This thesis presents a progression of novel planning algorithms that culminates in
a new family of diverse Monte-Carlo methods for probabilistic planning domains.
We provide a proof for performance guarantees and analyze how these algorithms
can resolve some of the shortcomings of traditional probabilistic planning
methods. The direct policy search methods we developed in collaboration with
our local fire department resulted in counter-intuitive recommendations to
improve response times for emergency responders. We describe a new family of
planning algorithms that combines existing Monte-Carlo methods such as UCT
and Sparse Sampling. These algorithms have resulted in groundbreaking
empirical performance bounds for one of the most popular computer games of all
time, Klondike Solitaire, solving more than 40% of randomly drawn games.
These new results significantly improve over our own previously reported results,
which represented the first published bound of any kind for Klondike. This builds
upon our foundational work that improved performance bounds for a variant of Klondike where the identity of face down cards is known, solving more than 82% of games.
Monte-Carlo Planning for Probabilistic Domains

by

Ronald Vance Bjarnason, Jr.

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented December 1, 2009
Commencement June 2010
Doctor of Philosophy dissertation of Ronald Vance Bjarnason, Jr. presented on December 1, 2009.

APPROVED:

________________________
Co-Major Professor, representing Computer Science

________________________
Co-Major Professor, representing Computer Science

________________________
Director of the School of Electrical Engineering and Computer Science

________________________
Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

________________________
Ronald Vance Bjarnason, Jr., Author
ACKNOWLEDGEMENTS

Many have helped me along this path.

Foremost, I love and appreciate my wife, Denise, for her support and her patience.

I recognize my fellow graduate students, especially Neville Mehta, Charlie Parker, Aaron Wilson, Sriraam Natarajan, Ethan Dereszynski, Jana Doppa and Scott Proper. Working together provided years of insightful conversation and lively discussion, with occasional brilliant forays into research and academic topics.

My respect and appreciation go to Prasad and Alan. They deserve more credit than they typically receive.

This research has been supported by the National Science Foundation under Grant IIS-0329278, by DARPA under grant number FA8650-06-C-7605 and by the Army Research Office under grant number W911NF-09-1-0153. Special thanks to the Corvallis Fire Department for valuable consultation and the data logs which were invaluable to our work on the Fire, Rescue and Emergency Response domain.
# TABLE OF CONTENTS

1  Introduction to Probabilistic Planning  
   1.1 Previous Approaches  
   1.2 Monte-Carlo Approaches  
   1.3 Domains  
      1.3.1 Klondike Solitaire  
      1.3.2 Thoughtful Solitaire  
      1.3.3 Fire Rescue and Emergency Response  
   1.4 Overview  

2  Probabilistic Planning Frameworks  
   2.1 Reinforcement Learning and the Bellman Equations  
      2.1.1 Value Iteration and Policy Iteration  
      2.1.2 Temporal Difference Learning  
   2.2 Planning and Complex Search  
   2.3 Monte-Carlo Methods  
      2.3.1 Policy Rollout  
      2.3.2 Hindsight Optimization  
      2.3.3 UCT  

3  Direct Policy Search  
   3.1 Fire, Rescue and Emergency Response  
   3.2 CFD System and Domain  
   3.3 Simulation  
      3.3.1 Response and Travel Time  
      3.3.2 Assignment Duration  
      3.3.3 Personnel, Vehicles and Units  
      3.3.4 Response District Grid  
   3.4 Optimization Methods  
      3.4.1 Reward  
      3.4.2 Learning Unrestricted Policies  
      3.4.3 Learning Restricted Policies  
   3.5 Evaluation of Results  
      3.5.1 Policy Comparison  

Page
---
1  
3  
4  
5  
5  
7  
8  
10  
12  
14  
15  
17  
19  
21  
22  
23  
24  
26  
26  
28  
31  
31  
31  
32  
33  
34  
35  
36  
38  
42  
42
<table>
<thead>
<tr>
<th>Page</th>
<th>TABLE OF CONTENTS (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>4 Deterministic Planning with Rollouts</td>
</tr>
<tr>
<td>49</td>
<td>4.1 Klondike Solitaire and Variants</td>
</tr>
<tr>
<td>52</td>
<td>4.1.1 Thoughtful Solitaire</td>
</tr>
<tr>
<td>52</td>
<td>4.1.2 A New State Representation: K(^+) Solitaire</td>
</tr>
<tr>
<td>53</td>
<td>4.2 Traversing the Search Tree</td>
</tr>
<tr>
<td>55</td>
<td>4.2.1 Staged Rollouts</td>
</tr>
<tr>
<td>56</td>
<td>4.3 The Thoughtful Solitaire Solver</td>
</tr>
<tr>
<td>57</td>
<td>4.3.1 Pruning</td>
</tr>
<tr>
<td>59</td>
<td>4.3.2 Caching Visited States</td>
</tr>
<tr>
<td>60</td>
<td>4.3.3 Global Loop Prevention</td>
</tr>
<tr>
<td>60</td>
<td>4.3.4 Action Restrictions and Ordering</td>
</tr>
<tr>
<td>61</td>
<td>4.4 Empirical Results</td>
</tr>
<tr>
<td>62</td>
<td>4.4.1 Thoughtful Lower Bounds</td>
</tr>
<tr>
<td>63</td>
<td>4.4.2 Performance Under Time Constraints</td>
</tr>
<tr>
<td>66</td>
<td>5 Probabilistic Planners for Klondike</td>
</tr>
<tr>
<td>68</td>
<td>5.1 Hindsight Optimization and Averaging Over Clairvoyancy</td>
</tr>
<tr>
<td>71</td>
<td>5.2 UCT Approaches</td>
</tr>
<tr>
<td>75</td>
<td>5.2.1 Combining UCT with HOP</td>
</tr>
<tr>
<td>76</td>
<td>5.2.2 Ensemble-UCT</td>
</tr>
<tr>
<td>77</td>
<td>5.2.3 Results</td>
</tr>
<tr>
<td>78</td>
<td>5.3 Sparse UCT</td>
</tr>
<tr>
<td>80</td>
<td>5.3.1 Results</td>
</tr>
<tr>
<td>83</td>
<td>6 A Unified Approach</td>
</tr>
<tr>
<td>83</td>
<td>6.1 Performance Characteristics of Sparse-Ensemble-UCT</td>
</tr>
<tr>
<td>84</td>
<td>6.1.1 Evaluating Time</td>
</tr>
<tr>
<td>88</td>
<td>6.1.2 Evaluating Performance</td>
</tr>
<tr>
<td>91</td>
<td>6.2 Analysis of Sparse UCT</td>
</tr>
<tr>
<td>95</td>
<td>7 Exploring Other Update Methods: Expectimax UCT</td>
</tr>
<tr>
<td>95</td>
<td>7.1 Expectimax Updates</td>
</tr>
<tr>
<td>98</td>
<td>7.2 Expectimax Updates in Practice</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>8 Learning Default Behavior</td>
<td>103</td>
</tr>
<tr>
<td>8.1 Linear Function Approximation Features for Klondike</td>
<td>104</td>
</tr>
<tr>
<td>8.2 Adjusting Behavior of the Learning Algorithm</td>
<td>105</td>
</tr>
<tr>
<td>8.3 Interpretation of Learned Weights</td>
<td>107</td>
</tr>
<tr>
<td>8.3.1 General Conclusions</td>
<td>110</td>
</tr>
<tr>
<td>8.4 Performance of UCT with Improved Default Policy</td>
<td>112</td>
</tr>
<tr>
<td>9 Contributions of This Research</td>
<td>115</td>
</tr>
<tr>
<td>9.1 Algorithmic Contributions in Ensemble and Sparse UCT Variants</td>
<td>115</td>
</tr>
<tr>
<td>9.2 New Approaches for Probabilistic Planning</td>
<td>116</td>
</tr>
<tr>
<td>9.3 Klondike and Thoughtful Solitaire</td>
<td>117</td>
</tr>
<tr>
<td>9.4 Fire, Rescue and Emergency Response</td>
<td>118</td>
</tr>
<tr>
<td>10 Discussion and Future Work</td>
<td>120</td>
</tr>
<tr>
<td>10.1 Probabilistic Planning</td>
<td>120</td>
</tr>
<tr>
<td>10.2 Bounds for Klondike and Thoughtful Solitaire</td>
<td>121</td>
</tr>
<tr>
<td>10.3 Fire, Rescue and Emergency Response</td>
<td>121</td>
</tr>
<tr>
<td>Bibliography</td>
<td>121</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The Windows version of Klondike Solitaire.</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Map of Corvallis, OR. The five fire stations within the city limits are listed. An additional station is located three miles north of station 3 in the town of Lewisburg.</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Basic descriptions of (a) standard rollouts and (b) nested rollouts and (c). A heuristic, $h$, evaluates a state, $s$, with $h(s)$. Each $n$ is a level of nesting, associated with a heuristic. At each level of nesting, the algorithm attempts to determine which action, $a$, should be taken.</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Representation of search paths taken for (a): greedy search (analogous to level 0 nested rollouts) and (b): standard rollouts (level 1 nested rollouts)</td>
<td>23</td>
</tr>
<tr>
<td>3.1</td>
<td>Comparison of response policies over four metrics to various baselines including a running order approximation of an LSPI policy, an approximation of current CFD policy, a “closest station first” (CSF) policy, and a version of our SOFER policy initialized by our CFD policy. Our SOFER policy was driven by a reward function defined over various response features. We demonstrate the adaptability of SOFER by reporting results when the weights of two of these features (OP = penalty for overwhelming the system, FR = elapsed time when first responder arrives) are eliminated (=0) and when they are multiplied by 10.</td>
<td>46</td>
</tr>
<tr>
<td>4.1</td>
<td>A screenshot of Klondike Solitaire. Each card stack is labeled with one of four types: the Talon, the Stock, the 4 Foundation Stacks and the 7 Tableau Stacks.</td>
<td>51</td>
</tr>
<tr>
<td>4.2</td>
<td>An illustration of the search tree compression resulting from the elimination of the Stock. The search tree for Klondike is compressed by eliminating each of the <em>Turn Stock</em> actions.</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>A graphical display of $K^+$ Solitaire.</td>
<td>54</td>
</tr>
<tr>
<td>4.4</td>
<td>A description of multistage nested rollouts. A heuristic, $h$, evaluates a state, $s$, with $h(s)$. Each $n$ is a level of nesting, associated with a heuristic. At each level of nesting, the algorithm attempts to determine which action, $a$, should be taken.</td>
<td>55</td>
</tr>
</tbody>
</table>
LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5</td>
<td>Representation of search paths taken for (a): multistage nested rollouts with two heuristics, each with level 0 search, and (b): multistage nested rollouts with two heuristics, each with level 1 search.</td>
</tr>
<tr>
<td>4.6</td>
<td>The value functions used to determine the value of a state in the search tree. H1 is designed for opening the game and H2 for endgame scenarios. The total number of occurrences of each feature type is listed in parentheses.</td>
</tr>
<tr>
<td>4.7</td>
<td>(a) A dead end detected in the relaxed domain. The Q♠ is blocking the K♥, K♦, and 2♠. (b) A dead end undetected in the relaxed domain, but detected as a special case pattern. After the 10♦ is built upon the J♠, the 10♦ and the 10♥ have no viable path to the Foundation stacks, as the 10♥ it is blocking the J♣, 9♦ and 5♥.</td>
</tr>
<tr>
<td>4.8</td>
<td>A description of the multistage nested rollouts algorithm with modifications described in Section 4.3. On line 2, the history from the current state back to the root search node is checked for loops. On line 4, the current state is checked against the cache for the current heuristic and nesting magnitude.</td>
</tr>
<tr>
<td>4.9</td>
<td>Results for rollout tests, from our solver (a), and as reported in Yan et al., 2005 (b). The degree of nesting is indicated by $n$ when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by $n_0$ for H1, and $n_1$ for H2. Results in (b) used a different heuristic which required a nested rollouts method slightly different from the method described in Figure 2.1. H1 and H2 are described in Figure 4.6.</td>
</tr>
<tr>
<td>4.10</td>
<td>Results for tests under time constraints. The degree of nesting is indicated by $n$ when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by $n_0$ for H1, and $n_1$ for H2. Use of multiple heuristics wins a higher percentage of games in a shorter time period than using a single heuristic. H1 and H2 are described in Figure 4.6.</td>
</tr>
<tr>
<td>4.11</td>
<td>Comparison of winning percentage (as a function of time) for various levels of nesting of the multistage nested rollout algorithm. Error bars indicate 99% confidence intervals.</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES (Continued)

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>A simple example provided in [46] when correlated futures will cause a deterministic planner to miscalculate the value of an action, in this case, the value of action $b$ from the root. Taking action $a$ will surely lead to a win, while taking action $b$ will only lead to a win a fraction of the time.</td>
</tr>
<tr>
<td>5.2</td>
<td>(a) A state in Klondike Solitaire. (b) A possible state after $K\spadesuit\rightarrow T4$. (c) A dead end forced by having to guess.</td>
</tr>
<tr>
<td>6.1</td>
<td>A graphical illustration of the variants of UCT in terms of three variables: number of trees per decision, number of trajectories used to construct each tree and the action branching factor of those trees. UCT, Sparse-UCT, Ensemble UCT and other variants can be described in terms of these variables.</td>
</tr>
<tr>
<td>6.2</td>
<td>A comparison of the time per game (graphed on a log scale) as a function of the number of trajectories used to generate a UCT tree and the size of the allowed branching factor of those UCT trees.</td>
</tr>
<tr>
<td>6.3</td>
<td>A comparison of the time per game (graphed on a log scale) as a function of the number of trees per decision ensemble.</td>
</tr>
<tr>
<td>6.4</td>
<td>A comparison of the performance of a UCT tree based on the number of trajectories per tree (graphed on a log scale) and the size of the allowed branching factor. The data points for branching factor of 1 and 21 were generated with a local search to remove duplicate states.</td>
</tr>
<tr>
<td>6.5</td>
<td>A comparison of how performance increases with three variables that define our family of UCT algorithms: 1) # of trees in the ensemble, 2) # of trajectories used to construct those trees and 3) the branching factor. Performance is measured against the average time to complete each game (on a log scale). Points that appear on the upper left edge can solve the most games in the smallest amount of time. Based on these trajectories, it is initially most beneficial to add trajectories to existing trees, but in the limit it appears that highest performance for the search time can be achieved by increasing branching factor, ensemble size and trajectories per tree.</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7.1</td>
<td>An example of a simple search tree in which average reward update may lead UCT exploration to temporarily underestimate the true value of taking action $b$ from the root node, $S_0$, where expectimax updates using the same exploration policy will correctly evaluate the value of the same action with far fewer trajectories.</td>
</tr>
<tr>
<td>7.2</td>
<td>An example from [14] as a bad example for UCT, potentially leading to hyper-exponential regret.</td>
</tr>
<tr>
<td>7.3</td>
<td>An example of expectimax updating in domains with both deterministic and non-deterministic transitions. After following a series of deterministic actions (marked $a_{dt}$), a non-deterministic action leads to a (possibly rare) eventual win. As a result, state values along the entire trajectory are initially updated to a value of 1 and maintained throughout possibly exponential additional exploration until action $a_{nd}$ is taken again from state $S_i$.</td>
</tr>
<tr>
<td>8.1</td>
<td>Learned weights for feature Class 1 by card rank.</td>
</tr>
<tr>
<td>8.2</td>
<td>Learned weights for feature Classes 2, 3 and 4 by card rank.</td>
</tr>
<tr>
<td>8.3</td>
<td>Learned weights for feature Classes 5, 6 and 7 by card rank.</td>
</tr>
<tr>
<td>8.4</td>
<td>Learned weights for feature Class 8 by maximum depth of face down cards.</td>
</tr>
<tr>
<td>8.5</td>
<td>Comparison of learned weights for all feature classes.</td>
</tr>
<tr>
<td>8.6</td>
<td>A comparison of previously published UCT results (found in Table 5.2) with more recent results described in Table 8.2. Percentage of games won (on a linear scale) is measured against the time required to finish each game (on a log scale). Thus results on the upper left ridge represent those algorithms that win the most games in the smallest amount of time. It is clear that the yet unpublished improvements which include local search for repeated states and the use of improved default policies have improved our performance significantly.</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>a: Default station assignments for the 8 units in our simulation. b: The simulated performance of this configuration using an approximated CFD running order compared to figures reported by CFD.</td>
</tr>
<tr>
<td>3.2</td>
<td>results from LSPI and an approximated running order based on the simulated LSPI policy.</td>
</tr>
<tr>
<td>3.3</td>
<td>Two example configurations generated by SOFER. Configurations X and Y were generated with rewards in which the FR component of the reward had ten times its normal value, and where it was eliminated completely, respectively. Both leave Station 5 unstaffed. Configuration Y also leaves station 4 unstaffed. These are compared to our simulated CFD policy and a “closest station first” policy, based on four metrics.</td>
</tr>
<tr>
<td>5.1</td>
<td>Results for various HOP tests</td>
</tr>
<tr>
<td>5.2</td>
<td>Results for various UCT tests. These results, presented at ICAPS 2009 [4] did check to eliminate duplicate states in the UCT tree.</td>
</tr>
<tr>
<td>7.1</td>
<td>Results for various UCT tests</td>
</tr>
<tr>
<td>8.1</td>
<td>A description of the 8 different classifications for the 371 features that define the heuristic value function for Klondike Solitaire.</td>
</tr>
<tr>
<td>8.2</td>
<td>A learned default policy is compared to the random policy and to an action-priority heuristic. The policies are compared as 1-step look-ahead greedy policies as well as default behavior within a UCT policy. The greedy tests were performed on 1 million games, while the UCT tests were performed on 9083 games. Identical games were tested for all three policies for both sets of tests. UCT decisions were made with 10 independent UCT trees, each generated by 100 trajectories, without restriction to branching factor.</td>
</tr>
</tbody>
</table>
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Two-phase hill climbing optimization</td>
<td>39</td>
</tr>
<tr>
<td>2</td>
<td>Running Order Optimization Method</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Unit Assignment Optimization</td>
<td>40</td>
</tr>
<tr>
<td>4</td>
<td>Generate UCT Tree Algorithm</td>
<td>79</td>
</tr>
</tbody>
</table>
DEDICATION

To Denise, Charley, Joseph, Sam and Ben
Chapter 1 – Introduction to Probabilistic Planning

Planning is a general problem encountered in even our most mundane tasks. We formulate travel plans each morning: How long will the drive be? How much time does this leave for eating breakfast? For fixing our hair and brushing our teeth? We formulate plans for attending conferences, beginning months in advance as we plan time for running experiments, meeting in committees and writing papers. In recent years, we have been able to pass this burden on to commercial planning solutions, such as a GPS unit for our car, or a desktop assistant that forecasts the required steps to our goal. Simply stated, planning is nothing more than designing a series of actions or decisions to achieve a goal based on access to a model. In these examples, our internal model is based on our personal experiences with driving or writing research papers or personal hygiene.

Classical planning theory is typically restricted to clean, deterministic, fully observable problems – very different from the planning we do in our everyday lives. The practical planning we do every day is dynamic and subject to unpredictable events. Poor driving conditions or a traffic accident will delay your drive. A research assistant may be conducting experiments completely different from those you requested and this misunderstanding/bug/insubordination will push back your research agenda. Somehow, despite plans to submit your paper a day early, there’s always a rush to make changes right at the 3am deadline [4]. The branch of
planning that solves these types of problems is known as probabilistic planning, which injects randomness into the classical planning problem. Agents may not be able to observe all pertinent features of their state. Results of their actions may be stochastic. The environment itself may change unpredictably. In each of these cases, the probabilistic planning agent will still rely on some model to make decisions.

In this thesis, we will examine our contributions to research in the field of probabilistic planning. This research has a focus on methods developed for domains too large and complex for other contemporary solutions. Our research has produced significant results in the grounded real world problem of emergency response optimization and the first published performance bounds for the common card game, solitaire. The algorithms developed for these domains directly address some of the shortcomings of existing solutions methods beyond the obvious scalability issues. The simulation based SOFER method we introduced for the Fire, Rescue and Emergency Response (FRER) learned policies within the policy restrictions required by local officials. Our work with Monte-Carlo planning algorithms revealed that current probabilistic planning languages cannot compactly represent certain interesting domains such as solitaire. We will first examine the historical and contemporary approaches to solving these probabilistic planning problems and describe our own contributions in this area.
1.1 Previous Approaches

Algorithms for probabilistic planning have generally borrowed heavily from successful algorithms from classical (deterministic) planning such as the GRAPHPLAN [7] and SATPLAN [22] algorithms. Like most approaches to classical planning, these algorithms attempt to find a path from some initial state in the domain to a pre-defined goal state in as few steps as possible. For trivial problems, this can be accomplished with a straightforward brute force search algorithm. If the goal state is included in one of the paths, then that path is a solution to the problem. If no such path exists, then there is no solution. In essence, classical planning attempts to reproduce the results of a brute force search with as few resources as possible and research in probabilistic planning has echoed these efforts, utilizing forward searches, similar to depth first tree search, or backward searches that work in reverse from the goal state in the style of GRAPHPLAN.

The fact that these methods use a search tree at all makes them inherently unscalable. For probabilistic/stochastic planning to scale to large domains will require new models that are not so tightly bound to the exponential growth of the search trees. Existing algorithms that meet these criteria, such as sparse sampling [23] and roll out methods [41] rely on Monte-Carlo sampling techniques that have not traditionally been directly associated with planning problems. Because Monte-Carlo methods estimate the value of individual actions through repeated random sampling and simulation, their running times do not directly increase with the exponential expansion of the search tree. These algorithms trade accuracy for
speed and scalability. While the resources required to find an optimal plan still increases exponentially with the size of the domain for these algorithms, they are able to perform well because in general, these probabilistic planning domains do not require highly optimal plans.

1.2 Monte-Carlo Approaches

Most recently, probabilistic planning methods have seen algorithms that combine these Monte-Carlo techniques with successful classical planners. At the 2004 International Planning Competition, the FF-Replan [45] algorithm was entered as a strawman baseline in the probabilistic planning track. To everyone's surprise, it handily won the competition. FF-REPLAN makes optimistic assumptions about the stochastic characteristics of the environment, and follows a deterministic planner based on those assumptions. When it reaches a state outside of its plan, it simply replans with new optimistic assumptions. Hindsight Optimization (HOP) [46] combines some of the principles of FF-Replan with simulation. Rather than form an entire plan a-priori, HOP estimates the value of each possible action at every decision point through multiple Monte-Carlo simulations from each action. After the simulations have run, HOP chooses the action that was, on average, most successful. Other work, such as the tree generating UCT algorithm [24] has been critical to advancements in domains outside of the scope of planning. These Monte-Carlo simulation-based methods have been shown to be surprisingly effective in the simple stochastic domains that have been the standard testbeds for
traditional probabilistic planning algorithms for years. Additionally, without the need to compute an entire plan and the reliance on simulation, these methods avoid the issue of generating a compact plan representation, and are able to scale to extremely large domains, most notably the game of Go, in which UCT-based computer players are currently the most successful in the world [19].

1.3 Domains

Our work has been conducted in three different planning domains. Klondike Solitaire and the Fire, Rescue and Emergency Response (FRER) domain are probabilistic planning domains while Thoughtful Solitaire is a deterministic domain. These domains are diverse and provided distinct challenges to contemporary planners. The necessary restrictions to the learned policies in the FRER domain eliminated many of the candidate solution techniques. Thoughtful and Klondike are very large domains with search trees far too deep and wide for planning techniques based on exhaustive forward or backward searches. The new methods developed to overcome the difficulties presented by these domains represents a significant contribution to research in the planning community.

1.3.1 Klondike Solitaire

Klondike Solitaire (commonly referred to simply as “Solitaire”) has recently been named the most important computer game of all time [28]. Despite its broad
appeal and near-ubiquitous presence, modern researchers have declared that “it is one of the embarrassments of applied mathematics that we cannot determine the odds of winning the common game of Solitaire” [43]. The Klondike Solitaire domain is problematic due to a large branching factor and the depth required to reach a goal state. Klondike Solitaire is a simple game played with a standard deck of 52 playing cards. Initially, 28 cards are dealt to 7 tableau stacks, 1 card to stack 1, 2 to stack 2, ... etc, with the top card of each stack face up. The remaining 24 cards are placed in a deck. Four foundation stacks (one for each suit) are initially empty. An instance of the common Windows version of Klondike can be seen in Figure 1.1. The object of the game is to move all 52 cards from the deck and the tableau to the foundation stacks. A more complete description of Klondike game play is included in Chapter 4.1. Variants of Klondike define specific elements of game play. Our research has dealt exclusively with a common variant that allows unlimited passes through the deck, turning three cards at a time, and allowing partial tableau stack moves.

Anecdotal evidence suggests that typical human players win between 7 and 15% of games [16, 35]. One common method of scoring games, known as “Las Vegas” style, pays players five fold their up-front per-card cost for each card that reaches a foundation stack. Such a payoff suggests that a strategy that wins over 20% of played games should be considered a success. In addition to these benchmarks, we constructed a greedy policy based on a simple preference over action types that was able to win 13% of games. Our initial efforts using Monte Carlo methods solved a surprising 28% of games. Our experimentations with other Monte Carlo
methods introduced a new family of algorithms based on UCT, Sparse Sampling and ensemble methods improved this to over 35%. This work was presented at ICAPS-2009 [4] and was recognized with a “Best Student Paper” award and has resulted in an invitation to submit an extended version of this paper for fast-track publication in Artificial Intelligence, one of the oldest and most respected journals in the field. Additional improvements have increased performance to over 40%. To our knowledge, our results provide the first performance bounds of any kind for general Klondike strategies.

1.3.2 Thoughtful Solitaire

Our work in Solitaire began in a variant of the game known as “Thoughtful Solitaire”, which adheres to the same rules and game play as Klondike except that the location of all 52 cards is known at all times. Thoughtful Solitaire is a domain
that falls within *classical planning*, but the large search tree required to efficiently search for solutions make the problem space too large for most classical planners.

Prior to our work in Thoughtful, a nested policy rollout algorithm had been used to demonstrate that at least 70% of Thoughtful games had solutions[43]. This work also reported that at least 1.19% of games had no solution, leaving roughly 29% of games uncounted. Our work in Thoughtful [5] significantly improved these results through a combination of algorithmic improvements and domain simplifications. We presented a multi-stage adaption of the nested policy rollout algorithm over a compacted representation of the domain and utilized a dead-end detection algorithm by searching over a relaxed state space. We were able to demonstrate that at least 82% of Thoughtful games have solutions and that at least 8.5% of games have no solutions. These improvements have reduced the percentage of uncounted games to under 10%.

### 1.3.3 Fire Rescue and Emergency Response

Responding to fires and medical emergencies is one of the critical functions of city governments. Many cities face the problem of optimal placement of their limited resources and adopting appropriate response policies to meet the desired quality of service goals. One of the challenges of this domain is that the policies produced by any optimization method must be implementable by people. This usually means that the response policies should be simple to describe and follow prespecified constraints. Many cities use a “running order” to define a response policy for
Figure 1.2: Map of Corvallis, OR. The five fire stations within the city limits are listed. An additional station is located three miles north of station 3 in the town of Lewisburg.

geographic response districts within the city. Each running order assigns a priority ordering over the set of stations that may respond to an emergency. The city is partitioned into response districts which are each assigned a running order.

In a 2009 paper [6], we presented SOFER, a two-phased hill climbing algorithm that combines simulation and optimization to find improved response policies. SOFER integrates two optimization problems in this domain: where to host resources and which resources respond to individual requests. We worked closely with the Corvallis Fire Department (CFD), who provided 5 years worth of 9-1-1 emergency response calls to help us test our algorithm. With this data, we developed a simulator that enabled us to compare our results with an approximation
of the actual response policy used by the CFD. We were able to demonstrate that our learned policies outperformed the approximate policy in use by CFD and compared favorably with optimized policies that violate some of the strict policy requirements maintained by the CFD. Our findings were quite surprising to our collaborators at the CFD, with our conclusions suggesting that overall response performance will be improved by entirely closing one of the six active stations and reapportioning the resources assigned to that station elsewhere.

1.4 Overview

The remainder of this thesis will continue as follows: In the following chapter we will describe the planning frameworks that have traditionally been used to solve the probabilistic planning problem. The remaining chapters will describe our contributions and improvements to these methods. In Chapter 3, we will describe our SOFER algorithm, a direct policy search approach to solving the Fire Rescue and Emergency Response domain. Following this, we will describe our work in the deterministic problem of Thoughtful Solitaire. Our novel Monte-Carlo approaches to solving this domain improved the state-of-the-art for nested rollout search and contributed significantly to our formulations of Monte-Carlo approaches for probabilistic domains, which we will describe in Chapter 5. This work has been conducted in the domain of Klondike Solitaire. We will extend performance guarantees associated with UCT and Sparse Sampling to a new family of Monte-Carlo algorithms that extend UCT, Sparse Sampling and ensemble methods. This
thesis concludes with a discussion of the impact that this work has had on the research community and the future directions that will be taken to continue to make advances in these research areas.
Chapter 2 – Probabilistic Planning Frameworks

We define the probabilistic planning problem as selecting actions that maximize the expected value of future states given a model of the stochastic environment. In this thesis we consider problems defined in terms of a Markov Decision Process combined with a strong simulator.

A Markov Decision Process (MDP) describes an environment in terms of a collection of states, a set of actions, a transition function and a reward, commonly represented as a 4-tuple \((S, A, T, R)\). All objects and features of the environment are captured in the state description \((s \in S)\). In each state, some subset of the actions \((A)\) is available to the agents within the environment. The state that results from those actions is determined by the transition function \(T: S \times A \rightarrow S\) a mapping from states and actions to probability distributions over states, specifying \(P(s'|s, a)\), where \(s'\) is the state resulting from taking action \(a\) in state \(s\). This transition function is restricted by the Markov property – the results of an action can only depend on the current state and the action taken (i.e. it cannot depend on prior states). Transition functions may be stochastic, in which case the current state and action will map to a distribution over some subset of states. For each action in each state, a reward is available to the agent, \(R: S \times A \rightarrow \mathbb{R}\). In goal oriented domains, the reward may be zero except in terminal states. In general, a solution for the MDP is a policy \(\Pi: S \rightarrow A\), mapping states to actions, that
maximizes total reward. In MDPs with an absorbing state, this policy will attempt to maximize expected reward. In recurrent MDPs, this policy will attempt to maximize average reward.

There are a number of variants to the MDP, and certain domains will require (or at the very least, lend themselves favorably to) descriptions in these variations. A partially observable MDP (POMDP) is an MDP in which not all of the domain features are visible to the agents within the domain. In these cases, the agent optimizes behavior by making decisions based on the variables it can see, and possibly some belief state that it tracks internally. A Semi-MDP (SMDP) is an MDP in which the duration of actions taken by agents is not uniform. A Relational MDP is an MDP in which the state description is defined by classes of objects and the relationships between them. Our FRER domain is most easily expressed as an SMDP, in which response times will vary according to the nature of the emergency request and the estimated time each requested unit will remain engaged with the request. Klondike Solitaire is most typically described as a POMDP in which the identities of the face-down cards is fixed, but unobservable to the user. Alternatively, it can be represented as an MDP with stochastic transitions, in which the identity of face-down cards is revealed according to the uniform distribution over the set of cards that have not yet been revealed.

Planning can be conducted easily in any of these MDP variants so long as the agent has a model of the environment. For our algorithms, we typically assume the agent has access to a strong simulator for that model, such that the agent can simulate the result of any combination of states and actions, independent of the
current state of the agent. The Monte-Carlo planning algorithms we discuss in this thesis differ from traditional probabilistic planning methods because they use this simulator to estimate the value of actions.

2.1 Reinforcement Learning and the Bellman Equations

The Bellman equations for a finite Markov Decision Process (MDP) defines the expected reward from following a policy \( \pi \) from a state, \( s \) in terms of the expected rewards of the other states in the domain. Solving an MDP is equivalent to solving the Bellman optimality equations for that MDP, which is defined as

\[
V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V^\pi(s')]
\]

where \( \pi(s, a) \) is the probability of taking action \( a \) in state \( s \) under policy \( \pi \), \( P^a_{ss'} \) defines the probability of transitioning from state \( s \) to \( s' \) under action \( a \) and \( R^a_{ss'} \) represents the reward for that same transition. The value \( \gamma \) defines a discounting factor over future rewards, so that immediate rewards have more value than distant rewards. The Bellman equations can be solved as a series of \(|S|\) equations with \(|S|\) variables, where \(|S|\) may be exponential in the description of the domain. The unique optimal solution to these equations is represented as \( V^* \) and defines a value function over the set of states in the domain with \( V^*(s) \). The Bellman equations
for the optimal policy for an MDP are known as the Bellman optimality equations:

\[ V^*(s) = \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V^*(s')] \]

Because it is intractable to directly solve the Bellman equations for a nontrivial domain, other methods are used to approximate the optimal value function. Many of the fundamental algorithms in Reinforcement Learning update these approximate values with equations based on the Bellman equations. The approximation of the value of a state is updated based on the values of the other states that are close in proximity. The order in which the updates are performed mimic the updated approximations of dynamic programming methods.

In these methods, the values of each state in the MDP are initialized to some (usually random) value. A variation of the Bellman equation updates the approximation of \( V^*(s) \) for \( s \) based on the approximated values of the states that can be reached from \( s \). Each update of \( V(s) \) is called a backup operation. (From the perspective of some initial state \( s_0 \), the values closest to the goal will be the first to approximate \( V^* \), and the improved approximations will slowly “back up” to \( s_0 \) over successive updates.)

2.1.1 Value Iteration and Policy Iteration

Two of the most basic algorithms that utilize this style of update are value iteration and policy iteration which sweep through the entire state space sequentially in order
to approach the optimal values in as few sweeps as possible. The backup operation for value iteration is performed on individual states and is defined as

$$V_{n+1}(s) = \max_a \left[ R_s^a + \sum_{s'} \gamma P_{ss'}^a V_n(s') \right]$$

A single iteration of the value iteration algorithm performs one of these backups for each state in the entire state space. These iterations are performed a sufficient number of times until differences between the old values ($V_n$) and the new values ($V_{n+1}$) is sufficiently small. Convergence of value iteration to $V^*$ is guaranteed independent of the initial values $V_0$.

Like value iteration, *policy iteration* approximates the value of each state by Bellman-like updates. Policy iteration alternates between *policy evaluation* and *policy improvement* to converge to an optimal policy. In policy evaluation, the value of each state is calculated as the expected discounted return of following the current policy

$$V_{n+1}(s) = \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[ R_{ss'}^{\pi} + \gamma V_n(s') \right]$$

The process of *iterative policy evaluation* computes these values for each state, passing through the entire state space, updating the value $v_n$ to $v_{n+1}$. The policy evaluation step repeats these passes through the state space until this difference is sufficiently small.

In policy improvement, a new policy is defined for each state as the action that maximizes the expected return using the value calculated in the policy evaluation
step

\[ \pi(s) = \arg \max_a \sum_{s'} P_{ss'}^a [R_{ss'}^a + \gamma V(s')] \]

Alternating between policy evaluation and policy improvement continues until a policy improvement step does not change the previous policy.

Policy iteration and value iteration are guaranteed to converge independent of the initial state values, but are very slow in large problems due to their dependence on value updates that require sweeps through the entire state space.

2.1.2 Temporal Difference Learning

Like policy and value iteration, Temporal Difference (TD) Learning represents a class of algorithms that solve MDPs by updating the values of states with Bellman-like updates. However, TD methods do not require deliberate ordered sweeps through the entire state space to reach such convergence. Instead, TD methods can utilize Monte Carlo trajectories through the state space to approximate the values of the state. After each action, the approximated value of a state visited at time \( t \), \( V(s_t) \), is updated using the difference between \( V(s_t) \) and the value of the next state in the trajectory \( V(s') \). The updates based on these temporal differences is where TD learning gets its name.

This mechanism is clearly illustrated in TD(0), the simplest of the TD methods, whose update equation is

\[ V(s) \leftarrow V(s) + \alpha \{r + \gamma V(s') - V(s)\} \]
where $\alpha$ is a learning rate and $\gamma$ is a temporal discount factor. The current reward, $r$, and the difference $\gamma V(s') - V(s)$ are the two central values used to update the approximate value of each state in the trajectory. A more complex TD method, TD($\lambda$), utilizes not only the current reward, but all of the future rewards encountered in a MC trajectory to update the approximation of $s$. In Chapter 5, we describe how we use TD($\lambda$) to learn a default exploration policy as a seed for our own learning algorithm.

Online TD methods, such as SARSA [39] update the approximate values of state action pairs, typically represented as $Q(s, a)$, or a $Q$-value. For a given policy $\pi$, $Q^\pi(s, a)$ is updated by taking actions that follow that policy. The SARSA update takes a similar form to the previously defined updates:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left\{ r + \gamma Q(s', a') - Q(s, a) \right\}$$

The SARSA update relies on the quintuple $(s, a, r, s', a')$ of values representing the current state-action pair, the reward and the following state-action pair to update the Q-values. It is from this quintuple that SARSA gets its name. The learned Q-values are an approximation of the value of each state-action pair under the policy used to choose the actions. The Monte Carlo trajectories through the state space must be generated by $\pi$ for the method to learn correct values.

Offline TD methods, of which Q-learning [42] is the original and most familiar, do not require the Monte Carlo trajectories to follow any specific policy. The update directly approximates the true Q-values, $Q^*(s, a)$ independent of the policy
used to explore the domain. The Q-learning update is represented as

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left\{ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right\}$$

So long as the policy used to choose actions is guaranteed to sample each state-action pair infinitely often, no matter how infrequently, the Q-learning method will converge to $Q^*$. As noted by the difference between this update and the SARSA update, the Q-value for the current state-action pair relies not on the action that the current policy will choose when the next state, $s'$ is reached. Instead, the current Q-value is updated based on the best action available. The current Q-value can be credited with finding a good path (as denoted by the max Q-value for the successor state) even if that path is not chosen on the current trajectory.

### 2.2 Planning and Complex Search

The methods for solving MDPs up to this point in the chapter are generally categorized as Reinforcement Learning (RL) algorithms. These algorithms attempt to approximate the true value function solution to the Bellman optimality equations, creating a solution for the entire state space. Unlike these RL approaches, classical planning algorithms attempt to find solution paths for individual states in the domain. Where typical RL algorithms assume that any knowledge about the model of the system must be gained by experience alone, planning algorithms assume access to a model of the system. Where RL systems attempt to approxi-
mate a brute force solution to the Bellman equations, planning systems attempt to approximate the results of a brute force search from a starting point to a stated goal.

In deterministic problems, these differences between the two approaches are relatively straightforward. Planning approaches will attempt to produce a general algorithm that will determine if individual states in the MDP can lead to a goal state. RL approaches will attempt to produce a general solution that will approximate the value of each state in the MDP. While RL approaches are also formed around reaching goals, the RL problem is more general and is applicable to domains without terminal goal states. If a terminal goal exists, the RL solution will find it, but this is more of a happy byproduct of an RL solution. For a planning algorithm, finding a path to a terminal goal is the only reason for existence and without it, the planning algorithm itself is ill-defined.

In probabilistic problems, the motivational differences between the two approaches becomes less clear. Planning algorithms will not be able to perfectly foresee the result from a probabilistic action, or may not be able to detect partially observable features of the domain. As would be expected, early steps in probabilistic planning closely followed the motivations of classical planning, and research flourished in domains where brute force search would still yield an eventual solution. *Contingent planners* such as CNLP [32], C-BURIDAN [17], PGRAPHPLAN [8], C-MAXPLAN and ZANDER [29] were designed to make plans for the possible outcomes of probabilistic actions. These planners assumed the ability to observe the results of probabilistic actions, and contingent planners could make
multiple plans based on the possible contingencies they may encounter. A sub-
class of contingent planners, known as conditional planners [31, 32] made plans for
every possible contingency.

*Conformant planners*, such as BURIDAN [26] and CGRAPHPLAN [37] generate
plans that will work independent of the stochastic state of the world, or results
of stochastic actions. If such plans do not exist, these planners will return the plan
that is most likely to succeed. These planners are useful in situations in which it
is impossible, or relatively expensive to detect the results of actions or the state of
the world.

*Probabilistic planners* assume the planners have some knowledge of the distri-
bution of outcomes over stochastic actions, which includes BURIDAN [26]. The
C-BURIDAN [17] and MANIHUR [31] are both probabilistic and conditional. Un-
fortunately, these algorithms rely on compact state representations and are gen-
erally very slow. In real world planning problems, these representations may be
difficult to generate, if they can be represented at all.

2.3 Monte-Carlo Methods

The algorithms that we present in this thesis borrow from both RL and plan-
ning roots using Monte-Carlo simulation. Monte-Carlo methods refer to a general
approach for approximating the dynamics of a system through repeated random
sampling. After each successive sample, the effects of the sampled policy can
be more closely approximated. Our work has borrowed most heavily from three
Monte-Carlo methods: policy rollout, hindsight optimization and UCT.

2.3.1 Policy Rollout

Rollouts were first described by Tesauro and Galperin [41] as an effective method for amplifying the effects of a search heuristic. Yan et.al [43] adapted rollouts through nested recursive calls to the basic method. Each degree of nesting increases the search time exponentially, but further amplifies the effects of the base heuristic.

Given a heuristic function that maps each state to a real number, a greedy policy with respect to this heuristic chooses the action that results in a state that has the highest value. We refer to this as a level 0 rollout policy. Given a level $n$ rollout policy $\pi$, level $n+1$ rollout policy is constructed as follows: the value of each action $a$ in a state $s$ is estimated by choosing $a$, and then following a level $n$ rollout policy thereafter for a predetermined length (or until termination). After
this step, the action that leads to the highest backed up value is returned. By this definition, a level 1 rollout policy is analogous to the traditional rollout method described in [41]. The standard rollout and greedy policies are described in Figure 2.1(a) and graphically depicted in Figure 2.2(a) and (b). Figure 2.1(b) describes the nested rollout algorithm.

2.3.2 Hindsight Optimization

Hindsight Optimization (HOP)[46] uses an existing classical (deterministic) to approximate solutions for a probabilistic planner. The general idea behind HOP is to estimate the value of each action in a state by way of calls to a deterministic planner on different determinizations of a probabilistic planning problem. In particular, a value of a state is estimated by forming a set of determinized problems...
from the state, solving each one with a deterministic planner, and then averaging the results. Fundamental to HOP is a process called *determinization* in which the probabilistic dynamics of a system are replaced by pre-determined outcomes. We can imagine a hypothetical probabilistic domain governed by the repeated roll of a standard pair of 6-sided dice. Determinization of this domain will fix the outcomes of each of the rolls, converting the probabilistic rolls of the dice to deterministic outcomes, essentially fixing the future outcomes before they occur. The new deterministic planning problem can be solved with an existing classical planner. If a solution for the determinized problem exists, the planner will find it and reveal which action is best for that particular future. Sampling a large enough set of possible futures should reveal which action, on average, will be a good choice from the current state.

### 2.3.3 UCT

UCT (for Upper Confidence bound for Trees) [24] is a method of generating search trees based on Monte-Carlo trajectories in stochastic domains. UCT intelligently samples trajectories based on an upper confidence bound calculated by an implementation of Hoeffding’s inequality. UCT extends recent algorithms for multi-armed bandit problems to sequential decision problems including general Markov Decision Processes and games. Most notably UCT has received recent stature as the premiere computer algorithm for the game of Go [19], resulting in huge advances in the field.
Given a current state, UCT selects an action by building a sparse look-ahead tree over the state-space with the current state as the root, edges corresponding to actions and their outcomes, and leaf nodes corresponding to terminal states. Each node in the resulting tree stores value estimates for each of the available actions, which are used to select the next action to be executed. UCT is distinct in the way that it constructs the tree and estimates action values. Unlike standard mini-max search and sparse sampling [23], which typically build depth bounded trees and apply evaluation functions at the leaves, UCT does not impose a depth bound and does not require an evaluation function. Rather, UCT incrementally constructs a tree and updates action values by carrying out a sequence of Monte-Carlo rollouts of entire decision making sequences starting from the root to a terminal state. The key idea behind UCT is to intelligently bias the rollout trajectories toward ones that appear more promising based on previous trajectories, while maintaining sufficient exploration. In this way, the most promising parts of the tree are grown first, while still guaranteeing that an optimal decision will be made given enough rollouts.

The new algorithms we describe in Chapter 5 are planning algorithms based on HOP and UCT that re-plan at every step. Prior to each action, the values of the immediate actions are approximated using a model of the system. These methods have roots in both RL and planning components. Like RL methods, a value function is approximated over a restricted subset of the state space. Like planning methods, access to a complete model of the system is assumed.
Chapter 3 – Direct Policy Search

Planning and Reinforcement Learning algorithms use approximations in place of exact solutions in order to reduce complexity and tackle more difficult problems. Despite these efforts, some problems are still too difficult for the most advanced RL and planning techniques. With the help of a model, it is often possible to construct a simulation of the domain, and improve performance through direct policy search.

3.1 Fire, Rescue and Emergency Response

Responding to fires and medical emergencies is one of the critical functions of city governments. Many cities face the problem of optimal placement of their limited resources and adopting appropriate response policies to meet the desired quality of service goals. The problem is compounded by the lack of knowledge of concrete future demands, the need to balance success across multiple criteria, and the sheer complexity of managing several pieces of equipment, people, and tasks at the same time. One of the major challenges of this domain is that the policies produced by any optimization method must be implementable by people. This usually means that the response policies should be simple to describe and follow prespecified constraints. The constraints are problematic for some RL methods which would
choose greedy actions based on approximations of optimal value functions. Traditional planning in such a domain is not tractable because of the inability to plan for all possible combinations of unseen future emergency requests. In this domain, the ability to implement a policy and directly approximate its performance will help to gradually improve the existing policy one step at a time.

Simulation is a critical tool often used to evaluate specific policies or policy changes in the domain of emergency response [33, 38]. What is missing in these studies and others like them is the ability to automatically improve the policies through search and optimization. Such methods have been shown to be effective in other domains optimizing across multiple metrics, such as school redistricting [15]. Isolated subproblems of our domain such as vehicle placement have been extensively studied in Operations Research through mathematical programming and other optimization methods [11, 20, 18, 33]. However, these studies are not based on simulations of real-world data nor do they study the problem comprehensively to include both resource placement and emergency response.

In a recent paper[6], we describe a highly effective and usable tool called SOFER for Simulation-based Optimization of Fire and Emergency Response that combines simulation and optimization to find improved response policies. SOFER combines the two problems of where to host resources, e.g., fire trucks, and how to respond to emergencies, e.g., which truck to send, into an integrated optimization problem. Using this tool, it is possible to optimize both resource placement and emergency response to changes in new developments, population densities, and resource budgets on the one hand and the importance of different quality metrics on the other.
Our system can simulate a given policy on real data and gather statistics on its performance. The optimizer is built on top of the simulator and uses a two-level random-restart hill climbing search. At the higher level of resource placement, the system decides the home stations for each resource, e.g., engines, ladder trucks, ambulances. At the lower level of emergency response, the system allocates particular resources to emergency calls based on the location of the emergency and the availability of the resources.

We evaluated SOFER on real emergency call data collected over 5 years in Corvallis, OR, a small university town of about 53,000 people. Our system finds placement and response policies that outperform best practice by optimizing over a weighted sum of different metrics of quality of service. The system produces response policies which are sensitive to the importance of different metrics such as time for first response, time for the arrival of full service complement, etc. Secondly, we can easily evaluate the impact of new developments or population changes by adapting the call distribution to the changed situation and then optimizing the policies. In our particular case study, the results suggested that it is possible to close one of the fire stations and still achieve the desired level of performance with the usual volume of calls.

3.2 CFD System and Domain

The Corvallis Fire Department utilizes a “response district and running order” policy for response to fire and emergency medical requests. The entire service area
is partitioned into separate response districts each of which maintains a preference over stations. This preference list is the fixed “running order” for that response district. Individual stations are led by lieutenants, who are responsible for memorizing their first-response district - those response districts where their station appears first in the running order. A lieutenant listening to a dispatch radio can alert his station to a coming call before the call is actually made. Response district boundaries are commonly drawn along major streets to simplify this memorization. When a request is received by the 9-1-1 dispatchers, it is identified by its geographic service district. The dispatchers assign responding units based on the running order and the currently available units. In cases when unit requests exceed available units, a request for assistance to neighboring departments is issued. In turn, CFD provides assistance to neighboring communities when necessary.

Individual firefighters and paramedics are loosely assigned to vehicles but tightly bound to their home stations. In many cases, a single complement of individuals may be co-located with multiple vehicles, and respond with whichever vehicle is currently requested, leaving the other vehicles in station and unstaffed. This flexible “either/or” response policy allows a smaller number of individuals to respond to a greater variety of requests in a shorter time period. In some stations, there are enough personnel to staff multiple vehicles simultaneously, and station lieutenants are at liberty to match appropriate personnel and vehicles as necessary.

Various metrics are used to measure the quality of responses to emergency requests. The National Fire Protection Association (NFPA) establishes performance standards [30] for local Fire and Rescue Departments, including standards
for response times to fire and emergency incidents. The NFPA standards are nonbinding and local departments adapt the standards for their particular communities. The Corvallis Fire Department (CFD) goals for requests within the Corvallis city limits are that 45% of first arrivals be under 5 minutes, and that 90% of first arrivals be under 8 minutes. Performance relative to these goals, as well as goals for other geographic areas within the service area, is reported to the City of Corvallis on a quarterly basis [12]. In this paper, we will focus on the performance for calls within city limits, where the great majority of calls originate.

In addition to the arrival of first response units, NFPA also establishes standards for the arrival of the full assignment for fire suppression incidents and the arrival of advanced life support equipment to emergency medical incidents. Rather than focus on average response times, NFPA recommends standards based on the 90th percentile of responses. In addition to these metrics, we also report the percentage of requests that result in system overwhelms, when CFD cannot provide the requested service due to previous assignments.

Our data consists of a call log of 24,739 9-1-1 call requests received by CFD from Oct 20, 2003 to Sept 2, 2008. Each data entry contains information describing the time, location and specific nature of a single request. For privacy reasons, the x/y locations provided for each request have been rounded to the nearest 200 feet.
3.3 Simulation

Our simulator is a simplification of actual CFD policy and performance. It is driven by the provided call log and a dispatch policy. Some parameters within the simulation are stochastic in nature, such as response time and assignment duration. To simplify the simulation process we have replaced these stochastic processes with fixed parameters, based on averages collected from the original data set.

3.3.1 Response and Travel Time

Entire response time is calculated as a combination of elements from each request. We approximate 1 minute for the dispatcher to request a service unit and an additional 1 minute for the service unit to turnout to the call. We approximate the travel speed of response units as a function of distance traveled. For distances under 5 miles, we approximate a travel speed of 31 mph; distance < 15 miles: 43 mph; distance < 25 miles: 50 mph; distance > 25 miles: 52 mph. We utilize a “taxi-cab” distance, summing the distances traveled in the x and y coordinates.

3.3.2 Assignment Duration

The duration of each assignment is determined by the nature and cause of the request. We grouped individual calls into distinct response templates. Because various emergencies require the same complement of resources for roughly the same duration, these can be grouped together as equivalent in the simulation. Each
template is defined by a unique complement of required personnel and vehicles and/or assignment duration for each unit type. Each assigned unit is considered to be “out of service” for the entire request period, which includes the 2 minutes of dispatch and turnout time, travel time to incident, completion of assignment at incident and return travel to home station. In practice, units may be called into service before returning to their home station, or even re-assigned to a different request while in service, although the latter is rare. In our simulations, each unit is considered to be out of service until it has completed the current assignment and returned to its home station.

3.3.3 Personnel, Vehicles and Units

Of the various types of vehicles employed by CFD, we simulate the responses of the four most commonly requested vehicle types: tanker engines, ladder trucks, ambulances and the battalion chief. In order to limit combinatorial search, our simulation abstracts individual personnel and vehicles into combined “units”. Each unit consists of personnel to operate a single vehicle and may contain one vehicle or two, not to be utilized simultaneously. When multiple units occupy a single station, the personnel of one unit may not operate the vehicles of another unit, even if it is idle and requested. Additionally, our simulation allows unlimited capacity for vehicles and personnel in each station, although the optimization process rarely exceeds the actual capacity of each station. The actual number of CFD staff in service varies by time of day and day of week. Our simulation maintains a
single battalion chief (BC) and sufficient personnel to staff 7 service units at all times. Our simulation maintains a vehicle for the BC and 11 additional vehicles: 3 ambulances, 6 engines and 2 ladder trucks. A map of the city of Corvallis is shown in Figure 1.2. Five of the six stations can be seen. Station 6 is to the northeast of Corvallis in the town of Lewisburg, part of the CFD service district. Our representation of CFD default station assignment is shown in Table 3.1a.

Among the data that we received from CFD was a file of 1068 requests that were annotated with both x/y location data and full running orders to each location. From this data we were able to construct an approximation of actual CFD policy within our simulator. For the 24,739 calls in the main data set, we approximate the CFD running order to each call with the running order to the closest of the 1068 calls in the annotated file. CFD reports their performance quarterly to the City of Corvallis [12] based on three metrics: average response time for first responder (FR avg), percentage of requests in which the first responder arrived in less than 5 minutes (FR<5), and the same metric for 8 minutes (FR<8). We compare our simulated performance with the actual data reported by CFD in Table 3.1b. Differences between our approximated policy and the most recent CFD data illustrate the difficulty in accurately modeling complex systems.

3.3.4 Response District Grid

Without explicit knowledge of CFD response districts, we have drawn response districts by overlaying the service area with a two-scale grid. The large grid is
Table 3.1: a: Default station assignments for the 8 units in our simulation. b: The simulated performance of this configuration using an approximated CFD running order compared to figures reported by CFD.

delineated by the boundary box defined by the entire service area and contains 6 square cells, measuring 100,000 feet to a side (roughly 360 square miles each). The smaller grid is delineated by the boundary box drawn around the 6 active stations, and contains 55 cells. Sides of small cells measure 5000 feet (0.9 square miles). Each of these 61 cells is assigned a running order policy for requests originating from within the boundaries of the cell.

3.4 Optimization Methods

Policies in the fire and emergency response domain can be altered in two distinct ways—by changing the basic unit composition/station assignments and by changing the running orders of the response districts. It is important that our optimized configurations and response policies are similar in structure to those actually implemented by CFD. Responding units must remain in their stations until requested,
and must be assigned based on the running order priorities of the requesting response district. Finding improved response policies within such restrictions can be achieved in one of two ways—either learning directly in the restricted space or approximating a restricted policy from an optimized unrestricted policy.

3.4.1 Reward

Collectively optimizing the various performance criteria will require compromise. To help us balance performance over all metrics, we have designed a weighted reward for individual responses based on four parameters. We formulate the optimization problem as maximizing the average expected reward per call. Each of the parameters is closely associated with a goal that will improve the quality of service within the response area. These are the four parameters:

**FR** measures the elapsed time between the placement of the 9-1-1 call and the arrival of the *first responder* on scene.

**FC** measures the elapsed time between the 9-1-1 call and the arrival of the last of the requested units, the time when the *full complement* of responding units is on scene.

**TB** is a fixed *time bonus* received when the first responder arrives on scene in less time than the stated time goal for that response region. For requests originating within the Corvallis city limits, this time goal is 5 minutes.

**OP** is a fixed *overwhelm penalty* received when there are not enough resources
available to meet current requests. In these cases, additional units are requested from neighboring municipalities.

The default reward is represented as

\[-(w_1 \cdot FR) - (w_2 \cdot FC) + (FR < 5 ? TB) + (\text{overwhelm} ? OP)\]

with \(w_1 = w_2 = 1\), \(TB = 10\) and \(OP = -100\) where units of all variables are measured in minutes.

### 3.4.2 Learning Unrestricted Policies

We considered a number of techniques to optimize the policies in this space. Some optimization techniques that rely on continuous variables, such as linear programming are not suitable to our discrete policy space. Traditional dynamic-programming-style methods are problematic because of our large state space, but using our reward function and a parameterized value function over state-action pairs, we were able to learn a successful policy using an approximate dynamic programming method called Least Squares Policy Iteration (LSPI) [27]. We use LSPI to learn a value function over available actions, given the current state, which takes the place of the “running order” method of choosing actions.

Our value function is a linear combination of 10 weighted features:

1..8 One feature for each of the 8 units indicating the duration (in minutes) until the unit returns to its home station.
The reward received from the current action, which calculation is described above, based on response times, a time bonus and system overwhelms.

A boolean feature indicating if the system is currently overwhelmed.

This function approximates the Q-value of taking an action, \( a \), from a state, \( s \), given the weights \( w \): \( Q(s, a, w) = \phi(s, a)^T w \), where \( \phi(s, a) \) is the matrix of basis features for the state-action pair \( (s, a) \). LSPI fits the weights, \( w \), to minimize the squared difference between the estimated Q-values and the discounted rewards observed in simulations of the call log. The weights learned by LSPI estimate Q-values and choose actions using an \( \epsilon \)-greedy algorithm with \( \epsilon \)-exploration decreasing over subsequent iterations. LSPI will converge to a set of weights that we can use to approximate the value of available actions.

The set of 10 features used in this function are rich enough to capture complex relationships, such as behavior that would prefer to not assign units if nearby stations are already vacant. It can also learn to balance these relationships with the available reward. These complex relationships are not captured in the fundamental language used to assign response policies by CFD. Our unrestricted action space allowed LSPI to assign any available unit to a request independent of station assignment or running order. The value function learned by LSPI is able to predict the value of actions in real time. Such a policy cannot be directly applied to our domain because response policies must be static and defined over a fixed set of running orders. We approximate a running order based on our LSPI policy by recording the frequency of station requests for each cell in the response district.
Table 3.2: results from LSPI and an approximated running order based on the simulated LSPI policy.

<table>
<thead>
<tr>
<th></th>
<th>FR% &lt;5</th>
<th>FR% &lt;8</th>
<th>FR avg</th>
<th>% over</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSPI</td>
<td>60.64</td>
<td>93.66</td>
<td>4.59</td>
<td>6.14</td>
</tr>
<tr>
<td>LSPI (apprx)</td>
<td>56.56</td>
<td>92.88</td>
<td>4.83</td>
<td>6.44</td>
</tr>
</tbody>
</table>

grid. The running order for each grid cell is fixed to the order of most frequently requested stations. In Table 3.2 we can see that the LSPI policy outperforms the approximated policy on all four metrics. In addition to the three metrics reported by CFD, we also measure the percentage of calls that result in a system overwhelm (%over). Given the absence of policy restrictions, it is not surprising that LSPI performs well, but it is not clear how to construct a good restricted policy from the unrestricted solution.

3.4.3 Learning Restricted Policies

Rather than rely on an intermediate step that constructs one policy from another, we have found a simple solution that directly improves an existing policy within the restricted space. Our two-phase hill climbing algorithm is shown in Algorithm 1. The running orders for the grid cells are fixed in place while the unit station assignments are optimized, which are then fixed while the running orders are optimized. These two phases continue until the policy converges.
1 while policy not converged do
2   optimize unit station assignments
3   optimize running orders

Algorithm 1: Two-phase hill climbing optimization.

3.4.3.1 Optimization of Running Orders

The hill-climbing technique for running orders optimizes each cell in the response grid individually while the running orders for the other cells are held fixed. Each of the 61 grid cells is optimized, in turn, by simulating the call log with each of the 720 (6!) possible running orders. This algorithm (described in Algorithm 2) is factorial in the number of stations. While this is practical for the 6-station CFD, additional approximation methods will be required for larger departments with a dozen or more stations.

1 for each cell, c do
2   for each running order, r do
3     t = simulate calls(using r for calls in cell c)
4     if t > best-score then
5       best-score = t
6       best-order = r
7     end if
8   end for
9   c.running-order = best-order

Algorithm 2: Running Order Optimization Method

3.4.3.2 Optimization of Unit Assignments

Without limits placed on station capacity, testing every possible unit assignment configuration would not only be combinatorially expensive, but would also test
many unreasonable configurations. To optimize the unit assignments, we employ a hill climbing method, described in Algorithm 3, that tests each possible single-vehicle-reassignment. These reassignments can be of two forms: either moving a vehicle from a multiple-vehicle-unit to a single-vehicle-unit or moving an entire single-vehicle-unit to a different station. Units are not allowed more than two vehicles are not allowed to contain multiple vehicles of the same type. On each pass through the loop, the reassignment that improves performance the most is made permanent.

```java
1 while improving do
2   improving = false
3   for each vehicle, v and each station, s do
4     if move (v to s) then
5       t = simulate calls(using fixed running order)
6       if t > best-score then
7         best-score = t
8         improving = true
9         best-v = v; best-s = s
10       move(best-v to best-s)
```

**Algorithm 3**: Unit Assignment Optimization

3.4.3.3 Random Initialization

This two-phase optimization process performs well if it is seeded with a reasonable unit configuration and accompanying running order. For our domain in general, this may be a reasonable assumption, as fire chiefs and city planners will have access to current best practices (or reasonable approximations) for their particular
department. However, by starting with fixed configurations and running orders, we may be predisposing our method against interesting and novel solutions. To avoid this, we prefer to start our algorithm with random unit configurations and random running orders. Unfortunately, the described algorithm overwhelmingly leads towards poor local minima when initialized randomly. Optimizing running orders for random unit configurations reinforces the mediocre station assignments. In turn, the unit assignment optimization favors station re-assignments that perform best for the bad running orders.

To break this self-reinforcing loop, we introduce a small amount of search in the unit configuration optimization step. **Algorithm 3** is expanded to include a partial optimization of running orders in the simulation on line 5. For each vehicle reassignment, an optimization over all grid cells similar to **Algorithm 2** is performed, but only for the first two elements of each running order. This reduces the number of simulations per cell from 720 to 30. While not exact, it approximates the optimization that will occur during the running order optimization step, should that particular vehicle reassignment be made permanent. Thus the evaluation of each vehicle reassignment is a more accurate reflection of its true value. Using this improved heuristic to evaluate station reassignments enables the algorithm to break out of self-reinforcing local minima.
3.5 Evaluation of Results

All results from SOFER are from tests on unseen data. We have trained our policies using the calls from the first half of the data set and report performance by simulating the learned policies on the second half of the data.

3.5.1 Policy Comparison

We compare our results against four baseline standards—the approximated performance of the LSPI algorithm, the simulated performance of the approximated CFD running order, a closest-station-first policy and our own SOFER algorithm initialized with the default CFD unit station assignments and approximated running order, without random restarts. Each of these reference policies were simulated using the second half of the call data. The SOFER policy was trained on the first half of the data. In addition to the standard reward, we report results for 4 additional adjusted rewards, in which weights for two of the parameters in the reward function have been changed to magnify performance for specific metrics. For the FR and OP parameters, we report performance for when the regular values of these parameters are multiplied by 10 as well as when they are eliminated completely. For each of the reward structures, SOFER was restarted and allowed to converge 5 times. The policy that performed best on the training data is retained and used to simulate the test data on which results are reported. These results are displayed in Figure 3.1. For the top two graphs, higher values indicate improved performance. The converse is true for the lower two graphs.
It is interesting to note that that “closest station first” (CSF) policy does not fit within the restricted policy space of our running order policies. Such a policy would require drawing response districts along unnatural boundaries. It is impressive that at least one of our SOFER policies (and usually more) are better than the CSF policy for each of the four metrics.

The results illustrate the necessity to balance performance across all metrics. When emphasis is placed on the arrival time of the first responder (FR×10), relative performance improves in the first three metrics, associated with arrival times of first responders, while performance decreases in the fourth graph, measuring the percentage of system overwhelms. A similar effect is seen when the overwhelm penalty is set to zero (OP=0). The opposite effect is seen when the value of the overwhelm penalty is increased (OP×10) or the value of the average first responder is eliminated (FR=0). One interesting artifact of our tests is the consistent tendency of SOFER to converge to policies that leave Station 5 unmanned, which occurred in 4 of the 5 random-restart tests, two of which we examine in Table 3.5.1. More surprising is that Station 5 is left unmanned in the SOFER policy initialized with default CFD responses, which we would expect to have less of a tendency to stray from the initial CFD configuration.

We have presented results in the Fire, Rescue and Emergency Response domain, based on a 5 year log of actual emergency requests. We presented SOFER, a two-level random restart hill climbing algorithm that optimizes unit station assignments and in-time response policies. Our simulated results using the SOFER algorithm improve on simulated best-known policy, and are sensitive to preference
Table 3.3: Two example configurations generated by SOFER. Configurations X and Y were generated with rewards in which the FR component of the reward had ten times its normal value, and where it was eliminated completely, respectively. Both leave Station 5 unstaffed. Configuration Y also leaves station 4 unstaffed. These are compared to our simulated CFD policy and a “closest station first” policy, based on four metrics.

<table>
<thead>
<tr>
<th></th>
<th>X: (FR×10)</th>
<th>Y: (FR=0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>St 1 B.Chief</td>
<td>St 1 B. Chief</td>
</tr>
<tr>
<td>U2</td>
<td>Eng &amp;Amb</td>
<td>Eng &amp;Amb</td>
</tr>
<tr>
<td>U3</td>
<td>Eng</td>
<td>Eng</td>
</tr>
<tr>
<td>U4</td>
<td>St 2 Eng &amp; Amb</td>
<td>St 2 Ldr &amp; Amb</td>
</tr>
<tr>
<td>U5</td>
<td>St 3 Ldr &amp; Amb</td>
<td>Eng</td>
</tr>
<tr>
<td>U6</td>
<td>Eng</td>
<td>St 3 Eng &amp; Amb</td>
</tr>
<tr>
<td>U7</td>
<td>St 4 Eng &amp; Ldr</td>
<td>Ldr &amp; Amb</td>
</tr>
<tr>
<td>U8</td>
<td>St 6 Eng</td>
<td>St 6 Eng</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>FR%</th>
<th>FR%</th>
<th>FR %</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;5</td>
<td>&lt;8</td>
<td>avg</td>
</tr>
<tr>
<td>X:(FR×10)</td>
<td>59.81</td>
<td>94.41</td>
<td>4.70</td>
</tr>
<tr>
<td>Y:(FR=0)</td>
<td>55.65</td>
<td>93.70</td>
<td>4.89</td>
</tr>
<tr>
<td>CFD(simulated)</td>
<td>56.95</td>
<td>93.46</td>
<td>4.84</td>
</tr>
<tr>
<td>CSF</td>
<td>59.12</td>
<td>94.78</td>
<td>4.73</td>
</tr>
</tbody>
</table>
over measured metrics and population density changes. One surprising conclusion of our results strongly suggest that performance across multiple metrics can be maintained even if one of the six stations is closed. This conclusion is counterintuitive, providing valuable insight into policy structures, especially for city governments that may be investigating policies that maintain performance utilizing reduced resources.

The policy restrictions inherent to our system prevent the direct application of some standard optimization techniques. We have shown that even approximations of very successful unrestricted policies perform poorly compared to our algorithm that optimizes within the restricted space.
Figure 3.1: Comparison of response policies over four metrics to various baselines including a running order approximation of an LSPI policy, an approximation of current CFD policy, a “closest station first” (CSF) policy, and a version of our SOFER policy initialized by our CFD policy. Our SOFER policy was driven by a reward function defined over various response features. We demonstrate the adaptability of SOFER by reporting results when the weights of two of these features (OP = penalty for overwhelming the system, FR = elapsed time when first responder arrives) are eliminated (=0) and when they are multiplied by 10.
Chapter 4 – Deterministic Planning with Rollouts

With access to a model in deterministic domains, a basic search mechanism has the ability to determine if a goal state can be reached from any starting position. In nontrivial domains, these basic search algorithms will not complete their search in a reasonable amount of time. Classical planning algorithms attempt to find solutions to these problems, largely by optimizing the search and short-cutting the process of searching through the entire state space. In very large domains, even the most sophisticated planning methods cannot expand their search sufficiently to solve most individual problems. Monte-Carlo methods that are suitable to probabilistic problems can also be useful in these problems. In a 2007 journal article [5], we presented an algorithm based on a Monte-Carlo policy rollout method to find solutions for individual games of Thoughtful Solitaire - a deterministic and observable variation of the common Klondike Solitaire game. Our algorithm utilized a heuristic search method to guide the trajectories through the search space. The deterministic planner described in this chapter will form the basis for a HOP planner that we will describe in Chapter 5.

Many AI problems from computer game playing to industrial process planning can be formulated as search problems, where the goal is to find a sequence of actions from an initial state to a goal state. In real-time search, one needs to commit to an action in a reasonable amount of time, i.e., before searching a significant fraction
of the state space. Heuristic search methods are among the most successful in solving planning and search problems [9] and are adapted to real-time search [25]. An effective heuristic can help focus the search along fruitful directions. However, simply choosing an action greedily according to one-step look-ahead is usually not effective because it leads to dead ends and local optima. Tuning of the heuristic via reinforcement learning is one way to circumvent this problem [1, 25], but for large domains, this requires approximating the heuristic function [40]. The error in the approximated function ultimately limits the effectiveness of greedy action choice.

Multi-step depth-first look-ahead is a standard search method effectively used in games. It has been shown that rollouts and nested rollouts also amplify the effectiveness of a search heuristic in some domains [41, 43]. We refer to action selection using greedy one-step look-ahead as a level 0 rollout. A level $n$ rollout considers all possible actions in the current state followed by level $n - 1$ rollouts until the end of the game (a win or a deadend). It then chooses the action that led to the best possible result.

Due to the inherent complexity of search spaces, multiple heuristics are often useful in different regions of the state space [2, 13, 44]. In our paper we introduced multistage nested rollouts, where different heuristics are used in different stages of the search space, and are combined with nested rollouts of different degrees of nesting. Multistage nested rollout enables tight control of search magnitude at each stage in the search tree. Search can be increased in difficult stages, and conserved in simple stages. We demonstrated the effectiveness of this algorithm
by creating a real-time solver for the game of Thoughtful Solitaire, a version of Klondike Solitaire where the location of all cards is known.

Prior to our work, the best published algorithm [43] solved up to 70% of the games using level-3 rollouts in an average of 105 minutes per game. While we cannot determine every game, we demonstrate that no less than 82% and no more than 91.44% of instances of Thoughtful have solutions. We also demonstrate that over 80% of games can be determined in less than 4 seconds, providing the foundation for a Thoughtful Solitaire game with real-time search capability, an example of which we have provided for free download.\(^1\)

What makes our search especially effective in Solitaire is the use of a compressed search tree, which combines sequences of individual moves into longer macros. A new representation for Thoughtful and Klondike solitaire based on this compressed search tree is described in Section 4.1.2. Section 4.2 describes our multistage nested rollouts algorithm. In Section 4.3, we describe additional domain-specific improvements to the search mechanism, including pruning methods and additional heuristics. Section 4.4 presents our results.

4.1 Klondike Solitaire and Variants

The following definitions apply to the game of Klondike Solitaire and its variants, including Thoughtful Solitaire. A screenshot of a game of Klondike Solitaire can be seen in Figure 4.1.

\(^1\)web.engr.orst.edu/~ronny/k.html
Suit: ♦ (diamonds) and ♣ (hearts) are red. ♠ (clubs) and ♠ (spades) are black.

Rank: Each suit is composed of 13 cards, ranked (in ascending order): Ace, 2, 3, 4, ..., 10, Jack, Queen, King. For our value functions, we also refer to a 0-based rank value: (A=0, 2=1, ..., J=10, Q=11, K=12).

Tableau or build or base stacks: seven stacks to build down card sequences in descending order and alternating color. A card $x$ can block another card $y$ if $x$ is above $y$ in a Tableau stack and $x$ is not resting on a face up card. Each card (except for Kings) has two tableau build cards, which are the two cards of opposite color and next higher rank that a card can be placed on to build down the Tableau stacks.

Foundation or suit stacks: one for each suit (♦,♣,♥,♠). The goal of the game is to build all 52 cards into the Foundation stacks in ascending order. Each card (except for Aces) has a foundation build card which is the suited card of preceding rank that a card can be placed on to build up the Foundation.

Stock or deck: holds face-down cards, which can be transferred to the Talon three at a time.

Talon or discard or waste: holds face-up cards transferred from the Stock. The top-most card on the Talon can be played to the Foundation or Tableau. When emptied, the Talon can be recycled by placing it as a unit face down onto the Stock.
Figure 4.1: A screenshot of Klondike Solitaire. Each card stack is labeled with one of four types: the Talon, the Stock, the 4 Foundation Stacks and the 7 Tableau Stacks.

Cards can be moved from the Talon to the Tableau and Foundation stacks according the rules defined above. Cards can also be moved back and forth between the Tableau and Foundation stacks. Some rules vary with versions of Klondike. All methods and results in this paper apply to those versions of Klondike with three card Stock turns with unlimited deals, allowing movement of partial tableau stacks and Foundation cards to the Tableau.
4.1.1 Thoughtful Solitaire

The rules and structure of Thoughtful Solitaire are identical to those of Klondike solitaire, with the exception that the identity and location of all 52 cards is known at all times. Because of this, the strategy of Thoughtful Solitaire differs from typical Klondike play, with much more emphasis placed on deterministic planning.

4.1.2 A New State Representation: $K^+$ Solitaire

In any given Klondike state, as few as one-third, or as many as two-thirds of the cards in the Talon and Stock are reachable through repeated play of the Turn Stock action, but only the card at the top of the Talon is playable. We introduce a new state representation that makes all reachable Stock and Talon cards immediately playable by combining states linked by Turn Stock actions. We call this new representation $K^+$ Solitaire. The action space of the compressed $K^+$ node is the union of the actions available from the uncompressed nodes, minus the eliminated Turn Stock actions. Figure 4.2 illustrates this tree compression. This new representation of the state-space can also be thought of as adding several macro-actions, where each macro consists of a series of Turn Stock actions followed by a Play action.

This new representation is graphically displayed in a Solitaire game we have made available at [3]. The face-up Talon is fanned out on the table, with all reachable cards highlighted for immediate play. A screenshot of this display can be seen in Figure 4.3. Essential game play using the $K^+$ representation is not altered. All solutions found in the compressed search tree are applicable to Klondike and
Figure 4.2: An illustration of the search tree compression resulting from the elimination of the Stock. The search tree for Klondike is compressed by eliminating each of the *Turn Stock* actions.

Thoughtful Solitaire.

### 4.2 Traversing the Search Tree

A deep search tree with a non-trivial branching factor prohibits brute force search methods from being effective in Klondike and its variants. Heuristic search methods fare well in such domains if a reasonable heuristic can be applied. Even with a good heuristic, search remains expensive. Yan, *et al.* [43] used a complex heuristic based on published best practices in playing Klondike. As seen in Figure 4.9(b), this heuristic alone only won 13% of games, with improved performance resulting
from nested rollouts. Our search method is motivated by two factors. First, it
appears that different stages of the game warrant different heuristics for guiding
the search. Second, all stages may not need deeply nested searches. By tuning the
level of rollout search for each stage, resources can be allocated to difficult stages
and conserved in simple stages, with an appropriate heuristic applied to each stage.
procedure multistage-nested-rollout \((s, h_0, h_1, \ldots, h_z, n_0, n_1, \ldots, n_z)\)

1. while \(s\) is not a dead-end or a win
2. if \(n_0\) is -1
3. return \(h_0(s,a)\)
4. else
5. \(val = \max_a \text{ multistage-nested-rollout}(\text{result}(s,a), h_0, \ldots, h_z, n_0-1, \ldots, n_z)\)
6. and let \(a'\) be the maximizing action
7. if \(h_0(s) > val\) and \(h_0\) is not the last heuristic
8. \(a' = \arg\max_a \text{ multistage-nested-rollout}(s, h_1, \ldots, h_z, n_1, \ldots, n_z)\)
9. \(s = \text{result}(s, a')\)
10. return \(h_0(s)\)

Figure 4.4: A description of multistage nested rollouts. A heuristic, \(h\), evaluates a state, \(s\), with \(h(s)\). Each \(n\) is a level of nesting, associated with a heuristic. At each level of nesting, the algorithm attempts to determine which action, \(a\), should be taken.

4.2.1 Staged Rollouts

The rollout procedures described in section 2.3.1 struggle if different stages of the game warrant different heuristics[13, 2]. Increasing the nesting would typically overcome the weakness of the heuristic, but only at the expense of increased search. An ideal solution would allow independent tuning of both the heuristic and the magnitude of search in all stages of the search tree.

We present a multistage search technique utilizing sequences of nested rollouts. The sequence of heuristics is applied to the search tree in order. When the first heuristic reaches a local maximum, the second heuristic is applied, and so on until the last heuristic, which proceeds greedily until termination. A level of nesting, \(n\), is associated with each heuristic, so that the coverage and search magnitude is tuned for each stage of the search tree. The multistage nested rollout algorithm is described in Figure 4.4.

Search proceeds in a manner quite similar to the nested rollouts algorithm
described in Section 2.3.1. Children nodes are searched and evaluated with a recursive call to the search algorithm. Unlike the unstaged rollout algorithms, a new heuristic is applied when a local maximum is reached. In its simplest form (\(n=0\) for all heuristics), a greedy path is chosen from the children nodes until reaching a local max, at which point the greedy search continues with the next heuristic. A graphical representation of this simple search mechanism with two heuristics can be seen in Figure 4.5(a).

Figure 4.5(b) displays a more complicated instantiation with two heuristics, each with a nesting level of 1. Search proceeds in a manner similar to that of the unstaged level 1 search (represented in Figure 2.2(b)), with each child of the root searched with a level 0 search. When a local maximum is reached at a node \(s\), a new level 1 search begins with the next heuristic. The children of \(s\) will be searched with a level 0 search using this new heuristic. The choice made at the root will be based on the local evaluations of all leafs reached by the final heuristic.

### 4.3 The Thoughtful Solitaire Solver

The multistage nested rollout algorithm searching through the compressed \(K^+\) search tree described in Section 4.1.2 provides the basis for our real-time solver for Thoughtful Solitaire. The compressed tree reduces the depth of the search tree while increasing the branching factor. This increased branching factor allows the heuristics to evaluate a larger number of children at minimal cost. We have developed two heuristics for this problem. The first (H1) is designed for opening
Figure 4.5: Representation of search paths taken for (a): multistage nested rollouts with two heuristics, each with level 0 search, and (b): multistage nested rollouts with two heuristics, each with level 1 search.

the game and the second (H2) for end-game scenarios. The two heuristics are represented as weighted value functions in Figure 4.6.

Changes made to this basic search algorithm make it especially effective in searching through our solitaire domain. These changes include a simple pruning method, node caching, global and local loop prevention and additional heuristics for action ordering.

4.3.1 Pruning

We prune branches of the search tree by searching a relaxed version of K Solitaire, which can be done in time linear with the number of cards. Relaxed games without
<table>
<thead>
<tr>
<th>Num</th>
<th>Description of Feature for Card ( x ) (# of elements)</th>
<th>( H_1 )</th>
<th>( H_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( x ) is in a Foundation stack (52)</td>
<td>5 - rank value</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>( x ) is face down in a Tableau stack (52)</td>
<td>rank value - 13</td>
<td>rank value - 13</td>
</tr>
<tr>
<td>3</td>
<td>( x ) is available to be moved from the K+ Talon (52)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>( x ) and the other card of same rank and color are both face down in some Tableau Stack (26)</td>
<td>-5</td>
<td>-1</td>
</tr>
<tr>
<td>5</td>
<td>( x ) is blocking a suited card of lesser rank (( 4 \times (12+11+\ldots+1) ))</td>
<td>-5</td>
<td>-1</td>
</tr>
<tr>
<td>6</td>
<td>( x ) is blocking one of its two Tableau build cards (48\times2)</td>
<td>-10</td>
<td>-5</td>
</tr>
</tbody>
</table>

Figure 4.6: The value functions used to determine the value of a state in the search tree. \( H_1 \) is designed for opening the game and \( H_2 \) for end-game scenarios. The total number of occurrences of each feature type is listed in parentheses.

A solution will not have solutions in the standard game. The relaxed domain is formed by removing delete effects from actions. Once a card is available to be moved or played upon, it remains so throughout the entire search. This is a standard technique used to construct a path cost heuristic in general planning algorithms [10, 21]. We do not include such information into our heuristic, using it only for pruning purposes. A secondary dead-end pattern is also checked. This pattern involves states in which two cards of same color and rank are in the same Tableau stack blocking both of their paths to the Foundation and one of their Tableau build cards. Two dead end games are shown in Figure 4.7. The first, (a), is detected in the relaxed domain while (b) is detected by the secondary pattern. Of 10 million sample games 855,741 were detected as dead ends using this algorithm, revealing that at least 8.56\% of Klondike games are dead ends before a single card is played. This is a dramatic improvement over previous estimate of 1.19\% [43].
Figure 4.7: (a) A dead end detected in the relaxed domain. The Q♠ is blocking the K♥, K♦, and 2♠. (b) A dead end undetected in the relaxed domain, but detected as a special case pattern. After the 10♦ is built upon the J♠, the 10♦ and the 10♥ have no viable path to the Foundation stacks, as the 10♥ it is blocking the J♣, 9♥ and 5♥.

4.3.2 Caching Visited States

Each heuristic caches up to 5,000 visited nodes. The caches are filled with nodes that have been explored with a nested level greater than 0 (storing nodes at level 0 will quickly fill the cache with minimal savings). Cached nodes will be re-explored if visited with an increased degree of nesting.
4.3.3 Global Loop Prevention

The return path to the root node is checked against the current state. If the current node has been previously explored by the same heuristic at an identical nesting level, then the search is in an infinite loop. These loops are not guaranteed to be detected by caching alone because of the limited number of cached nodes.

4.3.4 Action Restrictions and Ordering

In order to speed search, some actions that duplicate other actions sequences are prohibited from the search tree. There are three different types of action that are prevented. In addition to this, the actions are ordered so that the most productive actions have priority.

**Local Loop Prevention** An action is eliminated from a state, s, in a search tree if, by definition, state s can be restored in a single action following that action. In order to maintain a complete search, these actions are included only when paired with a second action such that s cannot be trivially recovered.

**Foundation Progression** Cards in the Foundation are prohibited from moving back to the Tableau in the case that all cards of rank two less than its own are already in the Foundation (or if the card is an Ace or Two).

**King to Tableau** When multiple Tableau stacks are empty, only the furthest left is made open for a King (because they are all equivalent). When a King is
already the bottom-most card in a Tableau stack, it is not allowed to move to an empty stack.

**Action Ordering** Actions are searched in the following order: 1) Actions that move a card from Tableau to Foundation in which a face-down card is revealed. 2) Actions that move a card to the Foundation. 3) Actions that move a card from Tableau to Tableau in which a face-down card is revealed. 4) Actions that move a card from the Talon to the Tableau. 5) Actions that move a card from the Foundation to the Tableau. 6) Actions that move a card from Tableau to Tableau without revealing a face-down card.

In addition to these modifications, a time-limit may be placed on the search algorithm. A time limit of 8 seconds is implemented in the downloadable game located at [3]. A more complete pseudo-code description of the actual solver is described in Figure 4.8.

### 4.4 Empirical Results

We present performance results of the Thoughtful Solitaire Solver described in Section 4.3 with and without time constraints. All tests referenced in this paper were performed on Dell Power Edge 1850 1U rack servers with Dual 3.4 Ghz Intel Xeon Processors with 2048 KB cache and 4GB of SDRAM.
procedure multistage-nested-rollout \( s, h_0, h_1, \ldots, h_z, n_0, n_1, \ldots, n_z \) 

1. if \( s \) is the goal return WIN
2. if \( s \) is in a loop return LOSS
3. if time has expired or \( s \) is a dead-end or \( n_0 \) is -1 return \( h_0(s) \)
4. if this state has been cached for \( h_0 \) and \( n_0 \)
5. if \( z = 0 \) return \( h_0(s) \)
6. else return multistage-nested-rollout \( s, h_1, \ldots, h_z, n_1, \ldots, n_z \)
7. while \( h_0(s) \) is not LOSS
8. \( \text{val} = \max_a \text{multistage-nested-rollout} \ (\text{result}(s,a), h_0, h_z, n_0-1, \ldots, n_z) \) 
9. and let \( a' \) be the maximizing action
10. if \( \text{val} \) is WIN return WIN
11. if \( \text{val} \) is LOSS or \( (z \neq 0 \text{ and } \text{val} < h_0(s)) \)
12. if \( z = 0 \) return \( h_0(s) \)
13. else return multistage-nested-rollout \( s, h_1, \ldots, h_z, n_1, \ldots, n_z \)
14. \( s = \text{result}(s,a') \)

Figure 4.8: A description of the multistage nested rollouts algorithm with modifications described in Section 4.3. On line 2, the history from the current state back to the root search node is checked for loops. On line 4, the current state is checked against the cache for the current heuristic and nesting magnitude.

4.4.1 Thoughtful Lower Bounds

Using a multistage nested rollout search method with 3 levels of nested rollout search for H1 and a single rollout search \( (n=1) \) for H2, we were able to demonstrate that at least 82% of Thoughtful games have winning solutions, with each game taking an average of 32 minutes to complete. Previous results by [43], using nested rollouts on a standard Thoughtful Solitaire search tree reported 70% wins, taking 105 minutes per game. Figure 4.9 compares the two methods at various levels of complexity.
<table>
<thead>
<tr>
<th>Search Method</th>
<th>Win Rate (99% Conf.)</th>
<th>Games Played</th>
<th>Avg Time Per Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=0</td>
<td>16.17 ± 0.42</td>
<td>50,000</td>
<td>0.02 s</td>
</tr>
<tr>
<td>n=1</td>
<td>60.98 ± 0.56</td>
<td>50,000</td>
<td>0.74 s</td>
</tr>
<tr>
<td>n=2</td>
<td>74.03 ± 0.51</td>
<td>50,000</td>
<td>7.37 s</td>
</tr>
<tr>
<td>n=3</td>
<td>74.94 ± 0.50</td>
<td>50,000</td>
<td>3.64 s</td>
</tr>
<tr>
<td>n=4</td>
<td>77.99 ± 0.79</td>
<td>18,325</td>
<td>302.85 s</td>
</tr>
<tr>
<td>n=5</td>
<td>78.94 ± 0.47</td>
<td>49,956</td>
<td>212.38 s</td>
</tr>
<tr>
<td>n=6</td>
<td>79.17 ± 3.05</td>
<td>1,176</td>
<td>89.17 min</td>
</tr>
<tr>
<td>n=7</td>
<td>82.24 ± 2.71</td>
<td>1,323</td>
<td>31.67 min</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search Method</th>
<th>Win Rate (99% Conf.)</th>
<th>Games Played</th>
<th>Avg Time Per Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human</td>
<td>36.6 ± 2.78</td>
<td>2,000</td>
<td>20 min</td>
</tr>
<tr>
<td>n=0</td>
<td>13.05 ± 0.88</td>
<td>10,000</td>
<td>0.02 s</td>
</tr>
<tr>
<td>n=1</td>
<td>31.20 ± 1.20</td>
<td>10,000</td>
<td>0.67 s</td>
</tr>
<tr>
<td>n=2</td>
<td>47.60 ± 1.30</td>
<td>10,000</td>
<td>7.13 s</td>
</tr>
<tr>
<td>n=3</td>
<td>56.83 ± 1.30</td>
<td>10,000</td>
<td>96 s</td>
</tr>
<tr>
<td>n=4</td>
<td>60.51 ± 4.00</td>
<td>1,000</td>
<td>18.1 min</td>
</tr>
<tr>
<td>n=5</td>
<td>70.20 ± 8.34</td>
<td>200</td>
<td>105 min</td>
</tr>
</tbody>
</table>

(a) (b)

Figure 4.9: Results for rollout tests, from our solver (a), and as reported in Yan et al., 2005 (b). The degree of nesting is indicated by $n$ when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by $n_0$ for H1, and $n_1$ for H2. Results in (b) used a different heuristic which required a nested rollouts method slightly different from the method described in Figure 2.1. H1 and H2 are described in Figure 4.6.

4.4.2 Performance Under Time Constraints

Because individual games may take excessively long periods of time to complete their search, it is not practical to implement the unrestricted solver into a game intended for the casual user. It is necessary that a time limit be placed on the search of the tree. In order to demonstrate the feasibility of solving solitaire in real time, we have implemented our algorithm into a game of Thoughtful Solitaire.² In the demo, the search algorithm caps search times to 8 seconds. Figure 4.10 presents time statistics of the unrestricted solver along side percentage of games won for various instantiations of our solver, using either one or two heuristics. Figure 4.11 displays the performance curve for four different heuristic search levels as a function of allotted search time. As the time allotted to each of the search methods increases from $2^{-3}$ to $2^{10}$ seconds, performance increases.

²web.engr.orst.edu/~ronny/k.html
<table>
<thead>
<tr>
<th>Search Method</th>
<th>Avg Time per Win</th>
<th>Avg Time per Loss</th>
<th>Avg Time Overall</th>
<th>% Wins &lt; 1 sec</th>
<th>% Wins &lt; 4 sec</th>
<th>% Wins &lt; 16 sec</th>
<th>% Wins &lt; 64 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=0</td>
<td>0.06 sec</td>
<td>0.02 sec</td>
<td>0.02 sec</td>
<td>16.17%</td>
<td>16.17%</td>
<td>16.17%</td>
<td>16.17%</td>
</tr>
<tr>
<td>n=1</td>
<td>0.60 sec</td>
<td>0.95 sec</td>
<td>0.74 sec</td>
<td>49.03%</td>
<td>60.38%</td>
<td>60.97%</td>
<td>60.98%</td>
</tr>
<tr>
<td>n=2</td>
<td>1.35 sec</td>
<td>10.50 sec</td>
<td>3.64 sec</td>
<td>56.98%</td>
<td>69.82%</td>
<td>74.23%</td>
<td>74.94%</td>
</tr>
<tr>
<td>n=3</td>
<td>1.45 sec</td>
<td>24.29 sec</td>
<td>7.38 sec</td>
<td>52.38%</td>
<td>67.88%</td>
<td>73.23%</td>
<td>74.03%</td>
</tr>
<tr>
<td>n=4</td>
<td>0.22 min</td>
<td>16.00 min</td>
<td>3.54 min</td>
<td>56.87%</td>
<td>70.47%</td>
<td>76.11%</td>
<td>78.06%</td>
</tr>
<tr>
<td>n=0:n0=1</td>
<td>0.29 min</td>
<td>21.89 min</td>
<td>5.05 min</td>
<td>53.41%</td>
<td>69.22%</td>
<td>75.33%</td>
<td>77.14%</td>
</tr>
<tr>
<td>n=1:n1=1</td>
<td>0.44 min</td>
<td>176.27 min</td>
<td>31.67 min</td>
<td>57.82%</td>
<td>72.26%</td>
<td>78.31%</td>
<td>80.57%</td>
</tr>
<tr>
<td>n=2:n2=1</td>
<td>2.12 min</td>
<td>420.02 min</td>
<td>89.18 min</td>
<td>55.44%</td>
<td>69.98%</td>
<td>74.92%</td>
<td>77.21%</td>
</tr>
</tbody>
</table>

Figure 4.10: Results for tests under time constraints. The degree of nesting is indicated by $n$ when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by $n_0$ for H1, and $n_1$ for H2. Use of multiple heuristics wins a higher percentage of games in a shorter time period than using a single heuristic. H1 and H2 are described in Figure 4.6.

Figure 4.11: Comparison of winning percentage (as a function of time) for various levels of nesting of the multistage nested rollout algorithm. Error bars indicate 99% confidence intervals.
This solver has demonstrated that no less than 82% and no more than 91.44% of instances of Klondike Solitaire have winning solutions, leaving less than 10% of games unresolved. We are confident that these bounds can be improved by using a more complex pruning method to decrease the upper bound and new heuristics to improve the lower bound.
Chapter 5 – Probabilistic Planners for Klondike

With our successful deterministic planner for Thoughtful Solitaire, we began to investigate the probabilistic domain of Klondike Solitaire, described in Chapter 4.1. Because it is a probabilistic problem, solving an instance of Klondike is much more difficult than solving the equivalent game of Thoughtful, but we will show that having access to the solver developed for Thoughtful proved key in our experiments.

Unlike many of the probabilistic planning domains introduced by planning researchers, Klondike Solitaire is independently developed and has a broad appeal to the general public. In addition to being highly stochastic, it involves complicated reasoning about actions with many constraints. There also exist many variants of the game which allows for a systematic study with different planning approaches. Klondike is also interesting in that it poses several problems for probabilistic planning including representation and search, as we argue below.

Representing Klondike in a standard probabilistic planning language, such as PPDDL, is problematic because there is no standard distribution for each action. Unlike standard probabilistic planning domains, each action may require a different distribution of possible outcomes based on the number and identity of the current set of face-down cards. Defining Klondike in such a manner is cumbersome and complicated, as the representation can potentially increase exponentially in the size of the state description. Without some language extension that allows the
enumeration of a set of unseen objects or events and removal of those objects from
the set once they have been encountered, the application of standard planning
algorithms to Klondike and other POMDP style domains will be problematic.

While it is not clear how to compactly describe Klondike in PPDDL, it is
not difficult to implement a simulation model for Klondike. Moreover, Monte-
Carlo sampling techniques such as Hindsight Optimization and UCT have shown
great promise in recent years and only require simulation models. Monte-Carlo
techniques approximate the values of states and actions by evaluating trajectories
through the modeled space, thus making it possible to reason about actions without
exploring entire sections of a search space.

Anecdotal evidence suggests that typical human players win between 7 and
15% of games [16, 35]. One common method of scoring games, known as “Las
Vegas” style, pays players five fold their up-front per-card cost for each card that
reaches a foundation stack. Such a payoff suggests that a strategy that wins over
20% of played games should be considered a success. In addition to these assumed
baselines, we used a greedy heuristic based on prioritized action preferences and
a random search as a baseline for our own work. We found that a random strat-
 egy won 7.135% of games while a greedy strategy based on the prioritized action
heuristic described in Chapter 4.3.4 won 12.992% of games, each tested on one
million randomly generated games. These results appear to confirm the estimates
of human play suggested by other sources [16]. As with our other strategies, the
random action selection mechanism utilized the greedy search method in states
with no face-down cards.
5.1 Hindsight Optimization and Averaging Over Clairvoyancy

Hindsight Optimization (HOP) is a straightforward way to use the existing deterministic planner from Thoughtful Solitaire in this stochastic environment. The general idea behind HOP is to estimate the value of each action in a state by way of calls to a deterministic planner on different determinizations of a probabilistic planning problem. In particular, a value of a state is estimated by forming a set of determinized problems from the state, solving each one with a deterministic planner, and then averaging the results. HOP uses these estimates to select actions via one-step greedy look-ahead. With our deterministic planner, applying HOP to Klondike is relatively straightforward. At each decision point, an action is taken and the unseen face down cards are shuffled. The locations of all of the cards are revealed to our deterministic solver which determines if a solution can be found given the action taken and the shuffling of the cards. In theory, the action that leads to the highest percentage of solutions will (on average) be the best choice.

Our deterministic planner, described in Chapter 4 is based on a nested rollout search algorithm. The time complexity of this search grows exponentially with the nesting or search level. The increase in search time prevented us from using the same high level of nesting that was used to explore in Thoughtful. Unlike Thoughtful, when HOP is applied to Klondike, many determinized problems must be solved for each action selection. In particular, in some of our tests, we solved 100 determinized problems for each action at each decision point resulting in many thousands of calls to the deterministic planner during the course of a single game.
Results for HOP Klondike Trials

<table>
<thead>
<tr>
<th>#Samp / decis</th>
<th>Search Level</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>22.96 ± 0.34</td>
<td>1000000</td>
<td>43.54</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>27.20 ± 0.80</td>
<td>20614</td>
<td>689.78</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>25.24 ± 0.78</td>
<td>20618</td>
<td>833.26</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>26.13 ± 2.66</td>
<td>1814</td>
<td>9056.58</td>
</tr>
</tbody>
</table>

Table 5.1: Results for various HOP tests

Finally, in our HOP experiments, as well as our UCT experiments, we utilized the greedy deterministic described in Chapter 4.3.4 to improve overall performance. Specifically, when all of the cards are finally revealed in a game, our greedy search is used to attempt to solve it, and if it can’t then HOP is resumed to select the next action.

**Results**

Our results for HOP based tests can be seen in Table 5.1. These results are impressive compared to our baseline, more than doubling the performance of the greedy method, even outperforming the “Las Vegas” standard of 20%. From these results, it appears to be more effective to increase the sampling amount than increasing the search level.

One potential problem with HOP is that it can be overly optimistic in some cases due to the determinization process. As described in [46] the degree to which this optimism is detrimental can be impacted by the type of determinization process used. In particular, the strongest guarantees were shown for determinization processes that produced independent futures. Unfortunately, the determinization process we used for Klondike, which was selected to make our Thoughtful Solitaire
planner applicable, does not have this property. Because of this, there is reason to believe that the HOP approach can fall into some of the pitfalls associated with being overly optimistic. In particular, below we show an example where our HOP approach will select an action that leads to eventual failure, even when there are actions available that will lead to certain success. This type of failure is discussed in [34] as a downfall of averaging over clairvoyancy.

In order to validate this effect, we have designed a simple instance where the deterministic planner may optimistically choose an action that can lead to a state where the goal can be reached only by random guessing. We illustrate this effect in Figure 5.2. From Figure 5.2a, the following sequence of actions will lead to a state in which a win is guaranteed, independent of the unseen identities of the four face-down cards: K♦→T4, Q♣→K♦ (leaving T2 empty), J♦→Q♣, J♥→Q♠, K♥→T2, Q♠→K♥ (leaving T3 empty), 9♦→8♦, 9♥→8♥, 10♦→9♦, 10♥→9♥, K♣→T3. Alternatively, choosing K♣→T4 from 5.2a may lead to a state (shown in 5.2b) in which a guaranteed solution exists only in the case that the identity of all cards is known to the planner. By guessing incorrectly from 5.2b, the planner may reach the dead end shown in 5.2c. Because the planner fixes the location of the cards prior to searching for a solution, the future states are correlated, and both of these initial actions (K♦→T4 and K♣→T4) will be valued equally, despite the fact that K♦→T4 is clearly preferred in this situation.

Unfortunately, it is not clear how to implement HOP trajectories in Klondike with independent futures with the provided value function and search mechanism. Because of these possible problems we were determined to investigate alternative
Figure 5.1: A simple example provided in [46] when correlated futures will cause a deterministic planner to miscalculate the value of an action, in this case, the value of action $b$ from the root. Taking action $a$ will surely lead to a win, while taking action $b$ will only lead to a win a fraction of the time.

methods that did not rely so heavily on deterministic planners.

5.2 UCT Approaches

UCT is a Monte-Carlo planning algorithm for probabilistic domains. UCT performs rollout trajectories from some initial state to model the value of each of the actions available from that state. UCT incrementally builds a search tree and stores the values for the actions taken from the nodes in the tree. By approximating the values for each of the actions already taken in the tree, UCT can bias the exploration towards the most promising paths.

Each node $s$ in the UCT tree stores the number of times the node has been visited in previous rollouts $n(s)$, the number of times each action $a$ has been explored in $s$ in previous rollouts $n(s, a)$, and a current action value estimate for
Figure 5.2: (a) A state in Klondike Solitaire. (b) A possible state after $K\spadesuit \rightarrow T4$. (c) A dead end forced by having to guess.

Each action $Q_{UCT}(s, a)$. Each rollout begins at the root and actions are selected via the following process. If the current state contains actions that have not yet
been explored in previous rollouts, then a random action is chosen from among the unselected actions. Otherwise if all actions in the current node $s$ have been explored previously then UCT selects the action that maximizes an upper confidence bound given by

$$Q_{\text{UCT}}^n(s, a) = Q_{\text{UCT}}(s, a) + c \sqrt{\frac{\log n(s)}{n(s, a)}},$$

(5.1)

where $c$ is a constant that is typically tuned on a per domain basis, which was set to 1 in our Solitaire domain. After selecting an action, it is simulated and the resulting state is added to the tree if it is not already present. This action selection mechanism is based on the UCB bandit algorithm and attempts to balance exploration and exploitation. The first term rewards actions whose values are currently promising, while the second term adds an exploration reward to actions that have not been explored much and goes to zero as an action is explored more frequently.

Finally, after the trajectory reaches a terminal state the reward for that trajectory is calculated to be 0 or 1 depending on whether the game was lost or won. The reward is used to update the action value function of each state along the generated trajectory. In particular, the updates maintain the counters $n(s, a)$ and $n(s)$ for visited nodes in the tree and update $Q_{\text{UCT}}(s, a)$ for each node so that it is equal to the average reward of all rollout trajectories that include $(s, a)$ in their path. Once the desired number of rollout trajectories have been executed UCT returns the root action that achieves the highest value. Performance results for UCT are presented in Table 5.2 in a later section for direct comparison to the performance of other methods. These results far surpass the results attained by HOP, winning over 34% of games. This is somewhat surprising considering our
UCT implementation does not utilize prior domain knowledge as it explores and builds the stochastic search tree.

Prior to our work, implementations of UCT for Go included a complete tree search at each step that added a new node to the UCT tree. This prevented states from being duplicated in a UCT tree. Like Go, duplicating states is a concern in Klondike. Occasionally two actions may be available simultaneously that can be taken in any order without affecting the immediate outcome of the other action. If the tree is allowed to grow separate branches for both of these cases, predictive power will be lost because the sampled trajectories will be split between the two branches. Because of the nature of Go, this type of behavior is very common. As opposed to Go, Klondike states are not likely to be repeated from one trajectory to another due to the nature of revealing face-down cards. The benefit of checking for duplicate states in Klondike is much lower than for Go, while the cost of searching an entire tree for a repeated state remains high. Early in our research, we decided to not check for states repeated outside of the current trajectory and to instead spend that time running additional trajectories in the tree.

All of our results prior to this publication employ this version of UCT. This method allows the number of children of any single action to potentially be as great as the number of trajectories in the entire tree. In some of the trees generated with a large number of trajectories, we observed that trees with large branching factors were taking measurably longer compared to Sparse trees with a fixed number of children per action. This result was reported in [4].

Recently, we have determined that it is beneficial to conduct local (as opposed
to global) searches for repeated states. After an action is taken, the children of that action can be quickly examined for a repeated state and that trajectory followed if a repeated state is found. For Klondike, this implies that the branching factor of actions is effectively limited to 21 - the number of cards that are face down at the beginning of each game. Only those actions that reveal the identity of these cards are probabilistic, and 21 branches can accommodate all of the possible outcomes of these probabilistic actions.

5.2.1 Combining UCT with HOP

While the UCT results showed significant improvement over HOP using existing deterministic Solitaire planners, performance appears to have leveled off with the number of trajectories. This led us to consider a new Monte-Carlo approach, HOP-UCT, which aims to explore the potential for combining HOP and UCT. As already noted, one of the potential shortcomings of our earlier HOP experiments was that the deterministic planners required correlated futures which can lead to poor decisions in Solitaire. This motivates trying to develop a HOP-based approach that can operate with independent futures, where the outcomes of each state-action pair at each time step are drawn independently of one another. This can be done in a natural way by using UCT as a deterministic planner for HOP.

To understand the algorithm note that an independent future can be viewed as a deterministic tree rooted at the current state. Each such tree can be randomly constructed by a breadth-first expansion starting at the root that samples a single
child node for each action at a parent node. The expansion terminates at terminals. Given a set of such trees one could consider running UCT on each of them and then averaging the resulting action values across trees, which would correspond to HOP with UCT as the base planner. Unfortunately each such tree is exponentially large making the above approach impractical.

Fortunately it is unnecessary to explicitly construct a tree before running UCT. In particular, we can exactly simulate the process of first sampling a deterministic tree and then running UCT by lazily constructing only the parts of the deterministic tree that UCT encounters during rollouts. This idea can be implemented with only a small modification to the original UCT algorithm [24]. In particular, during the rollout trajectories whenever an action $a$ is taken for the first time at a node $s$ we sample a next node $s'$ and add it as a child as is usually done by UCT. However, thereafter whenever $a$ is selected at that node it will deterministically transition to $s'$. The resulting version of UCT will behave exactly as if it were being applied to an explicitly constructed independent future. Thus, the overall HOP-UCT algorithm runs this modified version of UCT for a specified number of times, averages the action-values of the results and selects the best action.

5.2.2 Ensemble-UCT

The HOP process of constructing UCT trees and combining the results begs the question of whether other ensemble style methods will be successful. We constructed an Ensemble-UCT method that generates UCT trees and averages the
values of the actions at the root node in a similar manner to HOP-UCT. We would expect Ensemble-UCT to require fewer trees than HOP-UCT to achieve a similar reduction in variance of the estimated action values. In comparing HOP-UCT and Ensemble-UCT the total number of simulated trajectories becomes a basic cost unit and we will be able to gauge the benefit of spending trajectories on new or existing UCT trees.

5.2.3 Results

Comparing the performance of HOP-UCT and UCT trials (seen in Table 5.2) suggests that sampling multiple UCT trees boosts performance and decreases computing time compared to UCT trees with an equivalent number of total trajectories. We compare the 2000 trajectory UCT and the 100×20 HOP-UCT trials which both utilize a total of 2000 trajectories. Not only does the HOP-UCT approach slightly outperform the UCT method, it requires less than one third the time to do it. The performance of the 200×20 Ensemble-UCT trials also illustrate the trade off between performance and time complexity, which averages a higher winning percentage than other methods. It is faster and more successful than the 2000 trajectory UCT method and the 1000×5 HOP-UCT method. However, it is still within the 99% confidence interval of the 100×20 trajectory HOP-UCT method, which requires far less computing time.
5.3 Sparse UCT

One observation we made regarding HOP-UCT was that the time required per rollout trajectory was significantly less than the time for regular UCT. The primary reason for this is that the time for rollouts in regular UCT is dominated by the time to sample a next state given a current state and action. While this is generally a fast process, for example, in Klondike requiring that we keep track of unseen cards and randomly draw one, the time per trajectory is linearly related to the sampling time, making the cost of sampling very significant. The modified version of UCT used for HOP-UCT only required sampling a new state the first time an action was selected at a node and thereafter no sampling was required, which lead to a significant speedup.

This motivated us to consider a new variant of UCT called Sparse UCT, which limits the number of calls to the sampling process at each node in the tree to a specified sampling width $w$. In particular, the first $w$ times an action $a$ is selected at a node $s$ the usual sampling process is followed and the resulting children are added to the tree. However, thereafter whenever action $a$ is selected at a node one of the already generated $w$ children is selected at random. This random selection from $w$ existing children is generally significantly faster than calling the sampling process and thus can lead to a speedup when $w$ is small enough. However, for very small $w$ the approximation to the original UCT algorithm becomes more extreme and the quality of decision making might degrade. Note that when $w = 1$ the behavior of Sparse UCT is equivalent to that of HOP-UCT. The method for building a
Sparse UCT tree is outlined in Algorithm 4. This method can also be extended to Ensemble-Sparse-UCT to evaluate root action values based on averaged values from Sparse-UCT trees.

**Input:**
- $s = $ initial state
- $y =$ # of trajectories that generate uct tree
- $w =$ # sampling width

**Output:** values for each action in $s$

```plaintext
1 $s_0 = s$;
2 for $i = 1$ to $y$ do
3   $s = s_0$;
4   while not $s$.win AND not $s$.dead-end do
5     if all actions of $s$ have been sampled then
6       $a = \arg \max_a Q^{\oplus}_{UCT}(s,a)$;
7     else
8       $a =$ unsampled action chosen with default policy;
9     if $s$.childCount[$a] == $w then
10       $s' =$ randomly choose existing child of $(s,a)$;
11     else
12       $s' =$ transition$(s,a)$;
13       new child for $(s,a) = s'$;
14       $s$.childCount[$a]++;
15     $s = s'$;
16   update all visited $(s_i,a_j)$ pairs with $(s$.win ? 1 : 0);
```

**Algorithm 4:** Generate UCT Tree Algorithm

In addition to improved rollout times, there is another potential benefit of Sparse UCT for problems where the number of possible stochastic outcomes for the actions is large. In such problems, UCT will rarely repeat states across different rollouts. For example, if an action has a large number of uniformly distributed outcomes compared to the number of rollouts then it is unlikely that many of those outcomes will be visited more than once. As a result UCT will not accumulate use-
ful statistics for nodes in the tree, leading to poor action-value estimates. Sparse UCT puts an upper limit on how many children can be generated for each action at a node and thus will result in nodes to be visited repeatedly across rollouts accumulating non-trivial statistics. However, as the sampling width $w$ becomes small the statistics are based on coarser approximations of the true problem, leading to a fundamental trade-off in the selection of $w$. Below we consider what theoretical guarantees can be made regarding this trade-off.

### 5.3.1 Results

The experimental (shown in Table 5.2) results seem to indicate that there is little difference between UCT and Sparse UCT when comparing equivalent numbers of samples per UCT tree. A particularly informative example of this can be seen in the progression of four experiments that build a UCT tree with 1000 trajectories. There appears to be a decreased performance if the UCT tree has a sampling width of 1, but even with a small sampling width of 5, performance increases to within the confidence interval of the experiments with sampling width of 10 and infinity (the latter reported in the UCT section of Table 5.2). These results would seem to indicate that the number of outcomes associated with each action is not a significant limiting factor for UCT in Solitaire. Also interesting is the increase in time required to generate each tree. Experiments with a sampling width of infinity at most double the time of those experiments with sampling width of 1. This may be explained, in our experiments by the cost of generating random sequences,
which will be reduced with the frequent re-visits of existing states in those trees with small sampling width.
<table>
<thead>
<tr>
<th>#traj</th>
<th>samp. width</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>inf</td>
<td>24.64±3.04</td>
<td>1331</td>
<td>44.50</td>
</tr>
<tr>
<td>1000</td>
<td>inf</td>
<td>34.24±4.52</td>
<td>733</td>
<td>1042.56</td>
</tr>
<tr>
<td>2000</td>
<td>inf</td>
<td>34.41±4.01</td>
<td>933</td>
<td>2980.44</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#traj</th>
<th>#tree /dec</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>20</td>
<td>35.47±4.00</td>
<td>953</td>
<td>767.57</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
<td>30.58±4.53</td>
<td>690</td>
<td>3531.12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#traj</th>
<th>samp. width</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>22.65±2.41</td>
<td>2000</td>
<td>40.11</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
<td>28.66±4.21</td>
<td>764</td>
<td>566.65</td>
</tr>
<tr>
<td>1000</td>
<td>5</td>
<td>33.98±1.63</td>
<td>5583</td>
<td>495.68</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
<td>33.79±3.11</td>
<td>1536</td>
<td>747.82</td>
</tr>
<tr>
<td>2000</td>
<td>4</td>
<td>35.62±2.58</td>
<td>2288</td>
<td>1485.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#traj</th>
<th>#tree /dec</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>20</td>
<td>36.97±1.92</td>
<td>4206</td>
<td>2280.55</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>32.86±1.57</td>
<td>5986</td>
<td>669.90</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#traj</th>
<th>samp. width</th>
<th>Win Rate (99% conf.)</th>
<th># Games</th>
<th>Av. sec / Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>50</td>
<td>35.27±2.16</td>
<td>3249</td>
<td>1302.38</td>
</tr>
</tbody>
</table>

Table 5.2: Results for various UCT tests. These results, presented at ICAPS 2009 [4] did check to eliminate duplicate states in the UCT tree.
Chapter 6 – A Unified Approach

There are striking similarities among the UCT-based approaches we discussed in Chapter 5. UCT, Sparse-UCT, HOP-UCT and ensemble UCT can be described as a single algorithm with three variables. These three variables are: 1) the number of UCT trees in each decision ensemble, 2) the number of trajectories used to generate each UCT tree and 3) the branching factor of the stochastic outcome of each action. Figure 6.1 is a graphical illustration of these algorithms, where each variable represents an axis in 3-dimensional space. The description of these algorithms in the context of a unified approach will allow for a more complete analysis of the effects of these parameters. In this section, we will evaluate the effect that each of these variables has on performance, in terms of both winning percentage and average time spent per game.

6.1 Performance Characteristics of Sparse-Ensemble-UCT

In our experiments, we have tuned the variables within this family of UCT algorithms for time and performance. Increasing the number trees in an ensemble and the number of trajectories used to create each tree will both reduce the variance of the classifier, but at the cost of increased time. Reducing the branching factor of the trees will save time, but will increase the variance of each tree. In
Figure 6.1: A graphical illustration of the variants of UCT in terms of three variables: number of trees per decision, number of trajectories used to construct each tree and the action branching factor of those trees. UCT, Sparse-UCT, Ensemble UCT and other variants can be described in terms of these variables.

Our published research, we largely relied on intuition and random guessing rather than deliberate time/performance analysis to maximize our performance in time to meet publication deadlines. In this section, we more fully analyze the tradeoff of time and performance in relation to these three parameters.

6.1.1 Evaluating Time

Our published research represents the first lower bound on the performance of an optimal policy in the game of Klondike Solitaire. In the absence of a theoretical bound, it was important to demonstrate tight confidence intervals on the empirical
bounds demonstrated by our simulations. These tight confidence intervals required thousands of games for each tested policy, making the time required to complete each game just as important as the performance of those policies.

In Figure 6.2, we can see an evaluation of the average time required to complete a game as a factor of the number of trajectories used to construct each tree. This graph contains three plots, one for trees with an action branching factor of 1, of 21 and of an unlimited branching factor. The data points for branching factor of 1 and 21 were generated with local search for duplicate tree states, effectively limiting the maximum branching factor to 21. The four data points shown for branching factor of infinity are taken from data found in Table 5.2, which was generated without checking to remove duplicate states, allowing the theoretical branching factor to grow with the number of trajectories used to generate the tree.

Many things factor into the time required to complete each game. Time is required to construct each tree, and time is required to evaluate the final tree to make a decision. Some games may be more easily solved than difficult games, requiring less time. Trees that make poor decisions may quickly terminate as they fail to find solutions to certain games while other trees may take a significantly longer average time per game as their percentage of solved games increases.

We notice that the curves of our three data sets are not linear. We can conclude that it is more time consuming to aggregate a new trajectory into an established tree as compared to a new tree, and the greater number of trajectories that have already been collected in a single tree, the more costly (on average) the next trajectory will be to aggregate.
Figure 6.2: A comparison of the time per game (graphed on a log scale) as a function of the number of trajectories used to generate a UCT tree and the size of the allowed branching factor of those UCT trees.

Each of these three curves appear to be roughly following the same general performance characteristics, but there is a visible difference between the speed of trees with a greater branching factor and trees with a smaller branching factor. When an action has reached its branching limit, a previously recorded branch is taken, and it is not necessary to simulate the effects of the chosen action. It will be faster to traverse trees that quickly reach these limits - trees with a smaller branching factor.

In Figure 6.3 we see a similar graph depicting the average time per game as
a function of the number of trees increases in the decision ensemble. For these experiments, the number of trajectories used to construct each tree was restricted to 1.

Figure 6.3: A comparison of the time per game (graphed on a log scale) as a function of the number of trees per decision ensemble.

As opposed to the observed performance curve from Figure 6.2, this curve is very linear. It appears that any additional overhead associated with increasing the number of trees is relatively constant as the number of trees increases.
6.1.2 Evaluating Performance

Although the time required to complete each game is important, our main interest is in the overall performance of each policy. Figure 6.4 displays the performance of a single UCT tree as a function of the number of trajectories used to create the tree. Results are displayed for trees with a branching factor of 1 and trees with a branching factor of 21. For these experiments, there is little difference in performance until more than $2^5$ trajectories are used to generate the tree. After that point, the performance of the full-breadth trees increase substantially over the
performance of the restricted trees. Performance of trees with branching factor of 1 appear to be asymptotically converging to around 30%. No such convergence can be inferred from the performance of trees with full branching factor. In Section 6.1.1, we also included data for trees with an unbounded branching factor.

Figure 6.5: A comparison of how performance increases with three variables that define our family of UCT algorithms: 1) # of trees in the ensemble, 2) # of trajectories used to construct those trees and 3) the branching factor. Performance is measured against the average time to complete each game (on a log scale). Points that appear on the upper left edge can solve the most games in the smallest amount of time. Based on these trajectories, it is initially most beneficial to add trajectories to existing trees, but in the limit it appears that highest performance for the search time can be achieved by increasing branching factor, ensemble size and trajectories per tree.

Of course, the most important metric is how these variables combine to influ-
ence both time and performance. The most complete picture of performance can be constructed by measuring performance as a function of the speed of the implemented policies. This result is presented in Figure 6.5. In this graph, we plot the average time required to complete a game on a log scale against the average performance of those games. Points for the first data set increased the number of trees in an ensemble, where trees were generated with a single trajectory. The second and third data sets measured the performance change as the number of trajectories used to generate a single UCT increased, for trees with branching factors of 1 and 21, respectively. The fourth data set illustrates the performance of the algorithm with \( n \) trajectories per tree, \( n \) trees in the ensemble and a branching factor of \( n \). For this data set, each green triangle represents \( n \) as it increases by powers of 2, with the final point representing performance of an ensemble of 64 UCT trees that were each generated with 64 trajectories and a (maximum) branching factor of 21.

The resulting conclusion of Graph 6.5, and of this section in general is that for this implementation of Klondike solitaire, it appears to be initially more cost effective to increase the branching factor and the number of trajectories used to construct a tree prior to increasing the number of trees in a decision ensemble. In the limit, it appears that highest performance for the search time can be achieved by increasing some combination of all of these factors. As the bias of these trees should not be affected by these variables, we can assume that the variance of the constructed trees is reduced more quickly in this same order. From this data set, the crossover point appears to be around the time when the third and fourth data sets both require around \( 2^4 \) seconds to complete each game. This occurs when
64 trajectories are used to generate a single tree, and when 8 trees are generated with 8 trajectories each with a branching factor of 8. It is clear that all three of these variables contribute significantly to the performance of the algorithm, and the appropriate values for these variables will depend on the available time and the desired performance.

6.2 Analysis of Sparse UCT

The original UCT algorithm has the theoretical property that its probability of selecting a non-optimal action at a state decreases as $\text{poly}\left(\frac{1}{t}\right)$ where $t$ is the number of UCT trajectories. Here we consider what can be said about our UCT variants. We consider finite horizon MDPs, with a horizon of $D$, and for simplicity restrict to the case where the range of the reward function is $[0, 1]$. Our first variant, Sparse UCT, is identical to UCT only it considers at most $w$ outcomes of any state action pair when constructing the tree from the current state, which limits the maximum tree size to $O((wk)^D)$ where $k$ is the number of actions.

To derive guarantees for Sparse UCT, we draw on ideas from the analysis of the sparse sampling MDP algorithm [23]. This algorithm uses a generative MDP model to build a sparse expectimax tree of size $O((wk)^D)$ rooted at the current state $s$ and computes the Q-values of actions at the root via expectimax search. Note that the tree construction is a random process, which defines a distribution over trees. The key contribution of that work was to show bounds on the sampling width $w$ that guarantee near optimal Q-values at the root of a random tree with
high probability. While the analysis in that paper was for discounted infinite horizon MDPs, as shown below, the analysis extends to our finite horizon setting.

For the purposes of analysis, consider an equivalent view of Sparse UCT, where we first draw a random, expectimax tree as in sparse sampling, and then run the original UCT algorithm on this tree for \( t \) trajectories. For each such tree there is an optimal action, which UCT will select with high probability as \( t \) grows, and this action has some probability of differing from the optimal action of the current state with respect to the true MDP. By bounding the probability that such a difference will occur we can obtain guarantees for Sparse UCT. The following Lemma provides such a bound by adapting the analysis of sparse sampling. In the following we will define \( Q^*_d(s,a) \) to be the optimal action value function with \( d \) stages-to-go of the true MDP \( M \). We also define \( T_d(s,w) \) to be a random variable over sparse expectimax trees of depth \( d \), derived from the true MDP, and rooted at \( s \) using a sampling width of \( w \). Furthermore, define \( \hat{Q}^*_d(s,a) \) to be a random variable that gives the action values at the root of \( T_d(s,w) \).

**Lemma 1.** For any MDP with finite horizon \( D \), \( k \) actions, and rewards in \([0, 1]\), we have that for any state \( s \) and action \( a \), \( \|Q^*_d(s,a) - \hat{Q}^*_d(s,a)\| \leq d\lambda \) with probability at least \( 1 - d(wk)^d \exp\left(-\frac{\lambda^2}{177w}\right) \).

This shows that the probability that a random sparse tree leads to an action value estimate that is more than \( D\lambda \) from the true action-value decreases exponentially fast in the sampling width \( w \) (ignoring polynomial factors). We can now combine this result with one of the original UCT results. In the following we denote the error probability of Sparse UCT using \( t \) trajectories and sampling width \( w \) by
$P_e(t, w)$, which is simply the probability that Sparse UCT selects a sub-optimal action at a given state. In addition we define $\Delta(s)$ to be the minimum difference between an optimal action value and sub-optimal action value for state $s$ in the true MDP, and define the minimum Q-advantage of the MDP to be $\Delta = \min_s \Delta(s)$.

**Theorem 1.** For any MDP with finite horizon $D$, $k$ actions, and rewards in $[0, 1]$, if

$$w \geq 32 \frac{D^4}{\Delta^2} \left( D \log \frac{16kD^5}{\Delta^2} + \log \frac{D}{\delta} \right)$$

then $P_e(t, w) \leq \text{poly} \left( \frac{1}{t} \right) + \delta$.

**Proof.** (Sketch) Theorems 5 and 6 of [24] show that for any finite horizon MDP the error rate of the original UCT algorithm is $O(t^{-\rho(\Delta)^2})$ where $\rho$ is a constant. From the above lemma if we set $\lambda$ equal to $\frac{\Delta}{4D}$ we can bound the probability that the action-values of a randomly-sampled sparse tree are in error by more than $\frac{\Delta}{4}$ by $D(wk)^D \exp \left( -\left( \frac{\Delta}{4D^2} \right)^2 w \right)$. It can be shown that our choice of $w$ bounds this quantity to $\delta$. Note that this bounds the probability that the minimum Q-advantage of the sparse tree is greater than $\frac{\Delta}{2}$ by $\delta$. The UCT result then says that for trees where this bound holds the error probability is bounded by a polynomial in $\frac{1}{t}$. The theorem follows by applying the union bound. \qed

This result shows that for an appropriate value of $w$, Sparse UCT does not increase the error probability significantly. In particular, decreasing the error $\delta$ due to the sparse sampling requires an increase in $w$ that is of only order $\log \frac{1}{\delta}$. Naturally, since these are worst case bounds, they are almost always impractical, but they do clearly demonstrate that the required value of $w$ does not depend on
the size of the MDP state space but only on $D$, $k$, and $\Delta$. It is important to note that there is an exponential dependence on the horizon $D$ buried in the constants of the UCT term in the above bound. This dependence is unavoidable as shown by the lower-bound in [23].
Chapter 7 – Exploring Other Update Methods: Expectimax UCT

While the UCT algorithm has been very successful, it has been criticized because its optimistic exploration policy can lead to hyper-exponential regret in degenerate search spaces [14]. The success of UCT across varied domains suggests that such degenerate search spaces are rare. We offer an example that will lead to similar problems. Rather than adapting the exploration policy, as suggested in [14], we investigate an alternate update policy.

7.1 Expectimax Updates

After each trajectory, UCT updates the value of each action to be the average reward received from every trajectory that has passed through it. Following the UCT algorithm, many of these trajectories will presumably include poor choices that lead to low rewards. Including these low rewards in the value of actions will misrepresent the value of taking the action if other choices would have produced good rewards. This misrepresentation can be corrected by storing a value for each state in addition to storing the value of each action. The value of the state becomes the highest value of the actions available in that state. The values of actions are updated as an average of the values of the states that it has reached and the value of each state in the trajectory is updated with the new values of the actions and
the values of the actions are updated. As UCT continues to explore at the distant edges of the search tree, actions at the root will be partially buffered from poor decisions because the state values will be updated with the good actions they could have taken rather than the poor actions that were taken in the name of exploration.

We illustrate a simple example of this type of behavior in Figure 7.1. We see that taking action $a$ from the root will lead to a certain reward of 3.0. Taking action $b$ will lead with equal probability to either a state that will certainly receive a reward of 0, or to state $S_3$ from which action $b$ will receive a reward of 7.0. It is clear that taking action $b$ is worth an average value of 3.5, while taking action $a$ is worth 3.0.

Figure 7.1: An example of a simple search tree in which average reward update may lead UCT exploration to temporarily underestimate the true value of taking action $b$ from the root node, $S_0$, where expectimax updates using the same exploration policy will correctly evaluate the value of the same action with far fewer trajectories.
Even if initial exploration is lucky, reaching $S_3$ from $(S_0, b)$ and then choosing action $b$ from $S_3$, evaluating $(S_0, b)$ as an average of trajectory returns will eventually undervalue the state on subsequent visits to $S_3$, which will require exploration of actions $a$ and $c$. The value of $(S_0, b)$ will suffer from these poor returns, which will sink below $(S_0, a)$, despite the fact that exploration from $(S_0, b)$ has already revealed an action with a reward far above that of $(S_0, a)$.

If an Expectimax return is utilized instead, the state value stored at $S_3$ will not be devalued when actions $a$ and $c$ are taken in the name of exploration. From our example, under our “lucky” initial circumstances, an initial value of 7.0 is placed on $(S_0, b)$. Under further exploration of other actions from $S_3$, this value will not decrease because the expected value of $S_3$ is 7.0 and this value, rather than the poor returns from exploration, will be used to update $(S_0, b)$. Thus, even under unlucky exploration, where $(S_0, b)$ initially leads to $S_2$, only a small number of trajectories leading to $S_3$ will be needed to evaluate $(S_0, b)$ as greater value than $(S_0, a)$.

We also consider the example given in [14] in Figure 7.2. In this example, it will potentially take an exponential number of steps in $D$ to reach the reward of 1.0 by repeatedly taking action 1 from the root. Under average reward exploration, it will take an even greater number of exploration trajectories for that reward to overcome the built up average rewards through the tree. Under expectimax-style search, the reward will be immediately propagated back to the root, because the expected values of each state in the trajectory will be updated to reflect the newly discovered expected reward.
7.2 Expectimax Updates in Practice

We implemented expectimax updates for Klondike Solitaire and tested various configurations of our algorithm. In order to compare average reward and expectimax updating on equal grounds, the two methods were tested using the same test set. Our initial experiments without checking for duplicate states generally revealed a slight decrease in performance when using expectimax updating. Further experiments that included duplicate checks showed a significant decrease for expectimax trees, as compared to average reward updating. We present the various results for these various experiments in Table 7.1

While all of the experiments are useful to capture a more complete picture of
<table>
<thead>
<tr>
<th>#traj/tree</th>
<th>#tree/dec width</th>
<th>Avg. Reward</th>
<th>Expectimax</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No Search (99% conf.)</td>
<td>Local Search (99% conf.)</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>21.86(±1.51)</td>
<td>22.32(±1.54)</td>
</tr>
<tr>
<td>100</td>
<td>1 full</td>
<td>23.34(±1.54)</td>
<td>33.56(±2.22)</td>
</tr>
<tr>
<td>100</td>
<td>10 full</td>
<td>31.76(±1.87)</td>
<td>34.27(±2.58)</td>
</tr>
<tr>
<td>50</td>
<td>50 2</td>
<td>36.30(±2.82)</td>
<td>28.34(±1.64)</td>
</tr>
<tr>
<td>50</td>
<td>5 full</td>
<td></td>
<td>33.33(±2.93)</td>
</tr>
<tr>
<td>1000</td>
<td>1 full</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.1: Results for various UCT tests

the performance of the algorithms, we will discuss the first three rows in detail, as they represent a logical progression of UCT tree development. All three utilized 100 trajectories to generate each UCT tree. The first set of experiments generated a single UCT tree and limited the branching factor of each action to 1. Because the branching factor is limited in this way, we would expect similar experiments with and without local search to be statistically equivalent, which they are. We do notice that the average reward updates are performing slightly better than the expectimax updates.

The experiments represented by the second row increased the branching factor to allow a potentially unlimited number of children for each action in the tree. Because only 100 trajectories were used to generate each trajectory, this number is effectively limited to 100 for trees without local search. For those trees which employ local search for repeated states, the number of children can never surpass 21, the number of face down cards at the beginning of each Klondike game. Compared
to the first row, we see some improvement without restrictions on branching in all
categories except the final column, representing expectimax updates in trees with
local search. In this final category, performance is almost cut in half compared to
similar experiments.

The problem arises partially because we are mixing stochastic and deterministic
actions. Suppose that at the beginning of a game, actions are selected randomly
such that we perform n deterministic actions (moving cards down from the deck
or other actions that don’t reveal a face-down card). On the n+1st action, we
finally randomly select an action that reveals a face-down card, which reshuffles
the face-down cards. Suppose this trajectory leads to a win. On the subsequent
expectimax update, the values of all of the actions taken and all of the states
visited in the winning trajectory are updated to 1.0 (value for a win).

Because our UCT exploration policy requires that we explore all actions prior
to re-visiting any action, it may be a long, exponential wait before we revisit the
deterministic action that previously led to a win. When it is revisited, it will find
the previously visited state (because the action is deterministic) and see that one
of the actions has already been taken (which has a value of 1.0), so it will choose
a different action. Even if that action leads to a loss, the value of the state will
still be 1.0 because of the other action, which means the action from the root will
still have a value of 1.0 as well.

Using the standard UCT update will not preserve the value of 1.0 as the reward
for the actions taken in the displayed trajectory. Instead, those actions will be con-
tinually updated each with the rewards received from (presumably poor) decisions
Figure 7.3: An example of expectimax updating in domains with both deterministic and non-deterministic transitions. After following a series of deterministic actions (marked $a_{dt}$), a non-deterministic action leads to a (possibly rare) eventual win. As a result, state values along the entire trajectory are initially updated to a value of 1 and maintained throughout possibly exponential additional exploration until action $a_{nd}$ is taken again from state $S_i$.

leading to losing states. Unfortunately, it will not be clear to the UCT algorithm if a domain is more closely related to Figure 7.3 where Expectimax updates will be harmful or Figure 7.1 where such updates will be helpful.

At this point, this appears to be an artifact of combining expectimax updates with the local searches for duplicates states in domains where stochastic and deterministic actions are equally considered during action selection.

In reviewing the data, we had initially hoped that the expectimax updates had increased the predictive power of each tree, but also increased the (already
high) variance of the generated UCT trees. If this were the case, we might ex-
pect individual UCT trees to be poor classifiers, while ensembles of those same
trees would accurately predict relative action values. This was obviously not the
case, as illustrated in the third row of Table 7.1. These experiments repeated the
previous experiments with an ensemble of 10 UCT trees for each decision. It is
interesting to note that performance is significantly increased for all experiments
except expectimax with local search, which displays no statistically significant im-
provement. The most plausible explanation is that expectimax updates in these
circumstances decreases the predictive power of individual UCT trees.

Because of the demonstrated properties of Expectimax updates, it is not un-
reasonable to expect it to outperform simple UCT-style updates in many classes
of domains. Further work will need to be done to validate these assumptions.
Chapter 8 – Learning Default Behavior

The default behavior for UCT determines the behavior of UCT when it is choosing from among actions previously not taken. Previous work to learn this default behavior through traditional reinforcement learning methods has had mixed results. In particular, attempts to learn default behavior for the game of Go has shown that while TD(\(\lambda\)) can effectively learn a greedy heuristic policy that outperforms other greedy (or random) methods, this same method performs poorly in self play [19]. Recent work has attempted to correct this problem [36].

While the struggles in Go have not specifically been attributed to the complications of learning from self play, we have been able to successfully learn a default policy for Klondike that has demonstrated none of the previously observed complications.

Our initial efforts began with the hand-tuned policy that we had successfully used as a heuristic for our work in Thoughtful Solitaire. From this beginning, we eliminated those features that relied on knowledge of hidden cards and added some new features to cover the space as best as possible. The 371 features are defined over 8 different classes, described in Table 8.1. Ten separate, randomly initialized approximation functions were learned using the TD(\(\lambda\)) learning algorithm on 1 million training examples \(^1\). Of the ten, three outperformed the prioritized action

\(^1\)Trials were run with \(\lambda=0.99\), \(\gamma=0.99\) and the learning rate at 0.01. Weights randomly
rule heuristic as a greedy policy on 100,000 previously unseen games. The one that performed best won close to 17% of games and was retained as the default behavior of choice.

8.1 Linear Function Approximation Features for Klondike

Most of the features used in learning the function approximation for Klondike are easily understood. We chose features that clearly were related to the immediate goal of winning the game. For example, the first feature correlates the value of a card being in a foundation stack with the value of the state. Value is also associated with cards if they can be moved from their current location, either from the deck or from the tableau. Other features are not as closely tied to the direct goals of the game, such as if a card is located in the deck or the tableau. With the exception of Classes 5, 6 and 7, which classify cards as either face up or face down, each of the features, when paired with a specific action can be determined for the state that will result from the action, and can be used to determine the value of taking an action from the current state. For these other three features, the value of the feature is carried over from the original state to the next state.

initialized between 0.01 and 0.99. Reward for winning = 10. Penalty for losing = -1. Step penalty = -0.001
Class 1 (×52) True if card is in foundation
Class 2 (×52) True if card is in the deck and cannot be moved
Class 3 (×52) True if card is in the deck and can be moved to a foundation stack
Class 4 (×52) True if card is in the deck and can be moved to a tableau stack
Class 5 (×52) True if card is face up in a tableau stack
Class 6 (×52) True if card is face down in a tableau stack
Class 7 (×52) True if card is face down in a tableau stack and an action is available that will reveal a card
Class 8 (×7) True if this value (0..6) is the max number of contiguous face down cards in a tableau stack

Table 8.1: A description of the 8 different classifications for the 371 features that define the heuristic value function for Klondike Solitaire.

8.2 Adjusting Behavior of the Learning Algorithm

Early observations of output from the learning algorithm suggested a high variance in the learned weights. First, for seven of the eight feature classes, 52 weights were learned, one for each card in the deck. For these seven classes, there appeared to be little correlation in the learned weights among cards of equivalent rank (e.g. the four Queens in the deck). This raises concerns because, as the suits are isomorphic, we would expect learned weights to be similar across cards of equivalent rank. Secondly, the performance across the ten trials varied wildly, with some learned policies winning fewer than four percent of games (in comparison to a random policy which wins around seven percent).

In an effort to reduce the variance of the learned weights, while respecting as much as possible the learning process, the learned weights were averaged across
equally ranked cards. The seven weights learned for the eighth class were unaffected. This effectively reduced the number of independent weights from 317 to 98.

Additionally, the values of the weights were extremely high (no doubt due to the high values for $\lambda$ and $\gamma$). In order to simplify our own understanding of the weights, all weights were equally divided by 10 until all weights had values below 100. As a linear function, this preserved the relative value of each feature, and did not affect the results of the learning method in any way.

In an effort to make these corrections internal to the learning algorithm, certain steps were taken. The number of weights to be learned was reduced to 98 to accurately reflect the averaging being done across the original 317 learned weights. Values for $\gamma$ and $\lambda$ were tested with combinations of 0.99, 0.95 and 0.90 in order to decrease the need to manually adjust the learned weights. Also, an effort was made to more accurately represent the interactions of the features belonging to Classes 5, 6 and 7. Rather than carrying the weight over from the original state, the likelihood of each feature of these classes were calculated based on each action and the correct proportion of each class weight was included in the estimation of the following state\(^2\). This change eliminates the need for features of Class 7, which feature is now represented as a linear combination of features of Class 5 and Class 6. Tests with these (seemingly logical) changes performed horribly. After 50 separate trials failed to produce a single approximation algorithm that could

\(^2\)e.g. if we know that the 3♣ is face down along with 10 other cards, and we choose an action that will reveal a face down card, we can apply 9/10 of the value of the 3♣ remaining face down, along with 1/10 of the value of the 3♣ being face up
outperform the baseline greedy heuristics, these changes were abandoned, and performance testing continued with the weights learned prior to these amendments.

8.3 Interpretation of Learned Weights

We comment on the general progressions of learned weights for each feature class.

Class 1: Card in Foundation

This class of weights dominates the entire set, with the values increasing as the rank of cards increases.

Class 2: Card in Deck, Unmovable
Surprisingly, having unmovable cards in the deck seems to add value to a state. Intuitively, it would seem reasonable to think that cards being movable from the deck would be more valuable, but these weights typically dominate Classes 3 and 4, which correspond to these same cards being movable from the deck.

**Class 3: Card in Deck, Movable to Foundation**

With the exception of cards with rank of Ace, Two and Four, these values were close to zero, indicating that in general, this is not a very informative feature.

**Class 4: Card in Deck, Movable to Tableau**

Like the previous feature class, many of these weights were very close to zero. The ability to move mid-rank cards (Five, Six and Seven) seem to add value to a state, while higher-rank cards (Nine, Ten and King) detracted from the value of a state.
Figure 8.3: Learned weights for feature Classes 5, 6 and 7 by card rank.

**Class 5, 6: Card Face Up/Down in Tableau**

The values for all 13 ranks of cards in both of these classes were all positive, leaving us to wonder if it's better to have a card face up or down in a Tableau stack. Intuitively, it would seem that cards would universally be better if they were face up, which seems to be true for most ranks except Twos, Threes and Fours, for some reason.

**Class 7: Card Face Down in Tableau with Action to Turn a Card Face Up**

The values for this class were quite erratic and it is difficult to make general conclusions. Because of the large difference in weights between Class 6, one may conclude that there is a significant difference in the fundamental nature of states with and without available actions that will reveal a face down card.
Figure 8.4: Learned weights for feature Class 8 by maximum depth of face down cards.

**Class 8: Greatest Depth of Consecutive Face Down Cards**

As expected, decreasing this metric increases the relative value of a state.

8.3.1 General Conclusions

In Figure 8.5, we see a comparison of the eight feature classes. Class 1 dominates the group with strong positive weights, generally increasing with the rank of the card. The only clear progressions of weights with card ranks correspond to Classes 1 and 8. This is not surprising as the reward received for winning the game can be defined exactly by the features of Class 1, and are closely tied to features in Class 8. No clear progression exists for the other features, which is surprising in some
Figure 8.5: Comparison of learned weights for all feature classes.

sense, as general modes of play require cards to be played in progression according
to their rank, most notably, the building down of sequences of face down cards in
the Tableau. This progression seems to have negligible influence. If so, we may
expect to see a more noticeable difference between Classes 5 and 6, and some sort
of clear progression of values for feature Class 4, both of which are absent from
the set of learned weights.
<table>
<thead>
<tr>
<th></th>
<th>Greedy Performance as a Default Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Win Rate</td>
</tr>
<tr>
<td></td>
<td>(99% conf)</td>
</tr>
<tr>
<td>Random</td>
<td>7.00±0.066</td>
</tr>
<tr>
<td>Priority Heuristic</td>
<td>12.98±0.087</td>
</tr>
<tr>
<td>Learned Policy</td>
<td>16.95±0.097</td>
</tr>
</tbody>
</table>

Table 8.2: A learned default policy is compared to the random policy and to an action-priority heuristic. The policies are compared as 1-step look-ahead greedy policies as well as default behavior within a UCT policy. The greedy tests were performed on 1 million games, while the UCT tests were performed on 9083 games. Identical games were tested for all three policies for both sets of tests. UCT decisions were made with 10 independent UCT trees, each generated by 100 trajectories, without restriction to branching factor.

8.4 Performance of UCT with Improved Default Policy

Unlike our results with Expectimax updates, and unlike similar applications of improved default policy for the game of Go, using an improved default policy dramatically improved our overall performance. Table 8.2 displays performance results for three default policies: a random policy, our action priority heuristic and our policy learned with TD(λ). We compare these policies as greedy heuristics and as default policies within UCT. Each root action decision was made by constructing 10 UCT trees with 100 trajectories for each tree, without restriction to branching factor. 9000 randomly initialized games were used to test the performance of UCT with the different default policies. Improved performance as a greedy policy increased performance of the corresponding UCT ensemble, with over 41% of Klondike games won when the default policy was the learned policy.

These results outperform all of the previously published results. With the bene-
fit of local search, even the random default policy outperforms all other policies for the corresponding time to win each game. In Figure 8.6, we can see a comparison of the performance of the three UCT policies from Table 8.2 and the best policies from Table 5.2, which contains previously published results. When comparing all policies that have a winning rate of at least 30%, the benefit of our recent changes are made clear. Our new results with the learned policy as a default compare favorably to or outperform 8 of the 10 previously reported algorithms for time, and outperform all previously reported policies by more than a 10% increase over the previous best of almost 37% games won. Using the simple action priority heuristic default policy with our other changes increases over the previously published results shown here by both time and percentage of games won.
Figure 8.6: A comparison of previously published UCT results (found in Table 5.2) with more recent results described in Table 8.2. Percentage of games won (on a linear scale) is measured against the time required to finish each game (on a log scale). Thus results on the upper left ridge represent those algorithms that win the most games in the smallest amount of time. It is clear that the yet unpublished improvements which include local search for repeated states and the use of improved default policies have improved our performance significantly.
Chapter 9 – Contributions of This Research

The work defined in this thesis has improved results in a number of significant domains as a result of novel approaches and new algorithms. Our work has helped to demonstrate the shortcomings of current approaches to important problems and introduced new solution techniques for large branches of research.

9.1 Algorithmic Contributions in Ensemble and Sparse UCT Variants

Our work combining Monte-Carlo methods is an expansion of the family of Monte-Carlo style planners including FF-Replan and HOP. The unified method we have proposed will significantly expand the tools available to researchers in probabilistic planning domains. This family of algorithms easily scales to large domains, and has already been shown to be effective in probabilistic domains far beyond the scope of other probabilistic planning algorithms.

This work presents the results of a broad family of algorithms. UCT systematically builds a deep search tree, sampling outcomes at leaf nodes and evaluating the result. HOP solves several sampled determinized problems to approximate the value of root-level actions. Similarly, HOP-UCT and Ensemble-UCT sample several UCT trees, to approximate the same action values. In the case of HOP-UCT,
the trees are determinized by restricting the UCT tree to a single outcome for each action and state. Sparse UCT represents a compromise between a determinized UCT and a full UCT tree. In Klondike Solitaire, the UCT based methods have been shown to be significantly more successful than HOP.

The general algorithm can adjust the sparseness of the results of probabilistic actions, the number of UCT trees in a decision ensemble and the number of trajectories used to construct each tree. We have demonstrated the importance of each of these variables and their relative cost, as measured in the increase of the average time per game. We have also reported the increase in performance, as measured by the percentage of Solitaire games that can be solved. Our work extended theoretical guarantees associated with sparse sampling and UCT to Sparse-UCT that combines and generalizes these methods.

9.2 New Approaches for Probabilistic Planning

At the 2009 International Conference on Automated Planning and Scheduling (ICAPS), held in Thessaloniki, Greece, our work with Klondike Solitaire was presented to an audience of some of the most influential members of the planning and scheduling research community. Our submitted paper won the “Best Student Paper” award for the conference and has resulted in a recent solicitation from the journal *Artificial Intelligence* for a fast-track publication. Our work in this area has significant general implications for this research community and outlined the shortcomings of modern planning solutions. We presented novel solution
techniques that combine UCT, HOP and sparse sampling to solve a significant percentage of Klondike instances.

Much of probabilistic planning is currently focused on artificial domains designed by researchers. Unfortunately they do not bring out some of the representational issues that are readily apparent in natural domains, for example, the problem of having to represent uniform distributions over variable number of objects in Solitaire or the difficulty of representing the dynamics of a robot arm. We hope that encounters with real world domains might encourage researchers to consider novel problem formulations such as planning with inexact models or using simulators in the place of models.

9.3 Klondike and Thoughtful Solitaire

We presented a real-time search algorithm designed for planning tasks with multiple stages. This multistage nested rollout algorithm allows the user to control the amount of search done at each level and specify a distinct heuristic for each level.

We demonstrated that altering the action space can significantly improve search efficiency without compromising the essential play of the game. This was accomplished by creating macro actions that absorbed the *turn-deck* actions.

Our solver for Thoughtful demonstrated that no less than 82% and no more than 91.44% of instances of Klondike Solitaire have winning solutions, leaving
less than 10% of games unresolved. Previous to these results, the percentage of uncounted games was around 29%. To the best of our knowledge our results represent the first non-trivial empirical bounds on the success rate of a policy for Klondike Solitaire. The results show that a number of approaches based on UCT, HOP, and sparse sampling hold promise and solve up to 40% of random games with little domain knowledge. These results more than double current estimates regarding human level performance.

We introduced linear approximation function for Klondike and Thoughtful that included novel features expressing complex relationships in these games. The associated weights (both hand tuned and learned by TD methods) can illustrate important relationships in these games and provide insight to enthusiasts who regularly play these games.

9.4 Fire, Rescue and Emergency Response

We have presented results in the Fire, Rescue and Emergency Response domain, based on a 5 year log of actual emergency requests. We presented SOFER, a two-level random restart hill climbing algorithm that optimizes unit station assignments and in-time response policies. Our simulated results using the SOFER algorithm improve on simulated best-known policy, and are sensitive to preference over measured metrics and population density changes. One surprising conclusion of our results strongly suggests that performance across multiple metrics can be maintained even if one of the six stations is closed. This conclusion is coun-
terintuitive, providing valuable insight into policy structures, especially for city
governments that may be investigating policies that maintain performance while
utilizing reduced resources.
Chapter 10 – Discussion and Future Work

The work presented in this thesis has made significant contributions to a number of current areas of active research. It has raised additional questions that illustrate additional work that can make an immediate impact in these fields.

10.1 Probabilistic Planning

Our work helped introduce Monte Carlo methods to the planning community in general by demonstrating the success of MC methods in the difficult domain of Klondike Solitaire. In doing so, our work illustrated shortcomings of previous planning methods, especially related to complex and real world domains. Future work in this area will a) develop new stochastic planning algorithms by integrating MC methods with planning heuristics and value function learning, b) develop formally modeled benchmarks based on existing complex, real world problems and improve existing planning languages to compactly represent these domains, c) improve worst-case behavior of UCT, d) address the issue of how to provide theoretical guarantees for these MC planning methods and e) improving existing planning languages to accommodate the complexities of domains such as Klondike.
10.2 Bounds for Klondike and Thoughtful Solitaire

The work described in this thesis has improved the bounds in Thoughtful Solitaire and established the first bounds ever for the game of Klondike Solitaire. We believe that significant improvement in these domains is within immediate reach through a) learning improved value functions for the game of Klondike, b) continued exploration that balances the variables of the UCT family of algorithms, c) application of these algorithms to the deterministic game of Thoughtful Solitaire and d) improvement of the dead-end detection algorithm to lower the upper bound of solvable games.

10.3 Fire, Rescue and Emergency Response

Our work in the FRER domain has room for significant improvement. To this point, our work has not yet included a learning method that demonstrated significant improvement within the restricted bounds of the policy space. Possible learning methods that may prove successful include a) UCT methods to simulate future events b) TD learning methods to learn the value of taking actions and c) Approximate Policy Iteration methods that improve upon existing policies.
Bibliography


