BOUNDEDLY RATIONAL UTILITARIAN VOTING WITH OVERRIDES: AN ARCHITECTURE FOR AUTONOMOUS MOBILE ROBOT CONTROL

by

Thomas J. Palmer

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

July 2001

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Thomas J. Palmer

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Michael A. Goodrich, Chair
Date	Tony R. Martinez
Date	Bill R. Hays

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Thomas J. Palmer in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Michael A. Goodrich
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

Earl M. Woolley
Dean, College of Physical and Mathematical Sciences

ABSTRACT

BOUNDEDLY RATIONAL UTILITARIAN VOTING WITH OVERRIDES: AN ARCHITECTURE FOR AUTONOMOUS MOBILE ROBOT CONTROL

Thomas J. Palmer

Department of Computer Science

Master of Science

For the past fifteen or twenty years, a popular method for designing robot control systems has been to split the decision making into modular behaviors, each responsible for different objectives of the system. This method is generally referred to as behavior-based control. One popular behavior-based control paradigm is *utilitarian voting* wherein each behavior assigns a utility to candidate actions. These utilities are averaged or combined in some other way, and the action with the highest utility is taken. Such voting systems promote modularity and have been shown to be practical in mobile robots.

Difficulties arise, however, when trying to apply utilitarian voting to systems with very high resolution or many degrees of freedom. When there are too many actions, finding the action with the highest utility in real-time can become impossible. Extensive searches can also exhaust CPU resources. We address these problems in two ways:

1. The search for the best action can be cast as an anytime algorithm. This allows the use of directed searches that find actions with high utility quickly. The best action found so far can be used at any time.

2. An aspiration level can be kept which represents the utility achieved during recent, previous action searches. This aspiration level can be used as a performance monitor and a way for ending searches early.

Using limited searches in this way results in an inability to guarantee that the optimal action is taken. In general, it can be impossible to predict what action will be taken at any time step. To constrain the negative effects of unanticipated actions, we introduce the use of override behaviors that can either take direct control in dangerous circumstances or veto actions which should not be taken. Such override behaviors are modular and are therefore keeping in spirit with the behaviors normally used in utilitarian voting.

Within this framework of bounded search capability and constraints provided by overrides, we present a robot control architecture called Bruvo 1 (Boundedly Rational Utilitarian Voting with Overrides, version 1). We use this architecture in the control of mobile robots for simple wandering and goal-seeking tasks. We present empirical evidence from simulated and real-world tests that shows the practicality of our approach. In our test cases we specifically show a decrease in CPU usage over traditional techniques of up to 42% while increasing the time required to perform a sample task by only 6%. We also show that extensions to our technique may be able to improve CPU usage by an additional order of magnitude. Such results show that it is possible to use voting for full control of complicated robot control tasks.

ACKNOWLEDGMENTS

More than any other person, I thank my wife, Kristine, for supporting me in finishing this thesis. She helped me be motivated when I had trouble on my own, and she let me spend long hours working. I hope to avoid being so busy in the future. Our two little boys, Jacob and David, also had to put up with a lot.

I thank my parents and other family members for helping me get to where I am. Without their sacrifices, I would not have been in a position to start any such endeavor.

I also thank my advisor, Mike Goodrich, for his time and effort. He has had to put up with my frustrations more than once. Other professors at BYU deserve similar credit, though they haven't had to deal with me quite as much.

BYU itself, Motorola, Nissan, and the family of Edwin Smith Hinckley have helped fund my education over the years in the form of scholarships and other financial aid. I thank them for the help.

Also worth mentioning are the authors of the tools I used for writing this thesis. I made extensive use of systems such as GNU/Linux, Python, LaTeX, Vim, Ghostscript, and many others. These are given freely by their authors, and I am grateful for their willingness to help other people.

Last, but not least, I think it's worth remembering that all good comes from God. If there's anything valuable in this document, I feel that I should recognize the true source.

Contents

Chapter 1

Introduction

1.1 Mobile Robot Control and Utilitarian Voting

Mobile robot control is a popular research area, not only because of its practical applications, but also because it is a useful method for creating and studying decision processes. This thesis will focus primarily on a decision-making model, but it will also provide a complete implementation of a robot control architecture and a simple application of that architecture.

Multiple difficulties exist in the field of autonomous robotics for which any control architecture should account. These difficulties arise from the natural complexities of the real world, as in the following examples modified from [17]:

1. Unpredictable and incompletely known environment.

2. Imperfect sensors and actuators.

3. Limited time.

4. Multiple, concurrent objectives.

Many of these concerns are handled well by a decision process and robot control methodology known as *voting* or, to be more precise, *utilitarian voting*. In this methodology, each objective of the robot is handled by one or more *behaviors* that assign utilities to actions that may be performed by the robot. The utilities are summed for each action across all behaviors, higher utilities indicating more preferable actions. Figure 1.1 shows an example of two behaviors that want to reach separate goals.

Utilitarian voting is simple, allows heterogeneous behaviors to contribute simultaneously to action selection with minimal communication complexity, and allows all potential actions to be compared to one another. The only data types used for input to and output from the behaviors are states, actions, and utilities. This means each behavior is a black box whose internal workings may be implemented by any suitable algorithm.

This thesis presents a way of using utilitarian voting for mobile robot control when the number of actions to choose between is too large to be exhaustively searched in real-time[1]. A full architecture is described for handling such situations, and the architecture's implementation on a Nomad Super Scout mobile robot for simple wander and goal-seeking purposes is presented. Such tasks (a) exhibit enough complexity to be interesting problems, (b) serve as fundamental skills in an autonomous control system, and (c) allow us to compare our architecture against other voting schemes.

The remainder of the introduction will continue to address voting and the difficulties that arise when there are too many actions to exhaustively search and compare in real-time.

[1]In the context of this thesis, we use *real-time* to mean that the robot must operate while interacting with the real world and do not necessarily imply the concepts associated with formal real-time computation.

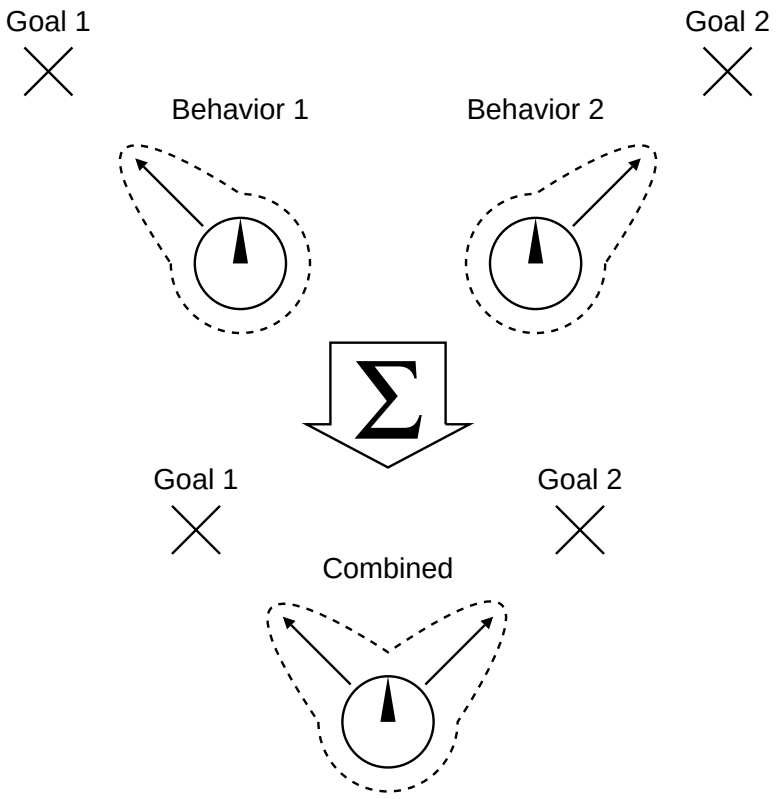

Figure 1.1: *An example of utilitarian voting, showing how the votes of behaviors are combined. The circles with triangles inside represent the robot. Two behaviors' votes are shown independently at the top, and their combination is shown beneath them. The dashed lines represent the utility assigned to each orientation: the further from the robot, the higher the utility. The solid arrows represent optimal actions that the robot could choose to perform.*

1.2 Related Literature: Seminal Work

In this section, we briefly review the literature on utilitarian voting. Further discussions of related literature are presented in sections 2.2 and 3.2.

Utilitarian voting for robot control originally derived from the general field of

behavior-based robotics. Notable originators of this field include Rodney Brooks [4] and Ronald Arkin [1]. In general, behavior-based control rejects the centralized, high-level reasoning of traditional AI as a fundamental construct for intelligence [5]. Instead, the focus is on bottom-up intelligence, based largely on modular, reactive behaviors. Real-world implementation is another vital aspect of this field of robotics.

Within the specific field of behavior-based voting, the most notable work is Julio K. Rosenblatt's Distributed Architecture for Mobile Navigation [21]. This work began in the late 1980s when Rosenblatt had difficulty adapting other architectures such that they worked sufficiently well for his tasks [23]. His architecture consists of asynchronous behaviors submitting votes to a central arbiter. The asynchrony allows for behaviors with lengthy processing to not hold up the voting speed of faster behaviors. Voting is performed independently for each degree of freedom. For instance, velocity is voted on independently from steering. Rosenblatt's architecture has been used primarily for control of outdoor ground vehicles, but it has been recently used for underwater vehicles [22] as well.

Utilitarian voting systems, including explicit extensions of Rosenblatt's architecture, have been used by a number of other researchers [6, 14, 17, 19, 25, 26]. Most use simple utility summations while others use more elaborate combinations. For instance, Roland Stenzel [26] argues the use of a T-norm over summation (or equivalently, an arithmetic mean). Many voting architectures have been built upon each other, but the main concept is intuitive and obvious enough to be used as a default in many systems. In our opinion, that is one of its main advantages.

Among utilitarian voting research, the work of Paolo Pirjanian deserves specific mention. In Pirjanian's work, utilitarian voting is cast into a more formal multi-objective decision framework [17]. Formal analysis of this type provides more theoretical soundness to voting systems. As a part of his discussion, he has shown how

Rosenblatt's architecture produces results that are Pareto optimal with respect to the objectives represented by the behaviors.

Both Rosenblatt and Pirjanian have extensively discussed various advantages of voting over other robot control architectures [17, 21].

1.3 The Problem of Large Action Spaces

Any system that assigns utilities to choices must be able to search through those choices to find the ones of highest utility. In utilitarian voting systems, little work has been done to address issues associated with searching through large action spaces. Large action spaces occur when there are many dependent degrees of freedom or when an extremely high resolution of control is required. Such large action spaces cannot be searched in real-time, and a decision maker addressing such a problem cannot find an optimal solution but is instead *boundedly rational*.

A significant related problem is that the deliberation process should not dominate CPU usage because the behaviors may need to perform background computation on separate execution threads. For instance, robots with cameras and vision processing requirements may need to perform extensive computation in addition to simply deciding actions. Searching a large action space without care can easily make such processing impossible.

All the architectures cited in the previous section assume the ability to search exhaustively through all possible actions at any time step. Rosenblatt's architecture, with its asynchronous voting, also assumes that all votes from all behaviors can be stored in memory simultaneously. Even virtual memory capabilities are insufficient for very large numbers of actions.

The ability to search exhaustively is usually a result of either assuming inde-

pendence between degrees of freedom (or *action dimensions*) or of limiting the voting resolution to a coarser level than that available to the actuators. When action dimensions are *not* independent, ad hoc fixes are often supplied. For instance, Rosenblatt scales velocity based on the chosen turning angle.

We address the issue of large action spaces in a more direct fashion by the use of *anytime algorithms* [7, 28]. Anytime algorithms are searches that can provide a valid answer at any time once the computation has started. The longer they run, the better the answer they can give. By searching the action space with anytime algorithms, large action spaces can be addressed directly.

In addition to whatever fixed time limits exist on decision making, processing time can be saved and standards of results can be maintained by the use of a technique known as *satisficing*. When satisficing, a decision maker does not attempt to optimize but simply to reach an *aspiration level* [24]. The aspiration level can adapt according to the utility of actions taken in the past [13].

Chapter 2 will discuss the issues of large action spaces, anytime search, and satisficing in more detail.

1.4 The Problem of Undesirable Emergence

It can be difficult to predict the actions that a voting system will take in all circumstances. Unpredictable activity resulting from the decision system of a robot is called *emergent behavior* or, simply, *emergence*. Such emergence is a well-known feature of systems built from modular behaviors [5]. At times, emergent behavior is better than expected. At other times, emergence can be very undesired. When a robot is at the edge of a cliff, it is important that a "wander off" behavior not emerge.

Action spaces large enough to prohibit exhaustive searches make prediction even

harder. Even if the entire multidimensional state/action/utility space can be properly visualized or analyzed, there is no guarantee that the optimal action will be chosen. It is therefore very important that if designer-specified constraints exist for a system, then some mechanism must exist to allow these constraints to be met.

Within voting, a previously used though not highly publicized technique for constraining undesirable emergence is the use of *vetoes* [6, 25]. When a behavior vetoes, it states that a particular action cannot be taken no matter what other behaviors say. Vetoes are practical because they can be used to guarantee fault avoidance and also because they are modular in nature. That is, just as voting behaviors can be added and removed from a system, vetoing behaviors can be as well. A vetoer provides an explicit, modular component for enforcing a constraint.

While vetoes are practical, they are not sufficient. If a robot is stopped near a wall, a veto to prevent moving into the wall may be useful, but if, for some emergent reason, the robot is moving toward the wall at full speed, there may be no time for deliberation. In such a case, a fault-avoiding behavior may *hijack* the robot, cause a full stop, then allow the normal decision process to continue. As hijacks prevent multiple behaviors from contributing simultaneously, they should be used *very* rarely. In general, the designer of the system should be aware of the hijacks and vetoes used by all fault-avoiding behaviors in the system.

Chapter 3 will discuss emergence, designer constraints, and hijack and veto override mechanisms in more detail.

1.5 Thesis Statement

In this thesis, we focus on the issues highlighted in the following thesis statement:

Behavior-based utilitarian voting systems for real-time mobile robot control can be extended with anytime searches and aspiration maintenance in order to directly address large, multidimensional action spaces without exhausting computational resources. Decentralized, encapsulated override mechanisms, such as hijacks and vetoes, can be used to effectively constrain undesirable emergence. In simulated tests, these techniques compare favorably against other utilitarian methods. Validation using real-world robot experiments also show the practicality of the approach.

In chapter 6, we show for our test application that we are able to decrease CPU usage by up to 42% while increasing the time needed to complete the required task by only 6% when comparing our satisficing approach to traditional exhaustive search techniques. We do this while accounting for dependencies between action dimensions. We also show that directed search techniques such as genetic algorithms show promise for significant further decrease in CPU usage over results achieved so far.

1.6 Direction of This Thesis

Chapters 2 and 3 discuss the issues of large action spaces and emergence, respectively, in more detail. Both chapters give additional related literature on their respective topics. Furthermore, justification is given to the direction taken by the architecture presented in this thesis. Chapter 4 discusses the software architecture that implements the solutions proposed in chapters 2 and 3. Chapter 5 discusses the implementation of the architecture and the behaviors used for control of Nomad Super Scout robots. Chapter 6 discusses the tests performed to validate the architecture, gives the results of those tests, and analyzes the results. The conclusion of the thesis follows in chapter 7.

Chapter 2

Large Action Spaces

2.1 Problem Overview

As discussed in chapter 1, actions spaces too large to search in real-time can result from multiple dependent degrees of freedom or very high resolution of control. There are specifically two problems that need to be addressed in relation to this issue:

1. Large action spaces cannot be search exhaustively in real-time.

2. Even searching a limited amount of a large space *can* exhaust computational resources.

The remainder of this chapter discusses the prior work related to this issue, analyzes the problem in more detail, and presents the use of anytime search and satisficing as a solution to the two problems listed above.

2.2 Related Literature

2.2.1 Current Voting Limitations

In general, all the voting architectures cited in the introduction assume the ability to choose the action that maximizes utility. Architectures that use asynchronous voting [21, 26] generally assume that all behavior votes can fit in memory simultaneously. These assumptions require that the total number of possible actions be small or that some high-level analysis of the action space is possible. Most applications of the cited architectures use voting to control just a single degree of freedom, often orientation.

As discussed in section 1.2, Rosenblatt's architecture performs voting for each action dimension independently [21]. Speed and steering, for instance, are decided independently, but these are obviously dependent at times. To address this, Rosenblatt applies ad hoc fixes unrelated to behavior-based voting. Speed is scaled according to the turning angle. Sharper turns reduce the speed more than shallow turns. No voting is involved in this decision. Rosenblatt has also considered implementation of feedback loops between different voting arbiters [21, page 143], but that still does not allow for behaviors to see the full action vector that is being chosen.

For processing speed concerns, Rosenblatt also has behaviors vote at a resolution lower than that provided by the actuators. For instance, in deciding vehicle turning commands, 51 possible actions were voted on [21, page 65]. In his architecture, when a low resolution is used, interpolation is performed on the resulting action utilities. This provides an effectively higher resolution of control, but the behaviors are still unable to vote on the action actually taken.

Pirjanian's work represents full-resolution actions as the cross product of translational and rotational velocities, resulting in an action space of some 150-200 thousand actions [17]. This is a far larger action space than other architecture applications we

have encountered, but it still is exhaustively searchable in real-time if the behaviors vote quickly. Additional action dimensions resulting from behavior control of more actuators, such as with active sensing, would eliminate the possibility of performing an exhaustive search.

2.2.2 Satisficing and Anytime Algorithms

Bounded rationality is a field that studies the process of decision making under limited resources. In the mid-1950s, the economist Herbert Simon began formally arguing against traditional rationality on the basis that it could not adequately describe human behavior [24]. He described what is now commonly known as *satisficing*, wherein decision makers attempt to achieve an *aspiration level* rather than optimize. In the case of natural agents (such as humans and animals), proponents of bounded rationality argue that optimization is rarely even a consideration [9, 10, 12, 15, 24]. That is, ecological conditions have led decision makers to use methods that are not directly related to classical rationality in any way. They simply get the job done.

Various formalisms of bounded rationality and satisficing exist [12, 13, 18, 27]. Most models are based on some form of utility metrics, while others avoid such issues as much as possible [9]. Satisficing has also been used in utilitarian voting robot control systems [17, 18]. In his methods, however, Pirjanian requires the ability to exhaustively search the action space. Satisficing, in his work, is mostly used as a way of testing that aspiration levels are met[1].

One satisficing formalism of note is that of Karandikar et al. who use a simple aspiration level between 0 and 1 that varies with the utility of taken actions [13]. Utilities are also constrained within the same interval. Whenever the current utility

[1]When the best is not good enough, meeting an aspiration level is a stricter requirement than optimization.

of the last action taken is below the aspiration level, a new action must be chosen. If the new utility is above the aspiration level, the chance of changing is less than 1 and drops off in a sigmoidal fashion. In their research, they have applied their method to studies of cooperation in simple two-action, two-player games such as the Prisoner's Dilemma, but the method can be extended to any utilitarian system.

For handling large action spaces, some technique must be used to find actions that are hopefully satisficing. *Anytime algorithms* are particularly well-suited to real-time control where exhaustive searches are not possible [7, 28]. Such algorithms may be stopped at any time yielding the best results obtained so far. Thomas Dean and Mark Boddy first introduced anytime algorithms in the 1980s primarily as a means for robot planning. They also introduced the concept of *performance profiles*, which plot the utility of solutions over time for a given algorithm [7]. Shlomo Zilberstein followed them as a champion of anytime algorithms and has done additional analysis including extensions to performance profile concepts [28].

Zilberstein has also discussed the relationship between satisficing and anytime algorithms [29]. He defines the term *bounded optimality* as "the selection of the best action subject to an arbitrary set of architectural constraints" and argues its value over informal satisficing algorithms based on heuristics. Within this context, anytime algorithms provide a means of achieving bounded optimality, and bounded optimality itself is a form of satisficing. On the other hand, too much metalevel reasoning can also be impractical [10, chapter 1].

2.3 Motivation for Large Action Spaces

2.3.1 Action Dimension Dependencies

Voting on separate action dimensions independently allows maximization along individual dimensions but not optimization of the final multidimensional action as a whole. Higher-order correlations between action dimensions cannot be addressed by the behaviors. This difficulty can be somewhat addressed by careful human planning. That is, the designer of a system can try to factor the action dimensions into mostly distinct concepts. Translational and rotational velocities are mostly distinct, and this is a common factorization. At the same time, they are not independent. For instance, turning a car sharply at high speeds can cause a flip. As another example, on a highway, it may make sense to slow down at times or to change lanes, but when another car is not far behind in the other lane, slowing down *and* changing lanes may be a bad idea.

Designers may impose rules independent of the voting behaviors to address such needs. As discussed earlier, Rosenblatt applies a rule after voting is finished that scales speed down for sharper turns. This does address *a* dependency between the two dimensions. Addressing more objectives and more dependencies would benefit from a voting process in the same way that voting helps simplify decision making within single dimensions. At this point, the full, combined action space would still need to be searched. In other words, for the benefits of voting to be available at all levels of control, dimensional dependence needs to be addressed.

In the general field of behavior-based control, behaviors are experts of the domain for which they are responsible. In the words of Rodney Brooks, "Each module merely does its thing as best it can" [4]. Spatial transformations may be necessary to best address a particular need. This is a common procedure in pattern recognition and

data mining fields. Behaviors need full multidimensional information to be able to apply any algorithm. Otherwise, there are implicit limitations in the system, and posterior fixes must be applied in a less systematic fashion.

Also worth noting is the exponential growth characteristic of adding action dimensions. For some applications, it may be possible to exhaustively search the full action space, but adding even one more degree of freedom can require significantly faster processing. Any system that may require additional degrees of freedom in the future cannot expect faster CPUs to solve the search problem.

2.3.2 High Resolution

For most practical situations, it is unnecessary to have an action space of higher resolution than the actuators themselves allow. Still, in many applications, even working at the resolution of the actuators or addressing the full range of choices leads to very large action spaces, especially in multidimensional settings. Consequently, designers who account for multiple dimensions may restrict resolution or bounds. For instance, Pirjanian did not include negative translational velocities. His decision may have been necessary for voting to be performed fast enough, but it also restricted what behaviors could have been implemented (and consequently, what behaviors might have emerged).

As another example, in the application presented in chapter 5 of this thesis, one actuator setting is the sonar firing rate, which controls how often the robot perceives distances to obstacles. In all, only two or three different firing rates are usually used, whereas some 255 values are possible. Voting on the full range of possibilities allows finer control, but also significantly increases the size of the action space. The reason for voting on the entire space is to avoid limiting what behaviors may be implemented.

Limiting the resolution or size of a dimension results in a loss of options as to what objectives may be handled.

Additionally, in an implementation where actuators are handled in a more abstract fashion, the resolution of the actuators may not be known to the behaviors or the voting procedure. A behavior need not know the full resolution provided that it can return an assessment of any action. While we have not yet implemented such an abstracted system, it is clear that any assumptions made on the resolution of unknown actuators may limit performance. In such a case, a continuous action space may be best. The architecture we present allows for such continuous spaces.

In general, reduced resolution may allow for optimization over a limited set of possibilities, but the resulting decision may not be optimal over the set of *all* available actions. Looping through the low-resolution actions is a search process, but it is an *undirected* search. That is, it does not try to focus on the areas where the best actions are most likely to be found. A directed search would allow higher resolution where it most likely matters, and less time would be spent unproductively.

Another benefit of high resolution is the reduced dependence on interpolation to find ideal actions. This is important because interpolation without evaluation (by behaviors) may, in extreme cases, lead to worse results than the original, more discrete choice.

2.4 Aspiration-Based Anytime Search

In order to address large action spaces without exhausting computational resources we present the use of anytime algorithms in conjunction with aspiration-based satisficing. Specifically, an anytime search can be used to find the best actions possible within time constraints. When the time limit expires, the best action found so far is

performed. However, performing intensive computation during the entire search time leaves little CPU time for background processes. Because of this, an aspiration level is maintained that approximates the utility of recently taken actions and therefore the expected utility of useful, near-term actions. If an action is found that meets or exceeds the aspiration, it is a satisficing action and may be taken without further search. The remaining search time may be spent sleeping, thereby allowing CPU resources to be used by other threads or processes. Figure 2.1 depicts how aspirations and anytime searches fit together.

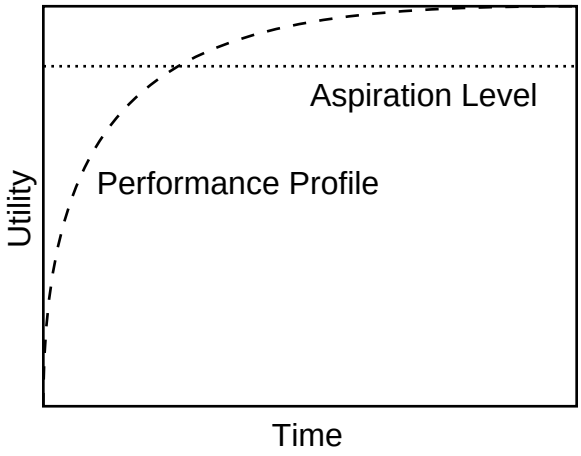

Figure 2.1: *Aspiration-based anytime search. As time increases, the utility of the best solution found increases. The aspiration level can be used as a cutoff resulting in a shorter search which still meets current expectations.*

For updating the aspiration level when an action is taken, we use a modified form of the equation in [13]:

$$aspiration_{next} = (1 - change\ rate) \times aspiration + change\ rate \times utility_{action} \quad (2.1)$$

The change rate affects how quickly the aspiration adapts and is a value in the set $[0, 1]$. For a change rate of 0, the aspiration is constant, and for a change rate of 1,

the utility of a taken action is always the aspiration for the next time step.

We have chosen a simple method for deciding when searches are performed. In all cases, if the action chosen at a previous time is still satisficing, the action may be taken without further search. When the previous action is not satisficing, however, a search must be performed (instead of simply switching the action as in the Prisoner's Dilemma application in [13]). When a satisficing action is found, it may be taken, but the aspiration level is unlikely to be raised if no effort is ever put toward finding even better actions. Because of this, when searching, it is better to have a method whereby the aspiration level must be exceeded by some amount. This thesis compares three separate search-ending strategies which are discussed in section 5.5.

2.5 Affected Issues

2.5.1 Synchronous vs. Asynchronous

Inherent in the searching of large action spaces is the inability to store all behavior votes in memory. Asynchronous voting schemes, as introduced by Rosenblatt and continued by others, rely on the ability to express votes for all actions and send them to an arbiter to be stored until further votes are sent. This obviously cannot be done for the anytime search method described in this chapter.

There are apparent benefits to asynchronous voting. Rosenblatt states, "Allowing the modules in a distributed architecture to operate asynchronously, each at the greatest rate of which they are capable, maximizes throughput and therefore reactivity" [21, page 53]. In the words of Pirjanian, asynchrony "allows the fast reactive system components to react in real-time whereas the slow planning components can run at a slower [pace]" [17, page 55]. It is true that among heterogeneous behaviors,

some may compute more slowly than others. This should be accounted for.

In our model, utility calculations must be performed quickly and synchronously to perform an anytime search on large action spaces. Therefore, excessively slow processing must be performed apart from the main voting procedure. In our model, two options are available:

1. At the beginning of each decision step, a short time is given for simple, introductory calculations so that voting may be performed as rapidly as possible. Section 4.2 discusses this in more detail.

2. Calculations which are very slow should be delegated to a separate thread or a dedicated processor. Ideally these would also perform anytime algorithms, so that proper judgment criteria could be decided at the beginning of any voting search step. Of additional note is that off-thread processing receives more CPU time when aspiration-based satisficing is used for limiting search time, thereby making more CPU resources available for complex computations.

2.5.2 State-Based vs. Action-Based

Another standard set by Rosenblatt in his work is that of voting on desired future states rather than on actions (or *commands* in his terminology). Among his conclusions are that "systems implemented with [state-based] utility fusion are less reactive but are better able to anticipate circumstances and to generate coherent action" [21, page 140]. State-based voting allows and requires the arbiter to perform more centralized reasoning. The arbiter becomes responsible for knowing which actions lead to which states, and it attempts to maximize expected utility based on the utility assigned by the behaviors to different states.

As Rosenblatt stated, there is both good and bad in this approach. As stated in this thesis on page 13, traditional behavior-based systems assume that behaviors are the experts. A smarter arbiter removes some expertise from the heterogeneous behaviors and places more into a centralized location.

Rosenblatt was able to report significantly better results from the state-based voting method than from the action-based. One of the primary reasons for this is related to the issue of synchrony discussed in the previous section. When voting asynchronously on actions, a vote from two seconds ago may be combined with a votes from a tenth of a second ago. The meaning of actions given the difference in state between two times can be significant. An action that was valid two seconds ago may not be so now, and combining utilities between the two times may be especially meaningless. Figure 2.2 shows how action votes from slow behaviors can become outdated. State-based voting helps alleviate the asynchronous voting problem because states from two seconds ago are the same as states now. Just the actions required to reach them may have changed.

Our methodology circumvents this problem by requiring that voting be synchronous, as discussed in the previous section. Behaviors, as domain experts, are responsible for knowing which actions to take and being sufficiently up-to-date on the environment to perform their tasks.

Note also that action-based voting and state-based voting are not mutually exclusive. Since behaviors may be implemented according to any algorithm, some behavior could be based on a state-based voting system. Another possibility is the use of a world model for slow behaviors to predict good actions within their own decision making.

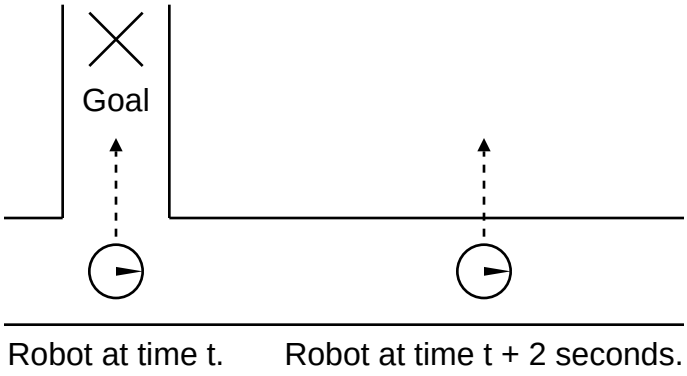

Figure 2.2: *When voting is asynchronous, action votes from slow behaviors can become outdated. Here, a behavior wants to reach the goal location. Voting to move left is valid at time t but not two seconds later when the behavior still has not updated its vote. A state-based vote asking to reach the goal location would still be valid.*

2.6 Summary

The motivation presented by this chapter is that the benefits described by previous research in voting are not applicable to control of full robot actuator systems without addressing the issue of large action spaces. In other words, *utilitarian systems cannot have full control of a complicated robot and still expect optimization.* Ad hoc fixes and control systems beyond the voting itself are required if this need is not addressed directly.

In order to address the needs of large action spaces, we present in this thesis an architecture based on

1. anytime search and

2. simple aspiration-based satisficing.

At the same time, the two traditional simplifications in utilitarian voting (ex-

haustively searching through each dimension independently and through combined dimensions at reduced resolution) can both be considered anytime searches. The problem is that they are not *directed* searches. The anytime search formalism, then, generalizes the concept of search to include both of the traditional simplifications. Implementations of each are compared with a genetic search. More details are found in chapters 5 and 6.

Chapter 3

Undesirable Emergence

3.1 Problem Overview

Whenever an algorithm is not implemented directly or analyzed completely, it can be difficult to be sure what the results will be. This applies to voting, because even when a single behavior is understood, the weighted combination of all behaviors may not be. The result is *emergent behavior* which is witnessed by an outside observer. Chapter 2 discussed the difficulty of dealing with large action spaces and how optimization is not always possible in real-time problems. This bounded rationality results in even more emergence.

At times emergence is good. Setting many constraints on a system may limit worse-than-desired results but it may also limit better-than-expected results. For instance, in the robot application of this thesis, no behavior specifically tries to make the robot go backwards at any time. However, when confronted with an obstacle, the robot sometimes backs up. This emergence results from a lack of "optimization" and is actually a beneficial feature.

Still, emergence can result in actions that are undesirable or even dangerous.

Designer constraints generally *do* exist on any real-world system, and boundedly rational voting does not, by itself, address such concerns.

3.2 Related Literature

Specifying the requirements of real-time control systems is an important issue [3], but little attention has been given to constraining the emergent character of voting systems. Some robot control architectures, such as Brooks's subsumption architecture, allow for designer-specified coordination between behaviors [4]. Such manual control provides a means to meet constraints. However, too much manual control defeats the loosely structured, decentralized control of voting [21].

One style of constraining emergence within voting systems is to override action choices using *vetoes* [6, 25]. With vetoes, a behavior can disallow unacceptable actions instead of simply being allowed to assign utilities. Vetoed actions cannot be taken no matter what other behaviors say. In both [6] and [25], utilities range between 0 and 1, and vetoes are expressed by some other value such as -1. [6] also describes a system wherein utilities below a certain threshold indicate vetoes. Neither paper emphasized the technique, but it is a flexible way of handling constraints in a decentralized, modular fashion. No complicated organization of behaviors is required.

Other constraint mechanisms have also been used, but without the full predictability of vetoes. Jukka Riekki, instead of using weighted averages, employs a "maximum absolute value" combination [19]. In this system, the action utilities are centered around 0. Negative values indicate bad choices, and positive values indicate good choices. Instead of summing utilities across behaviors, the vote with the greatest absolute value is chosen as the utility for the action. In this manner, a veto can be performed by voting more negatively than any other behavior votes positively. Such

a system is easier to predict than one that requires the negative vote to outweigh the *sum* of all other votes. Still, it requires interbehavior analysis to determine constraints, and in the end, only one behavior has a say at any time for any particular action. The dictatorial nature and the lack of predictability make it less practical than vetoes as an override mechanism.

Roland Stenzel uses a multiplicative T-norm rather than a weighted summation to fuse behavior votes [26]. Using votes in $[0, 1]$, a single vote near 0 results in a product of near 0. This utility fusion mechanism inherently causes that "inhibition of actions with low utility has high priority, while optimization between several actions with high utility is of low priority." In essence, the low product resulting from a low vote keeps the action from being performed. Because of this, all behaviors have some amount of veto power, except when all actions are voted very low by some behavior. Then the utilities would be less predictable, making this system less desirable than using vetoes for override purposes.

3.3 Vetoes and Hijacks

We use two types of override mechanisms for constraining undesirable emergence: *vetoes* and *hijacks*. These are implemented by distinct groups of behaviors. Veto behaviors are separate from hijack behaviors, and both are separate from ordinary voting behaviors.

When a veto behavior vetoes an action, that action has a minimum utility assigned to it, independent of the utilities assigned by the normal voters. This assures that a vetoed action is not taken unless all considered actions are vetoed.

While vetoes effectively eliminate unacceptable actions from being taken, vetoes are not sufficient. This point can be explained using an example given in the intro-

duction. When a robot is stopped near a wall, a veto may prevent it from moving into the wall. If, on the other hand, the robot were moving at full speed towards the wall, a veto would be insufficient. The deliberation process to find an acceptable action may delay too long to avoid a collision. In this case, a behavior could *hijack* the robot causing it to stop. Normal deliberation could follow once the robot were safe. Hijacks are decentralized in the same fashion as vetoes, but because only one behavior has influence, hijacks are best reserved for very high risk situations. In essence, hijacking provides a means to bypass deliberation to avoid catastrophic consequences.

For the present, our decision model is based on time steps. At the beginning of each time step, hijackers are responsible for deciding if any unacceptable consequences could occur before the step is finished. If a hijacker determines that such a consequence would take place, it hijacks at the beginning of the step. Once deliberation has begin, hijackers cannot interrupt the search process, since they have already had the opportunity to determine danger.

Hijackers are listed in order of priority, so that there is no ambiguity of what will happen in case more than one hijacker wishes to take control at the same time. The hijacker of highest priority has its action taken.

By separating veto and hijack considerations into one behavior per concern, the designer is free to mix and match constraints in the same way voting behaviors allow modular objectives. There is a difference, however; the designer should be aware of the effects of all vetoers and hijackers, because they break the ability for *multiple* objectives to be considered simultaneously.

3.4 Designer Considerations

3.4.1 Veto Flatness and Indecision

There are two substantial side-effects that can result from overuse of vetoes: (a) flat areas in the action-utility curve and (b) indecisiveness if all actions are vetoed.

Normally, a weighted utility summation results in a sloped curve that gives a smart search algorithm some idea of where to find better actions. If a substantial number of actions are vetoed, large flat areas in the curve could result which could make searching more difficult. Figure 3.1 further demonstrates this consequence. In

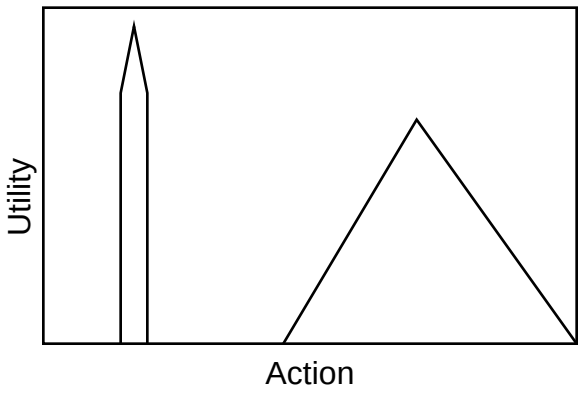

Figure 3.1: *Flat utilities produced by excessive vetoes. The left half of the figure represents a part of the action space where many actions have been vetoed. On the right hand side, no vetoes have been performed. Any action on the right half has neighbors which represent the slope towards the local maximum utility. The mode on the left is a greater maximum, but the excessive vetoes make it harder to find. Only actions very near the maximum give an indication of where the best action is.*

the extreme case where all actions were vetoed, there would be no way to tell which actions were truly better than others. Thus, indecision would result.

Stenzel's T-norm voting avoids these problems by not having hard vetoes but simply low votes instead [26]. The utility curve would be effectively scaled down instead of becoming flat and indecisive, unless pure 0 votes were made. However, separate veto behaviors are more predictable and defined, which is the primary purpose of overrides in this thesis.

Predictable vetoes may be maintained so long as they are not used excessively. As discussed earlier, the designer should be aware of all effects of all overrides used and should be sure that vetoes are not used excessively.

3.4.2 Hijack Disregard

Hijacks have the potential to damage the decision process even more than vetoes. When actions are vetoed, those actions will not be taken, but any action that *is* taken will have been evaluated by all voting behaviors.

When a hijack is performed, the remaining objectives are entirely ignored. For instance, if, in a mobile robot control system, a hijack behavior were responsible for braking when collisions were imminent, the system would ignore the opinions of other behaviors. If an obstacle in front of the robot could be dodged, and some other voting behavior knew this, that opinion would remain unheard. There might also be some other behavior aware of time limits for reaching some goal who would also be ignored. In other words, hijacks may be implemented by modular behaviors, but they do not represent multiobjective decision making. Still, there are times when such direct control *is* necessary to ensure the safety of the system, but the designer should be aware of all overrides used in the system.

3.5 Summary

To constrain undesirable emergence, we use a system of overrides. In all, we use three types of behaviors:

1. voters,
2. vetoers, and
3. hijackers.

Veto and hijack mechanisms allow for modular behaviors to override the standard voting process in order to guarantee that unacceptable consequences do not take place. This provides predictability that is necessary for many real-world systems.

Chapter 4

Bruvo 1 Architecture

4.1 Overview

In this thesis we present a software architecture named *Bruvo 1* which is based on the principles discussed in chapters 2 and 3. Bruvo 1 stands for Boundedly Rational Utilitarian Voting with Overrides, version 1. A Bruvo 1 implementation includes the following components:

1. Agent

   The agent, operating in time steps, is responsible for choosing an action given the state of the environment. For Bruvo 1, it follows the same procedure for any application. It maintains the aspiration level, handles satisficing, works with the behaviors, and calls the search algorithm.

2. Search Algorithm

   Any search algorithm based on utilities can be used by a Bruvo 1 agent so long as it supports being halted at any time. For brevity's sake, the search algorithm is also called the *searcher*.

3. Hijackers

   Listed in order of priority, hijackers can force an action to be taken by the agent before the satisficing or search process begins.

4. Voters

   The voters assign potential utilities from the set $[0, 1]$. Their votes are combined by a weighted average.

5. Vetoers

   Vetoers prohibit actions from being taken. If any single vetoer vetoes an action, that action's utility is effectively 0. In this sense, all vetoers have equal authority.

Figure 4.1 shows the Bruvo 1 architecture in UML notation. All classes with set procedures are shown, and interfaces are given which represent items that are implemented by the system designer working with a particular application. The remainder of the chapter discusses in more detail how the components fit together.

4.2 Bruvo 1 Agents

For the context of this thesis, an *agent* is a decision maker that maps states to actions at discrete time steps. An internal state is also allowed. Formally, this can be expressed as:

$$act : State_{environment} \times State_{agent} \rightarrow Action \tag{4.1}$$

An agent could be implemented in any number of ways, and Bruvo 1 provides a specific algorithm for agents to use. Figure 4.2 shows in UML notation the above concept of generic agents and, more specifically, how a Bruvo 1 agent is put together, including

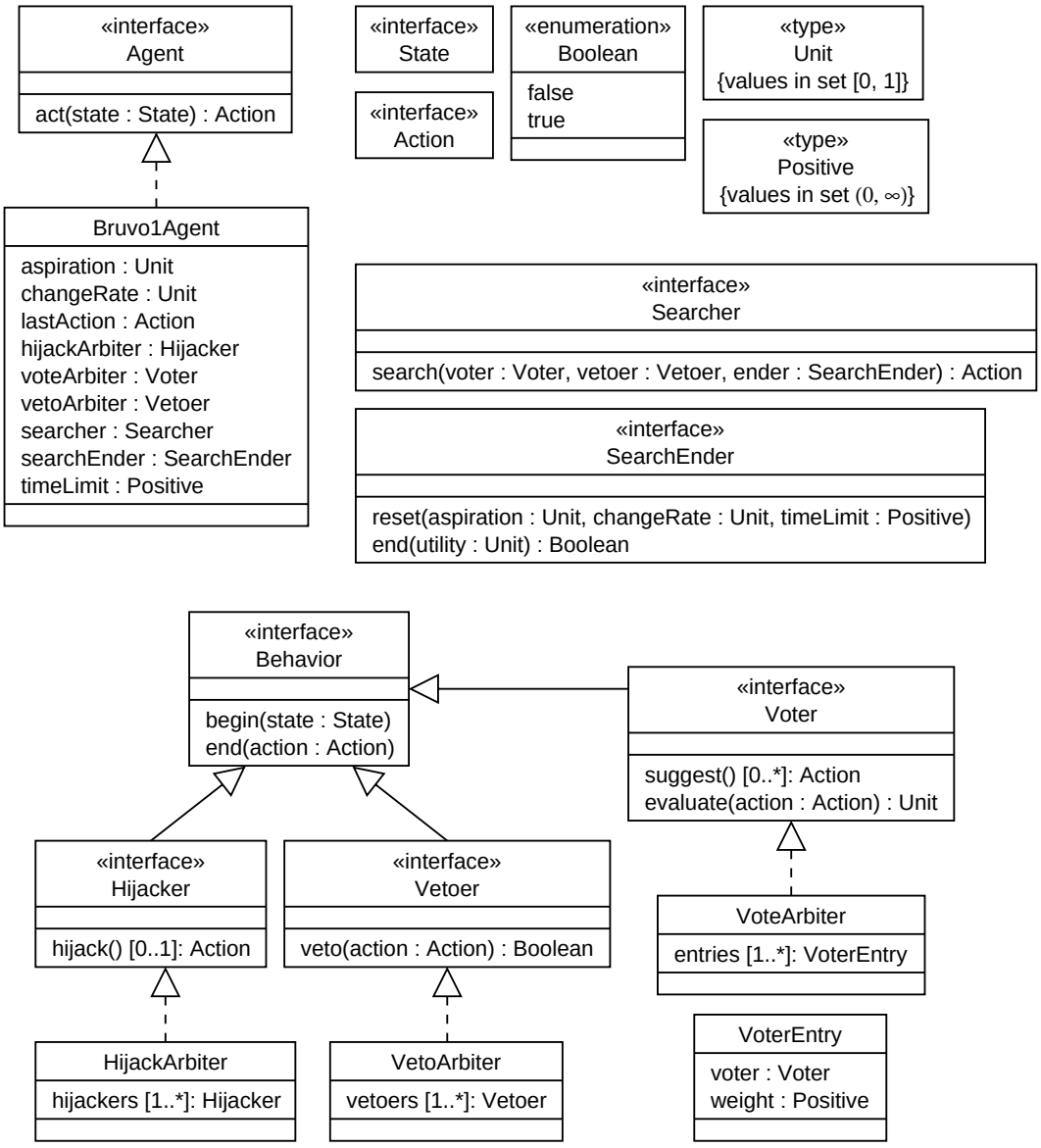

Figure 4.1: *UML diagram of the Bruvo 1 architecture.*

the features that compose its internal state. Note that actions and environmental states are very abstract data types and are only defined within the context of a specific application.

Each Bruvo 1 agent (from now on, simply *agent*) is aware of three *arbiters*. In Bruvo 1, an arbiter is a behavior which fuses a set of other behaviors. This follows

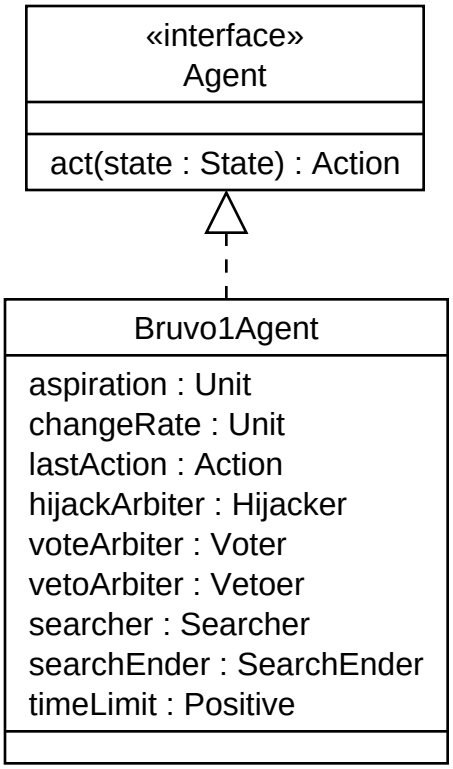

Figure 4.2: *UML diagram of the Bruvo 1 agent class and its generic agent interface. See figure 4.1 for a full diagram of the Bruvo 1 architecture.*

the *composite* design pattern of [8], where there is no distinction between an item and a collection of items at the interface level. So far, no explicit use is made of this composite structure, but it simplifies a view of the system.

At any time step, the agent performs the following algorithm:

1. Tell each of the three arbiters that a new step has begun, informing them of the current state of the environment.

2. Find out if the hijack arbiter wishes to hijack. If so, select its chosen action and go to step 6.

3. Find out if the vote arbiter evaluates the action taken at the last time step

as greater than or equal to the current aspiration level. If so, and if the veto arbiter does not veto it, select the last taken action and go to step 6.

4. Reset the *search ender*, and call the searcher with the vote arbiter, the veto arbiter, and the ender. Section 4.3 discusses how the search process works.

5. Select the best action found by the end of the search.

6. Tell the arbiters what decision has been made.

7. Update the aspiration level according to equation 2.1. If a vetoed action is taken, it is considered to have utility 0 when updating the aspiration.

Step 1 is for introductory computation that may be too expensive to handle in the main voting process. It also provides a context (along with step 6) so that the behavior can follow the process of the agent's decision making.

4.3 Search Process

The searcher has a very simple interface. It receives a voter, a vetoer, and a search ender. The voter is used to get search seeds and to evaluate potential actions. If an action is vetoed, the searcher should consider its utility to be -1[1]. The ender says when the search must terminate, either due to an expired time limit or some other satisficing criteria. Figure 4.3 shows the searcher and search ender interfaces.

In general, Bruvo 1 assumes that finding *valid* actions is relatively simple. In other words, most or all combinations of velocity, turning speed, and so on can be sent to the actuators, even if the result is unproductive. This has the effect that almost all

[1]By giving vetoed actions a utility of -1, they can only be taken if all other considered actions were also vetoed.

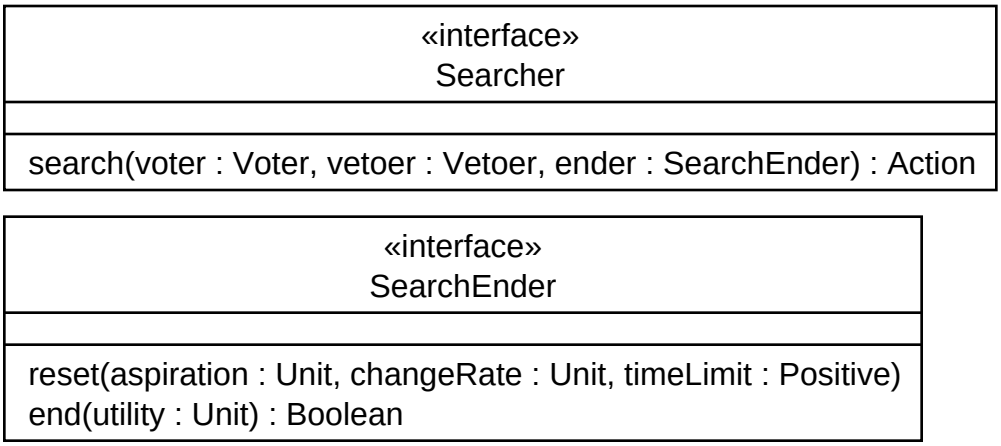

Figure 4.3: *UML diagram of the interfaces for searchers and search enders. See figure 4.1 for a full diagram of the Bruvo 1 architecture.*

simple searches are effectively anytime algorithms. At each iteration through its loop, any searcher can query the ender to know if the search should be stopped. If so, the searcher can return the best action found so far.

We have implemented three search algorithms, and each is discussed in detail in chapter 5. For instance, the genetic algorithm searcher combines and alters behavior suggestions until the search ends. The search could be ended, for instance, when an action meeting the aspiration level were found. Other search enders are also discussed in chapter 5. Of course, all search enders should end at the time limit when they do not end earlier.

For now, we are not concerned with hard real-time constraints. The searchers currently implemented only check the search ender once per iteration through their main loop. If the time limit expired just after the check, an indeterminate amount of time could be expended before ending the search process. Simple modifications to this architecture would be needed to handle strict time limits, such as interrupt-based ending of searches.

4.4 Behaviors

4.4.1 Common Traits

Every Bruvo 1 behavior shares two methods: *begin* and *end*. These methods (described earlier in the discussion of the agent's *act* method as steps 1 and 6, respectively) provide a way for the behaviors to be aware of the decision-making process. They allow the behaviors to know what state the agent is in and what action is finally decided at each time step. Figure 4.4 shows the behavior hierarchy.

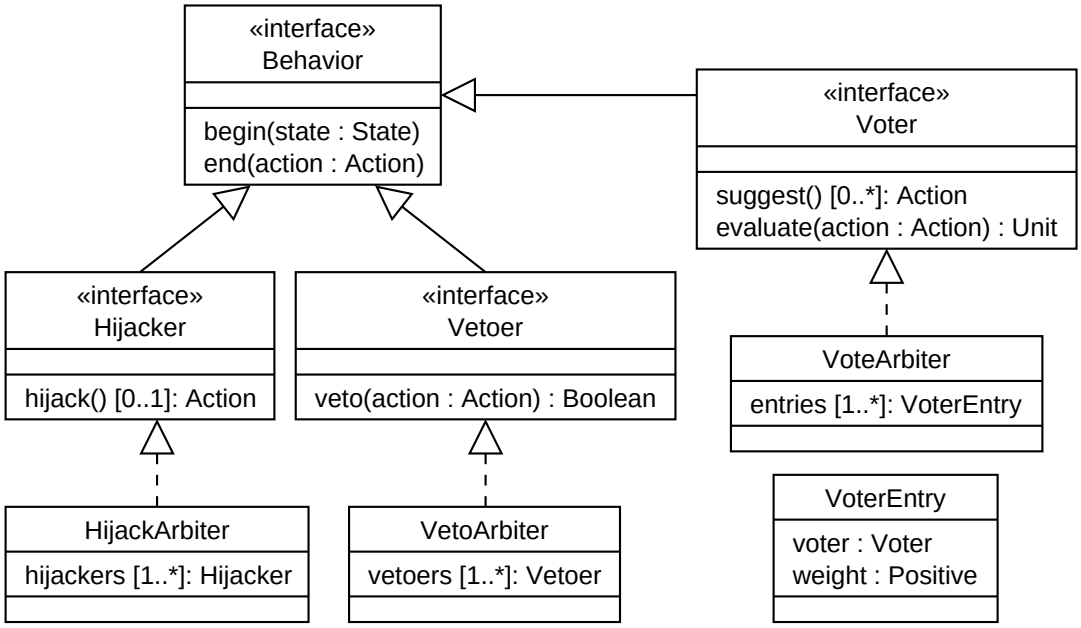

Figure 4.4: *UML diagram of all the behavior interfaces and classes. See figure 4.1 for a full diagram of the Bruvo 1 architecture.*

All preliminary computation needed by each behavior should be done in its *begin* method. This allows voting to proceed as quickly as possible. For instance, a goal-seeking behavior could decide in this method what direction it prefers to go. Assigning utilities to actions during the voting process would then be a simple function of the

preferred direction.

Hijackers, voters, and vetoers each have an associated *arbiter* class. Each arbiter is also a behavior, and the agent knows only about the arbiters and not about the individual behaviors that compose them. Each of the three types of arbiters perform their *begin* and *end* method the same. Whenever *begin* is called on an arbiter, it calls *begin* on each of its sub-behaviors. The same is true of *end*.

Sections 4.4.2, 4.4.3, and 4.4.4 discuss the different types of behaviors in more detail.

4.4.2 Hijackers

Hijackers are responsible for knowing at the beginning of the decision step whether or not they need to act. Bruvo 1 does not provide a consistent way of updating state information in the middle of a step, so there is little benefit in allowing hijacks after the beginning.

A hijacker returns one or zero actions from its *hijack* method. If it returns an action, that action is taken by the agent.

A hijack arbiter calls *hijack* on each of its sub-hijackers in the order they are listed. The first hijack action returned is selected as the action of the arbiter. If no sub-hijackers hijack, then neither does the arbiter.

4.4.3 Voters

Voters are responsible for the primary decision process. Most importantly, voters implement the *evaluate* method by which they assign utilities to actions. As described in figure 4.1, utilities are between 0 and 1. Vote arbiters vote by a weighted average of the evaluations of their sub-voters. Specifically, the utilities are combined by the

following equation:

$$evaluate_{arbiter}(action) = \frac{\sum weight_{voter} \times evaluate_{voter}(action)}{\sum weight_{voter}} \tag{4.2}$$

Voters also have a *suggest* method by which they can give aid to the searcher if requested. Many search algorithms are benefited by seeds. The voters suggest seeds by giving a list of preferable actions. Vote arbiters suggest by calling *suggest* on each of their sub-voters and returning a concatenated list of actions.

4.4.4 Vetoers

Vetoers affect the decision process by disallowing unacceptable actions. The method *veto* returns true to indicate that an action is vetoed. Veto arbiters veto if any of their sub-vetoers do so.

As discussed earlier, the search process assumes that vetoed actions have a utility of -1, and if a vetoed action is taken, the action is assumed to have a utility of 0 when updating the aspiration. This ensures that vetoed actions always have lowest priority and that the aspiration level stays in between 0 and 1.

4.5 Summary

Bruvo 1 is a simple software architecture which implements a basic utilitarian voting framework based on the principles discussed in this thesis. It allows for searching very large action spaces under time and CPU limitations. It also provides a system for overriding unacceptable actions and supporting designer constraints on the system.

The core architecture provides a way of using a searcher, as well as groups of hijackers, voters, and vetoers. Actual search and behavior algorithms must be supplied

for specific applications of the Bruvo 1 architecture. We discuss such an application (along with behaviors, searchers, and search enders) in the next chapter.

Chapter 5

Robot Implementation

5.1 Requirements

The specific problem targeted in this thesis is the use of a Nomad Super Scout robot to wander, navigate hallways, avoid collisions, and move to user-specified goal locations. In other words, the Bruvo 1 architecture has been used to develop a mobile robot capable of both fully autonomous control as well as semiautonomous goal seeking.

Internally, the robot runs Linux on a 233 MHz Pentium processor with 64 MB of memory, a hard drive, and a wireless Ethernet connection. Figure 5.1 illustrates the physical layout of the Super Scout. The wheels are controlled independently by a microcontroller that receives velocity and acceleration instructions from the main computer. 16 sonars line the perimeter of the robot, and the robot has odometers, a compass, and other sensors necessary to report location, angle, velocity, and acceleration information.

The action space used for the control system includes 501 different translational velocities, 301 rotational velocities, 390 accelerations, and 255 sonar firing rates. This results in a total of about 15 billion possible actions at any time step. Since, for

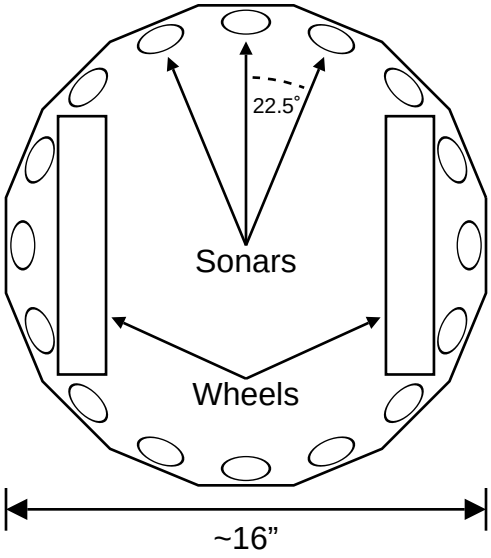

Figure 5.1: *Nomad Super Scout mobile robot. The figure is not to scale but does show the primary features relevant to our application.*

the tests reported in chapter 6, 0.05 seconds are allowed to find the best action possible before a decision must be made, a 300 GHz processor would be needed, even if all behaviors could evaluate an action and have their votes summed in one CPU clock cycle. This is obviously an unrealistic assumption given current computing capabilities, so some simplifications must be made in one way or another. The Bruvo 1 architecture addresses this need with boundedly rational techniques such as anytime search.

The time required for each decision step is divided into two parts: (a) the actual decision making, including behavior processing and the core decision system, and (b) sensing and actuating. We will now address three issues related to processor usage and overall execution time.

First, no behaviors in this application require extensive processing, but other applications could use video data processing and other CPU-intensive algorithms.

Having such behaviors share CPU resources with the core decision system should not be impractical. Consequently, a specific desirable property of the decision system is minimal CPU usage. The aspiration-based satisficing used by Bruvo 1 helps address this concern.

Second, the time limit of 0.05 is imposed on the core decision system in order to keep processing down. As is shown in chapter 6, the short time limit is sufficient for this application. The delay of reading sensors and controlling actuators on the robot is usually between two and four times longer than the decision time limit. Therefore, the decision time chosen is not the bottleneck in overall system execution time.

Third, controlling the speed and orientation of the robot is obviously useful, but the other action dimensions of acceleration and sonar rate may be less clearly needed. The primary motivation for control of acceleration is that usually a preset, moderate acceleration rate is useful, but at times, especially for collision avoidance, raising the acceleration rate (i.e., rapid deceleration in this case) can be very useful. Sonar rate is controlled because different sampling rates are required at different robot speeds. While complicated sonar firing control can be very practical [20], modifying sonar settings on the Nomad Super Scout is too slow to allow sophisticated control. Nevertheless, the simple control implemented here is a step towards making more complicated sonar control systems useful.

5.2 State and Action Representations

For the purposes of this application, we use the following state values:

1. 16 sonar distances, measured in inches, organized in a ring every 22.5°. The sonar values range from 0 to 255 inches, inclusive, but due to physical limitations of the sonars, distances closer than about 10 inches are not accurately

represented.

2. Current *translational* and *offset* velocities in tenths of inches per second. Recall that the microcontroller controls the left and right wheel velocities, v_l and v_r, respectively. The *translational velocity* is the mean velocity of the two robot wheels ($\frac{v_l+v_r}{2}$) and represents the forward or backward speed of the robot. The *offset velocity* is half the difference between one wheel's velocity and the other ($\frac{v_r-v_l}{2}$), and it represents the speed at which the robot rotates (counterclockwise) though it is not precisely a rotational velocity in the formal sense. The translational velocity ranges from -250 to 250 tenths of inches per second, and the offset velocity ranges from -150 to 150 tenths of inches per second.

3. Robot x and y coordinates in tenths of inches relative to an arbitrary origin.

4. Goal x and y coordinates in tenths of inches, if a goal exists. The origin is the same as for the robot's position.

5. The current autonomy mode of the robot, whether it should just wander or whether it should seek the goal point.

An action is a vector consisting of the following values:

1. The translational velocity (as defined in the state representation list) in tenths of inches per second. The value ranges from -250 to 250.

2. The offset velocity (also defined in the state representation list) in tenths of inches per second. The value ranges from -150 to 150.

3. The absolute value of the acceleration that should be used for changing wheel velocities, in tenths of inches per second squared. The value ranges from 1 to 390. Lower values cause the robot to accelerate (slow down or speed up) more

slowly when a change in translational or offset velocity is specified; the inverse effect takes place for higher values.

4. The sonar firing interval in four millisecond units. The time actually taken by the sonar to be sent and received is also added to this interval. The value ranges from 1 to 255, indicating intervals between 4 and 255 $\times$ 4 milliseconds. Very low intervals (or high firing *rates*) cause the sonar information to be more noisy because of echoes.

5.3 Behaviors

5.3.1 Overview

In this section, we discuss the behaviors used in this application of the Bruvo 1 architecture. These behaviors have not been optimized for task-specific functions but rather selected because they are straightforward building blocks useful for many tasks. They are sufficient for producing an autonomous robot capable of avoiding obstacles and seeking goals, as will be shown in chapter 6.

We have implemented eight behaviors: one hijacker, one vetoer, and six voters. The voters have simple voting patterns as is seen in the following detailed discussion of the behaviors. Usually, each voter decides an ideal action or actions to be taken, and assigns a vote of 1 for that action. Exponential drop-off then occurs toward 0 as actions differ from this *desired setting*. Specifically, the utility evaluations are based on the following equation:

$$evaluate(action) = \left(1 - \frac{|setting_{action\ dimension} - desired\ setting|}{total\ settings_{action\ dimension}}\right)^{exponent} \quad (5.1)$$

Most of the voters consider only one dimension of the action vector (although a few average their evaluation of multiple dimensions). The term *action dimension* refers

to that particular part of the action vector, either the translational velocity, offset velocity, acceleration, or sonar interval. The variable *total settings* refers to the total number of available actions for that dimension (e.g., 501 in the case of the translational velocity). The *exponent* is chosen by the behavior to express how strongly it cares about its *desired setting*. Note that this equation always results in a proper utility between 0 and 1, inclusive. Figure 5.2 illustrates the type of shape produced by this evaluation function.

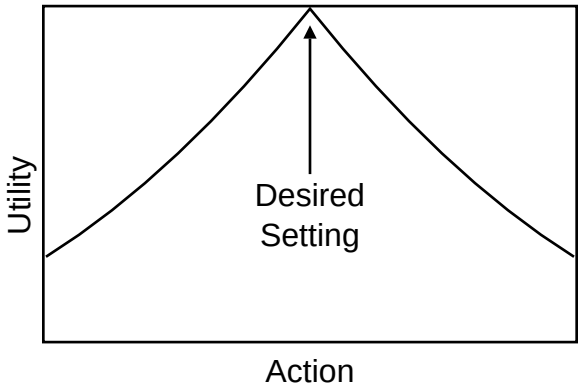

Figure 5.2: *Example utility function resulting from equation 5.1. In this case, an exponent of 2 was used with the desired setting directly in the middle. The* seek goal *behavior (see section 5.3.8) would vote this way if it were active and facing the goal.*

The following subsections describe each behavior individually. The voting and suggestion rules for each are described, and the weight setting for each is given for use in equation 4.2. The total weight of all voters is 8.6. One final comment about the suggestion rules is in order. Each suggestion is a full action vector, even when a voter uses only one action dimension for assigning utility.

5.3.2 Avoid Crash Hijacker

This behavior is responsible for assuring that the current state of the robot will not lead to a collision. There are two conditions that it checks for:

1. If the current velocity would lead it to collide with any obstacles detected by the five front or five back sonars before the next decision step, a hijack is performed. To determine when the next step will begin, it keeps track of the maximum delay between decision steps, as marked by calls to its *begin* method.

2. If any of the five front sonar distances is less than 10 inches and the translational velocity is positive, or if any of the five back sonar distances is less than 10 and the translational velocity is negative, a hijack is performed. The 10 inch limit is used because the sonar readings are inaccurate below that level.

The *avoid crash* hijacker always performs the same action when hijacking; it sets acceleration to the maximum value of 390 tenths of inches per second squared, it sets translation and offset velocities both to 0, and it sets the sonar interval to 20 (i.e., 80 milliseconds). This causes the robot to stop as fast as possible and to continue observing the environment via the sonars.

5.3.3 Avoid Crash Vetoer

This behavior is responsible for assuring that actions are not taken which lead the robot into immediate likelihood of collision. It uses both of the same metrics used by the *avoid crash* hijacker except that it considers the velocity proposed in a new action rather than the current velocity. In other words, any proposed actions with a velocity that would cause a hijack are vetoed.

The *avoid crash* vetoer also vetoes actions that have non-zero translational velocity and a sonar firing interval of above 20 (i.e., 80 milliseconds). This prevents the robot from reaching a state where it is unable to detect obstacles while still moving forward or backward.

5.3.4 Center in Hall

This behavior tries to keep equal distances to obstacles (such as walls) on both sides of the robot. The distances are determined by the minimum distance reported by the five sonars on the left and the minimum of the five on the right. The difference between the left and right distances is then used in a simple proportional-derivative (or PD) controller which determines the desired offset velocity. PD controllers are based on the following equation:

$$output(time) = k_p \times input(time) + k_d \times \frac{d\ input}{d\ time} \tag{5.2}$$

The terms k_p and k_d are constants that scale the relative importance of the proportional and the derivative components, respectively. The *center in hall* behavior uses the difference between left and right distances as the input ($input = distance_{left} - distance_{right}$), and the *output* is scaled and truncated to an appropriate offset velocity. A PD controller is used for this behavior because the derivative component helps to avoid overcompensation that otherwise results. That is, it is easy to overcorrect when trying to stay centered in the hallway.

The *center in hall* behavior determines utilities by equation 5.1 with the desired setting obtained from equation 5.2 and an exponent of 5, which is sharper than most other voters. It votes so sharply because staying centered in the hall requires a certain amount of precision. The robot can otherwise turn too much and become confused. It is given a weight of 1.0. It gives as a suggestion an action with its desired offset,

a sonar interval of 10 (i.e., 40 milliseconds), and velocity and acceleration of half the maximum value of each.

5.3.5 Move Forward

This behavior always wants to move forward at 250 tenths of inches per second (the maximum speed) with an acceleration of 200 tenths of inches per second squared. It provides the impetus for forward motion of the robot. It votes using equation 5.1 with an exponent of 2. The utility is calculated for both translational velocity and acceleration, and the average of the two is reported.

This behavior's weight is 0.5. It suggests going full speed, straight ahead, with a sonar interval of 10 (40 milliseconds) and an acceleration of 200 tenths of inches per second squared.

5.3.6 Regulate Speed

This behavior tries to maintain of safe driving distance from obstacles in front of or behind the robot. More specifically, it attempts to maintain a headway (the time it would take to reach a stationary obstacle) of 2.5 seconds. It determines the distance to obstacles by the minimum of five sonar readings to the front or back, depending on whether the robot is currently moving forward or backward, respectively. This behavior heuristically estimates headway using straight line trajectories (dependent only on the translational velocity) and ignores curved trajectories. We have previously used similar headway maintenance in other types of robot control systems [11]. Specifically, the headway is defined as follows:

$$headway = \frac{distance_{front}}{translational\ velocity} \tag{5.3}$$

If a candidate translational velocity is below the threshold for maintaining the proper headway, *regulate speed* votes 1.0 as a "don't care" vote; otherwise, it uses equation 5.1 with an exponent of 2.5. Figure 5.3 illustrates the voting pattern used by *regulate speed*.

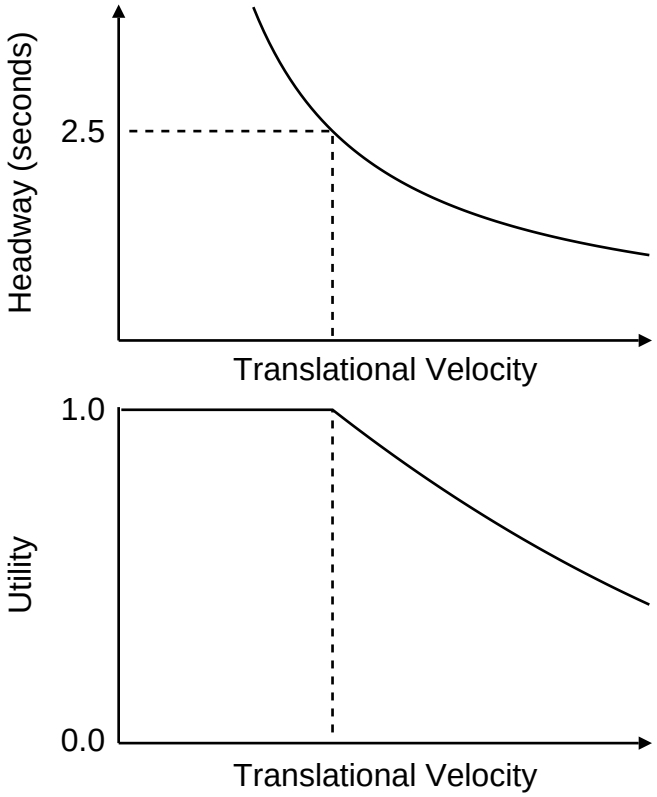

Figure 5.3: *Illustration of* regulate speed*'s voting pattern. Given the current distance to obstacles in front of the robot, the headway can be determined for any translational velocity. The* regulate speed *behavior votes down any velocity that would cause a headway of less than 2.5 seconds. Here, the desired maximum velocity is represented by the vertical dashed line.*

The *regulate speed* behavior's weight is 1.0, and it gives the same suggestion as *move forward*, except with its desired maximum velocity above which the 2.5 second

headway cannot be maintained.

This behavior is largely responsible for avoiding too much use of the override behaviors. As discussed in chapter 3, override behaviors do not maintain a multi-objective perspective. Because of this, it is preferable to use standard voters when possible. The speed regulation provided by this behavior helps to avoid situations where collisions are likely and can be used to cause the robot to follow another robot or human [11].

At the same time, a single behavior controlling the translational velocity is insufficient, which is why this behavior is separate from *move forward*. This behavior needs a suitably high weight to avoid dangerous situations, but *move forward*'s weight can be much smaller; at times it may need to be easily outvoted by goal-seeking or other objectives.

5.3.7 Silence

This behavior is the only voter that regulates the sonar firing interval. If either the translational or offset velocity of the robot is non-zero, it wants the sonar interval to be between 10 and 25 (i.e., 40 and 100 milliseconds). The desired interval is inversely proportional to the sum of the absolute values of the velocity and offset. On the other hand, if the robot has not moved at all for 1.5 seconds (usually due to a human disabling the motors), it desires the maximum interval of 255 (i.e., 1020 milliseconds).

Usually, it determines utilities by equation 5.1 with an exponent of 3, but if the sonar interval under consideration is below 10, it uses an exponent of 9. This is because the sonars become less accurate when fired too quickly.

Its weight is 0.1, making it the least influential of all the voters. This is because the *avoid crash* vetoer guarantees that the sonar interval is usually safe. However,

regulating the sonars is mostly for the comfort of nearby people, because the sonar chirping aggravates human operators and observers. It is not vital to the core functionality of the robot, so its small weight helps it to not influence the voting too much. However, this behavior could serve as the basis for more complex sonar firing patterns if so desired.

At each time step, it suggests its desired sonar firing interval with offset and translational velocities of 0 and an acceleration of 200 tenths of inches per second squared.

5.3.8 Seek Goal

This behavior implements a simple goal seeking strategy that makes no use of history. It simply considers the robot's current position and orientation, the goal's current position, and the current distances to obstacles surrounding the robot. It can be in one of three modes at any time: *pivot*, *orient*, or *inactive*.

The *seek goal* behavior is in the *pivot* mode whenever it would need to turn more than 90° to face the goal. When in pivot mode, this behavior desires a translational velocity of 0 and an offset velocity proportional to the relative angle to the goal. In figure 5.4, robot 1 would need to turn more than 90° to reach either goal 1 or goal 2. If either goal were active for it, its *seek goal* behavior would be in pivot mode. Neither of the two other robots would use the pivot mode to seek either goal. When in this mode, *seek goal* applies equation 5.1 with an exponent of 1 to evaluate translational velocities. This vote is averaged with a vote on the offset velocity using a variation of equation 5.1 with an exponent of 2. In this variation, angle differences are calculated in a circular fashion. For instance, an offset of -149 tenths of inches per second is considered to be only 2 away from 149. Strong right turns are not very different from

strong left turns.

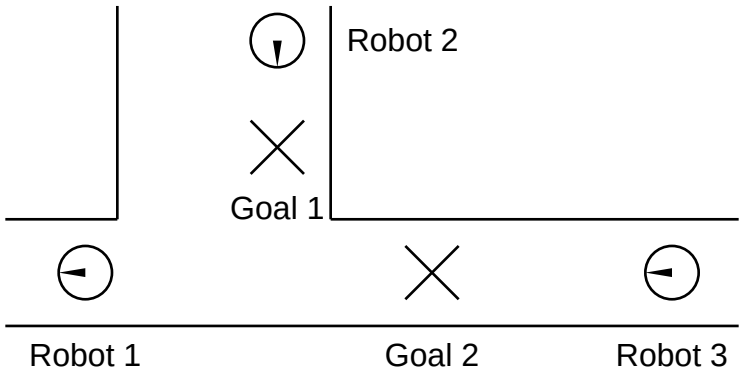

Figure 5.4: *Diagram for showing examples of the* seek goal *behavior.*

In the *orient* mode, the *seek goal* behavior tries to orient towards the goal but does not try to stop the robot's forward motion when orienting. The orient mode is used if both of the following conditions are met:

1. The walls to either the left or the right of the robot are more than 4.5 feet away, or the goal is less than 4.5 feet away. For instance, in figure 5.4, only robot 2 meets this condition. The distance of 4.5 feet was empirically chosen, largely based on the hallway widths in which the robot would be operating. In general, when the goal is far away, narrow halls can usually be more effectively navigated by the normal wandering ability produced by the other behaviors.

2. There is no obstacle between the goal and the robot, or the obstacle is more than 4.5 feet away. Again, in the figure, only robot 2 meets this condition.

In both cases, distances are based on the minimum value reported by the three sonars most directly facing the goal. If the orient mode is chosen, *seek goal* desires an offset proportional to the relative angle to the goal, but any translational velocity is

acceptable. It calculates utilities for offset velocities using circular differences and an exponent of 2, as when in pivot mode.

If *seek goal* is in neither the pivot nor the orient mode, it is *inactive*. When inactive, any action is acceptable, so it votes a "don't care" vote of 1.0 for all candidate actions.

This behavior's weight is 5.0, making it the most influential behavior. This is to allow it to control the robot when a goal is nearby but no other behaviors are interested in going towards the goal.

If it has not decided to be inactive, it suggests its desired offset with an acceleration of 200 tenths of inches per second squared and a sonar interval of 10 (i.e., 40 milliseconds). If it is in pivot mode, it suggests a velocity of 0. If it is in orient mode, it suggests going full speed. If it is inactive, no suggestion is made.

5.3.9 Turn

This behavior is responsible for turning away from nearby obstacles. It is only active when any of the front five sonars return a distance of less than 30 inches.

When active, it must decide which way to turn to avoid the obstacle. If any of the three left-most sonars say the distance to the left is 10 inches or less and the distance to the right is more than 10 inches, it wants to turn right. The opposite holds for turning left. For when neither case holds, *turn* tracks the recent tendency of the robot to turn left or right. If there has been a strong tendency one way or the other, it wants to turn that way. If there has been no strong tendency, it wants to turn either way.

If it wants to turn left, it desires an offset velocity of 20 tenths of inches per second or more, and if it wants to turn right it desires -20 tenths of inches per second or less. If it is willing to turn either direction, then all offset velocities with absolute

values greater than 20 tenths of inches per second receive a vote of 1.0.

Candidate offset velocities that are too small ($|\textit{offset velocity}| < 20$) or that correspond to the wrong direction (only applicable when both directions are not acceptable) have their utility drop off using equation 5.1 as with other behaviors. The exponent used by *turn* is 2 times the time in seconds that *turn* has been active. This increasing exponent is to encourage more turning the longer the robot has been facing an obstacle.

As with other selectively active behaviors, when *turn* is inactive, it votes 1.0 for all actions.

It has a weight of 1.0 and suggests its desired offset if it active or an offset of 0 if not. If either direction is fine, it gives two suggestions (one with offset -20 and the other with offset 20). All of its suggestions have an acceleration of 200 tenths of inches per second squared, a sonar interval of 10 (i.e., 40 milliseconds), and a velocity of one fourth the maximum.

5.4 Searchers

5.4.1 Overview

As discussed in detail in chapter 2, the state of the art in action search for utilitarian voting systems involves either reduction of actuator resolution or searching each action dimension independently or both.

For instance, Rosenblatt both reduces resolution and separates dimensions [21].

Pirjanian also uses reduced resolution; although his system combines both translational and rotational velocities, his system does not consider negative velocities nor does it consider sensor sampling rate, acceleration rate, and other actuator dimen-

sions [17]. In summary, every search algorithm used for utilitarian voting has been based on low resolution, split action space, or both[1]. No consideration has been given to degradation quality for searches cut short in attempts to reduce CPU usage, either.

To test our framework, as represented by the Bruvo 1 architecture, we have implemented three searchers: *genetic algorithm*, *low resolution*, and *split space*. The genetic algorithm represents a style of action search compatible with our framework that takes advantage of behavior suggestions and is meant to degrade well when searches are ended early. The other two searches represent the current state of the art, especially when used without satisficing. They are undirected searches and make no use of behavior suggestions. However, by extending them to be used within the Bruvo 1 architecture, we can apply satisficing techniques to them. This also allows us to analyze them from an anytime perspective. We show the results of such analysis in chapter 6. We discuss each of the three searchers in the next three subsections.

5.4.2 Genetic Algorithm

The genetic algorithm search is a simple directed search that combines and alters its population at the level of the separate action dimensions. As examples, the velocity of one action may be swapped with that of another, or the offset may be altered randomly. Fitness is based on utility with a small amount of noise added. The action with the highest utility, however, is always kept between rounds. This is to ensure a monotonic increase of utility during the search, a desirable feature of anytime algorithms [28].

Before the search begins, all behavior suggestions are replicated until the search population size is filled. In our experiments, we have used a population size of 50 actions. At each iteration, the following steps are followed:

[1]See chapter 2 for more discussion on this topic.

1. Utilities are assessed with noise added to all actions in the population except the best action. If there are more than one action with maximal utility, the best action is chosen randomly.

2. The top 50% of the actions are kept, and the remaining space is filled with combinations of these actions. Combinations are obtained by randomly selecting two of the surviving action vectors and, for each dimension, randomly choosing the value from one of the two actions or, with 0.1 probability, the mean of both values.

3. All but the previously best action are mutated with 0.05 probability. When an action is mutated, only a single, randomly chosen dimension is altered. The value of that dimension is changed to any possible value using a uniform distribution.

The parameters used in this algorithm (and the algorithm itself) have been empirically determined to produce functional robot behaviors. However, these parameters have not been optimized because such optimization is beyond the scope of this thesis. Still, the algorithm produces reasonable robot performance as demonstrated in the next chapter.

5.4.3 Low Resolution

The low resolution search reduces the resolution of the action space until it can test all remaining actions within the time limit. It is actually an interpolating search as well. Once the best action has been found from the original options, a quadratic fit is performed on the action vector along each dimension independently. The utility of the action and the setting for each dimension, along with its two nearest neighbors, is

used to perform the fit. This is the same type of interpolation used by Rosenblatt in his architecture [21]. If the interpolated action has a higher utility than the original, it is used instead.

The resolution along each dimension has been empirically chosen to allow exhaustive search in the current application:

1. The translational velocity is split into 26 steps of 20 each. That is, its resolution is 2 inches per second rather than 0.1 inches per second.

2. The offset velocity has retained the highest resolution because it is most carefully controlled by the behaviors. It is split into 51 steps of 6 tenths of inches per second each.

3. The acceleration is chosen from only two settings, the maximum and one half the maximum (390 and 180 tenths of inches per second squared). The only values asked for by the behaviors are 390 and 200 tenths of inches per second squared. This is reasonable, but because the low resolution has not been designed to exactly fit the current behaviors, a certain loss of utility results.

4. The sonar firing interval is chosen from two settings as well, 10 and 255 (i.e., 20 and 1020 milliseconds). These firing interval options are more tuned to the behaviors than are the acceleration options above. Within the 10 to 25 range usually used during any motion, 10 is the most commonly used at normal speeds. The 255 setting is used when the robot is stopped.

In all, 5304 actions are considered, followed by the previously described quadratic interpolation being applied to the velocity and offset dimensions. Interpolation is not performed along the other dimensions because not enough points are considered.

5.4.4 Iterated Split Space

We refer to this algorithm as an iterated split space search because it allows for multiple loops through each dimension of the action space. The search begins by setting the *best action* to the last chosen action vector, and the search considers the utility of this action with all possible acceleration settings. Each resulting action modification is presented as a candidate to the voting arbiter. If, at any time, the utility of this candidate action is better than that of the last action, the *best action* is set to this modified action.

After all accelerations have been checked, a copy of the *best action* has its offset velocity dimension searched. If, at any time, a new highest utility is found, the associated action becomes the *best action*. The sonar interval and velocity values are looped through next, respectively. The order in which the dimensions are checked was chosen arbitrarily.

Once every dimension has been looped through, the loops start over with acceleration settings to see if any behaviors have changed their opinions of acceleration after other changes have taken place in the other dimensions.

Once no changes have been made after checking all four dimensions iteratively, the search process is complete and the *best action* is chosen as the robot's action. This type of search reduces the number of actions tested from 15 billion to 1447 per iteration. Although this search algorithm does not necessarily stop after one iteration, for this application no changes were ever made after the first iteration. A partial second iteration is necessary to verify this, however. In general, the iterated search allows solutions that at least satisfy a Nash equilibrium (treating the different action dimensions as distinct agents).

5.5 Search Enders

The search process terminates either when (a) the time limit has expired, (b) an action of utility 1.0 is found, or (c) the search ender's own condition is reached. Three search enders have been implemented: *end at aspiration*, *end at max next aspiration*, and *end at 1.0*.

As the name indicates, *end at aspiration* ends the search when an action is found whose utility is equal to or greater than the current aspiration. When satisficing, it makes sense to stop at the aspiration level, but doing so can result in an undesirable effect. At times, it may be impossible to meet the aspiration level, causing the aspiration to drop. If an action that exceeds the aspiration is rarely found in the allotted time, the general tendency would be for the aspiration to gradually decline. The other two search enders are designed to address this potential difficulty.

End at max next aspiration calculates what the aspiration would be at the next time step if an action of utility 1.0 were found during this time step. If an action is found that meets the calculated aspiration, then the search ends.

End at 1.0 only ends the search when the time limit expires or an action valued at 1.0 is found. This ensures that any time a search is performed, the only way to achieve a non-optimal result is to run out of time.

5.6 Summary

In all, one hijacker, one vetoer, and six voters have been implemented to allow a Nomad Super Scout robot to navigate hallways and seek goals. Three searchers, one directed and two undirected, have been implemented to test traditional action selection in voting systems against Bruvo 1's more general paradigm. Three simple

search enders have also been implemented to test trade-offs between CPU usage and action utility. In chapter 6, we compare how well the system works across the different searchers and search enders.

Chapter 6

Results and Analysis

6.1 Objectives

In this chapter, we present results that show that the framework used by the Bruvo 1 architecture allows high utility actions to be found in less time than with traditional techniques. In particular, in section 5.4, we described three search algorithms and how they relate to the state of the art in action search techniques used by utilitarian voting systems. In this chapter, we analyze each of the search algorithms and how they compare from a satisficing anytime search perspective. This type of analysis shows how Bruvo 1 allows directed searchers, such as the implemented genetic algorithm, to outperform traditional search techniques.

In addressing how our framework meets the objective of finding high utility actions with less CPU usage, we also show general performance characteristics of the system. This gives a better understanding of the Bruvo 1 architecture as a whole and of the sample problem we have chosen for the purpose of testing the search framework.

6.2 Overview of Tests

To compare the different search techniques, we conducted a series of tests. In these tests, the three search algorithms discussed in chapter 5 were combined with the three search enders. For the tests, an initial aspiration of 0.95 and a change rate of 0.1 were used for equation 2.1. These were also compared against an unchanging aspiration of 1.0, which causes the agent to always search.

These tests were conducted in the environment diagrammed in figure 6.1. In these tests, the robot performs a clockwise lap through the hallways, by moving from one user-specified goal to another. When a goal is reached, it is removed, and the next goal is placed. These tests were conducted in both simulated and real-world environments.

The majority of results shown here are from a simulated world fashioned after the same kind of difficulties found in the real hallway environment. Figure 6.1 shows the simulated world used for the experiments. The simulated world allows easy comparison under controlled conditions, but limited experiments in the real world have been performed to validate that the system actually works and is consistent with simulated results.

Controlling the robot in the real world is more difficult than in the simulator because of imperfect actuators. More specifically, we not only have imprecise, noisy control of the actual robot's wheels, but also, due to weak motors, there is often a bias in the velocity of one or both wheels. We have simulated these real-world difficulties by adding suitably truncated Gaussian noise to each wheel. In the simulator, the velocity of each wheel could differ from the specified velocity by up to 1 inch/second; this accounts for imprecise and noisy velocity control. Additionally, the left wheel had up to 1 inch/second reduced from its speed, simulating a weak motor in one wheel.

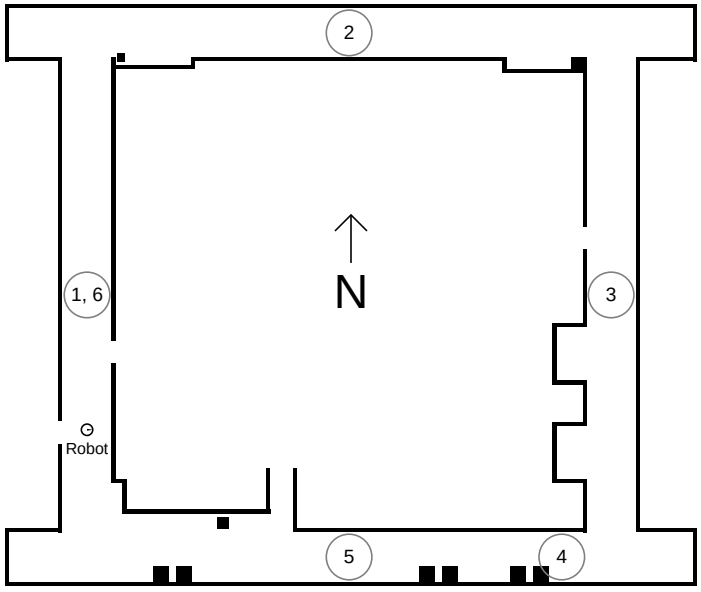

Figure 6.1: *The simulated hallway world used for the experimental results. Holes in the walls represent open doors to rooms which the robot does not enter. Block objects represent chairs and other obstacles. The circled numbers represent ordered goal locations. The robot is placed at the starting location and orientation.*

This is implemented by setting the mean and variance of the Gaussian distribution in proportion to the wheel's velocity.

There are two important differences between the tests performed in the simulated world and the real world. First, because the tests in the simulated world were performed on a much faster computer than the robot's, the time limit was reduced from 0.05 seconds (real robot) to 0.025 seconds (simulated robot). These time limits allow both the low resolution and split space searches to complete. Second, the dimensions (hallway length and width, positions of plants and chairs, etc.) of the simulated world differ slightly from the dimensions of the real world.

One final comment about the tests is in order. Goal locations were chosen for two

reasons: (a) to help the robot complete laps and (b) to allow different behaviors to be active at different times. Goals at hallway corners would be easier to seek, but less testing would be done of the autonomous wandering ability. With the goals at hall midpoints, both types of behavior are allowed to emerge. The midpoint goals still provide sufficient direction to help the robot complete laps. However, large cavities at the sides of the halls sometimes confuse the robot. Goal 4 was needed for this reason. Without it, the goal-seeking heuristic would try to reach goal 5 through the cavities in east hallway, sometimes indefinitely preventing its progress.

The majority of this chapter discusses results from the simulated world. Section 6.7 discusses how the real-world results compare with the simulated results.

6.3 Robot Performance Quality

6.3.1 Types of Quality

We are primarily concerned with the quality of results compared against the amount of processor usage. We address three different types of quality:

1. Observations of robot path characteristics, which represent a qualitative, human estimate of quality.

2. The utility of taken actions, which represents a behavioral estimate of performance quality.

3. The short-term and long-term consequences of taken actions, which represent a more objective measure of quality.

Section 6.3.2 discusses the general nature of paths taken by the robot. As for other types of quality, we present results in the form of mean utility, but without

more information, a utility measurement is not very useful. Because of this, we present relationships between utility and consequences in sections 6.3.3 and 6.3.4. Section 6.3.3 discusses short-term consequences (e.g., how far off is the robot's orientation at a given moment), and section 6.3.4 discusses long-term consequences (e.g., the time it takes to complete a lap).

6.3.2 Path Characteristics: Qualitative Results

In the simulator, each hall is 750 inches (62.5 feet) in length measured between the centers of the intersections. A square going around the hallways is 3000 inches in perimeter. The robot starts 213 inches south of the first goal. Counting rotation times as well, the robot could ideally perform a lap in about 130 seconds. In experiments, however, the best the robot could do was just under 160 seconds. Figure 6.2 shows a typical robot trajectory in the simulated world. Such indirect paths account for the extra 30 seconds that resulted.

The irregularities in the observed path primarily result when the *center in hall* behavior tries to accommodate changing hall widths. This behavior usually determines the robot's path because, during most of the lap, *seek goal* is inactive giving "don't care" votes of 1.0. One reason for this is that after a goal is found, the next active goal is around the corner, and a direct path is not possible. At the intersections, *seek goal* does take over, but usually it becomes inactive again quickly because of the necessity of negotiating through the nearby walls on either side of the robot.

One exception to this characteristic control occurs in the north hall, where the small obstacle at the inside of the northwest corner causes the robot to move somewhat toward the opposite wall before turning. Under these circumstances, the indented section on the south causes the wall to be so far away that *seek goal* stays in control.

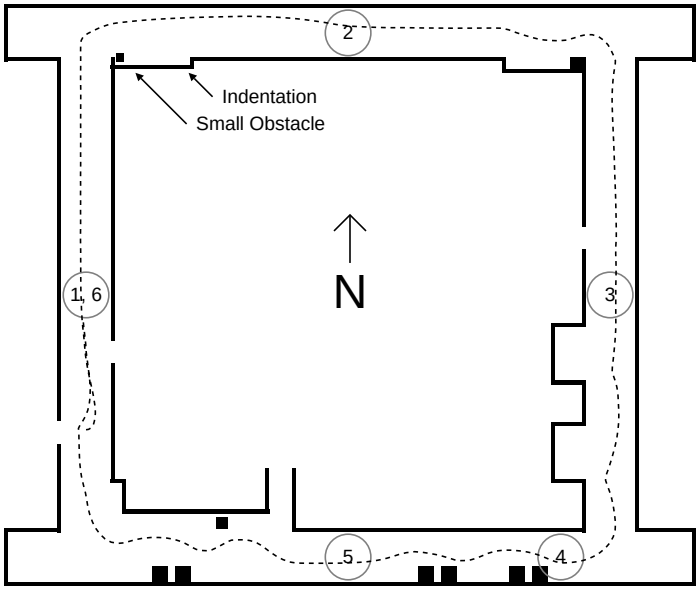

Figure 6.2: *Characteristic robot path. Here the genetic search was used with the aspiration search ender.*

It tries to orient directly towards the goal, but the weak left wheel keeps it from doing so. For that reason, it hugs the north wall.

A second cause of path irregularity is the weak left wheel, which causes the robot to tend slightly towards the outside wall during the whole lap.

Two final segments of the path deserve comment. First, the two large cavities toward the south end of the east hall cause the robot to veer significantly, due to *center in hall*. At the lower of the two indentations, the *turn* behavior helps the robot to return to its path. Second, the obstacle on the north side of west end of the south hall is also a place where *turn* is beneficially used to orient the robot towards an open path.

6.3.3 Utilities and Short-Term Consequences

The relationship between utility and short-term consequences is fairly direct. As shown in section 6.3.4, the difference in mean utility between the best and worst cases is on the order of 0.01. Utility differences of this magnitude can be easily translated into short-term consequences by considering what actions would be taken if all loss of utility were ascribed to a single voter. Table 6.1 shows such short-term consequences from a loss of 0.01 utility.

Behavior	**Action Dimension**	**Desired Value**	**Chosen Value**
Center in Hall	Offset Velocity	0.5 in/s	1 in/s
Move Forward	Translational Velocity	25 in/s	15.5 in/s
Regulate Speed	Translational Velocity	10 in/s max	11.8 in/s
Silence	Sonar Interval	1020 ms	492 ms
Seek Goal	Offset Velocity	0.5 in/s	0.8 in/s
Turn	Offset Velocity	2 in/s after 1 s	0.7 in/s

Table 6.1: *Sample effects of a loss of 0.01 utility on each behavior individually, assuming all utility were lost to a single behavior.*

6.3.4 Utilities and Long-Term Consequences

Long-term consequences are often more meaningful than short-term consequences because they show how small details affect the robot's performance overall. The primary long-term consequence we have analyzed in relationship to utilities is that of lap time.

In section 6.4, we compare lap times, utilities, and time spent searching for actions between the different search algorithms. Specifically, figure 6.4 shows mean utilities

for different configurations and figure 6.5 shows corresponding lap times. As can be seen, a small decrease in utility can result in a large increase in lap time. The correspondence is not completely predictable, since many issues may affect such long term consequences. Section 6.4 presents a more detailed analysis of these effects.

6.4 CPU Time, Utility, and Lap Time

Figures 6.3, 6.4, and 6.5 show average CPU time per decision step, average utility of actions taken, and average lap time, respectively. The different search methods are shown in different groups. For each group, results are given for the three search enders discussed in chapter 5 and for the constant aspiration of 1.0 discussed in section 6.2.

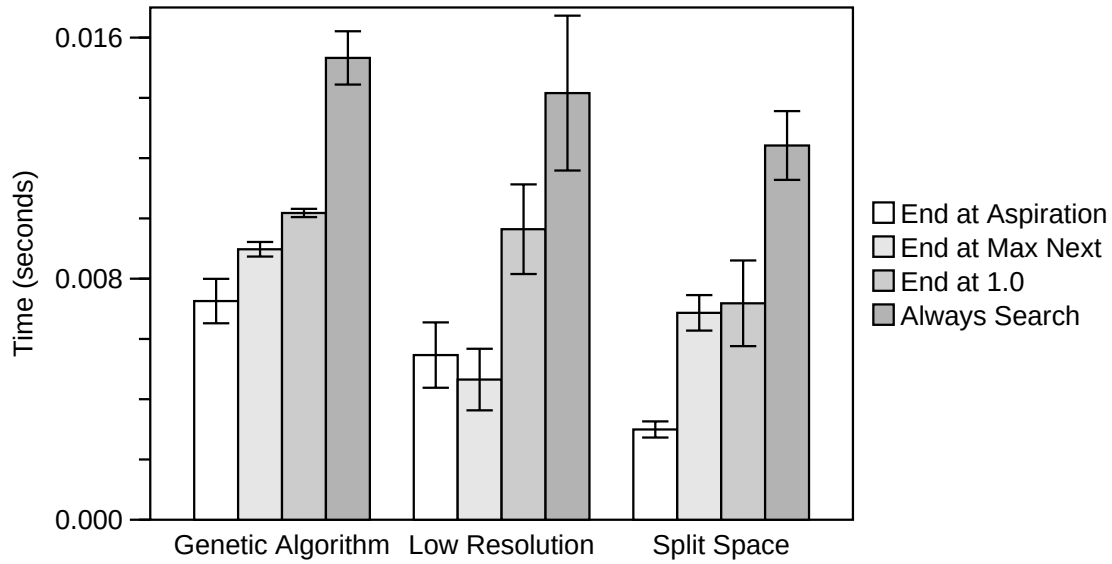

Figure 6.3: *Mean CPU time spent per decision step. Error bars represent 95% confidence intervals.*

The state of the art in utilitarian voting is represented by always searching (i.e., a constant aspiration of 1.0) with either the low resolution or split space searcher. It

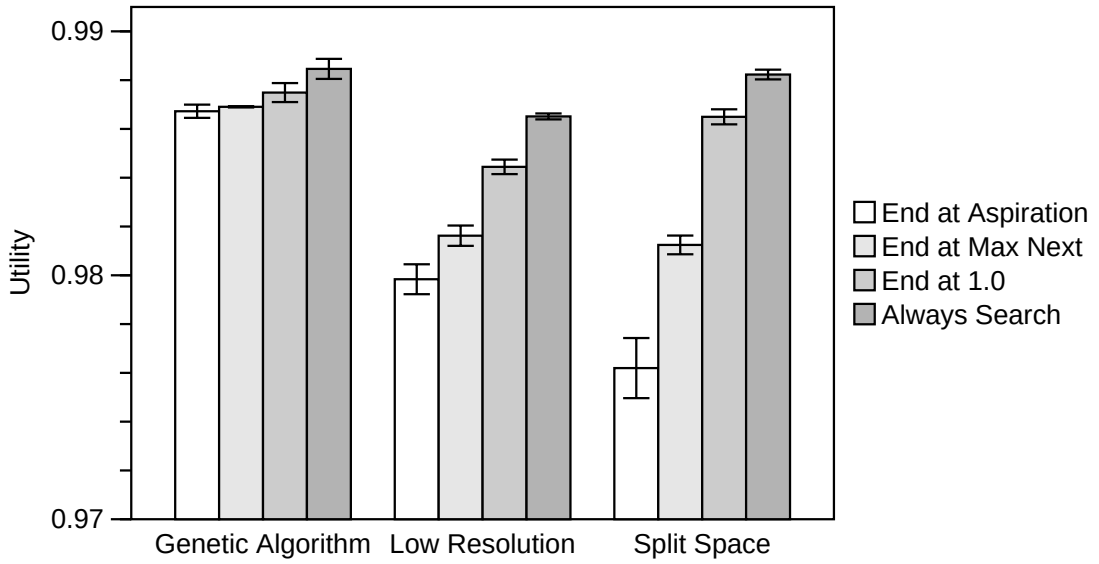

Figure 6.4: *Mean utility of taken actions. Error bars represent 95% confidence intervals.*

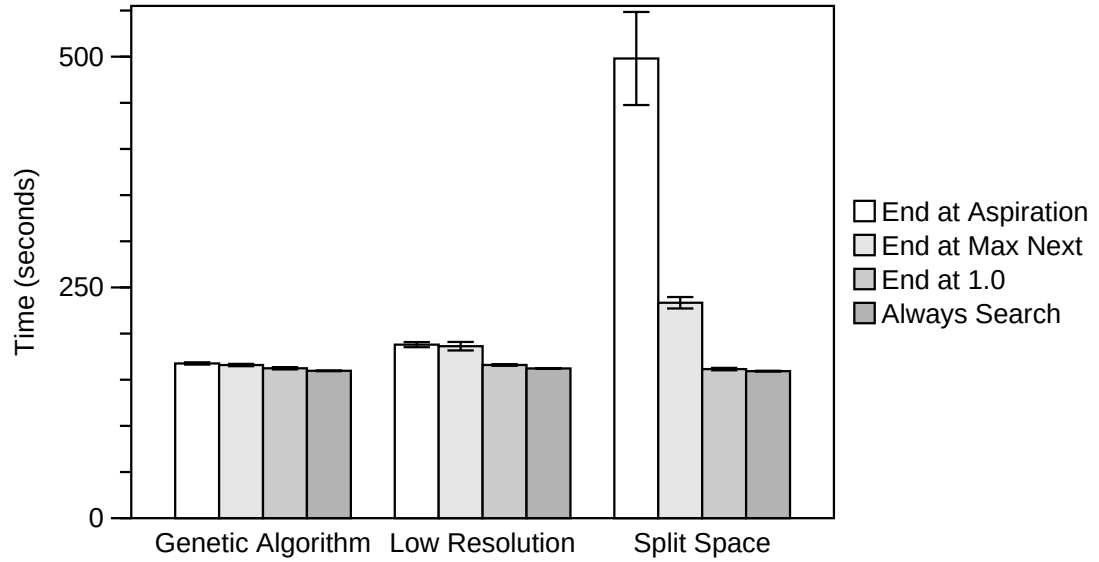

Figure 6.5: *Mean time to complete one lap. Error bars represent 95% confidence intervals.*

is clear that introducing any level of satisficing considered here significantly reduces CPU usage, as seen in figure 6.3. Specifically, the genetic algorithm using *end at aspiration* or the other two using *end at 1.0* have lap times that are no more than 6% slower than the fastest lap time (split space always searching). This small increase in lap time is accompanied by a decrease in CPU time of about 42% (for the genetic algorithm or split space searcher). More details are shown in table 6.2.

Searcher	**Configuration**	**CPU Time**	**Lap Time**
Genetic Algorithm	End at Aspiration	7.26 ± 0.74 ms	167.6 ± 1.2 s
Low Resolution	End at 1.0	9.64 ± 0.15 ms	166.0 ± 0.9 s
Split Space	End at 1.0	7.18 ± 0.14 ms	161.5 ± 1.3 s
Split Space	Always Search	12.42 ± 1.14 ms	159.3 ± 0.4 s

Table 6.2: *CPU time compared to lap time for different configurations. The bottom line, split space always searching, represents the best results for the current techniques normally used in utilitarian voting.*

Both low resolution and split space searches were designed to finish within time constraints, and they obviously perform fairly well when allowed to complete their search. On the other hand, they do not function well as anytime searches as indicated by the strong decrease of utility seen in figure 6.4 when the search is ended early.

When ending searches at the aspiration level, it is still desirable to overshoot rather than barely reach the goal. Whenever the aspiration cannot be met, it is lowered, and sometimes significantly. If the aspiration is often barely surpassed, the overall tendency is to reduce the aspiration. Both of the undirected searches suffer from these effects. The genetic algorithm tends to mix actions in such a way that when a new best action is found, it is often significantly better than the previously

best action. This allows the aspiration to be more strongly exceeded.

Another problem with the undirected searches results from their tendency to consider changes in only one dimension at a time. That is, at the inner loop of the low resolution search, only velocity options are changed. Different sonar firing intervals are looped through outside that, and so on. If an influential behavior cares enough about a particular velocity, its decision alone may make an action satisficing. This is especially prominent when the aspiration level is low. With the split space search, velocity is considered last, after *all* other aspects of the action have been fully decided. Since the most strongly weighted behaviors mostly govern orientation, they can easily cause an action to be satisficing before a good velocity has been found. At multiple points on the lap, this causes the robot to slowly back up for extended periods of time, resulting in the very slow lap times seen in figure 6.5.

The genetic algorithm makes use of suggestions and combines them. This means that it rarely considers highly undesirable actions at all. This seems to be the main reason why its performance remains high despite shortened searches.

On the other hand, when the genetic algorithm cannot find a satisficing action, it has no default ending point. Both other searches can actually perform full searches in slightly less time than permitted by the time limit, whereas the genetic algorithm always exhausts the entire time. When it cannot end a search short, it uses more CPU than the others. The split space search is fastest for the same reason. That is, its search space is the smallest, due to the additive nature of handling dimensions independently. On the other hand, as will be seen in section 6.5, significant further improvements in cutting off directed searches should be possible.

One additional point observable in figure 6.4 is that the low resolution search, even when carried to completion, does not quite reach the same utility levels as the other two searches. This seems to be a result of its inability to directly pinpoint ideal

values for each action dimension. The interpolation helps somewhat but does not entirely overcome the deficiency. Though hard to see in the bar plots, its best lap times are a few seconds behind the best times of the other searches.

6.5 Performance Profiles and Search Time

The CPU time and utility are direct results of the nature of the searches being performed. The performance profiles of the searches are shown in figures 6.6 and 6.7. Such profiles show how the utility increases with time for each search. Each profile depicts every search performed during one lap. Rather than show the overall trend, we have shown individual profiles so that the characteristics of the searches during each time are more visible. An optimizing search ender was used so no searches were ended early. To reduce the amount of data points, increases in utility of less than 0.01 are not shown.

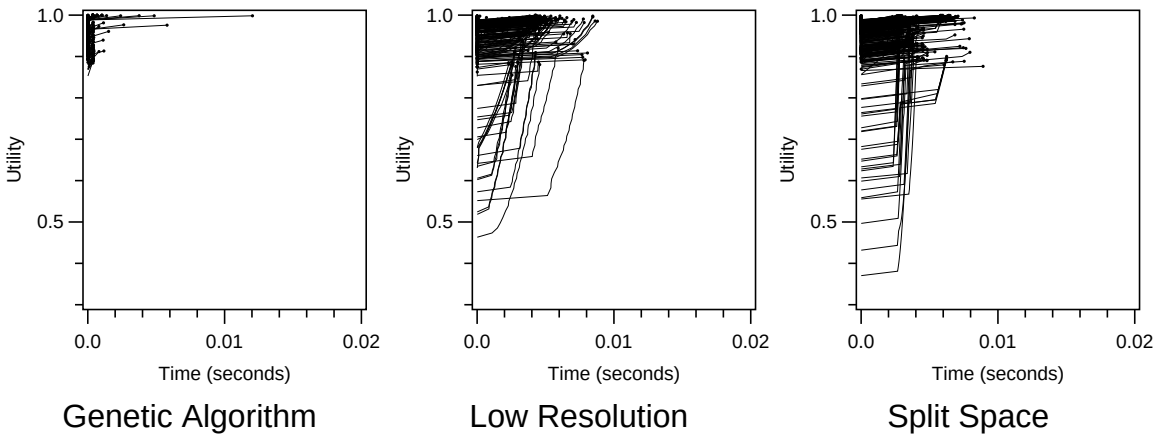

Figure 6.6: *Performance profiles for all three searchers.*

The genetic algorithm clearly performs a faster search than the other two methods; it reaches a high utility very rapidly. A primary reason for starting out so much

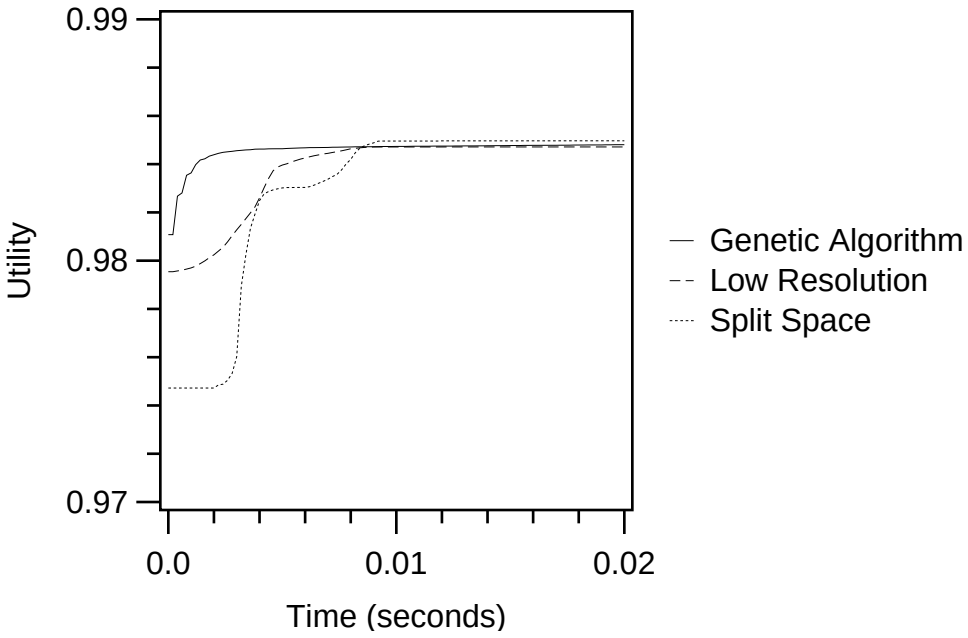

Figure 6.7: *Mean performance profile for each of the three searchers.*

higher is its use of behavior suggestions. Since all three searches at least default to the previously chosen action, utilities do not fall *too* low and often begin very high.

The undirected searches often exhibit a tendency to spend time searching in parts of the space that matter little, only to find an important part where every increment of a value raises the utility. This type of profile is very different from Zilberstein's condition of asymptotic search utilities in anytime algorithms [28]. This also shows again how easily these searches often just barely meet an aspiration level.

Of note for all three searches is that all significant improvement has taken place long before the time limit has expired. Specifically, the highest average ending utility is about 0.9850 with the split space searcher. This can be seen in figure 6.7. All the searchers achieve at least 0.001 less than this utility within the first few milliseconds of search. For the mean performance profiles shown here (recorded from one lap of the simulated world), the time to reach at least 0.984 utility is shown in table 6.3.

Searcher	**Time**
Genetic Algorithm	1.4 ms
Low Resolution	5.1 ms
Split Space	7.8 ms

Table 6.3: *Time for each search algorithm to reach 0.001 utility below maximum.*

For the low resolution and split space searchers, the time required to find actions of high utility is dependent on the order of action dimensions in the search loops. Better results could be obtained faster by reordering the loops, but the best order will always be dependent on the behaviors composing the system. On the other hand, the genetic algorithm has no such dependencies.

The performance profiles also show that the quality degrades more quickly for the undirected searchers. As stated in table 6.3, the genetic algorithm achieves within 0.001 utility of the highest achieved within 1.4 ms. At the same time within the search process, the low resolution and split space searchers are only within 0.005 and 0.010 of the best, respectively. We discussed in table 6.1 the consequences of a loss of 0.01 utility. The genetic search is clearly more suited to ending searches early.

A simple summary of the profile information is visible in figure 6.8, where clock time is shown, not CPU time[1]. The minimal search time refers to the time when either the best action found during the search was found or when the aspiration level was met. The actual search time is the time actually spent searching before such an action was decided on. In other words, when the aspiration level is not met, a full time-bounded search is performed without further benefits. The discrepancy

[1]Clock time can be more affected by other processes on the computer, but it is still mostly consistent and serves the additional purpose here of seeing how clock time compares to CPU time for these tests.

between the time spent searching and the minimal time needed show that the genetic algorithm could be dramatically improved if it were equipped with some ability to predict when the best action that will be found has been found already. In this case, the genetic algorithm search time could ideally be reduced by 9.78 times what we have achieved so far with aspiration-based satisficing.

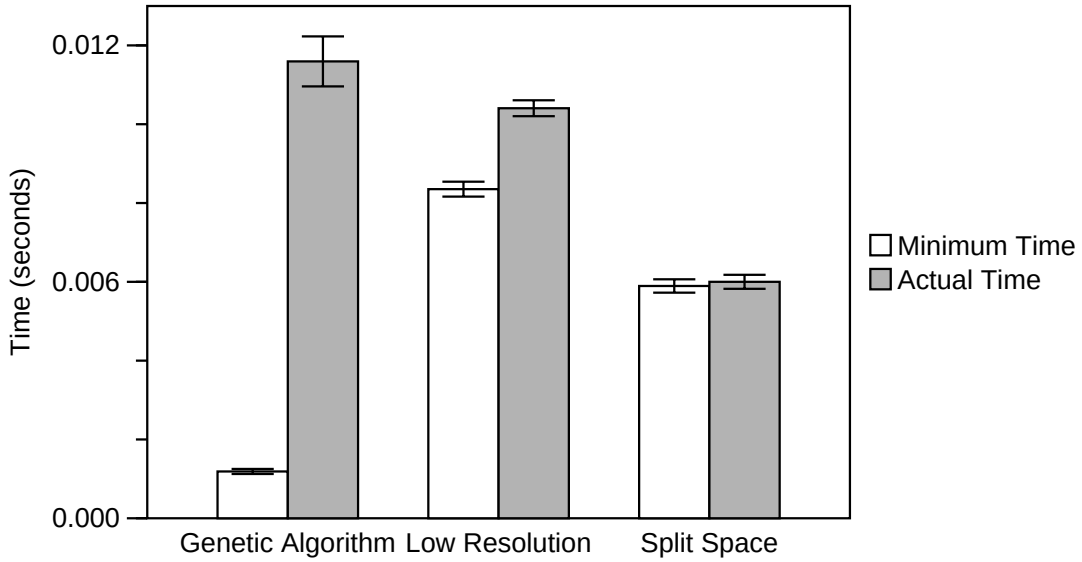

Figure 6.8: *Mean search time with the aspiration search ender. Error bars represent 95% confidence intervals.*

The significance of this analysis from an anytime perspective is that it makes it clear why the genetic algorithm performs better for shortened searches. It also becomes clear that the current status quo of low resolution and split space searching is not ideal for handling large action spaces. We have also seen that using other techniques for ending searches early may lead to even more reduction in CPU usage while not significantly lowering result quality.

6.6 Dimensional Dependencies

While a few of the behaviors consider multiple dimensions in their vote (via averaging), only one varies its decision based on the combination of dimension values given: the *avoid crash* vetoer. It allows long sonar firing intervals or translational velocity, but not both at once. Motion without environmental feedback could be unsafe.

This dependency causes a problem for the split space searcher. When the robot is stopped, it usually increases the sonar firing intervals. When a search for a new action is performed, the *silence* behavior will vote high for slow sonars. No other behaviors indicate a complaint since they are not concerned with sonars. Because of this, a long sonar interval is chosen. Afterwards, a velocity must be chosen. The *avoid crash* vetoer now vetoes any forward motion because slow sonars have been chosen. As a result, the robot stays stopped, and the sonars stay slow. Sometimes the robot eventually turns fast enough due to changing environmental conditions, *silence* speeds up the sonars, and the robot can move forward.

While specialized behaviors could be used to counteract this effect, increased system complexity would make such dependencies hard to find and predict. Because the genetic algorithm and low resolution searches consider whole actions at once, they do not suffer this effect. Additionally, the random nature of the genetic search quickly resolves such problems if they are encountered.

To overcome this problem for split-space search, we have applied a solution separate from the voting process; the robot moves forward when there is space to move, even if the behaviors did not vote for such an action. The results on split-space search reported in this chapter use this ad hoc fix. In general, such ad hoc fixes undermine the voting process as discussed in chapter 2. This ad hoc solution was only included here to allow consistent comparison of results from all the search methods.

6.7 Real-World Validation and Comparison

In the real world, running on the real robot and the robot's CPU, the system also performs sufficiently well from a designer perspective. Because of the slower CPU, the time limit is set to 0.05 seconds as discussed in chapter 5.

One major difficulty arises with seeking goals. The dead reckoning available in the robot sensors is imperfect, so position and angle information is faulty. When major errors in such information occur, goal seeking capability is very limited. Fortunately, errors are usually small enough that a human can move goal locations to allow the robot to complete its task.

To make automated laps possible and generate real-world results for comparison with the simulated results, we used a priori knowledge of the halls to correct sensor data. Landmarks such as hallway corners were used to recalibrate position information during the laps.

Figure 6.9 shows a sample lap performed by the real robot. Note that the locations of the walls and the path of the robot are only approximate, because the position corrections applied at each time step are not always accurate. The sharp spike in the path at goal 4 is a result of such approximation errors. Other, more subtle errors exist along the entire path.

An additional goal was required to help the robot through the real world. In the south hall, the real environment is far more congested (with chairs and other obstacles) than in the simulator. This added congestion often prevented normal wandering skills from reaching the southwest corner after reaching goal 5. The new goal, goal 5b, in the southwest corner keeps the robot oriented the right direction.

We performed four separate laps using the genetic algorithm searcher and the aspiration search ender. The minimum lap time for a simple rectangular path in the

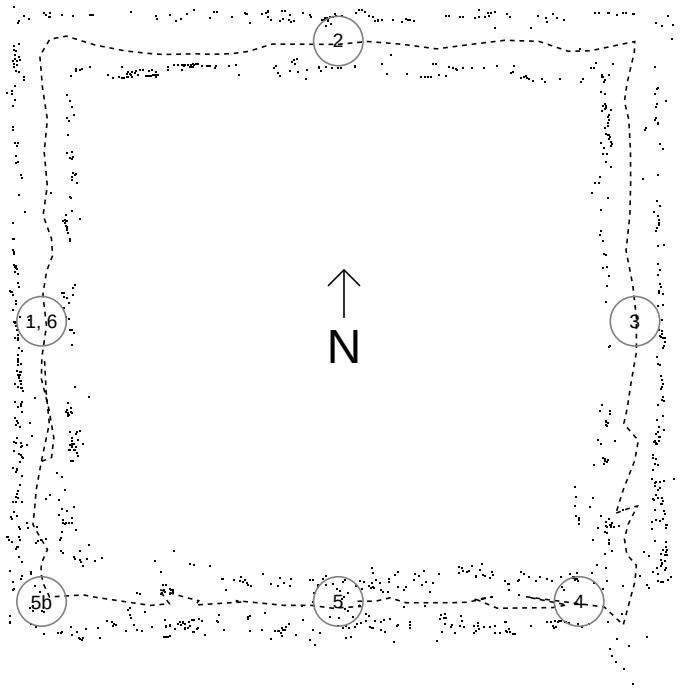

Figure 6.9: *Path followed by the real robot. The scattered dots represent sonar readings, the dashed line represents the path taken by the robot, and the circled numbers represent goals.*

real halls is about 146 seconds, because the real halls are slightly longer than in the simulator. The mean lap time for the actual robot was 220.3 ± 7.9 seconds. This means that the actual lap was about 51% longer than the ideal. In the simulator, the same configuration took about 27% more time than the ideal. Figure 6.10 shows how the ratio compared against the low resolution and split space searches as well.

The excess lap time for the real robot results from the added complexity of the real world. When the robot has trouble staying centered in the halls, it moves slower because the headway used by *regulate speed* takes into account obstacles at diagonal angles from the robot. The narrow, congested south hallway especially adds to such problems. In the tests, the south hall took about two thirds more time than the west

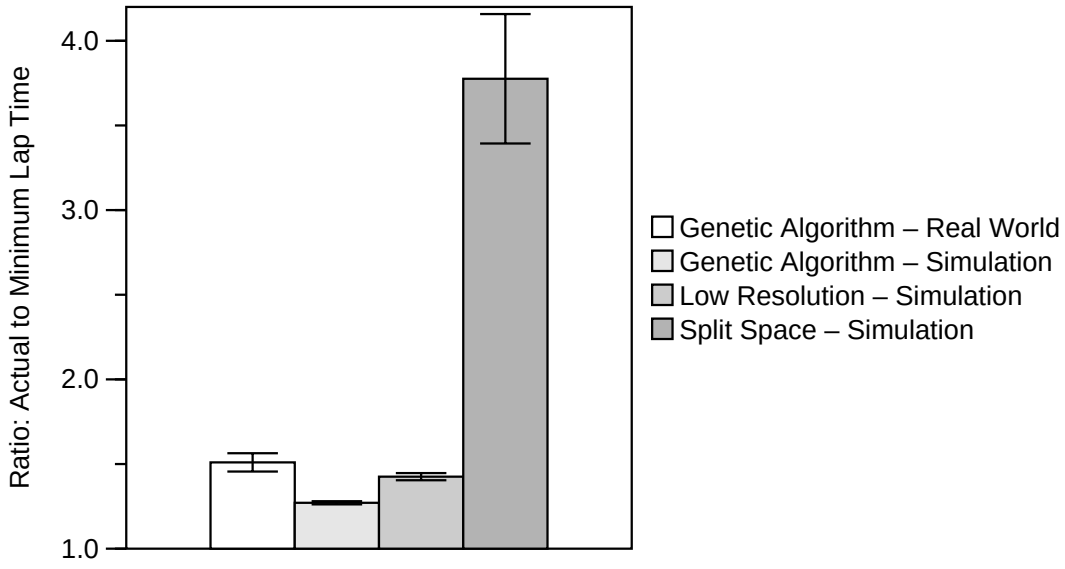

Figure 6.10: *Mean ratio of actual lap time to minimum lap time with the aspiration search ender. Error bars represent 95% confidence intervals.*

and north halls, which were easier to navigate than the other two halls.

The utility was also lower for the real robot than in the simulation, probably due to the difficulty of operating in the real world, especially the congested south hall. Figure 6.11 shows comparative utilities. While not statistically significant, the mean utility for the real robot was probably somewhere between the utility of the low resolution and split space searchers in the simulated world. Note that the utility loss cannot be directly translated into required lap time. In many cases in the real world, using the genetic algorithm search, most of the utility loss was likely due to *move forward*'s inability to proceed at full speed.

Search time also deserves some analysis. In the simulator, the total time spent searching was approximately 9.8 times more than the minimum time required to locate a satisficing action or the best action that was found during the search; see figure 6.8. In the real world, that ratio was approximately 8.2. Though the difference

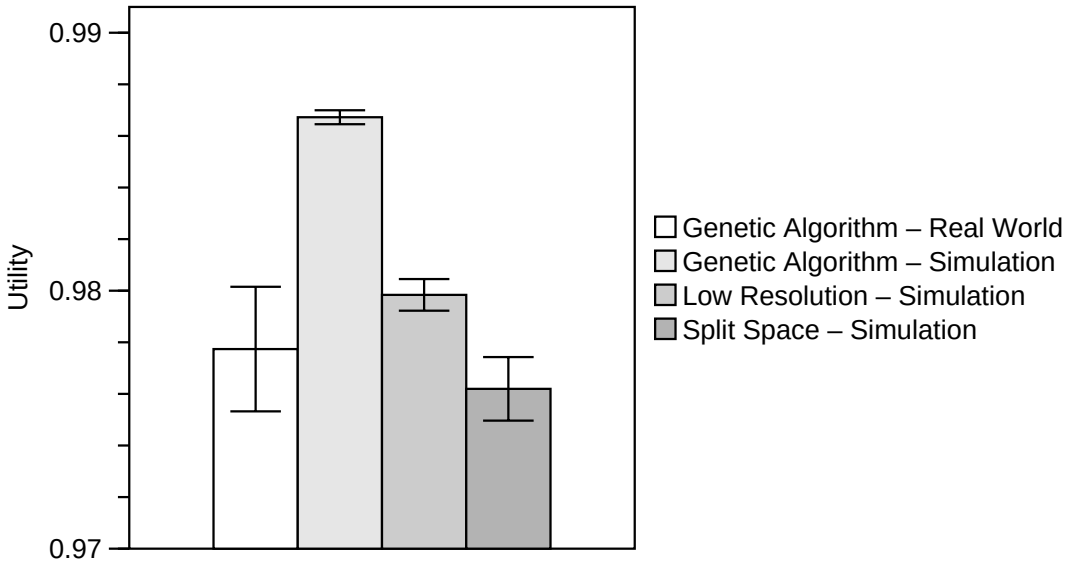

Figure 6.11: *Mean utility with the aspiration search ender. Error bars represent 95% confidence intervals.*

between the two is statistically significant, the ratio is still large and suggests that predictive search ending would still lead to significant reduction in CPU usage.

In all, the real robot performed sufficiently well to perform laps and avoid obstacles under ordinary circumstances. When we combine the simulated results, which imply that the satisficing anytime framework allows directed searches to outperform traditional voting action search techniques, with the real-world results, which demonstrate that Bruvo 1 based on the genetic algorithm produces a functional robot control system, we conclude that Bruvo 1 accomplishes its intended objective.

6.8 Emergence and Overrides

6.8.1 Qualitative Observations

We have addressed the overall expected behavior of the system and its relationship to the different search algorithms, but we still need to address unexpected emergence and how effective the overrides were in handling undesirable emergence. Emergence in this application results primarily from two main sources:

1. Random search order. Occasionally, the genetic algorithm, by chance, chooses a near optimal action without even knowing if an improvement lies nearby.

2. Searches shortened by satisficing. When a search is cut short because some cutoff utility has been reached, the result can be highly emergent, especially in the case of undirected searches.

The genetic search derives most of its emergence from the first cause, and the other two searches are affected by the latter.

Because most of the behaviors used in this application base more of their vote on the offset velocity, the translational velocity is more likely to show emergent characteristics. When confronted with a wall, the robot often backs up, while the "optimal" action may be to rotate in place. Allowing a bit of emergence can help or hurt the quality of short and long term consequences, but overrides provide a mechanism for countering unacceptable effects.

Originally, we did not design the *avoid crash* hijacker and vetoer to take reverse motion into consideration. It happened rarely, and the reverse motion was usually very slow and short-lived. However, as more behaviors were added and the system became more complicated, a bad form of emergence resulted. When the aspiration was low, the low resolution and split space searches sometimes found actions with good

orientation but whose velocities were in reverse at full speed. In other words, the first velocity tried with the proper orientation was "good enough". Once the *avoid crash* behaviors took reverse motion into consideration, only safer backups were allowed. Again, the genetic search has not been observed to attempt such extreme actions.

Also important is that the *avoid crash* hijacker makes decisions quickly enough to not significantly reduce time available for the rest of the system. Since Bruvo 1 requires all hijack decisions to be made at the beginning of the time step, it places a theoretical limitation on what kinds of algorithms can be used for hijack behaviors without reducing the decision time too much. In this case, however, there is no observable problem, since most productive search is completed long before the time limit expires.

Interestingly, no hijacks were ever performed based on collision predictions. Apparently, the other behaviors entirely prevented such situations. On the other hand, when wandering, the robot sometimes does get within 10 inches of a wall while moving very slowly. Hijacks were observed under such circumstances.

6.8.2 Empirical Results

In addition to the qualitative observations previously discussed, we performed one specific empirical test of the overrides. We placed the real robots in an environment with walls arranged like those illustrated in figure 6.12. We started the robot at three different locations (in the middle and near each side wall) facing toward the dead end. 30 tests were done with and 30 without the overrides active. Each test concluded when the robot had either collided with the wall or had turned around and was successfully leaving the dead end. Without overrides, the robot collided with a wall 13 times (or in about 43% of the trials), while it only crashed 3 times (or in 10%

of the trials) with overrides in place.

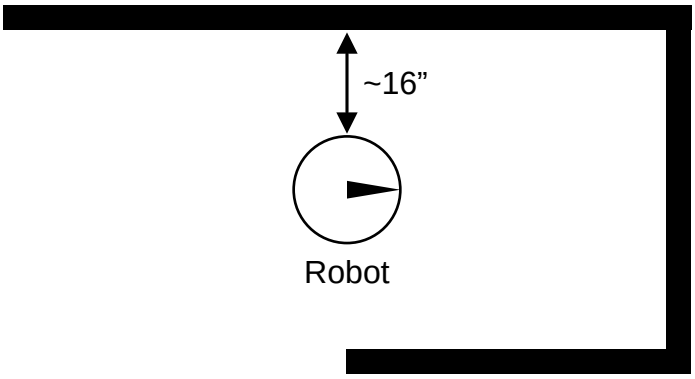

Figure 6.12: *Approximate representation of real-world environment used for testing veto and hijack behaviors.*

Strongly weighted voters with the same algorithms as the hijacker and vetoer may have prevented crashes as well, but having a high weight does not *guarantee* that a voter's desires will be recognized. In a system with many more behaviors, it may become very difficult to predict when standard voting would fail, whereas hijacking provides a modular way to enforce such guarantees.

6.9 Summary

By combining utilitarian voting with a satisficing, anytime action search framework, Bruvo 1 allows high quality actions to be found with less CPU usage than action search techniques previously used with voting. In section 6.4, we showed that for our sample problem addressed in this chapter we could reduce computation by up to 42% while increasing the time to perform a lap by only 6%. When using the genetic algorithm searcher, we also account for the dimensional dependencies to which the common split space searching paradigm is subject (see section 6.6). Using a directed

search, such as a genetic algorithm, also allows higher utility actions to be found in less time.

In section 6.5, we showed that the genetic algorithm also finished most productive searching over three times faster than the other searchers. At present, the aspiration-based satisficing is not sufficient to take full advantage of the speed of the genetic algorithm; we have shown that, ideally, the time spent searching with the genetic algorithm could be further reduced by close to ten times without loss in result quality. By this we have shown that directed search algorithms are more applicable to problems where CPU usage is important and action spaces are large.

Of additional significance, veto and hijack override mechanisms seem both theoretically and empirically to protect against undesirable emergence. In this application, the only hijacker makes decisions quickly enough to avoid significantly reducing decision time available to the rest of the system.

In all, Bruvo 1 shows that utilitarian voting can be used effectively on systems with multiple, dependent degrees of freedom and high resolution actuators. Because of this, the benefits of voting may be applied to more general robot control problems than have been addressed in previous voting applications.

Chapter 7

Conclusion

7.1 Contributions

This thesis has contributed a method for using utilitarian voting for autonomous mobile robot control with large action spaces in real-time. We have presented a generalization of the search process traditionally used for finding optimal actions in utilitarian voting systems, and we have shown how directed searches allow voting to be used effectively and uniformly for full control of robot systems. Aspiration-based satisficing used in conjunction with anytime search methodologies reduces CPU usage while producing a sufficiently high quality of robot performance. Specifically, we have shown a decrease in CPU usage of up to 42% while increasing the time required to complete a sample task by only 6%. We have done this while accounting for dependencies between action dimensions. Better predictive search ending may further reduce CPU usage by almost ten times without affecting result quality, but we have not achieved such reductions thus far.

We have also presented the concept of override mechanisms used in various control systems and have shown how vetoes and hijacks provide a means for constraining

undesirable, emergent behaviors. Such overrides provide direct control of the robot while retaining the component-based philosophy of voting mechanisms.

These principles have been implemented with the Bruvo 1 architecture for control of real-world mobile robots.

7.2 Disadvantages

While emergence is sometimes good and vetoes permit system designers to avoid unacceptable consequences, a lack of control does result from unpredictable search patterns. The increased emergence can be disconcerting where strong predictability is required or when certain performance levels must be guaranteed.

In many cases, voting with aspiration-based satisficing can cause a false sense of security. It can be easy to think that the system just works, when in fact there are multiple concerns unaccounted for. Certain overrides may be required, but the need may not be seen until the robot is already in action. This indicates a need to carefully plan ahead for full guarantees of reliability, but this planning seems to negate much of the ease of development provided by "plug and play" utilitarian behaviors.

Aspiration levels have another problem when situations change suddenly. When a goal suddenly appears or disappears, a behavior like *seek goal* can dramatically alter the utilities achievable. When this happens, a gradually adapting aspiration may not apply. Suddenly raising utilities equally for all actions tends to make poor actions seem satisficing, because the aspiration level is still adjusted to the lower level.

A specific disadvantage of Bruvo 1 is the inability of hijackers to interrupt the decision process. If they cannot decide quickly, they increase the starting time required for each decision step. Allowing them to interrupt the decision process would require that they be queried in the search loop or have an interrupt-driven I/O mechanism.

This would slow the search, but the added time might be negligible.

One additional limitation of the current implementation is that not much improvement has been shown over the CPU usage of traditional low resolution and split space searches. Better ability to end searches early would be required to fully take advantage of directed search methods. However, we have laid a foundation for future research in this area where real-time searches could be performed with significantly less CPU usage.

7.3 Future Directions

7.3.1 Predictive Search Ending

As can be seen in chapter 6, the genetic algorithm is able to find good actions in far less time than is spent searching. The other search algorithms also end productive searching far before the time limit.

One important area of research would be in search time reduction. An ability to predict when little further improvement were likely would help reduce CPU usage significantly. Techniques could range from simple heuristics [10, chapter 13] to statistical analysis of performance profiles.

7.3.2 Stepless Deliberation

Nomad Super Scouts have high latency in performing actions and reading sensors. Such latency makes the use of time steps natural for a decision making process.

Other situations, whether physical or artificial, may have different limitations and different requirements. If distinct sensors operated at different speeds or if actuators and sensor operated much more quickly, it could be useful to have the deliberation

process decoupled from the control thread or threads of environment communication.

Such a system would need to have an action available at any time rather than performing anytime searches at discrete time steps. Stepless deliberation would alter the use of aspiration-based satisficing. Sleeping until a time limit expired would be meaningless. Rather, some way of regulating smaller sleeps or altering thread priority would be needed.

Research along these lines would be needed to apply the concepts of this thesis to such problems.

7.3.3 Context-Dependent Aspiration Levels

When changes occur in the environment that cause behaviors to vote very differently from one moment to the next, aspiration levels do not always apply. Future research in techniques for handling such situations would be very important. Simply having a high aspiration change rate may be sufficient in some circumstances, but in others, more complicated handling may be necessary.

One possibility could be to have the aspiration rate adjust to environmental cues, so that when the robot returns to a previously encountered environment, it recalls the utilities it achieved in similar situations.

7.3.4 Multiple Aspiration Change Rates

Modifying the aspiration change rate alters the memory of the aspiration level. That is, high rates cause only recent utilities to be remembered, whereas low rates allow utilities from the past to have more effect.

In research so far, this memory does not seem to have a consistent, direct effect on utilities and search time. Of interest would be research not only into the effects of

this memory, but also into the effect of multiple change rates.

If the change rate for increasing the aspiration were different from that for decreasing it, there would be a direct, noticeable effect on the time spent searching. The exact results, however, especially for different types of situations, is harder to predict. Further research into general effects of aspiration adaptation would be of value.

7.3.5 Additional Behaviors

In addition to improving existing behaviors, additional behaviors could be introduced to improve and extend robot capabilities, including the following examples:

1. Avoid Past

   The robot can occasionally get stuck in "box canyons" or repeatedly try to reach a goal in the same unsuccessful fashion. A simple, low-level behavior designed to avoid past locations could help overcome such difficulties [2].

2. Plan Landmark Path

   Goal seeking could be dramatically improved in complicated situations with simple forms of map-making. By remembering hall lengths and landmarks that represent decision locations (i.e., forks in the road), the robot could more quickly navigate to goal points [16].

7.3.6 Abstracted Sensors and Actuators

If specific information about sensors and actuators were abstracted from the behaviors, the behaviors could be built to work on robots with different physical configurations. For instance, instead of knowing about 16 sonars that return values between 0

and 255, a behavior could ask the distance at any angle using floating-point precision. Such information may need to be interpolated and would give less direct access, but it would provide a proper abstraction for generic behaviors.

Actuators could also be abstracted, although bounds may still need to be known. For instance, translational velocity may still range from -25 to 25 in/s, but actuator precision would be hidden. Again, for example, floating-point numbers could be used. Bruvo 1 theoretically allows for such continuous (to CPU precision) action spaces, and empirical tests would be valuable.

Such abstracted systems go some way toward providing action-based voting with the kinds of generic behaviors hypothesized by Rosenblatt for state-based voting [21, page 57].

Bibliography

[1] Ronald C. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *IEEE International Conference on Robotics and Automation*, pages 264–271, 1987.

[2] Tucker Balch and Ronald C. Arkin. Avoiding the past: A simple but effective strategy for reactive navigation. In *IEEE International Conference on Robotics and Automation*, pages 678–685, 1993.

[3] Pierfrancesco Bellini, Mario Adres Bruno, and Paolo Nesi. Verification of external specifications of reactive systems. *Systems, Man, and Cybernetics*, 30(6):692–709, 2000.

[4] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Transactions on Robotics and Automation*, 2(1):14–23, 1986.

[5] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence Journal*, 47:139–159, 1991.

[6] Barry Brumitt and Martial Hebert. Experiments in autonomous driving with concurrent goals and multiple vehicles. In *IEEE International Conference on Robotics and Automation*, 1998.

[7] Thomas Dean and Mark Boddy. An analysis of time-dependent planning. In *National Conference on Artificial Intelligence (AAAI)*, pages 49–54, 1988.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley Longman, 1995.

[9] Gerd Gigerenzer and Daniel G. Goldstein. Reasoning the fast and frugal way: Models of bounded rationality. *Psychological Review*, 103(4), 1996.

[10] Gerd Gigerenzer, Peter M. Todd, and the ABC Research Group. *Simple Heuristics That Make Us Smart*. Oxford University Press, 1999.

[11] Michael A. Goodrich and Thomas J. Palmer. Discretionary behavior switching: Analysis and synthesis results. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 1330–1335, 2000.

[12] Michael A. Goodrich, Wynn C. Stirling, and Erwin R. Boer. Satisficing revisited. *Minds and Machines*, 10:79–110, 2000.

[13] Rajeeva Karandikar, Dilip Mookherjee, Debraj Ray, and Fernando Vega-Redondo. Evolving aspirations and cooperation. *Journal of Economic Theory*, 80:292–331, 1998.

[14] Jonas Karlsson. *Learning to Solve Multiple Goals*. PhD thesis, University of Rochester, 1997.

[15] Isaac Levi. *The Covenant of Reason: Rationality and the Commitments of Thought*. Cambridge University Press, 1997.

[16] Curtis W. Nielsen. Communication and map building in a cyber team system. Master's thesis, Brigham Young University. In preparation.

[17] Paolo Pirjanian. *Multiple Objective Action Selection & Behavior Fusion using Voting*. PhD dissertation, Department of Medical Informatics and Image Analysis, Aalborg University, 1998.

[18] Paolo Pirjanian. Satisficing action selection. In *SPIE Conference on Intelligent Systems and Advanced Manufacturing*, pages 153–164, 1998.

[19] Jukka Riekki. *Reactive Task Execution of a Mobile Robot*. PhD dissertation, University of Oulu, 1999.

[20] Thomas Röfer and Axel Lankenau. Ensuring safe obstacle avoidance in a shared-control system. In *International Conference on Emergent Technologies and Factory Automation*, pages 1405–1414, 1999.

[21] Julio K. Rosenblatt. *DAMN: A Distributed Architecture for Mobile Navigation*. PhD dissertation, Robotics Institute, Carnegie Mellon University, 1997.

[22] Julio K. Rosenblatt. Behavior-based control for autonomous underwater exploration. In *IEEE International Conference on Robotics and Automation*, 2000.

[23] Julio K. Rosenblatt and David W. Payton. A fine-grained alternative to the subsumption architecture for mobile robot control. In *IEEE/INNS International Joint Conference on Neural Networks*, volume 2, pages 317–324, 1989.

[24] Herbert A. Simon. A behavioral model of rational choice. *Quarterly Journal of Economics*, 69:99–118, 1955.

[25] Sanjiv Singh, Reid Simmons, Trey Smith, Anthony Stentz, Vandi Verma, Alex Yahja, and Kurt Schwer. Recent progress in local and global traversability for planetary rovers. In *IEEE International Conference on Robotics and Automation*, 2000.

[26] Roland Stenzel. A behavior-based control architecture. In *IEEE International Conference on Systems, Man, and Cybernetics*, pages 3235–3240, 2000.

[27] Shinzo Takatsu. Decomposition of satisficing decision problems. *Information Sciences*, 22:139–148, 1980.

[28] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

[29] Shlomo Zilberstein. Satisficing and bounded optimality. In *AAAI Spring Symposium on Satisficing Models*, pages 91–94, 1998.