REINFORCEMENT LEARNING IN THE JOINT SPACE: VALUE ITERATION IN WORLDS WITH CONTINUOUS STATES AND ACTIONS

by

Christopher Kenneth Monson

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
April 2003

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Christopher Kenneth Monson

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

______________________ ______________________________

Date Todd S. Peterson, Chair

______________________ ______________________________

Date Michael A. Goodrich

______________________ ______________________________

Date Michael D. Jones

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Christopher Kenneth Monson 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

Todd S. Peterson
Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce
Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

REINFORCEMENT LEARNING IN THE JOINT SPACE: VALUE ITERATION IN WORLDS WITH CONTINUOUS STATES AND ACTIONS

Christopher Kenneth Monson

Department of Computer Science

Master of Science

Continuous space reinforcement learning algorithms frequently fail to address the possibility of a continuous action space, presumably because of the difficulty of discovering the best action for a particular state. This can, in some cases, severely limit the ability of a learning algorithm to tackle some common problems where different portions of the state space require distinct action granularity. Naïve action discretization does not suffice for problems of this nature, so traditional reinforcement approaches that consider only the continuous state space fail to solve these kinds of problems.

JoSTLe (Joint Space Triangulation Learner) addresses the need for a reinforcement learning approach that can handle a continuous action space by means of intelligent discretization. It employs the variable resolution discretization techniques of Muños and Moore [MM02], but in an augmented “joint” space, one that includes actions as well as states.

The algorithm is shown to work on a problem that requires the treatment of a continuous action space, as well as one that does not. The efficacy of the algorithm as well as its sensitivity to parameter tuning are shown through mathematical arguments and experimental data.

ACKNOWLEDGMENTS

It is largely thanks to Todd Peterson that I entered graduate school for the third time and finally made it into a successful experience. Without his encouragement, friendship, and patience, this work would not be possible.

Special thanks also go to David Wingate for the initial ideas that began this process, for many lengthy and productive brainstorming sessions, and for the implementation of some of the algorithms upon which this work is based. His keen insight, boundless creativity, and giving nature have been tremendous assets to me.

My gratitude also goes to Mike Goodrich, who spent many hours painstakingly working through my math, methods, and overall communication approach. His ability to spot places that needed work and his willingness to exert a great deal of capable effort has greatly increased the robustness of this work.

Of course, my wife deserves the most thanks of all for her constant support and comfort during the difficult times, and for her willingness to put her life on hold during this process. She is a great blessing in my life.

Contents

1 Introduction **1**

1.1 Continuous Extensions 4

1.1.1 Value Function Approximation 4

1.1.2 Spatial Discretization 5

1.2 Adding Completeness 7

2 Related Work **9**

3 JoSTLe: Joint Space Triangulation Learner **13**

3.1 Value Iteration 15

3.1.1 Linear Simplex Interpolation 16

3.1.2 Boundary Maxima Proofs 21

3.1.3 Execution and Practical Considerations 26

3.2 Joint Space Splitting 27

3.3 Parameters 30

3.3.1 Parameter Values 30

3.3.2 A Note on Parameter Sensitivity 32

3.4 Computation of Intersections 32

3.4.1 Projected Intersection Proof 34

3.4.2 Generation of Projected Simplices 41

3.5 Convergence 44

4 Delaunay: The Road Not Taken 45

4.1 The Incircle Property 45

4.2 Boundary Triangles 46

4.3 Existence 48

4.4 Uniqueness 49

4.5 Projection and Value Iteration 50

4.6 Implementation 51

4.7 Other Triangulations 52

5 Illustrative Problems 53

5.1 1-Dimensional Golf 53

5.2 Mountain Car 55

6 Results 57

6.1 Complexity 60

6.1.1 Time 60

6.1.2 Space 62

6.2 Parameter Sensitivity 62

6.3 Additional Experiments 64

7 Future Research 68

7.1 Time Complexity 68

7.2 Post-Pruned KD-Trie 68

7.3 Splitting Criteria 69

7.4 Looser Splitting Constraints 69

8 Conclusion: Contribution to Computer Science 70

A Kuhn Triangulation 71

B Correction to Incremental Deletion in Delaunay Triangulations 73

B.1 Finding the Ears 73

B.2 Which Ear to Add 75

Chapter 1

Introduction

The idea of learning computers has captured the attention of a number of researchers. Though it is debated whether or not practical success can be attained in the development of programs that learn, the admittedly small successes achieved in the growing field of machine learning seem to indicate a promising future.

Control problems appear to be a good fit for learning machines because many very difficult nonlinear control problems arise in common and realistic situations. Many of these problems are difficult to solve directly or are completely unsolvable using current direct approaches. Even so, a number of them lend themselves to a high level description of desired outcomes. Theoretically, a computer could use such a high level description, coupled with the right approach, to discover a solution to the problem.

Reinforcement learning (RL), on the surface, appears to be just such an approach to developing intelligent problem-solving programs. The attraction is easily explained: a very simple program is given access to the environment[1] and minimal information about its task, and then proceeds to learn proper behavior through a series of experiments.

[1] An "environment" is anything that has state and can be acted upon by the agent.

The information given to a reinforcement learner typically consists of its current state and an immediate consequence for an action taken from that state. The consequence includes the new state of the environment and a reinforcement for the action. If it has reached a goal state, it gets a positive reinforcement. If it makes a mistake, it may receive a negative or very low reinforcement. Given solely this information, the learning program can develop a model of behavior that maximizes its rewards over time.

Since a number of control problems lend themselves to formulations of this nature, the RL approach is very appealing. While reward structures may be difficult to craft for some problems, a large class of problems lend themselves to the generation of these structures. For these problems, it is generally easy to obtain the information needed to help a reinforcement learner on its way to success, though it may be much more difficult to find the optimal solution directly.

As a RL agent explores its environment, it may or may not receive rewards for certain actions. In the event that reward information is obtained, that information is propagated back to the originating state and action. This information eventually propagates through the agent's model of its rewards (which may be incorporated into a model of the environment or not, depending on whether a model-based or model-free approach is used), assigning a value to every action at every state. This information allows the agent to choose the action from every state that is most likely to maximize its rewards.

Because of the need to propagate information throughout a model of the environment, RL (in its most common form) is well suited to discrete problems. An RL agent will typically store information in the product space of states and actions, keeping information about present and discounted future rewards in a bucket for each state-action combination. This stored information is then used to propagate time-discounted reward information back to other buckets and to generate a best action policy (denoted $\pi^\star(\mathbf{s})$ for each state. The process that accomplishes this in model-based[2] settings is called *Value Iteration* [KLM96, AS97].

Value iteration as an algorithm is simple. An agent takes an action $\mathbf{a}$ from a state $\mathbf{s}$ and receives information from the environment about the resultant state $S(\mathbf{s}, \mathbf{a})$ and reinforcement $R(\mathbf{s}, \mathbf{a})$. It then uses that information to find the value $V(\mathbf{s}, \mathbf{a})$ of taking $\mathbf{a}$ from $\mathbf{s}$ by taking into account potential future rewards. Associated with this process is a discount factor $\gamma \in [0, 1)$, which represents how much an agent values future rewards in comparison to immediate rewards. In practice, values are updated by means of dynamic programming. In discrete deterministic worlds, the following equation is used:

$$V(\mathbf{s}, \mathbf{a}) = R(\mathbf{s}, \mathbf{a}) + \gamma \max_{\mathbf{a}'} V(S(\mathbf{s}, \mathbf{a}), \mathbf{a}').$$

The update equation above is most easily implemented in a discrete model of the value function for the environment. Applying it to continuous environments requires some special consideration and results in extensions to the approach [KLM96,Moo91]; straightforward storage and lookup techniques become impossible in the continuous domain, where the product space of continuous states and actions is uncountable even if the space is bounded.

[2] In a model-based approach, the agent explicitly builds a model of its environment, then uses that model to discover the policy. In a model-free approach, the agent attempts to learn its policy without explicitly building a model of the environment. Sometimes the line between these two ideas is somewhat blurred, as is the case here.

1.1 Continuous Extensions

A number of methodologies have been proposed as means of providing the needed extensions to allow value iteration to function in a continuous setting. These methodologies employ various different techniques. Regardless of the extensions used, however, a good continuous RL algorithm should have the following characteristics:

- Robust: converges to a good policy in various situations
- Complete: allows for continuous states and actions
- Correct: generates a useful approximation to the true value function

As will be seen, no known algorithms have all of these characteristics simultaneously. To aid in the discussion of the relative merits of existing techniques, the existing approaches are divided into two basic categories: *value function approximation* and *spatial discretization.*

1.1.1 Value Function Approximation

Global function approximators have been used in an attempt to learn the environment's continuous value function $V(\mathbf{s}, \mathbf{a})$ over the entire state and/or action space using a finite representation. Among the many different approximators used are neural networks, linear regression models, and polynomial approximators [KLM96, BK93].

These techniques have been used with some success in specific problem domains. They provide advantages in that uncountable continuous spaces are represented in a finite and well-understood way. Even so, these techniques have some problems. In the case of a neural network representation of a fully continuous environment, for example, it is very difficult to find an action that yields the highest reward without searching the entire (infinite) action space at each state. Additionally, many of these methods not only fail to learn the value function correctly [BK93], they learn something so wrong as to be completely unusable [BM95].

Though function approximators are *complete*, they are neither *correct* nor *robust*. Many implementations are not even *complete* (they do not allow for continuous actions) though the underlying approach may allow for completeness.

1.1.2 Spatial Discretization

Because of their shortcomings, function approximators are not generally a good choice when applied to the value function over the entire state-action space at once. When applied to smaller discrete regions of space, however, a class of function approximators called "averaging approximators" can not only be useful, but provably reliable [Gor95]. Thus, when combined with spatial discretization, the function approximation approach becomes much more interesting as a technique for using RL in continuous settings.

Many spatial discretization techniques exist, but the most practical have proven to be *variable resolution discretization* techniques. Variable resolution techniques generally involve learning an efficient discretization of the state space. In contrast, *fixed resolution discretization* is the naïve partitioning of space into segments of equal size. Because the number of divisions of space always increases exponentially with dimension when using a fixed resolution (a phenomenon often referred to as "The Curse of Dimensionality"), variable resolution is often preferred.

Variable resolution techniques attempt to sidestep the curse of dimensionality by focusing discretization resources on the portions of space with the most complex and "important" features, while leaving the rest of the space less finely discretized. The definition of "important" varies widely among methods (e.g., value interpolation error, value change variance, influence, longest edge). Much of the work in this area has been done by Muños and Moore, and makes use of variable resolution hypergrids, limited to environments whose behaviors are accessible[3] [Moo91, Moo94, MM02, Dav97, Rey99].

Other approaches involve less strict limitations on the structure of the model, like unconstrained triangulation [MM98, Muñ96, Muñ97], multigrids [HA98], and more generalized and composite methods (e.g. polynomial fitting, heuristic boundary search, and others) [BK93, Atk94, CM, MAS95, MSBL98]. None of these other approaches appear to perform better than the simple variable resolution techniques employed by Muños and Moore, and often add complexity without improving accuracy, speed, or memory usage (see Chapter 4 for one such example).

In fixed and variable resolution techniques alike, each segment of space is treated as though it correctly approximates all of the points that it contains. Thus, each segment is a discrete unit of space, and (slightly modified) discrete RL techniques can then applied to learn the value function. A great deal of research has been done in this area with some good success.

The variable resolution techniques applied to date are both *correct* and *robust* in the sense that for particular problem domains, they can be proven to converge to the true value function, given enough resources. None of them, however, have satisfactorily addressed the issue of a continuous action space, either ignoring it entirely [MAS95, MSBL98, MM98, Moo91, MM02] or severely limiting its representation [BK93], and are therefore not *complete*.

[3] It is required that the learning agent have access to the actual results of taking an arbitrary action from an arbitrary state.

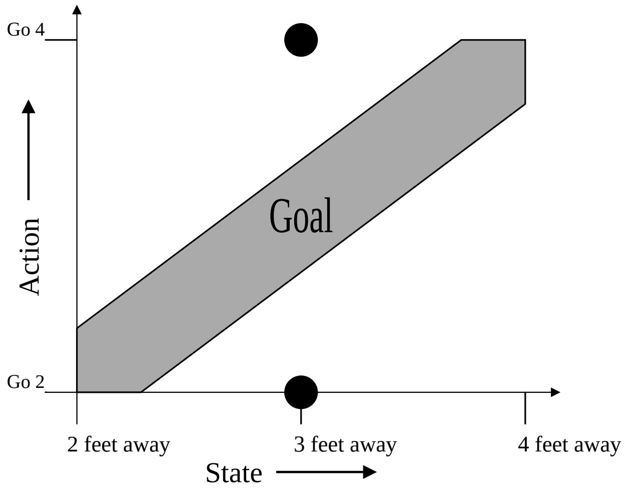

Figure 1.1: A naïve discretization of the action space can easily miss areas of high reward. The shaded area is an area of high reward, and the dots represent the actions available from the state "3 feet away". No matter how fine the state discretization, a reinforcement learner limited to a fixed action discretization will be unable to take the correct action from this state since only "Go 2" and "Go 4" are available.

1.2 Adding Completeness

Extending a variable resolution approach to accommodate continous actions presents some interesting problems. For example, how is the policy $\pi^\star(\mathbf{s}) = \arg\max_{\mathbf{a}'} V(\mathbf{s}, \mathbf{a}')$ computed? Methods that require actions to be discrete can simply search the list of available actions at a given state to discover which action yields the highest value. Since the policy changes over time (as it converges), it is important that the *current* action space be searchable at any given state.

In a continuous action setting, this can appear to be a crippling problem. The action space, even if it is bounded, is infinitely large, and therefore not searchable using the brute force iteration methods often employed by discrete action learners.

Though many interesting problems can be solved with a small and discrete action space[4], a number of problems are not well suited to discrete action spaces. Problems

[4] Minimum time control problems can be solved with "bang-bang" control, or control that only

where accuracy is required in a one-shot fashion (e.g., shooting basketballs, playing golf, throwing darts) are very common in the real world, and do not mix well with techniques which treat the action space as discrete or which discretize it naïvely. Obtaining accurate reward information in these problems is difficult for these approaches, since a naïve discretization can easily miss areas of high reward, especially when surrounded by large areas of low reward. This idea is illustrated in Figure 1.1 where the only actions available are "Go 2" and "Go 4". Those actions are unsuitable for the state "3 feet away".

Not only can a discrete action model not allow particular actions to be *taken*, it also cannot *know anything about them*, including information about their potential rewards. Without adequate reinforcement information, value iteration is doomed to fail, since at the heart of the approach is the ability to discount rewards into the past. If areas of reward are never found, the policy can never become optimal.

Even in light of its current shortcomings, variable resolution discretization is still a good idea. A continuous space is handled much more easily by an algorithm when it has been discretized. What is not always appropriate is focusing our attention solely on the state space and leaving the action space discretized *naïvely.* It turns out, fortunately, that the same approach to discretizing the state space can be extended to handle intelligent discretization of the action space as well, which is the focus of this thesis, where the JoSTLe (Joint Space Triangulation Learner) algorithm is presented.

involves the extrema of the action space [BH69]. For example, the Mountain Car problem is a minimum time control problem whose optimal solution requires only two actions: "full forward" and "full reverse" [MM02].

Chapter 2

Related Work

This thesis builds on existing variable resolution techniques, extending them to allow for a continuous action space. These extensions effectively add the missing *complete* characteristic to some techniques that are already *correct* and *robust*. In order to accomplish this, it addresses the problems of reward location and action selection as discussed in 1.2.

An understanding of the workings of JoSTLe is most easily obtained by first understanding its predecessor: the variable resolution discretization approach of Muños and Moore [MM02] (hereafter referred to as MM). A short summary of how MM works is given next, after which the extensions that make JoSTLe work will be discussed.

To solidify the concepts and help with the clarity of the discussion, 2-dimensional spaces will be used as examples. It should be understood that none of the concepts described here are limited to these lower-dimensional spaces. Indeed, the algorithm extends trivially to higher dimensionalities.

The basic idea behind MM is to begin with a very rough discretization of the state space (e.g., a single rectangle that covers the entire space) and to refine portions of it by progressively splitting interesting rectangles in half. The corners of the rectangles

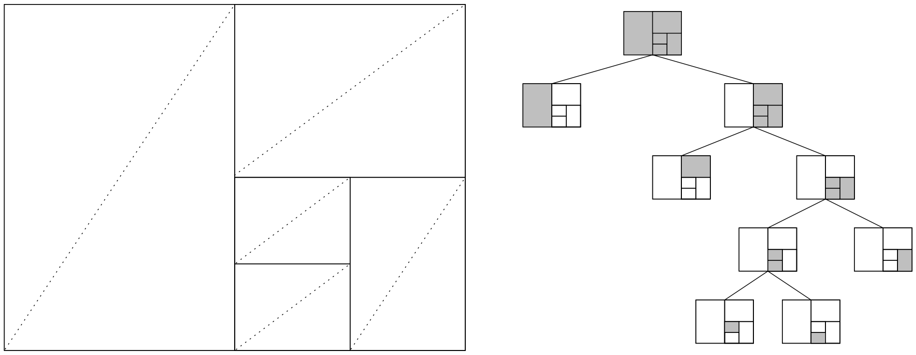

Figure 2.1: A discretization of a 2-dimensional state space and its corresponding kd-trie structure. Each node represents a region of space which may be subdivided. The Kuhn Triangulation of each leaf node is also shown.

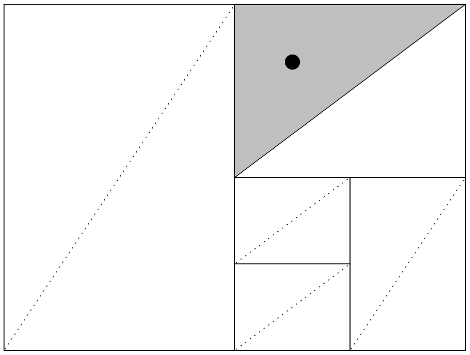

Figure 2.2: Interpolation of a point is performed by finding the relevant rectangle and then finding the appropriate triangle within it. Once found, the barycentric coordinates of the triangle allow interpolation to be performed for the point in question.

are vertices in the state space. Naturally, the vertices may be used to store any useful information, including values for each action that can be taken from that state. Because the space is split perpendicular to dimensional axes (in this case, either perpendicular to the x or y axis), the discretization is efficiently represented and queried in a kd-trie structure (Figure 2.1).

The corners of each rectangle (where information is stored) represent the actual experience or knowledge of the learning algorithm. At the corners, the values are assumed to be correct. Any value that is needed from within a rectangle is then obtained by means of interpolation.

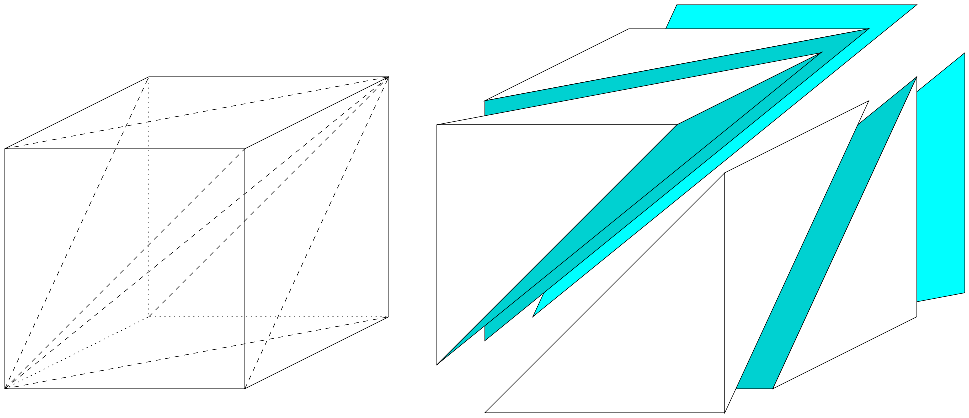

Figure 2.3: The Kuhn Triangulation of a 3-dimensional cube. Kuhn Triangulations split each d-dimensional rectangle into $d!$ simplices.

Since multilinear interpolation is expensive when compared to simplex interpolation [Dav97], each rectangle is split into triangles. Interpolation of an interior point is then done using the barycentric coordinates of the relevant triangle. Figure 2.2 shows a point in a triangle. The value of this point may be obtained via linear interpolation over the vertices of the shaded triangle.

The triangulation used is a Kuhn Triangulation, which effectively splits each d-dimensional rectangle into $d!$ triangles. An example of a 3-dimensional triangulation is shown in Figure 2.3. The factorial explosion of triangles is not a problem for MM because they are not explicitly stored. The properties of Kuhn Triangulations allow the appropriate triangle (which may be viewed simply as a strict subset of the corners of a rectangle) to be efficiently computed for any given point in the space.

Kuhn Triangulations have several other interesting and useful properties, including the ability to efficiently find the barycentric coordinates of interior points. This property allows for very efficient interpolation. For a more detailed treatment of how Kuhn Triangulation is used, the reader is referred to Appendix A and [MM02].

The discretization of the state space allows value iteration to be performed. In MM, value iteration is performed for each corner. Each action is chosen in turn and

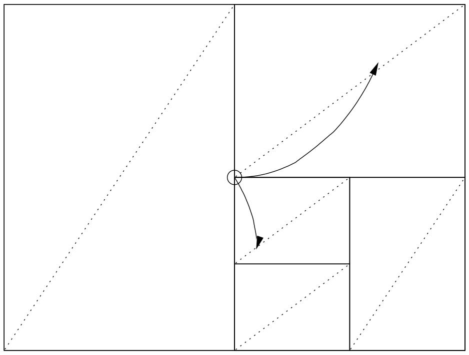

Figure 2.4: Value iteration is shown for a given point. Each action is taken from the point until the new state enters a new triangle. Then the value is interpolated in that new triangle and discounted back. Rewards are integrated over each trajectory.

taken until the trajectory has exited the current triangle as shown in Figure 2.4. Rewards are integrated over the trajectory, and the value of the point is updated by the value iteration update equation

$$V(\mathbf{s},\mathbf{a}) = \int_0^{\tau_{\mathbf{s},\mathbf{a}}} \gamma^t R(\mathbf{s}(t),\mathbf{a})\, dt + \gamma^{\tau_{\mathbf{s},\mathbf{a}}} \sup_{\mathbf{a}'} V(\mathbf{s}(\tau_{\mathbf{s},\mathbf{a}}),\mathbf{a}') \tag{2.1}$$

where $\mathbf{s}$ is the state represented by the corner's coordinates, $\mathbf{s}(t)$ is the state after taking action $\mathbf{a}$ for time t, $\tau_{\mathbf{s},\mathbf{a}}$ is the total length of time for which action $\mathbf{a}$ is taken from initial state $\mathbf{s}$ to get to a new triangle, $\gamma \in [0,1)$ is a discount factor, $R(\mathbf{s},\mathbf{a})$ is the reward for taking $\mathbf{a}$ from $\mathbf{s}$, and $V(\mathbf{s},\mathbf{a})$ is the value of a particular action and state combination.

Value iteration is performed until convergence. Once the values have converged, the model is tested for rectangles that need to be split in order to improve the discretization of the value function. Value iteration and splitting are performed alternately until the model reaches some satisfactory performance level. A number of different splitting criteria are employed by MM, the discussion of which is beyond the scope of this document. For full details, see [MM02].

Chapter 3

JoSTLe: Joint Space Triangulation Learner

MM works with a given *a priori* discretization of the action space and dynamically learns the appropriate discretization of the state space. JoSTLe uses data structures and techniques similar in many ways to those of MM but applies them in a novel way to the "joint space", which is the product space of states and actions (Figure 3.1). The dimensionality of the joint space is $d = d_s + d_a$, where d_s is the state dimensionality and d_a is the action dimensionality.

Figure 3.1 illustrates an important point. States in the joint space are not simply points as they are in the state space. They are represented as a *set* of points that satisfy a *state space constraint*. Thus, in the figure a state is not a point, but a line. Conversely, as a single point in the state space represents a state, in the joint space it represents both a state *and* an action.

The JoSTLe algorithm is simple in principle: value iteration is performed over all points in the space, then the space is split where necessary. The process is repeated until the value function is accurately represented or until some maximum splitting

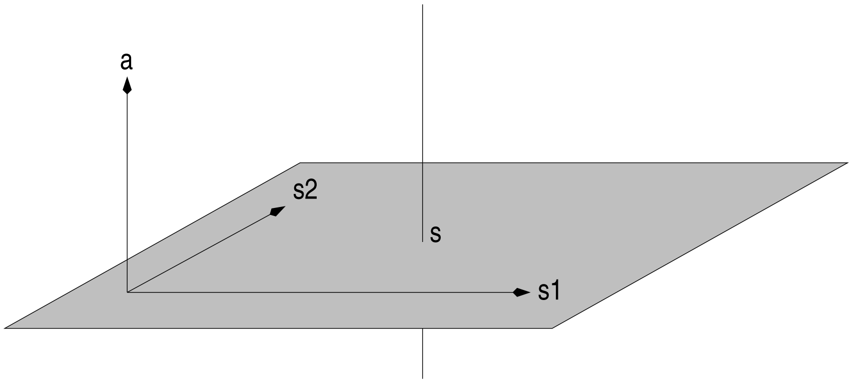

Figure 3.1: The Joint Space. The state space is now just a plane in the overall joint space. The action in this example is shown as an extension of the plane into a third dimension. In the joint space, a state is no longer a single point, but a set of points in the action space (shown here as a line).

criterion is reached. The basic algorithm follows.

JoSTLe: The Basic Algorithm

1. Taking each corner $\mathbf{v}$ in turn, decompose it into state $\mathbf{s}$ and action $\mathbf{a}$. Take action $\mathbf{a}$ from state $\mathbf{s}$ over time until the new state $\mathbf{s}'$ does not intersect any rectangles containing $\mathbf{v}$. Update the value of $\mathbf{v}$ with the following equation until values have converged:

$$V(\mathbf{s},\mathbf{a}) = \int_0^{\tau_{\mathbf{s},\mathbf{a}}} \gamma^t R(\mathbf{s}(t),\mathbf{a})\,dt + \gamma^{\tau_{\mathbf{s},\mathbf{a}}} \sup_{\mathbf{a}'} V(\mathbf{s}(\tau_{\mathbf{s},\mathbf{a}}),\mathbf{a}') \tag{3.1}$$

2. Evaluate each rectangle for splitting. A rectangle R is split if the maximum interpolation error within it is above a threshold ϵ (the interpolated value is given as $\widehat{V}(\mathbf{s},\mathbf{a})$):

$$\epsilon \leq \sup_{(\mathbf{s},\mathbf{a})\in R} \left| V(\mathbf{s},\mathbf{a}) - \widehat{V}(\mathbf{s},\mathbf{a}) \right| \tag{3.2}$$

Choose the dimension in which to split by evaluating which split will yield the new rectangle with the smallest maximum error. When all relevant rectangles have been marked, split.

3. Repeat steps 1 and 2 until the model is satisfactory.

At the heart of the algorithm is the alternation between value iteration and refinement. Value iteration is performed until convergence, after which every rectangle in the kd-trie is evaluated and possibly marked for splitting. Once all rectangles have

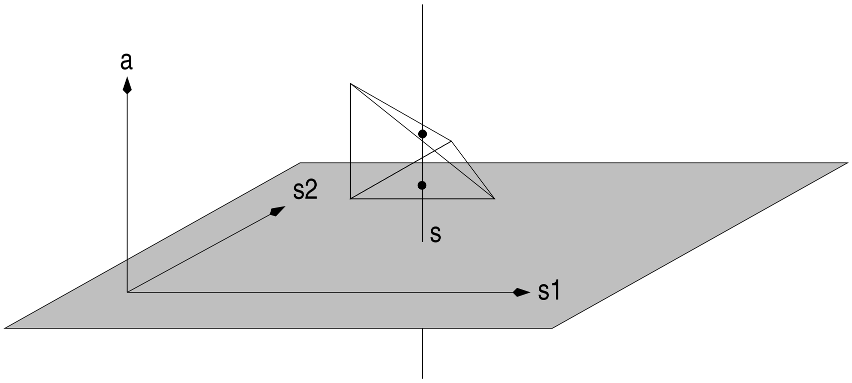

Figure 3.2: The state line crosses a tetrahedron at two points, both lying on boundary facets. Since interpolation is linear, only these points need to be considered as possible maxima.

been evaluated, all of the marked rectangles are split, and the whole process starts again. It terminates when the algorithm fails to mark any rectangles for splitting.

The details of value iteration and splitting, in addition to other algorithmic concerns like convergence, are discussed in the next sections.

3.1 Value Iteration

Equation 3.1 is simple to compute with a discrete action space. In a continuous action setting, however, the *sup* operator is problematic. In order to perform value iteration, the maximum-valued action must be found at the terminal state $\mathbf{s}(\tau_{\mathbf{s},\mathbf{a}})$. This state is no longer just a corner (0-dimensional space) in the joint space, but a hyperplane (d_a-dimensional space).

JoSTLe, like MM, uses Kuhn Triangulation within each rectangle to facilitate interpolation and point location. Since the space is discretized and value interpolation is linear within each simplex, it is possible to exchange the problematic *sup* operator of (3.1) for a more tractable *max* operator over a finite set of actions. This is done by making use of some convenient properties of linear interpolation.

Consider Figure 3.2. The state line passes through a tetrahedron in the joint space. This tetrahedron may be one of many which the state line passes through, but one is enough to illustrate the idea. Since the values at the interior of the tetrahedron are interpolated linearly from its vertices, the maximum of those values along the state line will occur at only one of two points (unless the interpolated value inside the tetrahedron is uniform, in which case the maximum occurs *at least* at those two points), both of which are at the intersection of the state line with a boundary facet of the tetrahedron.

Any given state will cross a finite number of tetrahedra at a finite number of boundaries. Since the maximum must occur at the boundary of a linearly-interpolated simplex, only these intersections need to be considered for value iteration and policy extraction.

The claim that a maximum must occur at a boundary is proven next. To prove this claim, mathematical backing is first developed for linear simplex interpolation. This is then used in the proof that maxima must occur at boundaries of the simplex. Once the proof is complete, the (then justified) algorithmic details of value iteration will be given.

3.1.1 Linear Simplex Interpolation

The d-dimensional simplex interpolation of the value of a point $\mathbf{p}$ can be found by noting that the $d + 1$ points of a d-dimensional simplex can be used to form a set of d basis vectors.

The vectors are formed by picking one vertex of the simplex to be the origin and labeling it $\mathbf{v}_0$. All other vertices are labeled arbitrarily as $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_d$ as in Figure 3.3. Each vertex has an associated *value*, written as $f(\mathbf{v}_i)$, or simply f_i in abbreviated notation. The goal is to interpolate over the vertices to find the value at point $\mathbf{p}$, denoted $f_{\mathbf{p}}$. The vertices are used to form basis vectors for a new Euclidean space. The vectors radiate out from $\mathbf{v}_0$ to the other vertices and are defined as $\vec{\mathbf{v}}_i = \mathbf{v}_i - \mathbf{v}_0, 1 \leq i \leq d$. A vector to point $\mathbf{p}$ is also defined such that $\vec{\mathbf{p}} = \mathbf{p} - \mathbf{v}_0$.

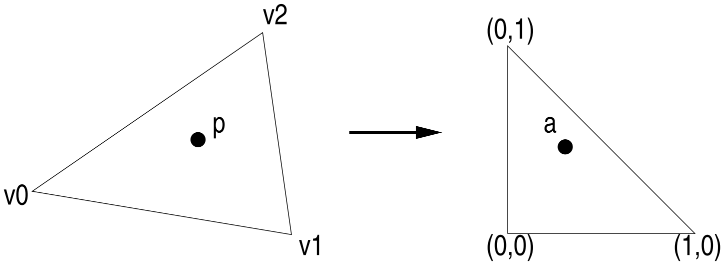

Figure 3.3: A point in a simplex and the same point in the space spanned by the vectors of that simplex. In the new space, the simplex is a set of unit vectors along dimensional axes.

The vector $\vec{\mathbf{p}}$ is transformed into the space spanned by $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_d$ through a standard coordinate transform, where $\vec{\mathbf{v}}_i$ and $\vec{\mathbf{p}}$ are row vectors:

$$\mathbf{a} = \vec{\mathbf{p}} \begin{bmatrix} \vec{\mathbf{v}}_1 \\ \vec{\mathbf{v}}_2 \\ \vdots \\ \vec{\mathbf{v}}_d \end{bmatrix}^{-1} . \tag{3.3}$$

This transform applies to all points in the span of $\vec{\mathbf{v}}_1, \ldots, \vec{\mathbf{v}}_d$, including the edge vectors themselves. Under this tranform, the edge vectors become unit vectors along dimensional axes. These unit vectors are denoted $\mathbf{u}_1, \ldots, \mathbf{u}_d$. Note that unless the simplex is degenerate, the inverse in (3.3) is always well defined. Non-degenerate simplices always have linearly independent edge vectors.

To facilitate interpolation, *value deltas* $\bar{f}_i = f_i - f_0$ are also defined. Interpolation is then done by treating value deltas as extra vector coordinates in the unit space, creating augmented vectors: $\mathbf{a}^+ = (a_1, \ldots, a_d, \bar{f}_{\mathbf{p}})$ and $\mathbf{u}_i^+ = (u_{i_1}, \ldots, u_{i_d}, \bar{f}_i)$.

The transform vector $\mathbf{a}^+$ is then constrained to satisfy the equation of a hyperplane spanned by the augmented unit basis vectors, yielding the unknown final coordinate $\bar{f}_{\mathbf{p}}$.

The equation of a hyperplane is $\mathbf{N} \cdot \vec{b} = 0$ where $\mathbf{N}$ is the hyperplane normal (which is fixed) and $\vec{b}$ is any vector tangent to the plane. In simplex interpolation, we have d basis vectors $\mathbf{u}_1^+, \ldots, \mathbf{u}_d^+$ that allow us to find $\mathbf{N}$, and a vector $\mathbf{a}^+$ that is tangent to the plane. The equation of the hyperplane thus becomes $\mathbf{N} \cdot \mathbf{a}^+ = 0$, where the only unknown value is the final coordinate of $\mathbf{a}^+$: $\bar{f}_{\mathbf{p}}$.

Interpolation is easier in the transformed space because all vectors are unit vectors, which is the motivation for the transform. It is always possible to transform a point into the coordinate space of the triangle, do the interpolation in the normalized coordinate system, and then transform it back.

The hyperplane normal is most conveniently expressed as a $d+1$-dimensional cross product of the basis vectors (including their delta values as element $d+1$), which is often computed with a form of the determinant where the top row is a set of unit vectors and the rest of the elements are scalars:

$$\mathbf{N} = \mathbf{u}_1^+ \times \mathbf{u}_2^+ \times \cdots \times \mathbf{u}_d^+ = \begin{vmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_d & \mathbf{u}_{d+1} \\ u_{1_1} & u_{1_2} & \cdots & u_{1_d} & \bar{f}_1 \\ u_{2_1} & u_{2_2} & \cdots & u_{2_d} & \bar{f}_2 \\ \vdots & \vdots & & \vdots & \vdots \\ u_{d_1} & u_{d_2} & \cdots & u_{d_d} & \bar{f}_d \end{vmatrix} .$$

Since the basis vectors are all unit vectors along an axis in this transformed space, this reduces to the following:

$$\mathbf{N} = \mathbf{u}_1^+ \times \mathbf{u}_2^+ \times \cdots \times \mathbf{u}_d^+ = \begin{vmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_d & \mathbf{u}_{d+1} \\ 1 & 0 & \cdots & 0 & \bar{f}_1 \\ 0 & 1 & \cdots & 0 & \bar{f}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \bar{f}_d \end{vmatrix}.$$

Breaking the determinant into minors yields:

$$\begin{aligned} \mathbf{N} &= (-1)^{1+1}\mathbf{u}_1 \begin{vmatrix} 0 & 0 & \cdots & 0 & \bar{f}_1 \\ 1 & 0 & \cdots & 0 & \bar{f}_2 \\ 0 & 1 & \cdots & 0 & \bar{f}_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \bar{f}_d \end{vmatrix} \\ &+ (-1)^{1+2}\mathbf{u}_2 \begin{vmatrix} 1 & 0 & \cdots & 0 & \bar{f}_1 \\ 0 & 0 & \cdots & 0 & \bar{f}_2 \\ 0 & 1 & \cdots & 0 & \bar{f}_3 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \bar{f}_d \end{vmatrix} \\ &\vdots \\ &+ (-1)^{1+d}\mathbf{u}_d \begin{vmatrix} 1 & 0 & \cdots & 0 & \bar{f}_1 \\ 0 & 1 & \cdots & 0 & \bar{f}_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & \bar{f}_3 \\ 0 & 0 & \cdots & 0 & \bar{f}_d \end{vmatrix} \\ &+ (-1)^{1+(d+1)}\mathbf{u}_{d+1} \left|\mathbf{I}\right|. \end{aligned}$$

All but the final determinant may be expanded down the right column. Each of the minors thus generated will have a determinant of 0 except for the $\bar{f}_j$ minor, matching the $\mathbf{u}_j$ vector that is multiplied by it. In this case, the value of the determinant will be $(-1)^{d+j}\bar{f}_j$. This yields the following simpler formulation of the cross product of these vectors:

$$\begin{aligned}
\mathbf{N} &= (-1)^{d+2}\mathbf{u}_{d+1} \\
&+ (-1)^{1+1}\mathbf{u}_1(-1)^{d+1}\bar{f}_1 \\
&+ (-1)^{1+2}\mathbf{u}_2(-1)^{d+2}\bar{f}_2 \\
&\vdots \\
&+ (-1)^{1+d}\mathbf{u}_d(-1)^{d+d}\bar{f}_d.
\end{aligned}$$

This may be further simplified by combining the powers of -1:

$$\begin{aligned}
\mathbf{N} &= (-1)^{d}(-1)^{2}\mathbf{u}_{d+1} \\
&+ (-1)^{d+1}(-1)^{2}\mathbf{u}_1\bar{f}_1 \\
&+ (-1)^{d+1}(-1)^{4}\mathbf{u}_2\bar{f}_2 \\
&\vdots \\
&+ (-1)^{d+1}(-1)^{+2d}\mathbf{u}_d\bar{f}_d.
\end{aligned}$$

The second power of -1 is always even and can thus be discarded. A $(-1)^d$ may also be factored out of the entire expression, yielding:

$$\mathbf{N} = (-1)^d\left[\mathbf{u}_{d+1} - \bar{f}_1\mathbf{u}_1 - \bar{f}_2\mathbf{u}_2 - \cdots - \bar{f}_d\mathbf{u}_d\right].$$

In other words,

$$\mathbf{N} = (-1)^d(-\bar{f}_1, -\bar{f}_2, \ldots, -\bar{f}_d, 1).$$

Because $\bar{f}_p$ is the last coordinate of $\mathbf{a}^+$, the equation of the hyperplane may be used to solve for this unknown:

$$\mathbf{N} \cdot \mathbf{a}^+ = (-1)^d \left(-a_1\bar{f}_1 - a_2\bar{f}_2 - \cdots - a_d\bar{f}_d + \bar{f}_p\right) = 0.$$

The $(-1)^d$ does not affect the outcome of the equation, yielding

$$\bar{f}_p = \bar{f}_a = a_1\bar{f}_1 + a_2\bar{f}_2 + \cdots + a_d\bar{f}_d.$$

The interpolated delta value at point $\mathbf{p}$ is thus the weighted sum of the values at the vertices. The actual value is found by adding back the value of the origin:

$$f_p = f_0 + \sum_{i=1}^{d} a_i\bar{f}_i.$$

Though this formulation was developed by myself from basic principles, the result is not new. That it fits intuition as well as the well known definition of barycentric interpolation serves to indicate that the derivation is correct.

Armed with the definition of linear interpolation in a simplex, it is possible to prove that maxima must occur at the boundary. In fact, the derivation given here is in a convenient form for the development such a proof, which follows. The formal treatment of the above development of interpolation is also given to facilitate later proofs.

3.1.2 Boundary Maxima Proofs

Definition 3.1.1. An arbitrary *simplex* $\mathcal{S}_d$ in d dimensions is a tuple $(\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_d)$, where each $\mathbf{v}_i$ is a point in Euclidean space. The *vector simplex* $\vec{\mathcal{S}}_d$ corresponding to simplex $\mathcal{S}_d$ is a tuple $(\mathbf{v}_0, \vec{\mathbf{v}}_1, \vec{\mathbf{v}}_2, \ldots, \vec{\mathbf{v}}_d)$ where $\vec{\mathbf{v}}_i = \mathbf{v}_i - \mathbf{v}_0$.

Definition 3.1.2. The *unit vector simplex* $\mathbf{\Delta}_d$ is a vector simplex such that

$$\mathbf{v}_0 = [0, \ldots, 0]$$

and

$$\vec{\mathbf{v}}_{i_j} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases} .$$

Definition 3.1.3. The *transformed normal vector* $\mathbf{a} = (a_1, \ldots, a_d)$ of a point $\mathbf{p} = (p_1, \ldots, p_d)$ into the space spanned by $\mathbf{\Delta}_d$ is given by

$$\mathbf{a} = (\mathbf{p} - \mathbf{v}_0) \begin{bmatrix} \vec{\mathbf{v}}_1 \\ \vec{\mathbf{v}}_2 \\ \vdots \\ \vec{\mathbf{v}}_d \end{bmatrix}^{-1}$$

where each $\vec{\mathbf{v}}_i$ is a row vector of the matrix.

Definition 3.1.4. The *value* of a vertex $\mathbf{v}_i$ is f_i. The value $f_{\mathbf{p}}$ of a point $\mathbf{p}$ that lies on the interior of a vector simplex $\vec{\mathcal{S}_d}$ and whose transformation is $\mathbf{a}$ is given by

$$f_{\mathbf{p}} = f_0 + \sum_{i=1}^{d} \bar{f}_i a_i$$

where $\bar{f}_i = f_i - f_0$.

Definition 3.1.5. A point $\mathbf{p}$ is *contained within* the vector simplex $\vec{\mathcal{S}_d}$ if and only if its transformed point $\mathbf{a}$ obeys the following:

$$[\forall i \in \{1, \ldots, d\} : a_i \geq 0] \wedge \left[\sum_{i=1}^{d} a_i \leq 1 \right] .$$

Definition 3.1.6. A point $\mathbf{p}$ is *on the boundary* of the vector simplex $\vec{\mathcal{S}_d}$ if and only if its tranformed point $\mathbf{a}$ obeys the following:

$$[\exists i \in \{1, \ldots, d\} : a_i = 0] \vee \left(\sum_{i=1}^{d} a_i = 1 \right) .$$

Definition 3.1.7. A point $\mathbf{p}$ is *strictly inside* of the vector simplex $\vec{\mathcal{S}}_d$ if and only if it is not on the boundary of the simplex. Formally:

$$[\neg\exists i \in \{1, \ldots, d\} : a_i = 0] \wedge \left(\exists \delta \in (0, 1] : \sum_{i=1}^{d} a_i = 1 - \delta \right).$$

The theorem that follows states that a maximum cannot occur uniquely in the interior of a simplex. This is useful because it allows the search for a maximally valued action to be limited to a finite number of points, specifically the points where the state hyperplane intersects the *boundaries* of simplices in the joint space.

Theorem 3.1.8. *Given a point* $\mathbf{p}$ *that is strictly inside of a simplex* $\mathcal{S}_d$*, there exists a point* $\mathbf{p}'$ *that is on the boundary of* $\mathcal{S}_d$ *such that* $f_{\mathbf{p}'} \geq f_{\mathbf{p}}$.

Proof. Let $\mathbf{a}$ be the transformed normal vector of $\mathbf{p}$ and $\mathbf{a}'$ be the transformed normal vector of $\mathbf{p}'$. From Definition 3.1.4, the value of a point $\mathbf{p}$ is given by

$$f_{\mathbf{p}} = f_0 + \sum_{i=1}^{d} \bar{f}_i a_i.$$

Also, because $\mathbf{p}$ is strictly inside of the simplex, Definition 3.1.7 dictates that

$$\forall i [a_i > 0] \text{ and } \sum_{i=1}^{d} a_i < 1.$$

In other words, there exists a $\delta \in (0, 1)$ such that

$$\forall i [a_i > 0] \text{ and } \sum_{i=1}^{d} a_i = 1 - \delta.$$

The rest of the proof requires the treatment of two cases.

Case 1: $\exists i, j; i \neq j : \bar{f}_i < \bar{f}_j$

Let $a'_i = 0$, $a'_j = a_i + a_j$, and $a'_k = a_k$ for all $k \neq i, k \neq j$. Since $\bar{f}_i < \bar{f}_j$:

$$\bar{f}_i a_i + \bar{f}_j a_j \quad < \quad \bar{f}_i 0 + \bar{f}_j (a_i + a_j) \quad = \quad \bar{f}_i a'_i + \bar{f}_j a'_j.$$

It follows that

$$\begin{aligned}
\bar{f}_i a_i + \bar{f}_j a_j &< \bar{f}_i a'_i + \bar{f}_j a'_j \\
f_0 + \sum_{i=1}^{d} \bar{f}_i a_i &< f_0 + \sum_{i=1}^{d} \bar{f}_i a'_i \\
f_{\mathbf{p}} &< f_{\mathbf{p}'}
\end{aligned}$$

where $\mathbf{p}'$ is on the boundary, since $a_i = 0$. This shows that it is always possible to find a larger value on a boundary satisfying $\exists i : a_i = 0$.

By further constraining $\bar{f}_j > 0$ and letting $a'_j = a_j + \delta$ and $a'_k = a_k, k \neq j$ it can be shown that it is also possible to find a maximum on a boundary satisfying $\sum_1^d a_i = 1$. Given these additional constraints, we know that

$$\bar{f}_i a_i + \bar{f}_j a_j \quad < \quad \bar{f}_i a_i + \bar{f}_j (a_j + \delta) \quad = \quad \bar{f}_i a'_i + \bar{f}_j a'_j.$$

It follows that

$$\begin{aligned}
\bar{f}_i a_i + \bar{f}_j a_j &< \bar{f}_i a'_i + \bar{f}_j a'_j \\
f_0 + \sum_{i=1}^{d} \bar{f}_i a_i &< f_0 + \sum_{i=1}^{d} \bar{f}_i a'_i \\
f_{\mathbf{p}} &< f_{\mathbf{p}'}
\end{aligned}$$

where $\mathbf{p}'$ is on the boundary because

$$\begin{aligned}
\sum_{i=1}^{d} a'_i &= \delta + \sum_{i=1}^{d} a_i \\
&= \delta + 1 - \delta \\
&= 1.
\end{aligned}$$

Case 2: $\forall i, j : \bar{f}_i = \bar{f}_j$

Since the point $\mathbf{p}$ is strictly inside of the simplex and $\bar{f}_i = \bar{f}_j$:

$$\bar{f}_i a_i + \bar{f}_j a_j \quad = \quad \bar{f}_i 0 + \bar{f}_j (a_i + a_j).$$

Letting $a'_i = 0$, $a'_j = a_i + a_j$, and $a'_k = a_k$ for all $k \neq i, k \neq j$, we have

$$\begin{aligned}
\bar{f}_i a_i + \bar{f}_j a_j &= \bar{f}_i a'_i + \bar{f}_j a'_j \\
f_0 + \sum_{i=1}^{d} \bar{f}_i a_i &= f_0 + \sum_{i=1}^{d} \bar{f}_i a'_i \\
f_{\mathbf{p}} &= f_{\mathbf{p}'}
\end{aligned}$$

where $\mathbf{p}'$ is on the boundary, since $a_i = 0$.

Similar to Case 1, further constraints may be imposed to force the maximum onto a different boundary. Specifically if $\bar{f}_j = \bar{f}_j > 0$ and we let $a'_j = a_j + \delta$ and $a'_k = a_k, k \neq j$ then

$$\begin{aligned}
\bar{f}_i a_i + \bar{f}_j a_j &< \bar{f}_i a_i + \bar{f}_j (a_j + \delta) \\
f_0 + \sum_{i=1}^{d} \bar{f}_i a_i &< f_0 + \sum_{i=1}^{d} \bar{f}_i a'_i \\
f_{\mathbf{p}} &< f_{\mathbf{p}'}.
\end{aligned}$$

where $\mathbf{p}'$ is on the boundary because

$$\begin{aligned}
\sum_{i=1}^{d} a'_i &= \delta + \sum_{i=1}^{d} a_i \\
&= \delta + 1 - \delta \\
&= 1.
\end{aligned}$$

□

Corollary 3.1.1. *Given a point* $\mathbf{p}$ *that is contained within the simplex* $\mathcal{S}$*, there exists a vertex* $\mathbf{v}$ *such that* $f_{\mathbf{v}} \geq f_{\mathbf{p}}$.

Proof. The boundary of a simplex is itself a simplex. Since the maximum value must not occur uniquely in the interior of *any* simplex, it follows that the maximum must occur at the boundary of a boundary simplex. The lowest-dimensional simplex that can exist is 0-dimensional, or a point. Thus, the maximum must occur at a vertex. □

3.1.3 Execution and Practical Considerations

The knowledge that maximum values always occur at simplex boundaries is very useful. Given this result, we can change a continuous piecewise linear approximation of a problem into a discrete problem, confident that maxima in the continuous action space will not be overlooked when we restrict our search to a finite set of well-chosen actions.

The space is discretized using hypercubes and Kuhn Triangulation. Value iteration is done exclusively in the discretized space, updating the values at each corner of each hypercube until convergence is reached.

Each corner $\mathbf{v}$ is decomposed into its state $\mathbf{s}$ and action $\mathbf{a}$. The action $\mathbf{a}$ is applied once, yielding a new state $\mathbf{s}_1$. This state represents a line (or hyperplane) within the joint space, and will intersect with a finite number of simplex boundaries. The simplices are searched for intersections. If $\mathbf{s}_1$ intersects with any hypercube that contains the corner $\mathbf{v}$ as a vertex, $\mathbf{a}$ is applied again, yielding $\mathbf{s}_2$. This continues until the final state $\mathbf{s}_N$ is reached, where one of the following must be true:

- $\mathbf{s}_N$ is an absorbing state
- $\mathbf{s}_N$ does not intersect a hypercube containing $\mathbf{v}$ as a vertex.

Once a suitable final state $\mathbf{s}_N$ has been reached, the value at $\mathbf{v}$ may be updated using the following discrete version of the (3.1):

$$V(\mathbf{s},\mathbf{a}) = \sum_{0=1}^{N} \gamma^{i\Delta t} R(\mathbf{s}_i,\mathbf{a})\Delta t + \gamma^{N\Delta t} \max_{\mathbf{a}' \in \mathcal{A}(\mathbf{s}_N)} V(\mathbf{s}_N,\mathbf{a}'). \tag{3.4}$$

The set $\mathcal{A}(\mathbf{s})$ is found by finding all simplex boundary intersections with the $\mathbf{s}$ hyperplane. This set is finite in a bounded continuous space. Equation (3.4) shows an implementation of Euler integration, but other techniques, like Runge-Kutta, could be used as well. Δt is generally set to a value appropriate for the dynamics of the environment. If fine-grained control is needed, a small Δt may be used, and vice-versa.

It is assumed that reward information is available upon request for any state/action pair. This is easily achieved in offline learning environments or in simulation, but is not so easily achieved in the real world. For this reason, like MM, JoSTLe is currently a strictly offline technique.

A final note is in order regarding the choice of a suitable N. The criteria above do not tell the entire story. It is possible for an agent to get endlessly "stuck" doing the same action over and over again. An example of this is a "do nothing" action. Since new actions are not chosen over a trajectory, it is possible that the agent will never find a hypercube that does not contain $\mathbf{v}$. This can also happen during the beginning stages of the algorithm when the space is not yet discretized.

To alleviate this problem, an upper bound can be set on the value of N that is reasonably high. In practice, this value can be chosen rather arbitrarily without causing the algorithm to degrade noticeably. That it be chosen at all is crucial, however, since otherwise the algorithm may get stuck. In all of the experiments in this paper, the maximum value of N was set to 100 and 1000, without any noticeable difference between the performance of the two.

3.2 Joint Space Splitting

Ideally, splits would be performed based on JoSTLe's ability to approximate the *true* value function. Since the true value function is not known, and thus cannot be used to determine approximation *error*, the model is refined based on a revised criterion: *maximum interpolation disappointment*. Let $V(\mathbf{p})$ be the value at a point $\mathbf{p}$ obtained by discounting and $\widehat{V}(\mathbf{p})$ be the value obtained by interpolation over the simplex $\mathcal{S}_d$

containing $\mathbf{p}$. The point disappointment $q_{\mathbf{p}}$ is given by

$$q_{\mathbf{p}} = \big|V(\mathbf{p}) - \widehat{V}(\mathbf{p})\big|. \tag{3.5}$$

The value $V(\mathbf{p})$ is simply $V(\mathbf{s}, \mathbf{a})$ from (3.4), where $\mathbf{s}$ and $\mathbf{a}$ come from the decomposition of $\mathbf{p}$. The only difference is that in this case $\mathbf{p}$ is in the interior of some simplex, rather than at a vertex.

Since the interior of each rectangle is itself an infinite continuous space, the maximum interior disappointment must be approximated. This is done by generating a random set $\mathcal{M}_R$ of Monte-Carlo points within the rectangle R and calculating the disappointment for each of the points. If the highest of the point disappointments is above a minimum threshold ϵ, the rectangle is marked for splitting. The algorithm for calculating disappointment is summarized here.

Calculating Disappointment

The calculation of the disappointment involves running through (3.4) for each Monte Carlo point $\mathbf{p} \in \mathcal{M}_R$. As is the case with value iteration, this is done until the point's containing hypercube is no longer intersected by the state (for some N such that the state at time $N\Delta t$ satisfies the above). This provides $V(\mathbf{p})$. Then the *interpolated* value $\widehat{V}(\mathbf{p})$ is found at each point (using the Kuhn Triangulation, as explained in Appendix A and [MM02]) to get the point disappointment $q_{\mathbf{p}}$. The set of these disappointments is then analyzed to determine whether a split must occur and in which dimension.

The best dimension in which to split is chosen by hypothetical splitting in each dimension. Each potential split will yield two new rectangles, each with its own disappointment properties. The dimension in which a split yields a rectangle with the smallest maximum disappointment (minimax) is chosen as the best candidate since it will improve the model the most. More formally, the split that creates a rectangle satisfying

$$\arg\min_{R\in\mathcal{R}} \max_{\mathbf{p}\in\mathcal{M}_R} q_{\mathbf{p}} \tag{3.6}$$

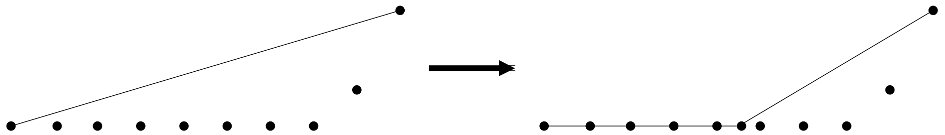

Figure 3.4: A one-dimensional illustration of the splitting process. The line shows the interpolated values, and the points are the values obtained by discounting. A split may only reduce the error of one of its new rectangles and leave the other with a large error.

is chosen, where $\mathcal{R}$ is the set of all rectangles that can be created by splitting the original rectangle in each dimension.

In some cases the chosen split will produce one rectangle with a very small maximum error and another with a larger maximum error. Figure 3.4 shows a line segment before and after splitting. The split is performed even though one of the final segments still has a large amount of disappointment. This is tolerable because the worst of the two will be split in future iterations until its sub-segments also exhibit minimal disappointment.

An important optimization can be applied to the splitting portion of the algorithm. If a rectangle's maximum disappointment is below ϵ, the rectangle can be marked as "untouchable", since splitting it will not be beneficial (it already has a low disappointment). This practice of culling already accurate rectangles from consideration gives the algorithm a real boost over time.

The algorithm stops when it decides that no more rectangles should be split. This occurs when every rectangle either has a low enough disappointment, or a volume that is less than the minimum feature volume. This minimum feature volume is explained in the next section.

3.3 Parameters

The algorithm as given requires some initial parameters to be set for it to function properly. In this section the way these parameters are chosen and the algorithm's sensitivity to them is discussed. The parameters of interest are

- $\mathbf{V}_{\min}$: The volume of the smallest feature of interest
- ϵ: The largest disappointment allowed
- $\mathbf{M}_R = |\mathcal{M}|$: The number of Monte-Carlo points to scatter in a rectangle R

3.3.1 Parameter Values

The volume of the smallest feature of interest in the space is $\mathbf{V}_{\min}$ and is given to the algorithm as an initial parameter. This is a best guess as to how detailed we want the discretization to get. No rectangles will be split that are less than this volume. In many interesting problems, this volume can be computed by assuming that the smallest reward feature is the smallest feature of interest.

The smallest feature volume of interest can actually be found more simply by defining the smallest feature of interest in each dimension l_i. This can be done because all splits are perpendicular to a dimension, and only rectangles are ever created. From this parameter, the volume is easily found from $\mathbf{V}_{\min} = \prod_{i=1}^{d} l_i$. These parameters are often much easier to come up with initially than a volume would be, since a minimum dimensional feature size is fairly simple to visualize.

Take, for example, the problem of balancing an inverted pendulum. The reward is defined as being 1.0 if the pendulum is balanced, and 0.0 if it is not. The reward is a step function with a finite width ("balanced" is usually defined as $-\delta < \theta < \delta$, with some small and previously determined δ). No feature of the value function can be less smooth than the reward itself, so the size of the interesting reward features (in this case 2δ) is a good starting point. In offline techniques like this one, the reward information is almost always known beforehand, so this is not an unreasonable heuristic.

The maximum allowable disappointment can be more difficult to determine depending on the problem at hand. For problems that have a terminal reinforcement only, it is often practical to set it based on the maximum range of rewards $\Delta R = R_{\max} - R_{\min}$, the time step Δt, and the discount factor γ. With terminal reinforcements, an upper bound on the disappointment is given by $\epsilon \leq \Delta R \Delta t$ since an integral of reinforcments will always end once a nonzero reward is received, and the best value acheivable occurs when a reward is one step away.

Problems with non-terminal reinforcements can use an altered upper bound, determined by accounting for an infinite string of discounted rewards (and taking advantage of the fact that when $\gamma \in [0, 1)$, $\sum_{i=0}^{\infty} \gamma^i = \frac{1}{1-\gamma}$)

$$\frac{\Delta R \Delta t}{1 - \gamma}.$$

These upper bounds can be used to define a more intuitive disappointment threshold. The threshold becomes a percentage of the upper bound in these cases, making it easy to come up with reasonable values. For example, 1% of the maximum error is often a reasonable error threshold.

The number of Monte-Carlo points $\mathbf{M}_R$ of a rectangle R whose volume is $\mathbf{V}_R$ can also be easily computed. This value is computed from the minimum feature volume thus:

$$\mathbf{M}_R = \left\lceil \frac{\mathbf{V}_R}{\mathbf{V}_{\min}} \right\rceil .$$

The formulation of $\mathbf{M}_R$ ties it directly to the minimum feature size. This number starts out large for large rectangles and decreases with the size of the rectangle. The smallest allowable feature will be allocated exactly one Monte-Carlo point. This fact makes it easy to determine when to stop splitting a rectangle. If $\mathbf{M}_R$ becomes 1, then the rectangle can no longer be split, since no interior points will ever be tested for the smaller resulting rectangles.

3.3.2 A Note on Parameter Sensitivity

In the problems of Chapter 6 (coming up), the algorithm showed surprisingly little sensitivity to the parameters. Often what seemed reasonable according to the definitions above was well within a window of workability. The disappointment threshold and the minimum feature size could be varied quite widely without adverse effects on the quality of the result.

Often the first thing tried was as good as any hand tuning that was done subsequently. That is very encouraging and shows the potential robustness of the technique. More detailed analysis of parameter sensitivity will be given in Chapter 6.

3.4 Computation of Intersections

The computation of intersections is central to JoSTLe, but has not yet been treated with the care it deserves. This section describes how the intersections are computed during the execution of the algorithm.

Finding the intersections of a 1-dimensional line with the boundaries of a 2-dimensional triangle is relatively easy to visualize and to code. Finding such an intersection when faced with arbitrary dimensionalities is much more difficult. The challenge is to develop an algorithm that extends easily into any number of dimensions.

The intersections are computed for the purpose of finding maximum values. Fortunately, the fact that these maxima occur at the boundaries of simplices allows us

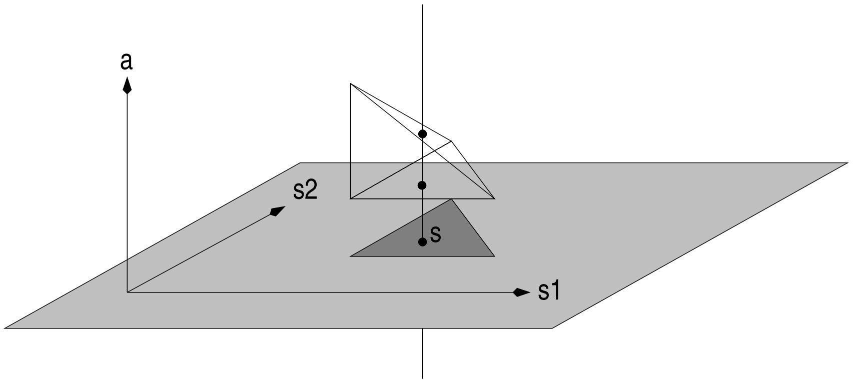

Figure 3.5: The state space projection of a joint space tetrahedron is shown as a shadow on the state plane. The state line becomes a point in the state space, and that point is contained within the projected shadow.

to take a significant shortcut. The intersections do not need to be computed directly. They can be derived from interpolation in the projected space. Figure 3.5 illustrates this concept in three dimensions. The points of intersection are shown in the figure, as is the projection of the tetrahedron's faces onto the state plane. Note how the state, which is a line in the joint space, is a point in the projected space.

Thus, rather than finding the points of intersection of the state line with the joint space tetrahedron, we can project the faces of the tetrahedron onto the state space, forming a shadow of the original. Some of the faces will overlap each other because the projected simplex is degenerate (e.g., the shadow in the figure is actually all four triangles, two of which project to lines, and the other two of which project to the shaded region of the state space).

We now have two triangles that contain the state as a single point. Because it is linear, interpolation over the intersections in the joint space yields (conveniently) the same value as interpolation over the point in the projected space. Since interpolation of a point in a simplex is a known quantity in multiple dimensions, this is a convenient result, indeed.

Thus, the joint space simplices have d_s-dimensional boundaries that can be projected into the state space. In Figure 3.5, for example, the tetrahedron is composed of four 2-dimensional triangle boundaries, corresponding to the 2 dimensions of the state space. We could have done the same thing with a 1-dimensional state space by projecting the edges of the tetrahedron onto a state line. In general, there are always $\binom{d+1}{d_s+1}$ of these projections, where $\binom{n}{m} = \frac{n!}{m!(n-m)!}$. Each projection that contains the state point can be interpolated at that point to find not only the value, but the missing action coordinates that correspond to it. One of those values will be the maximum.

This insight allows us to approach searching the action space in a systematic fashion. The state point in question can be used to find the relevant hypercubes, and their triangles can be projected into the state space. Those which contain this point are used for interpolation, and the best is chosen.

This result is crucial to the ability of JoSTLe to be implemented for problems of arbitrary dimensionality, and is therefore worth taking the time to prove. The proof is given in the following section.

3.4.1 Projected Intersection Proof

Lemma 3.4.1. *Let $\mathbf{\Delta}_d$ be a d-dimensional unit vector simplex and s be an integer such that $0 < s < d$. Let $\mathbf{\Delta}_s$ be an s-dimensional boundary of $\mathbf{\Delta}_d$ which contains the origin as one of its vertices. If $\mathcal{U}_s$ is the set of edge vectors in $\mathbf{\Delta}_s$, then any point $\mathbf{a}$ contained within $\mathbf{\Delta}_s$ will have zeros for all coordinates a_i where $i \in \{x : (1 \leq x \leq d) \wedge (\mathbf{u}_x \notin \mathcal{U}_s)\}$.*

Proof. The proof follows from the definition of a span. A point contained within the space spanned by a basis set $\mathcal{U}_s$ of unit vectors cannot have nonzero coordinates in any dimension not represented by that set of vectors. If it did, then some of its coordinates would not be representable as a linear combination of the basis vectors, and therefore it would not be in that space, but in a space orthogonal to the vectors in $\mathcal{U}_s$. □

The following theorem claims that the transformed point $\mathbf{a}$ of any point $\mathbf{p}$ in an *arbitrary* boundary simplex will only have non-zero coordinates with the same indices as the vectors in that boundary simplex. This establishes a connection between arbitrary boundary simplices in any basis space and boundary simplices in the unit vector space. This connection allows the results that follow to apply to any simplex in any space.

Theorem 3.4.1. *Consider an arbitrary simplex $\mathcal{S}_d$ in d dimensions with a boundary simplex $\mathcal{S}_s$ in s dimensions where $0 < s < d$. Let $\mathcal{U}_s = \left\{\vec{\mathbf{v}} : \vec{\mathbf{v}} \text{ is a vector of } \vec{\mathcal{S}_s}\right\}$. Then for all $i : \vec{\mathbf{v}}_i \notin \mathcal{U}_s$, the transformed coordinate $a_i = 0$.*

Proof. Since from the previous proof we know that for a point in the boundary $\mathbf{\Delta}_s$ of the unit vector simplex $\mathbf{\Delta}_d$ the coordinates corresponding to unit vectors not in the boundary are zero, we need merely show that the vectors in $\vec{\mathcal{S}_s}$ have the same indices as unit vectors in the corresponding unit vector boundary $\mathbf{\Delta}_s$. This is done in two parts, one for the origin vertex, and one for every other vertex.

Origin Vertex

Without loss of generality, let the origin vertex $\mathbf{v}_0$ of $\mathcal{S}_d$ be chosen such that it is also a member of $\mathcal{S}_s$. Let $\mathbf{p} = \mathbf{v}_0$ be the point for which we wish to find the transform:

By Definition 3.1.3 and the properties of linear algebra, the following is true:

$$\begin{aligned} \mathbf{a}\begin{bmatrix} \vec{\mathbf{v}}_1 \\ \vec{\mathbf{v}}_2 \\ \vdots \\ \vec{\mathbf{v}}_d \end{bmatrix} &= (\mathbf{p} - \mathbf{v}_0) \\ &= (\mathbf{v}_0 - \mathbf{v}_0) \\ &= [0, \ldots, 0]\,. \end{aligned}$$

For this equation to be true in general, all coordinates a_i of $\mathbf{a}$ must be zero, which shows that the transform of $\mathbf{v}_0$ is indeed the origin. Next we show that any non-origin vertex in the boundary simplex will have the same index in the transformed space.

Non-origin Vertices

Let $\mathbf{v}_i$ be an arbitrary non-origin vertex in $\mathcal{S}_s$. By Definition 3.1.1, $\vec{\mathbf{v}}_i = \mathbf{v}_i - \mathbf{v}_0$ is a vector in $\vec{\mathcal{S}}_s$. Let $\mathbf{p} = \mathbf{v}_i$ be the point for which we wish to find the transformed point $\mathbf{a}$.

By Definition 3.1.3 and linear algebra, the following is true:

$$\begin{aligned} \mathbf{a}\begin{bmatrix} \vec{\mathbf{v}}_1 \\ \vec{\mathbf{v}}_2 \\ \vdots \\ \vec{\mathbf{v}}_d \end{bmatrix} &= (\mathbf{p} - \mathbf{v}_0) \\ &= (\mathbf{v}_i - \mathbf{v}_0) \\ &= \vec{\mathbf{v}}_i. \end{aligned}$$

For the above to be true in general, the following equation must hold:

$$a_j = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}.$$

This means that $\mathbf{a} = \mathbf{u}_i$ from Definition 3.1.2.

Since $\mathbf{u}_i$ in $\boldsymbol{\Delta}_s$ corresponds to $\mathbf{v}_i$ in $\mathcal{S}_s$, it follows that each vector $\vec{\mathbf{v}}_i$ of $\mathcal{S}_s$ will transform to $\mathbf{u}_i$ in the unit vector space. Since the point $\mathbf{p}$ is in the span of vectors in $\mathcal{S}_s$, its transform $\mathbf{a}$ must be in the span of the vectors of $\boldsymbol{\Delta}_s$, which cannot have nonzero coordinates outside of that span. □

The following definitions and theorems establish that interpolation over a boundary in the joint space yields the same result as interpolation over the projected boundary in a smaller space.

Definition 3.4.2. A *constraint index set* is a set $\mathcal{X}$ such that every element is a dimensional index. A *constraint vector* $\mathbf{x}$ is a vector of values where x_i is a fixed value if $i \in \mathcal{X}$ and a wildcard otherwise.

Definition 3.4.3. A set of constraints defines a *constraint hyperplane*, where all points $\mathbf{p}$ satsifying $i \in \mathcal{X} \implies p_i = x_i$ are in the hyperplane.

Definition 3.4.4. Given a constraint index set $\mathcal{X}$ and a constraint vector $\mathbf{x}$, a *constrained projection* of a point $\mathbf{p}$ is a point $\widehat{\mathbf{p}}$ such that the following holds:

$$\widehat{p_i} = \begin{cases} p_i & \text{if } i \in \mathcal{X} \\ 0 & \text{otherwise} \end{cases} .$$

In other words, the point $\mathbf{p}$ is projected onto the space spanned by the vectors $\vec{\mathbf{v}}_i$: $i \in \mathcal{X}$.

Definition 3.4.5. The *constrained projection of a simplex* is a simplex such that all vertices have gone through a constrained projection as above.

Theorem 3.4.6. *Let $\mathcal{S}_d$ be a simplex in d dimensions. Let $\mathcal{X}$ and $\mathbf{x}$ be a set of constraints, and let $s = |\mathcal{X}|$. Let $\mathbf{p}$ be the point of intersection of the corresponding* *constraint hyperplane with an s-dimensional simplex $\mathcal{S}_s$, which is a boundary of $\mathcal{S}_d$. Assume such an intersection exists. Let $\mathcal{S}_p$ be the constrained projection of $\mathcal{S}_s$, and $\widehat{\mathbf{p}}$ be the constrained projection of $\mathbf{p}$.*

Given the above, the interpolated value at $\mathbf{p}$ using $\mathcal{S}_s$ is the same as the interpolated value at $\widehat{\mathbf{p}}$ using $\mathcal{S}_p$.

Proof. Without loss of generality, let us number the vertices of $\mathcal{S}_d$ such that the boundary simplex $\mathcal{S}_s$ contains the vertices $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_s$. It follows that $\vec{\mathcal{S}_s} = (\mathbf{v}_0, \mathbf{u}_1, \ldots, \mathbf{u}_s)$ and $\vec{\mathcal{S}_p} = (\widehat{\mathbf{v}_0}, \widehat{\mathbf{u}_1}, \ldots, \widehat{\mathbf{u}_s})$. Furthermore, let $\mathcal{X} = \{1, \ldots, s\}$

From the proof of Theorem 3.4.1, the coordinates of the transformed point $\mathbf{a}$ of $\mathbf{p}$ have the property that $a_i = 0$ if $i > s$, since $\mathbf{p}$ is contained within the boundary simplex $\mathcal{S}_s$. Interpolation of the value of a point $\mathbf{p}$ is done using the coordinates of of its transform $\mathbf{a}$ (Definition 3.1.4), so the value at point $\mathbf{p}$ is given by

$$\begin{aligned} f_{\mathbf{p}} &= f_0 + \sum_{i=1}^{d} a_i \bar{f}_i \\ &= f_0 + \sum_{i=1}^{s} a_i \bar{f}_i \end{aligned}$$

since only the first s coordinates of $\mathbf{a}$ are nonzero.

Discovery of the coordinates of $\mathbf{a}$ is accomplished through solving this system of equations, where $\vec{\mathbf{v}}_i$ is a row vector taken from $\mathcal{S}_d$

$$[a_1, \ldots, a_s, 0, \ldots, 0] \begin{bmatrix} \vec{\mathbf{v}}_1 \\ \vec{\mathbf{v}}_2 \\ \vdots \\ \vec{\mathbf{v}}_d \end{bmatrix} = (\mathbf{p} - \mathbf{v}_0)\,.$$

which may also be expressed as

$$\begin{array}{rcl} a_1v_{11} + a_2v_{21} + \cdots + a_sv_{s1} & = & p_1 - v_{01} \\ a_1v_{12} + a_2v_{22} + \cdots + a_sv_{s2} & = & p_2 - v_{02} \\ \vdots & & \vdots \\ a_1v_{1s} + a_2v_{2s} + \cdots + a_sv_{ss} & = & p_s - v_{0s} \\ \vdots & & \vdots \\ a_1v_{1d} + a_2v_{2d} + \cdots + a_sv_{sd} & = & p_d - v_{0d} \end{array} . \tag{3.7}$$

This leaves us with d equations in d unknowns. Of interest for value interpolation is the unknown quantity $\mathbf{a}$, of which only the first s coordinates are unknown. Conveniently, the first s coordinates of $\mathbf{p}$ are known, giving us s relevant equations in s unknowns. These first s equations may be solved for $\mathbf{a}$ to carry out interpolation.

If we do the same interpolation using the projected point and simplex, we get the following:

$$[\widehat{a_1}, \ldots, \widehat{a_s}] \begin{bmatrix} \widehat{\mathbf{u}_1} \\ \vdots \\ \widehat{\mathbf{u}_s} \end{bmatrix} = (\widehat{\mathbf{p}} - \widehat{\mathbf{v}_0})$$

This gives us the following system of equations:

$$\begin{array}{rcl} \widehat{a_1}\widehat{v_{11}} + \widehat{a_2}\widehat{v_{21}} + \cdots + \widehat{a_s}\widehat{v_{s1}} & = & \widehat{p_1} - \widehat{v_{01}} \\ \vdots & & \vdots \\ \widehat{a_1}\widehat{v_{1d}} + \widehat{a_2}\widehat{v_{2d}} + \cdots + \widehat{a_s}\widehat{v_{ss}} & = & \widehat{p_s} - \widehat{v_{0s}} \end{array} . \tag{3.8}$$

But, by Definition 3.4.4, the first s coordinates of each projected point and vector are equivalent to the originals, so we have $\widehat{v_{ij}} = v_{ij}$ for $1 \leq i, j \leq s$, and similarly for $\mathbf{p}$ and $\mathbf{v}_0$.

Therefore $\widehat{\mathbf{a}}$ is a solution to the same set of equations as $\mathbf{a}$, giving $\mathbf{a} = \widehat{\mathbf{a}}$. Thus the interpolated value of $f_{\mathbf{p}}$ is given by the following equation:

$$f_{\mathbf{p}} = f_0 + \sum_{i=1}^{s} a_i \bar{f}_i = f_0 + \sum_{i=1}^{s} \widehat{a}_i \bar{f}_i. \tag{3.9}$$

The result is the same whether using the intersection point or its projection. □

From the above proof, it follows that if we are searching the joint space at a particular fixed state, we know that the intersection of the hyperplane defined by that constraint will intersect with a number of simplices. Furthermore, we know that it will intersect with a set of d_s-dimensional boundaries of those simplices, where d_s is the dimensionality of the state space.

Because of this fact, we can generate all d_s-dimensional boundaries whose projections contain the state point in question and choose the interpolated value among them that is the highest. We are guaranteed that we will not miss any maximums, since we will have searched all of the boundaries and they will always occur there.

Corollary 3.4.1. *Discovery of the missing coordinates in a constrained intersection with the boundary simplex $\mathcal{S}_s$ is the same as treating the missing coordinates as values and doing interpolation on them.*

Proof. The $d - s$ unused equations of system (3.7) have $d - s$ unknowns, namely $p_{s+1}, \ldots, p_d$. Simplifying and making use of Definition 3.1.1 we get $p_i = v_{0i} + \sum_{j=1}^{s} a_j(v_{j_i} - v_{0i})$ for $s + 1 \leq i \leq d$, which is analagous to value interpolation as given in Definition 3.1.4. □

Thus, to find the actual coordinates of the intersection, we can project the simplex into the lower-dimensional space, find the values of $a_1, \ldots, a_s$ using the transform formula, then use those coordinates to not only interpolate the value, but to interpolate the missing (unconstrained) coordinates of the intersection.

This is very useful when obtaining the policy from the joint space. We can use the very same projection insight to find out which action results in the highest value as we do to find the value itself.

3.4.2 Generation of Projected Simplices

Knowing that a simplex projection makes interpolation easier is very helpful. This section details how these projections are generated.

Generation of projected simplices involves three basic steps:

1. Find the hypercubes that intersect with the state constraint.
2. Generate the Kuhn simplices of each hypercube.
3. Generate the boundary simplices of each Kuhn simplex.

Hypercube Generation

The kd-trie structure allows for more than the ability to efficiently find the hypercube containing a given point. It also allows for efficient *range queries*. The result of a range query is the set of hypercubes that intersect a subset of the full space. For our purposes, this is useful for finding those hypercubes that contain the state hyperplane.

A range query is performed by querying the kd-trie with a point where one or more coordinates are wildcards[1]. At each level of the trie, the path downward represents a split in exactly one dimension. If that dimension has a constrained coordinate in the query, it is easy to determine which path to take. If the split occured in a dimension corresponding to an unconstrained coordinate, then both paths are taken. The ambiguity inherent in this kind of a query results in multiple satisfactory rectangles.

[1] Actually, this is an oversimplification. A range query allows for value ranges as well as wildcards. The full power of range queries is not needed for JoSTLe to operate, however, so range values are not discussed here.

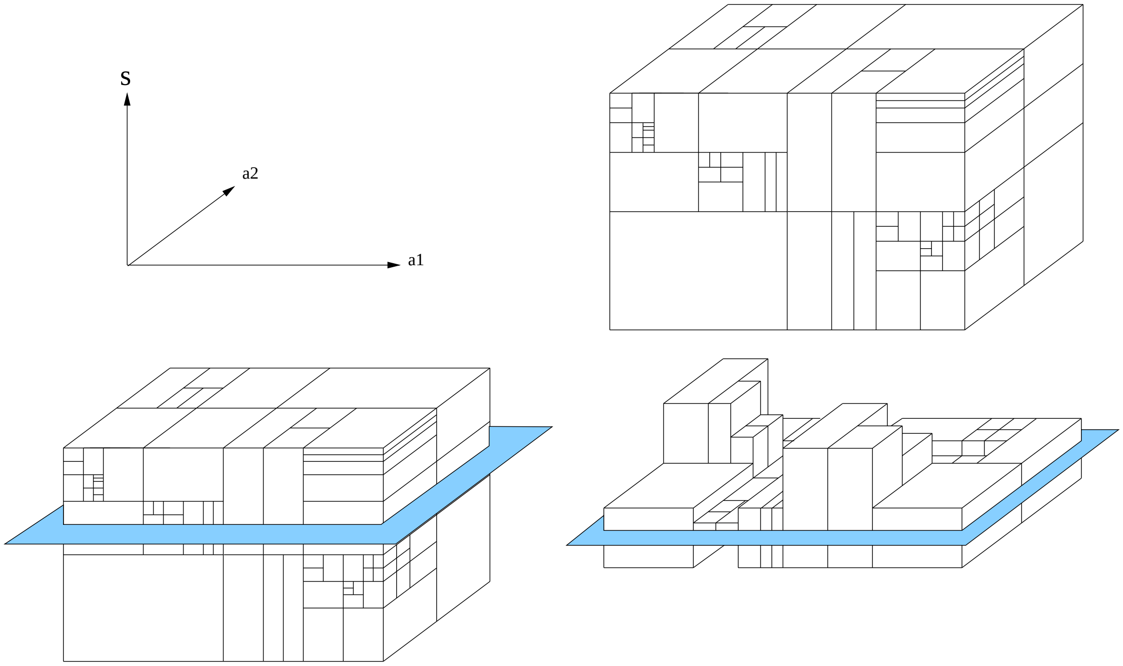

Figure 3.6: The joint space (1 state dimension and 2 action dimensions), tesselated with 3-dimensional rectangles, is shown as a whole, with a state constraint (this time a plane instead of a line), and with the rectangles relevant to that constraint.

Consider Figure 3.6. The joint space is shown as being tessellated with 3-dimensional rectangles. Also shown is the state plane (indicating that in this particular example, the state space is 1-dimensional and the action space is 2-dimensional) and the rectangles that contain the constraint plane. The kd-trie structure allows us to efficiently find only those hypercubes that contain triangles of interest, thus narrowing our search for useful simplices substantially.

Kuhn Simplex Generation

Once the relevant set of hypercubes is found, we can then generate the Kuhn Simplices for each one. Each hypercube has $d!$ simplices, which are easily generated (see Appendix A for details). Many of the simplices can be immediately discarded because they are parallel to the constraint hyperplane (the horizontal edges in Figure 3.6 will never intersect the constraint plane).

The factorial explosion of simplices in d is one of the unfortunate side-effects of the use of triangulation in the algorithm. It is not unique to Kuhn, however. All triangulation schemes known to the author exhibit factorial or exponential behavior in dimension.

Projected Boundary Generation

Once we have the relevant Kuhn simplices, we can project them all onto the state space. Projection of the simplices involves enumerating the d_s-dimensional boundaries of each joint space simplex and setting their action coordinates to zero. Each of these boundaries is itself a simplex (in Figure 3.6 each boundary is a line segment, or a 1-dimensional simplex).

We can further cull the projected simplices by removing those that do not contain the state point in question (since an intersection cannot occur in the space if the shadow does not contain the constraint point) and by removing any duplicates (joint space simplices will share boundaries). We then interpolate over the rest to get the value of each one at the state point.

The boundaries are easy to generate. A simplex $\mathcal{S}_d$ in d dimensions will have $d+1$ vertices $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_d$. Let us assume that we are constraining d_s coordinates. To enumerate all of the d_s-dimensional boundaries of the simplex $\mathcal{S}_d$, we can simply generate all $\binom{d+1}{d_s+1}$ combinations of vertices. Each combination of vertices corresponds to a boundary of the simplex, which is itself a simplex in d_s dimensions.

The reason that we know that the new boundary is itself a simplex comes from graph theory. Every simplex is a complete graph. Any subgraph of a complete graph is also complete. Thus, given any subset of vertices of a simplex, this subset of vertices also defines a simplex.

The combinatorial nature of the projected simplices is another unfortunate side-effect of the approach taken by this algorithm. Given all of the culling that can be done before reaching this point, however, it has not proven to be a crippling problem, especially with the low dimensionality of the problems presented here.

3.5 Convergence

When introducing new function approximation schemes into a RL setting, the question of convergence naturally arises. Gordon addressed the issue of convergence in [Gor95] at length, and showed that *averaging* function approximators will allow the value iteration process to converge. Among these are barycentric interpolators, of which the linear interpolation method described here is one.

Additionally, value iteration is done separately from the refinement process. Each state hyperplane crosses through a finite and static set of boundary facets, yielding a finite and static set of available actions for that state. Thus the problem is reduced to value iteration over a discrete space. That the number of actions at each state is fixed during value iteration and averaging approximation is used solidifies the argument for value convergence.

Chapter 4

Delaunay: The Road Not Taken

Before settling on Kuhn triangulation and the hypercube constraint, an attempt was made at discretizing the joint space in a free form manner. Joint space points were added as they were explored. Interpolation was needed between these points, and this necessitated using a tessellation technique. The Delaunay Triangulation appeared to be the clear winner for its duality with Voronoi Diagrams, its property of angle maximization and its relative stability in the face of vertex insertion and deletion [For92]. Indeed, it has been used before in the state space, though not in a completely free-form manner [Muñ97]. For several reasons, however, it was abandoned in the joint space. This section describes some of the things that were discovered in the process of attempting to use it.

4.1 The Incircle Property

Delaunay triangulation is defined by a deceptively simple criterion: no vertex falls inside of the circumcircle of any triangle to which it is not incident. Figure 4.1 shows the circumcircle of a particular triangle in a Delaunay triangulation. Note that no

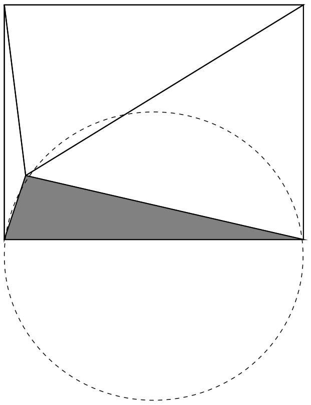

Figure 4.1: A Delaunay Triangulation showing the circumcircle of one of the triangles.

vertices outside of the triangle fall within the circumcircle shown.

This has some interesting side-effects: it tends to maximize the minimum angle in the triangulation (though unfortunately it does not also minimize the maximum angle) and it guarantees that the triangulation is unique as long as points are not perfectly uniformly spaced [For92, Tan93].

The maximization of minimum angles is a nice property because it tends to make the triangles smaller and more uniform in shape. This means that interpolation done over the triangles will be influenced only by local points. This is a desirable property.

The uniqueness property is equally appealing. Given a set of points, the triangulation is fixed and unambiguous. This is nice because it makes the triangulation invariant under insertion and deletion order.

While at first glance these appear to be very desirable properties for spatial tessellation, they can cause problems.

4.2 Boundary Triangles

Triangles in the interior of a tessellation tend to be small and fairly uniform in shape. This is one reason that Delaunay appears to be so useful. The problem is that this

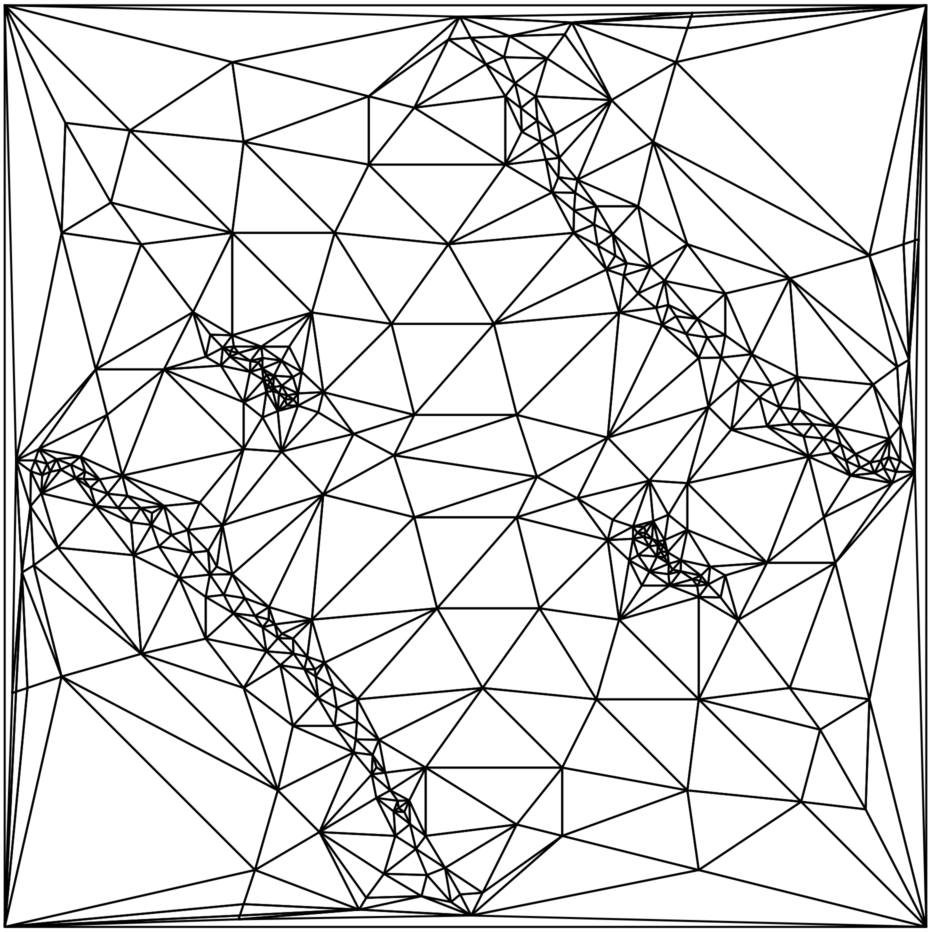

Figure 4.2: A DeLaunay Triangulation of the golf problem of Chapter 5. Note the very long and skinny boundary triangles.

property does not extend to the *boundaries* of a tessellated space. At the boundaries things are more constrained and the triangles are almost always long and skinny (this is allowed because Delaunay does not simultaneously minimize the maximum angle [Tan93]). The incircle property enforces this because the boundary points must be part of some triangle, and the only way to keep the circumcircle of that triangle from containing any other vertices is to have it extend outside of the space. As more points are added in the interior of the space, the boundary triangles become more and more skinny.

Serious problems occur when attempting to do interpolation on these triangles.

They tend to contain points on opposite ends of the space. This means that values interpolated across these triangles are influenced unduly by nonlocal features. This problem is compounded when learning in the joint space, since some of these boundary triangles span huge portions of the state space, and are thus present in the calculations for *every single state.*

Figure 4.2 shows a Delaunay triangulation of the golf problem described in Chaper 5. The boundary triangles are very long and skinny, and those on the top and bottom pose particularly difficult problems, since they span the entire state space and can throw off the value iteration process.

There are a number of things that can be done to correct for this problem, and it is a well-known and oft-discussed issue in literature. The triangles can be made smaller by adding new points at the boundaries. Deciding which points to add is a somewhat thorny issue, and in order to correct for these boundary problems an exponential explosion of points may occur.

4.3 Existence

Delaunay is guaranteed to exist in 2 dimensions, provided that the points are in *general form* [Müc95]. The general form constraint is fairly rigid. Certain configurations of points can cause some of the triangles to be degenerate (three vertices lying on a straight line, for example) and these are not allowed in general form. In 2 dimensions, there are only 2 kinds of degeneracy: 3 points of a triangle lying in a straight line, and 3 points that are the same.

In 3 and higher dimensions, however, the number of ways that degeneracy can occur increases linearly with dimension, causing an exponential increase in the likelihood of hitting one of them. The triangles at the boundaries of the space tend to be degenerate with alarming frequency, and a small number of evenly spaced points will also cause degeneracy to occur.

Degenerate triangles take up space without serving any useful purpose for interpolation, but they are necessary to make the triangulation exist. Several methods exist for overcoming this difficulty. Software that performs Delaunay triangulations often perturbs the offending vertices, making a very thin but nondegenerate triangle in place of a degenerate one. Even so, in 3 and higher dimensions this does not guarantee the removal of all degeneracies [Müc95].

In addition to all of this, deletion of points from unconstrained triangulations in higher dimensions can generate polytopes that are impossible to triangulate (the Schönhardt polytope is one of these [BLRG00]) with *any* triangulation.

4.4 Uniqueness

The uniqueness property is nice because it allows us to forget about point insertion and deletion order. No matter when a point is added to or removed from the tesselation, the triangulation will be the same for the same set of points.

This property has its costs. Deletion and insertion do not always cause purely local changes. Any triangle whose circumcircle contains a newly inserted point must be removed and new triangles formed [For92, Dev99]. This is not usually a problem in 2 dimensions, but in 3 and higher dimensions the number of triangles affected by a new point or a removed point increases exponentially due to the increased degrees of freedom. Thus, the effect of a new or removed point is often neither predictable nor local.

When doing value iteration over a tessellated space, stability is crucial. If the addition of a single point can wipe out tens or hundreds of nearby triangles, the interpolation information that these triangles contained is lost and the new interior values may not resemble the old ones at all. This causes large portions of the model to take on incorrect values whenever refinement is done and introduces an instability into the value iteration process.

This effect was most severe near the boundaries of the space, which turned out to be extremely important because of the fact that boundary triangles span such a large portion of the space.

4.5 Projection and Value Iteration

Fairly fast search algorithms exist that allow the triangle containing a particular point in space to be found [BT93, Dev98]. These algorithms are not clearly extensible to finding the set of triangles intersected by a hyperplane (range queries), however, which is crucial to value iteration in the joint space. The obvious implementation involves projecting *all* triangles in the joint space onto the state space and then performing a search to see which subset of the projected facets contains the state point.

At first glance, the fact that a point can be located in a Delaunay triangulation appears to solve the problem. The location algorithm, however, only applies to Delaunay triangulations, and the projection of a Delaunay triangulation onto a subspace is not only not Delaunay, it is not even a tessellation, since it is comprised of overlapping facets. No efficient algorithm is known at this time that compares with the range queries of a hypercube tessellation.

4.6 Implementation

Implementing Delaunay in an efficient manner is anything but simple [BT93,DGH01, Dev98]. It can be derived from the convex hull of its projection onto a hyperparabloid, and many algorithms exist for finding convex hulls. The difficulty is that all of the algorithms in existence work with an entire set of points and do not allow for deletions and insertions. The ability to insert vertices into an existing triangulation is crucial to dynamic refinement (otherwise every time a point is added or deleted everything has to be triangulated all over again).

The algorithms for insertion and deletion in Delaunay are well-known and widely understood in 2 dimensions. In higher dimensions, things fall apart again. Searching for triangles whose circumcircles contain the new point is inefficient and difficult to implement, and retriangulation in multiple dimensions is a non-trivial procedure.

Far worse is the case of point deletion. This is done when looking ahead to see if a particular change is going to benefit the model. A point must be added and then deleted if it is not helpful. The deletion literature treats 2-dimensional triangulations in detail, but leaves higher dimensionalities to the observant and highly motivated reader, saying that extension into these dimensions is “easy” [Dev99].

The incremental deletion algorithm is not easy to extend into dimensions higher than 2. In fact, the algorithm given in the literature does not extend above 2 dimensions without some subtle but important alterations to the methodology, though the motivating principles still apply. Fortunately, a correct algorithm exists (see Appendix B for details) and was discovered and implemented by the author of this work. The deletion algorithm developed appears to be the first successful adaptation of [Dev99] to any dimension higher than 2.

The correct algorithm is not efficient, and it is not yet clear how to improve it. It seems that the complexity can be no better than factorial in dimension.

In addition to these issues, the entire triangulation *must be stored.* It is not sufficient to simply store the vertices and dynamically determine the triangles pertaining to a particular vertex, or the triangle that contains a particular point. The triangulation must be computed and remembered. This imposes a tremendous space constraint in comparison to hypercubic methods, which may simply store points in some searchable structure.

4.7 Other Triangulations

The obvious question is "If not Delaunay, which triangulation should be used?" The answer is not obvious. Many triangulations exist, and they have different features and implementation details [AX00]. None of them are simple, and many of them share difficulties with Delaunay (the "Greedy" triangulation, for example, can be derived from a Delaunay triangulation).

The hypercubic tessellation of space, on the other hand, is simple, easy to store efficiently, easy to make range queries on (and thus find hyperplane intersections), simple to compute, well understood, stable, unique, existent, and it works well at the boundaries. The only thing missing is the ability to tessellate space using arbitrary points. Because of these properties, Delaunay was abandoned in favor of hypercubic tessellation and Kuhn triangle interpolation. It remains to be seen whether tessellation with arbitrary points and nice implementation properties is an attempt to simultaneously possess and consume a cake. It appears to be likely.

Chapter 5

Illustrative Problems

The smallest dimensionality of any useful joint space is 2, since at least one state and one action variable must be present for reinforcement learning to occur[1]. The dimensionality of RL environments is traditionally determined by the number of states, so the minimalist problem presented here is called 1-Dimensional Golf. In addition, the oft-discussed Mountain Car problem will be explained.

5.1 1-Dimensional Golf

The agent is in a 1-dimensional room with a golf iron, a golf ball, and a hole in the middle of the room. It attempts to get the ball into the hole in one shot. If it hits it too hard, it will go over the hole. If it hits it in the wrong direction, it may hit a wall and get a negative reinforcement. If it goes into the hole, it gets a positive reinforcement.

In this problem, the state space is continuous: $\mathbf{s} \in [-10, 10]$. The action space is

[1] This is beside the fact that single-state no-action problems are not very interesting, since the agent cannot act on the environment and therefore has nothing to decide. Single action no-state problems are equally uninteresting, though more humorous to consider.

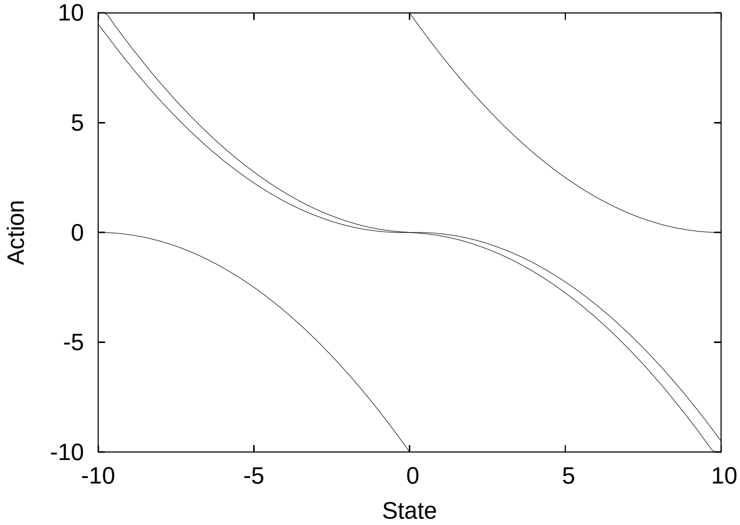

Figure 5.1: The 1-Dimensional Golf problem. The boundaries snaking through the middle of the joint space represent positive reward boundaries (the reward is between them), and the other lines represent negative reward boundaries (in the corners).

also continuous: $\mathbf{a} \in [-10, 10]$. The hole is centered at $(0, 0)$ and is 0.5 units wide (the positive reward is achieved when $\mathbf{s}' \in [-0.25, 0.25]$). The environment is deterministic and accessible, and the resultant state ($\mathbf{s}'$) from taking an action is determined by the following formula:

$$\mathbf{s}' = \mathbf{s} + \frac{\mathbf{a}}{|\mathbf{a}|}\sqrt{10|\mathbf{a}|}$$

If the new state is outside of the range $[-10, 10]$, it is clipped to fit inside of it. In other words, if the ball hits a wall, it does not bounce back. It stays up against the wall. If the agent hits a ball such that clipping must occur, it receives a reinforcement of -1. If $s' \in [-0.25, 0.25]$, a reward of 1 is received. In all other cases, a reward of 0 is received. Figure 5.1 is a graphical representation of the joint space with the positive and negative reward boundaries shown.

This problem has some interesting characteristics. The area of positive reward is very small compared with the rest of the space, and it is not strictly a boundary reward, making it somewhat difficult to find (especially near the center of the space, where it is extremely thin). Additionally, the optimal solution to this problem is always a one-step action, and the positive reinforcement is either on or off. Either the ball fell in the hole or it did not.

All of the above characteristics make this problem difficult to solve, especially for techniques that naïvely discretize or require *a priori* discretization of the action space. It is also minimalist, making it a very good comparison point for current variable resolution schemes and JoSTLe.

5.2 Mountain Car

A problem that is often treated in the variable resolution discretization literature is the Mountain Car problem. In this problem, the goal is to get a car from the bottom of a hill to the top. Figure 5.2 illustrates the concept pictorially. A reward is given if the car stops at the top of the hill on the right, and a penalty is given for driving off the left side, or for going too fast off the right side.

The car in this problem does not have enough thrust to make it from the bottom to the top without first backing up the hill on the left. This problem is minimum time, and as such can be solved with bang bang control (only “full forward” and “full reverse”) [BH69]. The reader is referred to [MM02] for a more detailed treatment of the problem.

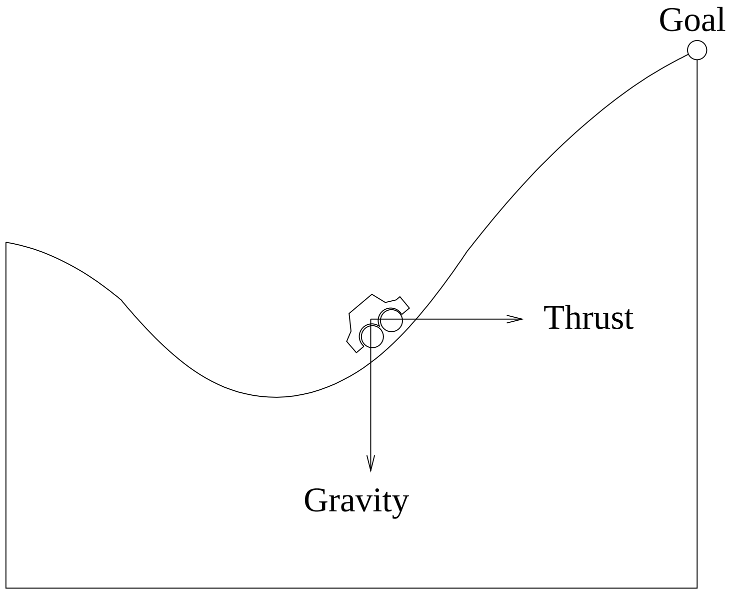

Figure 5.2: The Mountain Car problem. The car must reach the top of the hill on the right, but does not have enough power to do so without first backing up the hill on the left. A reward is given for stopping at the top, and a penalty is given for driving off either end.

Chapter 6

Results

Both JoSTLe and MM were applied to the golf problem. Though MM has many possible splitting criteria, in a 1-dimensional problem *average corner value difference* works as well as any of them (more complex criteria are only helpful in higher dimensions [MM02]). In MM splitting occurred if the value difference was above 0.001, and in the case of JoSTLe the regions were split when the disappointment was worse than 5% of the upper bound on the error (Section 3.3) with Δt assumed to be 1. Both used a γ of 0 for this problem as a way of optimizing the experiments' running time, since it was known beforehand that the positive reward was available everywhere in the state space and that discounting would not be necessary for the extraction of a good policy.

The corners of the joint discretization rectangles are shown in Figure 6.1 and their corresponding triangles are shown in 6.2. As can be seen from the figure, the learner concentrated its resources on areas of sharp reward transition, and approximated them quite well. This behavior is expected, as these areas of transition must be represented well in order to obtain a good value function.

For JoSTLe, no a priori action discretization was required. It began with a single

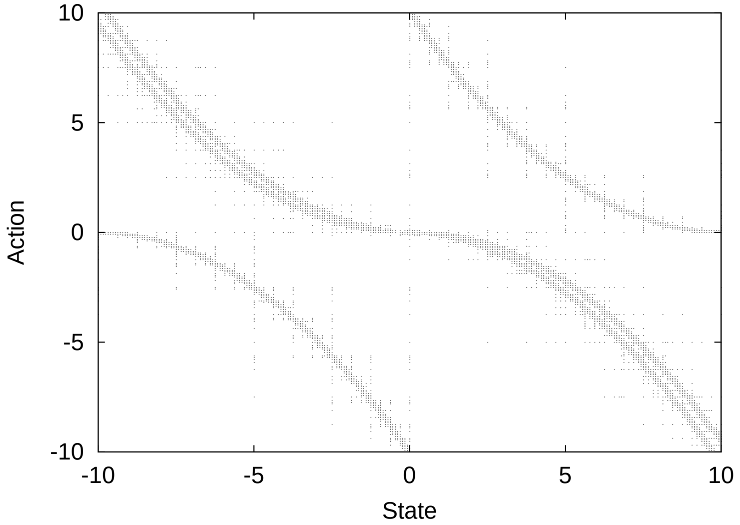

Figure 6.1: Corners of joint space rectangles. This plot shows how well they approximate the true reward boundaries.

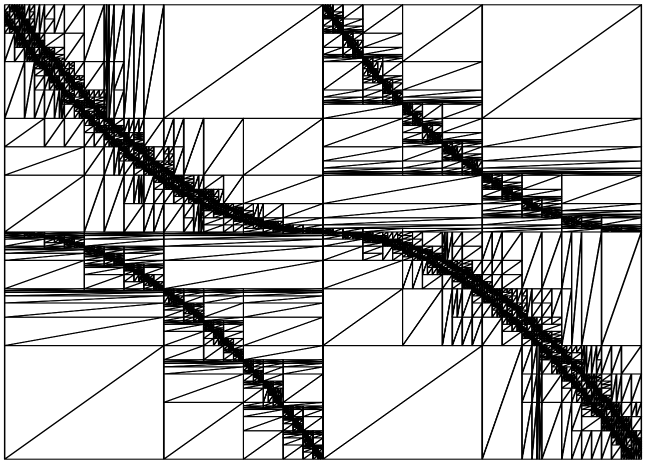

Figure 6.2: A triangulation of the joint space for the golf problem.

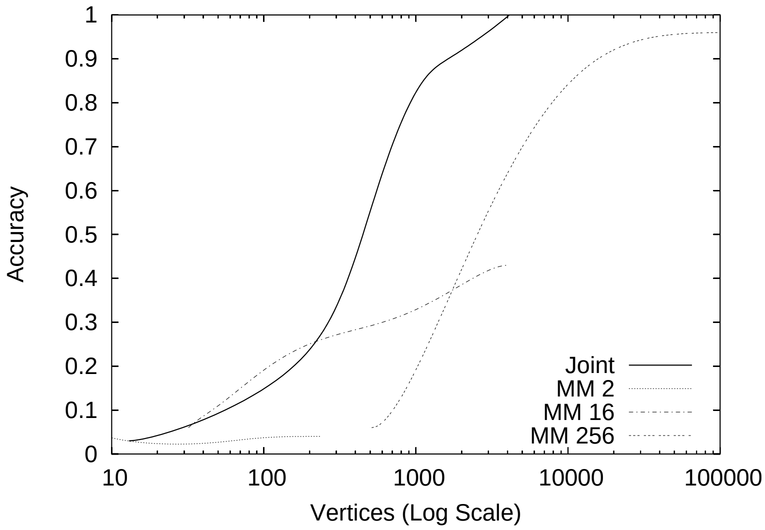

Figure 6.3: The policy accuracy as relates to the number of state/action pairs (vertices in the joint space) is shown for the two different learners. MM gets three different graphs, each of which represents the number of actions available at each state.

joint space rectangle and learned the appropriate discretization over time. The MM approach was given a single line segement to begin with in the state space and was applied using several different uniform action discretizations. The performance of three of them is shown in a later figure.

For each approach, policy accuracy was calculated after every round of splitting and iteration. Since the optimal policy is known for this problem and always consists of a single step, the accuracy was calculated by scanning the state space at 100 evenly-spaced points and querying the models as to which action was most appropriate at each point. The accuracy is the ratio of correct actions to total states queried.

Figure 6.3 shows the relationship between policy accuracy and number of state/action pairs (vertices, in joint space terms) for each learner. Since MM requires an *a priori* action discretization, its performance is shown when given 2, 16, and 256 actions per state.

It should be noted that the numbers used in the experiments were tuned to place the algorithms on as equal a footing as possible. The standard MM algorithm needed a low corner value difference threshold to approach a high accuracy in this problem, while JoSTLe did not need as low of a value disappointment threshold. In fact, to get the graph shown in Figure 6.3, it was necessary to explore the parameter space of JoSTLe in some detail. Several experiments were conducted with different error threshold and minimum feature size values. As these parameters were degraded, JoSTLe finished discretizing at different times and with different accuracies, exhibiting the behavior shown.

It is also important to note that the joint space learner received absolutely no information about the problem before it began learning. It had no idea which action values were important but was left to figure that out automatically. The results shown here indicate that it not only figured it out, but that it did so using fewer resources.

6.1 Complexity

One unfortunate characteristic of JoSTLe when compared to MM is its higher complexity. Because the dimensionality is increased, all of the worst space and time characteristics of MM are exacerbated in JoSTLe. Additionally, JoSTLe must enumerate all $d!$ simplices for a number of nodes during its operation, and then generate all $\binom{d}{d_s}$ boundaries of each of those. MM never performs an enumeration of its $d_s!$ simplices in each cube. Because of the need to find intersections, JoSTLe cannot take advantage of some of the nice properties of Kuhn Triangulations.

6.1.1 Time

Without any optimizations, the roughly worst case time complexity of value iteration for the two systems is shown in the following table. It is assumed that each algorithm has L leaf nodes in its kd-trie, has to take N time steps to reach a new triangle, and that MM has A actions available.

MM	JoSTLe
$2^{d_s} AL \left[N \log_2 L + d_s \log_2 d_s\right]$	$2^d L \left[2NL + Ld! \binom{d}{d_s} d_s! d_s^2\right]$
$= \mathcal{O}_L(L \log_2 L)$	$= \mathcal{O}_L(L^2)$
$= \mathcal{O}_d(2^{d_s} d_s \log_2 d_s)$	$= \mathcal{O}_d(2^d \frac{d!^2}{d_a!} d_s^2)$

The above table deserves some explanation. MM, during value iteration, takes $2^{d_s} LA$ trajectories, each of which goes for N time steps. For each time step, it walks down the kd-trie, which is basically a $\mathcal{O}(\log L)$ operation. Once it has exited its own triangle, it then performs interpolation, which is a $\mathcal{O} d_s \log d_s$ operation.

JoSTLe, on the other hand, does a bit more work, and in a higher dimensionality. It must do value iteration for $2^d L$ trajectories (the action space is not prediscretized, so there is no concept of A), each of which goes for N time steps. For each time step, it must do a range query, which without optimization is a $\mathcal{O}(2N)$ operation (since it must search the result to determine whether it is included or not in the query). Once it has exited its own space, it then enumerates all $d!$ Kuhn Triangles of each node. For each of those triangles, it generates all $\binom{d+1}{d_s+1}$ projection facets. For each facet, it must then perform interpolation, which is unfortunately more complex than interpolation for MM since many of the facets do not have the Kuhn property anymore. The interpolation is a $\mathcal{O}(d_s^2 d_s!)$ operation, since it involves a matrix inverse and vector multiplication.

The complexity is not as bad in practice as it is on paper, fortunately. Many nodes can be culled from the space in JoSTLe during a range query, and many of the triangles can also be culled before doing a projection. A substantial number of the generated facets will be duplicates or degenerate and can also be taken away before ultimately performing the interpolation step. Perhaps more importantly, L can be an order of magnitude smaller for JoSTLe than for MM in practice (Figure 6.3), which serves to alleviate the problem of increased complexity.

The interpolation step can also be sped up significantly by intelligent generation of a vertex-to-facet mapping before value iteration begins, making the algorithm quite practical in many cases. Still, the complexity is not good in the worst case, which would be interesting to address in future research.

6.1.2 Space

The space complexity of JoSTLe in space is determined solely by the kd-trie data structure, and is therefore exactly the same as MM, with two exceptions. The first difference is in the dimensionality of the space being split. JoSTLe has a higher dimensionality because it adds d_a to d_s, where MM only has d_s dimensions to worry about, and a constant set of actions. The second difference is in the fact that JoSTLe often works with a smaller number of nodes in general.

The space complexity of the algorithm was certainly not problematic in practice, even in the higher dimensions, since it tended to learn that some dimensions were more important than others (e.g., Mountain Car required very little action space discretization).

6.2 Parameter Sensitivity

The parameters of interest for JoSTLe are the minimal volume $\mathbf{V}_{\min}$ and the value disappointment threshold ϵ. Of interest when dealing with these parameters is not so much how to make the algorithm perform well (one simply tolerates little disappointment and allows for extremely fine discretization), but how gracefully the performance degrades as the parameters change for the worse.

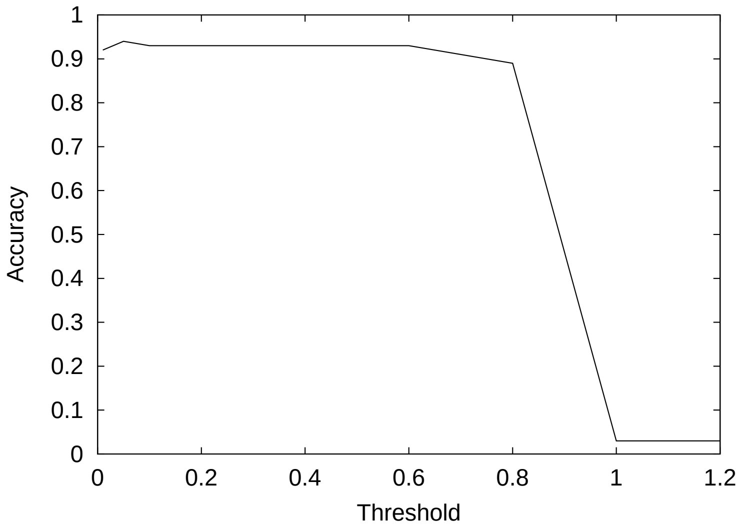

Figure 6.4: The accuracy of JoSTLe as the value disappointment threshold is increased. Since golf is a one-step problem, the performance is still very good until it approaches the maximum reward difference of 1.0.

The value disappointment threshold was varied from 0.01 to 1.2 as shown in Figure 6.4. As can be seen from the figure, the performance of the algorithm remained quite good until the allowable disappointment approached the maximum reward difference in the problem, which is 1.0.

The minimum feature size parameter was the more important parameter, as shown in Figure 6.5. The minimum feature volume is the square of the value on the x-axis, since that value represents the smallest feature in any dimension, and this is a 2-dimensional problem. The algorithm degrades nearly linearly with the increase of the minimum feature size. The other result that is encouraging is that a very high accuracy is still achieved with a minimum feature size of 0.1, even though the reward boundary is thinner than that in some places of the joint space.

It is also of note that the nominal value of 1% of the error bound given in 3.3 was definitely sufficient to solve the problem, and that a far looser bound is still useful.

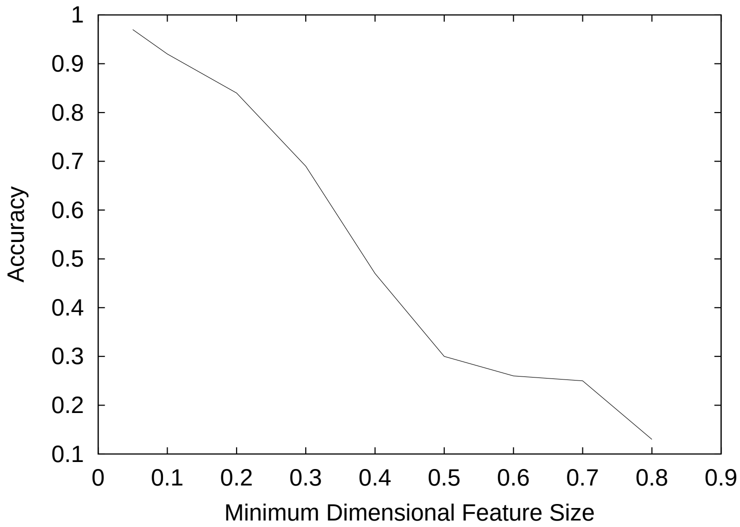

Figure 6.5: The accuracy of JoSTLe as the minimum dimensional feature size is changed. For this graph, the disappointment threshold was set arbitrarily at 0.5.

6.3 Additional Experiments

Problems with only one state and one action are not very interesting. In fact, a control problem is often considered to be minimal when it has 2 state dimensions and 1 action dimension, like the "Mountain Car" problem. For a joint space learner like JoSTLe, this problem is 3-dimensional.

JoSTLe was able to find a nearly optimal policy for the Mountain Car problem without any prior action discretization. Interestingly, in many cases it also did so without discretizing the action space, and in all cases without using the action discretization for the optimal policy. It learned that the Mountain Car problem, because it is minimum time, can be solved with bang-bang control.

An intermediate discretization of the joint space is shown in Figure 6.6. Though JoSTLe eventually discretized the action space, it focused far less attention on it than it did on the state space. The figure shows several views of the discretization corners: the state space discretization, the action space discretization, and the joint

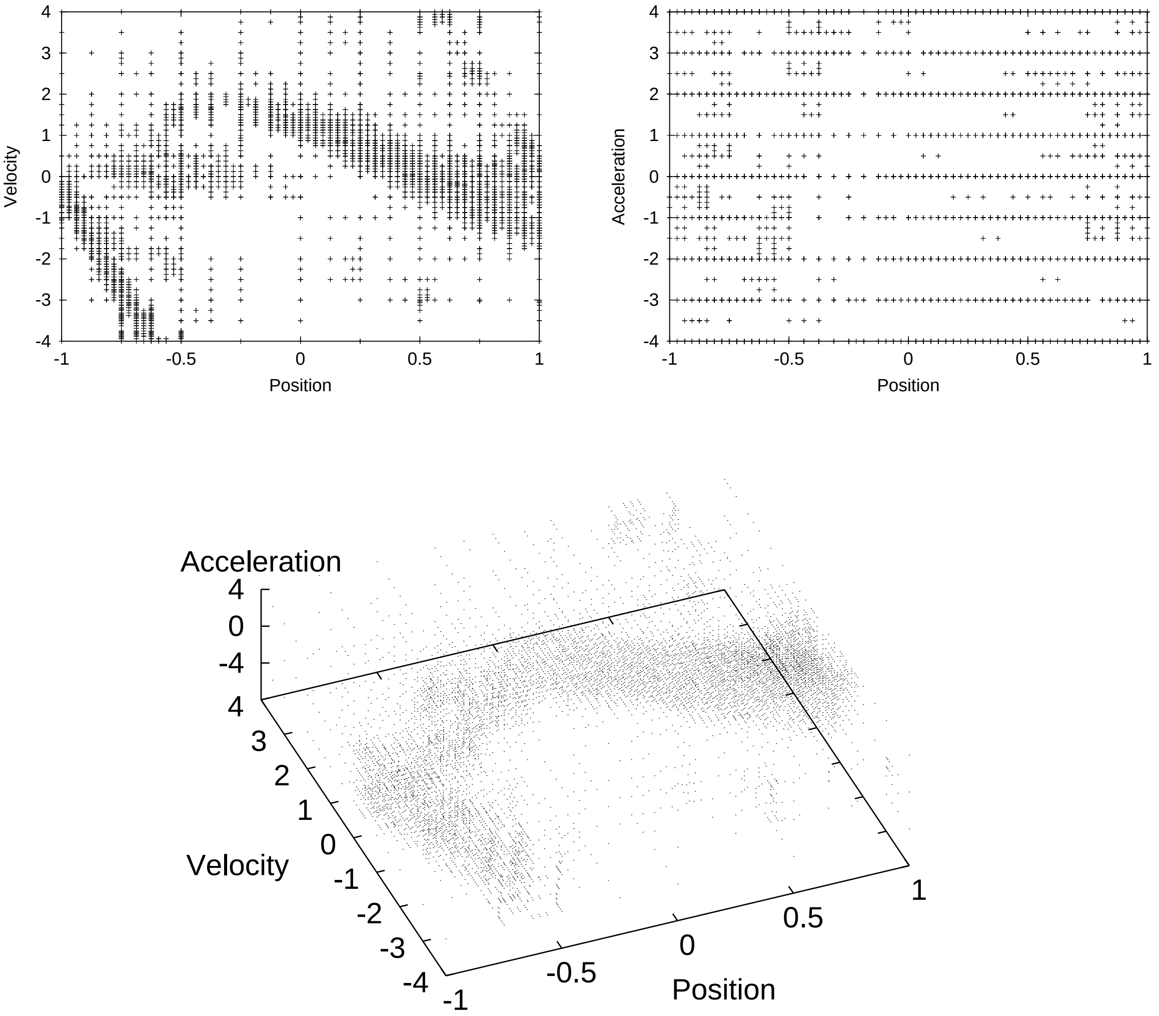

Figure 6.6: An intermediate discretization of the joint space for the Mountain Car problem. Only the rectangle corners are shown. Notice that though the action space is discretized somewhat (upper right), most of the algorithm's attention is focused on the discretization of the state space (upper left). It learned to do this without any prior knowledge of the environment.

space discretization.

Figure 6.7 highlights something even more interesting: though JoSTLe discretized the action space while learning Mountain Car, it generally did not use any intermediate actions in its final policy. It learned bang-bang control. The upper left figure shows a position-dependent projection of the policy. In several places a single intermediate “do nothing” action was chosen, but for the most part attention was focused on the “full forward” and “full reverse” actions. The upper right portion of the figure shows the outline of the policy boundaries. The dots are in areas representing the “full forward” action. The boundary looks very much like the policy learned in [MM02], which is extremely encouraging. Finally, the full policy is shown in the bottom part of the figure.

It is possible to turn JoSTLe into an algorithm much like MM by simply tuning one of the parameters. When the minimum *action* feature size is set to the size of the action space, JoSTLe is forced to not split the action space at all. In this case, it very nearly reduces to MM with the exception that some of the simplex boundary intersections may occur in the intermediate part of the action space.

Some artifacts do exist in the final policy, which is worth consideration. The algorithm is not perfect. The policy shown in Figure 6.7, for example, is only 86.4% correct. It still makes enough mistakes that more research is warranted, especially regarding the way splitting is done.

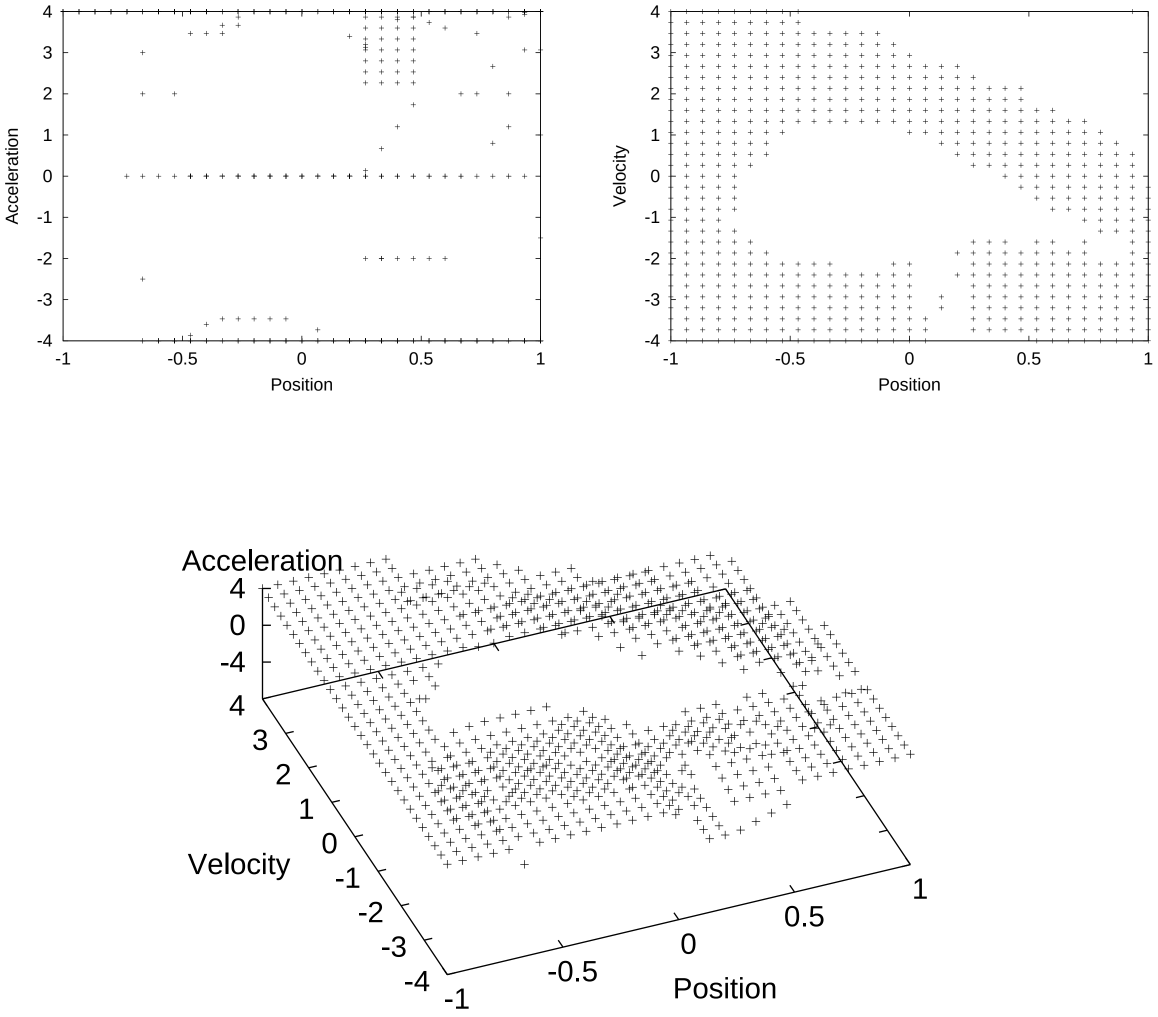

Figure 6.7: The policy learned for Mountain Car. Note that the two projected plots (upper) indicate that most of the time only the extreme actions are used in the policy. The lower graph shows a 3-dimensional plot of the policy.

Chapter 7

Future Research

The algorithm presented here has some potential for some interesting extensions. These areas of future research are presented here.

7.1 Time Complexity

The time complexity of JoSTLe is unfortunately worse than factorial in dimension, and it is also currently N^2 versus MM's $N \log_2 N$ complexity. Some optimizations are probably possible, but have not been explored in detail. Finding a way to minimize the time complexity of JoSTLe is an area of future research.

7.2 Post-Pruned KD-Trie

When value functions have converged to something useful and correct, they are often close to linear in large sections of the space. During the learning process (as information propagates through the space), however, they are not. This means that areas of the joint space must be discretized more finely during learning than is ultimately necessary for the final policy.

An interesting area of future study would be to examine the effects of pruning the kd-trie, effectively recombining adjacent parts of the space that can be approximated just as accurately within a larger portion of space. This is analagous to ID3's reduced error pruning, and could be an interesting area to explore.

7.3 Splitting Criteria

One of Muños and Moore's most interesting contributions to continuous state space RL is the idea of *influence* and *variance* as a means of discretizing only those portions of the space that affect the overall policy. The existence of an analogue to this idea in the joint space is not obvious, but there may be a way of accomplishing the same thing using similar ideas.

It is certain that the splitting criterion used in this paper is not always sufficient for problems of a higher dimensionality, since it tends to refine areas of the space that ultimately have no effect on the correct policy, as witnessed by the very fine discretization of negative reward boundaries in the golf problem.

7.4 Looser Splitting Constraints

Both JoSTLe and MM split regions of space in half. Sometimes it does not make sense to merely split things into two equal pieces. Additionally, the ability to represent oblique features (that are not perpendicular to any one dimensional axis) could allow for more a efficient representation of some problems. Some preliminary work in this area indicates that it could hold some promise, but that the complexity introduced often outweighs any benefits gained. More research is needed.

Chapter 8

Conclusion: Contribution to Computer Science

Joint space learning is a new way of thinking about RL. It allows for a continuous action space, which is something that has not been treated in-depth by the literature, though a class of problems exist that require treatment of the continuous action space.

The insight into spatial projection when using linear interpolation is invaluable in providing tools for working with joint space discretizations. It allows the admittedly increased complexity to be collapsed into a problem with action space search properties equivalent to those found in existing methods.

Additionally, the experience with Delaunay triangulation is valuable. The correction to the incremental deletion algorithm, though ultimately unfruitful for the core of the thesis, is in itself a contribution to a different field. Additionally, the breaking down of undesirable properties of Delaunay may help others to make early and informed decisions regarding its fitness for their particular problem domain.

Finally, this novel approach to the continuous action space and its supporting ideas have opened the doors for what appears to be interesting future research.

Appendix A

Kuhn Triangulation

The Coxeter-Freudenthal-Kuhn Triangulation (or simply the Kuhn Triangulation) is a simple triangulation of a hypercube[1].

The following is an attempt at explaining the information on Kuhn Triangulation found in [MM02]. Much of the description and notation in this section has been lifted from that paper, but is presented here in greater detail to facilitate other implementations.

Each d-dimensional hypercube is split into $d!$ simplices. These simplices can be found by doing the following:

- Label all vertices of the cube in a fixed (but arbitrary) order: $(v_0, v_1, \ldots, v_{d-1})$.
- Generate all permutations on the sequence $(0, 1, \ldots, d-1)$.
- For each permutation $p = (p_0, p_1, \ldots, p_{d-1})$, generate a sequence of indices using the following recurrence relation: $j_0 = 0, j_i = j_{i-1} + 2^{p_i}$. The vertices of the simplex corresponding to this permutation are $(v_{j_0}, v_{j_1}, \ldots, v_{j_{d-1}})$.

One of the great properties of Kuhn Triangulations is that given a point known to be in a particular hypercube, it is very easy to find the simplex that contains that

[1] At least, it is in this case. In general, the triangulation is capable of more than hypercubic tesselation, but for our purposes, this will suffice.

point. Once found, it is also simple to get the barycentric coordinates of the point for use in interpolation.

Given a point $p = (p_0, p_1, \ldots, p_{d-1})$ whose coordinates are relative to the hypercube $h = (v_0, v_1, \ldots, v_{2^d-1})$, the indices of the hypercube that correspond to the point's containing simplex are found by sorting the coordinates of p from highest to lowest: indices $z_0, \ldots, z_{d-1}$ exist such that $1 >= p_{z_0} >= \cdots >= p_{z_{d-1}} >= 0$. The indices of the simplex are found using the recurrence relation above: $j_0 = 0, j_i = j_i - 1 + 2^{z_i}$.

Given a particular set of indices into the vertices of the hypercube, it is trivial to find the barycentric coordinates $\lambda_0, \ldots, \lambda_{d-1}$ of the point in question: $\lambda_0 = 1 - p_{j_0}, \lambda_1 = p_{j_0} - p_{j_1}, \ldots, \lambda_k = p_{j_{k-1}} - p_{j_k}, \ldots, \lambda_{d-1} = p_{j_{d-1}}$.

If each vertex has value f_{v_i}, the interpolated value at point p is simply $\sum_{i=0}^{d-1} \lambda_i f_{v_i}$.

Appendix B

Correction to Incremental Deletion in Delaunay Triangulations

Devillers gives an algorithm for incremental point deletion in Delaunay Triangulations in [Dev99]. The algorithm is correct in 2 dimensions, but many of the assumptions make no sense in 3 and higher dimensions. In fact, the algorithm simply falls apart in higher dimensionalities, even though it is claimed that "generalization to d dimensions is easy".

The algorithm given consists of removing the point and all incident triangles, leaving behind a polygon that must be locally retriangulated. The process used is "ear cutting", which is a basically correct approach. The details on how to select which ears to cut, however, are only correct for 2 dimensions.

B.1 Finding the Ears

One of the first difficulties that arises when trying to extend the ear cutting algorithm to 3 and higher dimensions is that of deciding which adjacent facets form ears and which form mouths. In 2 dimensions, both ears and mouths are formed by connecting two adjacent line segments to form a triangle. The ears are on the interior of the polygon, and the mouths are on the exterior. In 3 dimensions, ears and mouths are formed by connecting adjacent facets (triangles) to form tetrahedrons. Again, ears are interior and mouths are exterior.

Determining whether a given triangle is an ear or a mouth is easy in two dimensions. Simply order the vertices of the adjacent line segments in counterclockwise order (with respect to the polygon), form an ear candidate, and then determine whether the vertices of that ear are also in counterclockwise order (with respect to the triangle). In 3 dimensions, however, the concept of "clockwise" is undefined and finding ears becomes more difficult.

To make this work, we could do complex interior tests (the polytope is not guaranteed to be convex, so these are not simple), or we can make use of another criterion in the paper: the power of the deleted point with respect to the triangle must be negative and maximal.

The power of a point is given by its distance from the circumcenter of the triangle (the formula may be obtained from [Dev99]) and is negative inside of the circumcircle, zero at its boundary, and positive at its exterior. The more negative the power, the closer the point is to the circumcenter. Thus the first ear to be added will be the one whose power is negative and closest to zero. That ear is then removed from consideration, new ears are formed, and the process is repeated until the polytope is triangulated.

It turns out that we don't need to worry about which triangles are ears and which are mouths because mouths aren't the best triangles to add according to this criterion. That problem, therefore, is easy to solve. There is one other problem, however, that is somewhat thorny.

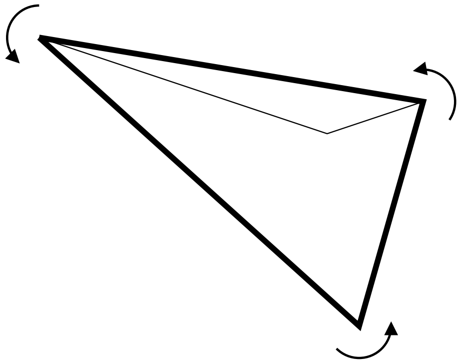

Figure B.1: A 2-d polygon and a candidate ear. The polygon is contained completely within the interior of the triangle. The triangle is not an ear, even though the vertices of the "ear" are in counterclockwise order.

B.2 Which Ear to Add

As stated above, the appropriate ear to add is the one that gives a negative but maximal power for the point in question. This idea comes from shelling the convex hull [Sei86] of the point set when lifted onto a hyperparabloid [For92]. The details of shelling are not discussed here. However, we may consider what happens when we add the ears to a polytope: it becomes increasingly small, and the edges begin to creep inward until there is nothing left to cut (the polytope itself becomes a simplex, and we are finished).

The algorithm given in the paper states that we *always add an ear whose power is negative and maximal.* At any given stage of the algorithm, this consists of adding the triangle whose circumcenter is farthest from the deleted point. This is wrong in general, even in 2 dimensions. While cutting ears, we are sometimes left with a polytope like that of Figure B.1. The polytope is contained completely within the triangle formed by connecting the two lowest line segments, and according to the paper, this is an ear! Again, we could do complex interior tests to determine which triangles are ears, but there is a simpler approach.

What we would really like to do is to add ears with *increasingly negative powers*, or

```
power_threshold = 0.0
new_polytope = polytope
while num_vertices(new_polytope) > d+2:
    candidates = get_all_candidates(new_polytope)
    calculate_powers( candidates, del_point )
    remove_above_threshold( candidates, power_threshold )
    ear = get_maximum( candidates )

    power_threshold = power( ear, del_point )
    new_polytope = add_ear( new_polytope, ear )
```

Figure B.2: The correct ear cutting algorithm.

ears whose circumcenters are increasingly close to the deleted point. Therefore, once an ear has been cut, *no ear with a less negative power may be considered.* After all, the ears that have already been cut are on the edges of the polytope, and therefore come first in the shelling order. The shelling order is defined by ever-decreasing power as the interior of the polygon is cut (see the paper for details). Thus, the algorithm given, which simply places ear candidates into a priority queue and selects the candidate with the highest negative power is wrong. Candidates such as that in the figure can crop up all too easily and mess up the algorithm.

The correct ear cutting algorithm is given in Figure B.2. It has been implemented successfully in high dimensionalities.

Bibliography

[ABK98] N. Amenta, M. Bern, and M. Kamvysselis, *A new Voronoi-based surface reconstruction algorithm*, Proceedings of the SIGGRAPH Computer Graphics Conference (Orlando, Florida), Addison Wesley, 1998, pp. 415–421.

[AS97] C. Atkeson and J. Santamaria, *A comparison of direct and model-based reinforcement learning*, Proceedings of the IEEE International Conference on Robotics and Automation (Albuquerque, New Mexico), IEEE Press, April 1997.

[Atk94] C. Atkeson, *Using local trajectory optimizers to speed up global optimization in dynamic programming*, Proceedings of Advances in Neural Information Processing Systems (Denver, Colorado) (J. D. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, Morgan Kaufmann Publishers, Inc., 1994, pp. 663–670.

[AX00] F. Aurenhammer and Y. F. Xu, *Optimal triangulations*, Encyclopedia of Optimization (C.A.Floudas and P.M.Pardalos, eds.), vol. 4, Kluwer Academic Publishing, 2000, pp. 160–166.

[BH69] A. E. Bryson and Y. C. Ho, *Applied optimal control: Optimization, estimation, and control*, Ginn and Company, Waltham, Massachussetts, 1969.

[BK93] L. C. Baird and A. H. Klopf, *Reinforcement learning with high-dimensional, continuous actions*, Tech. Report WL-TR-93-1147, Wright-Patterson Air Force Base Ohio, 1993.

[BLRG00] A. Below, J. De Loera, and J. Richter-Gebert, *The complexity of finding small triangulations of convex 3-polytopes*, Los Alamos National Laboratory arXiv:math.CO/0012177, 2000.

[BM95] J. A. Boyan and A. W. Moore, *Generalization in reinforcement learning: Safely approximating the value function*, Proceedings of Advances in Neural Information Processing Systems (Denver, Colorado) (G. Tesauro, D. S. Touretzky, and T. K. Leen, eds.), vol. 7, The MIT Press, 1995, pp. 369–376.

[BT93] J. Boissonnat and M. Teillaud, *On the randomized construction of the Delaunay tree*, Theoretical Computer Science **112** (1993), no. 2, 339–354.

[CM] P. Cichosz and J. J. Mulawka, *Integrated architectures for learning, planning, and reacting based on approximating TD(lambda)*, EPrint at http://citeseer.nj.nec.com/392847.html.

[Dav97] S. Davies, *Multidimensional triangulation and interpolation for reinforcement learning*, Proceedings of Advances in Neural Information Processing Systems (Denver, Colorado) (M. C. Mozer, M. I. Jordan, and T. Petsche, eds.), vol. 9, The MIT Press, 1997, p. 1005.

[Dev98] O. Devillers, *Improved incremental randomized Delaunay triangulation*, Proceedings of the Fourteenth Annual Symposium on Computational Geometry (Minneapolis, Minnesota), ACM Press, 1998, pp. 106–115.

[Dev99] ———, *On deletion in Delaunay triangulations*, Proceedings of the fifteenth annual symposium on Computational Geometry (Miami Beach, Florida), ACM Press, 1999, pp. 181–188.

[DGH01] T. Dey, J. Giesen, and J. Hudson, *A Delaunay based shape reconstruction from large data*, Proceedings of the IEEE Symposium in Parallel and Large Data Visualization and Graphics (San Diego, California), IEEE Press, 2001, pp. 19–27.

[For92] S. J. Fortune, *Voronoi diagrams and Delaunay triangulations*, Euclidean Geometry and Computers (1992), 193–233.

[Gor95] G. J. Gordon, *Stable function approximation in dynamic programming*, Proceedings of the Twelfth International Conference on Machine Learning (San Francisco, CA) (Armand Prieditis and Stuart Russell, eds.), Morgan Kaufmann Publishers, Inc., 1995, pp. 261–268.

[HA98] R. Heckendorn and C. Anderson, *A multigrid form of value iteration applied to a markov decision problem*, Tech. Report CS-98-113, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 1998.

[KLM96] L. P. Kaelbling, M. L. Littman, and A. P. Moore, *Reinforcement learning: A survey*, Journal of Artificial Intelligence Research **4** (1996), 237–285.

[MAS95] A. W. Moore, C. Atkeson, and S. Schaal, *Memory-based learning for control*, Tech. Report CMU-RI-TR-95-18, Carnegie Mellon University, Pittsburg, Pennsylvania, 1995.

[MM98] R. Muños and A. W. Moore, *Barycentric interpolators for continuous space and time reinforcement learning*, Proceedings of Advances in Neural Information Processing Systems (Denver, Colorado), vol. 11, Morgan Kaufmann Publishers, Inc., December 1998.

[MM02] ______, *Variable resolution discretization in optimal control*, Machine Learning **49** (2002), 291–323.

[Moo91] A. W. Moore, *Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued state-spaces*, Eighth International Machine Learning Workshop (Northwestern University, Illinois) (L. A. Birnbaum and G. C. Collins, eds.), The MIT Press, 1991, pp. 333–337.

[Moo94] ______, *The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces*, Proceedings of Advances in Neural Information Processing Systems (San Francisco, California) (J. D. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, Morgan Kaufmann Publishers, Inc., 1994, pp. 711–718.

[MP94] R. Muños and J. Patinel, *Reinforcement learning with dynamic covering of state-action space: Partitioning q-learning*, Proceedings of the International Conference on Simulation of Adaptive Behavior (Brighton, United Kingdom), vol. 3, MIT Press, 1994.

[MSBL98] A. W. Moore, J. G. Schneider, J. A. Boyan, and M. S. Lee, *Q2: Memory-based active learning for optimizing noisy continuous functions*, Proceedings of the 15th International Conference on Machine Learning (San Francisco, California), Morgan Kaufmann Publishers, Inc., 1998, pp. 386–394.

[Müc95] E. P. Mücke, *A robust implementation for three-dimensional Delaunay triangulations*, Proceedings of the 1st International Computational Geometry Software Workshop (Minneapolis, Minnesota), University of Minneapolis, 1995, pp. 70–73.

[Muñ96] R. Muños, *A convergent reinforcement learning algorithm in the continuous case: The finite-element reinforcement learning*, Proceedings of the International Conference on Machine Learning (Bari, Italy), Morgan Kaufmann Publishers, Inc., 1996.

[Muñ97] ———, *Finite-element methods with local triangulation refinement for continuous reinforcement learning problems*, Proceedings of the European Conference on Machine Learning (Prague, Czech Republic), Springer, 1997.

[Rey99] S. I. Reynolds, *Decision boundary partitioning: Variable resolution model-free reinforcement learning*, Tech. Report CSRP-99-15, School of Computer Science, The University of Birmingham, Birmingham, United Kingdom, July 1999.

[Sei86] R. Seidel, *Constructing higher-dimensional convex hulls at logarithmic cost per face*, Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing (Berkeley, California), ACM Press, 1986, pp. 404–413.

[Tan93] T. S. Tan, *Optimal two-dimensional triangulations*, Ph.D. thesis, University of Illinois, Urbana-Champaign, Illinois, 1993.