety of an exclusion radius around the agent, for which bounding spheres and cylinders work as an approximation. One thing we can gain from this simplifying assumption is exact checks for safety along continuous trajectories; While polyhedral models are normally tested at only specific points, leading to the possi-

bility of "tunneling" between objects, circular and spherical geometries can be treated be checked as continuous sweeps over a trajectory. Thus we trade completeness (in the sense of the actual agent's shape) in return for an exact queries on the simplified model.

In contrast to the simplicity of the agent model, for the agent's environment we would like to support as many rich object geometries as possible. In particular we would like to be able to describe man-made objects, natural terrains, and minimally processed sensor data such as range maps. Thus the approach we take is to draw on the best existing representations for complex domains in the literature, and then take advantage of our simplified agent geometries to gain efficiency compared to more general approaches.

Typical models of the environment consist of collections of geometric objects of varying size and geometry [68]. This feature has led practical approaches to collision detection to break the task into two stages, so called *broad-phase* and *narrow-phase* collision detection:

Broad-Phase: Determines which geometric objects need to be checked for a certain query, ruling out distant objects based on bounding boxes or other partitionings of the space.

Narrow-Phase: For the objects that overlap in the broad-phase, a check is performed between the query and a single primitive object that represents an obstacle. Often objects correspond to rigid bodies.

We first focus on the various methods for broad-phase collision detection, attempting to group some of the myriad of available approaches into broader categories with common properties. Narrow-phase collision detection is mostly driven by the exact geometry of the query and the objects used to model the environment, so we only examine it within the context of our system. A broad look at narrow-phase methods are given in Lin's summary [68]. In the second part of this chapter, the approach we have taken is examined in detail. This comprises two methods for broad-phase collision detection, including a novel approach called *extent-masks* which leverages the parallelism inherent in modern computers, and a scalable implementation of hierarchical bounding volumes. Next our methods for narrow-phase collision detection are described, including an approach for sphere-based narrow-phase collision checks that leverages a point-object distance primitive to derive all other necessary checks. Finally, we examine the performance of the developed collision detection system.

3.0.1 Contributions of this Chapter

- The *Extent Masks* algorithm is a novel contribution of this thesis research.

- The mathematical approach to iterative swept-sphere collision detection, although simple, appears to be novel. Quinlan's [75] related approach of "bubbles" does not use the explicit iterative formulation.
- The heuristic used for choosing splitting planes for building obstacle trees has not appeared in any previous work of which the author is aware.
- The summary of existing work contributes observations of similarity between several algorithms when applied to collision detection for motion planners. In particular, spatial partition trees and hierarchical bounding volumes become quite similar for motion planning applications when common optimizations are applied.

3.1 Existing Approaches to Broad-Phase Collision Detection

Many methods for broad-phase collision detection have been described in literature and employed in implementing collision detection libraries. Numerous variants and derived versions of the algorithms exist, making an exhaustive description impractical. Instead, we group the methods into several categories and cover their usual properties. Below we describe the following categories of broad-phase methods:

- Coordinate Sorting and Sweep-and-Prune
- Spatial Hashing
- Spatial Partitioning Trees
- Hierarchical Bounding Volumes
- Extent Masks

3.1.1 Coordinate Sorting and Sweep-and-Prune

One early method for broad-phase collision detection is *coordinate sorting*. In this method, a marker for the start and end $[x_k, x'_k]$ for each object k is placed in several arrays, representing the axes of the workspace $(x, y, \ldots)$. The arrays are kept sorted by value along that axis, and determining if two objects may overlap along an axis involves searching for start or end markers of one object contained between the other object's markers. To carry out the test in higher dimensions, each axis is checked in turn, and collisions can be ruled out if

their is no overlap in any of the dimensions. If the objects overlap along all three axes, this indicates that their axis-aligned bounding boxes (AABBs) overlap, and this pair of objects must be tested in the narrow phase of collision detection. This method works well when the objects are spread along most axes, but fares quite poorly if many objects are stacked along any axis, which leads to an $O(n^2)$ worst case. Unfortunately it is quite common in structured environments to have many aligned objects, so coordinate sorting is generally not used for collision detection. One adaptation of coordinate sorting that has proved quite popular is the *sweep and prune* method used in I-COLLIDE [28]. This method combines initial coordinate sorting followed by tracking objects over time. Sorted coordinate axes are maintained over time while objects move, and with high inter-frame coherence, the number of swaps of end markers that need to be performed each time is relatively low. The crucial insight for collision detection is that a collision can begin or end only when two end markers of different objects switch positions. Any time such an event happens, the pair of objects are checked for overlap between their bounding boxes, and this state is recorded until the objects no longer overlap (which can also only occur during a marker swap along some axis). This method works well for evolving systems with small step sizes (such as physical simulation). However, for path planning we are more interested in "random-access" queries without any inter-frame coherence (the obstacles normally have inter-frame coherence, but the queries do not).

3.1.2 Spatial Hashing

Another method for broad-phase collision detection based on approximating axis-aligned bounding boxes is the *uniform spatial hash table* (USHT). A uniform spatial hash table breaks the workspace into a set of cells, with evenly spaced divisions along each axis. Each cell has a linked list of objects that overlap that cell. In order to check a region for collision with a set of objects, first each object is added to all the cells it overlaps (which is easy to calculate for AABBs with ranges along each axis), and for the query every cell is checked and narrow-phase collision performed between the query and each object on the list. Because objects often overlap several cells, a narrow-phase check could be performed many times with the same object during a single query. To mitigate this, we can maintain a unique integer query ID, with each object listing the last query which it has been checked against. If a query encounters an object it has already checked, it can skip performing the more expensive narrow-phase check. Spatial hashing works well for multiple similar sized objects and queries, and unlike coordinate sorting, it still does well in cases where there is a significant amount of overlap in one or more dimensions. However, it has a speed-vs-accuracy tradeoff based on the cell size, where small cells are more accurate at determining collisions but involve more work in adding the object to the additional cell lists. For the best performance, the cell size

must often be manually tuned. There is also potentially a steep memory tradeoff, although this can be handled by hashing the cell id using a traditional hash function to map it to a smaller number of storage cells. Hash collisions are handled by the already existing linked lists at each bucket/cell.

```
type SpatialHashTable = array[MaxX,MaxY] of list[ObjectRef];

function AddObject(table:SpatialHashTable, o:ObjectRef) : unit
1    let e^0 = ⌊ (o.bbox.cen − o.bbox.rad) / CellSize ⌋;
2    let e^1 = ⌈ (o.bbox.cen + o.bbox.rad) / CellSize ⌉;
3    foreach y:ℤ ∈ [e^0_x, e^1_x] do
4        foreach x:ℤ ∈ [e^0_y, e^1_y] do
5            table[x,y] ← table[x,y] ∪ {object.id};

function CheckBox(table:SpatialHashTable, q:Query) : Status
1    let e^0 = ⌊ (q.bbox.cen − q.bbox.rad) / CellSize ⌋;
2    let e^1 = ⌈ (q.bbox.cen + q.bbox.rad) / CellSize ⌉;
3    foreach y:ℤ ∈ [e^0_x, e^1_x] do
4        foreach x:ℤ ∈ [e^0_y, e^1_y] do
5            foreach o:ObjectRef ∈ table[x,y] do
6                if Overlap(o.bbox, q.bbox) ∧ NarrowPhase(o,q) ≠ Free then
7                    return Collision
8    return Free;
```

Table 3.1: Algorithm for constructing and using a basic spatial hash table

3.1.3 Spatial Partitioning Trees

Another broad category of broad-phase collision detection methods are those based on spatial partition trees. The simplest spatial partition is the KD-Tree, which recursively splits the environment into two halves along one of the domain axes. To build the tree, a splitting axis and location is determined, and the obstacles are put in one of two subtrees based on which side the obstacle falls on. Obstacles that straddle the splitting plane can either be placed in both subtrees, or it can be placed in one of the child trees which are then allowed to overlap slightly. The former approach is common in graphics for visibility queries, while collision detection often employs overlapping subtrees. If we relax the constraint of axis-aligned splitting planes to allow arbitrary orientations, the structure is a binary space partition tree or BSP-tree [39]. Both KD-trees and BSP-trees can be constructed in a number of ways,

although the most common approach is recursive subdivision. The construction starts with all the primitive objects in a single node, and splits the node into two children based on a plane chosen using some heuristic. The construction then proceeds recursively on the children until some minimal number of objects is reached, resulting in a leaf node. Pseudo-code for implementing a variant of a spatial tree with per-node bounding boxes (making it technically equivalent to a AABB-tree) is given in Table 3.2. It assumes a function *ChoosePlane* for choosing an appropriate splitting plane based on a set of objects. Many heuristics for choosing splitting planes have been put forward, although the most common approaches are to evenly split the longest axis of the current subvolume, or to split the objects into sets with an equal number of objects. An examples KD-tree and BSP-tree can be seen in Figure 3.1. Once such a hierarchical representation is created, it can be used for queries in a straightforward manner. A query need only be checked against the subtrees that it overlaps; any time that the query volume is one one side of a splitting plane, only that child needs to be checked. If the query straddles a splitting plane, the query volume is subdivided and each portion is checked recursively. If the query is itself represented as a spatial tree (such as a BSP), the checking can be quite efficient [71].

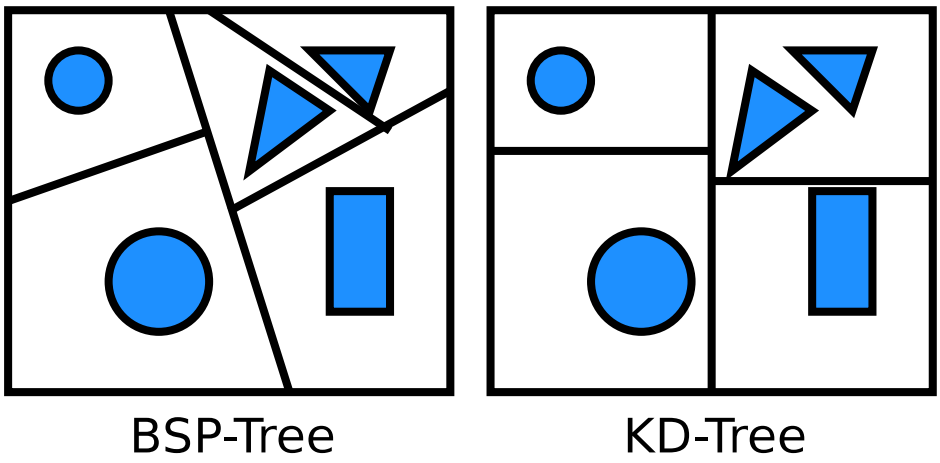


Figure 3.1: Examples of KD and BSP spatial partitioning trees

One method not mentioned above is the *interval tree*, which is popular in computational geometry [30, 31]. It can be thought of as a special case of a KD-tree with a special construction algorithm. The elements of an interval tree are pairs of extents for a particular axis $[x_k, x'_k]$, along with a maximum extent for the entire subtree rooted at this element. An interval tree is then a balanced binary tree built out of initial starting extent at each node. Then in a single linear pass the maximum subtree extents can be calculated. To go beyond a single dimension, the tree alternates the dimension represented at each depth level in the tree. An interval tree can thus be represented by an ordinary KD-tree, and we need only

to use the balanced binary tree for construction, which splitting planes based on the initial extents for the various objects. The KD-tree then needs to keep track of the maximal extent at each level of its tree. If a global AABB is constructed, then during any descent of the tree an AABB for subtree rooted at that node can be maintained.

3.1.4 Hierarchical Bounding Volumes

The final major category of broad-phase collision detectors are those that use hierarchical bounding volumes (HBVs) as their representation. The bounding volumes are typically either spheres, axis-aligned bounding boxes (AABBs) or oriented bounding boxed (OBBs). In these representations, a bounding volume is placed around all the objects at the root of a tree, and subtrees contain subsets of the objects with their own associated bounding volumes. Examples of an AABB-tree and an OBB-tree are shown in Figure 3.2. Since broad-phase collision detection is simply to find out which bounding volumes overlap and require further processing, HBVs and a natural hierarchical extension of the brute force method of checking every primitive. Pseudo-code for the spatial tree with per-node bounding boxes is listed in Table 3.3. In all variants, each group of primitives under a subtree has a "global" bounding volume around the entire group, so if the query does not overlap with that volume, we can rule out all that subtree and thus all the objects in that subset. If the query does overlap, then we recurse down that subtree checking further subtrees and eventually leaf objects if needed. For construction of HBVs many methods have been put forward, and in general they are heuristics designed to decrease the total volume of the bounding boxes as much as possible at each level in the tree, or in other words, to look for splits resulting in the tightest fitting volumes. This avoids the exponential time required for finding an optimal fit. Some of the most common heuristic methods for constructing HBVs are the same as for spatial partitioning trees – namely selecting a splitting plane based on some criteria and building the subtrees recursively. As was mentioned in the previous section, example pseudo-code for constructing this type of spatial/HBV tree variant is given in Table 3.2.

3.2 Approach to Collision Checking

3.2.1 Extent Masks

The *extent masks* approach is a novel method for broad-phase collision detection that offers exact bounding box queries and consistent performance suitable for a real-time planner. It

```
struct Plane
var p : Point
var n : Vector
end

struct SpatialTree
var left, right : SpatialTree
var bbox : AABBox
var objects : ObjectList
end

function Split(objects:ObjectList,plane:Plane) : (ObjectList,ObjectList)
1    var l, r : ObjectList
2    foreach o:Object ∈ objects do
3        if (o.cen - plane.p) · plane.n > 0
4            then r ← r ∪ {o}
5            else l ← l ∪ {o}
6    return (l,r)

function MakeLeaf(objects:ObjectList) : SpatialTree
1    var n : SpatialTree;
2    n.bbox ← EmptyBBox
3    foreach o:Object ∈ objects do
4        n.bbox ← Union(n.bbox,o.bbox)
5    n.objects ← objects
6    return n

function MakeInterior(objects:ObjectList) : SpatialTree
1    let plane = ChooseSplitPlane(objects)
2    let (l, r) = Split(objects,plane)
3    var n : SpatialTree;
4    n.left ← MakeSpatialTree(l)
5    n.right ← MakeSpatialTree(r)
6    n.bbox ← Union(n.left.bbox, n.right.bbox)
7    return n

function MakeSpatialTree(objects:ObjectList) : SpatialTree
1    if ListLength(objects) < LeafSize
2        then return MakeLeaf(objects)
3        else return MakeInterior(objects)
```

Table 3.2: Algorithm for building a spatial tree with per-node bounding boxes

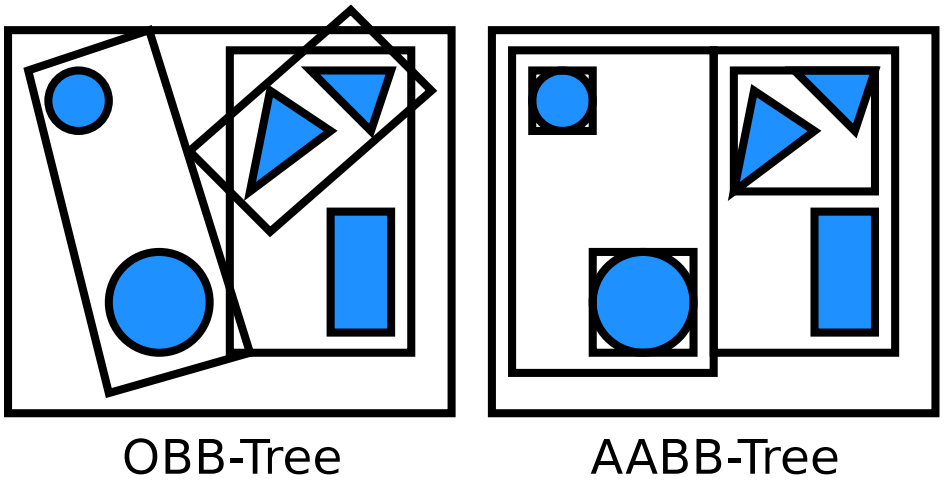


Figure 3.2: Examples of several space partitioning hierarchies

leverages the parallelism of bit operations on modern computers for speed, taking advantage of the common case of the relatively few dynamic obstacles present in many domains. First, we can define a mask function for the x axis as $m_x(x, i) = \{0, 1\}$, and analogously for the other axes. It is 1 if the object overlaps on that axis at location x, and 0 otherwise. An example domain with three obstacles is shown in Figure 3.3, with a color-coded mask functions shown along the x and y axes. To perform a query, we want to find the set of all "ones" for m at a given location x, and then integrate that function between two values x_0 and x_1 to get a union set of all obstacles overlapping a range, as shown in Figure 3.4. We then want to take the union of the sets from each axis to get the overall set of overlapping axis-aligned bounding boxes. Thus for the algorithm to work, we need thus primitives for taking the union along an axis, and intersection between axes.

To represent the mask function, we first use binary integers to represent the value of the function at any axis location x, with the i'th binary digit (bit) representing the value of $m_x(x, i)$. We refer to this binary integer representing a set as a *bitmask*. Next, noting that the function only changes at x values where an obstacle begins or ends (its extents), we can compactly represent the function with an array of x values and the associated binary integer to represent the set of overlapping obstacles. Finally, to allow for taking fast unions, we represent the function m in a cumulative fashion along x, recording all obstacles that have started before a value of x, and all that have ended before a value of x. Table 3.4 gives the representation and implementation in pseudo-code for creating extent masks. First in the function *MaskSignalAdd*, objects are added to the array unsorted, with entries for the beginning and ending extents where only the bit for that obstacle is set. The function

```
function CheckBox(tree:SpatialTree, q:Query) : Status
1    if Overlap(tree.bbox,q.bbox)
2        then if tree.objects ≠ {}
3            then foreach o:Object ∈ tree.objects
4                do if Overlap(o.bbox, q.bbox) ∧ NarrowPhase(o,q) ≠ Free
5                    then return Collision
6            else if CheckBox(tree.left,q) = Free ∧ CheckBox(tree.right,q) = Free
7                then return Free
8                else return Collision
9        else return Free
```

Table 3.3: Algorithm for checking a spatial tree with per-node bounding boxes

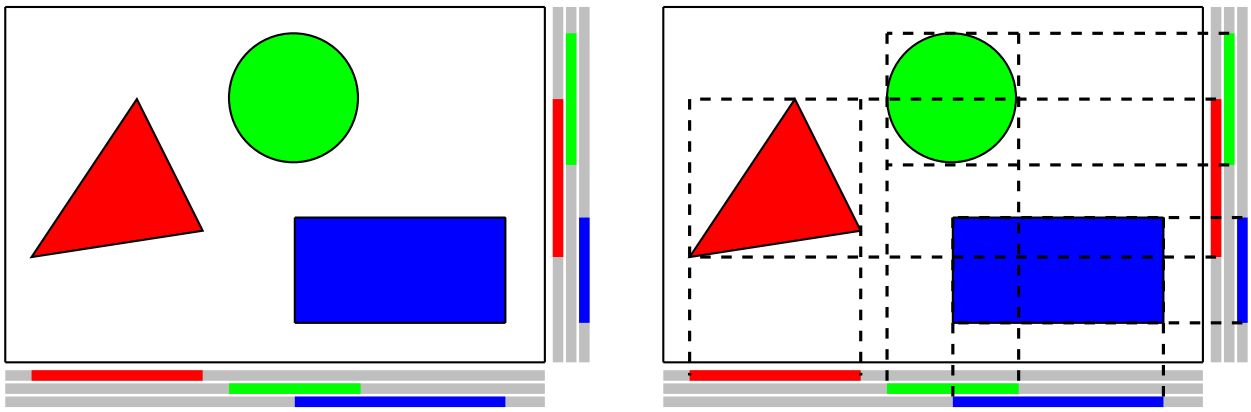


Figure 3.3: Example environment with the corresponding projected extent masks

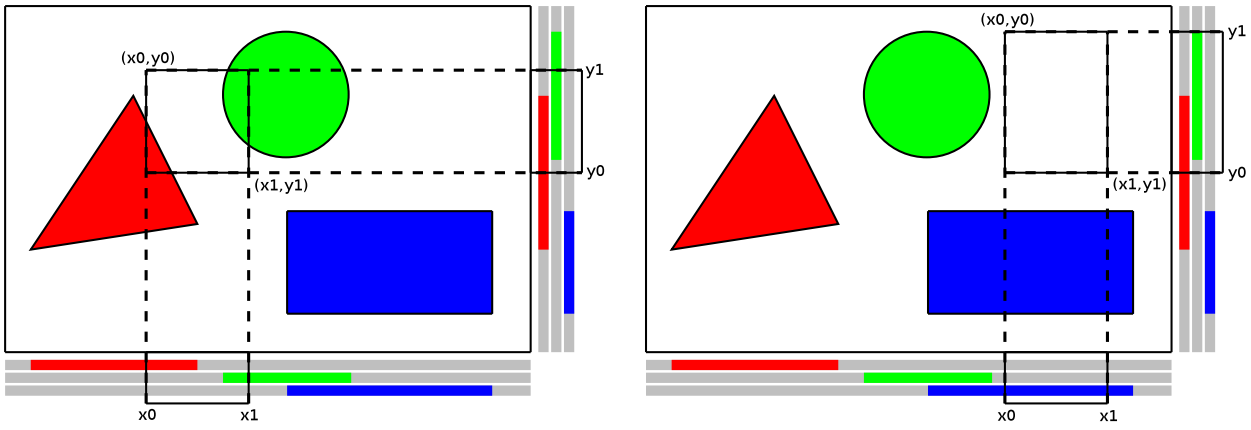


Figure 3.4: Two broad-phase bounding box checks using projected extent masks

MaskSignalSetup first sorts the array by axis value, and then takes the union of the start and end sets using a bitwise-or operation on the binary number representation. In Table 3.5, we can see how to query the extent mask efficiently. A standard binary search is used to find the correct entries for the start and end of a range, and bitwise operations are again used to extract all objects that have a starting extent by the end of the range, but which do not have an ending extent before the beginning of the range. This corresponds exactly to the set of obstacles overlapping the range. Extending the algorithm to multiple axes is trivial, as we need only query each axis and take the union using a bitwise-and to combine the results of *MaskSignalGet* from each axis.

The efficiency of the extent masks approach relies critically on the speed of the bitwise operations on the obstacle sets represented is integers. However, if the bitmask is represented by a 32-bit or 64-bit unsigned integer on the machine, all of these bitwise operations can be carried out in a single machine instruction. Thus as long as the number of obstacles is smaller than the number of bits the machine can use to represent an integer, those operations are constant time. Combined with the binary search, this leads to a complexity of $O(\log n)$ for one axis and $O(d \log n)$ for d axes. Of course, to scale beyond the word size of the machine integers, we must employ arrays of machine integers leading to an ultimate complexity of $O(dn)$, but this linear component is quite small in practice, and many domains do not require more than 64 dynamic obstacles. As we show later in this chapter, complex static geometries can be collected into a single large obstacle at the broad-phase collision level, so the limitation is not particularly restrictive in practice.

3.2.2 Heuristically Balanced AABB tree

In cases where large numbers of obstacles were needed, and static lookup structures were not appropriate, another method based on HBVs was implemented. It uses the algorithm of Table 3.2, constructing a tree of axis-aligned bounding boxes using splitting planes to build the tree in an efficient top-down fashion. The resulting tree does not guarantee logarithmic access times, but performs well in practice for relatively uniform distribution of obstacles. The method used for the *ChooseSplitPlane* function is to try a set of candidate splitting planes, evaluated with the function *EvaluateSplitPlane* listed in Table 3.6. The candidate planes are located at the median location of all the current objects, and oriented along each possible axis.

The evaluation function for splitting planes is a heuristic guided on two main principles:

- Queries are uniform, thus subtree being descended is proportional to its area

```
struct MaskSignalEntry
var x : ℝ
var start,end : BitMask
end

struct MaskSignal
var a : array[MaxObjects*2] of MaskSignalEntry
var n : ℤ
end

function MaskSignalAdd(m:MaskSignal, x0,x1:ℝ, i:ℤ)
1    m.a[n+0].x ← x0
2    m.a[n+0].start ← BitMaskSingleBit(i)
3    m.a[n+0].end ← 0
4    m.a[n+1].x ← x1
5    m.a[n+1].start ← 0
6    m.a[n+1].end ← BitMaskSingleBit(i)
7    m.n ← m.n + 2

function MaskSignalSetup(m:MaskSignal)
1    Sort(m.a, m.n)
2    var s,e : BitMask
3    s ← 0
4    e ← 0
5    for i = 0 to n-1 do
6        s ← BitwiseOr(s, m.a[i].start)
7        e ← BitwiseOr(e, m.a[i].end)
8        m.a[i].start ← s
9        m.a[i].end ← e
```

Table 3.4: Algorithm for building a mask signal

```
function BinarySearch(m:MaskSignal, x:ℝ, l,r:ℤ) : ℤ
1    while r - l > 1
2        let c = ⌊ (l + r) / 2 ⌋
3        if x < m.a[c].x
4            then r ← m
5            else l ← m

function MaskSignalGet(m:MaskSignal, x_0,x_1 : ℝ) : BitMask
1    let l = BinarySearch(m,x_0,0,m.n)
2    let r = BinarySearch(m,x_1,0,m.n)
3    return BitwiseAnd(m.a[r].start, BitwiseNegate(m.a[r].end))
```

Table 3.5: Algorithm for querying a mask signal

```
function EvaluateSplitPlane(objects:ObjectList, p:Plane) : ℝ
1    var num : array[2] of ℤ
2    var bbox : array[2] of AABBox
3    var s : ℤ
4    num[0] ← 0
5    num[1] ← 0
6    bbox[0] ← EmptyBBox
7    bbox[1] ← EmptyBBox
8    foreach o:Object ∈ objects do
9        if (o.cen - plane.p) · plane.n > 0
10           then s ← 1
11           else s ← 0
12       num[s] ← num[s] + 1
13       bbox[s] ← Union(bbox[s],o.bbox)
14   return Area(bbox[0])*num[0] + Area(bbox[1])*num[1]
```

Table 3.6: Our method to evaluate a splitting plane for building a hierarchical bounding box representation for collision checking.

(2D) or volume (3D)

- The worst case complexity for accessing a subtree is linear

The evaluation metric is thus to consider a split by calculating the resulting bounding boxes and number of objects in each subtree, and estimating the worst-case expected time as the sum of the areas of the bounding boxes multiplied by the number of objects in each subtree. If the splitting plane is chosen with the smallest cost evaluation value, then the tree greedily minimizes the expected worst-case running time for accesses. This is of course subject to the limitations of choosing from a small number of axis-aligned splitting planes, and thus does not minimize the running time in any global way for all the possible partitionings. However it tends to guide the tree building procedure to construct a tree balanced enough to support fast accesses as is shown in Section 3.2.4.

3.2.3 Narrow-Phase Collision Checking

In comparison to broad-phase collision detection, narrow-phase collision detection in our system is quite a bit more specialized in its approach. Much of the structure is dictated by our circle/sphere queries and the geometry of the primitive objects rather than adherence to an approach from the literature on collision checking. The primitive queries supported by obstacles our system are the following:

1. Checking a point q and radius r for collision
2. Checking a swept circle or sphere from q_0 to q_1 and radius r
3. Calculating the distance from q to the nearest point on an obstacle
4. Calculating the nearest point p on the obstacle to a query point q
5. Optionally calculating two tangents to the obstacle from a point q (this allows local avoidance heuristics in the planner)

In the originally developed 2D implementation, each of the above functions was implemented for maximum efficiency, although this required a larger amount of work than necessary for adding new obstacle types. In particular, moving to 3D complicated the geometry for swept queries significantly. This motivated a two-tiered approach where default implementations were provided for all but the nearest point queries, and a particular obstacle could override the default implementations with a more efficient specific version if needed. The virtual

method feature of C++ made this relatively easy to implement; Default versions of most functions were provided in the base class. In particular, the default implementations used the following properties:

- Calculating the distance to an object depends only on the distance from the query q to the nearest point on the obstacle
- Checking a point q can simply check the distance against r
- Checking a swept query can be implemented iteratively using only the distance query

While the methods for point queries are obvious, the method used for checking a swept circle or sphere requires more explanation. A visualization of the approach appears in Figure 3.5. For any given point along a swept trajectory, the current distance to obstacles, or clearance, determines how far along a trajectory is safe. The checker can then step forward by that distance, and recursively check the remaining swept area. The figure shows a dark blue trajectory to be checked, the light blue is the current clearance, and the red spheres show the steps that can be safely taken each iteration. Pseudo-code for the approach can be found in Figure 3.7 Normally, few iterations are required, although the algorithm can take many steps if it is very close to an obstacle. This is handled by failing after a certain number of iterations have been exceeded. Though this is yet another pessimistic approximation, long paths running very close to obstacles are not typically desirable for execution by mobile robots anyway.

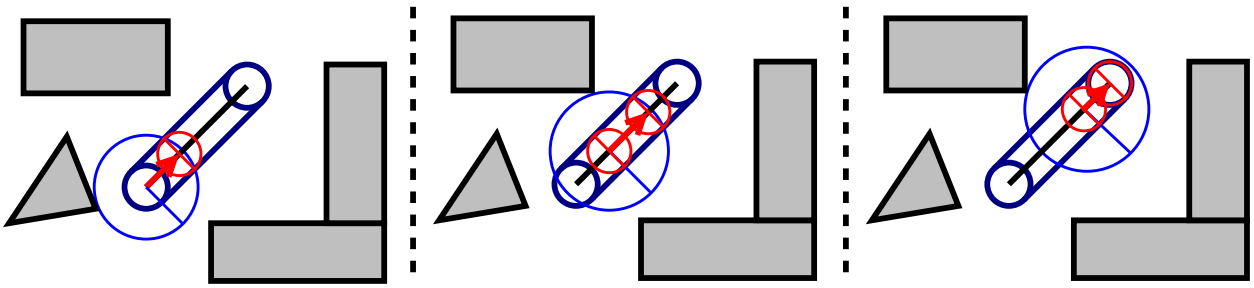


Figure 3.5: Example swept circle obstacle check using only distance queries.

While in our current implementation, only line-swept-spheres are supported, the distance query stepping method of checking a swept sphere can be applied to other trajectory functions as well. For some continuous trajectory function $x(t)$ where time $t \in [0, t_f]$, starting from an initial position $x(0)$ with a distance of at least $D(x(0)) = d$ from all obstacles, we need only find the first t such that $\|x(t) - x(0)\|^2 = d^2$, or verify that no such t exists for $t \in [0, t_f]$. For functions with bounded curvature, such as lines and circular arcs, these calculations are

```
function CheckSweptQuery(q_0, q_1 : Point, r : ℝ) : bool
1    let l =Dist(q_0, q_1);
2    let d = (q_1 − q_0)/l;
3    var s : ℝ;
4    s ← 0;
5    while s < l do
6        let q = q_0 + ds
7        let f = CalcFreeDist(q);
8        if f < r
9            then return Collision
10           else s ← s + √(f^2 − r^2)
11   return Free;
```

Table 3.7: Algorithm for checking a swept query based on distance

straightforward. In particular, for a linear trajectory defined by $x(t) = a + bt$, then the solution is $t_i = \sqrt{d^s - r^2}/\|b\|$. If $t_i > t_f$, then the trajectory is verified to be free, otherwise a recursive check must be made with a new trajectory starting at time t_i. Thanks to the square root, t_i increases quite rapidly from zero even with very small clearances, which is the reason few steps are normally needed for a typical check, and results in the approach being quite efficient in practice.

The following primitive obstacles are implemented by our collision checker:

- Axis-aligned rectangles
- Circles/spheres
- Convex polygons
- Regular 2D grids
- 3D terrain sampled on a regular grid (elevation map)
- Large 3D polygon soups

The complexity of the individual obstacles ranges greatly, from simple geometric shapes such as circles and rectangles all the way up to 200x200 terrain meshes and polygon soups. While the geometric shapes are straightforward to develop a distance metric for, the terrain obstacle posed the challenge of efficient distance calculation. Since the terrain is a 2D grid projected into 3D as a low-curvature mesh (i.e. a relatively flat manifold), the approach taken was to break it up using a 2D K-D tree in grid space, and then to bound the individual nodes

with an axis-aligned bounding box in the 3D space. The first few levels of a terrain tree are shown in Figure 3.6. To query the distance from a point to the terrain, we follow the branches of the KD tree nearest first, calculating a distance to the actual terrain once a leaf node is reached. After that, the remainder of the traversals and be compared against this known minimum and pruned if the bounding box is further than the current minimum. This aggressive pruning and the relatively flat nature of actual terrain meshes yields near logarithmic access times for the queries. The nodes descended during a particular distance query are shown in Figure 3.7. A similar approach was taken for polygon soups, where a large static set of triangles are stored in a KD tree with per-node bounding boxes. Planes splitting normals were determined by the largest axis, and the plane splitting locations were set at the median of that axis. Due to the terrains being static, the lookup structures could be calculated once for each object when loading the domain, and reused in many iterations of the planner. Construction times for the lookup structure were minimal compared to the time required to load the terrain from the disk.

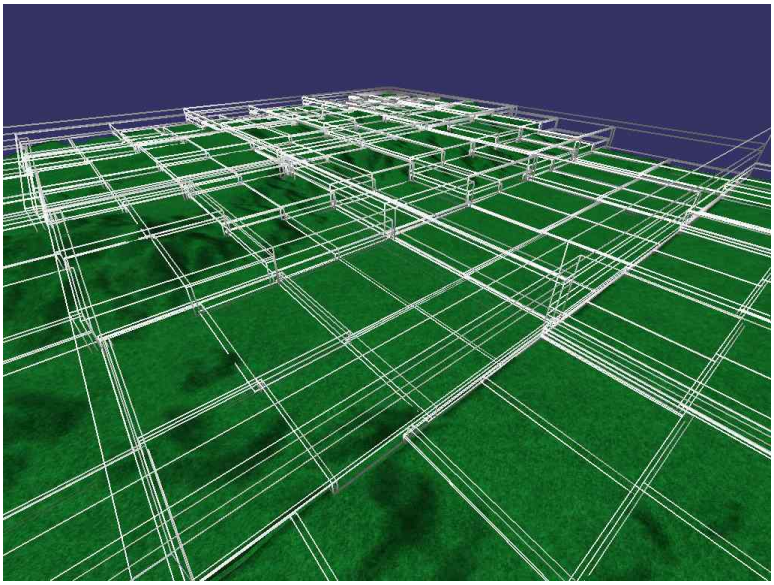


Figure 3.6: The first levels of bounding boxes on a terrain K-D tree.

3.2.4 Performance Measurement

Although we are ultimately interested in the performance of a motion planner as a whole, we can compare collision detection methods directly to get an idea of what sort of performance to expect. We will compare collision detection using three broad phase methods along

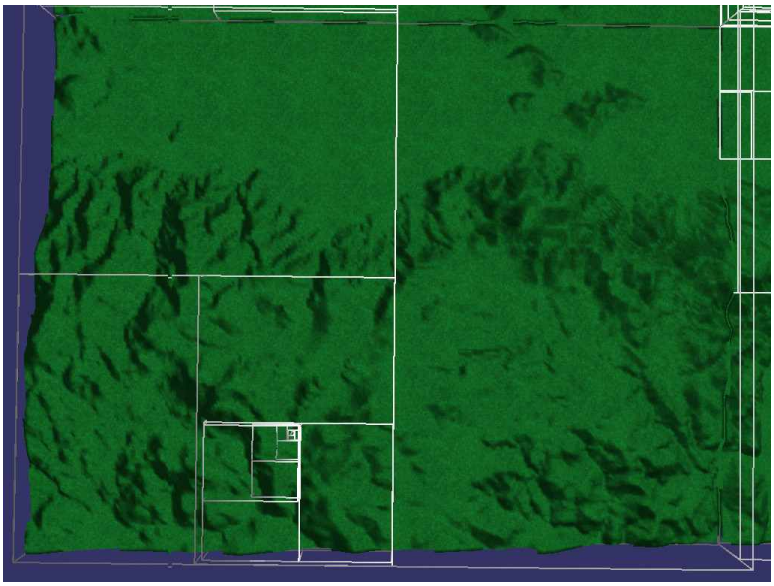


Figure 3.7: Nodes expanded on an example terrain distance query.

with our narrow phase approach, but we will limit our comparison to the two environments represented by Figure 3.8, called *circles* and *rectangles*. Each environment represents an enclosed 5.5m by 4m with 64 randomly placed obstacles. In addition, a variant of the circles domain was created with up to 256 circles with half the expected radius (thus covering approximately the same fraction of the workspace). A radius of 90mm was selected for each of the queries, thus representing an environment similar to the RoboCup small-size domain.

We first tested the mask extents implementation on the circles domain to establish the type of per-query run-time distribution. The probability density function is shown in Figure 3.9. Three variants are shown, one each for cases of a detected collision or a no-collision case, as well as the average case for the environment.

The following observations can be made of the collision query time probability density function are the following.

- The query time averages only 0.375 *microseconds*.
- The distribution is approximately Gaussian, and thus can be reasonably summarized by means and confidence intervals.
- The query time varies depending on whether the query location is free or not, however the difference is not so large that it is likely to have a significant effect on callers.

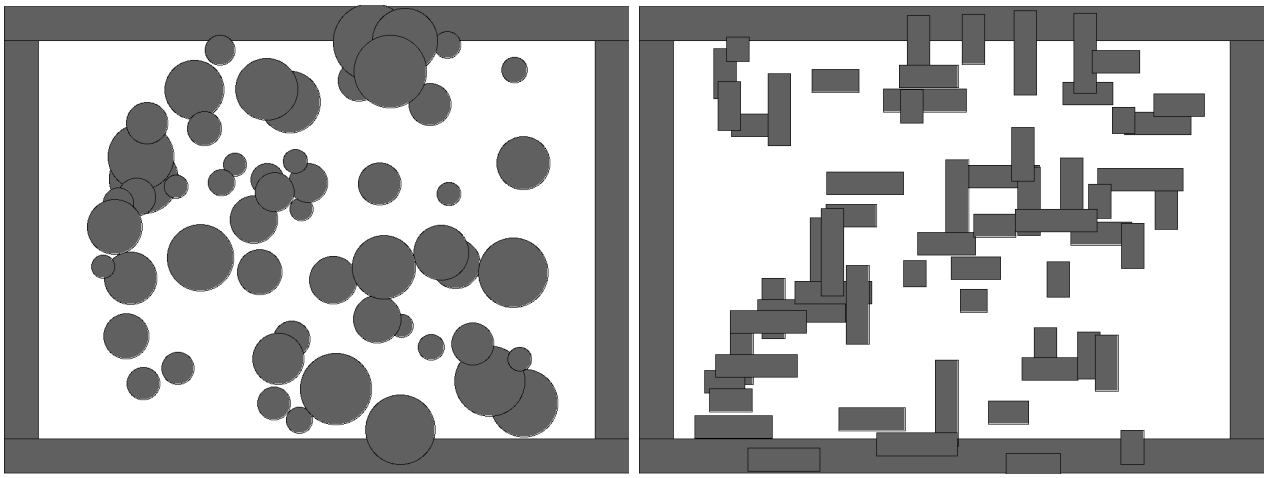


Figure 3.8: Two domains with 64 obstacles used as collision checking benchmarks.

After establishing that queries are indeed quite fast (over 2.5 million per second), the next question then becomes how well this speed scales with the number of obstacles in the domain. To examine this scalability we compared three algorithms; The extent masks approach, the AABB tree method, and a baseline linear array method that simply checks the obstacles one at a time in array order. The scaling tests are based on the 256-circle domain, but with only the first n obstacles included for each datapoint as n was varied from 0 to 256 obstacles. Figure 3.10 shows how extent masks scales. The major features are that the scaling is sublinear, that the confidence interval is reasonable narrow, and that it is limited to only 64 obstacles due to the machine representation of bitmasks. In Figure 3.11, we can see the scaling of the bounding volume tree approach. Note here that the variance is significantly more than for extent masks, however the average performance is superior. The AABB tree scales extremely well up to 256 obstacles, with a very sublinear trend on both the average case and the 95% (upper) confidence bound. The AABB tree also displays some non-monotonic variability based on the number of obstacles, which can be attributed to the different trees that are constructed from the different subsets of obstacles. Finally, the linear array baseline implementation is shown in Figure 3.12. The 95% confidence bound scales linearly, while the average case is improved due to short-circuiting on queries which hit an obstacle. The variance increases steadily with time, also due to the short circuiting.

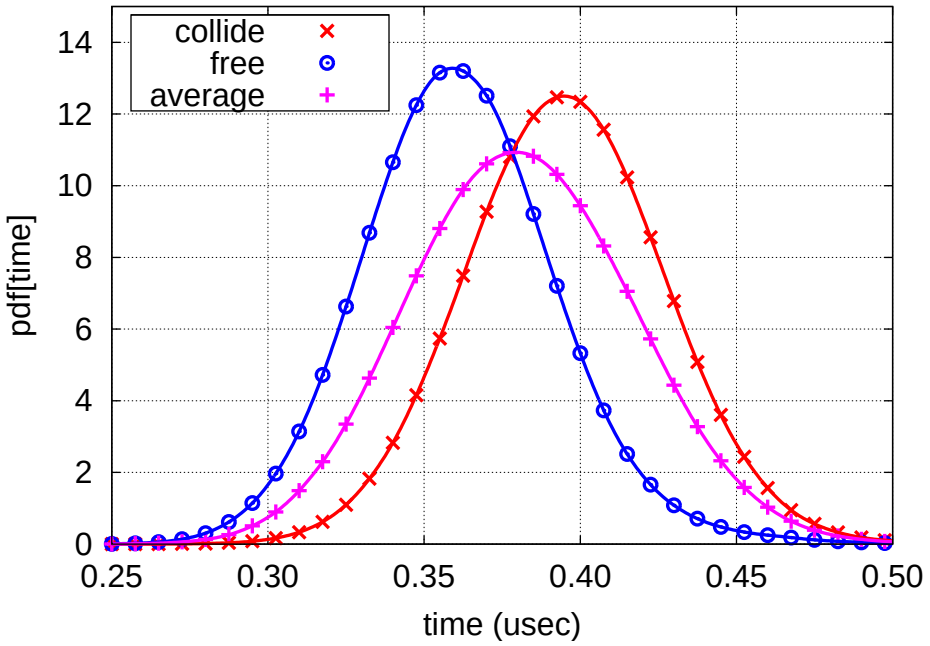


Figure 3.9: Collision query time distribution with 64 circular obstacles.

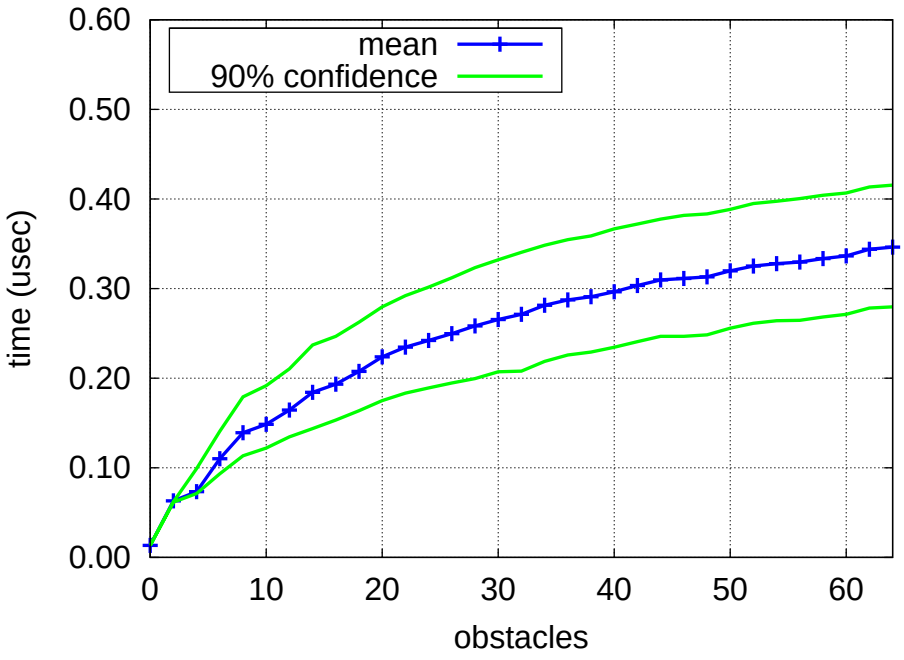


Figure 3.10: Scaling of collision queries with obstacles for extent masks

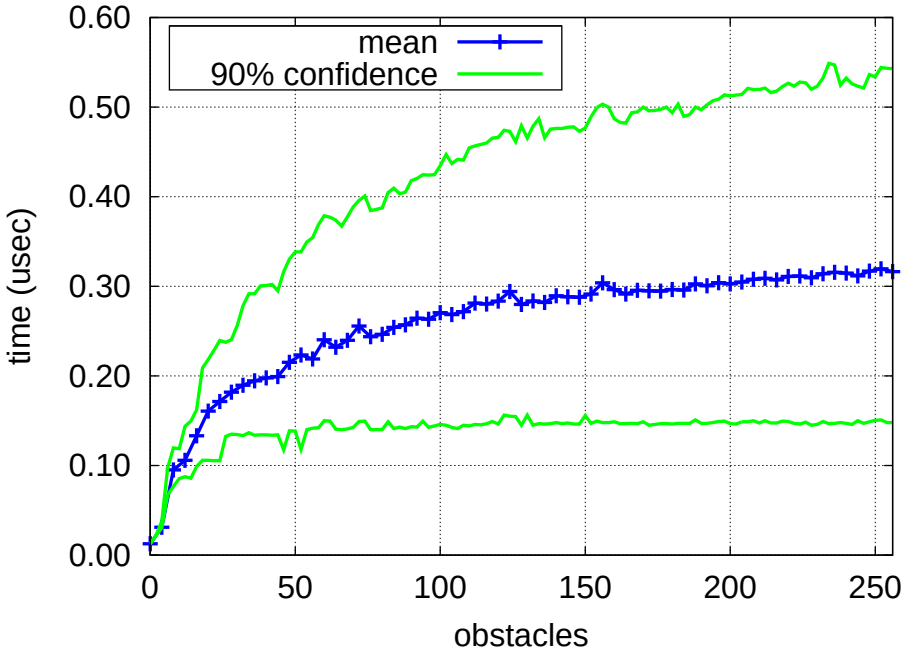


Figure 3.11: Scaling of collision queries with obstacles for AABB tree

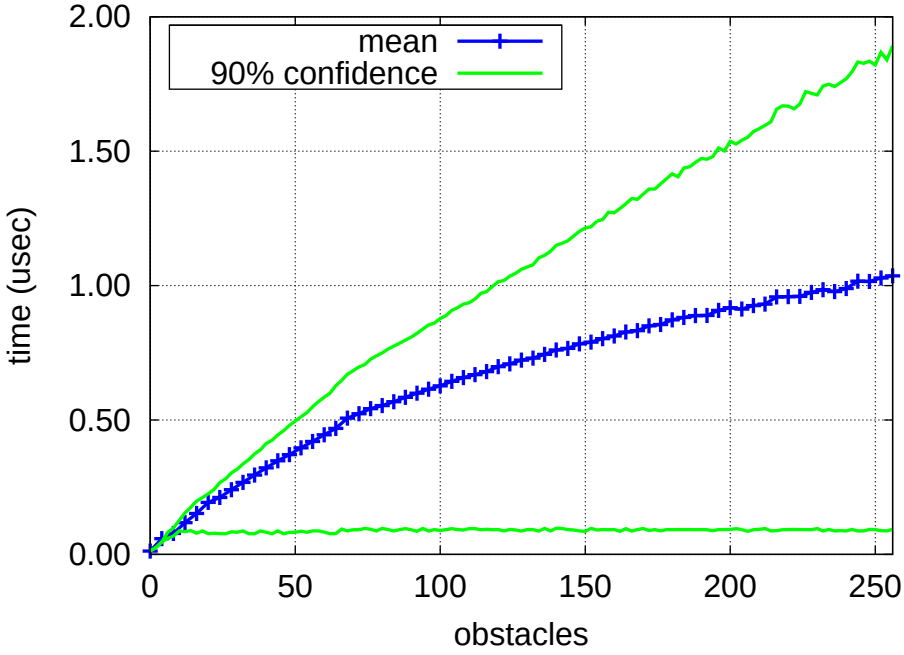


Figure 3.12: Scaling of collision queries with obstacles for a linear array

3.3 Summary

This chapter explored common methods for collision detection, and described the approaches taken for collision detection in this thesis. Broad-phase methods include the low-variance extent masks approach which takes advantage of the machine parallelism, and a heuristic for building axis-aligned bounding box hierarchies (AABB trees). The narrow phase uses a modular approach to allow easy implementation of new obstacle types using only a distance query, while numerous obstacle geometries are supported for narrow-phase collision detection. The collision detection method works very quickly for environments of moderate complexity, resulting in collision query times of less than half a microsecond.

Chapter 4

Motion Planning

4.1 Approach

In our domains (see 2), each environment is a mix of static and dynamic obstacles:

- The **RoboCup small-size** domain contains up to 10 agents moving at a given time, with five agents not controlled by our system. The teams in the environment are operating adversarial with conflicting goals.
- The **fixed-wing UAV** domain contains a static terrain paired with a variable number of geometric obstacles reflecting constraints that can change per-query.
- The **humanoid** domain contains an occupancy grid model which is being continuously updated, and thus it nearly always changes between queries. Much of the grid is highly correlated across queries however.

Thus, in each case unmodelled or unpredictable elements are modifying the environment. Dynamic elements, such as the trajectory of opponent robots, are not accurately predictable beyond a short time in the future. The result is that we cannot apply system which assumes a known future trajectory for moving obstacles, but instead one that allows new decisions to be made each control cycle. In addition, we assume no notification of which objects have moved or how their geometry has been altered. While in some domains, this information may be available, other domains may not be able to recover this information efficiently. The factors of the dynamic and unpredictable environment led to the choice of an iterative approach to replanning, were replanning is unconditional as new state information arises each cycle.

In addition, the geometry and dimensionality of the obstacles varies between domains. If we desire a single planning framework that works for all of the domains, this limits our choices of applicable planners. Of the available options (see 6), we have chosen randomized approaches to motion planning, which require no explicit model of their environment. Instead, the configuration space is discovered experimentally through queries to a collision detection system.

The two methods we have chosen to extend are RRT and PRM. We have developed a single planning framework which supports a derivative of RRT, called ERRT, as well as an incremental variant of PRM. As a result of the combination as a single parameterized method, we have removed some of the unnecessary distinctions between the algorithms. The resulting parameterized method can be tuned to act as the method most suitable for a given problem domain.

4.1.1 Contributions of this Chapter

- The waypoint cache, and its associated point distribution, are novel. It is introduced as part of the author's Execution Extended RRT (ERRT) algorithm, which extends the RRT algorithm of LaValle and Kuffner [65]
- The ERRT algorithm is tested on a physical robot platforms for a the RoboCup small-size team. As far as the author is aware, this is the first application of an RRT-derived planner to a physical robot platform (fall 2001), and the longest-running use at 5 years (It has been used in all subsequent RoboCup experiments and applications.)
- Bidirectional Multi-Bridge ERRT introduces novel parameters to the Kuffner's RRT-Connect algorithm [51]. These new parameters allow a tradeoff between planner efficiency and plan length optimality.
- The novel Dynamic PRM algorithm is introduced, which extends Kavraki et al.'s Probabilistic Roadmap (PRM) algorithm [54]. It is tested on the QRIO humanoid robot.
- This chapter identifies the requirements for robust motion planning for robotics applications, and in particular robustness to location error. The author is not aware of these requirements or their solutions being identified in existing work on randomized or graph-based motion planners.
- The efficient heuristic of an *active goal* is introduced to solve kinematically constrained planning problems where the goal is defined as a fixed-radius orbit around a point. It is demonstrated as part of a planner for fixed-wing unmanned

aerial vehicles (UAVs).

4.2 Existing Planning Methods

4.2.1 RRT Algorithm in Detail

In essence, an RRT planner searches for a path from an initial state to a goal state by expanding a search tree. For its search, it requires the following three domain-specific function primitives:

function *Extend* (env:Environment, current:State, target:State):State
function *Distance* (current:State, target:State):Real
function *RandomState* ():State

First, the *Extend* function calculates a new state that can be reached from the target state by some incremental distance (usually a constant distance or time), which in general makes progress toward the goal. If a collision with an obstacle in the environment would occur by moving to that new state, then a default value, EmptyState, of type state is returned to capture the fact that there is no "successor" state due to the obstacle. In general, any heuristic methods suitable for control of the robot can be used here, provided there is a reasonably accurate model of the results of performing its actions. The heuristic does not need to be very complicated, and does not even need to avoid obstacles (just detect when a state would hit them). However, the better the heuristic, the fewer nodes the planner will need to expand on average, since it will not need to rely as much on random exploration. Next, the function *Distance* needs to provide an estimate of the time or distance (or any other objective that the algorithm is trying to minimize) that estimates how long repeated application of *Extend* would take to reach the goal. Finally, *RandomState* returns a state drawn uniformly from the state space of the environment.

For a simple example, a holonomic point robot with no acceleration constraints can implement *Extend* simply as a step along the line from the current state to the target, and *Distance* as the Euclidean distance between the two states. Table 4.1 shows the complete basic RRT planner with its stochastic decision between the search options:

- with probability p, it expands towards the goal minimizing the objective function *Distance*,

- with probability $1 - p$, it does random exploration by generating a *RandomState*.

The function *Nearest* uses the distance function implemented for the domain to find the nearest point in the tree to some target point outside of it. *ChooseTarget* chooses the goal part of the time as a directed search, and otherwise chooses a target taken uniformly from the domain as an exploration step. Finally, the main planning procedure uses these functions to iteratively pick a stochastic target and grow the nearest part of the tree towards that target. The algorithm terminates when a threshold distance to the goal has been reached, though it is also common to limit the total number of nodes that can be expanded to bound execution time.

4.2.2 PRM Algorithm in Detail

Probabilistic Roadmap (PRM) planners take a related but alternative approach to planning compared to RRT. The main motivation for using PRM is that it allows planing to be split into two phases, the first of which can be reused in a static environment. The two stages are the following:

Learning Phase where a finite graph model, or roadmap, is constructed up to approximate the geometry of the free configuration space.

Query Phase where the roadmap is extended with a particular problem instance and searched for a free path.

An implementation of this variant of PRM can be found in 4.2. The learning phase proceeds by adding points randomly distributes within the environment, and then attempting to find pairs of nodes which can be reached with a local planner. If the local planner succeeds, an edge is added to the roadmap graph. This is implemented in the function *PRMLearn*, which first calls the function *AddStates* to sample random states from the environment. *AddStates* only adds states which are in C_{free}. Next, *PRMLearn* calls *ConnectStates*, which attempts to connect each state with its neighbors using the function *Connect*.

The function *Connect* is in a sense the core of the PRM planner, much like *Extend* for RRT. This implementation find finds all other states within a certain maximum distance (line 1), and attempts a straight-line connection with those neighboring states (line 3-4). Many variants of *Connect* exist in PRM literature, in particular by varying the neighbors

```
function RRTPlan(env:Environment, initial:State, goal:State) : RRTTree
1    var nearest,extended,target : State
2    var tree : RRTTree
3    nearest ← initial
4    tree ← initial
5    while Distance(nearest,goal) < threshold do
6        target ← ChooseTarget(goal)
7        nearest ← Nearest(tree,target)
8        extended ← Extend(env,nearest,target)
9        if extended ≠ EmptyState
10           then AddNode(tree,extended)
11   return tree

function ChooseTarget(goal:State) : State
1    var p : ℝ
2    p ← UniformRandom(0,1)
3    if p ∈ [0, GoalProb]
4        then return goal
5    else if p ∈ [GoalProb, 1]
6        then return RandomState()

function Nearest(tree:RRTTree,target:State) : State
1    var nearest : State;
2    nearest ← EmptyState;
3    foreach State s ∈ tree do
4        if Distance(s,target) < Distance(nearest,target)
5            then nearest ← s;
6    return nearest;
```

Table 4.1: The basic RRT planner stochastically expands its search tree to the goal or to a random state.

chosen or the local planner which is used [6, 49]. The implementation is the earliest variant as described in Kavraki et al. [53, 54].

The query phase for PRM is shown in the function *PRMQuery*. It begins by adding the initial and goal states to the roadmap, and then proceeds with a graph search to find a path on this augmented roadmap. The function *GraphSearch* can be implemented using A* search in order to find the shortest length path of those that exist in the roadmap graph.

One aspect of note for the two-phase PRM is that its completeness depends on a parameter n reflecting the number of samples with which to model the environment. This means the planner may fail in the query phase, and does not attempt any further search. Early work in PRM recognized this issue, and for cases where one wishes to continue planning until a solution is found, a one-shot version of PRM was developed. In Table 4.3, a one-shot PRM variant which continues expanding the search is shown. It simple alternates a learning phase with a search phase, increasing the number of samples added at each stage by a factor *ExpandFactor* which is greater than one. The sub-functions are all reused from the two-phase PRM planner.

4.3 Execution Extended RRT (ERRT)

Some optimizations over the basic described in existing work are bidirectional search to speed planning, and encoding the tree's points in an efficient spatial data structure [4]. In this work, a KD-tree was used to speed nearest neighbor lookup, but bidirectional search was not used because it decreases the generality of the goal state specification (it must then be a specific state, and not a region of states). Additional possible optimizations include a more general biased distribution, which was explored in this work in the form of a waypoint cache. If a plan was found in a previous iteration, it is likely to yield insights into how a plan might be found at a later time when planning again; The world has changed but usually not by much, so the history from previous plans can be a guide. The waypoint cache was implemented by keeping a constant size array of states, and whenever a plan was found, all the states in the plan were placed into the cache with random replacement. This stores the knowledge of where a plan might again be found in the near future. To take advantage of this for planning, Table 4.4 shows the modifications to the function *ChooseTarget*.

With ERRT, there are now three probabilities in the distribution of target states. With probability $P[goal]$, the goal is chosen as the target; With probability $P[waypoint]$, a random waypoint is chosen, and with the remaining probability a uniform state is chosen as before. Typical values used in this work were $P[goal] = 0.1$ and $P[waypoint] = 0.6$. Another

```
type StateSet = set of State
type EdgeSet = set of Edge
tuple RoadMap = (StateSet * EdgeSet)

function AddStates(env:Environment, V:StateSet, n:ℤ) : StateSet
1    for i = 1 to n
2        q ← RandomState();
3        if CheckObs(env,q)
4            then V ← V + s
5    return V

function Connect(env:Environment, (V, E):RoadMap, q:State) : EdgeSet
1    L = NearbyStates(V,q,MaxDist)
2    foreach s ∈ L do
3        if CheckObsLine(env,q,s)
4            then E ← E + (q, s)
5    return E

function ConnectStates(env:Environment, G:RoadMap) : EdgeSet
1    foreach q ∈ V do
2        E ← Connect(env,G,q)
3    return E

function PRMLearn(env:Environment, num:ℤ) : RoadMap
1    var (V, E) : RoadMap
2    V ← AddStates(env,∅,num)
3    E ← ConnectStates(env,(V,∅))
4    return (V, E)

function PRMQuery(env:Environment, initial:State, goal:State) : Plan
1    V ← V + {initial,goal}
2    E ← Connect(env,(V, E),initial)
3    E ← Connect(env,(V, E),goal)
4    P ← GraphSearch((V,E),initial,goal)
```

Table 4.2: A basic implementation of a PRM planner

```
function PRMOneShot(env:Environment, initial:State, goal:State) : Plan
1    var (V, E) : RoadMap
2    var P : Path
3    V ← {initial,goal}
4    E ← ∅
5    repeat
6        let n = Max(Size(V) · ExpandFactor, MinNumAddStates)
7        V ← AddStates(env,V,n)
8        E ← ConnectStates(env,(V, E))
9        P ← GraphSearch((V, E),initial,goal)
10   until P ≠ ∅
11   return P
```

Table 4.3: The code for a one-shot PRM planner

```
function ChooseTarget(goal:State) : State
1    let p = UniformRandom(0,1)
2    let i = RandomInt(0,NumWaypoints-1)
3    if p ∈ [0, GoalProb] then
4        return goal
5    else if p ∈ [GoalProb, GoalProb + WaypointProb] then
6        return WaypointCache[i]
7    else if p ∈ [GoalProb + WaypointProb, 1] then
8        return RandomState()
```

Table 4.4: The extended RRT planner chooses stochastically between expanding its search tree to the goal, to a random state, or to a waypoint cache.

extension was adaptive beta search, where the planner adaptively modified a parameter to help it find shorted paths. A simple RRT planner is building a greedy approximation to a minimum spanning tree, and does not care about the path lengths from the initial state (the root node in the tree). The distance metric can be modified to include not only the distance from the tree to a target state, but also the distance from the root of the tree, multiplied by some gain value. A higher value of this gain value (beta) results in shorter paths from the root to the leaves, but also decreases the amount of exploration of the state space, biasing it to near the initial state in a "bushy" tree. A value of 1 for beta will always extend from the root node for any Euclidean metric in a continuous domain, while a value of 0 is equivalent to the original algorithm. The best value seems to vary with domain and even problem instance, and appears to be a steep tradeoff between finding an shorter plan and not finding one at all. However, with biased replanning, an adaptive mechanism can be used instead that seems to work quite well. When the planner starts, beta is set to 0. Then on successive replans, if the previous run found a plan, beta is incremented, and decremented otherwise. In addition the value is clipped to between 0 and 0.65. This adaptive bias schedule reflects the idea that a bad plan is better than no plan initially, and once a plan is in the cache and search is biased toward the waypoints, nudges the system to try to shorten the plan helping to improve it over successive runs.

4.3.1 Testing ERRT for RoboCup

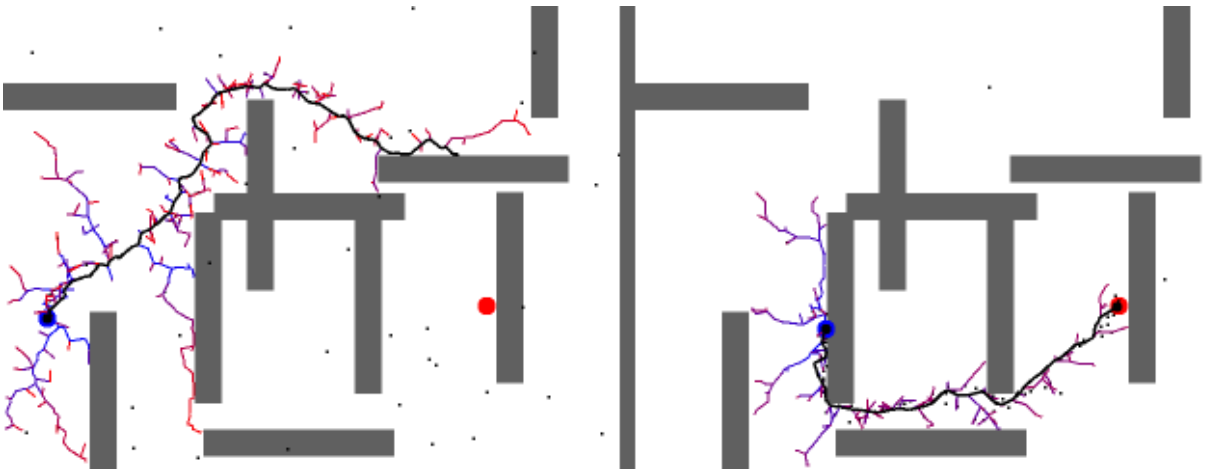


Figure 4.1: An example from the simulation-based RRT planner, shown at two times during a run. The tree grows from the initial state. The best plan is shown in bold, and cached waypoints are shown as small black dots.

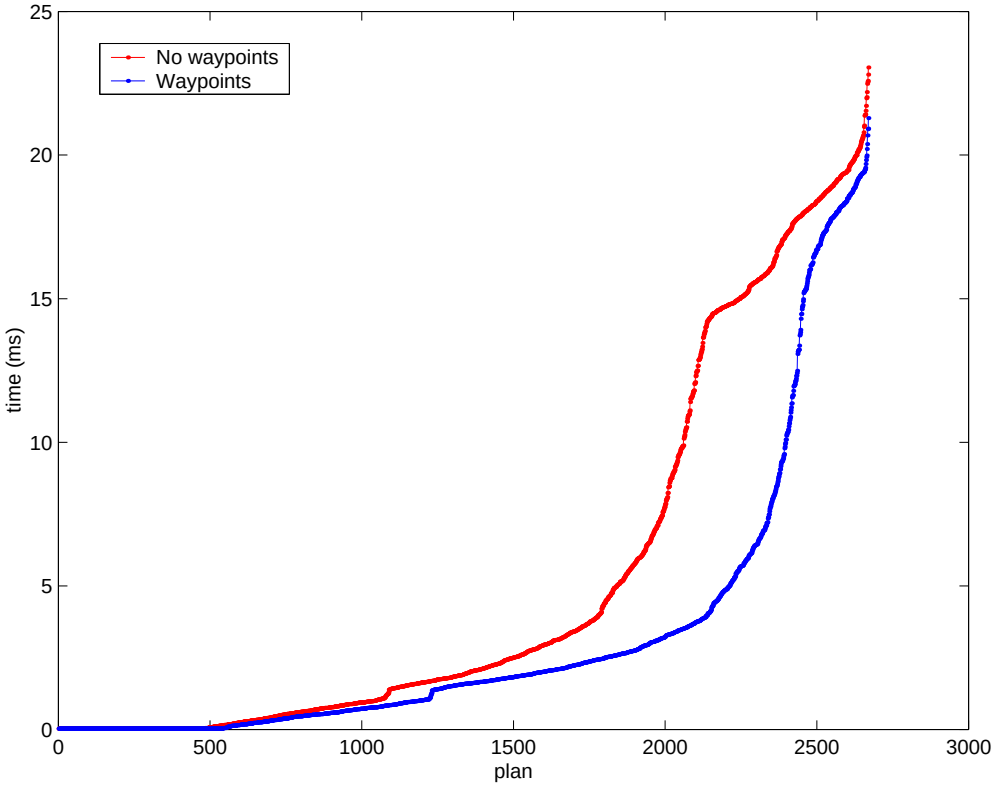


Figure 4.2: The effect of waypoints. The lines shows the sorted planning times for 2670 plans with and without waypoints, sorted by time to show the cumulative performance distribution. The red line shows performance without waypoints, while the blue line shows performance with waypoints (Waypoints=50, p[Waypoint]=0.4).

For a RoboCup like domain with 10 other obstacles, we found that expanding a constant number of nodes (N=500) and 50 waypoints worked fairly well, and would normally run in 20ms or less. The step size of the extensions was 8cm (the world is 320cm by 240cm). The goal was used as a target with $P[goal] = 0.1$ and the waypoints were used with $P[waypoint] = 0.4$. The adaptive beta parameter was turned off in the final version, since waypoints appear to achieve most of the benefits of beta search without increasing the search time, although in future work we'd like to explore the relationship between the two. The waypoints qualitatively seemed to help get the agent out of difficult situations, since once it found a valid plan to take it out of some local minima or other oscillation point, it would be highly likely to find a similar plan again since the waypoints had such a high probability of being chosen as targets. This effect of waypoints on performance was examined in an experiment, and the results are shown in in Figure 4.2. Since the curves diverge at moderate difficulty, it appears that waypoints help speed planning by offering "hints" from previous solutions. When the problem becomes impossible or nearly impossible, neither performs well. This is what one would expect, because waypoints are only available when a solution was found in a previous iteration. Overall, he simulation appeared successful enough that we set out to adapt the system for a real robot, and to employ a KD-tree to speed up the nearest neighbor lookup step to further improve efficiency.

For RoboCup F180 robot control, the system for must take the input from a vision system, reporting the position of all field objects detected from a fixed overhead camera, and send the output to a radio server which sends velocity commands to the robots that are being controlled. The path planning problem here is mainly to navigate quickly among other robots, while they also more around executing their own behaviors. As a simplification for data gathering, we first examined the more simple problem of running from one end of the field to the other, with static robots acting as obstacles in the field. In filling in the domain dependent metrics, we first tried metrics that maintained continuous positional and angular velocity, continuous positional velocity only, and fixed curvature only. None of these worked well, substantially increasing planning times to unacceptable levels or failing to find it at all within a reasonable number of node expansions (N=2000). All metrics were written in terms of time and timesteps so the planner would tend to optimize the time length of plans. Although this was a substantial setback, we could still fall back on the obviously physically incorrect model of no kinematic constraints whatsoever and fixed time step sizes, which had been shown to work in simulation. The extension metric then became a model of a simple heuristic "goto-point" that had already been implemented for the robot.

The motivation for this heuristic approach, and perhaps an important lesson is the following: (1) in real-time domain, executing a bad plan immediately is often better than sitting still looking for a better one, and (2) no matter how bad the plan, the robot could follow it at some speed as long as there are no positional discontinuities. This worked reasonably, but

the plans were hard to follow and often contained unnecessary motion. It did work however, and the robot was able to move at around $0.8m/s$, less that the $1.2m/s$ that could safely be achieved by our pre-existing reactive obstacle avoidance system. The final optimization we made was the post-process the plan, which helped greatly. After a path had been determined, the post processing iteratively tested replacing the head of the plan with a single straight path, and would keep trying more of the head of the plan until it would hit an obstacle. Not only did this smooth out the resulting plan, but the robot tended to go straight at the first "restricted" point, always trying to aim at the free space. This helped tremendously, allowing the robot to navigate at up to $1.7m/s$, performing better than any previous system we have used on our robots. Videos of the this system are available in the supplemental materials [21].

The best combination of parameters that we were able to find, trading off physical performance and success with execution time was the following: 500 nodes, 200 waypoints, $P[goal] = 0.1$, $P[waypoint] = 0.7$, and a step size of 1/15sec. To examine the efficiency gain from using a KD-tree, we ran the system with and without a KD-tree. The results are shown in Figure 4.3. Not only does the tree have better scalability to higher numbers of nodes due to its algorithmic advantage, but it provides an absolute performance advantage even with as few as 100 nodes. Using the tree, and the more efficient second implementation for the robot rather than the initial prototype simulator, planning was able to perform on average in 2.1ms, with the time rarely going above 3ms. This makes the system fast enough to use in our production RoboCup team, as it will allow 5 robots to be controlled from a reasonably powerful machine while leaving some time left over for higher level action selection and strategy.

4.3.2 Conclusions in Applying ERRT to Soccer Robot Navigation

In this work a robot motion planning system was developed that used an RRT path planner to turn a simple reactive scheme into a high performance path planning system. The novel mechanisms of the waypoint cache and adaptive beta search were introduced, with the waypoint cache providing much improved performance on difficult but possible path planning problems. The real robot was able to perform better than previous fully reactive schemes, traveling 40% faster while avoiding obstacles, and drastically reducing oscillation and local minima problems that the reactive scheme had. This was also the first application of which we are aware using an RRT-based path planner on a real mobile robot.

Several conclusions could be drawn from this work. First, a heuristic approach (incorrectly) assuming a purely holonomic robot for the extend operator may perform better than a more

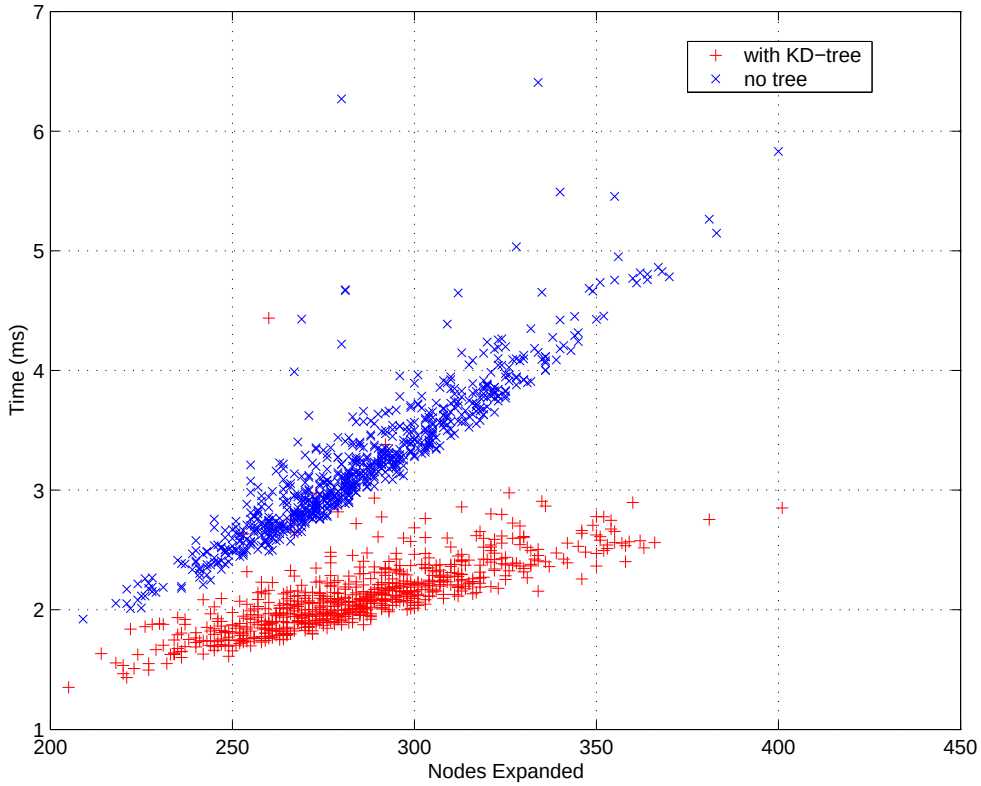


Figure 4.3: Planning times vs. number of nodes expanded with and without a KD-tree for nearest neighbor lookup. The KD-tree improves performance, with the gap increasing with the number of nodes due to its expected complexity advantage ($E[O(\log(n))]$ vs. $E[O(n)]$ for the naive version).

correct version with a dynamics model when planning time is critical. In other words, a better model may not improve the entire system even if it makes the generated plans better, because the more correct model severely limits the number of points or iterations that can be run. Second, it was found that plan simplification was necessary when incorrect motion models are used. Our real robot system worked without any plan post processing, but not nearly as well as when a local post-processing metric could apply its more accurate model over the head of the plan and thus remove most of its excess motion for the period about to be executed. Finally, the highly flexible extend operator of RRT demonstrates how existing reactive control methods can easily be added to improve the planner by adding domain specific control rules. ERRT can build on these reactive methods to help eliminate oscillation and local minima through its lookahead mechanism. Together they can form a fast yet robust navigation system.

4.3.3 Exploring waypoint Cache Replacement Strategies

In the base ERRT planner, the waypoint cache of previous plans is a fixed size bin with random replacement. However many other possible replacement strategies are possible, so we thought it to be an interesting area to explore. One approach that is even simpler than random replacement is to simply use the last valid plan in its entirety. By not splitting the plan, we still maintain a sequence ordering which allows an optimization: drawing waypoints from the cache which the search tree has already reached only wastes computation, since the tree already can reach that far up the plan. Thus the truncation heuristic was devised: Any time the planner reached a waypoint w that it was extending toward, successive draws from the waypoint cache are limited to nodes that came after w in the plan stored in the cache. This effectively limits and focuses the waypoints drawn to those areas not yet explored by the search tree. We called this heuristic "truncation" as can inactivate the leading segment of the waypoint cache during planning.

To test the effect, we created a sample set of domains inspired by the RoboCup small environment, with additional obstacles added to make the planning problems more complex. The planner was run for 2000 iterations on the first six example environments shown in Figure 4.7 (i.e. all those with 34 or fewer obstacles). Each environment is approximately 5.5m by 4.1m, and the agent's avoidance radius is 90mm. The initial and goal positions goal positions were varied in a vertical sinusoid to represent the changing robot position and newly calculated target points for navigation.

Since the waypoint cache's effect on planning is dependent on how frequently we select from the cache, the waypoint cache drawing probability was also varied. In all the problems, the

probability of picking the goal configuration was fixed at $p = 0.05$. The timing results from running on a 2.4GHz AMD Athlon-based computer are shown in Figure 4.4. First, we can see that all methods performed well, planning in less than 1ms on average. Next, we can see that there is little difference between the random replacement bin and the last single plan approaches. When we apply the truncation heuristic to the last single plan approach however, we achieve a noticeable speedup. On average, planning times improved by 25% in the range of "reasonable" waypoint cache probabilities from 0.5 to 0.8. Thus in further ERRT derived planners, the truncation heuristic as an alternative to random replacement bins of the original ERRT approach.

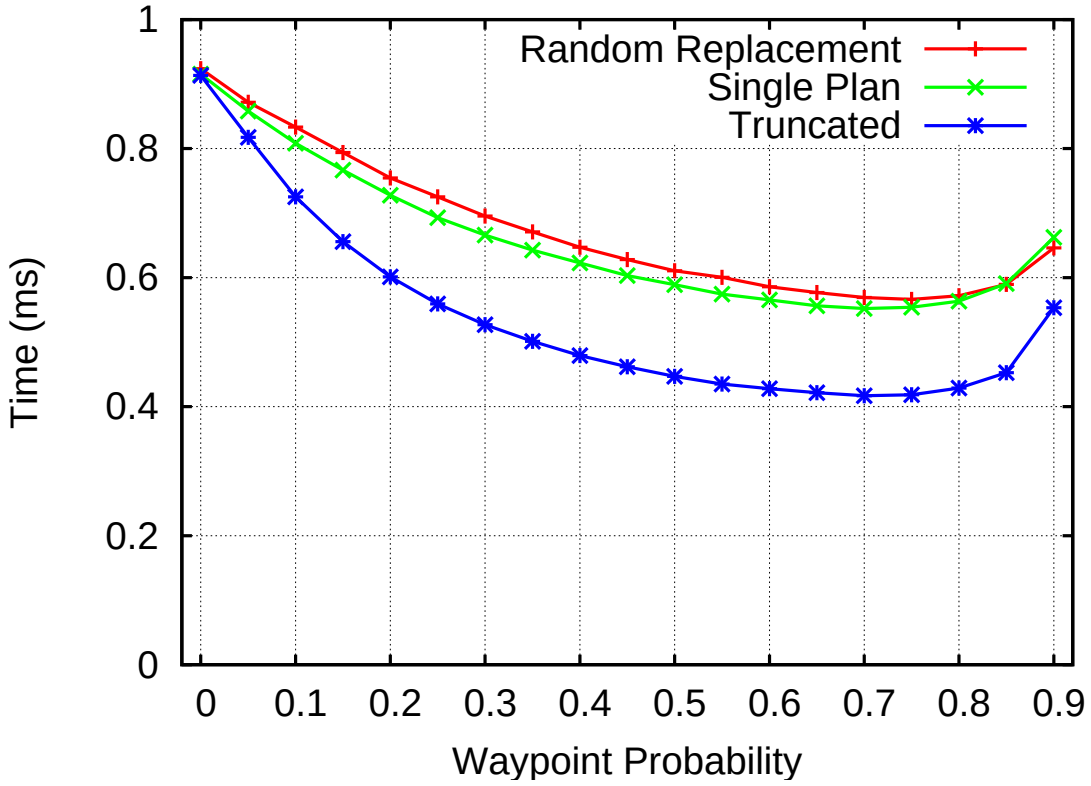


Figure 4.4: Comparison of several waypoint cache replacements strategies while varying the waypoint cache selection probability.

4.3.4 Bidirectional Multi-Bridge ERRT

One variant of the RRT planner which has demonstrated high performance for one-shot queries is RRT-Connect [51]. It combines bidirectional search with an iterated extension

step. In RRT-Connect, a random target is chosen just as with the base RRT planner. However, instead of calling the extend operator once, the RRT-Connect repeats the extension until either the target point is reached, or the extension fails (such as when it would hit an obstacle). The search is performed bidirectionally, with a tree growing from both the initial and goal configurations. In each step, after a random target point is chosen, one tree repeatedly extends toward that target, reaching some final configuration q'. The second tree then repeatedly extends toward q' (as opposed to the random target), in an operation referred to as *Connect*. After each iteration, the tree swaps roles for extending and connecting operations, so that both the initial and goal trees grow using both operations.

While these improvements can markedly improve RRT's one-shot planning time, they do have an associated cost. While RRT, with its single extensions, tends to grow in a fixed rate due to the step size, RRT-Connect has a much higher variance in its growth due to the repeated extensions. As a result, when planning is iterated, RRT-Connect tends to find more widely varying homotopic paths. This is not an issue for one-show planning, but can become a problem for iterated planning with interleaved execution. Thus ERRT adopts the improvements of RRT-Connect, but modified somewhat. First, ERRT supports repeated extensions, but only up to some maximum constant, which can be tuned for the domain. Figure 4.5 shows the effect of this parameter on the average plan length for a domain. Each datapoint is includes the mean and confidence interval for 300 test runs, where each run represents 240 iterated plans on the domain *RandRect* (see Figure 4.7), under sinusoidal motion of the initial and goal positions. As the number of extensions increases, the plan average length grows. While applications can apply smoothing to remove some of the inefficiency of less optimal plans, a shorter plan is more likely to be in the same homotopy class as the optimal plan. Thus plan length is at least one indicator of the reachable length even after smoothing, and stability in the homotopic path is important for interleaved execution of the plan. In the figure, it is unclear why the average plan length drops after 10 repeated extensions, although this change is not large in comparison to the confidence interval. Thus repeated extensions, while they may speed planning, may come at the expense of average plan length. ERRT, by using a tunable constant to set the maximum number of extensions, allows the system designer to trade off between the two. In most applications, we have used a value of 4, and found it to represent a reasonable compromise.

ERRT also adopts the notion of the connect operation from RRT-Connect, however this can also cause plan length to suffer. ERRT again offers a way to mitigate this with a tunable parameter, which is the number of connections between the initial and goal trees before planning is terminated. Thus, ERRT allows multiple connection points, and can choose the shortest path of those available when planning terminates. Implementing such a feature represents an important departure from RRT however; with multiple connections the connected planning "tree" is actually now a graph. This does not pose any significant

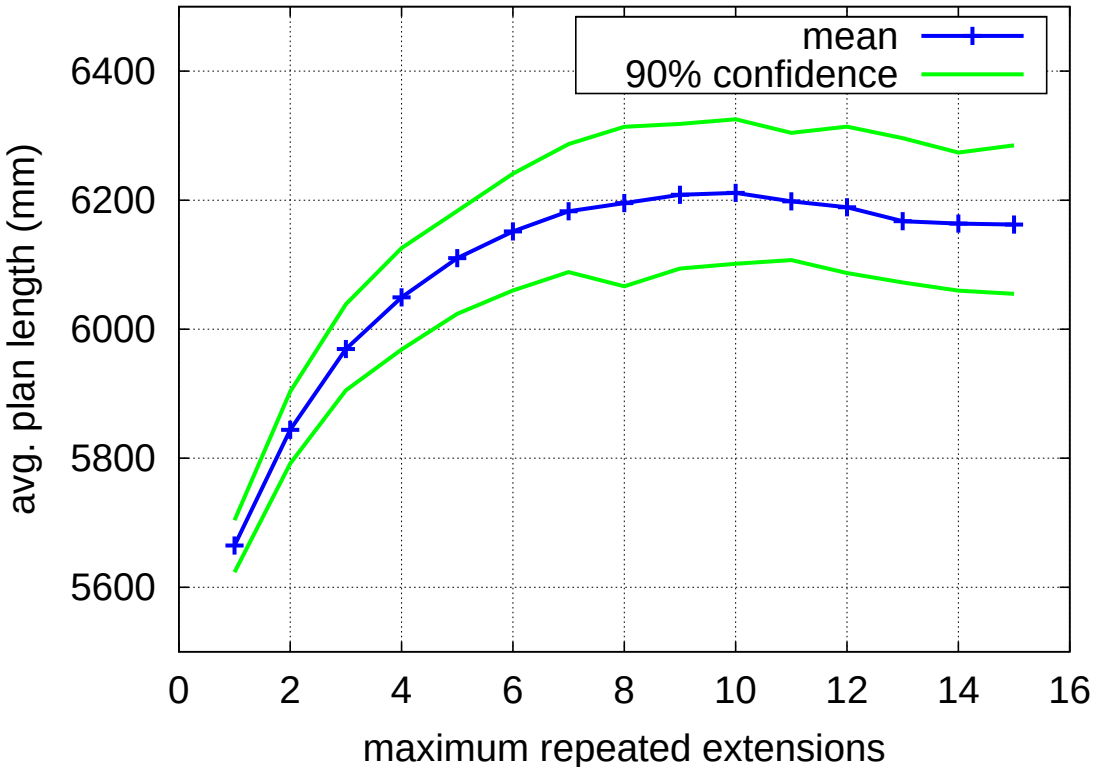


Figure 4.5: The effect of repeated extensions in ERRT on plan length.

theoretical problems to RRT, but requires the implementation to support graph operations efficiently instead of the normally faster tree operations. When planning terminates, A* search is used over the resulting graph. The effect of varying the number of connection points is shown in Figure 4.6. The methodology is the same as used for the maximum repeated extensions, but using the domain *RandCircle*. As can be seen, increasing the number of connections improves the plan length, although the effect decreases after 6-8 connection points. Multiple connection points by definition increase planning execution time, since ERRT continues to search even after a path has been found. However, after the first connection, ERRT can operate in a purely any-time fashion, using up idle time in the control cycle to improve the solution, but able to terminate and return a valid solution whenever it is needed. Again, by offering the number of connections as a parameter, the tradeoff can be set by the system designer.

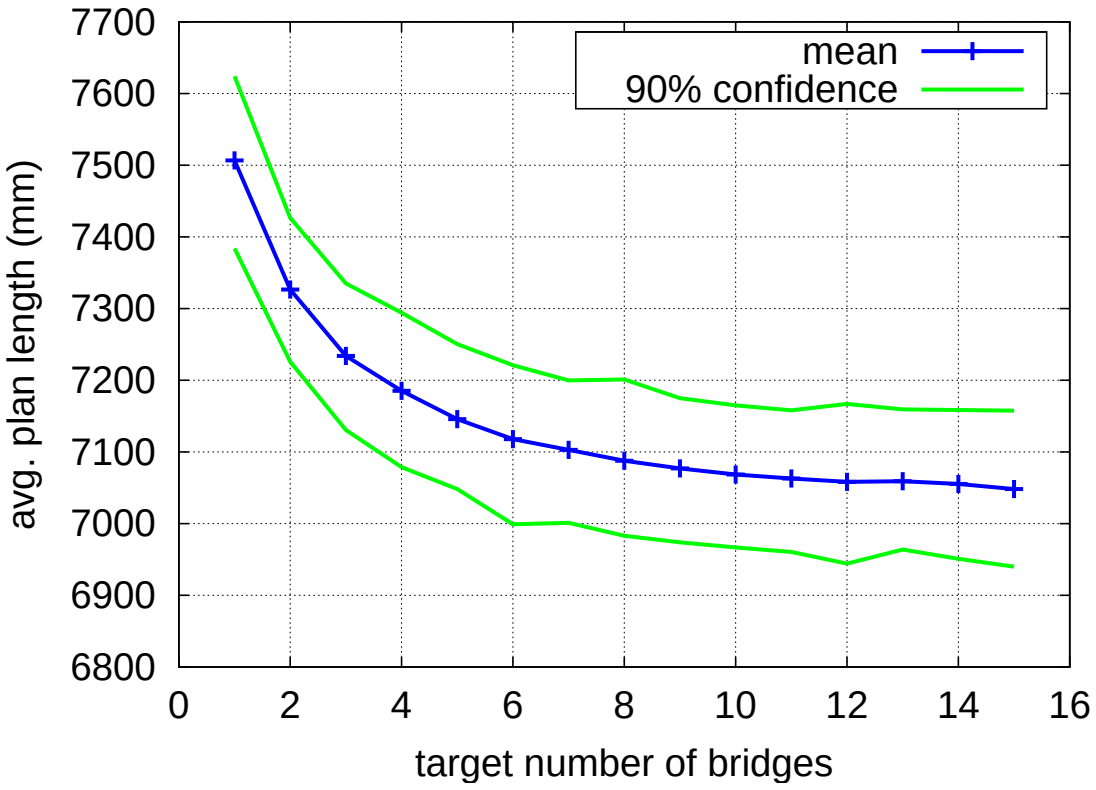


Figure 4.6: The effect of multiple connection points in ERRT on plan length.

Supporting multiple extensions and multiple connection points give ERRT the benefits of RRT-Connect, as well as the ability to tune the system as needed. By supporting a graph representation instead of search trees, as well as planning over the resulting roadmap graph,

ERRT takes a step toward unification with the PRM family of planners. The only differences that remain are in the sampling strategies for building and maintaining the search graph or roadmap structure.

4.3.5 Comparing ERRT with the Visibility Graph in 2D

Since ERRT operates in 2D for our test domains, it is a reasonable question to ask whether it offers any advantage over traditional methods. Because ERRT is a continuous domain planner, it is most directly comparable to the Visibility Graph method [69] (see 6.1.3). Although ERRT is not limited to 2D, it can be compared with the Visibility Graph within this context. In addition, as the Visibility Graph is a minimum-length-optimal planner, we can compare the plan lengths produced by ERRT to the optimal lengths. The environments used can be seen in Figure4.7, and range from four to 128 obstacles, and cover a broad range of free volume ratios and C-space connectivity. The domain is roughly modelled on RoboCup small-size, where each environment is 5.5m by 4.1m, with the agent's avoidance radius set at 90mm. The agents moved in a vertical sinusoid with a period of 120 iterations, and the data points are the mean values of 2000 planning iterations. The ERRT parameter values are shown in Table 4.3.5.

ERRT Parameter	Value
Maximum number of nodes	512
Goal target point probability	0.05
Initial target point probability (for bidirectional search)	0.05
Waypoint cache selection probability	0.80
Waypoint cache size	100
Extension step size	120
Max repeated extensions	4
Target number of connections	4

Table 4.5: The parameter values for ERRT used in the benchmark experiment.

The results of the benchmark testing can seen graphically in Figure 4.8, and a tabular summary in Table 4.3.5. First, it is apparent that ERRT has an advantage in execution time for more complex domains, particularly those with many small obstacles. Because the Visibility Graph must model all possible tangents in its constructed roadmap, it scales poorly with obstacles. It uses up to $O(n^2)$ edges for n obstacles, even for domains that conceptually that difficult, such as *CircleGrid* and *BoxGrid*. ERRT fares worst in a comparative sense

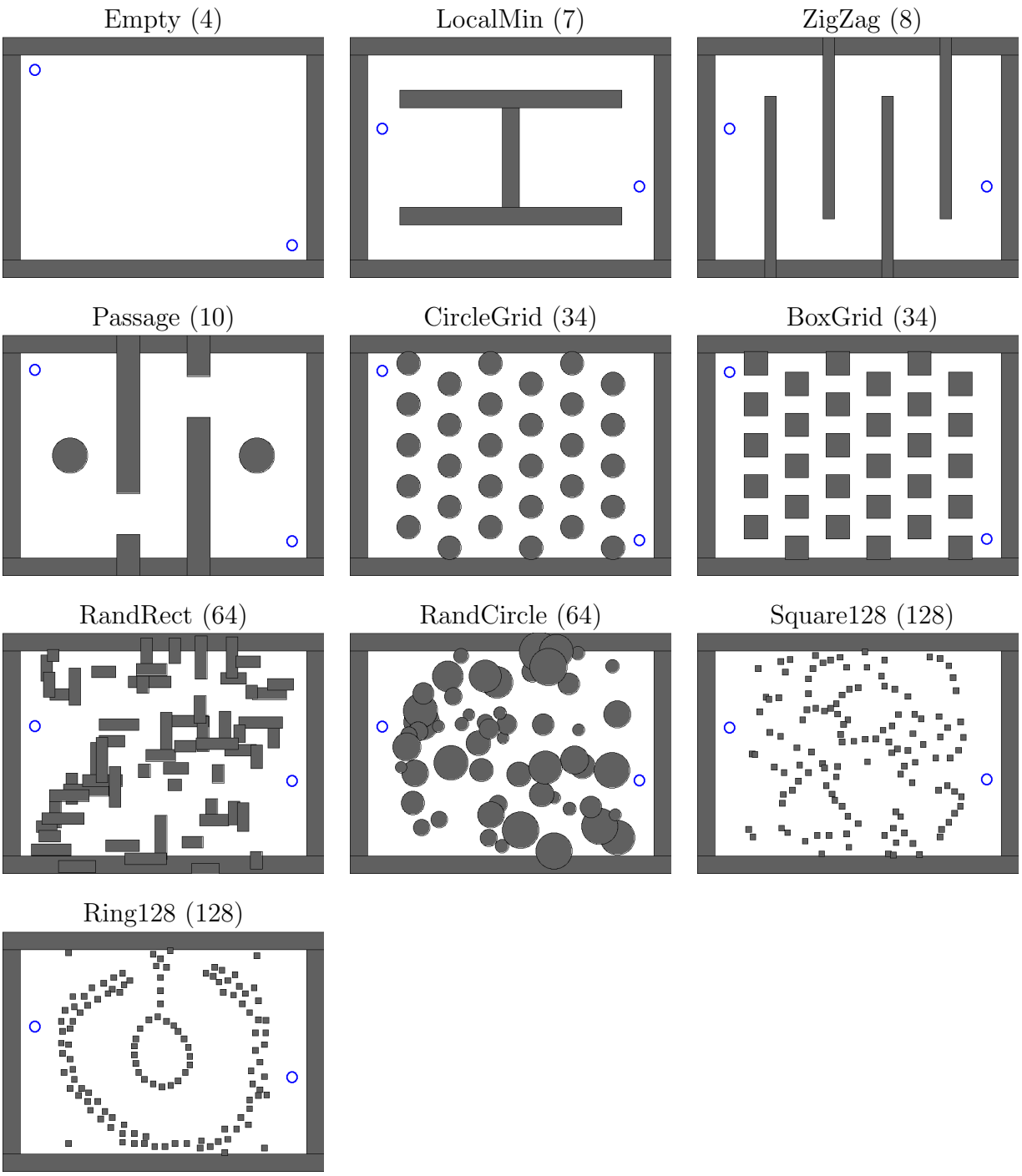


Figure 4.7: Environments used for benchmarking planners.

on *ZigZag*, which has few obstacles but a long solution path. ERRT takes almost 13 times the execution time of Visibility Graph, although ERRT still only requires 1.4ms of planning time on average for the domain. ERRT fares best on *Square128*, running 28 times faster than Visibility Graph, which takes over 12ms of planning time. ERRT's slowest execution time across all domains is only 2.2ms.

In terms of success probability, as shown in part (b) of Figure 4.8, Visibility Graph demonstrates its completeness, never failing to generate a solution, while ERRT is not far behind. ERRT is worst on *Ring128*, where it finds a solution 97.5% of the time, second worst on *ZigZag* at 99.6%, and 100% on the eight other domains. The success probability is adequate for all these domains when using iterated planner, as failures in one iteration only result in a short stall of execution. Finally, in terms of plan length ERRT is at worst 28% longer than the optimal path, even without whole-path optimization. This is aided by tuning the extension step size, maximum repeated extensions, and the number of connections for bidirectional search.

Domain	VisGraph exec. (ms)	ERRT exec. (ms)	speedup (VisGraph/ERRT)	relative plan length (ERRT/VisGraph)
Empty	0.057	0.091	0.626	1.046
LocalMin	0.102	0.558	0.182	1.154
ZigZag	0.110	1.401	**0.078**	**1.283**
Passage	0.154	0.300	0.513	1.225
CricleGrid	1.654	0.245	6.746	1.077
BoxGrid	1.623	0.522	3.107	1.226
RandRect	1.986	0.432	4.596	1.132
RandCircle	1.311	0.652	2.011	1.124
Square128	**12.194**	0.428	**28.471**	1.163
Ring128	7.329	**2.199**	3.333	1.246

Table 4.6: A comparison of ERRT with the visibility graph method across several 2D domains.

In summary ERRT compares favorably even against the classic optimal 2D Visibility Graph planner. Even in relatively simple domains, if numerous obstacles need to be supported with low execution times, ERRT makes a good choice in place of Visibility Graph. With some robotic applications modelling the environment with large numbers of obstacles retrieved from sensors, this can be an important property. If however, the solution paths are long, with only a few obstacles, ERRT does not appear to be the best choice for 2D planning. ERRT's advantage with large numbers of obstacles, along with its ability to operate in higher dimensions inherited from RRT, appears to make it a reasonable choice for general mobile robot planning applications.

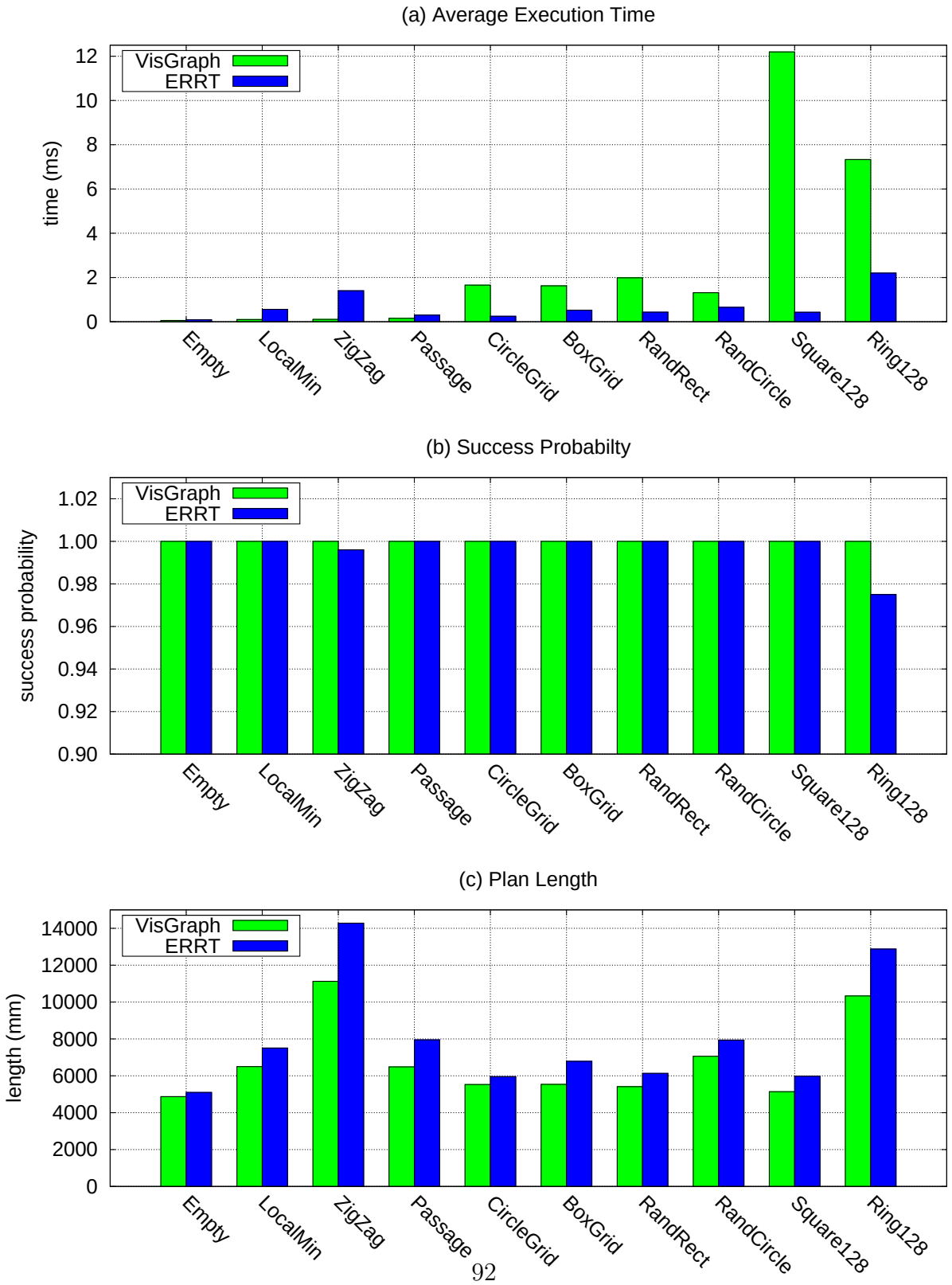


Figure 4.8: A comparison of ERRT with the visibility graph method across several 2D domains.

4.4 Dynamic PRM

PRM planners have proved to be the most popular randomized planner for static environments. Since they split planning into an environment learning stage and a very fast query stage, they work very efficiently for environments that do not change often. In addition, they support explicit minimum cost searches of the roadmap, which is useful when there are non-binary obstacles with different traversal costs. Unfortunately, for mobile robotics problems the environment is often non-static, which means the learning stage needs to be rerun frequently. However, a large class of environments can be considered *nearly static* however; For example, in a typical cluttered room only a few objects move at any time. One particularly motivating domain for this approach is the QRIO humanoid domain, where path stability is important due to high action latency, and the environmental obstacles evolve slowly over time.

The only change that needs to be made to support dynamic environments is make the learning stage operate incrementally. First, each node in the roadmap graph is augmented with a replacement probability. Every sensor update cycle, all nodes are checked for validity. They are marked as invalid if they no longer represent a free configuration. Additionally, they can be randomly invalidated based on their replacement probability. All edges are then checked to make sure they represent free paths, and marked invalid if they would result in a collision, or if either of their two endpoint nodes has been marked as invalid. After this step, the invalid nodes and edges are removed from the graph. The next step is to add nodes and edges to the graph so that roadmap size and connectivity are maintained. Unlike advanced static PRM planners, Dynamic PRM uses uniform sampling to generate configurations. In order to achieve non-uniform graph density however, the initial replacement probability of a node is based on its proximity to obstacles, decreasing as it gets further from obstacles. After many iterations, this converges to a distribution similar to Gaussian PRM [9], which samples more configurations near the boundary of free space. However Dynamic PRM can achieve that density without the necessity for running more costly rejection sampling techniques for free configurations, instead spreading the work over time.

Adding edges to a roadmap is an extremely costly operation if performed naively. The simplest algorithm to implement would be to test all pairs of nodes, which of course scales quadratically, and is not practical for large roadmap graphs. In [54], a variation is selected where the nearest neighbors from different connected components are chosen, up to some distance threshold or degree limit. This results in a forest, which can poorly describe workspaces with cycles; The query phase will only return one homotopic path, while in the case of cycles we would like it to return the shortest path (or any other cost metric minimized during the graph search). If we remove the tree constraint however, then local cliques may tend to form

where all the nodes are closer to each other than neighboring groups, and thus fail to connect to the graph outside of that group, even though a free path exists. Furthermore, querying the closest set of neighbors is a more complex operation than returning only a single nearest neighbor for a query.

In order to explore this effect, Dynamic PRM implements three possible sampling strategies for adding edges. The first technique, called *nearest-k* is the traditional PRM approach of attempting to connect to all nodes up to some limit k within some maximum connection distance d. The second approach, called *gauss*, randomly samples a point x near a particular node q based on the a multivariate Gaussian centered on that node with standard deviation $d/2$. The random point is then used to query the nearest node q' in the roadmap, and a connection is attempted between q and q'. In other words, the method takes a random step, and then looks for a neighboring node in that area with which to connect. This is repeated several times to form multiple connections. The third method is a refinement of *gauss*, called *ring*, which independently samples a direction and a distance. The direction is sampled uniformly, while the distance is a Gaussian centered at some non-zero distance. The resulting distribution for the random step in 2D is approximately[1] a Gaussian revolved around the origin, and thus resembles a "fuzzy ring".

To visualize how the sampling method of edge connection works, Figure 4.9 shows a simplified sampling method which samples at fixed angles in 2D. On the left, the sampled steps are shown extending radially from a node we wish to connect to other nodes in the roadmap. After taking the nearest neighbor to the end of each of the steps, and connecting those two nodes if a straight-line path exists, the resulting roadmap is shown on the right of the figure. The *guass* and *ring* methods operated similarly, although the randomized directions allow them to generalize to higher configuration space dimensionality.

Several enhancements to the basic algorithm can be applied to improve performance. First, in order to fill in gaps left behind moving objects, it is necessary to recycle nodes, freeing them up to be used in a new location. Although all nodes have a random replacement probability, the recycling rate can be more directly controlled by introducing a parameter called node aging. It is an increment applied to node replacement probabilities so that older nodes are more likely to be replaced than new ones. A lower increment is more efficient since less time is spent adding new nodes and edges to replace the replaced ones, while a higher rate helps fill in gaps faster for dynamic environments. A second optimization is that during the query stage, all nodes used in the final path have their replacement probabilities set to zero. This ensures they won't be replaced in the next sensor update cycle, and less likely immediately afterward than other nodes. It also effectively increases the sampling density

[1]It is approximate because the true density increases as one nears the axis. The larger the mean offset compared to the standard deviation, the more the distribution will approach a revolved Gaussian.

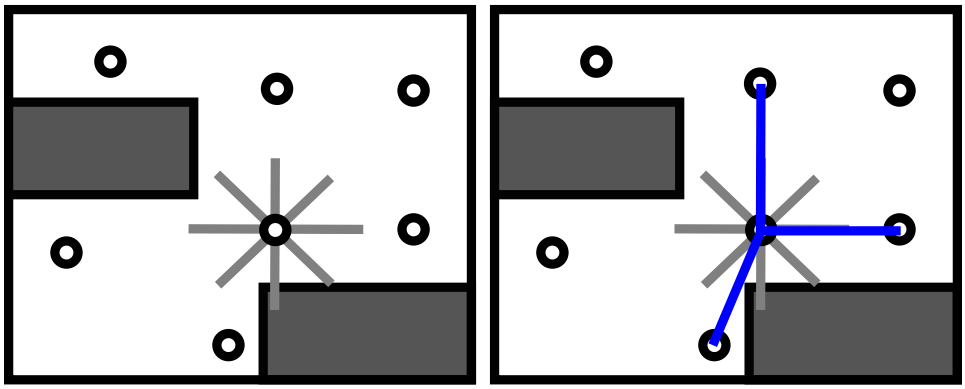


Figure 4.9: Dynamic PRM connection technique, using radial direction sampling. The sampled directions (left) and the resulting roadmap (right) are shown.

along the path since nodes are still added to those areas at the same rate. Higher sampling density helps to improve the local quality of the path.

To test the algorithm and the multiple sampling strategies, a test domain was constructed which is pictured in Figure 4.10. An user interface was created for testing where the a user can drag the goals and obstacles interactively as the system plans and navigates in real-time. For benchmarking the algorithm, the domain is altered systematically. First, the robot starts in one corner of the maze-like environment, and has to reach a goal in the next corner in a counter-clockwise fashion. Each time the robot reaches a goal, the goal position is moved to the next corner, resulting in the robot doing "laps" around the environment. The planner was allowed up to 500 nodes, and up to eight edges per node in constructing the roadmap. Although the obstacles do not move in the benchmark, the algorithm assumes arbitrary changes in the environment may have taken place, as is possible in the interactive testing mode. Thus, the Dynamic PRM method used aging and removing edges and nodes in the roadmap, and sampling continuously to extend the roadmap into existing or possibly new free configuration space.

The results of benchmarking can be seen in Table 4.4. The simulated robot was run for 32 laps around the environment for each sample, and the test was repeated 8 times to generate a mean and standard deviation for each measurement. The simulated time represents the total time for the agent to complete 32 laps, and is mainly depending on successful planning and the quality of the roadmap paths. A failing planner or poor quality paths will increase the total simulated time. Execution time represents the average execution time of the planner each control cycle. Success probability is the chance that the planner returns a full path to the goal each control cycle. Finally, roadmap entropy measures the connectedness of the roadmap by measuring the total entropy based on the sizes of the connected components. A lower

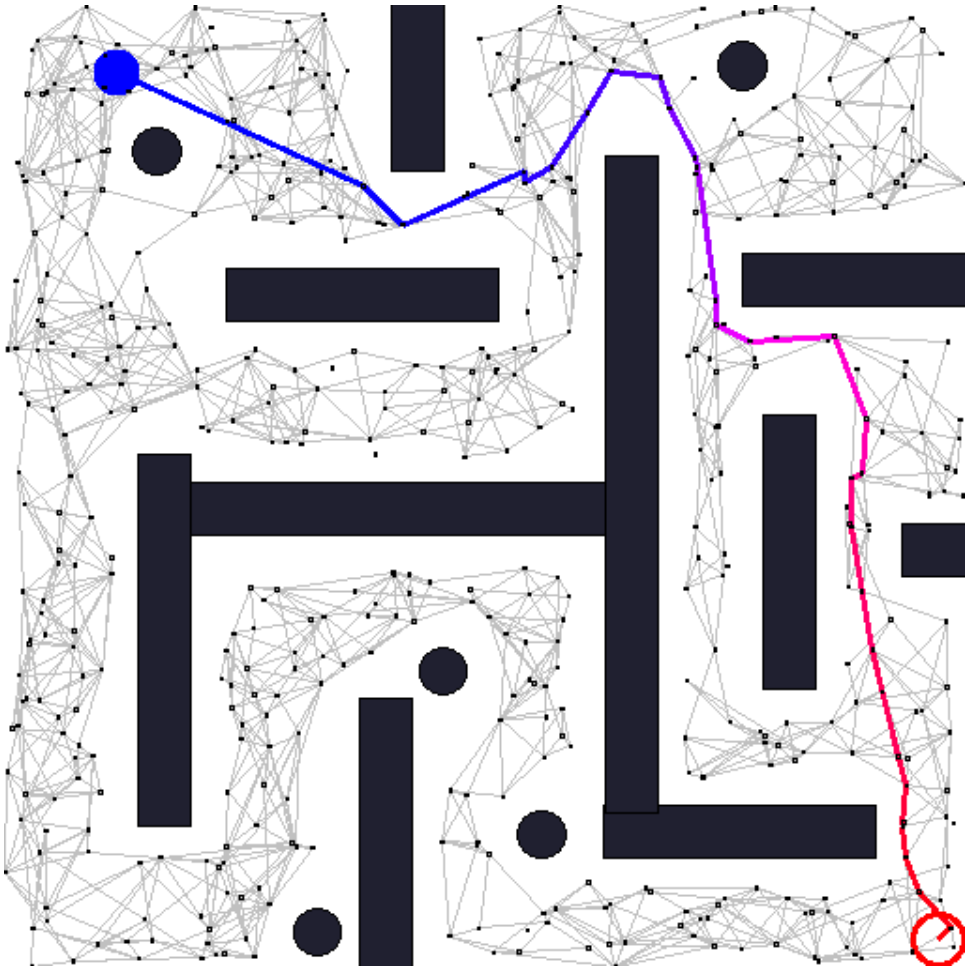


Figure 4.10: The test domain for interactive and benchmark testing of Dynamic PRM.

entropy indicates a more connected roadmap. As can be seen in the table, few differences exist between the methods. The *ring* sampling method leads *nearest-k* in all categories, and leads *gaussian* in all but one category, although the results are not statistically significant. The high success probability is due in part to the optimization of never removing edges from the roadmap used in the plan of a previous cycle, however even then the sampling methods do not have a particularly large impact. This can be seen as a positive result however, because the sampling methods are easier to implement than *nearest-k*. This is due to the fact that the sampling methods do not require a multiple-nearest-neighbor query to be implemented by the spatial data structure storing the roadmap. Larger differences have been found between the methods for roadmaps with an insufficient number of nodes, and thus a low planning success probability, however this is not particular advantage. Such badly-tuned parameters would be avoided in most implementations anyway, as tuning the number of nodes is just as easy as changing the sampling method.

Sampling Method	Simulated Time [sec (std)]	Avg. Execution Time [msec (std)]	Success Prob. [p (std)]	Roadmap Entropy [e (std)]
nearest-k	327.8 (6.49)	1.059 (0.00237)	0.9810 (0.00314)	1.030 (0.0173)
gaussian	324.6 (5.42)	1.051 (0.00212)	0.9808 (0.00315)	1.030 (0.0090)
ring	323.1 (4.00)	1.052 (0.00203)	0.9815 (0.00320)	1.029 (0.0122)

Table 4.7: A comparison of performance of Dynamic PRM with various approaches for roadmap connection.

Although not explored to the extent that RRT derivatives have been investigated, Dynamic PRM demonstrated high success probability and good coherence between control cycles, making it a good choice for slowly evolving domains such as the planner for the QRIO humanoid.

4.5 The Abstract Domain Interface

The domain interface resulted from an attempt to unify platform interfaces so that common planning code could be developed. In traditional planning work, there are typically three primary modules: Planner, collision detection, and a platform model. While this approach works well for robots of relatively similar type, the communication required between the platform model and collision detection are through the planner, complicating the interface so that the planner needs to know much more about the domain than is really necessary. Thus the current approach was devised, in which platform model and collision detection are wrapped into a single "domain" module that the planner interacts with. Internally, collision

detection and the platform model are implemented separately, but importantly the planner doesn't depend on anything involved in the communication between the two. This allows states in the configuration space to be a wholly opaque type to the planner (denoted by S), which needs only to be copied and operated on by the domain's functions. Additionally, to speed up nearest neighbor lookups, the state must provide bounds and accessors to its individual component dimensions. This allows the planner to build a K-D tree of states so that linear scans of the tree are not necessary for finding the nearest state to a randomly drawn target. While the traditional architecture can be made nearly as flexible, it typically does so at a cost in efficiency. The domain interface approach leaves open the opportunities for improved collision detection speed that can come with constraints or symmetry present in the agent model. The domain operations are as follows:

- *RandomWorldTarget():S* - Returns a state uniformly distributed in C
- *RandomGoalTarget():S* - Returns a random target from the set of goal states
- *Extend(s_0:S,s_1:S):S* - Returns a new state incrementally extending from s_0 toward s_1
- *Check(s:S):bool* - Returns true iff $s \in C_f$
- *Check(s_0:S,s_1:S):bool* - Returns true if swept-sphere from s_0 to s_1 is contained in C_f
- *Dist(s_0:S,s_1:S):real* - Returns distance between states s_0 and s_1
- *GoalDist(s:S):real* - Returns distance from s to the goal state set

Using these primitives, the ERRT planner can be built which operates across multiple platforms, and does so without sacrificing runtime efficiency. In addition, in the case of a purely holonomic platform without dynamics constraints, a PRM variant can be implemented using only these primitives (the local planner is formed by calling *Extend* repeatedly). One could argue that the abstract domain approach moves almost all of the "important" code into the domain itself, making the planner itself simplistic. In other words, all of the "planning intelligence" has been shifted to the domain itself. However, the crucial difference is that the code in the domain is relatively straightforward and self contained, while the intricate interactions and practicality driven fall-back cases of planning reside in the core planning code. Thus one could implement a domain with little or no knowledge of path planning, reaching a core goal of general modular programming.

4.6 Practical Issues in Planning

4.6.1 Simple General Path Smoothing

Due to their random nature, the randomized planning approaches generate paths with additionally unnecessary motion present. Although technically safe, such motion is undesirable for direct execution by the agent. ERRT as originally posed used a "straight line" heuristic to smooth the beginning of the path. This approach would check the straight line from the initial position to each successive vertex along the path to see if they were free. The longest free line found was used to replace the head of the plan. For iterative execution this works well in smoothing out the motion of the agent; Later areas of the plan need not be smoothed until the agent reaches that segment. Thus our 2D holonomic planner adopted this simplification method and it has been found to work well in practice.

The straight-line method is however quite limited, since it does not respect kinodynamic constraints. Thus for the UAV planner, a method was needed that respected kinodynamic constraints. At the same time, it is a desirable feature that the smoothing method not be tied to the exact constraints, so that the smoothing would not have to be reimplemented if the constraint model was changed. Thus a more general method was desired, as well as one which operated on entire paths if possible. The method we subsequently developed follows a "leader-follower" approach. The returned plan from ERRT is treated as a "leader", and a new path is grown as a follower. The lead point on the plan maintains a minimum distance from the follower path, and the follower path is grown using the domain's extend operator with the leader point as its target. This method is depicted in Figure 4.11.

In order to smooth the path maximally, the algorithm can be iterated, with successively longer minimum distances maintained by the leader in each pass. If the simplification fails (due to the follower's extension hitting an obstacle), the last successful simplification is returned. If the first few levels of simplification fail, and excess random motion is unacceptable for that platform, it can be treated the same as a plan failure. By running the planner several times, we can obtain a statistically very high probability of a successful smoothed plan being returned. Given the speed of the basic path planner, this can be a viable approach.

4.6.2 Robust Planning

An additional issue important for planning on real robot platforms is that of robustness. While iterative replanning helps deal with positional error within free space, it still does

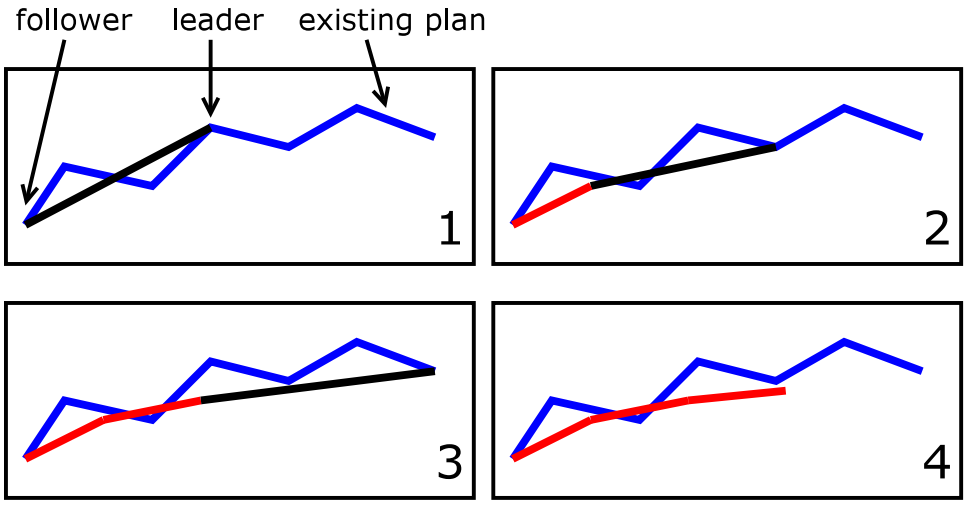


Figure 4.11: Path smoothing using the leader-follower approach.

not gracefully handle problems when the robot steps outside of free space. In practice this can happen quite easily due to sensing error in position, or action errors driving us off the expected course. Implementation on the ground robots found numerous instances where the robot would drive until an unfortunate series of sensor or action error resulted with the robot having 1% of its radius lying with an obstacle. Intuitively we'd like the robot to fail gracefully and plot a smooth path that leads out of the obstacle, but a typical planner only has a binary view of safety, thus it simply fails to generate a path at all. This is because all obstacle checks from the initial position fail at the initial position itself, preventing any extensions from growing the tree. In previous planning work we explored relatively ad-hoc approaches to returning to free space, but none was general enough to apply across domains. Ultimately, we decided to split the general situation of "in an obstacle" into two categories. The more common category is that the agent is only partially inside an obstacle while it's center coordinate is not. This is expressed as a obstacle distance from the center of less than the robot's radius r but greater than 0. The more serious case is when the robot's center is within an obstacle, resulting in an obstacle distance of 0 and meaning that deep penetration has occurred. While the latter case can only really be handled in a domain dependent way (some sort of safe failure or domain specific failure recovery), the partial case can be handled in a general way. Our planner does so in a graceful way to avoid abrupt changes in trajectory which might exacerbate the problem instead of correcting it. When planning is initiated, if the the initial position q_i is found be partially within an obstacle ($0 < D(q_i) < r$), then collision checks from q_i are treated specially. When checking a swept-line from q_i to any point b, we first find a point along the line f where $D(f) \geq r$. Then we check the swept line from q_i to f with a radius of $D(q_i)$, and a second swept line from f to b with a radius

of r. This is shown in Figure 4.12. The first condition ensures the trajectory does not penetrate the obstacle any worse than the initial position, while the segment from f to b ensures that the trajectory stays out of obstacles once it enters C_f. To keep the system independent of the exact geometry of obstacles, f is found by recursive bisection, with a domain specified maximum distance from q_i. Changing the maximum allows varying how aggressively the robot attempts to exit obstacles. The system has been found to work well in practice, handling issues where minor penetration into the obstacles occurs. In doing so, it helped decrease the occurrence of major penetration so that the domain specific "escape" method is only called in cases of serious error or radically changing environments.

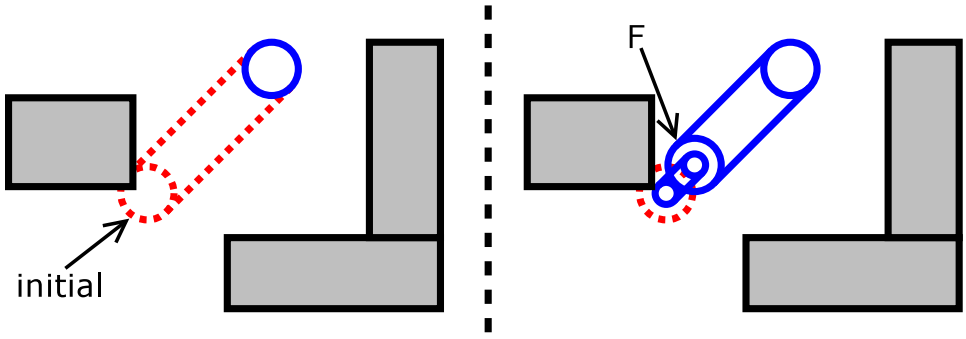


Figure 4.12: An example of a specially handled collision check when the initial position is partially penetrating an obstacle. The collision check is handled in two segments.

4.7 Applications

4.7.1 Fixed Wing UAV

In order to explore ERRT's capability as a planner in 3D environments, and in support of a project supporting FixedWing UAVs, a new abstract domain for ERRT to model UAVs (see 2.2. The UAV operated in 3D with limited climb and descent, as well as a severely limited turning radius. Kinematic constraints were supported by mapping extensions to the nearest reachable action, as in the example in Figure 4.14. The environment consisted of terrain segments, and a total of 1.5 million triangles (see 3.2.3). For speed reasons, a weighted Euclidean distance metric was used. Better metrics exist that depend on the kinematics, but these are difficult to support efficient nearest-neighbor queries for with a hierarchical spatial data structure. As a compromise, the distance function weights components of the position. x and y coordinates along the ground were defined with unity weights. z (altitude), was

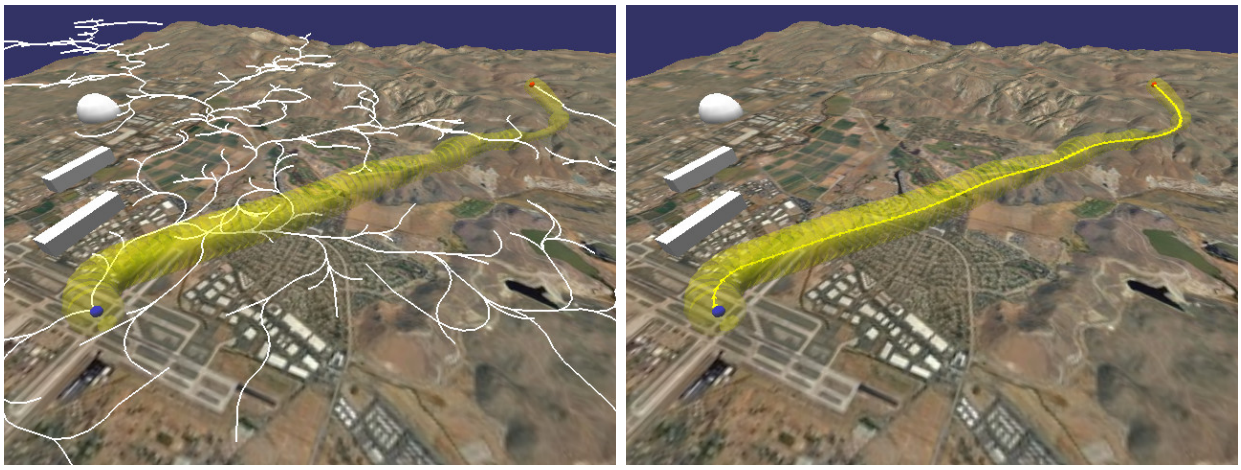


Figure 4.13: A kinodynamicly-limited search tree (left), and the corresponding simplified plan (right) for the UAV. The plan length is approximately 11km with waypoints every 100m. Obstacle clearance is shown as a translucent tunnel.

given a weight of 10, which was the minimum required to generate a "spiraling" behavior when repeated extensions are performed with a relative target that differs only in the z dimension. Distance for x,y orientation was approximated using a heading vector scaled by the current aircraft velocity. The orientation was given a weight of 2, although it did not appear to have a large effect on the plans. The weighted Euclidean distance metric worked, although the planner would sometimes get stuck in a local minima, requiring a target point to be selected in a very small area for additional progress to be made. In addition, application usage constraints enforced one-shot planning by the navigation system, with no context associated with a problem. We solved both problems by iterating the planner on an identical problem instance - local minima were avoided by "sampling" of the planner itself, and multiple solutions can be ranked by quality criteria to help ensure that the returned plan tended toward the shorter and smoother plans of those available. Because no context was associated with a problem, and the replanning samples were ideally independent, the waypoint cache was disabled for this domain. Multiple repeated extensions were still used, although bidirectional search was disabled because the abstract domain only supports forward kinematic constraints. In the final version, the planner replanned up to 48 times or 12 valid solutions. It simplifies any valid solution with the leader-follower path smoothing approach, and picks the a plan based on the minimum cost weighting of length of curvature changes. Execution time was highly dependent on the configuration, but averages between 0.5 to 4.0 seconds on a 2GHz processor.

One optimization that contributed to the performance on difficult problems was adaptive iteration thresholds. With repeated replanning, once we have found a solution, that can serve

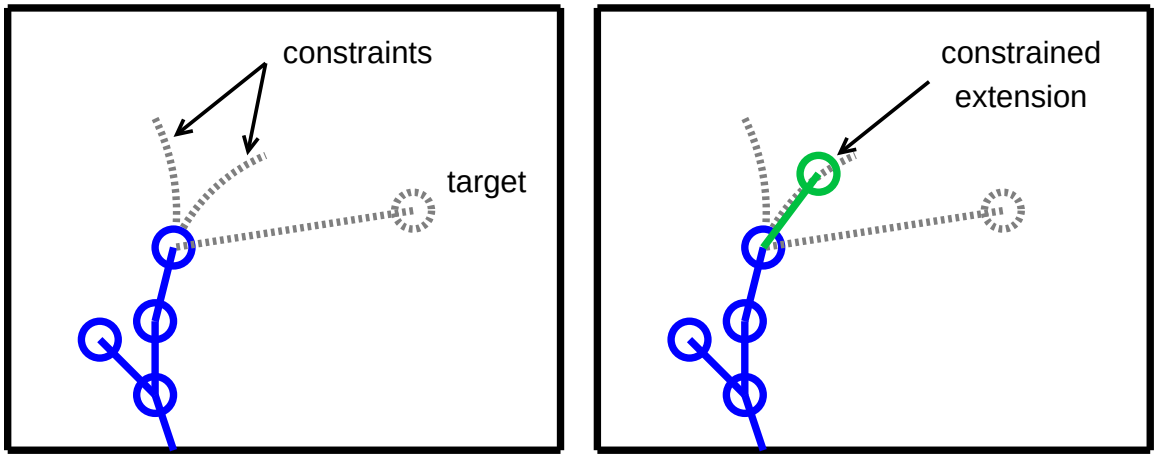


Figure 4.14: An example of limiting Kinematics for the RRT extend operator

as an existence proof that a solution can be found within a certain number of iterations. In subsequent iterations of the planner, we can use that number as a threshold speeding up failure detection if those searches are stuck in a local minima and will not find a solution. In order to help ensure multiple plan iterations are successful, the limit is set at a multiple of the number for the successful plans. Thus, we maintain a iteration threshold m, and update it whenever the planner is successful after i extensions using the rule in Equation 4.1, where k is some constant larger than one (we used $k = 2$ for our implementation).

$$m \leftarrow \min(m, ki) \tag{4.1}$$

The UAV domain motivated a unique new property for ERRT that mapped well onto the randomized sampling planning approach, and would be difficult to implement with a more model-driven approaches to planning. For fixed-wing UAVs, it is often desirable to fly between stable orbits, allowing an aircraft to maintain an average position while it maintains its required forward velocity to prevent stalling. Adding the ability to plan to an orbit is thus a useful extension, but one that could be expensive due to less defined nature of the goal along with the existing kinematic constraints. Because the planner is already rerun on the same instance multiple times however, we can get away with a heuristic that has a non-zero probability of achieving the objective (in fact all randomized planners are already this way due to probabilistic completeness). We thus developed a method we call *active goal*, where the nominal goal configuration is updated dynamically during search. For the fly-to-orbit feature, we want to fly to the tangent of some defined orbit. Using the active goal approach, we set the nominal goal to always be the tangent point relative to the closest point in the current search tree to the goal. Every time a new extension is added to the search tree, if it

is closer to the goal configuration than the previously nearest point, the tangent is updated for the new configuration. An example is shown in Figure 4.15. This method works much of the time, as the point nearest to a goal in an RRT planner is by far the most likely to be extended toward a goal. Due to random exploration, its possible for another branch of the search tree to wander in and hit the goal non-tangentially, but in those cases, that search can be aborted as a failure, leaving a subsequent rerun of the planner to find the solution. Two results from the planner can be seen in Figure 4.16. In practice, the active goal approach allowed the UAV navigation system to support orbits as goals with negligible additional overhead compared to a fixed goal configuration.

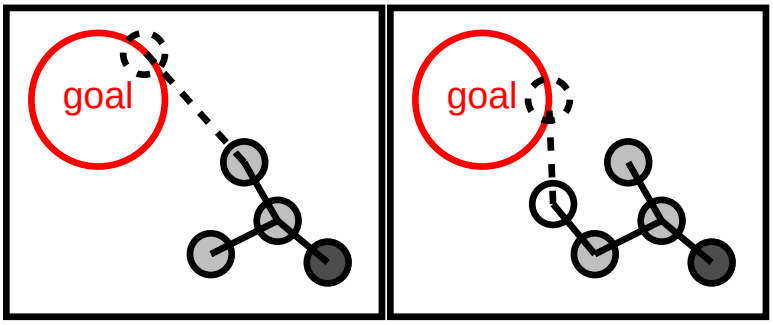


Figure 4.15: An example of a goal updating dynamically during search to try to maintain a tangent constraint. The original tangent, shown dotted (left), is updated as a the ERRT search tree expands (right).

Figure 4.16: Two examples of the fly-to-orbit capability of the UAV planner.

Development of a unified ERRT planner for multiple mobile robot platforms provided many insights that would be difficult to determine if only one platform or similar platforms were

considered for an implementation. However there are still many opportunities for further work. First and foremost, the relationship between distance metric, kinodynamic constraints, and accelerated nearest-neighbor search could be explored. Development of good distance metrics for kinematically constrained platforms is possible, but tedious, and more seriously it prevents most forms of accelerating nearest-neighbor search to fail because the triangle inequality is no longer satisfied. Using Euclidean distance worked, but generated local minima that could only be avoided by rerunning the planner several times, which is an inelegant solution. Better approximations which still allow the use of fast geometric data structures most likely exist.

4.7.2 QRIO Humanoid Robot Planner

Path planning for a small humanoid robot was also explored, using the Sony QRIO [42] (see 2.3). To implement the navigation module two different path planning approaches were attempted. QRIO builds a local occupancy grid of its environment, which it updates using depth maps calculated from stereo vision. Thus the environment is static except for the portion where the robot can currently see. New information is incorporated into the occupancy grid using Bayesian updates. ERRT was successfully adapted for purely 2D planning, but could not easily represent varying traversal costs which would be useful planning which incorporates 3D actions, i.e. climbing and descending.

First, a 2D path planner was implemented using the RRT-based method described above, which compared favorably to an existing grid-based A* implementation. For collision detection, a variant of the distance transform [62] was used on a thresholded occupancy grid, with each free cell storing a offset to the nearest non-free cell. By using the constraint that the nearest non-free cell is either the cell itself or one of the eight neighbors, the transform can be created in two passes, a forward pass where the neighbors to the left and above a cell are checked for their closest non-free cell, and a reverse pass where the neighbors to the right and below are checked. This approach avoids the overhead of the queue based implementation normally used, and results in an approximation which is perfect for convex obstacles, and has at most half of a cell of error in concave corners. The result is shown in Figure 4.17.

To extend the planner to 3D stepping actions, the occupancy grid world model was extended to include height values for each cell. This creates a heightmap, similar to that used in Chestnutt et al. [25]. The point obstacle check used for nodes in the roadmap was modified to check the flatness of the heightmap, using the standard deviation of the cells on which the robot would stand as a metric. The check for edges was separated into two cases, one for flat surfaces and one for stair transitions. The flat surface check used the point query to see if

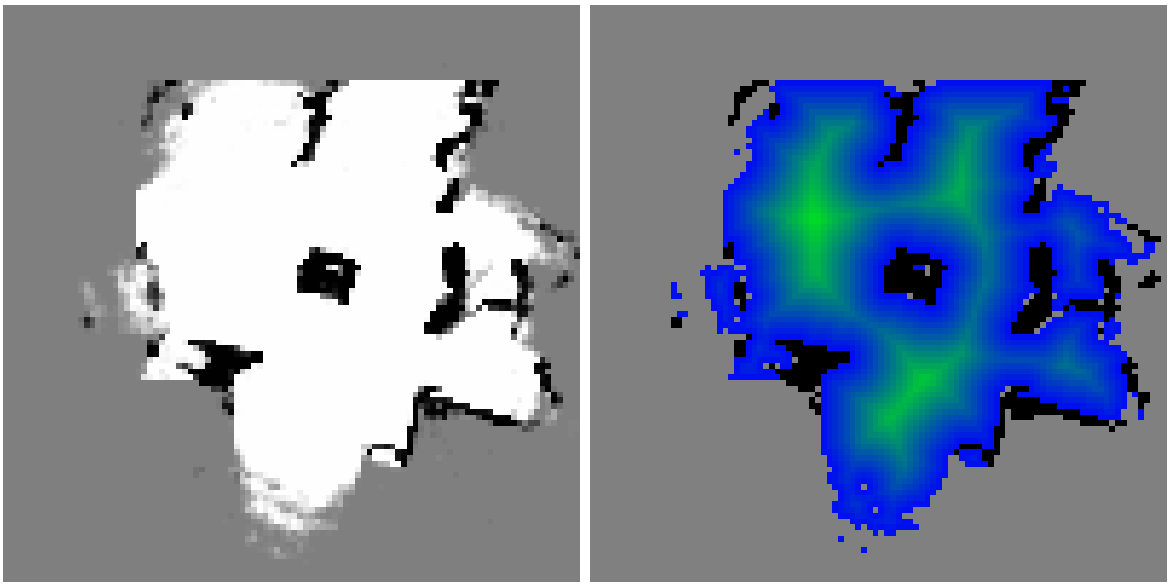


Figure 4.17: An occupancy grid and its distance transform.

robot could stand at intermediate positions after each step. A 2.5D planner using Dynamic PRM was subsequently developed using this collision detection model. The planner used heightmaps to check that no obstacle impinged on the legs or torso for a given location, in addition to the flatness constraint on the ground. The Dynamic PRM representation or free space was updated incrementally as new sensor data was gathered, with cycles occurring several times a second. In addition to planning paths on a single plane, the robot considered actions to step between two planes if it detected such a transition. The stair transition check determined that the change in height was less than the maximum climbable stair size, and that the transition was short enough to be stepped over. Addition metrics for foot placement, and step transition costs as described in [25] were not used.

During testing it was only occasionally successful. However, the system demonstrated that it could autonomously identify, plan, and execute paths between different planes, without being told that the height discontinuities exist before planning, nor the position, or height of those transitions (see Figure 4.18). It was also noteworthy in that most numerical information, including the difference between plane heights, was determined through sensor data rather than supplied as a prior model. Videos of the system are available in the supplementary materials [21].

Figure 4.18: QRIO on top of an autonomously reached step obstacle.

Chapter 5

Safe Navigation

This chapter describes a novel method for safety among a set of cooperating robots. The robots operate under the realistic dynamics constraints of bounded acceleration and bounded velocity while maintaining safety among themselves and with a static environment. Under conditions of noiseless sensing, perfect dynamics, and perfect communication, the system can guarantee no collisions will take place as the robots move about the environment. In more realistic settings, we've found the system to significantly decrease the number and severity of collisions for a team of high performance robots playing RoboCup soccer. The system was used for the past two years on our RoboCup small-size team. We lead the chapter by describing the dynamics model used for the robot and the motion control system employed to reach target points. The following section then describes the safety method, which operates as a post-process to the motion control.

5.0.3 Contributions of this Chapter

- The novel multi-agent *Dynamics Safety Search* (DSS) algorithm is described. It is based on Fox et al.'s single-agent *Dynamic Window* approach [38].
- An outline of a proof of single-agent and multi-agent safety of DSS is contributed. The *Dynamic Window* approach cannot guarantee exact safety even in the single agent case.
- DSS is shown to have polynomial complexity (as opposed to exponential complexity for joint planning), and near linear complexity in simulation testing.

- DSS is demonstrated to aid in collision avoidance on real robots from the RoboCup small size domain.

5.1 Robot Model

While there are many possible configurations of wheeled robots, each with their associated dynamics, three classes covers many types of wheeled robots:

- Differential drive robots (two or more unsteered wheels or tracks)
- Holonomic robots (three or more omni-directional wheels)
- Car-like steered robots

In this chapter we will assume a holonomic robot model, as those are the robots with which we have the most experience and we have available for testing. While the motion control for each type of robot is significantly different, the safety method from this chapter can be extended for different types of robots, with the core assumptions being that a robot can remain at rest, and can come to a stop while traveling on a straight line. In the derivation of our algorithm, we will use the more restrictive assumptions which attempt to model an electrically driven holonomic robot. The assumptions and limitations of the robot model are summarized as follows:

1. The robot is contained within a safety radius
2. The robot has immediate and direct control of its acceleration
3. The robot acceleration restricted to be within some set
4. The robot has a maximum "emergency stop" deceleration
5. The robot can only change the acceleration at fixed time intervals (control period)
6. The robot has a maximum bounded velocity
7. The robot has no minimum required velocity

Clearly these meet the core assumptions, as an agent can stop on a straight line and come to rest if it directly controls its deceleration up to some bound, with no minimum velocity

necessary. In Figure 5.1 we show the layout of the drive system of one the CMDragons RoboCup robots. It has four omni-wheels at fixed angles which. One axis of each wheel is oriented toward the center of the robot and driven, while the other axis rolls freely due to the omni-wheel's perpendicular rollers. We will explore the model more fully in Chapter 5.1, but for the purposes of the safety search it will suffice that the robot and its control system meet the assumptions enumerated above.

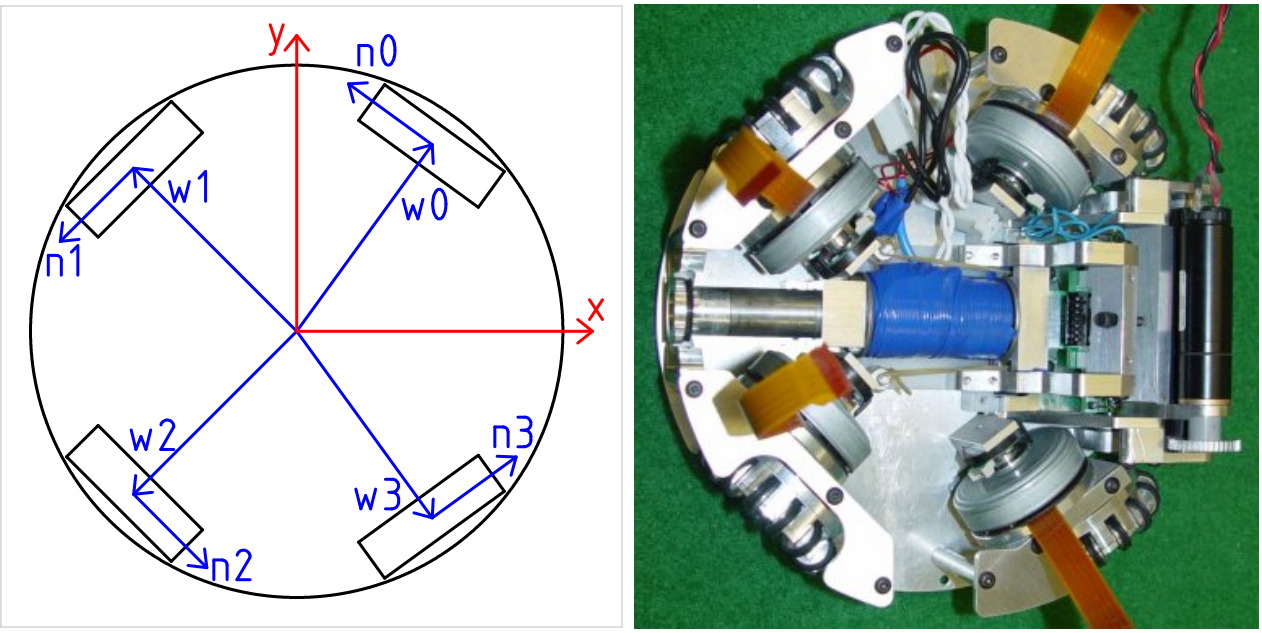


Figure 5.1: Drive layout for a CMDragons holonomic robot with four omni-directional wheels (left), and a picture of a partially assembled robot (right).

The notation used for the model is the following:

- $q(t)$ denotes the robot's position at time t
- $\dot{q}(t)$ denotes the robot's velocity
- $\ddot{q}(t)$ denotes the robot's acceleration
- $A(v)$ denotes the set of possible accelerations at velocity v
- D denotes the maximum "emergency stop" deceleration magnitude
- R denotes the safety radius
- a^k denotes the desired acceleration at decision cycle k (algorithm input)
- u^k denotes the acceleration command at decision cycle k (system control input)

- C denotes the fixed control period

With this notation, we can define the system transfer functions for position (f) and velocity ($\dot{f}$) using Newtonian dynamics:

$$f(q, \dot{q}, u, t_\Delta) = q + \dot{q}t_\Delta + \frac{1}{2}u_i t_\Delta^2 \tag{5.1}$$

$$\dot{f}(\dot{q}, u, t_\Delta) = \dot{q} + u_i t_\Delta \tag{5.2}$$

Within the context of the safety algorithm, we will at times need a more complex denotation for q. At each decision cycle k the algorithm will construct a new piecewise function q for each robot i. Thus $q_i^k(t)$ denotes the position function over t at cycle k for robot i. k will be omitted when it is obvious, such as within a single iteration of the algorithm. To identify the individual components of the piecewise function q, we will use $q_i(t, s)$ to denote $q_i(t)$ at component s. At some times, in particular within pseudocode, a shorthand will be used for the "current" state of an agent at time t_0 in iteration k. It uses the following definitions:

$$x_{i0} = q_i^k(t_0) \tag{5.3}$$

$$\dot{x}_{i0} = \dot{q}_i^k(t_0) \tag{5.4}$$

$$\ddot{x}_{i0} = \ddot{q}_i^k(t_0) \tag{5.5}$$

Also of note, is that we did not explicitly define the maximum speed bound for an agent, as it can be expressed through the set of valid accelerations, $A(v)$. The set $A(v)$ should be chosen to reflect the agent's physical constraints on acceleration, as well as encoding the maximum achievable or allowable velocity. This can be represented by restricting the set of accelerations to those that would not exceed the maximum velocity after time C, or in other words:

$$\forall_{u \in A(v)} \, \|v + Cu\| \leq V_{\max} \tag{5.6}$$

Although in the safety algorithms presented below, no particular form for $A(v)$ is assumed, practical implementations must choose a representation. In modelling a holonomic robot, a "traction circle" has been found to work well in practice [52,77]. This corresponds to an $A(v)$ which is circular (or spherical in higher dimensions) with some radius F, and centered on

zero in acceleration space. However, some robots may be able to decelerate faster than their maximum acceleration ($D > F$), and thus it should be included in the model if possible. One way of doing this is a "partial ellipse", where $A(v)$ is a union of a traction circle of radius F and half of an ellipsoid with a major axis of D and a minor axis of F. The half-ellipse is oriented with its major axis away from the current velocity. A representative acceleration space plot of this shape is shown in Figure 5.2. While both the traction circle and the partial ellipse are only approximations to the true set of possible accelerations (see Sherback et al. [77] for a detailed study), the partial ellipse model has been found to be a good approximation for the CMDragons robots. If $D = F$, the partial ellipse model is equivalent to the traction circle. With either model, $A(v)$ must still be modified to exclude velocities that exceed the maximum bound as described above.

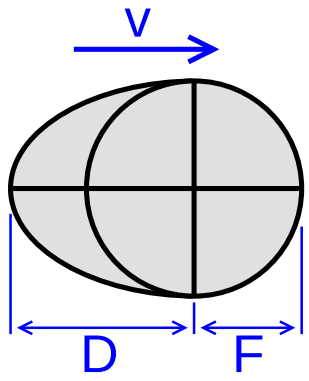


Figure 5.2: The model of $A(v)$ used in applying DSS to small-size soccer robots. F is the maximum acceleration and D is maximum deceleration. It is oriented by the current velocity v.

Finally, we can give a formal definition of safety. This is a refinement of the weaker notion of safety at a particular time expressed in Equation 1.2, as this new version uses a bounded radius model for robots, and even more importantly it describes safety at all times past a particular point, rather than only at a particular moment in time. This "strong" notion of safety is represented as the following function S, which is comprised of two parts. S_e is a boolean function indicating safety between a robot i and the environment. S_r is a safety function indicating safety between n mobile robots. Overall safety in a domain consisting of a static environment and mobile agents is described by the function S, which is a conjunction of S_e and S_r.

$$
\begin{aligned}
S_e(k, t_0) &= \forall_{i \in [1,n]} \, \forall_{t > t_0} \, q_i^k(t) \in C_{free} && (5.7) \\
S_r(k, t_0) &= \forall_{i,j \in [1,n], i \neq j} \, \forall_{t > t_0} \, \|q_i^k(t) - q_j^k(t)\|^2 \geq (R_i + R_j)^2 && (5.8) \\
S(k, t_0) &= S_e(k, t_0) \vee S_r(k, t_0) && (5.9)
\end{aligned}
$$

S_e indicates if an agent i stays in the free configuration space after some time t_0, and thus if true, guarantees the agent has not collided with an obstacle within the configuration space. S_r indicates if the Euclidean distance between any two distinct agents is always at least the sum of their safety radii. Thus if true, it guarantees the agents never pass close enough to collide after t_0. The following section details the Dynamic Window algorithm, which attempts to maintain S_e for a single agent. It is followed by the Dynamic Safety Search algorithm, which guarantees for a set of cooperating robots that S true in one iteration remains true in subsequent iterations, while the robots attempt to execute externally specified goals.

5.2 The Dynamic Window Approach

The goal of the safety system is to act as a post-process to motion control, which normally does is not concerned with avoiding obstacles, instead focusing only on convergence given a goal and the agent's dynamics. Safety could be handled at the motion planning level, subsuming all obstacle avoidance, motion control, and dynamics safety into a single algorithm. However, current solutions treat the resulting problem as a higher dimensional planning problem, resulting in exponential complexity in the number of agents. Thus what is desired is a polynomial complexity algorithm practical for multi-agent teams, and while not complete, can still guarantee safety algorithmically. Our method will extend a single-agent algorithm to reach these goals.

The "Dynamic Window" approach [38] is a search method which elegantly solves the problem of collisions between a robotic agent and the environment. It is a local method, in that only the next velocity command is determined, however it can incorporate non-holonomic constraints, limited accelerations, maximum velocity, and the presence of obstacles into that determination. It can thus provide for safe motion for a robot in a static domain. The search space is the velocities of the robot's actuated degrees of freedom. Fox et al [38] derived the case of a synchro-drive robots with a linear velocity and an angular velocity, while Brock et al [12] developed the case of holonomic robots with two linear velocities. Both methods use the concept of a "velocity space" where actions can be tested for safety. each point in

the velocity space corresponds to reaching that velocity within the control period C, and then executing a stop at a deceleration of D until the agent has come to a complete halt. A velocity can considered safe if the robot can travel up to that command during C and then stop without hitting an obstacle. This corresponds to not hitting an obstacle during C and then not hitting an obstacle during the stop (which traces out a line in world coordinates). It would be difficult to search all of velocity space to find the action that minimizes a cost metric, however, due to limited accelerations, velocities are limited to only a small window that can be reached within the acceleration limits of $A(v)$ over the control cycle. An example of a velocity space with an obstacle and an acceleration window is shown in Figure 5.3 [1].

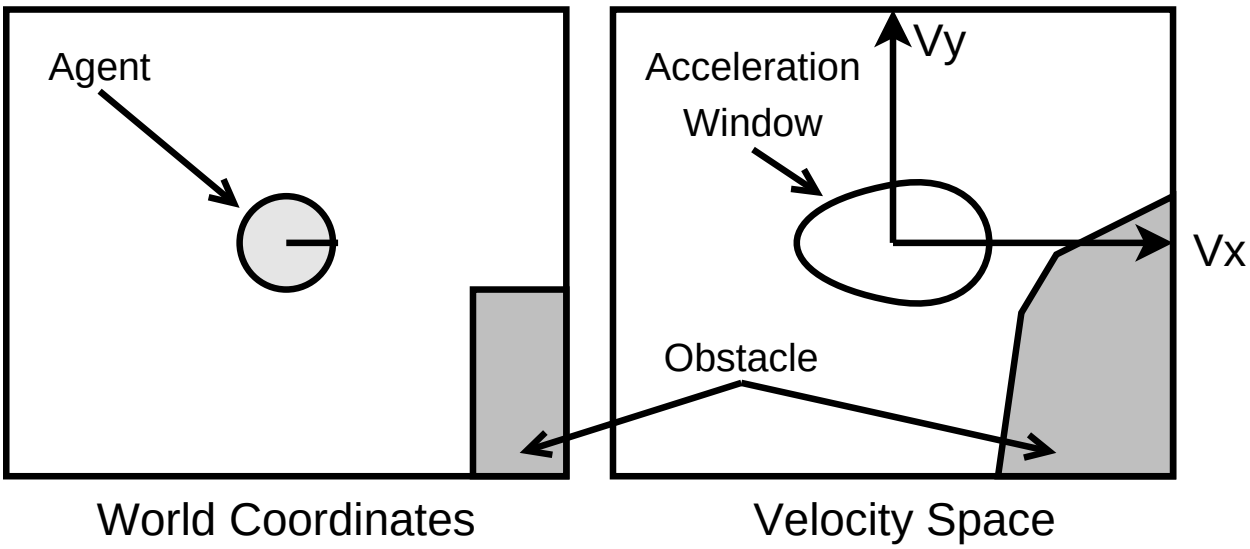


Figure 5.3: Example environment shown in world space and velocity space. Note that this figure is hand-drawn and thus only approximate.

In order to find a safe command, the Dynamic Window method creates a grid over the acceleration window, and evaluates each grid cell with a combination of a safety test and an evaluation metric. The safety test checks if a velocity command would hit an obstacle, resulting in an infinite cost. If the command is safe, then it is evaluated based on heuristics for reaching a desired target (Dynamic Window includes both safety and motion control in a single algorithm). Pseudocode for a variant is given in Table 5.1. The variation is that we have replaced velocity space with a related concept of an acceleration space. The acceleration space is defined using the possible accelerations in the robot model $A(v)$ for one control cycle C, with the results as defined in the transfer functions f and $\dot{f}$. In the code,

[1]Note that the velocity obstacle on the right of the figure was hand drawn rather than calculated, and is thus only approximate

the function *CheckSafetyObs* calculates the position after the control cycle (lines 1-2), and then calculates the stopping deceleration and the time required to stop (lines 3-4). If both of these trajectories are safe, then the command can be executed safely[2]. The main search function, *DynamicWindowSearch* searches a grid for the lowest cost acceleration command that is safe. In Brock et al [12], the Dynamic Window approach was used successfully for a robot moving up to 1m/s in cluttered office environments with dynamically placed obstacles.

function *CheckSafetyObs*(*i*:*RobotId*,u') : *Status*
1 **let** $p_1 = f(q_i, \dot{q}_i, u', C)$
2 **let** $v_1 = \dot{f}(\dot{q}_i, u', C)$
3 **let** $h = -D\frac{v_1}{\|v_1\|}$
4 **let** $t_\Delta = \frac{\|v_1\|}{D}$
5 **let** $p_2 = f(p_1, v_1, h, t_\Delta)$
6 **return** *CheckObsLine*(q,p_1,$R+\epsilon$)=Safe $\wedge$ *CheckObsLine*(p_1,p_2,R)=Safe

procedure *DynamicWindowSearch*(*i*:*RobotId*)
1 **let** G = *SampleUniformGrid*($A(\dot{q})$)
2 $e_i \leftarrow \infty$
3 **foreach** $u' \in G$ **do**
4 **if** *CheckAccel*(i,u)=Safe $\wedge$ *Cost*(u') $< e_i$ **then**
5 $u_i \leftarrow u'$
6 $e_i \leftarrow$ *Cost*(u')
7 **end**

Table 5.1: The Dynamic Window method for a single agent

5.3 Dynamics Safety Search

Our approach, called *Dynamics Safety Search* or DSS, extends the Dynamic Window approach to multiple velocity and acceleration bounded robots, and replaces the grid-based sampling with a randomized sampling approach which guarantees the preservation of safety if no sensor or action noise is present. It operates for n robots with the model described in Section 5.1. In Table 5.3, Dynamics Safety search is compared against the Dynamic Window

[2]A small positive real number, ϵ, is used to ensure that the parabolic path traced during the control period is included within the straight-line obstacle check. For typical short control cycles $C < 0.1s$, the effect is minimal. Alternatively, a parabolic trajectory check could be implemented by the collision detection library.

approach on which it is based as well as a hypothetical complete motion planner operating on the joint configuration and space defined by the n agents. Compared to Dynamic Window, DSS supports an exact guarantee of safety, and does not require a sufficient resolution in order to find a solution. DSS also supports multiple agents running the algorithm, and supports avoidance of uncontrolled moving obstacles (but without safety guarantees). Finally DSS does not use a fixed grid or resolution, and thus can operate as an anytime algorithm (only constant overhead per agent). Compared to a joint state-space planner, DSS is not complete, and is not guaranteed to find solutions avoiding moving obstacles. However, DSS can operate as an anytime algorithm with $O(n^2)$ complexity, rather than the non-anytime exponential complexity of the planner. If we replace the complete planner with a randomized variant, we can achieve better complexity guarantees but give up both completeness and guaranteed safety. DSS thus occupies a middle ground that is intended to be practical for implementation of multi-robot teams.

	Dynamic Window	Joint Planning	Dynamics Safety Search
Complete	No	Yes	No
Safety guarantee	Resolution	Exact	Exact
Multiple agent support	No	Yes	Yes
Moving obstacle support	No	Yes	Partial
Anytime algorithm	No	No	Yes
Multi-agent complexity	N/A	Exponential	$O(n^2)$

Table 5.2: A comparison of properties of Dynamic Window, explicit planning, and Dynamics Safety Search

The easiest way to understand the DSS algorithm is in a top-down manner. The high-level routines are listed in Table 5.3. The main function *DynamicsSafetySearch*, runs once per decision cycle (not per robot). It starts each iteration by setting each agent's command in u_i as the stopping deceleration if the agent is moving (lines 3-5). This is guaranteed to be a safe action to perform by the previous iteration. An evaluation cost based on the difference between this stop command and the desired acceleration a_i^k is stored in e_i (line 6). This will be used to rank alternatives to find one which most closely matches the desired command. The current approach uses squared Euclidean distance as the metric. Next, the duty cycle for the command, γ_i is set to ensure that the agent comes only to a stop and does not begin to accelerate in the opposite direction (line 7). For small control periods, such as $C < 0.1s$, the effect is negligible, but it is necessary for completeness. In practical implementations, γ_i is almost always equal to C. After this "initialization to stop" phase from (lines 1-7), *DynamicsSafetySearch* then calls a helper function *ImproveAccel* for each agent, which will try to better match the agent's desired action while maintaining safety.

```
function CheckAccel(i:RobotId, u′:Vector) : Status
1    if CheckSafetyObs(i,u′)=Unsafe
2        then return Unsafe
3    foreach j ∈ [1, n], j ≠ i do
4        if CheckRobot(i,u′,j,u_j)=Unsafe
5            then return Unsafe
6    return Safe

procedure ImproveAccel(i:RobotId)
1    if CheckAccel(i,a_i^k)=Safe then
2        u_i ← a_i^k
3        e_i ← 0
4        γ_i ← C
5    else
6        foreach j ∈ [1, m]
7            u′ ← RandomAccel(q̇_i)
8            if CheckAccel(i,u)=Safe and ‖u′ − a_i^k‖² < e_i then
9                u_i ← u′
10               e_i ← ‖u′ − a_i^k‖²
11               γ_i ← C
12           end
13   end

procedure DynamicsSafetySearch()
1    foreach i ∈ [1, n] do
2        let s = ‖q̇_i‖
3        if s > 0
4            then u_i ← −D q̇_i / s
5            else u_i ← 0⃗
6        e_i ← ‖u_i − a_i^k‖²
7        γ_i ← min(s/D, C)
8    foreach i ∈ [1, n] do
9        ImproveAccel(i)
```

Table 5.3: The high level search routines for velocity-space safety search.

The procedure *ImproveAccel* consists of two major stages. In the first stage, it checks to see if the agent's desired command can be carried out without causing any failure in the safety. It does this by calling the *CheckAccel* function which returns whether or not using a particular acceleration as a command maintains safety for all of the agents. If *ImproveAccel* finds that the desired command can be safely executed (line 1), then it sets that command in u_i (line 2) to be carried out for the duration of the control cycle (line 4). The evaluated cost is zero, since the current command matches the desired command. As a result, further search is not necessary as no better matching command could be found. In practical implementations, it is this short-circuit that results in the efficiency of the algorithm. If the agents are not interfering with each other or close to C-space boundaries, it is normally the case that the agent's desired action will be safe. If however, the desired action cannot be performed, a search is carried out (lines 6-12), which tries to find an acceleration that is safe, but with a lower evaluated cost. In line 7, a random acceleration is sampled from the set of accelerations possible at the current velocity ($A(v)$ in the robot model). This is checked for safety, as well as having a lower cost than the current action (line 8). If both these conditions are met, the action is set and the evaluation cost calculated (lines 9-11).

For the high-level search routines, this leaves the function *CheckAccel*. It returns if a give acceleration is safe, and has a straightforward implementation based on the definition of safety in equations 5.7-5.9. Safety with respect to the environment is handled using the same function as we used in the implementation of the Dynamic Window approach (lines 1-2). Robot agents are handled by checking the new action with each other robot using the function *CheckRobot*, which is described below (lines 3-5). If neither of these checks finds a violation of safety, the action is considered safe (line 6).

In Figure 5.4, and example run of DSS for a single control cycle can be seen. Each agent is a shaded circle, with the desired acceleration shown as an arrow in the first frame. An extruded circle "pill shape" denotes the trajectory of the robot including its action and stopping phases. Because DSS never violates the safety of an existing trajectory, the extruded area can be thought of as a reserved space to stop [3]. The stopping trajectory also indicates the current velocity of the agent, which is along the same direction as the stopping trajectory. To the right of each of the depictions of the environment, a plot of the horizontal velocity with time is given for both agents. In part (a), each agent has its action initialized to stop, and the arrows display the desired acceleration command. The velocity plot shows both agents decelerating to a complete stop. In part (b), agent 1 has chosen a safe action (depicted by the blue arrow) which is as close as possible to the desired action (shown in light gray). The effect on the velocity plot is to accelerate up to time C, and then stop within a new

[3]Although it is a useful approximation to think of these areas as non-overlapping, it is not the case that these areas are necessarily disjoint. It is only the case that at any given point in time, the robots cannot occupy the same circles defined by their safety radius.

reserved stopping area. Agent 1's action was constrained by the rectangular environment obstacle, so it could not match the desired acceleration. In part (c), agent 2 chooses an action depicted by the dark arrow, which matches the desired action as closely as possible while maintaining safety. The effect on the horizontal velocity is shown on the right. Agent 2's action is constrained by the need to avoid hitting agent 1 during the stopping trajectory. In part (d), the agents have advanced by time C and have executed their actions. The remaining trajectory is a valid stopping action. A new iteration of the algorithm can begin. Thus, using DSS attempts to iteratively delay stopping into the future with each iteration, while always maintaining that as an action to fall back on if needed. This is similar to the Dynamic Window approach, but because the stopping action is treated specially, rather than integrated into the grid search as in Dynamic Window, DSS can always guarantee finding a safe action.

We now proceed in describing the lower layers of the algorithm. In Table 5.4, the mid-level functions for DSS are shown. The function *MakeTrajectory* is used to construct a trajectory for an agent starting with a given acceleration u_i. Mathematically it is constructing the piecewise components of the function $q_i^k(t)$. Line 1-3 calculate the time and the resulting position and velocity of executing action u_i, starting from the current state of the robot in x_{i0} and $\dot{x}_{i0}$. The system transfer functions are used to perform the forward prediction of position and velocity (Equations 5.1 and 5.2). In lines 4-6, the results of a stopping action are calculated. h_i is acceleration opposite the agent's current velocity with a magnitude of D, while t_s is the time when the agent will come to a stop. Again the system transfer function is used to compute the position, and the velocity will be zero. Based on the acceleration model of the robot, each of these three phases of motion for the robot (control, stopping, stopped) define a trajectory from t_0 onward, and during each segment the acceleration is constant. Thus the entire trajectory can be modelled as parabolic segments. For this purpose, the parabolic tuple is defined, with the order of members being the position, velocity, and acceleration vectors, respectively, followed by the time interval for which that parabolic motion segment is defined. Lines 7-9 of *MakeTrajectory* construct parabolic tuples for the three phases of motion. These three items are then used the construct a trajectory tuple (Line 10).

Given the function *MakeTrajectory*, and a primitive for checking parabolic segments for collision, the implementation of *CheckRobot* is as follows. Lines 1-2 construct the trajectories for agents i and j. Then each pair of parabolic segments is checked against one another using the primitive checking function *CheckParabolic* (Lines 3-6). The function *CheckParabolic* returns unsafe if two parabolic segments pass within a certain distance of one another (in this case the sum of i and j's safety radii). If no collision is found between any pair of segments, then the actions u_i and u_j are safe in the sense of S_e (Equation 5.7) and the *Safe* status is returned (Line 7).

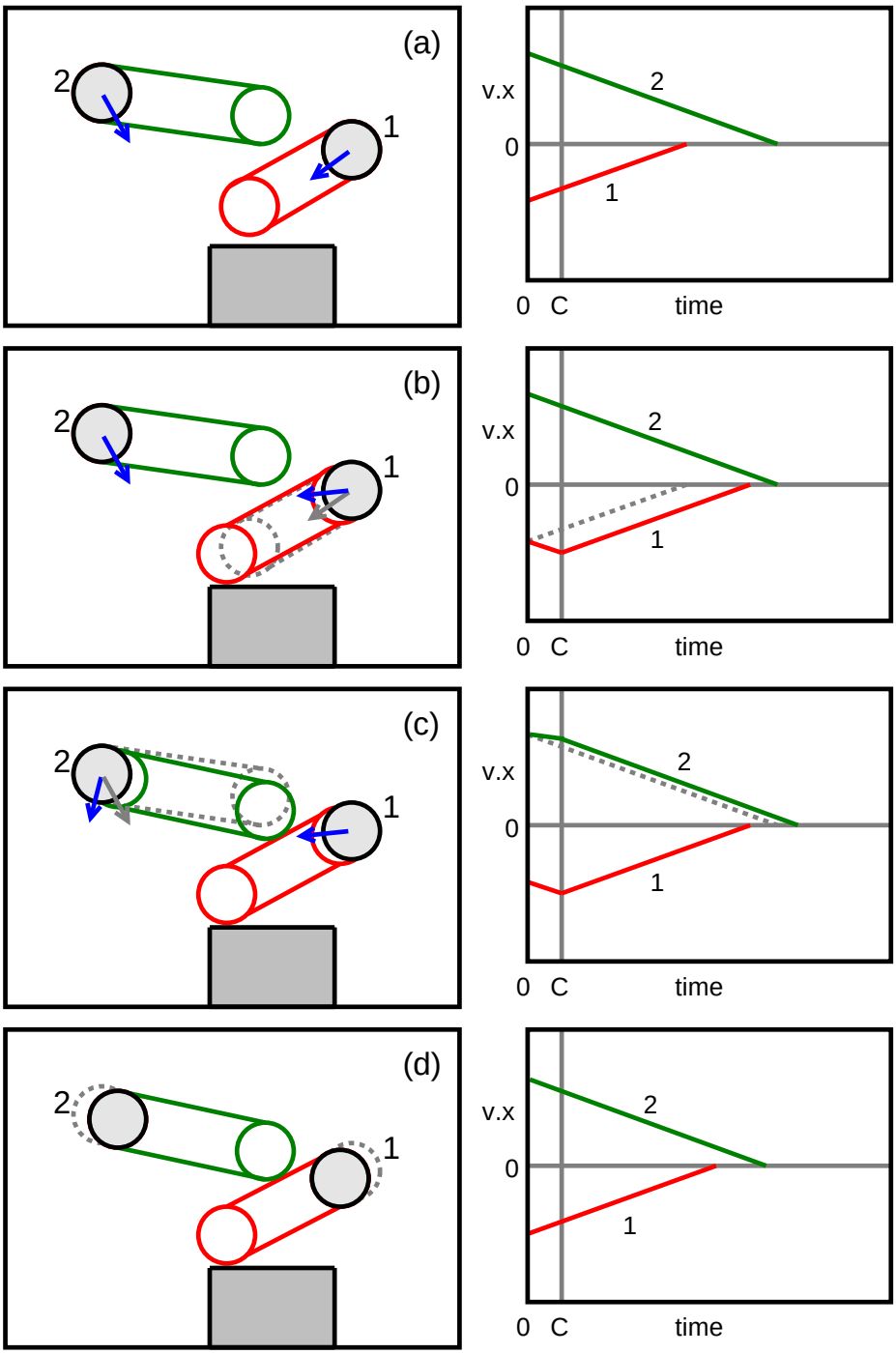


Figure 5.4: An example of an iteration of DSS with two agents. Each agent starts by assuming it will stop (a), and then each agent chooses an action (b)-(c), while making sure the action will allow a safe stop afterward. Finally, the actions are executed (d) and the agents can safely assume that stopping is a valid action.

tuple *Parabolic* = (*Vector* * *Vector* * *Vector* * [*Time*, *Time*])
tuple *Trajectory* = (*Parabolic* * *Parabolic* * *Parabolic*)

function *MakeTrajectory*(*i*:*RobotId*, u_i:*Vector*) : *Trajectory*
1 **let** $t_c = t_0 + C\gamma_i$
2 **let** $x_{i1} = f(x_{i0}, \dot{x}_{i0}, u_i, t_c - t_0)$
3 **let** $\dot{x}_{i1} = \dot{f}(\dot{x}_{i0}, u_i, t_c - t_0)$
-
4 **let** $h_i = -D\frac{\dot{x}_{i1}}{\|\dot{x}_{i1}\|}$
5 **let** $t_s = t_c + \frac{\|\dot{x}_{i1}\|}{D}$
6 **let** $x_{i2} = f(x_{i1}, \dot{x}_{i1}, h_i, t_s - t_c)$
-
7 **let** $P_{i0} = Parabolic(x_{i0}, \dot{x}_{i0}, u_i, [t_0, t_c])$
8 **let** $P_{i1} = Parabolic(x_{i1}, \dot{x}_{i1}, h_i, [t_c, t_s])$
9 **let** $P_{i2} = Parabolic(x_{i2}, 0, 0, [t_s, \infty])$
10 **return** *Trajectory*(P_{i0},P_{i1},P_{i2})

function *CheckRobot*(*i*:*RobotId*, u_i:*Vector*, *j*:*RobotId*, u_j:*Vector*) : *Status*
1 **let** (P_{i0},P_{i0},P_{i0}) = *MakeTrajectory*(i,u_i)
2 **let** (P_{j0},P_{j1},P_{j2}) = *MakeTrajectory*(j,u_j)
-
3 **for** $a = 0$ **to** 2 **do**
4 **for** $b = 0$ **to** 2 **do**
5 **if** *CheckParabolic*(P_{ia}, P_{jb}, $R_i + R_j$) = *Unsafe*
6 **then return** *Unsafe*
7 **return** *Safe*

Table 5.4: Robot-robot checking primitive for safety search.

The pseudocode for the primitive function *CheckParabolic* is shown in Table 5.5. After assigning names to the components of the parabolic tuples (lines 1-2), the function calculates the overlapping time interval of the trajectories (line 3). If there is no overlap in the time intervals, there is by definition no time at which a collision could occur, thus the parabolic segments are safe with respect to one another (line 4). In lines 6 and 7, trajectory functions are defined (matching the system transfer function f in the robot model). Next, a fourth-order polynomial "clearance" function $d(t)$ is defined, which is equal to the squared distance between the parabolic function minus the squared total radius. Any value of $d(t)$ less than zero over the interval T indicates a collision, a zero value indicates constant, and the absence of any such values indicates a safe trajectory. Thus the last block of the function (lines 9-15) implements the classical function minimization technique for differentiable functions over an interval. First, lines 9-10 check the boundaries of the interval, and next the real roots of the derivative of $d(t)$ with respect to t are determined. Any root within the time interval T with a non-positive value of $d(t)$ results in the status *Unsafe* being returned (lines 12-14). If none of the extrema are unsafe, the function returns a status of *Safe* (line 15). The approach taken in *CheckParabolic* defining $d(t)$ is similar to the "relative velocity" method of Fiorini et al. [36] for an agent planning among multiple moving obstacles.

```
function CheckParabolic(P1:Parabolic,P2:Parabolic,r:ℝ) : Status
1     let (x1, ẋ1, u1, [t1a, t1b]) = P1
2     let (x2, ẋ2, u2, [t2a, t2b]) = P2
-
3     let T = Intersection([t1a, t1b],[t2a, t2b])
4     if T = ∅ then return Safe
5     let [ta, tb] = T
-
6     let p1(t) = x1 + ẋ1(t − t1a) + ½u1(t − t1a)²
7     let p2(t) = x2 + ẋ2(t − t2a) + ½u2(t − t2a)²
8     let d(t) = ‖p1(t) − p2(t)‖² − r²
-
9     if d(ta) ≤ 0 or d(tb) ≤ 0
10        then return Unsafe
11    let M = RealRoots(Deriv(d(t),t),t)
12    foreach t ∈ M do
13        if t ∈ T and d(t) ≤ 0
14            then return Unsafe
15    return Safe
```

Table 5.5: The parabolic trajectory segment check.

Overall, the Dynamics Safety Search algorithm is a fairly involved, but highly modular approach. Each of the subfunctions has a well defined interface and semantics. While this method is not complete in a kinodynamic planning sense, it scales polynomially with the number of robots, and can guarantee safety among multiple moving agents with realisitic motion constraints. The achievement of objectives is difficult to analyze, since DSS uses a reactive and oppurtunistic method for achieving goals within its safety assumptions. However when paired with a path planner which already lacks completeness, the loss may not prove as problematic. In particular, the assumptions of DSS fit well with applications where achievement of task objectives is secondary to safe operation.

5.4 Guarantee of Safety

Ultimately, we want to satisfy the notion of safety described by Equation 1.2 from Chapter 1.3.2. However, that definition of safety is "weak" in that it only considers the position at the current time, and thus it is difficult to show that this definition is met while dynamics constraints are respected. Thus DSS defines the stronger definition of safety embodied in Equation 5.9, where safety is modelled over an entire future trajectory from the current time forward. The DSS safety guarantee applies to agent bounded by a radius, rather than the more general set theoretic definition in Equation 1.2. Within the case of such a circular or spherically bounded agent however, the DSS guarantee of Equation 5.9 is strictly inclusive of the weak definition.

DSS maintains safety by treating Equation 5.9 as an invariant during all operations on the world state. The statement of safety embodied in it defines exactly the kind of safety we would like an algorithm to provide; at all times it maintains a future trajectory that can safely be executed indefinitely. Of course, if a problem instance for DSS starts out as unsafe, there is no guarantee that DSS can return the system to safety. Thus the guarantee we can seek from the safety search method is that it can *maintain* safety if started from any safe situation. In particular, we wish to demonstrate that:

Theorem 5.4.1. *For n robots given the model from Section 5.1, if $S(k, t_0)$ holds, then after time C, and the execution of DynamicsSafetySearch, then $S(k+1, t_0+C)$ holds.*

Due to the complexity of DSS, a full proof of the algorithm would be very long, thus the proof is only sketched here in enough detail to justify the claims of safety. Operations on the world state for DSS fall into two main categories, both of which maintain the strong safety invariant:

- The passage of time C maintains the safety invariant if new actions are set to stopping the agent
- Any modification of actions performed by *ImproveAccel* maintains the safety invariant

These two operations cover everything occurring in the top level procedure *DynamicsSafetySearch*. For the first type of system transition, the passage of time is governed by the system transfer functions (Equations 5.1-5.2. Immediately following this, the first part of *DynamicsSafetySearch* sets a stopping action. The other type of transition is a "decision cycle" where the latter part of *DynamicsSafetySearch* calls *ImproveAccel* on each agent. If both of these operations maintain safety individually, the safety invariant overall is maintained. We will first focus on time passage, and then follow with action modification.

In order to show that advancing time does not cause the safety invariant to fail, the critical point is that trajectory $q_i^{k+1}(t)$ follows the path of $q_i^k(t)$ after time C has passed, allowing the safety at iteration k to imply safety at the next iteration. In other words, given a trajectory function q_i^k defined as the following:

$$q_i^k(t) = \begin{cases} q_i^k(t,0) & \text{if } t \in [t_0, t_c] \\ q_i^k(t,1) & \text{if } t \in [t_c, t_s] \\ q_i^k(t,2) & \text{if } t > t_s \end{cases} \tag{5.10}$$

where $t_c = t_0 + C$, and the individual components of q represent the parabolic segments P created in *MakeTrajectory*, we want to show that:

$$q_{i+1}^k(t) = \begin{cases} q_i^k(t,1) & \text{if } t \in [t_c, t_s] \\ q_i^k(t,2) & \text{if } t > t_s \end{cases} \tag{5.11}$$

The can be shown to be the case because the calculations in line 1-3 of *MakeTrajectory* exactly mirror the system transfer functions f and $\dot{f}$. As a result, $q_{i+1}^k(t)$ follows the "tail" of $q_i^k(t)$ for any time after $t_0 + C$. With this fact holding for all robots, and the assumption that all other obstacles are static, any safety guarantees for the original trajectory functions will carry over. Thus, for any trajectory constructed in the previous frame by *MakeTrajectory*, the passage of time will maintain the safety invariant. Of course, if a new action is not chosen, we must show that the stopping action from *DynamicsSafetySearch*. The critical point there is that lines 1-3 construct a function that matches Equation 5.11, with the the

action set to the derivative of $q_i^k(t, 1)$. Thus, overall the safety invariant in maintained by advancing time.

Next, we argue that *ImproveAccel* maintains the invariant. This is straightforward, as both cases where an action is set in the function *ImproveAccel* are guarded by conditional statements asserting *CheckAccel* returns a status of *Safe*. The function *CheckAccel* can be shown to mirror the definition of S in Equation 5.9. Both parts of S are expressed, as lines 1-2 matches S_e (Equation 5.7) while lines 3-5 matches S_r (Equation 5.8)

DSS maintains the safety invariant from Equation 5.9 at all times, and thus satisfies the weaker definition of safety at any time instant expressed in Equation 1.2. This guarantee rests on the assumption that all moving agents are participating in the algorithm, and that the obstacles in the configuration space are otherwise static. Also, there can be no sensory error in the positions and velocities, and no action error as expressed in the system transfer functions (Equations 5.1-5.2).

5.5 Improving Efficiency

For a practical implementation, a few approaches can lead to significant gains in the execution speed of the algorithm. The first approach is to cut down on the number of parabolic segments that need to be checked against one another as a nested loop in the function *CheckRobot*. In a typical situation for a pair of robots, the plot of speed versus time would look similar to Figure5.5. In the common case where $\gamma = 1$, the control periods for the two agents are identical, so the first two parabolic motion segments must be checked against one another, but will not overlap with any other segments. After that, both agents will be decelerating, so the "stopping" parabolic segments will need to be checked against one another. Finally, the second agent to come to a stop much check its stopping parabolic segment against the "stationary" segment of the other agent. After this time, both agents are stopped, and the status of their safety will not change. Using this approach, the number of parabolic pairs that need to be checked drops from nine to three. Even in the case where $\gamma < 1$, a similar approach means that only four checks need to carried out.

Another helpful approach is to use broad-phase collision detection techniques as explored in Chapter 3, and in particular the Extent Masks approach described in 3.2.1. A bounding box can be associated with each trajectory tuple, and *CheckRobot* can first check for overlap between the bounding boxes before proceeding with a trajectory check. In cases with many agents, where only a few pairs of agents are within close proximity, this can result in a large speedup. Taking the broad-phase approach further, based on the structure of $A(v)$ one

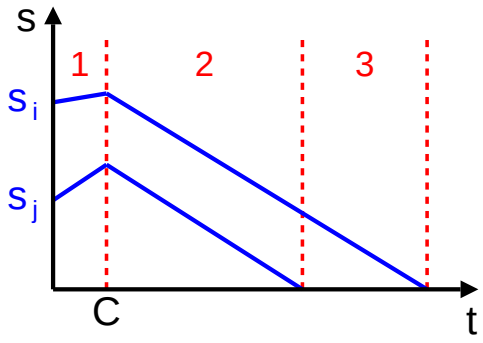


Figure 5.5: Example velocity profile of two agents i and j. Each agent starts at a distinct velocity and executes a control acceleration for time C, and then comes to a stop using deceleration D. This defines three segments of relative motion, each with a constant acceleration.

could construct a bounding box which included *all possible* trajectories, or a box B_i such that for any $u \in A(\dot{x}_{i0})$, the resulting trajectory function $q_i'(t)$ lies within B_i. Any agent or obstacle outside this bounding box need never be checked for safety. The agents and obstacles overlapping the box could be placed in a "potentially colliding set" calculated in the function *ImproveAccel*, placing it outside the search loop for accelerations. This optimization would likely result in a large speedup for domains with many agents.

In the implementation of DSS used in the evaluation and application of the algorithm, both the parabolic pairs optimization and the trajectory bounding box optimization were incorporated. The potential colliding set optimization was not implemented due to the more invasive changes required for the algorithm and supporting libraries, as well as the sufficient performance of the current implementation.

Another question one can ask is what the minimum complexity of the general DSS approach could be. For n agents, m obstacles, and k random samples in the acceleration search, the worst-case complexity is of *DynamicsSafetySearch* $O(kmn^2)$. This results from the complexity of *CheckAccel*, which is $O(mn)$, which is called up to k times in *ImproveAccel*, which is thus $O(kmn)$. The n calls to *ImproveAccel* by *DynamicsSafetySearch* leads to the final complexity. The complexity of the top level procedures derive directly from the approach itself, leading the focus to trying to improve *CheckAccel* is possible. The scaling with obstacles is subject to the problems discussed in Chapter 3, however improving the n factor would be helpful for large numbers of agents by removing the n^2 factor. However, if the algorithm is called with an environment like that shown in Figure 5.6, no spatial data structure based on bounding volumes on trajectories will allow a sub-linear number of checks for a single agent. Thus, the complexity cannot be improved within the bounds of the DSS approach

unless addition constraints on parameters or environments are made. In practice however, with more uniform distributions of agents and obstacles, performance has been found to be adequate. This will be explored in detail in the next section.

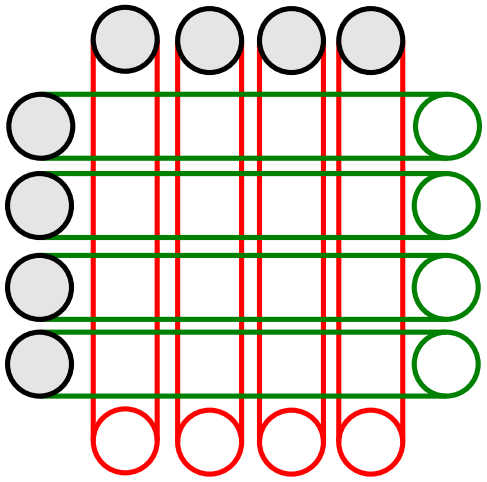


Figure 5.6: An example situation showing n agents with $\Omega(n^2)$ overlapping trajectories.

5.6 Evaluation and Results

The Dynamic Safety Search algorithm was implemented as a C++ object following the approach presented in the pseudocode, but with the optimizations described in Section 5.5. It was first tested in a simple simulator of an environment similar to the RoboCup small size league, but with more complicated environmental obstacles. It was subsequently added to the CMRoboDragons system in 2005 and an updated version in the CMDragons 2006 team.

5.6.1 Simulation Evaluation

The evaluation domain consists of up to ten simulated robots modelled after idealized RoboCup small size robots. The task is to alternately achieve goals on the left and right side of the environment with several obstacles. The domain used for testing is shown in Figure 5.7. The state pictured in the figure is just at the beginning of a test run, with the robots represented as filled circles and their respective goals represented as outlined circles. The straight line indicates the motion control target, and the remaining jagged path is the unoptimized result from an ERRT planner. The stop trajectory is represented as before, as

a swept circle, but it is not visible in the figure because the agents are not yet moving. In all the tests, the robots have a diameter of $90mm$ and environment is $5m$ by $4m$. Each robot agent has a command cycle of $C = 1/60$ sec, a maximum velocity of $2m/s$. The maximum acceleration is $F = 3m/s^2$, and the deceleration is $D = 6m/s^2$, and $A(v)$ is modelled as a partial ellipse (see Figure 5.2).

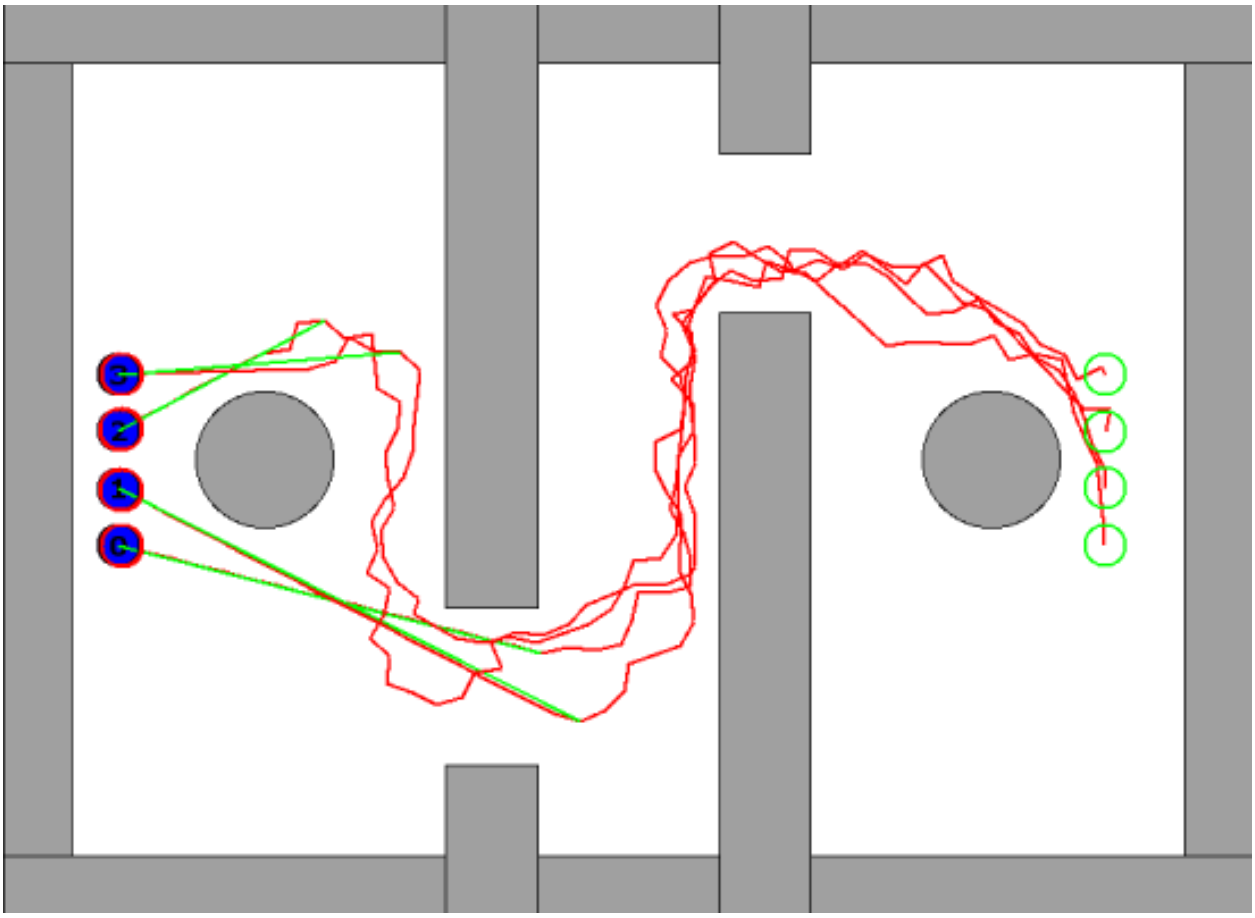


Figure 5.7: The evaluation environment for the DSS algorithm.

For the collision avoidance experiments, four robots were given the task of traveling from the leftmost open area to the rightmost open area, and back again for four iterations. Each robot has separate goal point separated from the others by slightly more than a robot diameter. Because the individual robots have differing path lengths to traverse, after a few traversals robots start interacting while trying to move in opposed directions. Figure 5.8 shows an example situation in the middle of a test run. On average, four full traversals by all of the robots took about 30 seconds of simulated time.

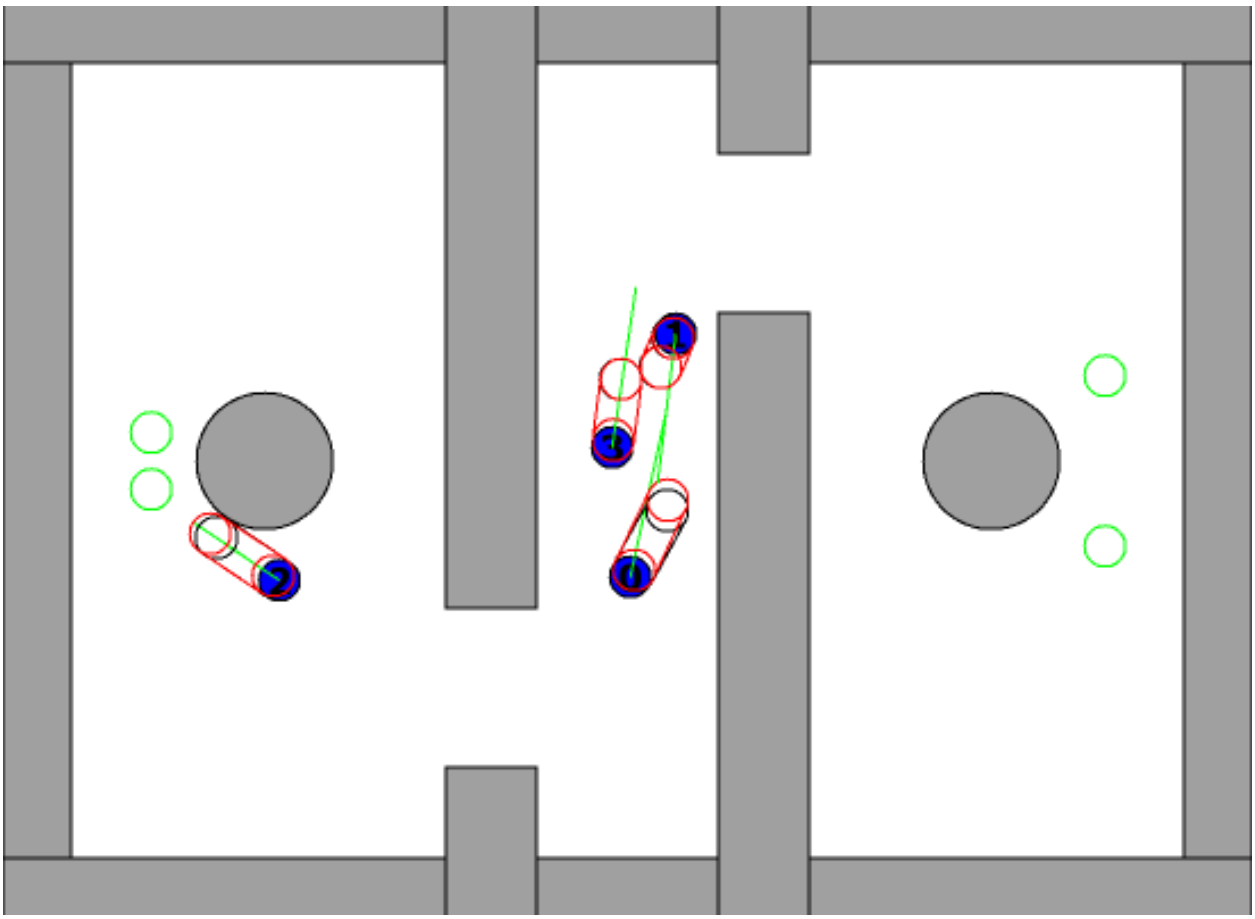


Figure 5.8: Multiple robots navigating traversals in parallel. The outlined circles and lines extending from the robots represent the chosen command followed by a maximum rate stop.

For the evaluation metric, we chose interpenetration depth with obstacles multiplied by the time spent in those unsafe states. This captures both the frequency and severity of collisions in a single metric. To more closely model a real system, varying amounts of position sensor error were added, so that the robot's reported position was a Gaussian deviate of its actual position [4]. This additive random noise represents vision error from overhead tracking systems. It also stresses the system by testing the random search; with Gaussian noise the stopping action is not guaranteed to be safe, and thus a search is required to return the system to safety. Velocity sensing and action error were not modelled in the simulation for simplicity; these errors depend heavily on the specifics of the robot and currently lack an accurate model with wide applicability.

For the first test, we compare two options which both use ERRT and a motion control system, but enable or disable the safety search. Each data point is the average of 40 runs (4 robots, each with 10 runs), representing about 20 minutes of simulated run time. The results are shown in Figure 5.9. It is evident that the safety search significantly decreases the total interpenetration time. Without the safety search, increasing the vision error makes little difference in the length and depth of collisions. Ideally the plotted curve without safety search would be smooth, but due to the random nature of the collisions it displays extremely high variance, and many more runs would be needed to demonstrate a dependence on vision noise. However, even with the noise, it is clear that the curve is significantly worse than when safety search is used. Next, we evaluated only the system with safety search enabled, but using varying extra margins of $1 - 4mm$ around the $90mm$ safety radius of the robots, plotted against increasing vision error (see Figure 5.10). As one would expect, with little or no vision error even small margins suffice for no collisions, but as the error increases there is a benefit to higher margins for the safety search, reflecting the uncertainty in the actual position of the robot. Thus this supports adding an extra margin of safety around the robots based on the expected noise. Such an approach was adopted for the real CMDragons robots.

The other variable of interest is the cost in running time of planning and the safety search. In the tests above, the ERRT planner was limited to 1000 nodes, and the safety search was limited to 500 randomly sampled velocities. The system executed with an average run time of $0.70ms$ per control cycle without the velocity safety search, and $0.76ms$ with it. Thus safety search does not add a noticeable overhead to the navigation. Since this is a real-time system however, we are most interested in times near the worst case. Looking at the entire distribution of running times, the 95th percentiles are $1.96ms$ without safety search and $2.04ms$ with it. In other words, for 95% of the control cycles, runtime was less than $2.04ms$ with safety search, and the execution time was only 4% longer than without safety search. Thus, for systems with an existing execution time budget for path planning, adding the

[4] A description of the CMDragons vision system, including experiments to determine the error model and the appropriate magnitudes appear in Appendix A. The particularly relevant section is A.2

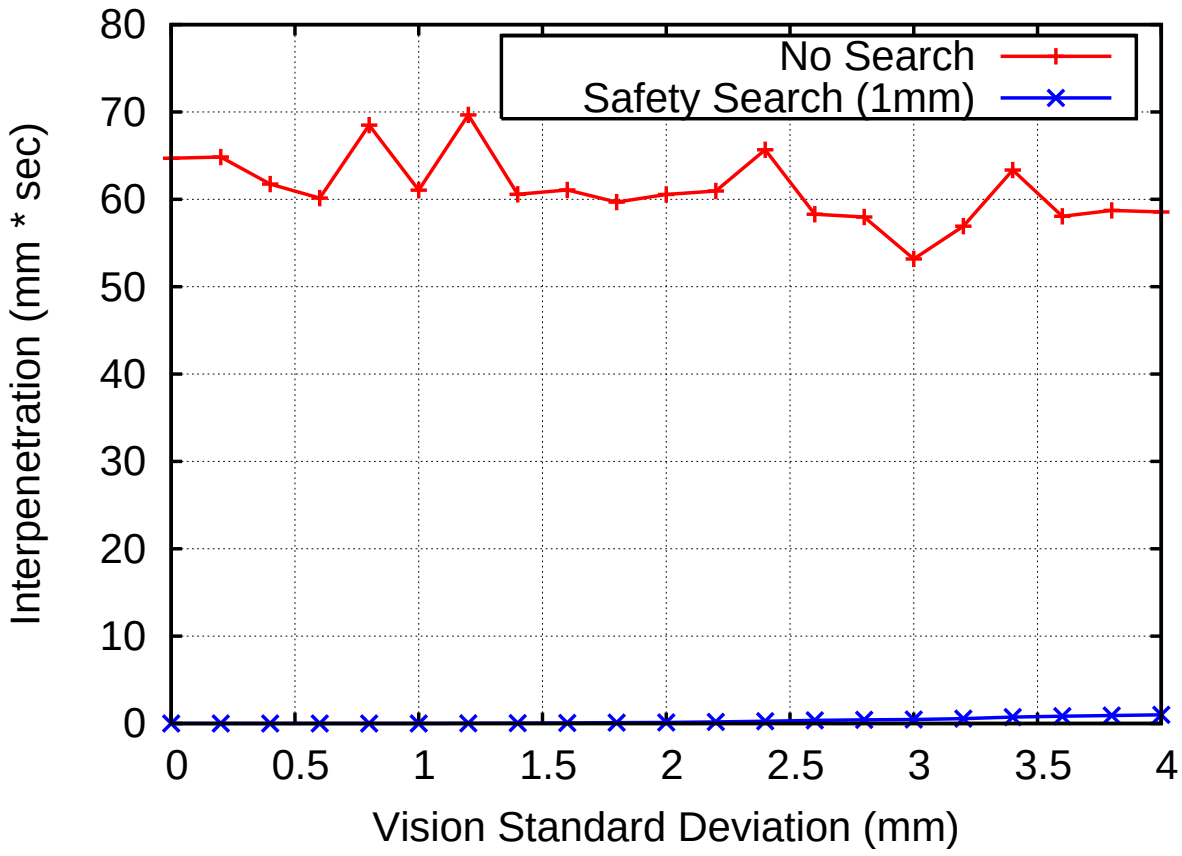


Figure 5.9: Comparison of navigation with and without safety search. Safety search significantly decreases the metric of interpenetration depth multiplied by time of interpenetration.

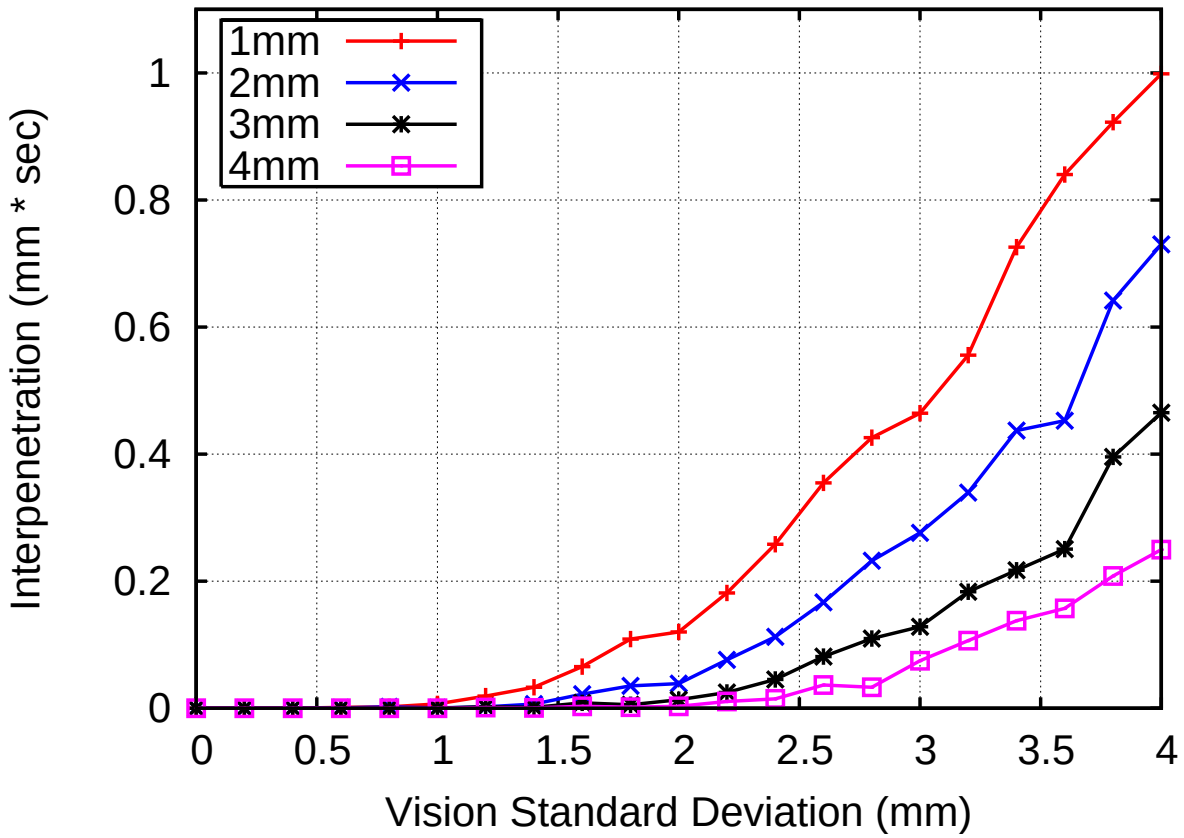


Figure 5.10: Comparison of several margins under increasing vision error. The four different margins used are listed in the key, while increasing vision standard deviation is plotted against the collision metric of interpenetration depth multiplied by time of interpenetration.

safety search is a small additional overhead.

Next, to measure the scalability of the safety search approach, the same traversal task was repeated while varying the number of robots from 1 to 10. With increasing numbers of agents in a fixed-size environment, we can hope to gauge how well the algorithm performs under increasing amounts of clutter due to moving objects. The timing results for safety search are shown in Figure 5.11. The function appears to scale in a roughly linear fashion for more than one agent, though it is too noisy to determine with any certainty. The most important observation is that it does not scale in a particularly super-linear fashion, which would cause difficulties for moderately large teams. As shown earlier, the worst case for DSS is $O(n^2)$, but such cases do not appear to arise in the experiment. The runtime cost of DSS demonstrates that it is applicable to control of agents at high rates of replanning. An equivalent method using joint state-space planning would need to encode at least position and velocity, resulting in a 40 dimensional problem for ten robots. The author is not aware of any current method which could approach 60 Hz replanning in the joint state-space.

5.6.2 Real Robot Evaluation

On the physical robots, objective measurement has proven difficult, although we have qualitatively noted that the frequency of collisions between teammates goes from several times a minute to once every several minutes, for regular soccer play, while it drops very significantly for tasks with highly conflicting navigation goals. In order to demonstrate the effect of DSS, a test domain was created where two pairs of robots would swap positions in a 2.8m traversal across a RoboCup field. An image sequence in Figure 5.12 shows the robots traversing the field without additional obstacles, while Figure 5.13 shows a traversal with five static obstacles in the environment. A video of this test is available on the supplemental materials web page [21]. During 60 seconds of testing, a only a single collision occurred between the moving robots. Minor contact also occurs between a robot and a static obstacle in the second stage of the test. By comparison, this test could not be run with DSS disabled due to the risk of damage to the robots. In limited testing with a decreased speed, the robots could not complete a single traversal without a collision while DSS was disabled.

A second application of DSS was as a post-process to a user tele-operation program. The driving program allows a user to set the target velocity for a robot using a joystick. DSS can be used as a post-process to the user specified command to maintain safety while trying to achieve the target velocity. This allows a safe method for tele-operation, even for novices [5].

[5]Having a novice operator damage a robot during tele-operation at a demo was one of the original motivations for implementing DSS

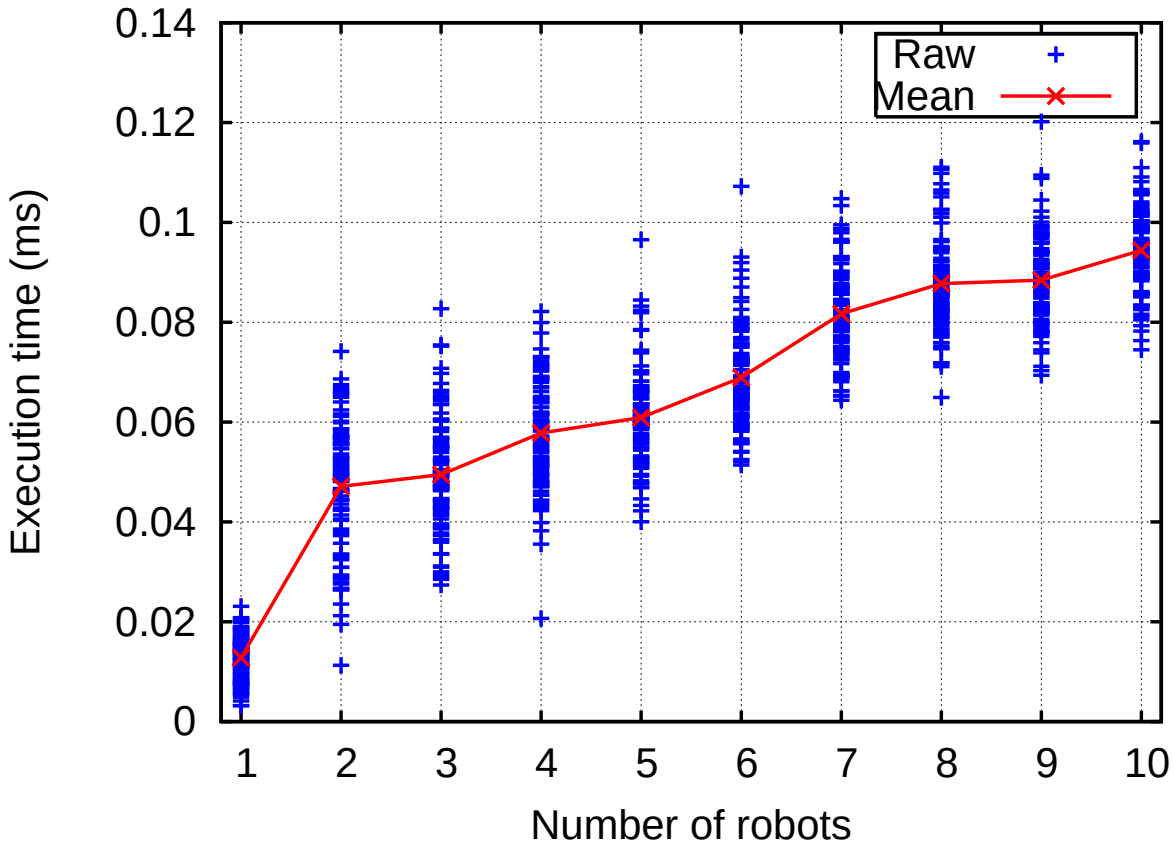


Figure 5.11: Average execution time of safety search for each agent, as the total number of agents increases. For each robot count, 100 trials of the left-right traversal task were run. Both the raw data and means are shown.

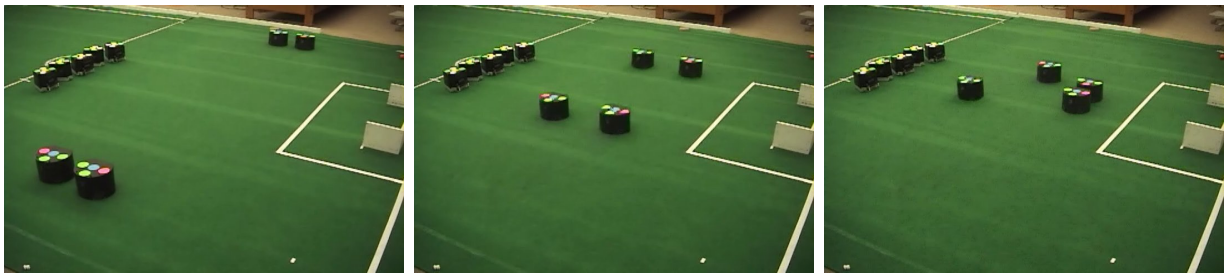


Figure 5.12: A sequence captured while running DSS with four CMDragons robots. All agents completed their 2.8m traversal within 3.1 seconds.

Figure 5.13: A sequence captured while running DSS with four CMDragons robots and five small static obstacles. All agents completed their 2.8m traversal within 2.8 seconds.

A video of this system is available in the supplemental materials [21]. At first, DSS is enabled and the user navigates among a field of static obstacles, with minimal avoidance required on the part of the user. In the second half of the video, DSS is disabled and an expert operator attempts to replicate the motions while avoiding obstacles. With DSS enabled, one collision occurs during 45 seconds of operation, while with DSS disabled, 18 collisions occur during the same time period.

In the RoboCup competitions, there is one impartial source of measurement for the safety of navigation systems. In the small-size league rules, a penalty can be called by a referee against a robot for pushing or hitting an opponent excessively, as well as for a non-goalie robot entering the team's own defense area. The DSS algorithm has been used by our teams for the past two years in the international RoboCup competition; An early version of the DSS algorithm was adopted for the 2005 team entry CMRoboDragons, and the version as described in this chapter was adopted for the 2006 entry CMDragons. In two years of play (13 games), our team received only a single navigational penalty, which occurred while DSS was disabled [6]. During this time, our team finished fourth and first in the tournament. None of the first place teams in 2003-2005 were able to complete the tournament without a pushing penalty.

In terms of execution time, the DSS algorithm has proved efficient when applied on the real robots. When applied during either the traversal task or soccer gameplay, the safety search contributed to less than 1% to overall navigation execution time. This is because the in the more open RoboCup environment, close proximity between agents and with obstacles is rare, allowing DSS to avoid its random search in most cases.

Although the algorithm seeks to guarantee that no collisions occur, on the real robots collisions do still arise. The remaining collisions generally appear to be a result of our imperfect

[6] A behavior for attempting to steal the ball from an opponent mistakenly disabled DSS while driving backwards with the ball, and resulted in a penalty when the robot entered its own defense zone.

model of the robots while operating at high speed, and the resulting errors in tracking. This is particularly true for latency, which while nominally $100ms$, varies up to a whole cycle due to occasional radio packet transmission errors, and more frequently up to $10ms$ due to variations in scheduling and processing time on the controlling host computer. A robot travelling at $1.8m/s$ covers $30mm$ in $10ms$, thus resulting in "glancing" collisions such as occur in the traversal video.

5.7 Conclusion

This chapter described a novel real-time control system for multiple robots acting cooperatively while moving near the limits of their dynamics. The Dynamic Safety Search (DSS) algorithm has the following properties:

- It extends the Dynamic Window Approach to multiple cooperating agents
- It can guarantee safety for robots which operate without any error
- It does not guarantee completeness
- It allows agents to change goals every control cycle
- It scales at $O(n^2)$ with the number of agents, and linearly in practice
- It works well on real robots even with modelling error
- It can be used to create an intuitive safe tele-operation system

It is hoped that DSS offers a practical solution for the safe centralized control of multiple coordinating agents. The primary contribution is to demonstrate theoretical and practical safety for a class of robot systems, to serve as a successful model and example for similar problem domains, and as a starting point for future work which relaxes some of the assumptions. It is hoped that the algorithm can be extended in the future, in particular for distributed control of multiple agents. While the current solution is centralized, relying on perfect communication of world state and actions, the system purposely does not rely on additional communication other than the broadcast of world state and actions. Thus communication is bounded and linear in the number of agents, and in particular, no communication is required for deliberation or other explicit coordination.

Chapter 6

Related Work

6.1 Motion Planning

Motion planning is one of the most studied problems in mobile robotics. Latombe [62] gives a thorough overview of early approaches, while Reif [76] establishes the exponential complexity of the general path planning problem. This complexity has inspired many approximation methods, such as local minima free grid-based potentials [56], and common application of the A* algorithm [72] on cost grid representations of the robot's state space. Stentz's D* algorithm [80] builds on A* to create a variant which only recalculates portions of the problem where costs change, achieving significant speedup for domains where the environment changes slowly over time. Much recent work has centered around the idea of randomized sampling for approximation, such as LaValle and Kuffner's RRT algorithm [64,65], and planners based on Kavraki et al.'s Probabilistic Roadmap (PRM) framework [53, 54]. RRT grows random trees in configuration space to solve single-query problems efficiently, and was extended by Bruce [23] to work efficiently in unpredictably changing domains by using continuous replanning with a bias from past plans. PRM separates planning into a learning and query phase. In the learning phase, a random subgraph of the configuration space is build by sampling points and connections between points to find free locations and paths, respectively. In the query phase, this graph can be used with ordinary graph search methods such as A* to solve the path planning problem. It relies on largely static domains to achieve efficiency, since an in an unchanging workspace the learning phase need only be computed once. Boor et al. [9] changed the sampling method of PRM to focus on the boundaries of free space in their Gaussian PRM approach. Amato and Wu [1, 2] create another modified sampling approach called Obstacle PRM, and Hsu [47] created a bridge test to bias sampling to difficult

narrow passages. Isto [49] looked at applying complex local planners to PRM, replacing the commonly used "straight-line" method for local planning to connect two points. It was found to significantly improve graph connectivity for difficult problems resulting in more reliable planning.

Our particular domain contains multiple moving objects, which must be treated as obstacles for safe navigation. Edrman and Lozano-Perez [32] look at the effects of multiple moving objects on the planning problem and offer some early solutions by adding time to the configuration space. Latombe [62] provides background and investigates the effects of moving obstacles on the planning problem. Fiorini and Shiller [36] focus on using relative velocity to simplify the planning problem from each agent's point of view, leading to a more tractable problem for mobile robots. Their work was extended to construct explicit velocity obstacles, first for linear paths and then for arbitrary nonlinear paths [37, 61]. The approach assumes that the complete obstacle paths are known in advance. More recently, Hsu [46] applied randomized planning to domains with moving obstacles, and tested the system on physical robots. This approach assumes constant velocity obstacles, but recovers via replanning when a change in velocity is detected.

6.1.1 Scope and Categorization

One could not hope to cover every path planning method that has been developed, so our attention will be instead restricted to several representative approaches. Some are from recent research while others are classical approaches that have proved popular in applications. They can be compared against the desirable attributes useful for control of a mobile robot. We'd like for an approach to be complete, meaning that if a feasible path exists it is always found. However, path planning has been shown to be PSPACE-hard [76], indicating that any complete planner will likely be exponential in the the number of degrees of freedom. This has led to research into approximate algorithms with relaxed notions of completeness. One relaxation is *resolution completeness*, where a planning algorithm with a resolution is complete for a particular input given a sufficient finite resolution. An example of this is a planner using a grid approximation of the environment; with a sufficiently fine grid a solution can be found if one exists. However, for problems with no solution, such planners are typically not capable of indicating that no path exists with a finite resolution. Another form of relaxation is *probabilistic completeness*, where an algorithm has a nonzero probability of finding a path if one exists. Many randomized sampling based planners have this property. Another important property of planners is what notions of optimality a they can capture. Since all any search can cover a finite number of cases, one way of viewing path planners is in how they reduce the continuous domain problem down to a finite graph for searching.

This aids in analyzing what kind of optimality the algorithm can provide. The reduction to a graph can (and usually does) impose limits on most metrics for optimality (such as length optimality). In addition, given a graph, an algorithm may or may not return an optimal result given its graph representation.

While the previous two properties apply to most planning domains, mobile robots benefit from additional properties since they execute the plan. During execution, new information about the environment is obtained, which can force updates the robots model of reachable free space, as well as changing the ultimate goal the robot is trying to reach. Thus for efficiency, we would like planners to have an efficient method for updating the environment or the query. Of course efficiency is a relative measure and not very meaningful without clarification of its definition within this context. One useful measure is to look at the speed relative to the size in a change in the environment or requested goal. A large change in the problem would be expected to take the same time as a new problem, while a small change in the problem should ideally only require a short update. Thus if a planning algorithm has a large speed component that is proportional to the amount of change in the environment since the last plan was calculated, then we can say it allows for efficient environment updates. We can define efficient goal updates similarly; running time should mostly depend on how far the goal has been moved.

The final property we will consider is if the planner is capable of planning for non-holonomic robots or robots with dynamics constraints. A robot is said to be holonomic if the actuated degrees of freedom match the robot's total degrees of freedom; or alternatively phrased, that the robot can begin accelerating along any degree of freedom at any time. Normally a robotic arm is considered holonomic, as well as some mobile robots with special omnidirectional drive systems. Robots with motion constraints are said to be non-holonomic, such a car-like robot, or a robot with two independently driven wheels on a common axis (called a differential-drive robot). Planning for a non-holonomic robot is in general substantially more difficult. Another similar difficulty is dealing with dynamics constraints such as bounded accelerations and velocities. A planner that can deal with both kinematic and dynamics constraints is referred to as a kinodynamic planner [65].

6.1.2 Graph and Grid Methods

One of the most classical approaches for low dimensional planning are grid methods, where the workspace and configuration space are represented with uniform rectangular grids. Common approaches involve using Dijkstra's algorithm or A^* to plan discrete actions on the grid [72]. Using Dijkstra's algorithm filling C_{free} outward from the goal generates a minima-

free distance on the grid, called a Navigation Function [56]. Such a function has only one global minimum at the goal point, so the robot can follow the gradient downward to reach the goal. Although A^* can deal with arbitrary edge costs, while Dijkstra's is limited to unit length costs, the latter has a significant speed advantage when implemented on regular grids. Specifically, a queue can be used in place of a priority queue, along with several other minor optimizations. Thus for mobile robot applications with frequent replanning, navigation functions have proved popular. However, A^* can be extended to efficiently propagate edge cost changes, as shown by the D^* algorithm [80] and the related Incremental A^* or "D^* Lite" [57]. These algorithms maintain dependency information so that the solution can be updated when edge costs can change. Starting from vertices bordering edges with updated costs, changes are propagated to only the affected nodes, thus saving the cost of a total replan as would be required with A^*. Both algorithms are guaranteed to return a path with the same cost as A^*, so cost optimality is maintained. Thus D^* and its variants support efficient updates to the environment model without affecting optimality. If backwards planning is used, the initial position can be changed efficiently (which is important for mobile robots), while if forward planning is used, the goal position can be moved efficiently. To a limited extent, the other endpoint can be moved in each case by shifting every obstacle and the opposite endpoint (in effect, changing a goal move into a start location move, and vice-versa). This may or may not be more efficient than replanning from scratch for a given environment and representation. In terms of optimality, A^* and D^* are optimal with respect to edge weight, while a navigation function is optimal with respect to the number of edges in a path only (or alternatively viewing all edges as equal weight). The primary problems of grid methods is that they not scale well with the degrees of freedom, and cannot directly plan non-holonomic actions.

Another problem with grid methods is that they are traditionally limited to motion along the directions of the grids or diagonals (four connected or eight connected grids), resulting in paths which are not as straight as they could be, and thus non-optimal in a continuous sense. In the case of an eight-connected grid, paths can be up to 8% longer than optimal [34]. To address this issue, a variant of D^* called Field D^* has been developed [34, 35]. Field D^* modifies traditional grid based planners by allowing edges to traverse at intermediate angles. First, it modifies the search graph by moving the search vertices to the corners of the cost grid cells. This allows edges connecting from an adjacent cell to a neighboring cell to pass through only one cost region (instead of two when the vertices are at the center of cost cells). While the D^* algorithm will generate path lengths for each vertex, Field D^* also estimates the cost of traversing to any point along the edge of a cell, using interpolation between the two neighboring vertices. This heuristic can fail in certain situations, since the cost variation may not be linear between nodes. As a result Field D^* is not guaranteed to find a path with a cost at least as low as D^*. In most problem instances the heuristic works well however, and results in significantly straighter paths which are approximately 4% shorter than D^* or A^*,

while less than doubling the planning time [34]. Thus it works quite well in practice however, and has been applied to many robotic domains to replace classical grid-based planners [35].

6.1.3 Visibility Graph

Due to the high space requirements for grids with more than a couple degrees of freedom, much of the modern work in path planning has tried to use randomization to create "summary graphs" of a workspace using randomized sampling techniques. These summaries are referred to as *roadmaps* [62]. One well known 2D roadmap method is the visibility graph method [69], which noted that a point robot following an optimal trajectory in a field of convex obstacles is always either: (1) following a boundary of an object or (2) following a tangent between two objects (the initial and goal configurations are treated as objects with radius zero). Thus a finite graph could be created that still contained the optimal (minimum length) path in from continuous space. Unfortunately, this method does not generalize to higher dimensions, nor does it scale well with the number of obstacles. It also cannot optimize other metrics, such as those that encourage safety margins around obstacles. In fact, without postprocessing, the plan will skim every obstacle along the path. Thus this simple, optimal method works very well, but only for a very limited environment. It cannot be extended for additional capabilities and thus is the algorithmic equivalent of a local maximum.

6.1.4 Randomized Path Planner (RPP)

Due to the limitations of the classical approaches such as grids and exact roadmap methods, alternative approaches were explored. Randomization has proved a powerful tool in this pursuit. One of the first randomized planners was RPP (Randomized Path Planner) [62]. It constructed a navigation function in the workspace, which may however contain local minima in the configuration space. The planner proceeds by following the navigation function until it reaches a local minimum, and then executes random motions in an attempt to escape the attraction well of the minimum. It records each minimum so it can determine if a random motion escaped the attraction well. A list of minima along the current path is maintained so that the search can backtrack to a random configuration if the last minima cannot be escaped after several iterations. RPP is capable of solving difficult problems, but by relying on both grids and randomization it is only probabilistically resolution complete. It does not guarantee any form of optimality if a local minima is reached during the search, and does not offer any efficient environment or query updates. It scales well with the degrees of freedom

of the robot, but not with workspace dimensions. In its plain form it does not deal with holonomic or dynamic constraints.

6.1.5 Rapidly Exploring Random Trees (RRT)

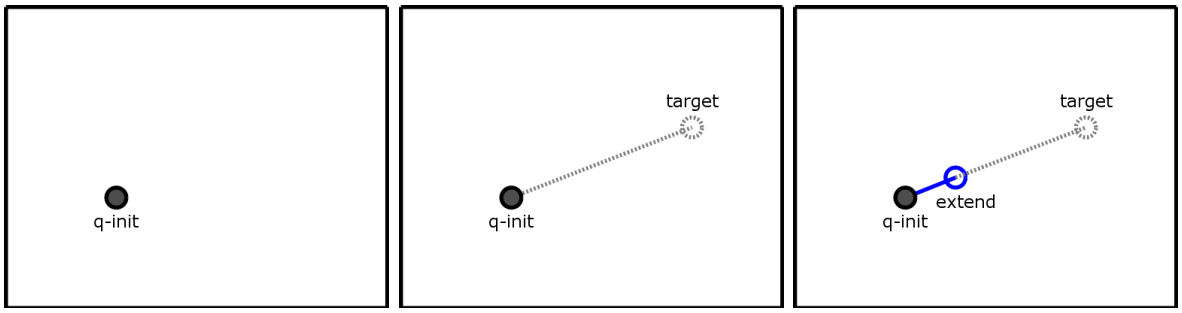


Figure 6.1: Algorithm steps in RRT

One of the relatively recently developed randomized planning approaches are those based on Rapidly-exploring random trees (RRTs) [64]. RRTs employ randomization to explore large state spaces efficiently, and can form the basis for a probabilistically complete though non-optimal kinodynamic path planner [65]. Their strengths are that they can efficiently find plans in high dimensional spaces because they avoid the state explosion that discretization faces. Furthermore, due to their incremental nature, they can maintain complicated kinematic constraints if necessary. A basic planning algorithm using RRTs is as follows: Start with a trivial tree consisting only of the initial configuration. Then iterate: With probability p, find the nearest point in the current tree and extend it toward the goal g. Extending means adding a new point to the tree that extends from a point in the tree toward g while maintaining whatever kinematic constraints exist. In the other branch, with probability $1 - p$, pick a point x uniformly from the configuration space, find the nearest point in the current tree, and extend it toward x. Thus the tree is built up with a combination of random exploration and biased motion towards the goal configuration. Search efficiency can be improved by using bidirectional search growing a tree both from the initial and goal configurations [51]. RRT planners do not support efficient environment or query updates. However they are fast enough that realtime rates can be achieved for relatively simple problems, and with additional extensions quite reasonable performance can be achieved.

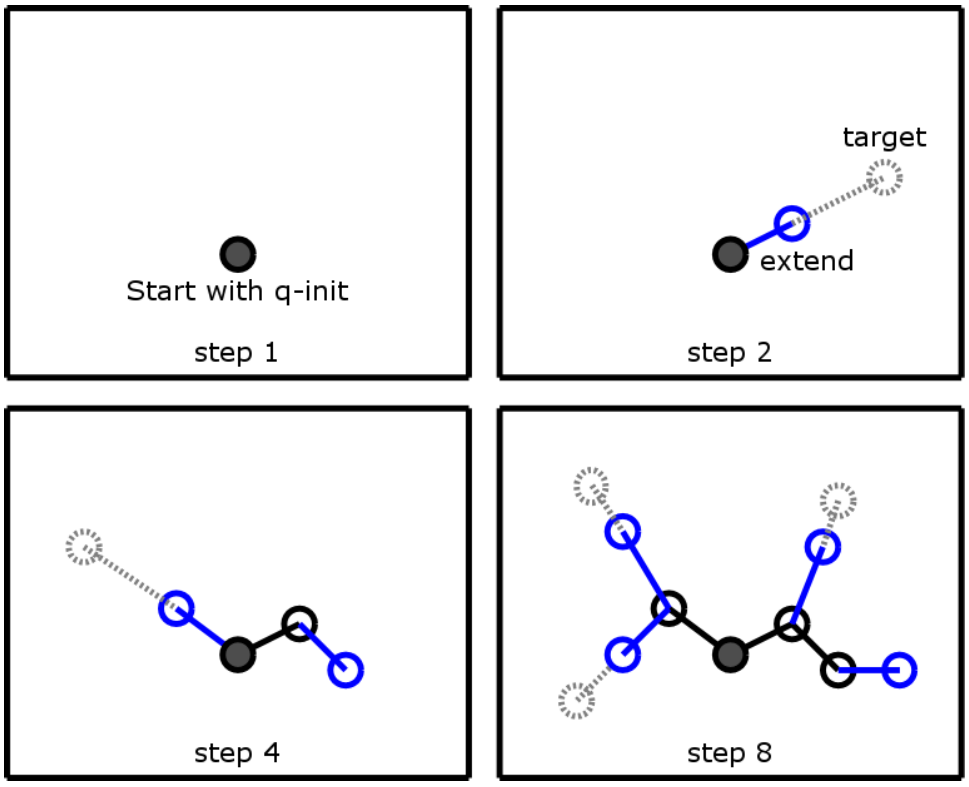


Figure 6.2: Example Growth of RRT

6.1.6 RRT Variants

Two variants of RRT exist which attempt to tackle the replanning problem. The first, called Reconfigurable Random Forest (RRF) [66], extends RRT-Connect by allowing a forest of random trees instead of just the two rooted at the initial and goal configurations. For an initial plan, RRT-Connect is executed as normal, but for replans, search starts with the existing trees. First, all the vertices and edges which may be affected by a change in the environment are checked for collisions, and those that are no longer in C_{free} are removed. Removing edges can "orphan" parts of the search tree, and those subsets are considered as new trees and added to a list of trees to use during search. RRT-Connect proceeds as normal, extending the initial and goal trees, but the Connect operation is replaces with a Merge-Tree operation. The Merge-Tree operation attempts to connect a newly added node to all other trees on the search list, and if the connection succeeds those two trees are re-parented into a single tree. Search continues until some maximum number of iterations, or the initial and goal trees have been connected. Because RRF continues adding nodes with each new query, eventually the search tree may become too large. Thus RRF introduces a pruning

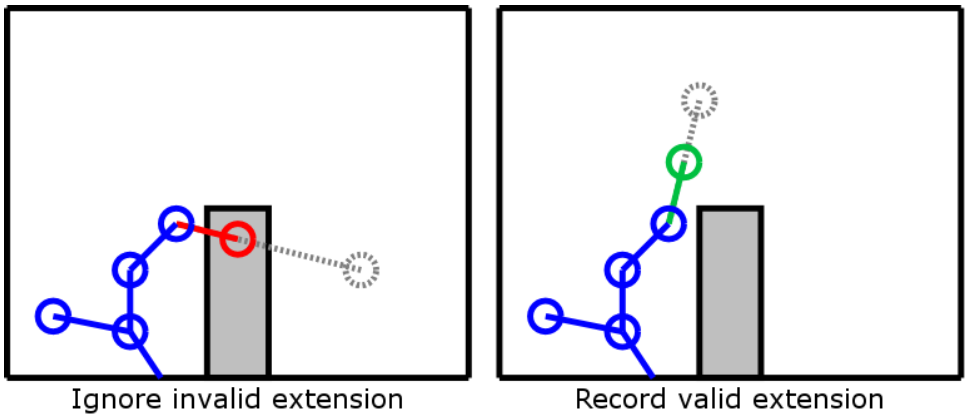


Figure 6.3: RRT extensions with obstacles

operation to decrease the number of nodes to a more reasonable number while maintaining good coverage. While RRF offers good coverage of domains, it suffers from two drawbacks. The first problem is that its runtime tends to by cyclic, increasing as nodes are added, rising dramatically when pruning occurs, and the dropping to a lower level. This could be addressed with continuous online pruning, but at the expense of even more algorithm complexity. The second problem is that despite good coverage, the tree representation means there is always a unique path between any two vertices in the tree. In configuration spaces with loops, this can result in highly non-optimal paths being returned. RRT, by growing a new tree outward from the initial and goal points each query, tends to avoid this problem.

The second extension of RRT for replanning is Dynamic Rapidly-Exploring Random Trees (DRRT) [33]. This work can be seen as a more conservative version of RRF which aims to be a continuous planner analogous to D^* on grids. For an initial query, DRRT grows a tree backward from the goal configuration using the standard RRT algorithm until the initial configuration is reached. To handle an environment update, all possibly affected vertices and edges are checked for collisions and removed if they are no longer in C_{free}. Unlike RRF however, DRRT removes the subtree of any such node, thus preventing orphaned trees from being created. Replanning queries proceed from the existing tree, and thus can be quite efficient when updates occur near the initial configuration. DRRT works best when the sample distribution for search is biased toward areas where obstacles have been updated. The drawbacks of DRRT are that it does not support moving the goal configuration, that obstacle changes near the goal can invalidate large parts of the tree, and that for efficiency it needs to be notified explicitly of which areas of the environment have changed and which have remained static.

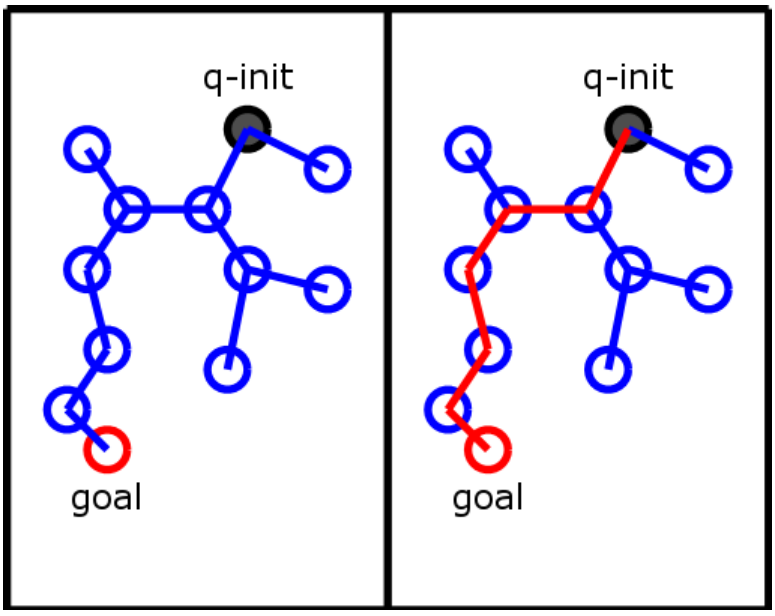


Figure 6.4: RRT as a motion planner

6.1.7 Probabalistic Roadmaps (PRM)

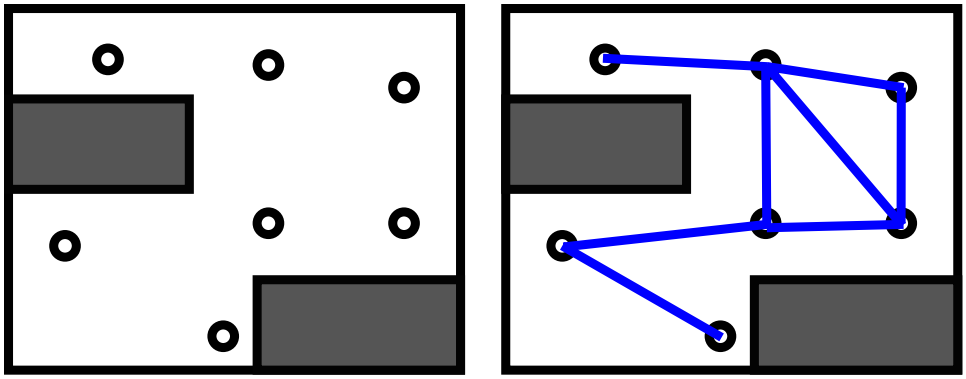


Figure 6.5: Uniformly Sampled PRM Example

Finally, a widely researched planner for static workspaces with higher dimensionality are those based on the concept of Probabilistic Roadmaps, or PRMs [53, 54]. PRMs have two distinct planning phases; The learning phase where a roadmap is built from the configuration space, and the query phase where it is used to solve a particular problem. The learning phase needs to be run whenever the environment changes, to create the summary that the planning stages use. First, free configurations are found through some sampling distribution (such as

uniform). Then, connections are made among these configurations using a "local planner'. A simple straight line planner is often used, which checks if one configuration could be linearly interpolated to another configuration without hitting an obstacle. In figure 6.5 a workspace is shown first after sampling for free configurations using uniform sampling, and then after connections have been made to create the roadmap. The connections in this example are attempted with a simple straight line planner. At the end of the learning phase, we have a graph where the free configurations are the vertices, and the successful local plans are the edges. If the local plan weights are recorded as weights in the graph, we can apply A^* or any other graph searching method to get from one configuration in the graph to any other configuration in the same connected graph component. In the query phase, the initial and goal position are added to the graph, by trying to connect them to nearby vertices using the local planner, then the graph search algorithm is used to find the actual path. The planner fails if it cannot connect the initial or goal configurations to the graph, or if the graph sections they connect to are not connected by the roadmap. Taken together, these two phases comprise the basic PRM planning algorithm. It is probabilistically complete, and when using A^* can offer optimal plans (with respect to the roadmap, which is an finite approximation of the paths in free space). It does not allow efficient environment updates, because the learning stage and query stage must both be recomputed (i.e. it starts from scratch). Query updates however tend to be very efficient, because only the graph search is required and PRM graphs can be relatively small when compared to grids. In addition, if the graph is large enough that speed becomes a problem, D^* could be used since it applies to general weighted graphs [80]. PRM has good scalability with degrees of freedom thanks to its random sampling techniques. The plain version of PRM does not handle non-holonomic constraints. It can be extended to solve some non-holonomic problems using specialized local planners that satisfy the constraints [81].

6.1.8 PRM Variants

The PRM approach is a flexible one, allowing modification of various components while maintaining its basic capabilities. This has led to a large number of variants which try to address limitations or improve efficiency. The three major variables are the sampling strategy, the roadmap construction method, and the local planner used. For sampling, it has been found that a uniform distribution generally does not perform as well as modified distributions that are a more complicated function of free space. Obstacle PRM [1, 2] is a variant which generates samples on the boundary of free space. It first generates samples on the surface of one obstacle, and uses rejection sampling to remove those samples that are inside other obstacles. The bridge test [47] sampling method is a rejection sampling method that accepts free configurations that lie on the midpoint of two non-free configurations. The

length of the bridge is varied with a Gaussian distribution. Finding these bridges is very time consuming due to the higher fraction of candidate samples being rejected. However when combined with a small uniform distribution to fill free space, relatively few nodes are needed to describe fairly complicated environments with narrow passages. Thus it is very efficient for connecting the roadmap configurations and in the query phase.

The next kind of modification of PRM is to change the roadmap construction approach, such as which sampled configurations we try to connect, and when we check them. Connected components analysis is used in some PRM variants, for example, to speed roadmap construction by only trying to connect configurations from different components. While a significant speedup, this also means no redundant paths are found, thus the resulting roadmap is a tree. These are most useful for problems where path length optimality is not an issue. Another optimization trades slower queries to gain a faster learning phase. In Lazy PRM [7,8], collision checks are not done while constructing the roadmap; instead of testing if an edge can be added it is added as a "potentially free" local path. During the query, the graph is searched to find the shortest potentially free path. Then the path is checked and any edges that are not free are removed. If no edges were removed, the path is a valid solution, while if edges are removed the query is repeated with the pruned graph. Eventually a free path will be found if it exists in the graph, and only a minimal number of edges will have to be checked for collisions.

The third kind of modification of PRM is to replace the local planner, which tries to connect two configurations when building the roadmap or connecting the query configurations to the roadmap. For non-holonomic problems for which a local planner exists (which can fail in some circumstances, but must work when no obstacles are present), PRM can use that local planner to become a non-holonomic global planner. Such a local planner has been derived for car-like robots in [81]. Isto showed in [49] that increasing the power of the local planner improves the overall performance of PRM on most benchmarks. Specifically, allowing the local planner to slide along obstacles after contact, or using discretized heuristic planners improved roadmap compactness and connectedness. Finally, some planners combine powerful local planners with advanced construction methods. Probabilistic roadmaps of trees (PRT) [6] is a PRM planner that uses RRT as the local planner to connect a relatively small number of configurations. It has been shown to work for high dimensional problems of at least 18 degrees of freedom. A related approach is the previously mentioned Reconfigurable RRT Forest [67], where a tree is maintained by using RRT-Connect to connect a forest and merging the trees together by re-parenting. It can be thought of as a PRM variant instead of an RRT derivative in many respects. Unfortunately the tree representation of the roadmap means that only one path can ever be found in the graph search, and it will generally be far from optimal.

6.1.9 Randomized Forward Planners

In contrast to the RRT and PRM approach of "pulling" a search graph by choosing random samples and then trying to connect a path to those points, some planners "push" samples by first choosing some vertex to expand, and then extending it using a random action. These algorithms can be thought of as modern descendents of the RPP algorithm. One of the first descendents was Hsu's Expansive Configuration Space Planner (ECSP) [48]. ECSP initializes its search with the initial and goal configuration as roots of two search trees. It then executes and *Expand* operation on each tree, where a vertex v is chosen at random with probability $1/w(v)$. The weight $w(v)$ is derived from a local density estimate of the search tree. The chosen configuration v has a number of random points v' sampled around it, which are kept with probability $1/w(v')$ if a free direct path between v and v' exists. After each tree has been expanded, a *Connect* operation is employed which tries to link all pairs of vertices in each tree if they are below some distance threshold. If the link is a free path, the trees have been connected and search can terminate. Otherwise, search continues until some time limit has been reached. ECSP has not found broad application due to efficiency that is generally less than PRM or RRT, but it inspired much additional research.

The Guided Expansive Space Tree planner [73] extended ECSP through the addition of a goal distance heuristic to bias search, much in the way A^* uses a heuristic to limit search on finite graphs. It also made use of the forward planning possible with ECSP to sample actions, thus allowing highly non-holonomic problems to be solved. The PDST-Explore [59] planner takes this further, planning exclusively in action space using forward simulation. Local density estimates are calculated in a coarse way using KD-trees, and nodes are chosen for expansion deterministically, based on the volume of the KD-tree cell containing the node, and the number of times the node has been chosen. However, instead of expanding that specific node, PDST-Explore samples a point randomly along the continuous path from the initial configuration to that node's configuration, which it then extends using a random action. As a result, PDST-Explore can converge to uniform coverage of control space, and thus is probabilistically complete even for non-holonomic problems. It has been applied successfully to non-holonomic problems with drift and under-actuation [60]. However, it does not currently have a variant tailored for replanning.

6.2 Safety Methods

6.2.1 Dynamic Window

Though not a path planner in the same sense as the other algorithms, the dynamic window approach [38] is a search method for controlling mobile robots in light of both kinematic and dynamics constraints. It is a local method, in that only the next velocity command is determined, however it can incorporate non-holonomic constraints, limited accelerations, and the presence of obstacles into that determination, guaranteeing safe motion. The search space is the velocities of the robot's actuated degrees of freedom. The two developed cases are for synchro-drive robots with a linear velocity and an angular velocity, and for holonomic robots with two linear velocities [12, 38]. A grid is created for this velocity space, reflecting an evaluation of velocities falling in each cell. First, the obstacles of the environment are considered, by assuming the robot travels at a cell's velocity for one control cycle and then attempts to brake at maximum deceleration while following that same trajectory. If the robot cannot come to a stop before hitting an obstacle along that trajectory, the cell is given an evaluation of zero. Next, due to limited accelerations, velocities are limited to a small window that can be reached within the acceleration limits over the next control cycle (for a holonomic robot this is a rectangle around the current velocities). Finally, the remaining velocities are scored using a heuristic distance to the goal. Like all local methods, the dynamic window approach is incomplete, but it demonstrated practical applicability on real robots moving at relatively high speeds. In addition, when combined with a navigation function for mid-level planning, a non-optimal but resolution complete planner was developed and tested at speeds up to 1m/s in cluttered office environments with dynamically placed obstacles [12].

6.3 Algorithm Summary

Taken together, these algorithms and approaches solve many variants of the path planning problem. Many have limitations when viewed A summary of all related algorithms mentioned in this section is shown in Table 6.1.

Table 6.1: Comparison of related planning algorithms

Approach	Complete	Optimal	Efficient Environ. Updates	Efficient Query Updates	Good dof Scalability	Non-Holonomic
Grid A^*	res	grid	no	no	no	no
Grid D^*	res	grid	yes	yes	no	no
Field D^*	res	no	local	part	no	no
Nav Func	res	grid	no	no	no	no
RPP	prob+res	no	no	no	no	yes
RRT	prob	no	no	no	yes	yes
RRF	prob	no	yes	yes	yes	no
DRRT	prob	no	local	part	yes	no
PRM	prob,res	graph	no	yes	yes	some
Lazy PRM	prob	graph	yes	yes	yes	no
PRT	prob	no	no	yes	yes	no
ECSP	unkn	no	no	no	maybe	no
Guided ECSP	unkn	no	no	no	yes	yes
PDST-Explore	prob	no	no	no	yes	yes
Dynamic Win	no	res	no	no	no	yes

Key	Term	Meaning
res	resolution complete	path is found if the resolution is sufficiently small
prob	prob. complete	path is found with nonzero probability
unkn	unknown completeness	depends on unproven conjecture
grid	grid optimal	shortest path on grid is always found
graph	graph optimal	shortest path in roadmap is always found
local	local environ. updates	efficient updates supported near the initial configuration
part	partial query updates	initial configuration can change, but not goal
some	some variants	specialized variants support non-holonomic planning

Appendix A

Machine Vision for Navigation

The need for sensing in any truly autonomous robot is ubiquitous. Among the various sensors that can be applied, one of the most powerful and inexpensive is that of machine vision. Color-based region segmentation, where objects in the world are identified by specific color (but not necessarily uniquely), has proved popular in robotics and automation, because color coding is a relatively unobtrusive modification of the environment. With the coding, balls, goals, obstacles, other robots, as well as other object can be detected. As the primary sensory process for many mobile robots, vision goes hand-in-hand with navigation. A system may use a camera on a robot to detect relative obstacles for navigation, or in the case of many small robots, a camera fixed in the world space to detect both robots and nearby obstacles.

Although popular as a sensor due to low hardware cost, vision has sometimes proved difficult due to high processing requirements and a large input stream to sift through in order to generate perceptual information for higher levels in the system. Thus the problem is that of mapping an input video stream to a perceptually more salient representation for other parts of the agent. The representation popular in hardware and domain-specific approaches to this problem is to segment the video stream into colored regions (representing all or part of a colored object). This is the representation we also choose, as it has proved successful in many applications [50].

A.1 CMVision: The Color Machine Vision Library

A.1.1 Color Image Segmentation

By far the most popular approach in real time machine vision processing has been color segmentation. It is currently popular due to the relative ease of defining special colors as markers or object labels for a domain, and has proved simpler than other methods such as the use of geometric patterns or barcodes. Among the many approaches taken in color segmentation, the most popular employ single-pixel classification into discrete classes. Among these, linear and constant thresholding are the most popular. Other alternatives include nearest neighbor classification and probability histograms.

Linear color thresholding works by partitioning the color space with linear boundaries (e.g. planes in 3-dimensional spaces). A particular pixel is then classified according to which partition it lies in. This method is convenient for learning representations such as artificial neural networks (ANNs) or multivariate decision trees (MDTs) [13].

A second approach is to use nearest neighbor classification. Typically several hundred pre-classified exemplars are employed, each having a unique location in the color space and an associated classification. To classify a new pixel, a list of the K nearest exemplars are found, then the pixel is classified according to the largest proportion of classifications of the neighbors [15]. Both linear thresholding and nearest neighbor classification provide good results in terms of classification accuracy, but do not provide real-time performance using off-the-shelf hardware.

Another approach is to use a set of constant thresholds defining a color class as a rectangular block in the color space [50]. This approach offers good performance, but is unable to take advantage of potential dependencies between the color space dimensions.

A final related approach is to store a discretized version of the entire joint probability distribution. So, for example, to check whether a particular pixel is a member of the color class, its individual color components are used as indices to a multi-dimensional array. When the location is looked up in the array the returned value indicates probability of membership. This technique enables a modeling of arbitrary distribution volumes and membership can be checked with reasonable efficiency. The approach also enables the user to represent unusual membership volumes (e.g. cones or ellipsoids) and thus capture dependencies between the dimensions of the color space. The primary drawback to this approach is its associated high memory cost.

A.1.2 Color Spaces

The color space refers to the multidimensional space the describes the color at each discrete point, or pixel, in an image. The intensity of a black and white image is a segment of single dimensional space, where the value varies from its lowest black value to its highest at white. Color spaces generally occupy three spaces, although can be projected into more or fewer to yield other color representations. The common RGB color space consists of a triplet of red, green, and blue intensity values. Thus each color in the representation lies in a cube with black at the corner (0,0,0), and pure white at the value (1.0,1.0,1.0). Here we will describe the different color spaces we considered for our library, including RGB, a projection or RGB we call fractional YRGB, and the YUV color space used by the NTSC and PAL video standards, among other places.

In our choice of appropriate color spaces, we needed to balance what the hardware provides with what would be amenable to our threshold representation, and what seems to provide the best performance in practice. At first we considered RGB, which is a common format for image display and manipulation, and is provided directly by most video capture hardware. It's main problem lies in the intensity value of light and shadows being spread across all three parameters. This makes it difficult to separate intensity variance from color variance with a rectangular, axis aligned threshold. More complex threshold shapes alleviate this problem, but that was not possible in our implementation. An equally powerful technique is to find another color space or projection of one that is more appropriate to describe using rectangular thresholds.

This limitation lead us to explore a software transformed RGB color space we called fractional RGB. It involves separating the RGB color into four channels, intensity, red, green, blue. The color channels in this case are normalized by the intensity, and thus are fractions calculated using the following definition:

$$Y' = \frac{(R+G+B)}{3} \tag{A.1}$$
$$R' = \frac{R}{Y'} \tag{A.2}$$
$$G' = \frac{G}{Y'} \tag{A.3}$$
$$B' = \frac{B}{Y'} \tag{A.4}$$

The main drawback of this approach is of course the need to perform several integer divides or floating point multiplications per pixel. It did however prove to be a robust space for describing the colors with axis-aligned threshold cubes. It proves useful where RGB is the only available color space, and the extra processing power is available.

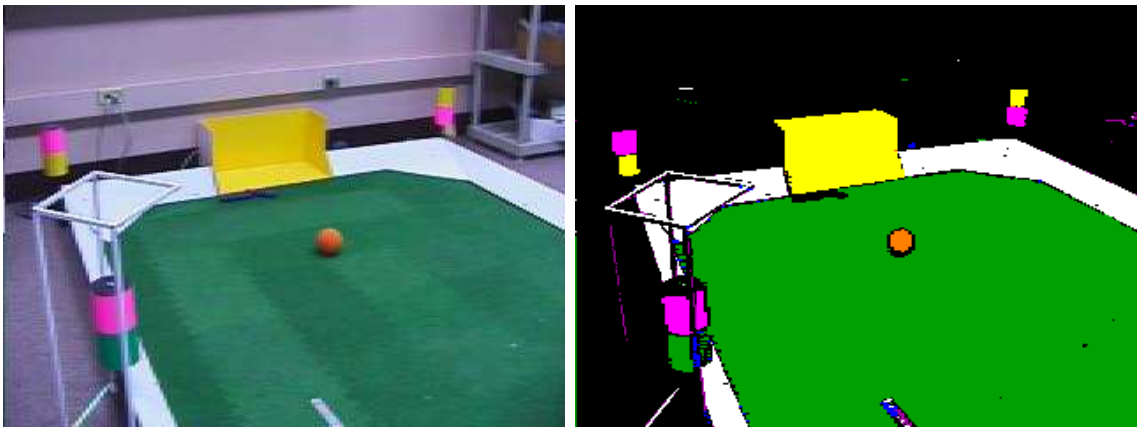


Figure A.1: A video image and its YUV color classified result

The final color space we tried was the YUV format, which consists of an intensity (Y) value, and two chrominance (color) values (U,V). It is used in video standards due to its closer match with human perception of color, and since it is the raw form of video, it is provided directly by most analog video capture devices. Since intensity is separated into its own separate parameter, the main cause of correlations between the color component values has been removed, and thus is a better representation for the rectangular thresholds. This is because the implementation requires axis aligned sides for the thresholds, which cannot model interactions among the component color values. Thus YUV proved to be robust in general, fast since it was provided directly by hardware, and a good match for required assumptions of component independence in our implementation. An example YUV histogram, with a threshold shown outlining a target yellow color is given in figure A.2

One color space we have not tried with our library is HSI, or hue, saturation, intensity. In its specification, hue is the angle on the color wheel, or dominant spectral frequency, saturation is the amount of color vs. neutral gray, and I is the intensity. Although easy for humans to reason in (hence its use in color pickers in painting programs), it offers little or no advantage over YUV, and introduces numerical complications and instabilities. Complications primarily arise from the angle wrapping around from 360 to 0, requiring thresholding operations work on a modular number values. More seriously, at low saturation values (black, gray, or white), the hue value becomes numerically unstable, making thresholds to describe these common colors unwieldy, and other calculations difficult [74]. Finally, HSI can be approximated by computing a polar coordinate version of the UV values in YUV. Since YUV is available directly from the hardware, it is simplest just to threshold in the pre-existing YUV space, thus avoiding the numerical problems HSI poses.

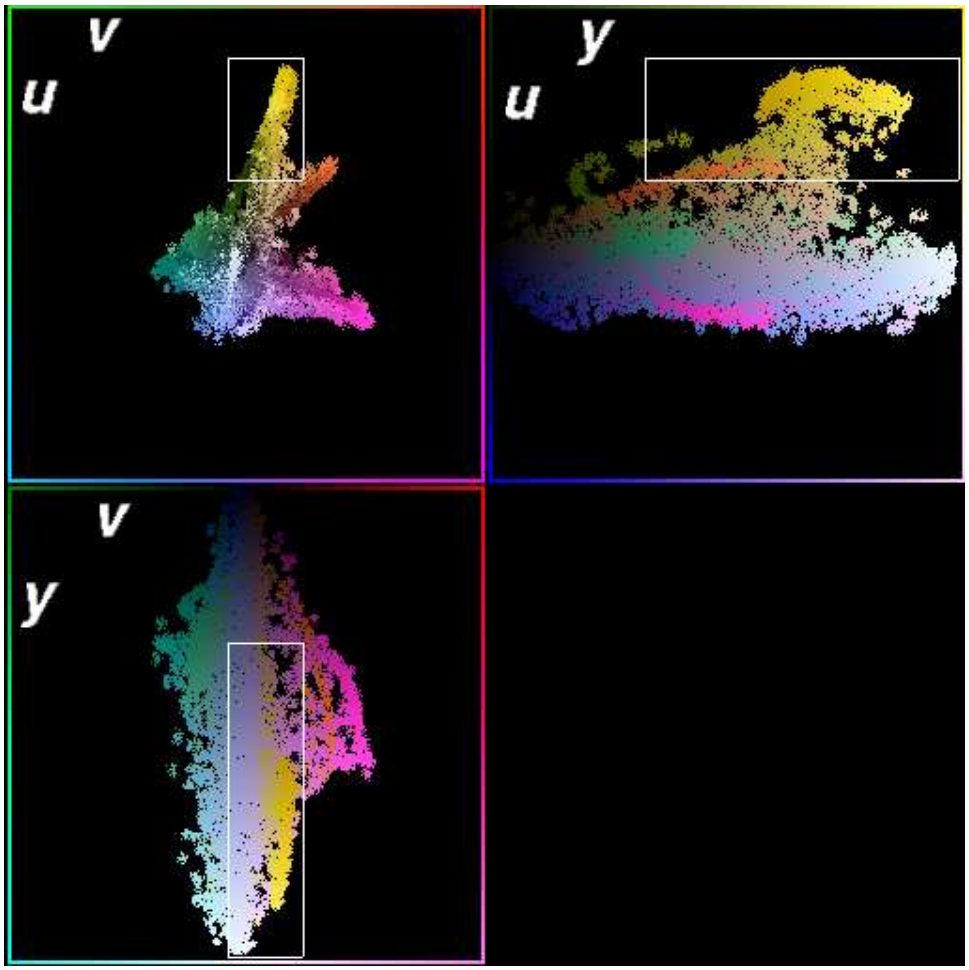


Figure A.2: A YUV histogram with a threshold defined

A.1.3 Thresholding

The thresholding method described here can be used with general multidimensional color spaces that have discrete component color levels, but the following discussion will describe only the YUV color space, since generalization of this example will be clear. In our approach, each color class is initially specified as a set of six threshold values: two for each dimension in the color space, after the transformation if one is being used. The mechanism used for thresholding is an important efficiency consideration because the thresholding operation must be repeated for each color at each pixel in the image. One way to check if a pixel is a member of a particular color class is to use a set of comparisons similar to

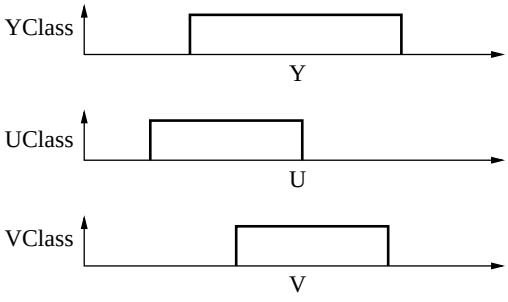
```
if ((Y >= Ylowerthresh)
   AND (Y <= Yupperthresh)
   AND (U >= Ulowerthresh)
   AND (U <= Uupperthresh)
   AND (V >= Vlowerthresh)
   AND (V <= Vupperthresh))
   pixel_color = color_class;
```

to determine if a pixel with values `Y, U, V` should be grouped in the color class. Unfortunately this approach is rather inefficient because, once compiled, it could require as many as 6 conditional branches to determine membership in one color class for each pixel. This can be especially inefficient on pipelined processors with speculative instruction execution.

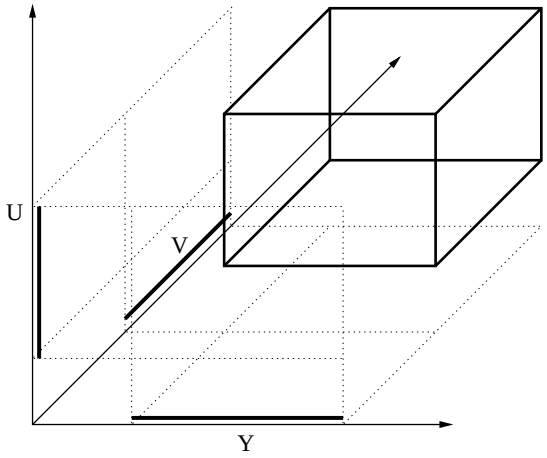
Instead, our implementation uses a boolean valued decomposition of the multidimensional threshold. Such a region can be represented as the product of three functions, one along each of the axes in the space (Figure A.3). The decomposed representation is stored in arrays, with one array element for each value of a color component. Thus class membership can be computed as the bitwise `AND` of the elements of each array indicated by the color component values:

```
pixel_in_class = YClass[Y]
              AND UClass[U]
              AND VClass[V];
```

The resulting boolean value of `pixel_in_class` indicates whether the pixel belongs to the class or not. This approach allows the system to scale linearly with the number of pixels and color space dimensions, and can be implemented as a few array lookups per pixel. The operation is much faster than the naive approach because the the bitwise `AND` is a significantly lower cost operation than an integer compare on most modern processors.



Binary Signal Decomposition of Threshold



Visualization as Threshold in Full Color Space

Figure A.3: A 3D region of the color space represented as a combination of 1D binary functions.

To illustrate the approach, consider the following example. Suppose we discretized the YUV color space to 10 levels in each each dimension. So "orange," for example might be represented by assigning the following values to the elements of each array:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {0,0,0,0,0,0,0,1,1,1};
VClass[] = {0,0,0,0,0,0,0,1,1,1};
```

Thus, to check if a pixel with color values `(1,8,9)` is a member of the color class "orange" all we need to do is evaluate the expression `YClass[1] AND UClass[8] AND VClass[9]`, which in this case would resolve to 1, or `true` indicating that color is in the class "orange."

One of the most significant advantages of our approach is that it can determine a pixel's membership in multiple color classes *simultaneously*. By exploiting parallelism in the bit-wise `AND` operation for integers we can determine membership in several classes at once. As an example, suppose the region of the color space occupied by "blue" pixels were represented as follows:

```
YClass[] = {0,1,1,1,1,1,1,1,1,1};
UClass[] = {1,1,1,0,0,0,0,0,0,0};
VClass[] = {0,0,0,1,1,1,0,0,0,0};
```

Rather than build a separate set of arrays for each color, we can combine the arrays using each bit position an array element to represent the corresponding values for each color. So, for example if each element in an array were a two-bit integer, we could combine the "orange" and "blue" representations as follows:

```
YClass[] = {00,11,11,11,11,11,11,11,11,11};
UClass[] = {01,01,01,00,00,00,00,10,10,10};
VClass[] = {00,00,00,01,01,01,00,10,10,10};
```

Where the first (high-order) bit in each element is used to represent "orange" and the second bit is used to represent "blue." Thus we can check whether `(1,8,9)` is in one of the two classes by evaluating the single expression `YClass[1] AND UClass[8] AND VClass[9]`. The result is `10`, indicating the color is in the "orange" class but not "blue."

In our implementation, each array element is a 32-bit integer. It is therefore possible to evaluate membership in 32 distinct color classes at once with two `AND` operations. In contrast, the naive comparison approach could require 32×6, or up to 192 comparisons for the same operation. Additionally, due to the small size of the color class representation, the algorithm can take advantage of memory caching effects.

A.1.4 Connected Regions

After the various color samples have been classified, connected regions are formed by examining the classified samples. This is typically an expensive operation that can severely impact real-time performance. Our connected components merging procedure is implemented in two stages for efficiency reasons.

The first stage is to compute a run length encoded (RLE) version for the classified image. In many robotic vision applications significant changes in adjacent image pixels are relatively infrequent. By grouping similar adjacent pixels as a single "run" we have an opportunity for efficiency because subsequent users of the data can operate on entire runs rather than individual pixels. There is also the practical benefit that region merging need now only look for vertical connectivity, because the horizontal components are merged in the transformation to the RLE image.

The merging method employs a tree-based *union find* with path compression. This offers performance that is not only good in practice but also provides a hard algorithmic bound that is for all practical purposes linear [82]. The merging is performed in place on the classified

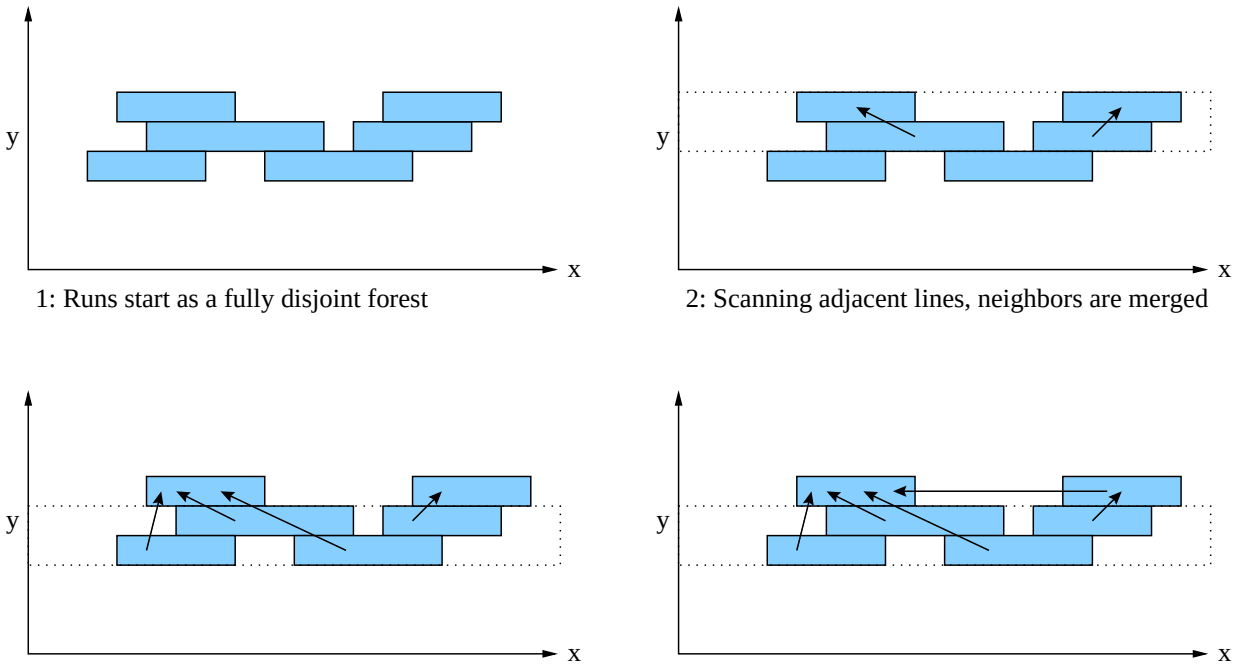


Figure A.4: An example of how runs are grouped into regions

RLE image. This is because each run contains a field with all the necessary information; an identifier indicating a run's parent element (the upper leftmost member of the region). Initially, each run labels itself as its parent, resulting in a completely disjoint forest. The merging procedure scans adjacent rows and merges runs which are of the same color class and overlap under four-connectedness. This results in a disjoint forest where the each run's parent pointer points upward toward the region's global parent. Thus a second pass is needed to compress all of the paths so that each run is labeled with its the actual parent. Now each set of runs pointing to a single parent uniquely identifies a connected region. The process is illustrated in Figure A.4).

A.1.5 Extracting Region Information

In the next step we extract region information from the merged RLE map. The bounding box, centroid, and size of the region are calculated incrementally in a single pass over the forest data structure. Because the algorithm is passing over the image a run at a time, and not processing a region at a time, the region labels are renumbered so that each region label is the index of a region structure in the region table. This facilitates a significantly faster lookup. A number of other statistics could also be gathered from the data structure, including the convex hull and edge points which could be useful for geometric model fitting.

After the statistics have been calculated, the regions are separated based on color into separate threaded linked lists in the region table. Finally, they are sorted by size so that high level processing algorithms can deal with the larger (and presumably more important) blobs and ignore relatively smaller ones which are most often the result of noise.

A.1.6 Density-Based Region Merging

In the final layer before data is passed back up to the client application, a top-down merging heuristic is applied that helps eliminate some of the errors generated in the bottom up region generation. The problem addressed here is best introduced with an example. If a detected region were to have a single line of misidentified pixels transecting it, the lower levels of the vision system would identify it as two separate regions rather than a single one. Thus a minimal change in the initial input can yield vastly differing results.

One solution in this case is to employ a sort of grouping heuristic, where similar objects near each other are considered a single object rather than distinct ones. Since the region statistics include both the area and the bounding box, a density measure can be obtained.

The merging heuristic is operationalized as merging pairs of regions, which if merged would have a density is above a threshold set individually for each color. Thus the amount of "grouping force" can be varied depending on what is appropriate for objects of a particular color. In the example above, the area separating the two regions is small, so the density would still be high when the regions are merged, thus it is likely that they would be above the threshold and would be grouped together as a individual region.

A.1.7 Performance

CMVision was tested at several resolutions using pre-captured data from an overhead camera. For the test setup, a 60-image sequence was taken at 3 resolutions, up to a maximum of 640x240 (interlaced NTSC video), and stored in a memory buffer so that the library performance could be isolated from image capture overhead. The YUV colorspace colorspace (native NTSC YCrCb422 format) was used for image capture and thresholding. The thresholding, connected components, region extraction, and region sorting methods were run (density-based region merging was disabled) were run 1000 times to generate averaged timings. The tests were run on a 2.4 GHz Athlon computer, and compiled using g++ with optimization enabled. The results are shown in Table A.1.7. As can be seen, CMVision can run quickly on a modern computer, leaving plenty of processing time for other tasks. Multiple 60 Hz video sequences can be processed on a single machine, and are limited by the bandwidth between the capture cards and main memory[1] instead of the vision processing itself. Alternatively, it can be applied on lower-speed embedded processors such as those likely to be found onboard small robots.

Frame Size (w x h)	Processing Time (ms)	Throughput (frames/sec)
640x240	0.968333	1032.70
320x240	0.498148	2007.44
160x120	0.126860	7882.71

Table A.1: Performance of the CMVision low-level color vision library at various resolutions

[1]Bandwidth for a single steam of digitized NTSC video can reach 18 MB/s without counting overhead. The maximum theoretical bandwidth of a PCI bus is 127 MB/s, although typically much less can be achieved in practice with multiple PCI devices.

A.1.8 Summary of the CMVision Library

The CMVision library is a system to accelerate low level segmentation and tracking using color machine vision. In creating it, we evaluated the properties of alternative approaches, choosing color thresholding as a segmentation method, and YUV or fractional YRGB as robust color spaces for thresholding. The created system can perform bounded computation, full frame processing at camera frame rates. The system can track up to 32 colors at resolutions up to 640x480 and rates at 30 or 60Hz without specialized hardware. Thus the primary contribution of this system is that it is a software-only approach implemented on general purpose, inexpensive, hardware. This provides a significant advantage over more expensive hardware-only solutions, or other, slower software approaches. The approach is intended primarily to accelerate low level vision for use in real-time applications where hardware acceleration is either too expensive or unavailable.

Building on this lower level, CMVision has been applied to several applications, including Carnegie Mellon's RoboCup small-size league team entries (since 2000), as well as the Sony legged (Aibo) league entries (1999-2005). As applied in the CMDragons system, CMVision has been used to track robots one two cameras at 60Hz on a single computer, while leaving sufficient processing time available for robot behaviors and motion planning. The next section describes the pattern detection system built for the CMDragons robot system.

A.2 Pattern Detection

Fast pattern detection and identification is a fundamental problem for many applications of real-time vision systems. The desirable characteristics for a solution are that it requires little computation, localizes a pattern robustly and with high accuracy, and can identify a large number of unique pattern identifiers so that many of these markers can be tracked within a field a view. We will present a system that can accurately track a broad class of patterns both accurately and quickly, when used with a suitable low level vision system that can return calibrated coordinates of regions in a image. Both pattern design and the detection algorithm are considered together to find a solution meeting the above criteria. Along the way, assumptions are verified to make informed choices without relying on guesswork, and allowing similar systems to be designed on a solid experimental and statistical basis.

Object identification and tracking is one of the most important current applications of machine vision. Much work has focused on object detection and tracking for complex or variable objects, such as faces, cars, and doors. While much progress has been made, many of the

algorithms require substantial amounts of processing and are less accurate than can be achieved with patterns specifically designed for detection. Thus many current applications of vision-based object detection and tracking use customized patterns, such as in automated part placement or package routing systems. Although the use of customized patterns prevents the system from being usable in every environment, in many cases requiring a pattern to be used is not a major limitation. Of course, if the pattern can be specified in order to suit the capabilities and limitations of the machine vision system and detection algorithm, in return we expect very high performance from that system. Specifically, the detection for the pattern should be fast and highly accurate; especially when compared to more general object detection and tracking systems. We will describe such a system in the following sections. For low level vision, we will employ the freely available [17] CMVision [18] library. It performs color segmentation and connected components analysis to return colored regions at real time rates without special hardware or dedicating the entire CPU to the task.

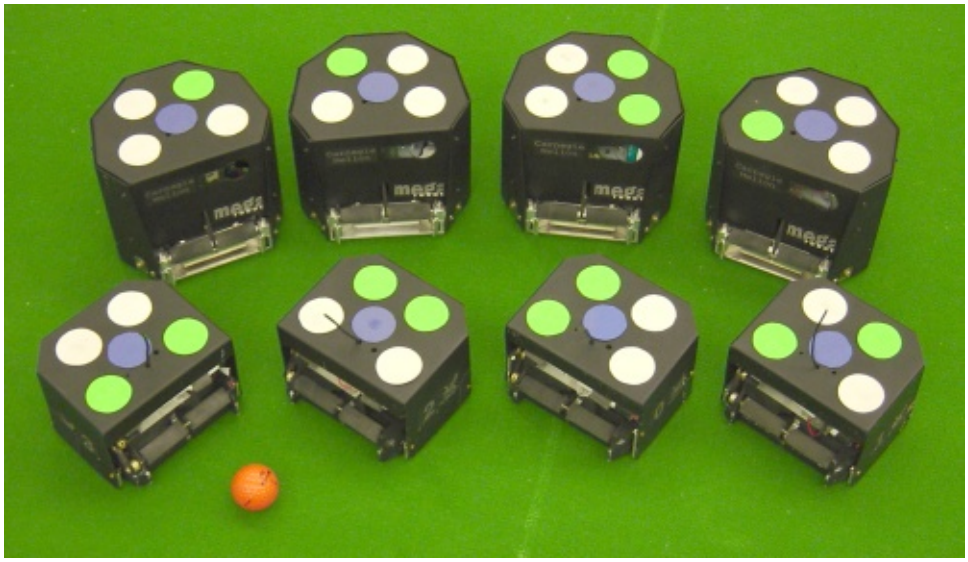


Figure A.5: The CMDragons'02 Robots. More recent robots use use the same marker pattern.

The environment in which most of this work has been done is the RoboCup [55] F180 "Small Size" League, where robots up to 18cm in diameter play soccer on a 2.8m by 2.3m carpeted soccer field. The game is played with two teams with five robots each, and uses an orange golf ball for the ball. One team must have a 40mm blue colored circle centered on the top of its robot while the other team must have a 40mm yellow patch. Teams may add extra patches and colors to the top of their robot to aid in tracking, so long as those colors are differentiable from the three standard colors (orange golf ball, yellow and blue team patches). The robots from our team, CMDragons'02 [22], can be seen in Figure A.5. Each team can

control its robots either onboard, or offboard, and cameras are allowed to be placed above the field. Thus, most teams use a single overhead camera, with an offboard PC interpreting the camera signal and sending commands to the robots via a radio link.

This environment thus poses a tracking problem for up to 11 small objects in known planes (in this case the possibly different, but known, heights above a ground plane). Few other environments currently demand accurate, multiple pattern detection at very high speed, but one such environment is Virtual or Augmented Reality. For these environments, patterns are tracked in order to localize head mounted displays and locate objects in the physical environment that are mapped into the virtual environment [26, 27]. Detection must be fast and accurate to minimize observable lag and jitter in the visualization. Due to work in these two environments, fast, accurate, multiple pattern detection has become better understood and more practical. Thus we expect many more applications for such tracking systems in the future.

Another aspect that makes the RoboCup F180 environment challenging is the high speeds of the tracked objects, since the robots move quite quickly relative to their size. Robot speeds peaking in excess of 2m/s are not uncommon, and ball speeds (via robot kicking mechanisms) can reach up to 5m/s. Thus we feel this is a good testbed for a tracking system. Two other vision tracking and identification systems for the F180 league have been described in [78] and [45]. Each describes a working system used by a team, but neither motivates the choice of pattern by a thorough analysis of the underlying feature error, or attempts to generalize detection to other similar patterns in order to compare their performance. In this paper we will outline the choices and trade-offs made in designing an identification and tracking pattern by gathering real and simulated data at each step so that informed tradeoffs can be made. We hope that this will help others to implement similar high performance tracking systems both within RoboCup and in many other environments where a similar problem exists. Such designs should not have to rely on any guesswork.

In the first section, we will motivate the type of patches chosen from which to build patterns, and examine their error distributions when viewed from a camera. In the second section, we will describe several common patterns and a broad class that includes most of the patterns. We will motivate the use of this most popular class for its simplicity and accuracy, and for which an efficient generic detection algorithm can be created. In the following section, the performance of several such patterns will be examined in simulation. Finally the best performing pattern from simulation will be evaluated on a real-time vision system.

A.2.1 Single Patches

In order to build up a detection pattern, we must have some simpler building block on which to build. We will use simple colored patches whose position can be calculated accurately. To detect orientation, multiple patches can be employed. For single patches, the simplest design to detect is to use a regular geometric shape of a single color. Detection in the vision system can be carried out on a binary or multiclass threshold image from which connected regions of common color class can be extracted. This approach is common, and is known to be quite efficient, so this is the approach we will use. The next variable to determine is patch shape. We chose circles, because they guarantee rotational invariance, and analytical corrections for the projective distortions of their image centroids are known [44]. In addition, they are compact, minimizing the length of the border with other regions, where thresholding is most difficult. In experiments, other regular shapes such as squares, hexagons, and octagons, perform roughly on par with circles. However, they do not offer any benefits to motivate their use in light of the analytical guarantees for circles.

The more difficult parameter to determine is what size of patch to use. When the dimensions of the overall pattern are known, this still leaves the question of whether it is better to have a pattern with a few large patches, or more patches where each is of a smaller size. To address this, we created a test setup where a small moving platform would carry three different sized white patches 2 meters across the field of view of a camera looking down from 3 meters. The platform moved at a slow constant speed (about 23mm/sec) allowing large amounts of data to be gathered from a variety of locations across the field. Using this setup, we gathered positional data at 30 samples/sec for 40mm, 50mm, and 60mm circles. A total of 5 runs were gathered, each one having about 2570 data points. As a convention, we labelled the dimension along the primary direction of travel as x, and the dimension perpendicular to the direction of travel as y.

Although there is no ground truth from which to measure true error, the error can be estimated by smoothing the data with a large Gaussian kernel ($\sigma = 10$) and then comparing single samples with the smoothed version of the signal. The aggregate errors appear to follow a Gaussian distributions quite well, as can be seen in Figure A.6. However, more outliers occurred than would be expected in a pure Gaussian distribution, and the variance seemed to change noticeably between runs, and even varied over different segments in the course of a single run. The most surprising result however, is that the size of the patch had very little effect. The overall standard deviations were around 0.52mm in both x and y for all sizes with only slight (although significant) variation. The cumulative distributions of absolute error in x and y are shown in Figure A.7.

The estimated standard deviations for each patch size can be found in Table A.2, along

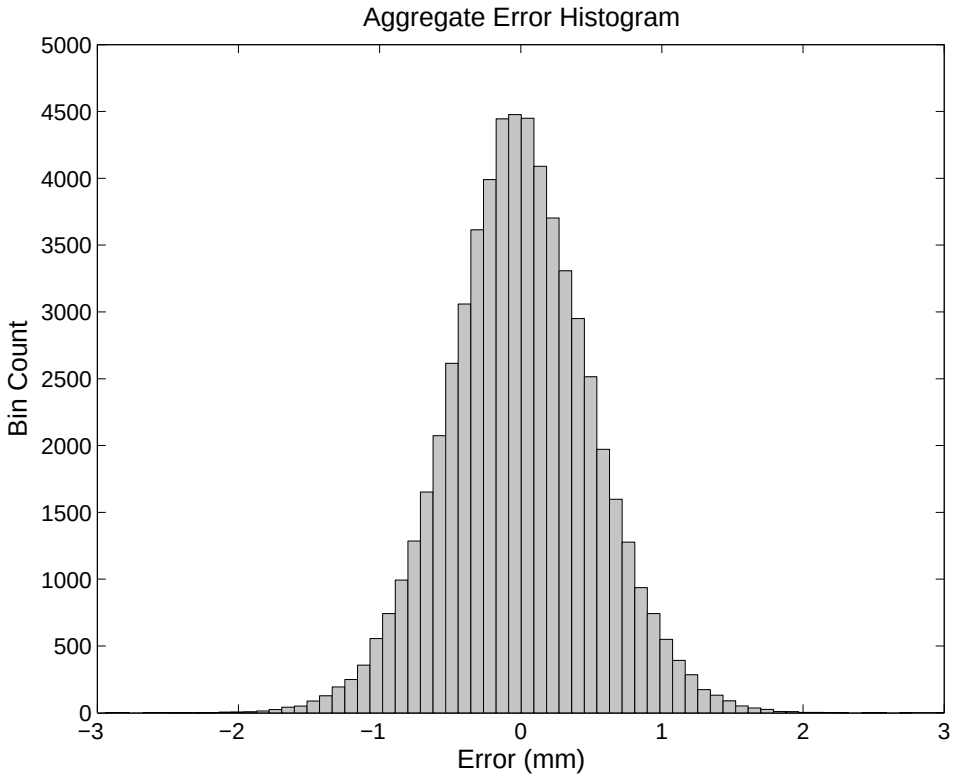


Figure A.6: Aggregate error distribution for all samples including all three patch diameters. This is actual data collected from the vision system.

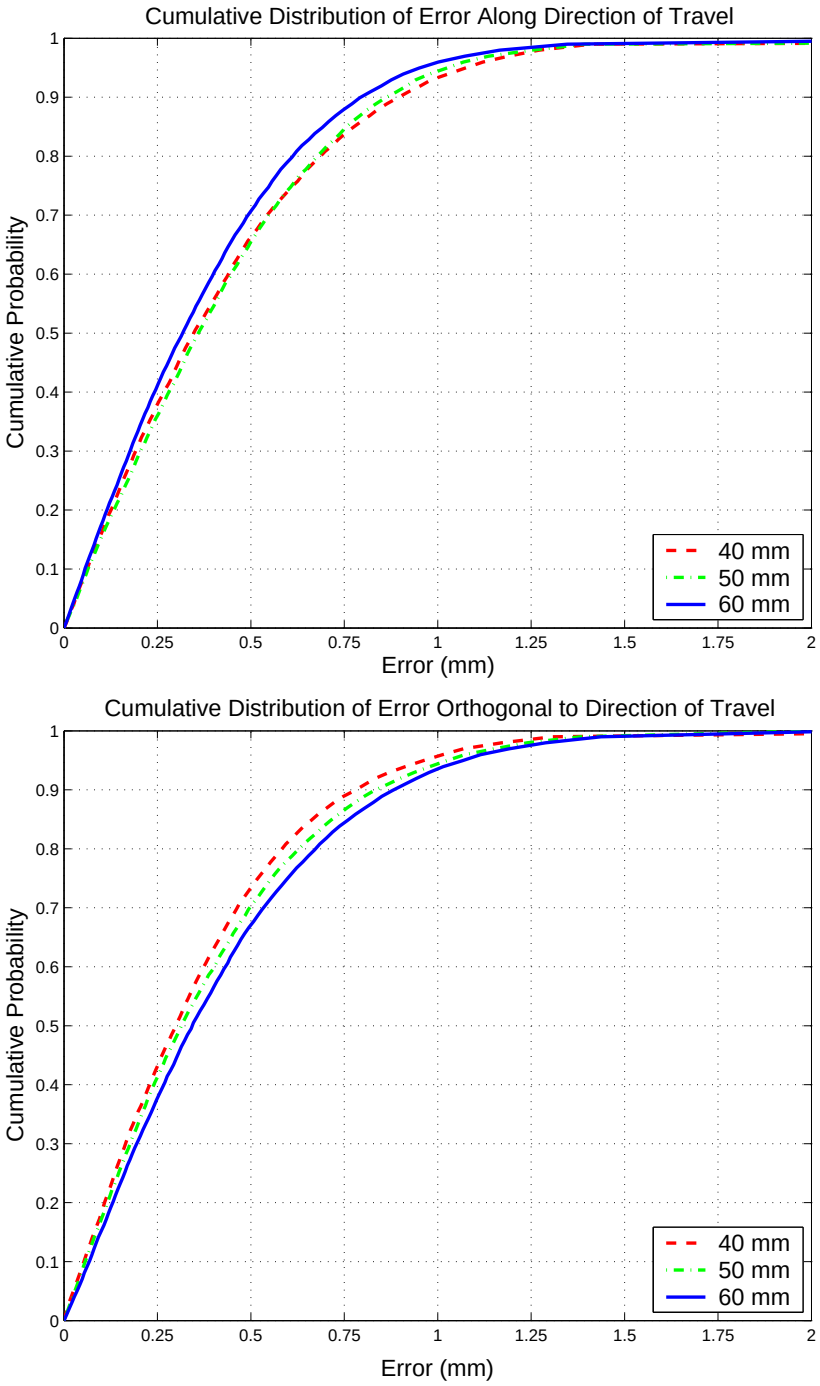


Figure A.7: Cumulative distributions of absolute error. Note that patch size does not have a large effect on error. Along the direction of travel, the largest patch size decreases error somewhat, but does worse than the smallest patch size perpendicular to the direction of travel.

Table A.2: The estimated standard deviations and 95% confidence intervals by patch size.

Diameter	σ_x	σ_y
40 mm	$0.553, [0.546 - 0.561]$	$0.473, [0.467 - 0.480]$
50 mm	$0.543, [0.535 - 0.550]$	$0.504, [0.497 - 0.511]$
60 mm	$0.489, [0.482 - 0.496]$	$0.533, [0.526 - 0.541]$

with 95% confidence intervals for the standard deviation. For hypothesis testing, we used the non-parametric Wilcoxon signed-rank test due to its robustness to outliers and lack of strong assumptions about the distributions being tested. The significant results (in all cases, $p < 0.0001$) were that along the direction of travel (x), the 60mm diameter circle had significantly less error than both the 50mm and 40mm patches. Perpendicular to travel (y) however, error *increased* with patch size, with all means being significantly different. Combined error in x and y indicated that the 50mm patch was slightly worse than the other two, with $p = 0.02$ against each of the other patches. Given the number of data points (over 10,000) however, we do not consider a difference at $p = 0.02$ ultimately conclusive. In addition, the difference in error from best to worst is less than 5%, which is much less variation than expected since the largest patch has 2.25 times the area of the smallest patch.

Thus the conclusion we can draw are that patches should be large enough they can be detected reliably, but need not be made any larger for purposes of accuracy. This is important in that it is contrary to conventional wisdom about region detection. It seems that other factors, such as quantization at edges due to pixels, play a larger part in determining the error than the area of the region. As we will show later, we can do somewhat better by adding more patches rather than using fewer patches and increasing their size.

A.2.2 Patterns and Detection

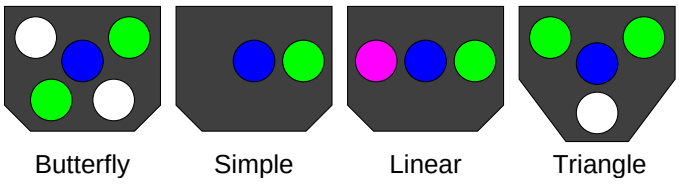


Figure A.8: Examples of common tracking patterns from the RoboCup F180 environment.

Now that the basis for choosing patches has been established, we can use this knowledge to evaluate tracking patterns. One source for many different ideas are the various patterns used by the over 20 teams in the RoboCup F180 League. The rules for the patterns on the top

of the robots in that league has naturally led to many tracking and identification patterns being tried. Examples of some of the more popular designs can be seen in Figure A.8. The approaches taken thus far generally fall into one of three broad categories. By far the most common type is like that shown above, which we call *patch based*, where in addition to the team marker patch in the center, one or more additional circular or rectangular patches are used to encode position and orientation. Patch based systems have the advantage that position and orientation detection can combine several features (patches in this case) with sub-pixel accuracy. One alternative to this is to have a key patch marking the center of the pattern, surrounded by radial "pie slices" of two or more colors. Orientation and identification can be performed by scanning at some constant radius from the central patch [45]. Unfortunately, by depending on features (color edges in this case) that are difficult to quickly detect with sub-pixel accuracy, it is difficult to get very accurate orientation using this method. Another type of tracking pattern that has been tried is to use a central patch with an nearby line feature. By finding the edge points of that feature, a least squares fit can be made to get an accurate orientation measurement. Unfortunately, both these classes of patterns do not offer a straightforward way to improve the position estimate. For a given pattern size, we would like to make the most of the space. Ideally, we would like all of the patches to contribute to position, orientation, and identification detection. In this regard, patch based systems tend to do quite well, which has led to them becoming the most common class of pattern used for tracking.

If we look again at Figure A.8, we can notice similarities that can aid in creating a generic detection algorithm. All are keyed by a colored patch (in the center of the pattern) indicating the presence of a pattern. Each patch occurs at some unambiguous angle radially from the key patch. Thus there is a distinct circular ordering of the non-key patches that can be calculated even in the presence of moderate noise. This means a generic detection algorithm can start by searching for the key patch, and then detecting and sorting the additional patches radially. All that needs to be done after that is to find the rotational correspondence of the additional patches with the geometric model of the pattern. In the case of the Simple pattern, the correspondence is trivial. For the Linear and Triangle patterns, the colors of patches can be used to disambiguate geometrically similar correspondences. Finally, in the case of the Butterfly pattern, geometric asymmetry can be used to find the rotational correspondence, assuming the distances between patches can be measured accurately enough. As shown in the previous section, this boils down to the difference of two Gaussians. When we consider the standard deviations determined in the previous section, distances that differ by over 10mm should be differentiated correctly with very high certainty. Using geometric asymmetry offers the benefit of freeing up the patch colors to encode only identification, rather that both identification and rotational correspondence. This gives the butterfly pattern (or any other geometrically asymmetric pattern) an identification advantage over symmetric patterns, as shown in Table A.3.

Table A.3: The number of uniquely identifiable patterns that can be detected using a certain number of colors (excluding the key patch and key patch color)

Pattern	2 colors	3 colors	4 colors	n colors
Butterfly	16	64	256	4^n
Simple	2	3	4	n
Linear	1	3	6	$n \cdot (n-1)/2$
Triangle	2	6	12	$n \cdot (n-1)$

Once the correspondence is established, the position and orientation estimates must be made. What we would like is to get near optimal detection but without resorting to iterative methods or other time consuming operations. Here we take a simple approach that turns out to be not only fast but in practice nearly indistinguishable from optimal formulations. First, the mean location of the patches is determined. This is an optimal estimate, although for many patterns this location is offset from the actual location we want to report (so it is not an optimal estimator of that point). After this mean position has been determined, we get displacement vectors between a pre-specified set of "orientation pairs" from the patches. These pairs should be well separated (because error decreases with distance), and different pairs that share a patch should be as orthogonal as possible (to avoid correlated errors). For the butterfly pattern, we use vectors between the four non-key patches. For the Triangle pattern we Similarly take the triangle edges formed by the pattern's three external patches. For the Linear and Simple pattern, only one nearly orthogonal pair exists. For the Linear pattern we choose the longest option of the opposite patches because this will minimize the error compared to the two shorter vectors that include the central key-patch. After the separation vectors are determined, they can be rotated into a consistent frame of reference because their angle relative to forward is known from the model of the pattern. Once all the vectors are lined up by the model, they can be added to form a single vector, and the arctangent calculated to get the angle measurement.

The motivation for adding the vectors comes from the observation that the angular error of a vector is roughly proportional to the separation of the patches when the separation distance (d) is much larger than the positional standard deviation (σ), or:

$$\sigma_\theta \approx \frac{\sqrt{2\sigma^2}}{d}$$

The term $\sqrt{2\sigma^2}$ comes from the subtraction of two Gaussians (since the separation distance is large this is roughly a 1D subtraction). The division by d is the result of arctangent being linear near the origin. In practice, we've found this approximation works well when $d > 10\sigma$. Finally, once the angle estimate is made, we can use this to project the mean of the patches to the coordinates of the patch that are to be reported (normally the origin of the patch

model coordinate system).

For comparison, we also derived an iterative Maximum Likelihood (ML) estimation method that co-optimizes position and angle estimates assuming Gaussian positional error for the patches. Its full derivation is omitted here for brevity. First, it is a well known fact that minimizing sum-squared-error in 1D is identical to maximizing the log likelihood (and thus likelihood) of samples from a Gaussian error distribution. Since in the 2D case variances can be added, this correspondence carries over into the 2D case. Thus by minimizing the sum-squared-error of the measured position of patches from their model positions given the estimated pattern position and orientation, we can obtain an ML estimate. So for a pattern with n patches, we define the current estimate of robot position as r, marker locations as $v_i \ldots v_n$, and patch locations from the pattern model as $p_i \ldots p_n$. If we let $s = \sin(r_\theta)$ and $c = \cos(r_\theta)$, then we have following derivatives for sum-squared-error E:

$$\begin{aligned} x_i &= r_{ix} + cp_{ix} - sp_{iy} - v_{ix} \\ y_i &= r_{iy} + sp_{ix} + cp_{iy} - v_{iy} \end{aligned}$$

$$\begin{aligned} \frac{\partial E}{\partial r_{ix}} &= \sum_{i=1..n} 2x_i \\ \frac{\partial E}{\partial r_{iy}} &= \sum_{i=1..n} 2y_i \\ \frac{\partial E}{\partial r_\theta} &= \sum_{i=1..n} 2x_i \cdot (-sp_{ix} - cp_{iy}) + 2y_i \cdot (cp_{ix} - sp_{iy}) \end{aligned}$$

These partial derivatives, along with the obvious implementation of the error function itself, can be used to create an iterative ML estimation method using Newton's method.

A.2.3 Pattern Comparison

In order to evaluate the patch-based patterns introduced earlier, a small simulator was created that would generate patch positions using a Gaussian error model for a pattern at random positions and orientations. Then the detection algorithm was run on the patches, and the resulting position and orientation measurements compared to the true values used to generate the input patches. The results for the simulation are shown in Figure A.9, plotted as pattern position and orientation standard deviation vs. input patch positional standard deviation. Each data point was generated from 100,000 simulated detections. One can

easily see that multi-patch patterns have a distinct advantage for both position and angle measurements. The Simple pattern can fair no better than a single patch using the generic detection algorithm described in the previous section. It could perhaps benefit more from maximum likelihood detection, but this would make detecting the Simple pattern slower than detecting the more complicated patterns. The Butterfly pattern has the most accurate position estimation, followed by Triangle and Linear at somewhat decreased accuracy. For angular error, the Butterfly pattern again shows the lowest error, with Triangle close behind. With their multiple patches allowing several well separated orientation pairs to be used, they both perform much better than Linear or Simple, each of which only have a single orientation pair. Linear fares better than the Simple pattern because the separation distance for its orientation pair is twice that of the Simple pattern. Another dimension along which we might compare is execution time. The simple pattern's position could be identified the fastest, taking $0.5\mu s$ on a 900MHz Athlon, while the Butterfly pattern required $2.67\mu s$. As expected, the time was nearly proportional to the number of patches and orientation pairs. However, these times negligible compared to the roughly $1000\mu s$ it takes to threshold and segment a frame of video.

Since the Butterfly pattern worked best for both positional and angular error in simulation, we decided to make further tests to evaluate its performance using the iterative maximum likelihood detection and then measure the pattern's performance on a real vision system. To compare our generic detection algorithm with the ML estimate, we ran each of the detection methods in simulation. Even after 100,000 samples, no statistically significant differences could be detected with a Kolmogorov-Smirnov test, leading to the conclusion that at least for complicated patterns, the simple detection algorithm was indistinguishable in terms of accuracy from the ML estimation.

Finally, we evaluated the Butterfly pattern on a real vision system. We ran 5 runs similar to those for single patches but this time with a pattern being tracked rather than individual patches. The overall standard deviations were $\sigma_x = 0.3766$mm, $\sigma_y = 0.3432$mm, and $\sigma_\theta = 0.0070$rad. This was significantly better than a single patch ($p < 0.0001$), although not as low as predicted by simulation. The error was about 70% higher than predicted, which is most likely explained by some correlation in the patches' error (such as error from camera jitter). The pattern error was more consistent with a patch standard deviation of around 0.8mm, and thus could also have been due to outliers affecting the measurements.

Another possible problem (but one that is easily measurable) is correlation of errors over time. Typically filters assume that all readings are independent measurements, however this may not be the case for some sources of error. In Figure A.10, we show 2D scatter plots of adjacent readings for a single patch (top) and for a full pattern (bottom). The single patch shows structure indicating that errors are likely to repeat (the center diagonal stripe) or jump up or

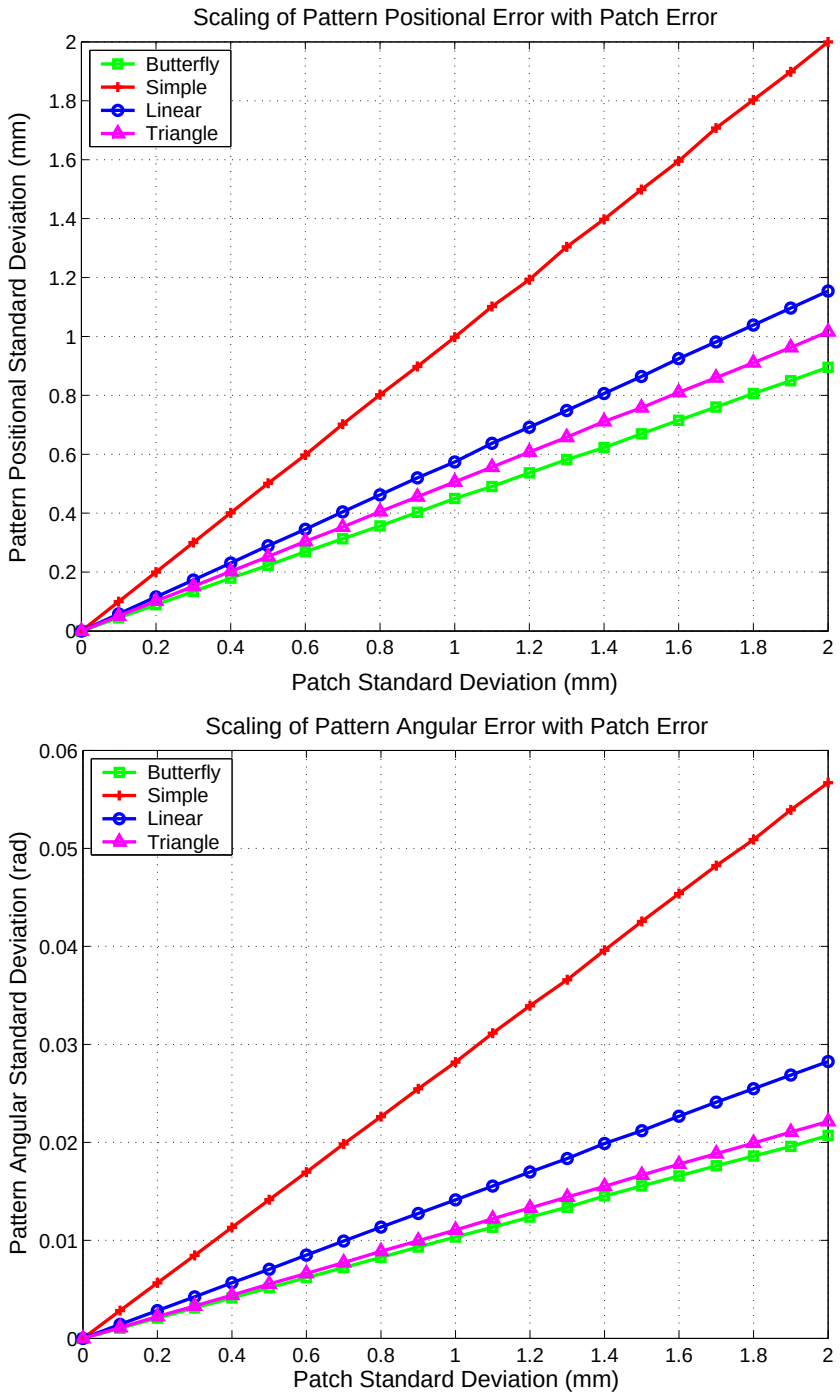


Figure A.9: Comparison of the positional and angular error of different patterns as the error of individual patches vary. For relatively small patch standard deviations, a linear relationship exists between the two, although the number of patches and layout of the pattern vary the factor.

down by a fixed amount (the upper and lower diagonal stripes). As best we could determine, this appears to be due to the binary color segmentation; As the patch moves across the camera image, pixels switch from background color to patch color (and back to background) abruptly, changing the location of the centroid by fixed amounts. The full pattern (bottom) does not display this structure (most likely because combining 5 patches made the structured error of individual patches small enough to make it unnoticeable). However the plot is still not an unbiased circular cloud, so adjacent readings are still somewhat correlated. With a few time steps separating readings, no observable correlations are present. Thus, assuming measurements are independent for patterns seems to be a reasonable simplification, although increasing the standard deviation to a more conservative estimate may be prudent. Assuming independence for patches may be more problematic, so for tracking single patches the extra complexity of modeling error correlation may be necessary.

A.2.4 Summary of Pattern Vision

This section presents the derivation of an efficient and highly accurate detection algorithm along with an analysis of the performance of many different patch-based patterns. It is designed to be paired with a low-level vision system (such as CMVision) to construct a global vision system for a multi-robot system. We first look at the performance of single patch detection, noting that size, although important for robust detection, does not have a large effect on the accuracy of the positional measurement of a patch. We presented a fast patch based detection algorithm along with an iterative ML variant, which perform similarly in terms of accuracy. We compared several patterns in simulation to find out how accurate their detection scaled with the error of the patches form which they were made. We then tested a pattern on a real vision system with positive results, and examined the assumption of independence on which higher levels of an object tracking system rely. In particular, the data supports the common assumption of objects with additive Gaussian noise on position, and provides measurements to guide the design of error tolerance into decision and navigation systems for robot agents.

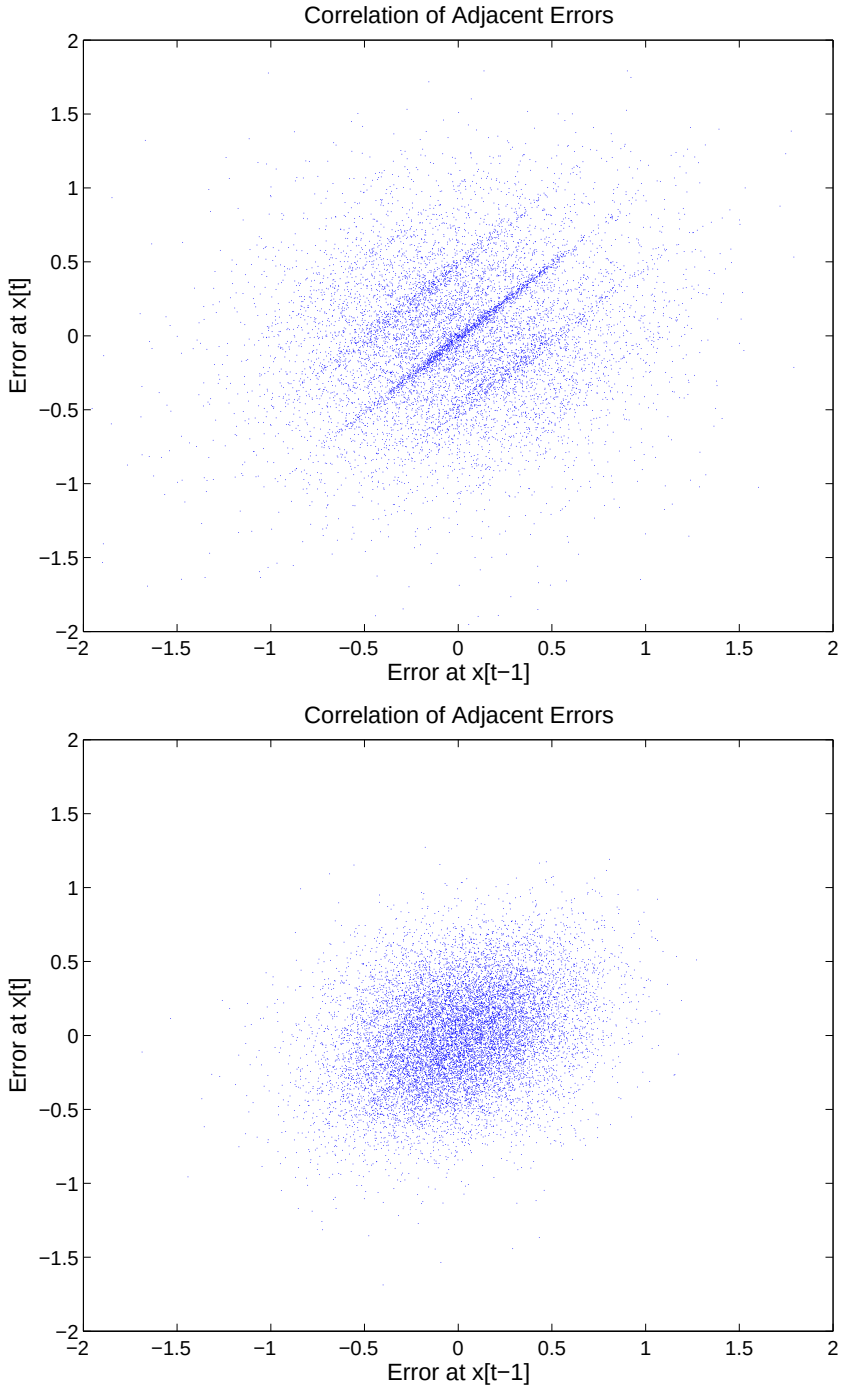


Figure A.10: 2D scatter plots of adjacent readings for single patches (top) and for a full Butterfly pattern (bottom). Uncorrelated readings would show up as a circular 2D Gaussian cloud. Note the structure in the plot for a single patch, and the unstructured but non-circular distribution for the full pattern.

Appendix B

The CMDragons Multi-Robot System

For the past eight years, we have been pursuing this domain as a research problem of teams of autonomous robots acting in adversarial, dynamic environments. Our RoboCup Small-Size League entry, CMDragons 2006 builds on the successful past entries of CMDragons and the joint team CMRoboDragons, which entered in RoboCup 2004 and 2005 and placed fourth overall both years. It also builds on the prior experience of CMDragons entries in 1998-99 and 2001-03 [16, 22, 70]. Our team entry consists of five omni-directional robots controlled by an offboard computer. Sensing is provided by two overhead mounted cameras linked to frame-grabbers on the offboard computer. The software then sends driving commands to the individual robots. Each robot has four drive wheels, a kicker, and a dribbler. The robot can also send status data back to the offboard control computer to augment the overhead vision information.

Motivated by this challenging multi-robot scenario, we have focused on investigating general multi-robot path planning. Effective multi-robot path planning is a very complex problem especially in highly dynamic environments, such as our robot soccer task. Robots need to rapidly navigate avoiding each other and obstacles while aiming at reaching specific objectives. We decompose the multi-robot path planning problem into two parts: (i) the definition of target points that each robot needs to achieve, and (ii) the multi-robot planning and safe navigation towards the assigned target points. This article briefly addresses the setting of objectives, and focuses in detail on safe multi-robot navigation. Although the systems described here are motivated by the specific requirements of the RoboCup domain, by looking at the common elements across many versions we realize these parts are of wider applicability in other single-robot or multi-robot domains, especially where high speed and safety are desired.

Throughout our design and algorithms, we assume a system where the robotic agents are distributed, but decisions are made in a centralized fashion. In particular, all agents share a common world state with only pure Gaussian noise in state estimates, and we assume perfect communication of actions between all agents. Though centralized, the system is still distinct from a completely monolithic design where all agents share a single world state vector and joint action space. First, the execution time of our approach scales linearly with the number of agents (rather than the exponential scaling typical of a monolithic design). Second, the state required to be communicated between agents is of fixed bounded size, and is not directly related to the complexity of per-agent local calculations. Thus, although centralized, our approach is closer to meeting the restrictions of a distributed algorithm, and we expect it would serve as a good starting point for such a future development.

Offboard communication and computation is allowed, leading nearly every team to use a centralized approach for most of the robot control. The data flow in our system, typical of most teams, is shown in Figure B.1 [16, 22]. Sensing is provided by two or more overhead cameras, feeding into a central computer to process the image and locate the 10 robots and the ball on the field 30-60 times per second. These locations are fed into an extended Kalman filter for tracking and velocity estimation, and then sent to a "soccer" module which implements the team strategy using various techniques. The three major parts of the soccer system are: (1) world state evaluation, (2) tactics and skills, and (3) navigation. World state evaluation involves determining high level states about the world, such as whether the team is on offense or defense. It also involves finding and ranking possible subgoals, such as evaluating a good location to receive a pass. Tactics and skills implement the primitive behaviors for a robot, and can range from the simple "go to point" skill, up to complex tactics such as "pass" or "shoot". A *role* in our system is defined as a tactic with all parameters fully specified, which is then assigned to a robot to execute. Given these parameters, along with the world state, the tactic generates a navigation target either directly or by invoking lower level skills with specific parameters. Finally, these navigation targets are passed to a navigation module, which employs randomized path planning, motion control, and dynamic obstacle avoidance. The resulting velocity commands are sent to the robots via a serial radio link. Due to its competitive nature, over the years teams have pushed the limits of robotic technology, with the some small robots travelling over $2m/s$, with accelerations between $3-6m/s^2$, and kicking the ball used in the game at up to $10m/s$. Their speeds require every module to run in real-time to minimize latency, all while leaving enough computing resources for all the other modules to operate. In addition, all the included algorithms involved must operate robustly due to the full autonomy requirement.

Two parts of our soccer system which have proven stable in design over the several iterations of our team are the world state evaluation for position determination and navigation module for real-time motion planning. First, determining supporting roles through objective

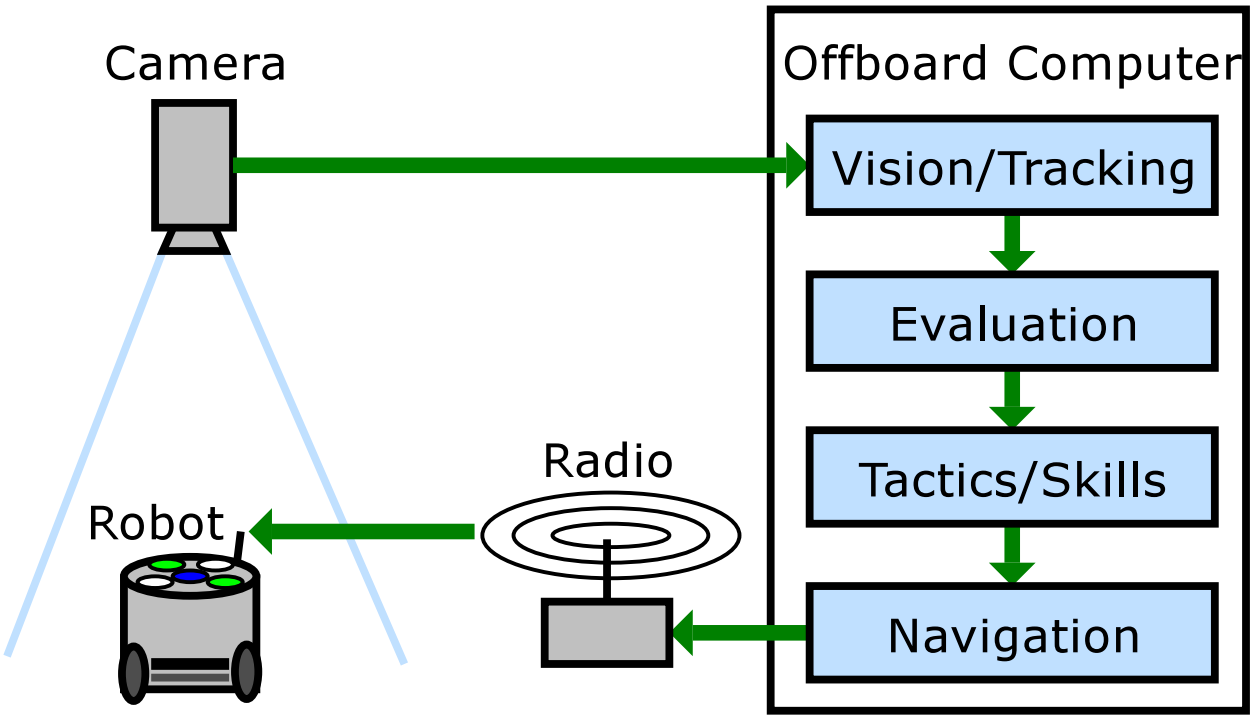


Figure B.1: The overall system architecture for CMUnited/CMDragons. Thanks in particular to Brett Browning and Michael Bowling for their contributions to the overall team.

functions and constraints has proven a very natural and flexible way of allowing multiple robots to support an active player on offense, and to implement defensive strategies on defense. This paper will describe the common elements of this module which have been present throughout its evolution. Second, navigation has always been a critical component in every version of the system. Parts of the system described here were first present in 2002, with the latest dynamic safety module debuting in 2005. The navigation system's design is built on experience gained since 1997 working on fast navigation for small high performance robots [11, 86].

In the RoboCup domain, we found the navigation system to work well in practice, helping our team consistently place within the top four teams, with comparatively few penalties for aggressive play. In testing, the planner in the RoboCup achieved execution times below 1ms to meet the tight timing requirement of the small-size system (in practice the planner is aborted early if too much time is used, with the associated degradation in navigation performance).

B.1 Software Overview

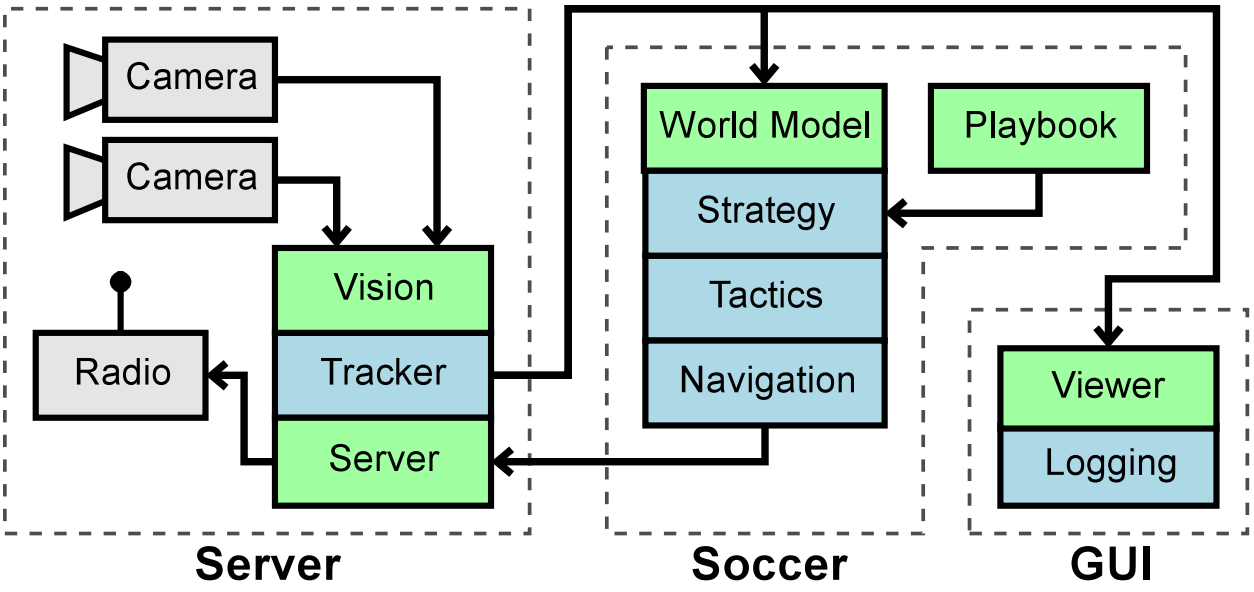


Figure B.2: The general architecture of the CMDragons offboard control software.

The software architecture for our offboard control system is shown in Figure B.2. The major organizational components of the system are a server program which performs vision and manages communication with the robots. Two other major client programs connect to the server via UDP sockets. The first is a soccer program, which implements the soccer playing strategy and robot navigation and control, and the second is a graphical interface program for monitoring and controlling the system.

The server program consists of vision, tracker, radio, and a multi-client server. The vision system uses CMVision2 for low-level image segmentation and connected region analysis [17, 18]. On top of this system lies a high-level vision system for detecting the ball and robot patterns. Our robot pattern detector uses an efficient and accurate algorithm for multi-dot patterns described in [20]. Tracking is achieved using a probabilistic method based on Extended Kalman-Bucy filters to obtain filtered estimates of ball and robot positions. Additionally, the filters provide velocity estimates for all tracked objects. Further details on tracking are provided in [16]. The radio system sends short commands to each robot over a RS232 radio link. The system allows multiple priority levels so that different clients may control the same robots with appropriate overriding. This allows, for example, a joystick tele-operation program to override one of the soccer agents temporarily while the soccer system is running.

The soccer program is based on the STP framework [16]. A world model interprets the incoming tracking state to extract useful high level features (such as ball possession information), and act as a running database of the last several seconds of overall state history. This allows the remainder of the soccer system to access current state, and query recent past state as well as predictions of future state through the Kalman filter. The highest level of our soccer behavior system is a strategy layer that selects among a set of plays [10, 22]. Below this we use a tree of tactics to implement the various roles (attacker, goalie, defender), which in turn build on sub-tactics known as skills [16]. One primitive skill used by almost all behaviors is the navigation module, which uses the RRT-based ERRT randomized path planner [23, 51, 65] combined with a dynamics-aware safety method to ensure safe navigation when desired [24]. It is an extension of the Dynamic Window method [12, 38]. The robot motion control uses trapezoidal velocity profiles (bang-bang acceleration) as described in [16, 22].

B.2 Robot Hardware

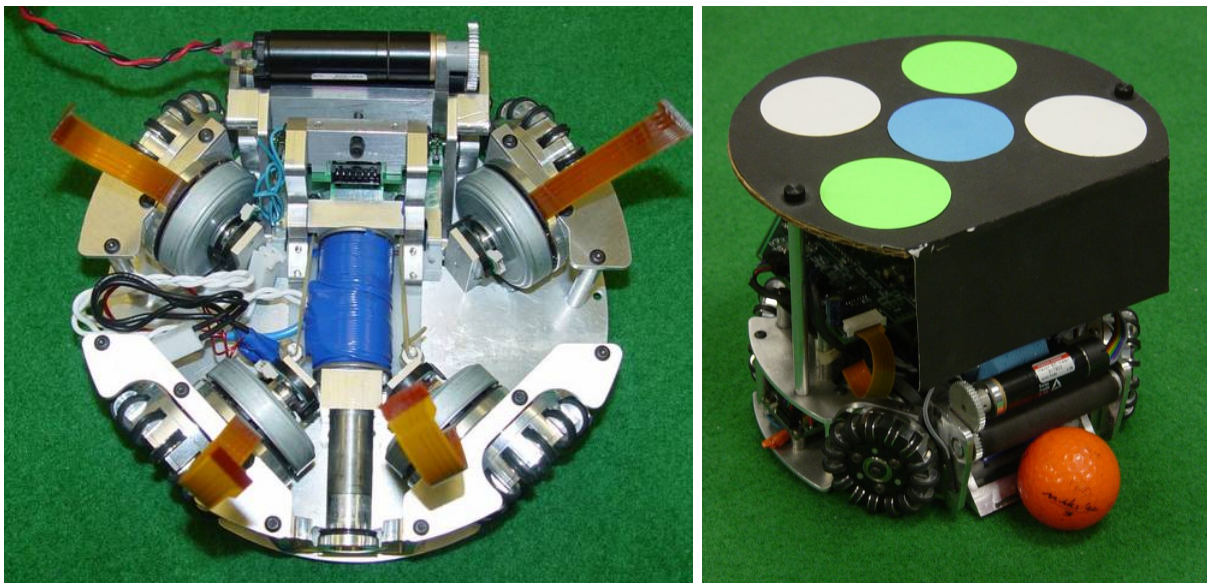


Figure B.3: View of the robot drive system, kicker, and dribbler (left), and an assembled robot without a protective cover (right).

Our team consists of five homogeneous robot agents. Each robot is omni-directional, with four custom-built wheels driven by 30 watt brushless motors. Each motor has a reflective quadrature encoder for accurate wheel travel and speed estimation. The kicker is a large

diameter custom wound solenoid attached directly to a kicking plate, which offers improved durability compared to the previous design which used a standard D-frame solenoid pushing a kicking plate with guide rods. Ball catching and handling is performed by an actuated, rubber-coated dribbling bar which is mounted on a hinged damper for improved pass reception. The robot drive system and kicker are shown in Figure B.3. Our robot fits within the maximum dimensions specified in the official rules, with a maximum diameter of 178mm and a height of 143mm. The dribbler holds up to 19% of the ball when receiving a pass, and somewhat less when the ball is at rest or during normal dribbling.

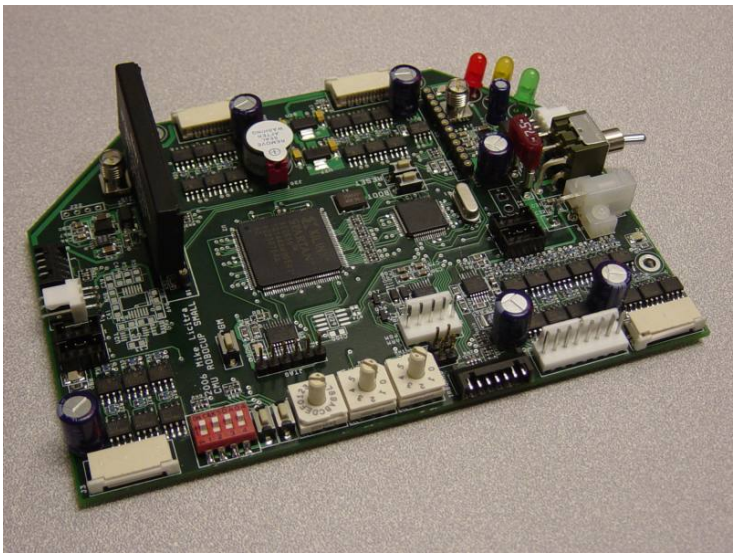


Figure B.4: The main robot electronics board for CMDragons 2006.

The robot electronics consists of an ARM7 core running at 58MHz linked to a Xilinx Spartan2 FPGA. The ARM7 core handles communication, runs the PD control calculations, and monitors onboard systems. The FPGA implements the quadrature decoders, PWM generation, serial communication with other onboard devices, and operates a beeper for audible debugging output. The main electronics board, which integrates all electronic components except for a separate kicker board and IR ball sensors, is shown in Figure B.4. This high level of integration helps to keep the electronics compact and robust, and helps to maintain a low center of gravity compared to multi-board designs. Despite the small size, a reasonable amount of onboard computation is possible. Specifically, by offloading the many resource intensive operations onto the FPGA, the ARM CPU is freed to perform relatively complex calculations.

For improved angular control, the robot also incorporates an angular rate gyroscope. It is linked into the drive control system via a secondary proportional controller, and can be used

to move to or maintain a specific heading. Whenever the robot is visible to the overhead camera, the coordinate system is updated periodically using the angle determined by the vision system. This allows the local heading on the robot to match the heading for the global coordinate system. Using the gyro and secondary control loop, the robot is able to maintain a stable heading even during periods of no vision lasting several seconds.

B.3 Robot Model

Thus we can assume wheel centers at vectors $w_i, i \in [0, 3]$ relative to the center of the robot, each pointed perpendicular to the w_i vector along unit vectors n_i.

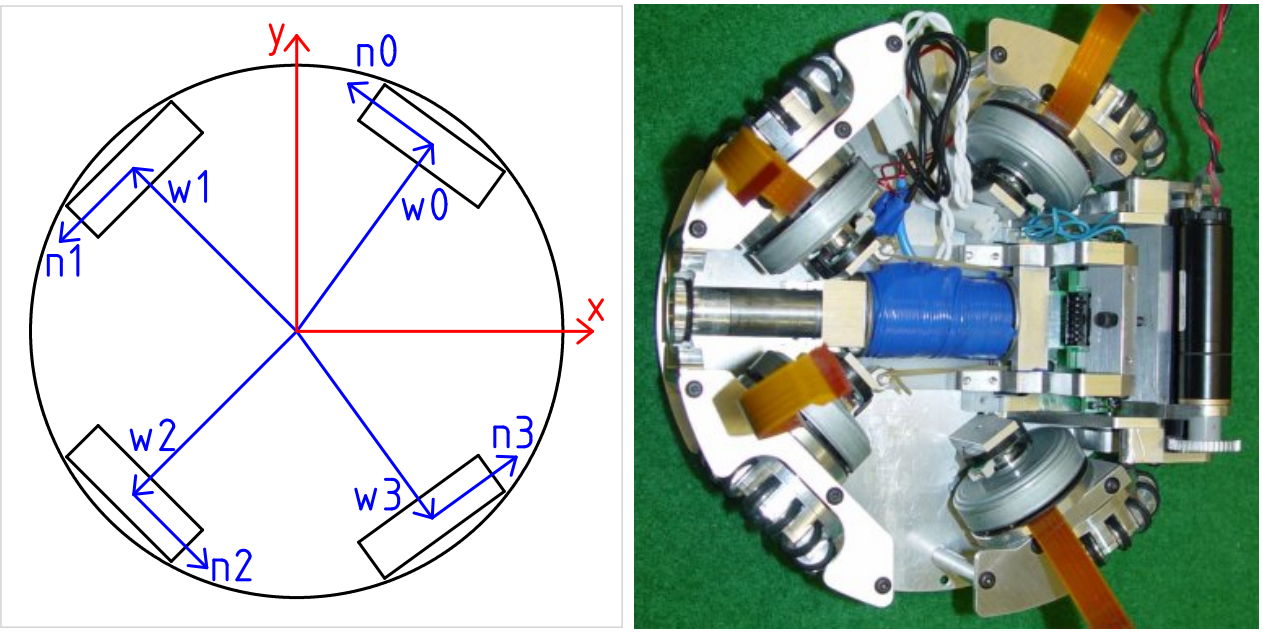


Figure B.5: Model of a CMDragons holonomic robot with four omni-directional wheels (left), and a picture of a partially assembled robot (right).

If each wheel of the robot is driven by an electric motor, we can assume the time to generate a particular force on a wheel is negligibly small, and as a simplified approximation we can treat the wheels as points which can exert force along their respective n_i vectors, up to some bounded maximum velocity. This is shown graphically in Figure B.5 The force an electric motor (and thus the wheel) can exert decreases linearly from a maximum when the wheel is stationary down to a force of zero when the wheel is rotating at the maximum no-load speed.

This leads to the following dynamics equations, where F is the total force on the robot and τ is the torque:

$$F(t) = \sum_{i=0..3} n_i f_i(t) \quad \text{(B.1)}$$

$$\tau(t) = \sum_{i=0..3} w_i \times n_i f_i(t) \quad \text{(B.2)}$$

$$\ddot{q}(t) = mF(t) \quad \text{(B.3)}$$

$$\dot{\omega}(t) = I\tau(t) \quad \text{(B.4)}$$

For a relatively high performance robot however, the motor torque often exceeds the friction the wheel can exert on the ground, resulting in a traction-limited maximum force. Although again reality is quite a bit more complex, we will assume a fixed maximum no-slip traction force. These traction problems lead to controller stability issues with a full multiple-input multiple-output (MIMO) controller, as shown for a similar RoboCup robot in Sherback et al. [77]. As in [77], we adopt the simpler single-input single-output (SISO) controller model, with a velocity PD loop at each wheel. Unfortunately this leads to some torque imbalance during acceleration, as Equation B.2 is not guaranteed to be equal to zero. However, with a sufficiently stiff controller, the imbalance is within tolerable error.

To model the robot, we decouple translation and rotation, reserving a small amount of the total control effort for angle, and leaving the rest for translation. The robot has a small moment of inertia, and little need to turn at maximum speed, so this tradeoff is acceptable. To model the maximum traction force, we adopt a translational acceleration limit as in [52] and [77]. We add an additional ability to decelerate at a constant larger than the acceleration, leading to a bounding half-ellipse shape (See Section 5.1, and in particular Figure 5.2).

B.4 Motion Control

Once the planner determines a waypoint for the robot to drive to in order to move toward the goal, it feeds this target to the motion control layer. The motion control system is responsible for commanding the robot to reach the target waypoint from its current state, while subject to the physical constraints of the robot as described above. The model we will take for our robot is a three or four wheeled omnidirectional robot, with bounded acceleration and a maximum velocity. The acceleration is bounded by a constant on two independent

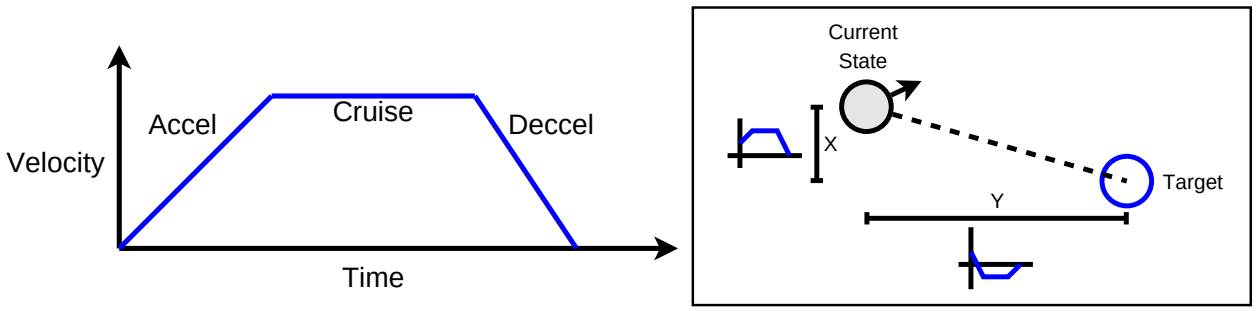


Figure B.6: Our motion control approach uses trapezoidal velocity profiles. For the 2D case, the problem can decomposed into two 1D problems, with the division of control effort balanced using binary search.

axes, which models a four-wheeled omnidirectional robot well. In addition, deceleration is a separate constant from acceleration, since braking can often be done more quickly than increasing speed. The approach taken for motion control is the well known trapezoidal velocity profile. In other words, to move along a dimension, the velocity is increased at maximum acceleration until the robot reaches its maximum speed, and then it decelerates at the maximum allowed value to stop at the final destination. An example velocity profile is shown in Figure B.6. The area traced out by the trapezoid is the displacement effected by the robot. For motion in 2D, the problem is decomposed as two 1D motion problems along x and y. For each direction, we allow some multiple of the total control effort, based on a position on the boundary of the acceleration half-ellipse (see the previous section for a description). The relative weights are altered to minimize the maximum execution time of the two directions. This balancing is achieved using binary search, always increasing the weight of the direction requiring more time to execute [77].

While the technique of trapezoidal control is well known, the implementation focuses on robustness even in the presence of numerical inaccuracies, changing velocity or acceleration constraints, and the inability to send more than one velocity command per cycle. First, for stability in the 2D case, if the initial and target points are very close, the coordinate frame can become degenerate. In that case the last coordinate frame above the distance threshold is used. For the 1D case, the entire velocity profile is constructed before calculating the command, so the behavior over the entire command period (1/60 to 1/30 of a second) can be represented. The calculation proceeds in the following stages:

- If the current velocity has a different sign than the difference in positions, decelerate to a complete stop.
- Alternatively, if the current velocity will overshoot the target, decelerate to a

complete stop.

- If the current velocity exceeds the maximum, decelerate to the maximum.
- Calculate a triangular velocity profile that will close the gap.
- If the peak of the triangular profile exceeds the maximum speed, calculate a trapezoidal velocity profile.

Although these rules construct a velocity profile that will reach the target point if nothing impedes the robot, limited bandwidth to the robot servo loop necessitates turning the full profile into a single command for each cycle. The most stable version of generating a command was to simply select the velocity in the profile after one command period has elapsed. Using this method prevents overshooting, but does mean that very small short motions will not actually occur (when the entire profile is shorter than a command period). In these cases it may be desirable to switch to a position based servo loop rather than a velocity base servo loop if accurate tracking is desired.

B.5 Objective Assignment for Multi-Robot Planning

Multi-robot domains can be categorized according to many different factors. One such factor is the underlying parallelism of the task to be achieved. In highly parallel domains, robots can complete parts of the task separately, and mainly need to coordinate to achieve higher efficiency. In a more serialized domain, some part of the problem can only be achieved by one robot at a time, necessitating tighter coordination to achieve the objective efficiently. Occasionally, even tighter coordination is needed with multiple robots executing joint actions in concert, such as for a passing play between robots, or joint manipulation.

Robotic soccer falls generally between parallel and serialized domains, with brief periods of joint actions. Soccer is a serialized domain mainly due to the presence of a single ball; At any given time only one robot should be actively handling the ball, even though all the teammates need to act. In these domains, multi-robot coordination algorithms need to reason about the robot that actively addresses the serial task and to assign supporting objectives to the other members of the team. Typically, there is one *active* robot with multiple robots in *supporting* roles. These supporting roles give rise to the parallel component of the domain, since each robot can execute different actions in a possibly loosely coupled way to support the overall team objective.

A great body of existing work exists for task assignment and execution in multi-agent systems. Gerkey and Mataric [43] provide an overview and taxonomy of task assignment meth-

ods for multi-robot systems. Uchibe [85] points out the direct conflicts that can arise between multiple executing behaviors, as well as complications arising when the number of tasks does not match the number of robots. A module selection and assignment method with conflict resolution based on priorities was presented. D'Angelo [29] et al. present a cooperation method that handles tight coordination in a soccer domain via messaging between behaviors executing the cooperating agents. Task assignment for our early robot soccer system is given in Veloso et al. [86], while more recent methods are described by Browning et al. [16, 22]. Beyond assignment of tasks to agents, their still lies to problem of describing how each agent should implement its local behavior. Brooks [14] presents a layered architecture using subsumptive rules, while Tivoli [84] presents an artificial potential field for local obstacle avoidance. Arkin [3] presents the method of motor schema, which extends the idea of potential functions to include weighted multiple objectives so that more complex tasks can be carried out. All three of these approaches calculate behaviors based on local sensor views. Koren and Borenstein [58] point out the limitations of direct local execution of potential functions. In Latombe [62], potentials are defined over the entire workspace and used as guidance to a planner, alleviating most of the problems with local minima. The MAPS system [5, 83] described by Tews et al. uses workspace potential functions which are combined to define behaviors in a robotic soccer domain. The potentials are sampled on an evenly spaced grid, and different primitives are combined with weights to define more complex evaluations. The potential is used as input to a grid-based planner and to determine targets for local obstacle avoidance. The method of strategic positioning via attraction and repulsion (SPAR) is presented in Veloso et al. [86], and describes the first method used in our system. It combines binary constraints with linear objective functions to define a potential over the workspace. The system could be solved using linear programming or sampling on a regular grid defined in the workspace. It was successfully applied in the RoboCup small size environment using grid sampling. More recently, Weigel et al. [88] uses a potential approach similar to SPAR but with purely continuous functions defined on a grid. Continuous functions guarantee all locations have a well defined value so that A* [72] search can be directly applied. Laue et al. [63] describe a workspace potential field system with a tree-based continuous planner to calculate a path to the location of maximum potential.

Task allocation in our system is described in Browning et al. [16]. Our systems adopts a split of active and support roles and solves each of those subtasks with a different method. Active roles which manipulate the ball generate commands directly, receiving the highest priority so that supporting roles do not interfere. Supporting roles are based on optimization of potential functions defined over the configuration space, as in SPAR and later work. With these two distinct solutions, part of our system is optimized for the serialized aspect of ball handling, while another part is specialized for the loosely coupled supporting roles. We address the need for the even more tight coupling that is present in passing plays through behavior dependent signalling between the active and supporting behaviors as in D'Angelo

et al. [29]. In this paper we briefly present the potential function maximization used by support roles, and then describe how the resulting navigational target is achieved.

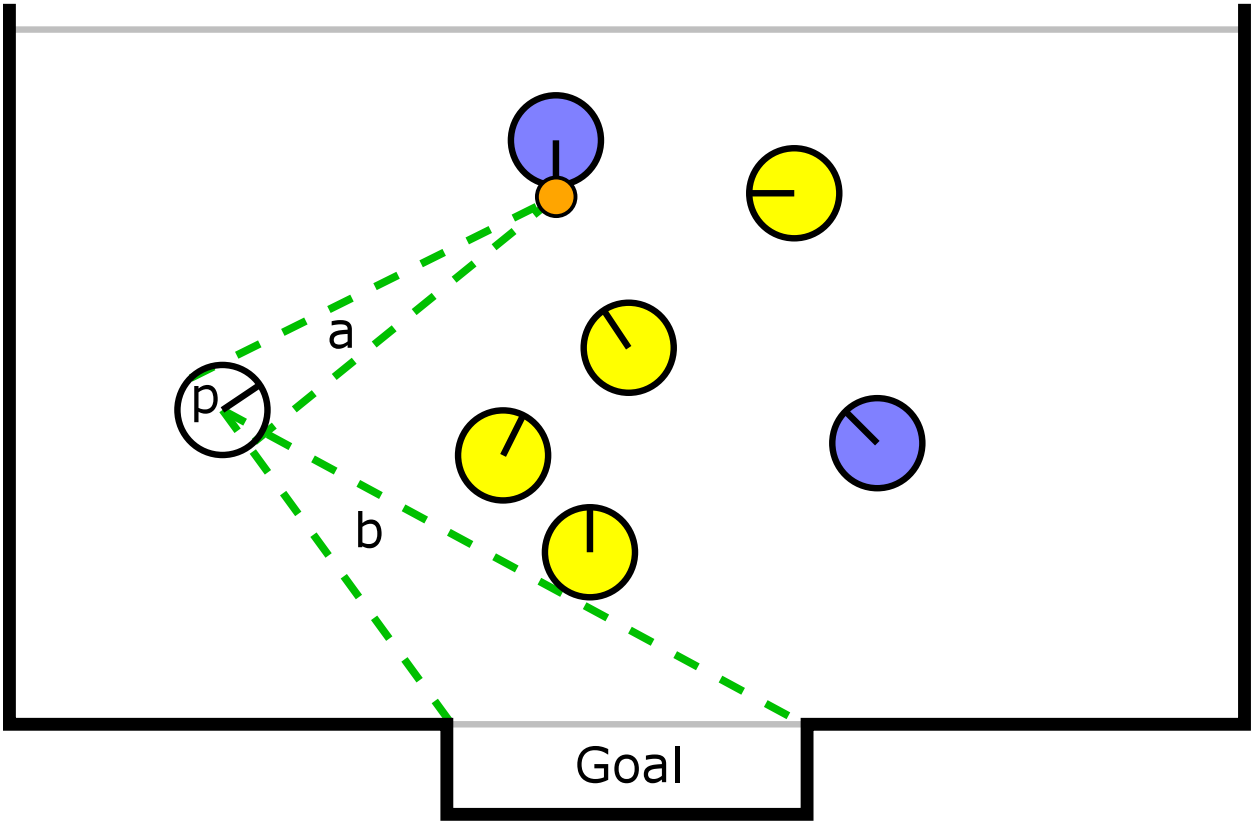


Figure B.7: A example situation demonstrating the passing evaluation function inputs. The input values for evaluating a point p are the circular radius and subtended angle of the arcs a and b, and the angle between the center line segments of the two arcs. These are combined using simple weighted functions to achieve the desired behavior.

In our system, the navigation targets of supporting roles are determined by world state evaluation functions defined over the entire soccer field. Each function holds the world state external to a robot constant, while varying the location of the robot to determine a real valued evaluation of that state within the workspace. Hard constraints are represented using multiplicative boolean functions, whereas soft constraints are modelled as general real-valued functions. An example of the general form of the pass position evaluation function is given in Figure B.7. Passing in our system is optimized for a single pass-and-shoot behavior, and the parameters are set to value states where this can execute. First, the angular span of a receiving robot's area at a workspace position p, and the angular span of the open goal area aiming from p are introduced as linear functions with positive weights. Next the

relative lengths of the passes are combined to encourage a pass and shot of equal length, maximizing the minimum speed of the ball at any point due to friction. This is introduced using a Gaussian function of the difference in lengths of the pass (a) and shot (b). Finally, interception for a shot is easiest at right angles, so a Gaussian function is applied to the dot product of the pass and shot relative vectors ($a \cdot b$). The weights were set experimentally to achieve the desired behavior. The resulting plots from two example passing situations are shown in Figure B.8.

While the exact parameters and weights applied in evaluation functions are highly application dependent, and thus not of general importance, the approach has proved of useful throughout many revisions of our system. In particular, with well designed functions, the approach has the useful property that large areas have a nonzero evaluation. This provides a natural ranking of alternative positions so that conflicts can be resolved. Thus multiple robots to be placed on tasks with conflicting actions, or even the same task; The calculation for a particular robot simply needs to discount the areas currently occupied by other robots. Direct calculation of actions, as is used for active roles, does not inherently provide ranked alternatives, and thus leads to conflicting actions when other robots apply the same technique.

In order to generate concrete navigation targets for each robot, we must find maximal values of each robot's assigned evaluation function. We would like to do so as efficiently as possible, so that the complexity of potential functions is not a limiting factor. We achieve the necessary efficiency through a combination of hill climbing and randomized sampling. First, a fixed number of samples is evaluated randomly over the domain of the evaluation function, and the sampled point with the best evaluation is recorded. We call this point the *sampled-best*. Two other interesting points are the point of maximum evaluation recorded from the last control cycle, called *previous-best*, and the current location of the robot *current-loc*. For each of these three points, we apply hill climbing to the point with a limited number of steps to find a local maximum. The best of the three is taken as the maximum, and is used as the robots navigation target. It is also recorded for use in the next control cycle as *previous-best*. Using this approach, few random samples need to be evaluated each frame, lending itself to real-time performance. However, because previous maximal values are recorded, the calculations of several control cycles are leveraged to quickly find the global maximum for a sufficiently smooth and slowly changing function. The *current-loc* point is added to provide more consistent output in situations where the maximum of the function changes rapidly. This is normally due to rapid changes in the environment itself.

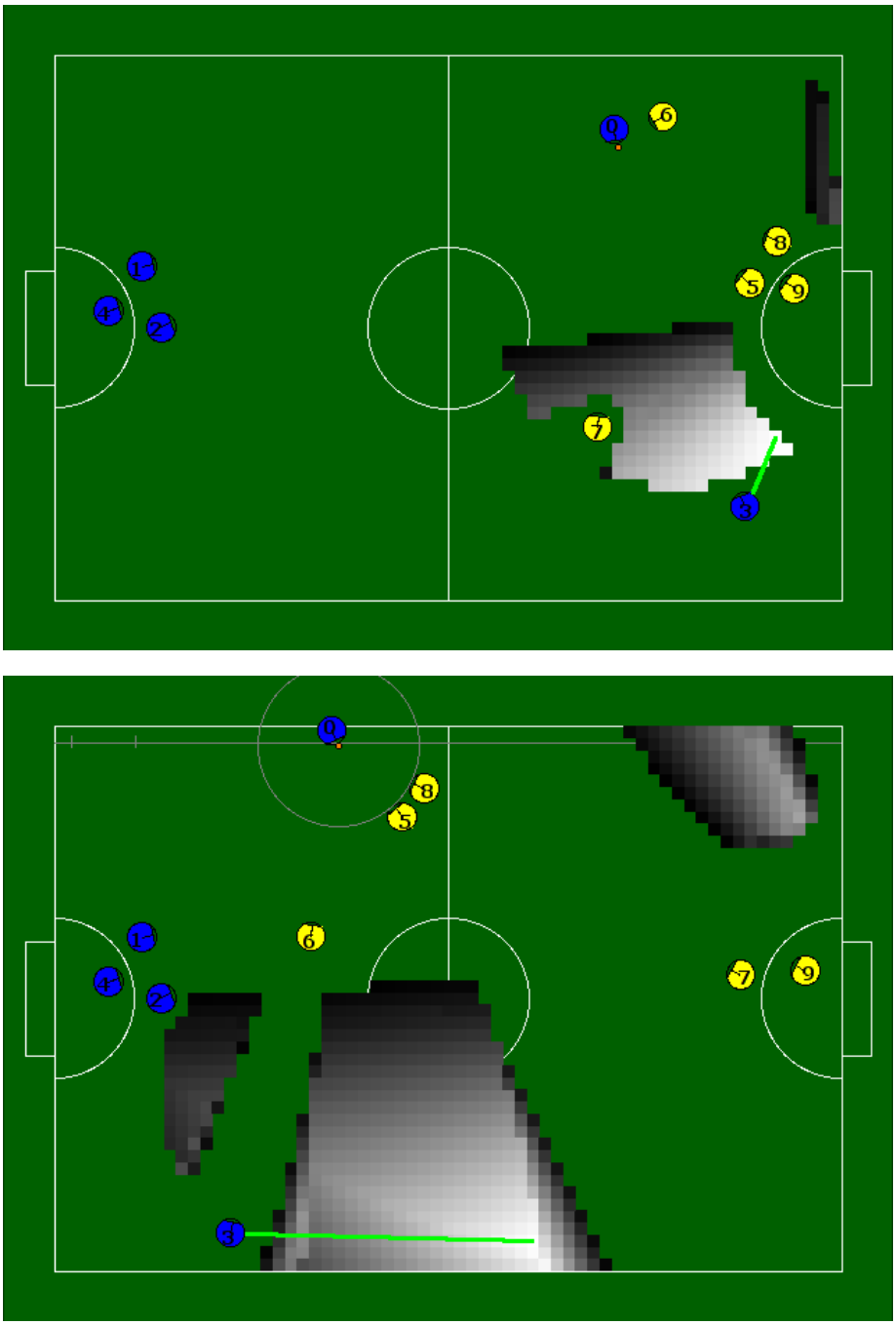


Figure B.8: Two example situations passing situations are shown with, the passing evaluation metric sampled on a regular grid. The values are shown in grayscale where black is zero and the maximum value is white. Values below 0.5% of the maximum are not drawn. The same evaluation function is used in both the short and long pass situations, and the maximum value is indicated by the bold line extending from the supporting robot.

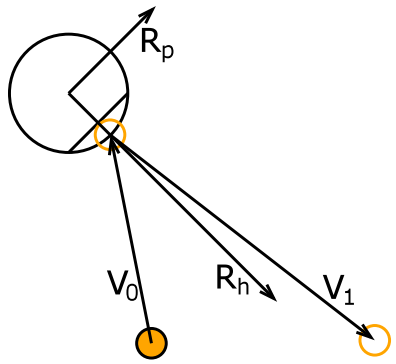


Figure B.9: System model for one-touch ball control. The final velocity of the ball v_1 still contains a component of the initial velocity v_0, rather than running parallel to the robot heading R_h.

B.6 One-touch Ball Control

One significant contribution developed shortly before the 2005 competition was CMRoboDragons' ability to accurately redirect a moving ball. The system we have developed combines our existing ball interception and target selection routines with a method for determining the proper angle to aim the robot to accurately redirect the ball to the target. In order to kick an incoming ball in a different direction without explicitly receiving the ball, the most important necessary new component was a model of how interaction with the kicker will affect the speed of the ball. In particular, while the kicker adds a large forward component to the ball velocity, effects from the ball's original (incoming) velocity are still present and non-negligible.

After measurement and testing of several models, we ended up using a simple linear damping model. The system model is shown in Figure B.9. The initial (incoming) velocity of the ball is denoted as v_0, while the final velocity after kicking is denoted v_1. Normalized heading and perpendicular vectors for the robot are R_h and R_p, respectively. The kicker provides an impulse to the ball in the direction of R_h, propelling a ball initially at rest to speed k (i.e. $\|v_0\| = 0 \rightarrow \|v_1\| = k$). Using this model, we can estimate the final velocity using the following equation:

$$\hat{v_1} = kR_h + \beta(R_p \cdot v_1)R_p \tag{B.5}$$

In our testing, we found values of β ranging from 0.8 with no dribbler present, down to 0.3 for a robot with a dribbler spinning at maximum speed. Of course, simply having a forward

model is not sufficient, as the control problem requires solving for the robot angle given some relative target vector g. However, it is easy to calculate this using a bisection search. The bounding angles for the search are the original incoming ball angle (where the residual velocity component would be zero) and the angle of target vector g (where we would aim for an ideal kicker where $\beta = 0$. The actual solution lies somewhere in between, and we can calculate an error metric e by setting up the equation above as a function of the robot angle α.

$$\begin{aligned} R_h(\alpha) &= \langle \cos\alpha, \sin\alpha \rangle \\ R_p(\alpha) &= \langle -\sin\alpha, \cos\alpha \rangle \\ \hat{v_1}(\alpha) &= kR_h(\alpha) + \beta(R_p(\alpha) \cdot v_1)R_p(\alpha) \\ e(\alpha) &= \hat{v_1}(\alpha) \cdot g \end{aligned}$$

Thus when $e(\alpha) > 0$ the solution lies with α closer to g, while if $e(\alpha) < 0$ the solution is closer to v_0. A solution at the value of α where $e(\alpha) = 0$, so bisection search is simply terminated whenever $\|e(\alpha)\| < \epsilon$. While it is possible to invert many models so that search is not required, using a numerical method for determining α allowed rapid testing of different models, since only the forward calculation needed to be made. Bisection search has proven quite fast in practice since the calculations for a forward model are relatively simple, and only a logarithmic number of evaluations need to be made to achieve the desired accuracy.

Overall, the one touch ball control proved quite useful, allowing our team in 2005 year to score most of its goals via passing, even with a relatively slow kicker $3.75m/s$. In 2006, combined with the kicker capable of $15m/s$ and accuracy improvements allowing passes as speeds ranging from $2m/s$ to $4m/s$, it allowed our team to be quite dangerous on offense. We also adapted the 2D version of ball deflection to the 3D problem of soccer "headers". The chip kicker was used to kick the ball in the air, and a dynamics model of the ball fit a parabolic trajectory to the observed ball position. This allowed us to intercept a ball still in the air to deflect it into a goal. The ground pass deflection and air deflection together greatly contributed to our scoring record and eventual championship.

B.7 Summary and Results

This chapter gives an overview of the CMDragons system, covering both the robot hardware and the overall software architecture of the offboard control system. The hardware has built on the collective experience of our team and continues to advance in ability. The software uses our proven system architecture with many improvements to the individual modules. The CMDragons software system has been used at part of two national and six international RoboCup competitions, placing within the top four teams of the tournament every year since 2003, and finishing 1st in 2006. The results are listed in Table B.7. The competition and the resulting tournament placing are listed, along with the author's contribution to the total team effort for the software system, based on source code and hours spent. The author's contributions to the robot hardware were generally limited to design input and testing.

Competition	Result	%Contribution
RoboCup 2001	RR	30%
RoboCup 2002	QF	35%
US Open 2003	1st	40%
RoboCup 2003	4th	40%
RoboCup 2004	4th [1]	40%
RoboCup 2005	4th [1]	95%
US Open 2006	1st	85%
RoboCup 2006	1st	85%

Table B.1: Results of RoboCup small-size competitions for CMDragons from 2001-2006

[1]Provided software component as part of a joint team with Aichi Prefectural University, called CMRoboDragons

Bibliography

[1] Nancy M. Amato, O. Burchan Bayazit, Lucia K. Dale, Christopher Jones, and Daniel Vallejo. OBPRM: An obstacle-based PRM for 3D workspaces. In *Workshop on the Algorithmic Foundations of Robotics*, 1998.

[2] Nancy M. Amato and Yan Wu. A randomized roadmap method for path and manipulation planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 113–120, April 1998.

[3] Ronald C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research, August 1989*, 8(4):92–112, 1989.

[4] Anna Atramentov and Steven M. LaValle. Efficient nearest neighbor searching for motion planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.

[5] David Ball and Gordon Wyeth. Multi-robot control in highly dynamic, competitive environments. In *7th International Workshop on RoboCup*, volume 7. Lecture Notes in Artificial Intelligence, Springer, 2003.

[6] Kostas E. Bekris, Brian Y. Chen, Andrew M. Ladd, Erion Plaku, and Lydia E. Kavraki. Multiple query probabalistic roadmap planning using single query planning primitives. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS)*, 2003.

[7] Robert Bohlin and Lydia E. Kavraki. Path planning using lazy PRM. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 521–528, 2000.

[8] Robert Bohlin and Lydia E. Kavraki. A lazy probabilistic roadmap planner for single query path planning. *International Journal of Robotics Research*, 2002.

[9] V. Boor, M. H. Overmars, and F. van der Stappen. The gaussian sampling strategy for for probabilistic roadmap planners. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1999.

[10] Michael Bowling, Brett Browning, and Manuela Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of International Conference on Automated Planning and Scheduling (ICAPS'04)*, 2004.

[11] Michael Bowling and Manuela Veloso. Motion control in dynamic multi-robot environments. In *International Symposium on Computational Intelligence in Robotics and Automation (CIRA'99)*, November 1999.

[12] Oliver Brock and Oussama Khatib. High-speed navigation using the global dynamic window approach. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1999.

[13] Carla E. Brodley and P. E. Utgoff. Multivariate decision trees. *Machine Learning*, 19(1):45–77, 1995.

[14] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[15] T. A. Brown and J. Koplowitz. The weighted nearest neighbor rule for class dependent sample sizes. *IEEE Transactions on Information Theory*, 25:617–619, 1979.

[16] Brett Browning, James R. Bruce, Michael Bowling, and Manuela Veloso. STP: Skills tactics and plans for multi-robot control in adversarial environments. In *Journal of System and Control Engineering*, 2005.

[17] James Bruce. CMVision realtime color vision system. The CORAL Group's Color Machine Vision Project. `http://www.cs.cmu.edu/~jbruce/cmvision/`.

[18] James Bruce, Tucker Balch, and Manuela Veloso. Fast color image segmentation for interactive robots. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS)*, Japan, 2000.

[19] James Bruce, Michael Bowling, Brett Browning, and Manuela Veloso. Multi-robot team response to a multi-robot opponent team. *Proceedings of IROS 2002 Workshop on Cooperative Robotics*, 2002.

[20] James Bruce and Manuela Veloso. Fast and accurate vision-based pattern detection and identification. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Taiwan, May 2003.

[21] James R. Bruce. Real-time robot motion planning in dynamic environments: Supplemental materials (http://www.cs.cmu.edu/ jbruce/thesis/).

[22] James R. Bruce, Michael Bowling, Brett Browning, and Manuela Veloso. Multi-robot team response to a multi-robot opponent team. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Taiwan, May 2003.

[23] James R. Bruce and Manuela Veloso. Real-time randomized path planning for robot navigation. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS)*, 2002.

[24] James R. Bruce and Manuela Veloso. Safe multi-robot navigation within dynamics constraints. *Proceedings of the IEEE, Special Issue on Multi-Robot Systems*, 2006, to appear.

[25] J. Chestnutt, J. J. Kuffner, K. Nishiwaki, and S. Kagami. Planning biped navigation strategies in complex environments. In *Proc. IEEE Int. Conf. on Humanoid Robotics*, October 2003.

[26] Youngkwan Cho, Jongweon Lee, and Ulrich Neumann. Multi-ring color fiducial systems and a detection method for scalable fiducial tracking augmented reality. In *Proceedings of IEEE International Workshop on Augmented Reality*, November 1998.

[27] Youngkwan Cho, Jun Park, and Ulrich Neumann. Fast color fiducial detection and dynamic workspace extension in video see-through self-tracking augmented reality. In *Proceedings of the Fifth Pacific Conference on Computer Graphics and Applications*, pages 168–166, October 1997.

[28] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*. ACM Press New York, NY, USA, 1995.

[29] A. D'Angelo, E. Menegatti, and E. Pagello. How a cooperative behavior can emerge from a robot team. In *Proceedings of the 7th International Symposium on Distributed Autonomous Robotic Systems*, 2004.

[30] H. Edelsbrunner. A new approach to rectangle intersections: Part 1. *International Journal of Computation Mathematics*, 13(3):209–219, 1983.

[31] H. Edelsbrunner. A new approach to rectangle intersections: Part 2. *International Journal of Computation Mathematics*, 13(3):221–229, 1983.

[32] Michael Erdmann and Thomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2(4):477–521, 1987.

[33] David Ferguson, Nidhi Kalra, and Anthony Stentz. Replanning with RRTs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[34] David Ferguson and Anthony Stentz. Field D*: An interpolation-based path planner and replanner. In *Proceedings of the International Symposium on Robotics Research (ISRR)*, 2005.

[35] David Ferguson and Anthony Stentz. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, February 2006.

[36] P. Fiorini and Z. Shiller. Motion planning in dynamic environments using the relative velocity paradigm. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 560–565, May 1993.

[37] P. Fiorini and Z. Shiller. Motion planning in dynamic environments using velocity obstacles. *International Journal of Robotics Research*, 17(7):760–772, July 1998.

[38] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics and Automation Magazine*, 4, March 1997.

[39] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. *ACM SIGGRAPH Computer Graphics*, 14(3):124–133, 1980.

[40] M. Fujita and K. Kageyama. An open architecture for robot entertainment. In *Proceedings of the first international conference on Autonomous agents*, pages 435–442. ACM Press New York, NY, USA, 1997.

[41] M. Fujita, Y. Kuroki, T. Ishida, and T. T. Doi. A small humanoid robot sdr-4x for entertainment applications. *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, 2, 2003.

[42] Masahiro Fujita, Y. Kuroki, T. Ishida, and T. Doi. A small humanoid robot sdr-4x for entertainment applications. In *Proc. of the Int. Conf. on Advanced Intelligent Mechatronics (AIM)*, 2003.

[43] B. P. Gerkey and M. J. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9):939–954, 2004.

[44] Janne Heikkila. Geometric camera calibration using circular control points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22:1066–1077, 2000.

[45] Shinya Hibino, Yukiharu Kodama, Yasunori Nagasaka, Tomoichi Takahashi, Kazuhito Murakami, and Tadashi Naruse. Fast image processing and flexible path generation system for robocup small size leauge. In *Proceedings of the RoboCup-2002 Symposium*, 2002.

[46] D. Hsu, R. Kindel, J.C. Latombe, and S. Rock. Randomized kinodynamic motion planning with moving obstacles. *International Journal of Robotics Research*, 21(3):233–255, 2002.

[47] David Hsu, Tingting Jiang, John Reif, and Zheng Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2003.

[48] David Hsu, J. C. Latombe, and R. Motwani. Path planning in expansive configuration spaces. In *Proceedings of The IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, 1997.

[49] Pekka Isto. Constructing probabilistic roadmaps with powerful local planning and path optimization. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS)*, pages 2323–2328, 2002.

[50] Ramesh Jain, Rangachar Kasturi, and Brian G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

[51] Jr. James J. Kuffner and Steven M. LaValle. RRT-Connect: An efficient approach to single-query path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2000.

[52] Tamás Kalmár-Nagy, Raffaello D'Andrea, and Pritam Ganguly. Near-optimal dynamic trajectory generation and control of an omnidirectional vehicle. *Robotics and Autonomous Systems*, 46(1):47–64, 2004.

[53] Lydia E. Kavraki and Jean-Claude Latombe. Randomized preprocessing of configuration space for fast path planning. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 2138–2145, 1994.

[54] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabalistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Transactions on Robotics and Automation*, volume 12, pages 566–580, 1996.

[55] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/ALife*, 1995.

[56] D. E. Koditschek. Exact robot navigation by means of potential functions: Some topological considerations. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1–6, 1987.

[57] Sven Koenig and Maxim Likhachev. Incremental A*. *Advances in Neural Information Processing Systems (NIPS)*, 2002.

[58] Y. Koren and J. Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1398–1404, April 1991.

[59] Andrew M. Ladd and Lydia E. Kavraki. Fast tree-based exploration of state space for robots with dynamics. In *Algorithmic Foundations of Robotics VI*, pages 297–312. Springer, STAR 17, 2005.

[60] Andrew M. Ladd and Lydia E. Kavraki. Motion planning in the presence of drift, underactuation and discrete system changes. In *Robotics: Science and Systems I*, pages 233–241, Boston, MA, June 2005. MIT Press.

[61] F. Large, Z. Shiller, S. Sekhavat, and C. Laugier. Towards real-time global motion planning in a dynamic environment using the nlvo concept. In *Proceedings of the IEEE Conference on Intelligent Robots and Systems (IROS)*, October 2002.

[62] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer, 1991.

[63] T. Laue and T. Rofer. A behavior architecture for autonomous mobile robots based on potential fields. In *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276, pages 122–133. Lecture Notes in Artificial Intelligence, Springer, 2005.

[64] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. In *Technical Report No. 98-11*, October 1998.

[65] Steven M. LaValle and Jr. James J. Kuffner. Randomized kinodynamic planning. In *International Journal of Robotics Research, Vol. 20, No. 5*, pages 378–400, May 2001.

[66] Tsai-Yen Li and Yang-Chuan Shie. An incremental learning approach to motion planning with roadmap management. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 4, 2002.

[67] Tsai-Yen Li and Yang-Chuan Shie. An incremental learning approach to motion planning with roadmap management. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 2002.

[68] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, 1998.

[69] T. Lozano-Perez and M. A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. In *Communications of the ACM*, volume 22, pages 560–570, 1979.

[70] Kazuhito Murakami, Shinkya Hibino, Yukiharu Kodama, Tomoyuki Iida, Kyosuke Kato, and Tadachi Naruse. Cooperative soccer play by real small-size robots. In *Proceedings of the 2003 RoboCup Symposium*, 2003.

[71] Bruce Naylor, John Amanatides, and William Thibault. Merging BSP trees yields polyhedral set operations. *ACM SIGGRAPH Computer Graphics*, 24(4):115–124, 1990.

[72] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, 1980.

[73] J. M. Phillips, N. Bedrosian, and L. E. Kavraki. Guided expansive spaces trees: A search strategy for motion and cost-constrained state spaces. In *Proceedings of The IEEE International Conference on Robotics and Automation (ICRA)*, pages 3968–3973, New Orleans, LA, April 2004. IEEE Press.

[74] Charles Poynton. Poynton's color FAQ, 1997. http://www.inforamp.net/~poynton/ColorFAQ.htm.

[75] S. Quinlan and O. Khatib. Elastic Bands: Connecting Path Planning and Control. *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*, pages 802–807, 1993.

[76] J. H. Reif. Complexity of the mover's problem and generalizations. In *Proc. of the 20th IEEE Symp. on Foundations of Computer Science*, pages 421–427, 1979.

[77] Michael Sherback, Oliver Purwin, and Raffaello D'Andrea. Real-time motion planning and control in the 2005 cornell robocup system. *Robot Motion and Control*, 335(1):245–263, 2006.

[78] Mark Simon, Sven Behnke, and Raul Rojas. Robust real time color tracking. In *Lecture Notes in Artificial Intelligence 2019, RoboCup 2000: Robot Soccer World Cup IV*, 2001.

[79] RoboCup small-size Technical Committee. RoboCup F180 league 2006 rules (http://www.cs.cmu.edu/ brettb/robocup/rules/).

[80] Anthony Stentz. Optimal and efficient path planning for unknown and dynamic environments. In *International Journal of Robotics and Automation, Vol. 10, No. 3*, 1995.

[81] P. Svestka and M. H. Overmars. Motion planning for car-like robots, a probabilistic learning approach. *International Journal of Robotics Research*, 8:119–143, 1997.

[82] Robert E. Tarjan. *Data structures and network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[83] A. Tews and G. Wyeth. MAPS: A system for multi-agent coordination. *Advanced Robotics*, 2000.

[84] R. B. Tilove. Local obstacle avoidance for mobile robots based on the method of artificial potentials. *General Motors Research Laboratories, Research Publication GMR-6650*, September 1989.

[85] E. Uchibe, T. Kato, M. Asada, and K. Hosoda. Dynamic task assignment in a multiagent/multitask environment based on module conflict resolution. In *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 3987–3992, 2001.

[86] Manuela Veloso, Michael Bowling, Sorin Achim, Kwun Han, and Peter Stone. The CMUnited-98 champion small robot team. In *RoboCup-98: Robot Soccer World Cup II.* Springer Verlag, 1999.

[87] CW Warren. Multiple robot path coordination using artificial potential fields. *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 500–505, 1990.

[88] T. Weigel, J. S. Gutmann, M. Dietl, A. Kleiner, and B. Nebel. CS-Freiburg: Coordinating robots for successful soccer playing. *IEEE Transactions on Robotics and Automation*, 18:685–699, 2002.