Robust Multiagent Plan Generation and Execution with Decision Theoretic Planners

by

William H. Turkett, Jr.

Bachelor of Science
College of Charleston, 1998

Submitted in Partial Fulfillment of the

Requirements for the Degree of Doctor of Philosophy

in the Department of Computer Science and Engineering

College of Engineering and Information Technology

University of South Carolina

2004

Major Professor	Chairman, Examining Committee
Committee Member	Committee Member
Committee Member	Dean of The Graduate School

Acknowledgements

- Thanks to God for giving me the knowledge, ability and strength to complete this research.

- Thanks to my Mom and Dad, Mary and Bill Turkett, and to my sisters, Alison, Ashley, and Laura, for their support and encouragement (in so many ways) on this journey.

- Thanks to my advisor, Dr. John Rose, for his patience, wisdom, and guidance. Thanks also for helping me to keep food on the table and a roof overhead! Finally, thanks for teaching me what it truly means to be a teacher and a researcher.

- Thanks to my thesis committee - Dr. Michael Huhns, Dr. Marco Valtorta, Dr. Jose Vidal, and Dr. James Lynch - for the time they have spent on me and the guidance they have given me. Thanks to Dr. Manton Matthews for everything he does within the Computer Science Department.

- Thanks to Clint Fuchs for many a good lunch and beer and a great friendship. Thanks to Vance Holderfield and Len Bowers for wonderful friendships and for allowing graduate school to be the experience it has.

- Finally, thanks to my officemates and classmates for the many stimulating conversations and tons of support they have provided over the last six years.

Abstract

This thesis details research concerning the development of multiagent systems that can support robust goal achievement in realistic environments. Partially Observable Markov Decision Processes (POMDPs) are the de-facto technique for modeling environments with uncertainty. Traditional approaches to finding optimal policies for POMDPs are unable to scale to problems with more than tens of states. In many POMDP domains there is significant structure which, if exploited, can significantly reduce the computational requirements for generating policies. Run-time knowledge can also be used to further reduce computational requirements during policy generation. The main hypothesis of this research is that the exploitation of structure and runtime knowledge in the use of dynamic decision networks for POMDP planning will scale better than existing POMDP algorithms for policy generation, while still supporting generation of high quality policies. The scalability of algorithms which exploit domain structure and run-time knowledge in POMDPs is demonstrated by comparison to current exact and approximate approaches for finding POMDP solutions. To support social-level goals in a distributed environment, researchers have developed POMDP models encompassing multiagent beliefs and actions. These models are extremely computationally complex. Other popular approaches for supporting social goal achievement either do not fit well within frameworks for reasoning under uncertainty or do not balance individual and social goals well. A second hypothesis of this work is that socially-driven utility functions can be defined within a MDP framework and that agents planning with such a utility function can effectively balance social and individual goals while performing localized decision-theoretic planning. The performance of multiagent systems using a socially-driven utility function is compared

to the performance of systems acting under alternative approaches to multiagent planning and execution.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With progress being made in the research and development of multiagent systems, agent approaches to problem solving are becoming increasingly popular. As research demonstrates that the defining characteristics of agents, such as their autonomous and distributed nature, enable flexible and robust systems, agents are poised to become the architecture of choice for many problems. Artificial intelligence planning is a popular method of agent control that has been put to use to solve real-world problems. NASA has recently demonstrated the use of a constraint-based planner in autonomous spacecraft control (Muscettola et al., 1998) and is actively supporting planning research for rover control (Estlin et al., 2003) and satellite management (Chien et al., 2002).

Realistic environments for intelligent systems of the future will require large numbers of interdependent agents with complex goals. The environments will be both dynamic and partially observable. Agents will also need to interleave decision-making with execution. Classical planning approaches lack the ability to handle one or more of these domain features and are not viable options for these types of environments. Extensions to classical planning, such as decision-theoretic planning and continual planning, have made strides in filling the gaps left by classical approaches. A major

problem that still exists, however, is the inability of planning systems to support large, realistic domains, both in terms of the size of the domain that can be modeled and in supporting the interaction among groups of agents operating in such domains. The aims of this research are two-fold. The first is to propose and evaluate approximate algorithms that can exploit the inherent structure found in many POMDP problems. This should support significant scalability while also still providing high-quality plans. The second aim is to propose and evaluate approaches to agent-control that support effective interaction among groups of agents operating in realistic domains. The approach developed in this work is called E-Plan (Emergent-Planner), a term that describes the potential ability of groups of agents acting under the proposed algorithms to achieve global coherency through interactions at the local level.

Partially Observable Markov Decision Processes are the standard mechanism of choice for representing problems where an agent must reason under uncertainty. POMDPs allow for optimal reasoning under the conditions of both action and state uncertainty. Action uncertainty exists when an agent only knows a probability distribution over the possible outcomes of its actions. State uncertainty exists when the observations about the world available to an agent do not provide the exact world state, but rather indicate a probability distribution over the possible states. It is this state uncertainty that both allows POMDPs to represent realistic problems and causes significant computational complexity. Under the most commonly used problem constraints of fixed discount rewards and an infinite horizon, optimal actions can be found for each possible state for a fully observable domain in polynomial time (Littman et al., 1995b). When state uncertainty is introduced, finding optimal solutions is undecidable for infinite horizon problems and exponential for finite horizon problems (Boutilier et al., 1999b). Exact POMDP solvers have been developed, but they generally have impractical computation requirements on problems that are much

larger than tens of states, constraining such solvers to be useful mainly for research-lab and toy problems.

Dynamic decision networks (DDNs) have recently become popular as a means of encoding POMDP problems (Russell and Norvig, 2002; Forbes et al., 1995). DDNs are a graph-based means of encoding and manipulating POMDPs which support factored encodings of problems. Encodings that allow for factored representations have already shown promise for speeding up reasoning under uncertainty (Boutilier et al., 1999b). The key hypothesis of this research is that the exploitation of structure and runtime knowledge in the use of dynamic decision networks for planning will scale better than existing POMDP approaches while still allowing for high quality solutions to be generated.

Multiagent systems can be defined as groups of agents working together to achieve group-level goals. Some popular mechanisms for integrating planning under uncertainty and multiagent systems rely on large models that cover the beliefs of multiple agents, requiring extensive computation and centralization (Boutilier, 1996; Nair et al., 2003). Other approaches use sociological concepts, such as social laws or emergent conventions, to bring about group behavior (Shoham and Tennenholtz, 1997; Fitoussi and Tennenholtz, 2000). Social laws can be arbitrarily restrictive while social conventions take time to become useful. Neither are not a clean fit for decision-theoretic planning frameworks. The second hypothesis of this work is that socially-driven utility functions can be defined within the Markov Decision Process framework. Agents planning with such a utility function should be able to effectively balance social and individual requirements while performing localized decision-theoretic planning.

An implementation of E-Plan has been developed and used in a set of experiments to evaluate the two hypotheses of this work. Experiments for scalability compared implementations of several structure and run-time knowledge exploiting

algorithms against current exact and approximate methodologies for finding policies for POMDPs. Experiments testing the ability of groups of agents to achieve global coherency compared an implementation of a planner reasoning under a socially-biased utility function to agents reasoning under selfish utility functions and to a centralized approach.

Chapter 2 of this work provides background material on traditional approaches to single-agent and distributed planning. The end of chapter 2 details the problem under evaluation in light of this background material and introduces the proposed solutions. Chapter 3 introduces the set of domains used for algorithm description and experimental validation. Chapter 4 provides an overview of the novel architecture and algorithms developed in this work. Chapter 4 begins with a description of the high level implementation of the E-Plan architecture and implementation-specific details. The next sections then describe the algorithms for exploiting structure and run-time knowledge in POMDP domains. These descriptions are broken into sections on the exploitation of hierarchical structure, exploitation of limited belief states, and exploitation of run-time knowledge. The last section of Chapter 4 describes an approach for developing a socially-biased utility function for multiagent environments. Chapter 5 details design issues and configuration parameters which, though not the main focus of this work, can effect planning scalability and quality. Chapter 6 describes the results of experimental validation of the algorithms presented in Chapter 4. Finally, Chapter 7 reviews the contributions made in this work and discusses new questions that have been raised. Background material related to specific algorithms and not found in Chapter 2 can be found within the appropriate related sections.

Chapter 2

Background

2.1 Classical Planning

The traditional planning problem can be loosely defined as the process of determining which of a set of actions will result in a transition from a given current state to a desired goal state. Classical planning algorithms operate under four basic assumptions which simplify this problem. In a classical algorithm, the planning agent has total knowledge of the world, has total knowledge of how its actions will affect state transitions, is the only actor in the domain, and pursues the same goals over time. With these four assumptions in place, if there is a path from the current state to the goal state, it is possible for the agent to enumerate a list of actions that will allow the agent to achieve the goal state. Since the agent knows the outcome of its actions, by taking those actions and not having any other agents interfere, the agent is guaranteed to eventually enter the goal state. The agent can generate an entire plan first, and then execute that plan (desJardins et al., 1999).

Classical planning algorithms operate by searching through a domain plan space. At its simplest, a plan space can be defined as the application, at each timestep, of all possible actions to the potential current states. This process is repeated, applying all

possible actions to all output states from performing the previous action. Because the number of states grows exponentially over time, the performance of classical planning algorithms is dependent on their ability to effectively prune the plan space.

Graphplan (Blum and Furst, 1997) is a popular example of a traditional planner. It is also one of the fastest due to its use of a plan graph to prune the search space. A plan graph consists of alternating layers of propositions, where odd layers represent the state of the world and even layers represent actions to be taken. Edges in the plan graph from propositions to actions represent preconditions, while those from actions to the next level of propositions represent add and delete effects of actions. To solve a planning problem, the plan graph is iteratively constructed from the domain specification. In each iteration, another action and proposition level is added to the graph and the graph is searched using backward chaining to see if a plan exists that enters the desired goal states from the current state. Part of the power of Graphplan is its ability to recognize and propagate mutual exclusion constraints through the graph, eliminating large numbers of paths that need to be examined.

2.2 Hierarchical Planning

An alternative approach to planning is the use of Hierarchical Task Networks (HTNs). A HTN consists of multiple abstract levels of task structures. At the highest level is the abstract goal an agent is interested in achieving. At lower levels, more concrete refinements of the abstract goals are defined. Refinements are plans for achieving a goal, and the actions that make up the plan are recursively viewed as goals themselves and can be further refined. This process is continued to the leaves of the task structure which are primitive actions the agent can perform. A planning algorithm with HTNs consists of selecting the appropriate method of decomposition for each abstract goal. Typically, there are multiple ways to decompose an action in a HTN, supporting

the ability to select decompositions according to some measure of utility (i.e. one approach may cost less or complete earlier than another).

HTN planning, in its strictest form, only allows action decompositions to generate plans. Alternative implementations incorporate HTNs into classical planning algorithms by allowing decompositions to be a step that can be taken in plan generation along with other plan refinements, such as adding ordering constraints to a plan or adding subgoals to satisfy preconditions. When using hierarchical planning, a previously unseen problem of inconsistency between plan steps arises. By planning with abstract actions, the internal effects generated by a task are not explicitly seen by the planner until that task is further refined. It is thus impossible to detect potential conflicts between decompositions of sequential actions at this abstract level. Accordingly, the development of the task hierarchy should try to avoid decompositions where these types of conflicts could occur (Russell and Norvig, 2002).

Significant work on the complexity of HTN Planning has been performed by Erol (Erol et al., 1994), and HTN planning has been implemented in multiple planning systems. One HTN planner of particular interest is SHOP, the Simple Hierarchical Ordered Planner, developed by Nau *et al.*(Nau et al., 1999). SHOP uses ordered task decomposition to perform hierarchichal planning. The essence of this approach is that tasks are decomposed in the same order they will be executed. By using ordered task decomposition, many of the problems seen in other HTN planners caused by interacting tasks are removed. Additionally, this approach allows the planner to know the expected state of the world when it decomposes a task, as everything up to the task has been planned for. Nau notes that this provides for a significant amount of expressivity, allowing for very efficient domain encodings.

2.3 Decision-Theoretic Planning

For most real world problems, the assumptions made by classical planners do not hold. In particular, actions are generally not deterministic. Instead, there is a probability distribution over the possible outcomes of an action. Agents also may not be able to perfectly sense the state of the world, leading to beliefs over a distribution of world states. A number of decision-theoretic planning techniques have been developed to support problem solving in these types of domains. An overview of these techniques can be found in Blythe's review of decision theoretic planning (Blythe, 1999).

One such approach is to integrate probabilities into classical planning algorithms. PGraphPlan and TGraphPlan (Blum and Langford, 1999) are two extensions of the original Graphplan algorithm (Blum and Furst, 1997) that include probabilistic actions. PGraphPlan performs forward chaining through a plan graph to find an optimal contingent plan for a finite number of steps, guided by heuristics on which nodes in the graph contribute to reaching goal states and how much each node contributes to solving the problem. TGraphPlan performs backward chaining through a plan graph to select a trajectory with the highest probability of achieving goal states.

C-Buridan (Draper et al., 1993) produces plans that will reach goal states with a probability above an input threshold if such a plan exists. Plans can contain sensory actions to gather information about the state of the world and contingent plan branches to be followed dependent on the outcome of these sensory actions. C-Buridan is also capable of handling imperfect sensing actions (while PGraphPlan and TGraphPlan require a fully-observable world). Plans are generated through a series of assessment and refinement steps. In an assessment step, the current plan is evaluated to determine if it reaches the goal state with a probability above the given threshold. If it does, the current plan is returned. Otherwise, a refinement is performed on the current plan, where a refinement can be either the addition of a new subgoal link or

the resolution of threats to a subgoal.

An alternative approach to planning under uncertainty is to pose the planning problem as a Markov Decision Problem (MDP). This is different from the previously mentioned planners in that the planning process is viewed as selection of a set of actions of high expected utility rather than as generation of ordered steps that will satisfy domain propositions after execution. Boutilier, Dean, and Hanks provide a concise definition of MDPs as the process of "control of a stochastic dynamic system" (Boutilier et al., 1999b). The world can be viewed as being in one of a number of well-defined states at any instance. Events trigger transitions between states and agents perform actions, events that move the agent through the state space. A more formal definition (Boutilier, 1996) says a system can be modeled by a finite set of states S and a finite set of actions A. Transitions between states are probabilistic and modeled by a transition function *Pr(s,a,t)* which yields the probability of entering state t when the agent is in state s and action a is performed. To represent the goals of an agent, there exists a reward function, *R(s)*, which returns the utility of being in state s. This reward function can be extended to incorporate the cost of performing an action and may also be discounted over time. An MDP also typically has a horizon, defined as the number of steps an agent will take during execution (an infinite horizon represents an agent that acts forever). A standard MDP assumes full-observability while partially observable MDPs (POMDPs) relax this assumption.

The solution to an MDP is a policy Π: $S \rightarrow A$. Depending on the current state an agent is in, the policy returns what action should be taken. To compare policies, it is generally useful to compare the expected total reward obtained by following the policy in each given state. A common reward function used is expected total discount reward, where β ($0 < \beta \leq 1$) represents a discounting factor:

$$V_{\pi}(s) = R(s) + (\beta \sum Pr(s, \pi(s), t) * V_{\pi}(t)); \forall t \in \mathbf{S}$$

This reward is based on the reward for the current state and the sum of rewards for future states, weighted by the probability of entering those states. Using this measure of optimality, an optimal policy Π is one where the value of the policy Π at any state s is greater than or equal to the value of any other policy Π' at state s for all s in S and all policies Π'.

Standard algorithms for generating optimal policies are based on dynamic programming techniques and include value-iteration (Bellman, 1957) and policy-iteration (Howard, 1960). Value iteration for MDPs attempts to determine the value function for a specific planning horizon by determining the value function for all horizons from 1 to the specified horizon. For a horizon of length 1, future effects do not need to be taken into account, so simply selecting the action with the highest reward in each state provides the policy and value function. For a horizon of length 2, the reward for being in a state is already known from the first iteration. Thus it suffices to add the effect of selecting each action in a state to the current state value. The reward for entering a new state is weighted by the probability of entering that state given the current action executed and the current state. This calculation is repeated for each state represented in the MDP in each iteration of the algorithm. Value iteration completes when two consecutive value functions are minimally different in value (Cassandra, 1999b).

The policy iteration algorithm begins by selecting a random policy for the MDP. The value of the policy is then determined by calculating (through solving a set of linear equations) the reward for each state that will be accumulated by following the selected policy. This random policy is then improved at each state. If selecting another action in a state improves the value for the state, it also improves the overall value for the policy and the policy is updated by selecting the alternative action for that state. Policy iteration continues until no further state updates can be made

(Kaelbling et al., 1996).

Solving POMDPs (Cassandra, 1999b) is a much harder problem as knowledge of the current state of the world is not known. Instead, this information is represented as a probability distribution over the possible states of the world. A given probability distribution is called a belief state, and the set of possible distributions is the belief space. Given a current belief state, an action that has just been executed, and the corresponding observation, one can determine the next belief state. Thus, the problem of updating belief states is still Markovian in POMDPs. However, if one is planning, an actual observation cannot be made. Instead, after selection of an action, a probability distribution over observations is given. This probability distribution over observations forces a probability distribution over the outcome states. Since the problem of updating belief states is Markovian given the current state, action, and observation, one approach to solving the POMDP problem is to cast it as the simpler MDP problem, but defined over a continuous belief space. This MDP can then be solved by a modified value iteration algorithm. Instead of representing the value function as a table of states ands values, the value function is now represented as a piecewise, linear convex function over the entire belief space. Another modification to the original value iteration algorithm is that it is no longer possible to loop over all possible next states in calculating expected rewards as the set of next states is infinite with a continuous belief space. Since the value function is piecewise, linear convex, it can be represented as a set of vectors called alpha-vectors. The value of a belief state is the maximum dot product of the vector that represents the probabilities over states and one of the alpha-vectors that makes up the value function. For a horizon of length one, the alpha-vectors are simply the set of vectors that describe the immediate rewards for performing an action in a belief state (which in turn depends on the rewards for individual states and the probability of being in each individual

state). For a horizon of length 2, the problem becomes how to generate the next set of alpha-vectors, V', from the current alpha-vector set V. Given that an action has been selected, the reward for performing a next action is dependent on the observation that is seen. A strategy is defined as a mapping of observations to actions, so if a certain observation is seen, the action defined by the strategy is the best one to take. Across all belief points, many different strategies may be appropriate. These strategies define another set of piecewise linear convex alpha-vectors that are the value function for horizon two, given a specific action is taken in the first step. Since the value function must be defined over all possible first actions, this process of defining strategies is repeated for each possible first action. The resulting vectors are combined to generate the actual horizon two value function over the entire belief space. Often times, a subset of the alpha-vectors can be discarded as they are completely dominated by other alpha-vectors and would never generate the maximum dot-product. This entire process is repeated for increasingly longer horizons. Since all possible belief points cannot be enumerated, several different algorithms (Littman, 1994; Zhang and Liu, 1996) have been developed which determine regions over which different vectors apply in the value function.

The worst case computational complexity for determining a policy for a discounted, infinite-horizon MDP is $O(|S|^2|A| + |S|^3)$ for each iteration of value or policy iteration. The complexity of finding a maximizing policy for total discounted finite horizon reward POMDPs is exponential in the number of states and the horizon. For an infinite horizon, the problem is undecidable (Boutilier et al., 1999b).

When an agent is uncertain about its next action to perform, it also may be useful to gain more information before making a decision. The usefulness of gathering such extra information can be explicitly calculated through a value-of-information (VOI) function. The computation of VOI is critical in POMDP domains. The value of

perfect information can be calculated as the value received from executing the best action given the new piece of evidence versus the value of choosing the best action only using the current evidence. For new information to be worth obtaining, the value of this new information should be less than its cost and the agent must be open to allowing new information to change its decision. The value of obtaining more information also has the property of being expectedly non-negative (some benefit is expected to be gained through collection of information, as an agent now has a broader view of the world), the property of being non-additive if multiple pieces of information are obtained (there may be some overlap in two pieces of information gathered), and the property that the value of multiple pieces of information is order dependent (how useful a piece of information is to you depends on what information you already have) (Russell and Norvig, 2002).

The number of states is particularly influential in determining the complexity of MDP and POMDP problems. Because of this, a large body of work has centered on methods for factoring or abstracting states to ensure smaller representations. Instead of explicitly describing states, it is often easier to describe a system with a set of features. The entire state space can be generated from the cross-product of all feature vectors and use of features provides for simplification in representation of the key parts of an MDP, such as rewards and actions. As an example, actions can be specified as having effects on particular features rather than enumerating their effects over all possible states (Boutilier et al., 1999b).

Bayesian networks are a datastructure that allow for factored, variable-oriented representations of probability distributions. In a Bayesian network, each node represents a random variable and edges between nodes represent directed probabilistic dependencies between the variables (and implicitly, independencies). The probability of each state for a variable must be defined conditionally for all possible combinations

of the states of its parent variables. For those variables without parents, unconditional probabilities must be provided for each state of the variable. Given a Bayesian network, evidence can be entered on the random variables. The structure of the graph is then used in propagating evidence through the variables and determining the probability of being in various states (Boutilier et al., 1999b).

The standard Bayesian network can also be extended to an influence diagram, a structure that incorporates decision and utility nodes along with random variables. Decision nodes represent actions that can be taken, while utility nodes determine rewards over variable states. Influence diagrams are particularly well suited to partially observable environments. Edges between variable nodes and decision nodes represent the variables that are observable when a decision is being made, while the ability to work with probability distributions over variable states allows representation of uncertain belief in a variable. If evidence is entered into an influence diagram, the expected utility of performing each action in the decision node can be calculated and used to determine the highest expected utility actions to perform (Boutilier et al., 1999b).

Temporal Bayesian Networks represent progressive states of a system over distinct timeslices. The initial timeslice represents the initial state of the system. Edges between timeslices represent variables that are dependent (at the minimum, there should be edges between equivalent nodes in each timeslice) and the conditional probabilities are the transitional probabilities between timeslices (Boutilier et al., 1999b).

An additional factoring of Bayesian networks can be performed if the conditional probability table has sufficient structure. This table can often be mapped to a simple set of rules which are compactly represented by decision trees and decision graphs (Boutilier et al., 1999b).

Value functions, which represent rewards for states and costs for actions, can also

be represented in compact forms. Rewards are generally dependent on a small set of feature variables. In influence diagrams, a value node can be associated with just the variables that contribute to the reward. Furthermore, if there is regularity in the structure of rewards for the value node, the rewards can be represented concisely by datastructures such as decision trees or sets of logical rules. Multi-attribute utility theory provides another means of factoring rewards through definition of independent value functions. If each contributor to the reward function is independent of other contributors, each reward contribution can be represented individually, preventing the exponential blowup of defining rewards for all combinations of features. A simple function is then used to define the overall reward from the individual contributions (Boutilier et al., 1999b).

Beyond factored representations of actions and rewards, there are also a number of algorithms that can be exploited to make decision-theoretic planning more feasible. In classical planning goal regression, the planner starts with an initial set of goals. From these goals, it determines which action is applicable that satisfies one or more of these goals and which does not delete a previous effect or conflict with rest of the goal set. This action is added to the plan and the satisfied goals are removed from the goal set and the preconditions for the selected action are added as subgoals. This process is iterated until the initial state satisfies the set of subgoals, meaning a plan has been generated, or there are no further applicable actions, which means that backtracking must be performed.

A similar technique can be applied in decision-theoretic approaches. In decision-theoretic regression, the states are initially clustered according to the reward function, generating sets of states that have the same value. A new set of states which are equivalent in expected value is calculated for successive horizons. This is done by determining which states represent the same value and which transitions occur to

those states with the same probability (Boutilier et al., 1999b; Boutilier, 1997). The number of equivalence states generated should be smaller than the original number of states, reducing the complexity of the value function.

Another approach is to abstract the problem by disregarding variables that have little effect on the reward. This may not generate an optimal plan for the problem, but this plan can be generated quickly and be used to guide and constrain policy search on the actual problem (Dearden and Boutilier, 1997).

A third technique for speeding up decision-theoretic planning is to use reachability analysis to determine from which areas one can move toward high utility states and which areas are recurrent, staying in the same area. This can produce a serial decomposition of the plan space in which the recurrent states can be solved, and then that information used in solving for the rest of the states. The effectiveness of this approach depends on high independence between different subproblems (Lin and Dean, 1995; Boutilier et al., 1999b).

A final alternative approach to reducing the complexity of solving an MDP is to generate and solve in parallel several sub-MDPs (Boutilier et al., 1997). A global action A is run on each of the sub-MDPs, and if it is a useful action in each sub-MDP, it is deemed useful for the global MDP. The global state space is usually the cross-product of the state space of the sub-MDPs and decompositions are usually along the lines of independent rewards. The value functions for each sub-MDP can be integrated to approximate a value function for the global problem.

Dynamic decision networks (DDNs) are an extension of influence diagrams which can be used to implement utility based planning. Dynamic decision networks extend general influence diagrams over a finite planning horizon and can represent both partially observable states and probabilistic actions (Russell and Norvig, 2002). A general influence diagram predicts the utility of performing one of a set of actions

given current evidence entered in chance nodes. In a DDN, the influence diagram is extended to multiple time slices. The state of the agent at each time slice affects the state at the next time slice. A decision node also exists at each time slice to represent the action to be performed in that slice. Utilities can be represented in each time slice if the reward is time-separable or at the end of the sequence of actions. By selecting an action in the initial time step, the predicted states resulting from performance of that action can then be used in evaluating the actions available in the next time step (Russell and Norvig, 2002; Forbes et al., 1995).

Utility functions in a dynamic decision network need to support goal-directed behavior to effectively be used in plan generation. This problem is not trivial, as classical goal-based planning and decision theoretic planning are not explictly equivalent in semantics (Haddaway and Hanks, 1993). A classical planner examines orderings of actions and their preconditions and effects to ensure goal constraints are met after execution. Decision theoretic planning, however, is more suited to determining the best action to pursue when the world is probabilistic and uncertain. Haddaway and Hanks (Haddaway and Hanks, 1993) state that utility functions for decision-theoretic goal-directed agents should be able to represent partial satisfaction of temporal and atemporal goal constraints. By including partial satisfaction, it is possible to evaluate trading off between multiple goals and pre-empting actions, while incorporation of temporal constraints allows for deadlines and maintenance goals.

Recent research is beginning to show the potential benefits of exploiting structure in generating POMDP policies. One approach to exploiting structure is to consider abstract actions as aggregates of lower-level agent actions. Reasoning over abstract actions can often dramatically reduce the size of the action set and relevant state and observation sets that need to be reasoned over. Two significant developments in this field are Pineau's Hierarchical POMDP algorithm (Pineau and Thrun, 2002) and

Theocharous' Hierarchical Hidden Markov Model algorithm (Theocharous, 2002).

Pineau's work on Policy Contingent Abstraction (PolCA) (Pineau and Thrun, 2002) provides an algorithm for finding approximate solutions to POMDPs through action abstraction. In this approach, a POMDP is represented by a hierarchy of smaller POMDPs. At each level, the action space can be made up of primitive and abstract actions, where each abstract action is represented by another POMDP. A policy for the overall top-level problem is then built bottom-up by finding policies for each smaller problem. Individual policies are generated through exact solution methods, such as pomdp-solve (Cassandra, 1999a), where transition and reward functions for an abstract action are defined by the transition and reward functions of the optimal actions defined at the lower level. Essentially, this process determines a policy for implementing each of the abstract actions. Along with reduced action spaces, reductions in the state and observation space are seen. For example, a taxi navigation task only requires information about the present location of a taxi and its destination, but doesn't need knowledge of whether a rider is present in the taxi or is waiting to be picked up. The location of the rider, which would need to be considered in a flat POMDP solution algorithm for a pickup-and-deliver problem, can be abstracted out in the hierarchically-based subtask for navigation. In PolCA, state and observation abstraction for higher-level tasks is delayed until the lower level problems have been solved. This allows a significant level of minimization of the state space at each level. By abstracting tasks, however, it is possible that rewards will have to be introduced into a local problem that aren't present in the real problem. Although this appears as extra work, these quasi-rewards can be seen as a way of adding additional information to the problem that guides the planner more efficiently.

During execution, an agent determines what action to take given its current belief state, a probability distribution over the actual states of the world. To select an

action, the policy tree is traversed down a path from the root to an appropriate leaf. If a primitive action is selected, it is executed. On the other hand, if an abstract action is suggested as optimal, its policy is recursively queried. This approach, by examining policies down the tree during execution, removes the need to be able to determine if a subtask is completed, a problem that is often difficult when working in POMDP domains as knowledge of the current state is often unclear to the current agent.

Theocharous (Theocharous, 2002) has studied the case of modeling POMDPs with Hierarchical Hidden Markov Models. A Hidden Markov Model (HMM) consists of a set of states S, a transition function T that maps from a state to another state, a set of observations Z, an observation transition function O, which maps from states to observations, and π, an initial probability distribution. Only observations are available to the agent, and the agent must infer the state of the world from the observations. Connections between HMMs and Bayesian networks and between HMMs and POMDPs can clearly be seen from this definition. Hierarchical Hidden Markov Models (HHMM) are HMMs where the hidden true states of the world can also be probabilistic processes. An HHMM can be represented by a tree, where the leaf nodes are product states, generating observations to the agent, and internal nodes either represent hidden states of the model or end-states that force the model to return from a given abstract state. An internal non-end state of a model has links to a horizontal transition matrix, which describes the probability of transitions among the state's children, and a vertical transition vector, which describes the likelihood of a child being activated.

This model is expanded to include actions and reward functions to represent a Hierarchical POMDP. Theocharous' HPOMDP model extends HHMMs by formalizing the idea of an abstract state, which can consist of product (observation-generating)

states, other abstract states, end-states to allow returns from the abstract state, and entry states that cause a child to be activated. Additionally, a set of actions, a transition function defined over exit states and actions, and a reward function over product states are modeled. Theocharous describes how learning POMDP models with this representation can be achieved by an extended Baum-Welch algorithm. A description of planning and execution with HHMMs is then given. Given the macro-actions available for abstract states, the top-level reward and transition functions are calculated in a bottom up fashion (much as in Pineau's work). A set of equations, derived from the Bellman equation, are used to calculate these functions. During execution, probability distributions are calculated for each abstract level in the model. This is achieved by maintaining a global belief state over product states, updated by observations and primitive actions, and then for any abstract state, computing the belief state by summing over the beliefs for all children of the state. Given this belief state, an action is chosen by one of two strategies, Hierarchical Most Likely State (HMLS), or Hierarchical QMDP (HQMDP).

At each level in HMLS, a set of most likely states is chosen (all within some bound of each other in likelihood). Most likely state sets are recursively found until a product state is encountered. Given a most likely state set, the next most likely state set is found by randomly choosing one of the states from the set and moving down a level in the hierarchy. Once a product state is chosen (by chance), the most likely state from the children of the parent of the product state is chosen. This state may be a product state, wherein the primitive action defined by the policy of the parent given this state is executed. Otherwise, if the most likely state is abstract, the most likely entry state is calculated for that abstract state. This is akin to the agent believing it has just entered an abstract state via one of multiple entry states. A macro-action to be executed is determined from the parents policy for that entry

state. The most likely child of this abstract state is then calculated, and the process repeated until a product state is found. After a primitive action is executed, an observation is received and the belief state can be updated accordingly. The agent then re-determines the set of likely states it is in for each level of the hierarchy. If the current state's parent is no longer in the set of most likely states, the agent returns control to the level above the current state, another state is chosen from the new most likely state set (this new state replaces the node that has been referred to as the parent as the controlling state), and the process continues. Otherwise, the new most likely state from the current parent is selected, and the process continues.

The HQMDP approach is an alternative execution method that attempts to determine the Q value for state-action pairs and then acts based on the maximal Q value. The agent selects its action to execute and supposed state S as the set that maximizes the Q-function. This action is continually executed until another state-action pair becomes maximal. In essence, this approach looks over all possible rewards by multiplying the reward obtained by any possible action for a given product state with the probability of being in that state.

Another structure-based approach for approximating POMDP solutions under reduced time constraints is to apply grid based techniques. One of the most recent algorithms of this type is Pineau's PBVI (Pineau et al., 2003). Pineau's work modifies the exact value iteration algorithm for POMDPs to perform value iteration on a constrained, finite set of reasonable belief points. PBVI computes an alpha-vector, a hyperplane which defines the value function over part of the belief space, for each of the current belief points maintained by the algorithm.

The initial belief state for an agent for a given problem is set as the initial belief point. Given this first initial point, the initial alpha-vectors representing the value function are generated through N rounds of modified value iteration, where N is the

horizon of planning required. Since only a finite set of belief points is maintained, an efficient backup operation is usable which runs in polynomial time and does not require significant pruning of alpha-vectors (a costly step used in exact POMDP solvers to remove alpha-vectors which are unused because another alpha-vector is always maximal when computing expected values). After a round of backups, the set of belief points is expanded to form a new set of belief points that are highly likely to be reached from the initial set. For each current belief point in the belief set, new belief points are generated by simulating the execution of all possible actions. This provides a new belief point set, for each initial point, of size equal to the number of actions. Belief points already in in the initial set are removed, and the distance from the current belief to each of the newly generated points is computed. The farthest belief point from a given initial point is then added to the initial belief point set. By taking the most distant points as new belief points, PBVI attempts to cover as much of the reachable belief space as possible when computing its policy.

Pineau's work is a variant of grid-based approaches to solving POMDPs, where the core idea is to define both an appropriate grid over the belief space for computing value functions and an appropriate metric for computing the nearest neighbor point on the grid to use for action selection. Other grid based work includes that of Hauskrecht (Hauskrecht, 2000) and Bonet (Bonet, 2002).

Past work has also examined the use of dynamic construction of Bayesian networks to allow for effective representation of problems (Provan, 1993; Nicholson and Russell, 1993; Nicholson and Brady, 1994; Ngo et al., 1997). This work has primarily been directed towards dynamic construction from a knowledge base in response to a query presented by the user. Provan (Provan, 1993) states that, due to the complexity of inference in temporal networks, dynamic construction is key to being able to perform inference under reasonable time constraints. Provan's work is directed

towards construction of temporal influence diagrams for use in medical applications. He describes heuristic techniques such as temporal modeling of just a subset of the variables (such as those driving the network or just the observation variables) and metrics for evaluating the effects of the loss of information from such pruning. Nicholson and Brady (Nicholson and Brady, 1994) describe an exact technique for removing unneeded states from a network. Removing states is possible when the probability of being in a given state is set to zero. This state no longer needs to be modeled and can be removed from a given node. When such a state is removed, states in the successor nodes can also be removed if they are no longer possible given the states with probability greater than zero.

2.4 Distributed Planning and Execution

It has been noted by many in the agents field that agents maximizing local utility are not guaranteed to bring about improvements or achieve goals at the higher social level. Jensen and Lesser describe these cases as an instance of a *social pathology*, "a system behavior in which two or more agents interact such that improvements in local performance do not improve system performance" (Jensen and Lesser, 2002). A prime example of such behavior is the problem of the Tragedy of the Commons (Jensen and Lesser, 2002; Turner, 1993). This problem describes systems that, when an agent attempts to improve its local use of a shared resource to increase local utility, result in a significant decrease in utility across all agents. The most common example is that of cow herds on a shared field, where addition of extra cows onto the field from one herdsmen begins to deteriorate the field as a whole. This then leads to problems for all cows on the field. Oftentimes the process is self-propagating - other herdsmen need to move more cows onto the field to make up for their loss in individual utility, propagating and quickening the deterioration.

These problems suggest that, in cooperative multiagent environments, to ensure success at the mission or team level, it may be required that agents do not always try to optimize at the individual level. Rather, an agent should be willing to "take one for the team", selecting a potentially sub-optimal individual action to ensure that mission goals are not harmed. In fact, this idea is mentioned by Turner (Turner, 1993) as a means of staving off the Tragedy of the Commons.

There are three major approaches taken for handling distributed planning and execution among multiple planning agents. The first approach, Cooperative Distributed Planning (CDP), takes the individual planning process and distributes it among a subset of an agent society such that the generation and execution of a plan requires the interaction of several specialized agents. The second approach, Negotiated Distributed Planning (NDP), is based on the concept that agents should primarily be focused on the individual, but can use plans as a means of coordinating actions. Generally, CDP is focused on generating an optimal global plan, while NDP is focused on generating optimal local plans that work well together. To this effect, CDP algorithms typically consist of agents with homogeneous plan representations and a large amount of overhead in broadcasting and refining plans, while NDP agents are typically more heterogeneous and transmit information to other agents so that an individual's preferences can be met. At its limits, CDP can be seen as pure parallelism of the planning process, while NDP is purely conflict determination - finding out where agents are unable to achieve their own local goals because of what other agents have done in the system. Most algorithms developed are not at these extremes, however, and tend to incorporate the useful features of both approaches (desJardins et al., 1999). The third approach is to use an algorithm that develops a plan centrally by taking into account the global world state. The developed policies are then distributed to individual agents for execution. In this sense, the third approach is

similar to CDP in that the goal is to develop an optimal global plan. By centralizing the planning process, these approaches are computationally complex.

Durfee's Partial Global Planning (PGP) (Durfee, 1999) system is a NDP approach which takes advantage of continual planning. This planning framework is primarily focused on how agents can attain global coherence while planning at the local level. An assumption of the PGP system is that tasks are already well decomposed to agents. No agent has knowledge of the global goals or tasks, and agents may also have little or no knowledge of the tasks that other agents are planning for or interdependencies between tasks. Durfee states that local agents should develop plans that contain contingent actions dependent on the results of others and external updates to the environment. Agents should also be able to view plans at various levels of abstraction. By abstracting actions, an agent can commit to performing certain tasks, while not being forced to select the actual method in which the tasks will be performed. Action abstraction also facilitates the sharing of plans between agents. In PGP, agents, using knowledge of the world and other agents, determine which local plan steps will be most useful to others in accomplishing tasks. Communication is guided by a meta-level organization structure that details which agents are interested in the plans of a specific agent and which agents can influence a specific agent's plans.

The most interesting aspect of PGP is how agents reason over the dependencies between their actions. Agents attempt to identify partial global goals by examining their tasks and the abstract plans provided by other agents. Through this evaluation, an agent may determine that its task and that of another agent are both working towards the same global goal. If so, the agent recognizing the global goal evaluates its planned actions and those of other agents working towards the same goal to determine ways to improve coordination in their activities. In PGP, these improvements are mainly directed towards preventing redundant actions and finding orderings of actions

that facilitate the work of others. After generating new local plan steps for each of the agents working towards a global goal, the same agent will also evaluate which agents need to share results of their tasks and plan for communication acts. After the partial global plan has been developed and communicative acts planned, these plans are passed back to the individual agents. If an agent's partial global plan is modified, it translates the modifications to its abstract local plan. This abstract local plan then influences the actions an agent selects to perform. Local plans may also be modified if there are significant changes in the environment or unexpected outcomes of actions. If local plans are significantly modified, these changes should be propagated to other agents that are working towards the same global goals. In PGP, a threshold can be set by the system designer for the level of temporal modification allowed before communication and recoordination is required. A threshold that is too low may hinder an agent system as agents are continuously updating each other with modifications that aren't important, while a threshold too high hinders global coordination as agents may have false views of the state of execution of other agents. PGP also enables task balancing between agents through sharing of plans. An overburdened agent can propose a partial global plan to an under-burdened agent which distributes the work among the two agents. The under-burdened agent may accept, thereby reducing the work load of the original agent, or reply with a proposed partial global plan of its own.

A generalized version of PGP, GPGP, has been extensively developed by Victor Lesser (Lesser et al., 2002). In GPGP, agents are executing to achieve multiple subgoals that together make up group high-level goals. High-level goals have differing priorities and deadlines which filter down to lower level subgoals. Agents coordinate over local actions to obtain optimum performance of the high-level goals.

Within each agent, goal structures are represented by an AND/OR goal tree (very

similar to a task network description of goals). In the goal tree, a high-level goal is repeatedly decomposed into alternative subgoals, any of which can satisfy the ancestor goal. This continues until primitive actions are reached at the leaves of the tree. This tree also represents constraints between subgoals, such as when one subgoal needs to be accomplished before another can be executed or when one subgoal can improve performance of a second subgoal.

Agents that are working within the same region of a goal tree have interacting subgoals and should coordinate over their actions. This is done through evaluation of the agents goal and task structures, based primarily on the methods defined in PGP. By determining current success in achieving goals (such as percent satisfaction or expected time to completion) and inter-task (enable, disable, hinder, facilitate) and task-resource relationships (produce, consume, limits), agents decide on the orderings of actions to be performed. These orderings are integrated with the agents local scheduling mechanisms, determining which of multiple goals to satisfy and the proper approach to satisfy such goals. This approach, local optimization balanced by knowledge of interactions between agents, has been shown to perform well in a wide range of domains, particularly those which require bottom-up coordination.

Recent extensions to GPGP have increased its suitability for domains that require top-down coordination. Top-down coordination is used when agents request other agents to generate certain results. An agent may have the ability to generate a result or accomplish a task, but does not find it appropriate to pursue such actions unless requested. Two of the major extensions to GPGP to support top-down coordination are determination of when to break commitments and the ability to perform contracting. Agents can reason over when a commitment should be broken by using task structures and knowledge from other agents. This is done by by evaluating the potential that a commited action will fail, the potential for a task to become infeasible

to perform, and the potential that a task will no longer be desirable to be performed in terms of utility.

Contracting is implemented through negotiation between agents and the concept of virtual tasks. An agent uses virtual tasks to represent tasks that could be performed by other agents in its local schedule. By assuming the successful accomplishment of this task by another agent, an agent can continue with its own local scheduling. To actually fulfill the virtual task, the agent generates a task structure which contains constraints on performance of the action, such as the required quality of result, start-time, resource costs, and duration. This task structure is sent to an agent with the required capabilities. This contacted agent then reasons locally if it is interested in performing the task and can perform it in light of the requested constraints. A series of negotiations between the agents is used to form a final agreement on how the task is to be performed. The current negotiation approach (Zhang et al., 2000) consists of numerous proposals sent back and forth between the two agents. Each proposal is evaluated locally in terms of the marginal utility gain and cost over multiple attributes an agent would receive from accepting the proposal. Given this utility, the agent responds with a counter-proposal or acceptance as appropriate. Some initial work (Xuan and Lesser, 2002) is also being performed on using MDPs local to each agent to handle uncertainties.

Earlier work by Lesser (Lesser, 1991) on functionally accurate, cooperative (FA/C) agent societies also focuses on the problem of distributed problem solving and planning and is the theoretical basis for much recent work (such as PGP and GPGP). Lesser states that for a large set of problems, the traditional agent solution of decomposing the problem into independent tasks that can be solved with as little interaction as possible is not viable. Due to the nature of such problems, there are no appropriate divisions onto agent skills or resources that allow for truly independent work. Lesser

proposes that agents should work together according to the following model:

1. Each agent works on large grained subproblems
2. Agents should be able to generate partial results, even if these results are uncertain or the agent is missing information
3. Agents should communicate partial results to other agents asynchronously as the partial results are developed
4. Agents should use partial results from other agents to help resolve their own uncertainties and guide individual solutions

Following these steps, agents can converge on a correct solution, even in the face of incorrect partial solutions. This convergence stems from the combination of local control, demonstrated through individual agent actions, and cooperative control, brought on through the interaction of agents and through defining organizational structures. Cooperative control mechanisms act to constrain which approaches are taken, deciding when approaches should be terminated and determining which subgoals are of the highest priority, while local control is focused on generating solutions and sharing information and partial solutions among agents. This style of planning allows for reduced communication compared to systems where agents are telling other agents everything they are doing, fewer idle agents as agents do not have to synchronize before performing each action, and higher levels of parallelism in problem solving and plan execution.

Local problem solving in FA/C agents relies on optimizing partial solutions using currently available information and being able to re-evaluate a solution when new information is presented. In order to successfully merge local problem solving results into the global state, agents should rely on constraints inherent in received partial solutions when developing local solutions, ensuring results generated by multiple agents

are compatible. Agents should also have an explicit understanding of the uncertainty involved with their problem and be able to handle that uncertainty, including maintaining alternative solutions paths and applying heuristics to account for missing information. In order for cooperative control to work effectively, local control must be able to provide the cooperative control mechanism with up-to-date information on the agent's current state, current results, and potential future results; must be able to reorganize in order to better align itself with the global problem-solving state; and must be complete enough to recover from poor global decisions.

Significant work on distributed decision-theoretic planning has recently been introduced. Boutilier (Boutilier, 1996) poses the fully-observable distributed planning problem as a multiagent Markov Decision Process. In an MMDP, there exist multiple heterogeneous agents, each with different capabilities and resources, and a goal that needs to be accomplished. Agents are assumed to be fully cooperative in that all agents have the same utility valuation of any given system state. A probabilistic transition function maps the current system state and a joint action (a combination of all individual actions) to a next system state, while a joint reward function, defined over the set of system states, provides the system reward. The goal of solving an MMDP is to specify local behaviors that will generate the maximum joint reward in the system. Individual policies can be described as mapping from states to actions, while the joint policy maps the current system state to joint actions (or distributions over joint actions if policies are randomized instead of deterministic). Ideally, agents will adopt the same optimal joint policy (optimal joint policies are not guaranteed to be unique). However, without a central coordination mechanism, agents must develop policies locally.

Boutilier states that this problem is akin to the problem of equilibrium selection in an n-person game. To facilitate finding a solution, an MMDP assumes that all agents

can compute the optimal value function for the joint policy. Given this, one can determine which joint actions are optimal for any state, producing a set of potential actions the agent can perform individually. Coordination between agents is then only required if there are states for which more than one optimal action exists that can be taken by an agent. This coordination is modeled as a state game. Local agents determine the appropriate actions to select using knowledge of the expected payoff from taking any combination of actions. Boutilier's paper continues by focusing on how the use of designed conventions, such as lexographic ordering of decisions, and learning of conventions can facilitate the local coordination problem.

Tambe's Markov Team Decision Problem (Pynadath and Tambe, 2002; Nair et al., 2003) defines an approach for integrating distributed agents and partially-observable decision theoretic problems.. This work belongs to the third class of distributed planning and execution models, where planning is centralized but execution is distributed. The underlying model for a MTDP is very similar to that of MMDPs. A domain is defined by a set of world states, a set of (potentially different) actions available to each agent, a transition function which maps between world states given a joint action, an observation function dependent on the current state and the just applied joint action, and a reward function which is uniformly shared between all the agents. For MTDPs, development of an optimal global plan is performed centrally. Execution is distributed, however, with each agent having knowledge of the policies of the other agents. An agent does not know the observations or actions of the other agents, but uses knowledge of the other agent's policies to be able to respond appropriately considering the possible actions the other agents may take and observations the other agents may see. An optimal solution for an MTDP can be found by exhaustive analysis of the expected value of all joint policies. This exhaustive analysis is extremely computationally intensive, so an alternative approach to exhaustive search is to find

a locally optimal policy that provides a joint equilibrium. An exhaustive algorithm for finding the joint equilibrium of n agents is to repeatedly fix the policy for *n-1* agents and for the other one agent, find the individual policy which maximizes the joint reward. This process is repeated until no policies change. In the worst case, this algorithm is as complex as exhaustive search for the globally optimal joint policy. Tambe also introduces an alternative approach which takes advantage of dynamic programming and reasons over a joint belief state. This joint belief state takes into account the agents beliefs over the local state and the possible observation histories of all other agents. Experimentally, this alternative approach has been shown to be significantly faster than the exhaustive search for an equilibrium. The class of decentralized POMDPs, of which the MTDP methodology is a member, has been shown to be NEXP-hard by Bernstein (Bernstein et al., 2002). NEXP is the class of problems for which, if there is a solution, the solution can be checked in exponential time.

2.5 Principled Agents

The successful application of multiagent systems relies on groups of agents that are able to exhibit and exploit synergy. A combination of distinct agent abilities and knowledge often allows groups of agents to accomplish goals individual agents could not achieve in isolation. Synergy is not guaranteed in multiagent systems, however, as unconstrained interactions may lead to incoherence, redundancy in work, and poor execution. An approach taken to avoid these problem is to apply concepts from human sociology to multiagent systems.

As an example of this approach, consider human vehicle traffic. Traffic is a very-complex multi agent system with highly-autonomous drivers as the agents and the potential for many hazardous situations. However, due to social laws and conventions, traffic often flows very smoothly. This can be partially attributed to a driver's ability

to predict the expected behavior of others. Laws are essential in this ability, especially in hazardous situations. One can expect that other drivers will stop at a red-light or will drive the correct way down a one-way street. By constraining the actions that a driver can do, laws constrain the situations a driver must plan for. Conventions are also important in allowing drivers to predict the actions of others and respond accordingly. Although it is legal to pass on the left or right on the interstate, most people tend to pass on the left. From this convention, slower cars tend to stay on the right side of the interstate, allowing faster moving traffic to drive on the left and maintaining good traffic flow.

Social laws define constraints on the actions that an agent can perform. Since all agents are aware of and behave according to these laws, social interaction between agents is facilitated. Shoham and Tennenholtz have studied the off-line generation of social laws (Shoham and Tennenholtz, 1995). Given a state of the world, a social law restricts the set of all possible actions that could be taken to a limited set applicable in that state. Knowledge of social laws allows agents to make predictions about the potential states of other agents in the environment without needing complete knowledge of the system. An *useful* law is defined as a social law under which a plan exists such that, for any two states of interest, any execution of the plan (actions are non-deterministic so the actual execution results may differ) moves the agent from the first state to the second. The general problem of determining useful social laws, however, is NP-complete. This work has been extended (Fitoussi and Tennenholtz, 2000) to define *minimal* and *simple* social laws, which are useful in selecting which of a set of social laws to apply to a system. A *minimal* social law is a useful law that defines the least number of constraints on an agents behavior. By being less restricted, an agent is able to be more flexible in determining ways to achieve its goals and adapt to the environment. A *simple* law is the useful social law that is

the easiest to follow. This type of law would require the least from agents in terms of sensing and computational power, and should therefore be the laws most easily learned and obeyed by agents and the laws most applicable to the widest range of agents.

An alternative approach to guiding social behavior for a given system is to allow social conventions to arise from the interactions of the agents. These conventions are often called norms and constitute expected behaviors for agents. The emergence of norms facilitates social interactions much like social laws, as norms allows agents to be able to predict how other agents should act in specific situations. Shoham and Tennenholtz (Shoham and Tennenholtz, 1997) note that conventions arise from the sharing of information between agents during local interactions. Adaptations to behavior are made due to increased knowledge between agents and end up defining a particular strategy for an agent to follow. The speed at which conventions can emerge is dependent on a number of variables, including the history the agent maintains, how often updates are made to local rules, and the update rule that is used.

A third sociological approach is to apply a social philosophy or ethics to each agent that guides an agent's choice of actions. This approach is not as constraining nor domain-dependent as social laws that explicitly define which actions are available in which states. It is also different from social conventions as there is no period in which the agents have to learn what strategy should be taken. Instead, the goal of this approach is to have an agent select actions which are likely to satisfy its local needs as well as facilitate global interactions.

Early work in this area by Etzioni and Weld (Weld and Etzioni, 1994) integrates Asimovian principles into classical planning. The motivation behind this work is to facilitate acceptance of agents into human society by determining verifiable methods for making agents safe. Agents are free to perform any actions, as long as the outcome

of their actions is safe or can be made safe by further actions. Three main concepts are explored in the paper: how to define harm in the context of planning, how to determine and avoid harmful actions in a plan in a generic and computationally feasible manner, and how to integrate the dual, and potentially conflicting, goals of harm prevention and task achievement. Two primitive constraints are defined which agents are required to obey during planning. The first primitive, *don't disturb*, holds over all periods of execution and should not be violated even when requested by a user. This primitive has as a parameter one function free logical sentence. The truth value of this logical sentence should not be violated by execution of any action. A plan can be considered to satisfy the constraint if for all totally ordered and consistent orderings of actions, the logical sentence holds true before and after execution of each planned action. To ensure that generated plans are safe, all effects of each action to be added to a plan must be verified to satisfy all *don't disturb* constraints. If a conflict between the effect of an action and a *don't disturb* constraint is found, one of four repair steps is executed: disavow, confront, evade, or refuse. Disavowing occurs when the effect of the action is true in the initial state. If this case holds, then performing the action does not create additional constraint violations and can be executed without problem. Confrontation adds an additional action before a potentially harmful action to prevent violating effects. If a harmful effect of action A is conditional on a state S, then the agent may add another action B before action A which ensures *not* S when A is performed. Evasion uses goal regression to determine the preconditions for ensuring the constraint is not violated after execution and adds actions to be performed which bring about those preconditions. Finally, refusal is when an alternative action is searched for to replace a harmful action in the plan.

The second primitive is termed *restore* and requires that the state of the world not be violated after execution of the entire plan. During execution the agent can make

any changes necessary to the state of the world as long as it restores the constrained states before execution terminates. This primitive is very relaxed compared to *don't disturb* and is considered secondary to user requests. Development of a plan that satisfies restore constraints can be accomplished through two phases of planning. In the first phase, a standard search for a plan is performed. After this plan is generated, the agent can check the effects of every action in the plan to verify that they do not violate any of the *restore* constraints. If a constraint is violated, a new goal is generated that resolves the violation and the agent attempts to revise its previous plan with new actions that achieve the modified goal set. Since *restore* constraints are secondary to goal fulfillment, if an agent can not find a suitable plan that resolves a violation, the violation is ignored. Each resolving action added to the plan may violate other constraints, so constraint resolution must continue until there are no further violations. Etzioni and Weld note that this method is not guaranteed to generate a violation-free plan even if one exists. Given two action branches that achieve goals, the agent may arbitrarily select one due to the method of search used by the agent's planning algorithm. This branch may introduce violations that are not fixable, while the alternative branch would not. To correctly handle this problem, exhaustive search over the entire plan space would be required. Fortunately, this case rarely happens as most actions are reversible by other actions providing for easy restoration of values. Additionally, most restoration actions have few interactions and can be easily serialized. A third constraint, *min-consume*, is suggested for ensuring efficient use of resources, but is not completely examined in the paper. The authors mention that this type of constraint would most likely require examination of all potential plans to determine the most efficient plan for a resource.

Another approach being developed to support principled agents is the work of Rose and Huhns on philosophical agents (Rose et al., 2002). Their paper develops

an action selection mechanism based on a seven-layer deliberative architecture. This architecture starts with a pure reactive layer and is extended to include, in order, theorem proving and utility maximization; beliefs, desires, and intentions; rationality and decision theory; social commitments; and societal norms and conventions. At the top of the deliberative architecture, Rose and Huhns suggest the addition of a layer of philosophical principles. Though many different philosophies could be applied at this level, it is believed that a philosophy applicable to ensuring global coordination and coherence embeds rules that are similar to those of Asimov's Laws of Robotics. Seven principles are delineated that agents should follow:

1. An agent shall not harm the mission through its actions or inactions.

2. Except where it conflicts with principle 1, an agent shall not harm the mission participants.

3. Except where it conflicts with the above principles, an agent shall not harm itself.

4. Except where it conflicts with the above principles, an agent shall make rational progress towards mission goals.

5. Except where it conflicts with the above principles, an agent shall follow established conventions.

6. Except where it conflicts with the above principles, an agent shall make rational progress towards its own goals.

7. Except where it conflicts with the above principles, an agent shall operate efficiently.

This approach is motivated by the philosophical theory of deontology. Agents acting under this philosophy consider the rightness or morality of their actions as

more important than the progress made towards goals. An alternative philosophy, the teleological theory, espouses the belief that as long as a valuable final state is reached, the set of actions that were used to achieve that state and their consequences don't matter. Given that agents want to perform right or moral actions, agents need a way of determining whether or not an action is right. A general method for this determination is for an individual to examine the performance of an action as if it was a universal law (Kant's categorical imperative). If everyone followed the same reasoning for performing the action, what would the effects be? Assume that an agent could steal to make money or go to work to make the same amount of money. Although stealing may be easier, if all agents stole, then no one would be working and there would be nothing to steal. This leads to the belief that it is not right for agents to steal. Alternatively, one can define right actions as those that are applicable within a set of moral laws . The principle hierarchy defined by Rose and Huhns implements such a set of moral laws that an agent should act under, and each principle can be validated as right or moral through application of Kant's categorical imperative. For example, if all agents decided to operate inefficiently, then the system would perform very poorly. Thus, it is not right for one agent to operate inefficiently. This same reasoning can be used to validate the rest of the principle hierarchy (Rose and Huhns, 2001).

Agents that follow this set of principles should tend to select and perform actions that are applicable primarily to the success of the mission, then those actions that aid the society of agents, and finally those actions beneficial to the agent itself. Preventing harm to other agents is considered to be of higher importance than goal completion, as endangering other agents can lead to failure of the system if some agent roles are critical and limited.

Preliminary experiments using this approach simulate space exploration in which

groups of agents search for and collect mineral specimens on an unexplored planetary surface. In these experiments, various subsets of principles were selected for experimentation and agents were designed to those principles. This work demonstrated that a measure of global coherence can be achieved even when control is localized in a large number of autonomous agents, that cooperation among agents (helping each other to achieve mission goals) provides a much higher level of mission success than self-interested or competitive approaches, and that agents using such a principled approach are able to discover and recover from defective behavior shown by other agents. Although this approach has only been applied to relatively small and homogeneous groups of agents, it appears to be a viable starting point for expansion through the addition of planning and by scaling to large numbers of agents. Since an agent's principles guide action choices at a high level, this approach should be applicable to any groups of agents independent of what primitive actions the agents can perform. In this work, agents were coded specifically to act in accordance with different sets of philosophical principles.

To implement a means of reasoning over social principles in a decision theoretic framework, algorithms are required for mapping from social principles to local qualitative preferences and from preferences to utilities. Research into qualitative preferences and mapping preferences to utilities has been fairly active. An area of particular importance is *ceteris paribus* preference statements (Boutilier et al., 1999a). Given a state S and a set of actions A that can executed, there exists a set of potential outcomes O. For any two outcomes *o1* and *o2* in O, $o1 \geq o2$ represents a relative preference for outcome *o1* over *o2*. A preference ranking over all possible outcomes is then defined as a total pre-ordering using the relation $\geq$. Preferences, however, are not usually defined over outcomes, but in terms of features of the outcomes. Feature sets can be described as preferentially independent of other features sets (typically

their complement) if the preferences over assignments to a given set, when all other assignments do not change, remain the same independent of the values assigned to the other features. If this is true, then within the set, the preferences are *ceteris paribus* - they always hold given all other things equal. If the other features are not the same, however, no relative comparison can be made in this framework. Conditional preferential independence is defined as preferential independence between two feature sets if and only if a third feature set is assigned a specific feature vector.

Mutual preferential independence (MPI) is an extension of preferential independence which holds if a subset of features S is preferentially independent of its complement in the feature set F - S, for all subsets of the feature set. Given a mutually preferentially independent set of n features, it is possible to define a global utility function as the weighted sum of utility functions for the n individual features: $U(F) = \sum_{i=0}^{n} w_i * U(F_i)$ (Marichal and Roubens, 2000).

2.6 Problems With Historical Planning Methods

As described in this chapter and the introduction, the various popular classical planning methods are incapable of supporting all of the necessary features that will be seen in real-world domains. One area in particular where such algorithms fail is in handling action and state uncertainty. Fortunately, algorithms for optimally solving Partially Observable Markov Decision Processes (POMDPs) have been developed as POMDPs can effectively and correctly model decision-making under uncertainty and are applicable in a wide variety of domains. Unfortunately, however, these optimal solutions are computationally expensive to find, leaving exact solvers to work well only for small domains.

POMDPs are difficult to solve for two main reasons. The first cause is described as the *curse of dimensionality*. The curse of dimensionality is related to the size of the

state space for a given problem. Given a state space of size N in POMDP, the belief space for an agent in that domain is continuous and of size *N-1*. The second cause for computational complexity in POMDPS is the *curse of history*, which can be related to the size of the action and observation space. As the number of possible actions and observations grow, the agent must consider a much larger set of action-observation sequences (Pineau et al., 2003). An additional problem with general POMDP solvers is the general approach taken of computing a complete policy beforehand. Since optimal solutions are difficult to find, and a good policy for any state may not be developed until several value iteration steps have been taken in a general solver, it is often not possible to develop an effective control system for an agent within a reasonable amount of time.

Often times there is a significant amount of structure in POMDP domains which general POMDP solvers are unable to take advantage of. By exploiting such structure, it is possible to significantly reduce the computation time required to find solutions. Examples of methods for exploiting problem structure are factored representations (Boutilier et al., 1999b), hierarchical abstraction (Pineau and Thrun, 2002; Theocharous, 2002; Hansen and Zhou, 2003), and point-based computation (Pineau et al., 2003). Some methods, such as factoring, do not affect the quality of the solution that is developed, while others, such as the introduction of abstractions, only allow approximations to the optimal solution.

Traditional approaches to planning for distributed agents and social coordination also suffer from problems. In environments with uncertainty, POMDPs over multiple agents (Bernstein et al., 2002; Nair et al., 2003), where planning is centralized, are extremely computationally complex. The Multiagent Markov Decision Problem approach (Boutilier, 1996), while allowing distributed decision making, requires knowledge of the global world state and a global utility function by all interacting

agents. On the opposite end of the scale, emergent social conventions (Shoham and Tennenholtz, 1997) require a period of potentially faulty interaction before conventions among agents begin to develop. Social laws (Shoham and Tennenholtz, 1995) have the potential to be arbitrarily restrictive. Alternative approaches that have seen success in bringing about useful social coordination, such as FA/C and GPGP, suggest that distributed approaches to planning are possible. These approaches make use of continual planning, in which agents are able to constantly revise the actions that are taken and the knowledge that is shared with other agents. Recent work has also begun to integrate MDPs (Xuan and Lesser, 2002) into local planning mechanisms to support local uncertainty, suggesting that multiagent systems of MDP-based agents are viable.

2.7 Proposed Solution

A novel model for planning agents is proposed which is capable of exploiting multiple types of structure in POMDP domains, providing for both fast and high quality decision making when such structure is present. In particular, the proposed model moves from pre-execution planning to run-time planning to be able to take advantage of run-time knowledge about the world. The proposed model also uses dynamic decision networks as the reasoning architecture, which allows for a factored representation of a given problem. By taking advantage of run-time knowledge, the model should support policy construction and reuse at run-time and allow construction of minimal-size decision networks at runtime. Finally, the proposed model supports hierarchical abstractions in dynamic decision networks, allowing for reductions in state, action, and observation space.

To support planning and social coordination with groups of agents reasoning under uncertainty, a novel model for distributed planning is proposed. In this model, agents

act under a small set of principles which define their preferences for states of the world, with the principles being biased to support social goal achievement. By mapping these preferences to utilities and optimizing in a MDP-type framework, agents can select actions that are able to support social goal achievement as well as allow local optimization.

Chapter 3

Descriptions of Test Domains

This chapter describes the set of POMDP domains and multiagent MDP domains used in this research. The domains are presented here to allow familiarity with the types of problems that were considered. Each domain presented was used for testing the developed algorithms. Descriptions of the developed algorithms, presented in the next chapter, also reference these problems.

3.1 Parts Problem

The parts planning problem (Draper et al., 1993) is a relatively simple 4-state problem which requires both information gathering and task actions. In this domain, a factory agent is presented with a part which it must process. A part is described by three features: *flawed (FL)*, *blemished (BL)*, and *painted (PA)*, and the agent can perform four primitive actions: *Inspect*, *Reject*, *Paint*, and *Ship*. The part can be in one of four states during its lifetime, *!FL-!BL-!PA*, *!FL-!BL-PA*, *FL-!BL-PA*, and *FL-BL-!PA*. Initially, the part is not *painted* and can either be *flawed (FL-BL-!PA)* or not *flawed (!FL-!BL-!PA)*. A flaw always shows a blemish, unless the part has been painted, which covers any blemishes. *Paint* works 90% of the time and fails 10% of

the time, while the *Inspect* action returns the true *blemished* state of the part 75% of the time and the incorrect state 25% of the time. All other actions always return *!BL*. The agent receives a reward of +1 for performing *Reject* on a *FL-BL-!PA* part or *Ship* on a *!FL-!BL-PA* part, and pays a cost of -1 for *Ship* or *Reject* in any other state. The agent can *Inspect* and *Paint* the part for free. An optimal policy for this problem can be found by an exact POMDP solver. The optimal policy for this domain is for the agent to take an initial action of *Inspect*. If the *Inspect* action returns *BL*, the agent should perform *Reject*, otherwise the agent should perform, in order, the operators *Paint* and *Ship*.

3.2 Cheese Taxi

The cheese taxi problem (Pineau and Thrun, 2002) was originally tested by Pineau for the HPOMDP algorithm. This problem is a combination of Dietterich's taxi problem (Dietterich, 2000) and McCallum's cheese maze problem (McCallum, 1993). In this domain, the agent can be in one of eleven locations, labeled *0* through *10*, and have one of three destinations: *NULL*, *0*, and *4*. These locations and destinations are shown in Figure 3.1. The agent can perform seven actions: *North*, *South*, *East*, *West*, *Pickup*, *Putdown*, and *Query*. If the agent has a *NULL* destination, its task is to *Pickup* the passenger at location *10*. The agent should then navigate the passenger to its requested destination and *Putdown* the passenger. The problem is made more difficult, however, as the passenger does not mention where he would like to go unless a *Query* action is performed. Also, there is a 5% chance the passenger will change their mind on which destination they prefer if the taxi is located in position *2* or position *6* and is performing a navigation action. Once again, the passenger must be queried for the new destination. The taxi agent suffers a loss of -1 for each individual action that is performed, and a cost of -10 if the taxi agent attempts the actions of

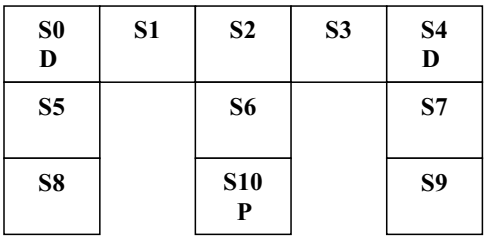

Figure 3.1: A map of the cheese taxi domain.

Pickup or *Putdown* in the wrong place. A reward of +20 is obtained for performing a *Putdown* action at the correct destination.

3.3 Large Cheese Taxi

The Large Cheese Taxi domain is an extension of the original Cheese Taxi problem, preserving most of the features of the original problem but scaling up the number of states the agent has to reason over. This domain consists of 30 possible taxi locations and 10 destinations for a total of 300 possible states, approximately an order of magnitude larger than the original problem. In this domain, the passenger can change his mind if navigating through location *2* or *6* and the destination is labeled *0* or *4*, as in the original problem, and can also change his mind between destinations *20* and *24* if navigating through locations *18*, *22*, or *26*. The actions available in this domain are the same as for the Cheese Taxi domain. Other than the mentioned changes, the rewards and transition probabilities are also the same as those in the original problem. Because there are more destinations and fewer chances for the passenger to change their mind, this version of the problem is less dependent on the Query action. This version of the problem also allows multiple alternative routes to destinations, so the taxi agent can navigate around potential problem areas if the alternative route does not impose too large of a penalty. Figure 3.2 is a representation

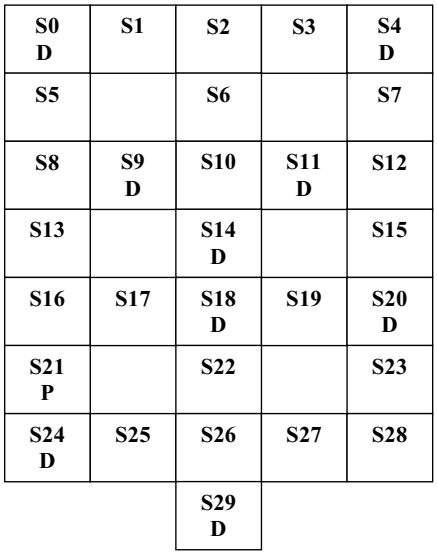

Figure 3.2: A map of the large cheese taxi domain.

of the large cheese taxi domain.

3.4 Twenty Questions

A fourth problem used for testing was Pineau's twenty questions domain (Pineau and Thrun, 2002). This domain is different from the previous domains as it is very information intensive. In this problem, an outsider selects an object, while the planning agent asks yes or no questions to try to determine what the object is. The outsider can change objects (11% probability) after answering a question and sometimes gives a noisy (5% probability) or incorrect (10% probability) answer to a question. Some objects can also be ambiguous. For example, if the object is an apple, which can be either green or red, and the question is *Is it red?*, there is a 50% chance of a *Yes* answer, a 45% chance of a *No* answer, and a 5% chance of a *Noise* answer. Each question costs the asking agent -1, an incorrect guess has a cost of -20, and a correct guess has a reward of +5. In the tested problem there were 12 possible states and

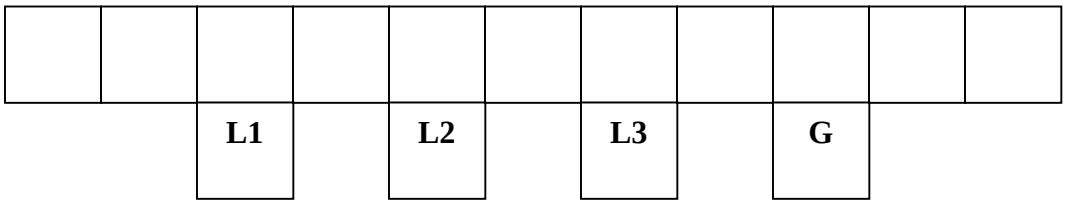

Figure 3.3: A map of the hallway domain. Landmarks are marked with an L, and the goal state is marked with a G.

20 possible actions. The possible states corresponded to objects the outsider could choose and consisted of *Monkey*, *Rabbit*, *Tomato*, *Potato*, *Apple*, *Ruby*, *Marble*, *Coal*, *Robin*, *Mushroom*, *Banana*, and *Carrot*. The twenty actions were a *GuessX* action for each object, as well as *askAnimal*, *askVegetable*, *askMineral*, *askWhite*, *askRed*, *askBrown*, *askFruit*, and *askHard*.

3.5 Hallway

The Hallway problem was originally introduced by Littman, Cassandra, and Kaelbling (Littman et al., 1995a). In this problem, an agent must navigate through a series of rooms to a specified goal state. The agent can perform five different actions: *Stay*, *MoveForward*, *TurnLeft*, *TurnRight*, and *TurnAround*. There are 60 true states the agent can be in during execution. The agent can be in any of four directions within 15 possible different rooms. Transitions are noisy, leading the agent to end up in the wrong direction or wrong room fairly often. Observations are also noisy, with many observations possible for each state. When in three specific rooms facing south and when in the goal state, the agent is guaranteed to receive a landmark observation which gives the agent knowledge of its exact location. Figure 3.3 represents the layout of the hallway domain.

3.6 Robot Tag

The robot tag problem is another problem from Pineau which resembles the popular game of laser-tag. In this problem, a robot player can be in one of 29 positions on a board and the opponent can be in one of the same 29 positions or in the *Tagged* state, providing for an 870 state problem. There are five actions available to the robot: *North*, *South*, *East*, *West*, and *Tag*. All actions are deterministic in that the robot will move correctly where it desires to move. The location of the robot is fully observable, while the location of the opponent is unobservable until the robot and opponent are in the same location. There are thirty possible observations - a "same position" observation returned when the robot and opponent are in the same location, and the robot location for all other states of the world. When this problem was used for testing, significant discounting in the DDN, at the level of 0.3 per timestep, was required. Higher discount factors were tried and most led the agent to perform fairly well. High discount factors, however, would eventually lead the agent to get stuck in a position unable to decide which move to make. A discount factor of 0.3 causes the reasoning mechanism to focus on the most current states of the world and gives less weight to future states, resolving cases where the agent would get stuck. Figure 3.4 represents the layout of the robot tag world.

3.7 Helo Domain

The HeloDomain problem was originally developed by Tambe and Pynadath in evaluating their theory of Communicative Markov Team Decision Problems (Pynadath and Tambe, 2002). This problems consists of two helicopter agents - a transport that needs to deliver its troops to a specific enemy location and an escort designed to take out the enemy radar and lead the transport to that spot safely. A simplified version

					S26	S27	S28		
					S23	S24	S25		
					S20	S21	S22		
S10	S11	S12	S13	S14	S15	S16	S17	S18	S19
S0	S1	S2	S3	S4	S5	S6	S7	S8	S9

Figure 3.4: A map of the robot tag domain.

of this problem was designed as a means of verifying the proposed socially-oriented planning algorithm.

The transport agent can be in one of 11 physical locations *(start, 1..9, and destination)* or in the *destroyed* state. It also has knowledge of whether the enemy radar has been destroyed. There are two means of obtaining this information. One is to visually sense the destruction of the radar by being in the same location as the escort when the escort shoots the radar. The other is from communication initiated by the escort. The transport agent has two actions: *FlyLow*, which progresses the helicopter forward one step at a time and is ensured to evade the radar, and *FlyHigh*, which moves the helicopter two steps at a time but results in a destroyed transport if the radar isn't destroyed.

The escort agent can be in one of six physical locations *(start,1..4, and destination)* (it moves twice as fast as the transport) and the *destroyed* state. It also knows the location of the transport agent, whether or not the radar has been destroyed, and if it hasn't, whether or not the radar is in the same location as itself, if it has flown past the radar, or if it has not yet seen the radar. The escort also knows whether or not it has communicated to the transport agent. Finally, the escort knows that if the transport believes the radar has been destroyed, it will automatically move to

FlyHigh (so it has some knowledge of the policy for the transport). The escort can take three actions - *Fly*, which moves it ahead one step; *Destroy*, which destroys the radar if the radar is in the current location but doesn't move the escort ahead; and *Communicate* which tells the transport the radar has been destroyed. Performing *Communicate* results in the escort being shot and a transition to the *destroyed* state for the escort if it hasn't already landed at the destination. Both agents are also aware of a strict deadline of 13 timesteps for achieving the mission goal states.

The independent goal for each agent is to reach the destination location as fast as possible to receive a progressively higher individual reward for more time left at the destination (+1 for each remaining timestep). The social goal is solely to get the transport to the end state before the deadline (it must be there with at least one timestep left) without it being destroyed. This social goal interferes with the escort's individual interests and makes the problem interesting as the escort will sometimes have to consider being shot down to achieve the mission goal.

A fully-observable version of this problem was also developed. This version of the problem, with perfect information, allows for evaluation under optimal conditions. In the fully-observable problem, the *Communicate* action is no longer needed, as the true state of the radar is known by both the transport and escort.

3.8 Tragedy of the Commons

Tragedy of the Commons is a commonly seen problem in multiagent systems (Turner, 1993). In general, this problem can be defined as agents which, in attempting to optimize their use of a community resource, actually end up wastefully consuming the community resource. In the general cow grazing version of this problem, there exists a community field for grazing and a number of agents that have cows that need to graze. Each of the agents tries to get as many cows as possible to graze on

the common field. When too many cows begin to graze, reaching the fields carrying capacity, things break down as the field is destroyed. This then causes problems for all cows on the field. This problem is directly related to many real world domains where there is a single resource needed by multiple systems (such as bandwidth management and processor scheduling).

In the tested variant of the problem, there are three agents that manage one cow and one pasture apiece as well as share a common pasture. At any timeslice, each agent has two actions it can take: *MoveToCommon* and *MoveFromCommon*. *MoveToCommon* moves a cow to the common pasture and *MoveFromCommon* moves a cow to the local pasture. Cows that graze in their local pasture put on weight at a rate of 1 unit per timeslice. The common pasture has better grass, allowing for a weight increase of 2 units per timeslice. While the individual pastures have an unlimited amount of grass, the common pasture, if it is completely consumed, will not regrow. If not completely consumed, it regrows at a rate of one grass unit per timeslice. Maximum weight for all cows is 12 weight units, and the maximum amount of common grass is 15 units. Agents have 6 timeslices to fatten their cows for market. To make any profit, a cow must weight at least 10 units, and the profit increases by 1 for each unit over 10. An individual cow at 10 weight units sells for 110 dollars, 11 units for 111 dollars, and 12 units for 112 dollars. Each agent knows how much grass is in the common pasture, its own cows weight, how many cows are on the common pasture, whether its cow is on the common pasture or in the local pasture, and how much time is left. This constitutes complete knowledge of an individuals own state, but not complete knowledge of the entire world state. Agents also do not know the policies of the other agents (what actions others are going to take in a state) or any preferences the others have over actions. Each agent maintains just a list of the available actions the others could take and the effects of taking each action in the

list.

The natural individual achievement reward for this problem is to maximize profit when the cow is sold. Accordingly, individual goal achievement states are designated as those states where the agents cow weighs more than 10 units when there is no time left. Without the common pasture, it becomes much more difficult to add weight to cows in the future. Thus, the notion of the social goal for all of the agents is to ensure that the common pasture never reaches a state of zero grass. This leads to a definition of social goal achievement states as those where the amount of grass in the pasture is greater than zero when there is no time left.

Chapter 4

Architectural Overview

This chapter describes the core components of the E-Plan agent architecture. Section 4.1 begins the chapter with a definition of the individual agent software components. This definition covers the high-level architecture of the agent. Section 4.2 describes implementation-specific details of the current E-Plan agent architecture. The rest of the chapter is used to introduce the novel algorithms that are the focus of this work. Sections 4.3 through 4.5 present approaches for exploiting structure and run-time knowledge in POMDP domains, while section 4.6 describes the algorithm for developing a principle-based, socially-biased utility function for cooperative MDP domains.

4.1 Local Agent Components

An E-Plan agent acts under the traditional sense-plan-act cycle control mechanism. In this control mechanism, the agent starts by sensing the world and updating its beliefs about the world given the observations it has just made. After updating its beliefs, the agent reasons about the appropriate action to take given those beliefs. Once an action has been chosen, the agent acts by executing that action. The performance

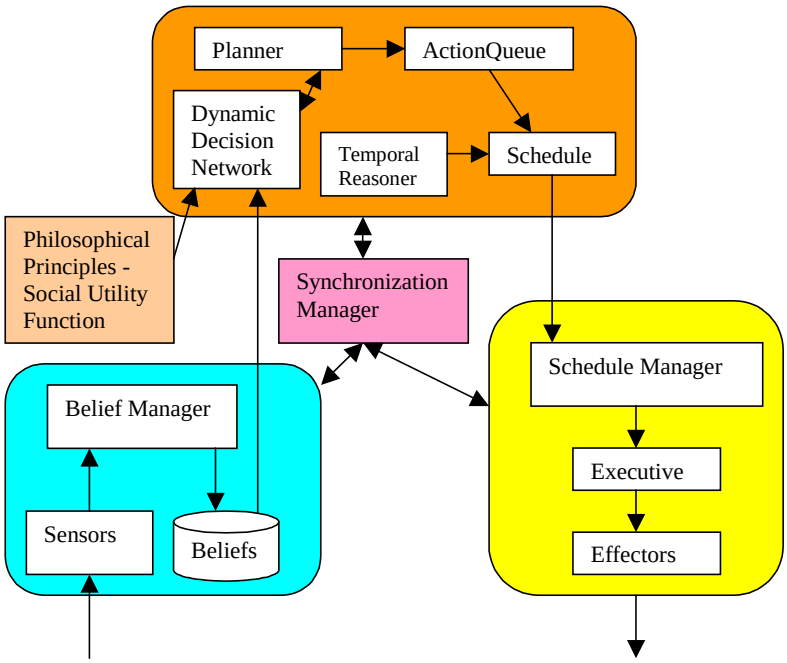

Figure 4.1: A depiction of the high-level E-Plan agent control mechanism.

of an action often affects the state of the world. These effects, as well as natural changes made by the world itself, are sensed in the next control cycle. The E-Plan software architecture contains components that are directly related to each part of the sense-plan-act control mechanism. Figure 4.1 is a graphical representation of the E-Plan local agent components and their interaction during a control cycle.

An E-Plan agent maintains multiple *Sensors*, each of which provide observations about the state of the world. Each *Sensor*, when updated, signals the agent that the *Sensor* state has changed and is read to obtain the latest observation about the state of the world. The observation read from the sensor affects the beliefs the agent has about the true state of the world. In E-Plan, a *Beliefs* component is used to store both the observation read from the sensor and the agents beliefs about the true state of the world. In the *Beliefs* component, the agent maintains a table, called the beliefs table, which stores the current observations read from the available sensors,

and a priors table, which maintains the agents beliefs for each of the random variables used in the factored representation of the world state. A domain-dependent *BeliefManager* handles the updating of the *Beliefs* component and supports mechanisms for integrating and interpreting sensor results as needed to update the *Beliefs* component.

When an agent has completed sensing the available observations about the world, it uses a small Bayesian network to update the priors table it is maintaining. This Bayesian network is referred to as an update network, as it allows the agent to update its beliefs given its previous beliefs and the observations it has just received. The update network contains variables that represent variables of the world across two timesteps, an observation variable attached to the second timeslice, and an action variable. The agents current priors are mapped onto the first timestep variables. The previously performed action is mapped to the action variable, while the current observation is mapped onto the observation variable. By propagating evidence through this network, a new set of beliefs about the variables representing the true state of the world can be obtained by reading the probability distributions of the variables represented in the second timestep. A *DynamicDecisionNetwork* software component is used as a generic component for handling Bayesian networks and decision networks and is the component used for managing update networks.

Once an agent has updated its beliefs about the true state of the world, it should then choose the appropriate next action to perform to affect the world. This is handled by the agents *Planner* component. The *Planner* component uses dynamic decision networks to determine the next best action to perform. A *DynamicDecisionNetwork* software component is used to manage the dynamic decision network algorithms. A dynamic decision network for planning consists of 1) variables representing the state of the world and observations across multiple timesteps, 2) action nodes between each

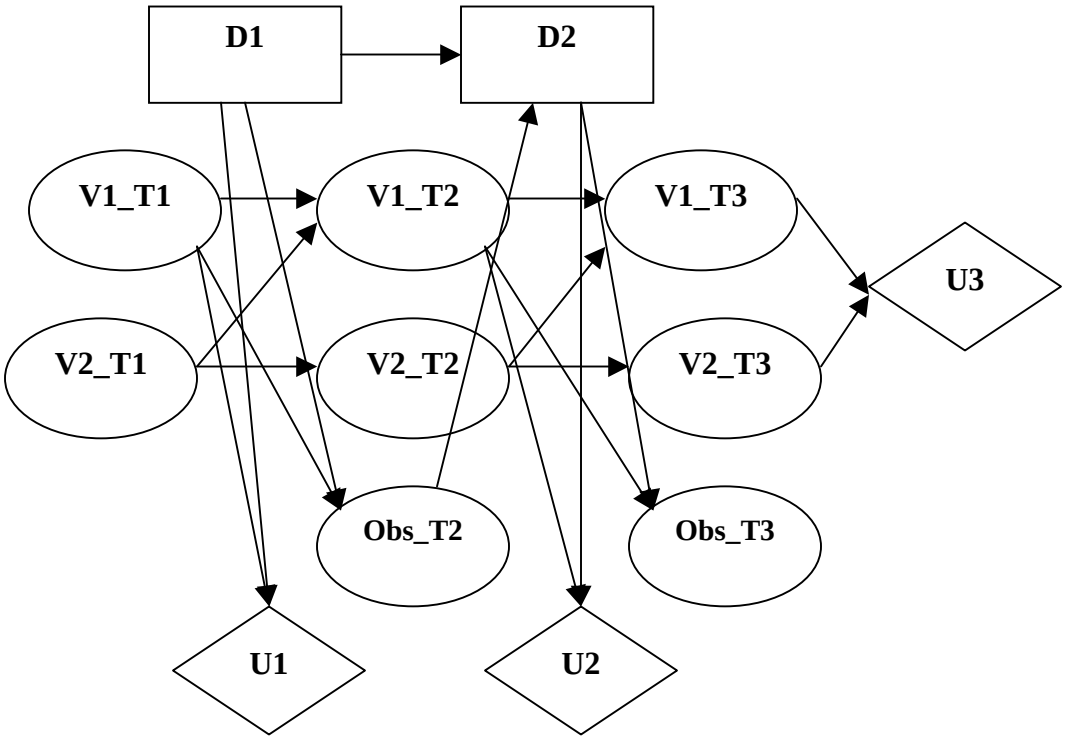

Figure 4.2: A dynamic decision network (DDN) used for planning. Oval nodes represent chance nodes in the DDN, with nodes labeled V1 and V2 representing the true state of the world and nodes labeled as Obs representing observations an agent makes about the world. Square nodes are decision nodes and represent the actions an agent can take. Diamond shaped nodes are utility nodes and are used to represent the costs and rewards of the domain.

timestep representing the available actions at each timestep, and 3) utility nodes for representing the cost of performing actions and the utility of being in specific states. The agents current beliefs about the true state of the world, from the *Beliefs* prior tables, are set in the variables in the first timestep of the dynamic decision network. By propagating these beliefs through the network and maximizing over action sequences, an optimal sequence of actions (of length equal to the number of timesteps represented in the network) is returned from the *DynamicDecisionNetwork*. A depiction of a dynamic decision network used for planning is shown in Figure 4.2.

Since DDNs are the datastructure used for planning, the underlying algorithm

for determining the optimal sequence of actions to perform relies on the process of variable elimination in the decision network (Jensen, 2001). This solution method begins with the last decision. The goal of the variable elimination algorithm is to find a function for the last decision which, given any past, determines the maximum utility action to take. Once this function (a policy function) is found, the agent works its way backwards to the first decision. The implementation of variable elimination for influence diagrams is derived from the procedure for propagating beliefs through a standard Bayesian network.

In variable elimination, there exists a group of potentials (utilities and probabilities) from which one potential at a time is removed. While any ordering of elimination is allowed in Bayesian network belief propagation (allowing for optimized orderings), the elimination order for decision networks is constrained by the temporal ordering of the nodes. Chance nodes in the set I_n are removed first, followed by the decision nodes in D_n. This is repeated for chance nodes in I_{n-1} and decision nodes D_{n-1}, continuing until I_0 is reached. For the example dynamic decision network depicted above, the final *V1* and *V2* nodes in timestep *T3* would be eliminated first, followed by the decision node *D2* located between timesteps *T2* and *T3*, and then repeating until all nodes in timestep *T1* are eliminated. The complexity of the potential computations is dependent on the largest set of nodes that is needed during this process. For general dynamic decision networks, this largest set of nodes contains all action nodes and observation nodes present in the network, as well as single nodes for the belief variables. The fact that dynamic decision networks algorithms are really examining all possible sequences of actions and observations is seen in the composition of this largest set of nodes.

The returned sequence of actions, represented as an *ActionQueue*, is then added to the agent's *Schedule*. In conjunction with a *TemporalReasoner* component, actions

are added to the schedule as needed. The *TemporalReasoner* component is a simple component which provides the agent with expected completion times for performing different actions. Each agent has an internal *Clock* component, which consistently updates a *ScheduleManager* with the current time. If an action exists in the schedule to be executed at the current time, the *ScheduleManager* removes the action from the schedule and triggers its execution. Execution is triggered by passing the action to an *Executive* component. The *Executive* is responsible for monitoring both the initialization and completion of all tasks performed by the agent. A domain-specific *ActionThreadHandler* is used to generate threads that correspond to performing tasks by the agent. Each possible agent action corresponds to a domain-dependent *Action-Thread* component, which performs the real work required by each task. When the task completes, the *Executive* is notified and the sense-plan-execute cycle is repeated. A *SynchronizationManager* component is used to control the interaction between the sense, plan, and execute components of the control cycle.

4.2 Implementation Details

The current implementation of the E-Plan agent architecture is developed on top of Zeus, a generic agent architecture developed by British Telecom. The Zeus agent architecture is used to support low-level message passing and directory services. The decision networks and algorithms used in planning are developed on top of the Hugin Belief Network API, which provides the basic decision-network implementations and algorithms. The variable elimination algorithm (Jensen, 2001) as implemented in Hugin compiles a dynamic decision network into a strong junction tree. Messages are passed through this tree to propagate probabilistic information and maximize over utilities. Java was used as the language to implement all higher level components of the E-Plan agent architecture.

4.3 Structural Exploitation

4.3.1 Hierarchical Dynamic Decision Networks

In the developed architecture, abstract actions are to be performed as if they were primitive actions to the agent. In this sense, planning for an abstract action is a means of finding a way to implement the abstract action, and ideally, the agent should be capable of implementing the action without needing to return to a higher task and replan. If abstract actions are modeled in this manner, however, one does not now know the probability for transitions between states and the rewards associated with completing an abstract action. The true model of the world that the agent is given only represents these values over primitive actions.

An example domain for describing this problem is the parts planning problem (Draper et al., 1993), described in detail in Chapter 3. In this problem, the agent must handle a part in a manufacturing plant. Each part is described by three features. These features characterize the part in terms of whether the part is *flawed (FL)*, *blemished (BL)*, and *painted (PA)*. The agent can perform four actions on the part. The *Inspect* action returns the *blemished* characteristic of the part. *Paint* paints the part. Finally, *Reject* removes the part from the plants inventory, while *Ship* sends the part to a customer. The optimal policy for this problem is for the agent to take an initial action of inspecting a part. If the inspect action returns that the part is blemished, the part is rejected. Otherwise, the agents paints and ships the piece.

An appropriate hierarchical decomposition of this problem is for the agent to reason over whether to *Inspect*, *Reject*, or *Process* the part, where processing is an abstract action that is the aggregation of the *Paint* and *Ship* actions. If the *Process* action is chosen, the agent needs to determine the appropriate means of implementing *Process* by finding an optimal ordering over *Paint* and *Ship* actions.

The agent's model of the world for this domain needs to describe transition probabilities between states given that the abstract action *Process* is taken, probabilities for observations given the *Process* action is taken, and rewards for taking the *Process* action when the part is in a specific state. Finding the actual values for these probability and reward variables requires knowledge of the true optimal method of implementing the abstract action. The true values can be found by finding an exact solution to a reduced POMDP, the approach taken by Pineau's PolCA algorithm (Pineau and Thrun, 2002), but these subproblems may be nearly as hard as the original problem to solve.

The approach developed in this work is to model the hierarchical meaning problem similar to the manner of HTN planning. This model assumes that the effects of executing an abstract action are the effects of successful completion of the action. The rewards and costs for taking actions are estimated by finding an exact solution assuming that the world is completely observable for each abstract action. Starting at the lowest level abstract actions, fully-observable problems representing the implementation of that action are solved as MDPs. Once the low-level abstract actions have a reward estimate, these estimates are used to solve MDPs for the higher level actions. This provides a polynomial-time method for generating initial estimates of the true utility functions. These approximations are not guaranteed to be good, however, as they do not take into account value of information and the use of pure information-gathering actions, both of which are important in POMDP problems.

Continuing the parts painting example, a model of the *Process* abstract action effects can be developed by using the effects of successful painting and shipping. Since the primitive action *Ship* is a terminal action for this problem and the transition introduces a new part (moving the agent to the initial state of being either *!FL-!BL-!PA* or *FL-BL-!PA*), the transition on the *Process* abstract action is also set to

!FL-!BL-!PA or *FL-BL-!PA*.

Similarly, the rewards for performing a *Process* action can be approximated by solving an MDP over the two primitive actions, *Paint* and *Ship*, that make up the *Process* abstract action. Since the *Process* action, once chosen, is supposed to be completed, the approximated reward depends on a part being painted and shipped. Given the part needs to be both painted and shipped, the rewards for *!FL-!BL-!PA* and *!FL-!BL-PA* are defined as .9 and 1 respectively (both move the agent to the optimal state of having painted and shipped the part almost all the time), while states *FL-BL-!PA* and *FL-!BL-PA* both have rewards of -1, the reward received on shipping a flawed part. These approximate rewards can then be used in the higher level POMDP where the agent reasons over primitive actions *Inspect* and *Reject* and the abstract action *Process*.

Given this approach, a hierarchy of dynamic decision networks is defined that an agent can use to select actions. In the parts domain, a two level hierarchy is defined: a root DDN, which reasons over *Inspect*, *Reject*, and *Process* actions, and a *Process* DDN, which reasons over *Paint* and *Ship* actions. Although the developed model of planning for abstract actions is akin to finding how to completely implement the current abstract action, an agent solely deliberating over actions for implementing the current task may miss opportunities to perform alternative actions that may be of use. A solution to this problem is to only schedule the first action from a generated plan and then replan at each level of the task hierarchy before the next action is taken. Using this approach, the agent can switch between different abstract tasks or levels of the hierarchy when necessary.

To select the next action to perform, an agent first instantiates the top-level decision network, generates an optimal sequence of actions to perform at that level, and selects the first action. If this action is a primitive action, it can immediately

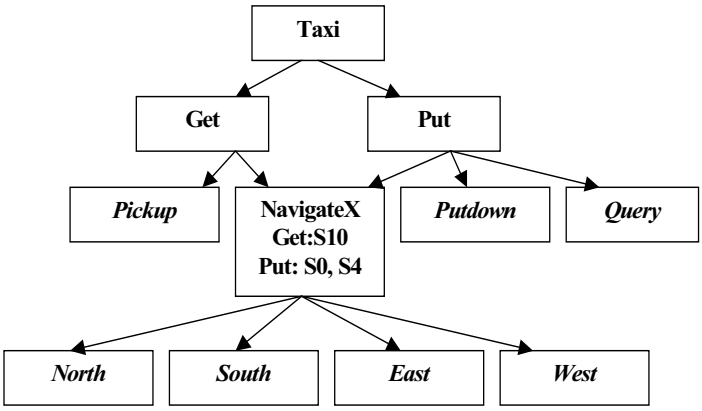

Figure 4.3: A hierarchical decomposition of a taxi navigation problem.

be executed. Otherwise, when the action is abstract, the agent instantiates the corresponding decision network for that abstract action, generates an optimal sequence of actions to perform at the new level, and selects the first action. This process is repeated until a primitive action to perform is found. Algorithm 1 presents a pseudo-code description of the hierarchical dynamic decision network planning algorithm. Figures 4.3 and 4.4 represent a likely hierarchy for a taxi navigation problem and the corresponding initial plan that would be developed. Primitive actions are italicized in each figure.

Algorithm 1 Hierarchical Dynamic Decision Network Planning

```
while agentRunning == TRUE do
  selectedAction ← Root
  while isPrimitiveAction(selectedAction) == FALSE do
    currentNetwork ← getHDDNetwork(selectedAction)
    setBeliefsInNetworkFromKnowledgeBase(currentNetwork)
    selectedAction ← variableElimination(currentNetwork)
  end while
  newBeliefs ← execute(selectedAction)
  updateKnowledgeBase(newBeliefs)
end while
```

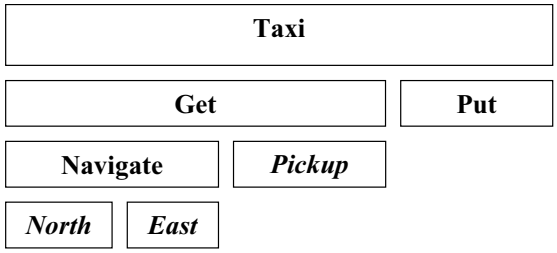

Figure 4.4: An sample initial plan developed by the HDDN algorithm given the preceding decomposition.

4.3.2 Plan Caching

As defined previously, agents acting under a POMDP control mechanism maintain a current belief state. This belief state is a probability distribution over the true states of the world. The total number of possible belief states is infinite, ranging over all possible combination of reals between zero and one that sum to one assigned to the states for each random variable (from a DBN point of view) needed to represent the environment. The exact approach to solving POMDPs requires finding the PWLC value function that covers the range of possible beliefs. Many of these belief states, however, will never be encountered during execution by an agent. In fact, many of the states may not even be reachable from a given start state. Additionally, in environments that are fairly deterministic, there are trajectories of belief states that the agent will encounter fairly often. Since agents in the described architecture are planning at runtime, they can discover and take significant advantage of their knowledge about which belief states are important.

To take advantage of such knowledge, a plan cache has been developed in which an agent can store the appropriate action for a given belief state. The primary advantage of such a cache is that an agent can re-use precomputed optimal actions when the agent enters a belief state it has visited before. In the developed architecture,

cache entries are preserved across trials so the agent can make use of all of its past experiences. When computation times are still too expensive to allow for true run-time planning, an agent designer could allow the agent to simulate a large portion of the state space pre-execution, storing its chosen actions during the simulation in the cache. This cache can then be used as the basis for a reactive policy, with the appropriate actions for the states that the agent is likely to enter at runtime already precomputed. To implement a traditional cache, a hashtable is used for storing a representation of a belief state and the corresponding optimal action. The hash key for the cache table is a string that represents the factored belief state of the agent. When the cache size is limited, entries are replaced using the least recently used (LRU) replacement algorithm. A timestamp is stored when a belief state is either added to the cache or accessed to support the LRU algorithm.

A second approach to using caches in POMDP planning is to simulate grid-based POMDP algorithms, allowing for a fine grid to be built around the interesting and reachable parts of the state space. This algorithm is inspired by the success of other point-based and grid based approaches to finding policies for POMDPS. An example of such an algorithm is Pineau's Point-Based Value Iteration algorithm (Pineau et al., 2003), described in the background chapter.

To implement a grid-based POMDP approximation using a plan cache, the agent uses a belief similarity retrieval scheme. In this scheme, the agent designer can set a threshold for the difference between the agent's current belief state and a cached belief state. The smallest threshold would be zero, wherein the cache is equivalent to the standard exact-match cache. Thresholds greater than zero, while potentially useful for approximating the best action, can also degrade the performance of the agent as the use of an action optimal for the closest belief point may not always be optimal for the current belief point. The current implementation of the grid-based cache defines

the difference in belief states as the cumulative shift in the probability distribution. For two belief states, this shift is computed as the sum of the difference in likelihoods over all underlying true states. Other measures, such as Euclidean distance or KL divergence between the two belief points, could also be made use of in this context.

If the difference between the current belief state and one in the cache is below the designer's threshold, the agent is willing to accept the cached action for the nearby state as the best action for the current state. If two or more points are below the threshold, the closest point is used as the approximating belief point. This approach can grow computationally more expensive as the cache grows, as the agent needs to compute the distance between each belief point in the cache and the current belief point. The timestamp on the belief state present in the cache that is most similar to the current belief state is updated to reflect its usage. The true belief state is not added to the cache, however, to ensure that the incremental propagation of small belief differences does not introduce significant error. Figure 4.5 demonstrates a grid-based cache for a 2-dimensional belief space. Several cached belief points and the line segments representing their surrounding threshold regions are highlighted. Grid-based caches are also referred to as epsilon-caches later in this work.

For the grid-based cache, the semantics of a shift in belief space are based on differences between true belief states. Accordingly, the comparisons can not be performed between vectors representing factored state entries. The present implementation of a grid-cache uses a representation of the non-factored belief state for each cache key. This requires a considerably larger amount of space for storage than factored representations. An alternative would be to store in the cache the factored representation and then compute the non-factored belief state for every cache entry when the cache is searched. Since caches can grow fairly large, the current implementation takes advantage of available memory to save the conversion during cache searches.

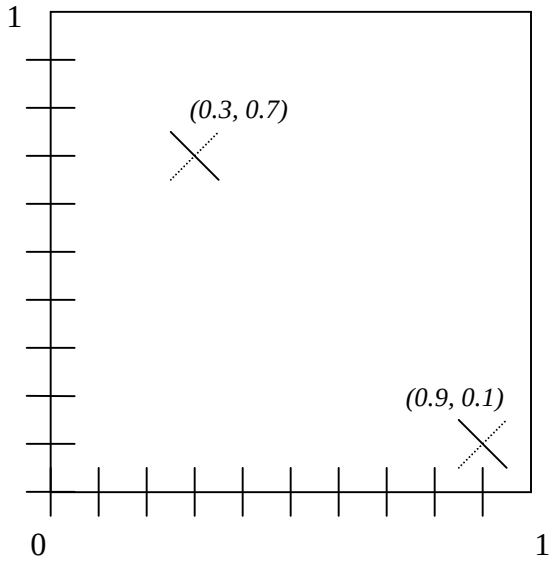

Figure 4.5: A 2-dimensional grid-based cache with cached actions and surrounding 0.10-threshold regions.

4.3.3 Dynamic Construction of Decision Networks

Common implementations of DDNs for planning contain factored representations of the entire state space. All possible states for each variable are represented in the respective variable nodes. This approach allows one DDN to be designed by an agent designer which can be used at anytime by the agent. If an agent is planning at runtime, however, it has specific beliefs about the current state of the world. After performing several actions, an agent's beliefs often start to center around only a small number of states, with the likelihood of the other states becoming zero. As an example, many grid-world and navigation domains have state uncertainty due to the fact that multiple locations look the same to the agent with respect to the available sensors. Once an observation is made in a domain, the set of locations the agent could be in is reduced to just the set which could provide such an observation. After multiple actions are taken, this set begins to shrink even further as the agent refines its knowledge about the world. A related structural feature that can be exploited in

many domains is fully observable variables in the state space. In these instances, the beliefs states over such a variable are always collapsed to a single belief in one state.

By building a DDN at runtime that only represents the set of reachable states and valid observations given the current state, the costs of inference in the DDN for action selection can be reduced. This approach is related to the Bayesion network minimization technique developed by Nicholson and Brady (Nicholson and Brady, 1994). A tradeoff is encountered, however, as the agent spends extra computation building the DDN. If the reachable states and observations are limited, the DDN building and planning processes can be performed in much less time than that required for planning with a full state-space, designer-built DDN.

Given a description of the variables needed in a DDN, a description of the states for each variable, and the transition and observation probabilities, it is possible to specify, for any current state of the world, all future states of the world that can be reached and the corresponding observations that could be seen, given a finite number of decisions are made.

The process for finding such states is implemented through dynamic programming, starting with the assumption that one action is taken. Given a current state of the world, a set of next possible states is defined for each action in the transition table. The set of all possible next states for the current decision is the union of the outcome states that are entered from all possible actions. The corresponding observations can be found at the same time, as they are tied to previous-state, action pairs or outcome states. For each timestep after the first, the set of reachable states is the union of one-step reachable states from the set of reachable states found for the previous timestep. Given a state set S and an action set A, in the worst case, every state could be reachable from all possible first states, giving an upper bound for every iteration of $0(|A| * |S|^2)$. Since each DDN represents a lookahead of a finite number of decisions

n, this process costs at most $O(n * |A| * |S|^2)$. A pseudo-code description of this algorithm can be found as Algorithm 2.

Algorithm 2 Computing Needed (Reachable) States and Observations

```
for all s ∈ States do
  for all a ∈ Actions do
    for all sNew ∈ destinationStates(s,a) do
      s.nextStates[0] ← s.nextStates[0]+ sNew
    end for
    for all o ∈ O(s,a) do
      s.nextObservations[0] ← s.nextObservations[0] + o
    end for
  end for
end for
for all s ∈ States do
  for i ← 1,2,... n-1 do {n is the number of decisions}
    for all sLast ∈ s.nextStates[i-1] do
      for all sNew ∈ sLast.nextStates[0] do
        s.nextStates[i] ← s.nextStates[i] + sNew
      end for
      for all o ∈ sLast.nextObservations[0] do
        s.nextObservations[i] ← s.nextObservations[i] + o
      end for
    end for
  end for
end for
```

This algorithm is performed before execution, so that an agent at runtime only needs to consider the set of reachable states and does not need to compute such states. The structure of the DDN is also known in advance and the structural representation does not change as the agent is executing. At runtime, the agent only needs to set the possible states in each variable and fill in the conditional and prior probability tables. Given a description for each timeslice of the states necessary from a given start state, an agent can dynamically build a DDN representing the needed state space using Algorithm 3 described in the next paragraph and presented later in pseudo-code.

Given a current belief state, the agent has beliefs that it is in some subset of the possible states of the world. This subset could range from the entire set of states (if it

has no knowledge about its state in the world) to a single state (if it knows exactly the state it is in). The DDN that the agent is constructing needs to represent all states the agent could possibly be in. Thus, the actual set of states and observations represented in each timeslice of the DDN is the union of the states and observations that can be reached in that timeslice from any of the states which the agent currently believes it could be in (has a belief greater than 0 for that state). The reachable states from a single current state have already been defined from the previous reachability analysis process, so the agent solely needs to determine their union. The agent then needs to compute all possible sets of table indexes (parent state to child state relationships) that are needed for each timeslice and add the appropriate entry to the conditional probability tables. Probabilities for transitions and observations for each of these states can be found in the transition and probability tables provided by the designer.

An alternative approach more tailored to specific situations was also developed. This algorithm is usable when a single variable representing a feature of the world is known to be fully observable, while other variables tend to be fairly distributed. In this approach, the agent designer provides the definitions of all possible future states that can be reached for the fully observed variable given the state of the fully observed variable. The other variables are represented by their entire state space, and the possible observations at each timeslice are precomputed as before. At runtime, since the agent is relying on the one fully observable feature to guide its construction of the DDN, the agent does not need to find the union of possible future states as in the previous algorithm. The needed parent entries can also be precomputed. This can reduce some of the computations required at runtime for building the decision network. If beliefs for all variables tend to collapse into a small set of states, however, the previous algorithm should be used. The fully-observable build algorithm is presented as Algorithm 4.

Algorithm 3 Runtime Construction of Dynamic Decision Network

```
for all n ∈ decisionNetwork.nodes do
  if n.type == Chance then
    possibleConfigurations ← {allCombinations(p, s) | p ∈ n.parents ∧ p.type ≠
    Action ∧ s ∈ p.states ∧ P(s) > 0 }  {allCombinations(p,s) generates the
    unfactored combination of states from a set of factored variables; in this case,
    only those combinations with probability > 0}
    for all pc ∈ possibleConfigurations do
      n.neededStates ← n.neededStates + precomputedNeeded(n, pc)
    end for
  end if
  if n.type == Action then
    for all a ∈ Actions do
      n.neededStates ← n.neededStates + a
    end for
  end if
end for
for all n ∈ decisionNetwork.nodes do
  if n.type == Chance then
    possibleConfigurations ← { allCombinations(p, s) | p ∈ n.parents ∧ s ∈ p.states
    ∧ P(s) > 0 }
    for all pc ∈ possibleConfigurations do
      for all s ∈ n.neededStates do
        n.probabilityTable.add(precomputed(P(pc,s)))
      end for
    end for
  end if
  if n.type == Utility then
    possibleConfigurations ← { allCombinations(p, s) | p ∈ n.parents ∧ s ∈ p.states
    ∧ P(s) > 0 }
    for all pc ∈ possibleConfigurations do
      n.utilityTable.add(precomputed(U(pc)) * n.discount)
    end for
  end if
end for
```

Algorithm 4 Dynamic Construction on Fully-Observable States

```
for all n ∈ decisionNetwork.nodes do
  if n.type == Chance then
    n.neededStates ← n.neededStates + precomputedNeeded(n, currentState)
  end if
  if n.type == Action then
    for all a ∈ Actions do
      n.neededStates ← n.neededStates + a
    end for
  end if
end for
for all n ∈ decisionNetwork.nodes do
  if n.type == Chance then
    possibleConfigurations ← precomputedConfigurations(n, currentState)
    for all pc ∈ possibleConfigurations do
      for all s ∈ n.neededStates do
        n.probabilityTable.add(precomputed(P(pc,s)))
      end for
    end for
  end if
  if n.type == Utility then
    possibleConfigurations ← precomputedConfigurations(n, currentState)
    for all pc ∈ possibleConfigurations do
      n.utilityTable.add(precomputed(U(pc)) * n.discount)
    end for
  end if
end for
```

4.4 Embedding Principled Behaviors

As detailed earlier, the environments that agents encounter in realistic problems often require that agents are capable of handling non-deterministic actions and problems in sensing. Markovian control mechanisms have been shown to be an effective approach in these types of environments. Since Markovian control mechanisms are based on maximization of expected utility, they can be viewed as agent-centric or selfish. When agents are acting in social environments, however, local actions can affect other agents and improvements in local utility do not always translate to improvements in utility at the social level. In cooperative domains, where agents must work together to achieve the social level goals, agents should act to ensure achievement of such goals, while also optimizing locally when possible. This idea can also be cast into a philosophical framework, where the agent has a set of principles that it uses to guide its actions, and those principles favor social goal achievement over local goal achievement. These principles need to be mapped to preferences held by the agent, which can then be mapped to utility functions.

In many domains, only local utility functions are defined a priori for the problem. Common examples are individual rewards for completion of tasks such as meeting a deadline or saving fuel. To be able to map social-level preferences into this same utility maximization framework, a new utility function needs to be developed for agents within a social group which takes the social goals into account. In Markovian environments, such a social utility function can be designed given the individual level Markovian problems (states, transitions, and rewards) and a list of social goal states. The social utility function developed is a strong-safety utility function which was inspired by the work of Ha and Musliner (Ha and Musliner, 2003). The strong-safety utility function ensures that an agent maximizing its utility under such a function is selecting actions that are probabilistically most likely to achieve social level goals.

This probabilistically most likely guarantee is possible because of the fact the agent is working in a Markovian environment. If the agent is in a situation where all of its actions are equally likely to move the towards the social level goals, it can locally optimize as desired.

4.4.1 Proposed Agent Philosophy

In Asimov's *Foundation and Empire* (Asimov, 1952), four laws of robotics are defined which Asimov believes should govern a robot's decision making progress. These four laws are defined as follows:

Law 0 - A robot may not injure humanity or, through inaction, allow humanity to come to harm.

Law 1 - A robot may not injure a human being or, through inaction, allow a human to come to harm.

Law 2 - A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

Law 3 - A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

While these rules can be interpreted as enforcing human dominance over robots, they also provide for a form of machine ethics. It has been suggested (Rose and Huhns, 2001; Rose et al., 2002) that a similar set of rules can be applied to independent agents that need to work together in groups and for which the designer needs to ensure the achievability of social level goals. In particular, there are two aspects of Asimov's laws that seem to be highly applicable to cooperative agents - a preference for supporting a higher purpose over the individual, and a preference for preventing harm over goal achievement.

An initial set of principles for cooperative agents loosely derived from Asimov's

laws are as follows:

Principle 1 - An agent shall not harm the societal mission through its actions or inactions.

Principle 2 - Except where it conflicts with the previous principles, an agent shall not harm itself.

Principle 3 - Except where it conflicts with the previous principles, an agent shall make rational progress toward social goals.

Principle 4 - Except where it conflicts with the previous principles, an agent shall make rational progress toward its own goals.

By acting under this type of philosophy, the actions of an agent should be steered in directions that are useful for accomplishment of global goals. Agents can perform actions that are useful locally, but only to the point where the local action does not interfere with the global effort. Take as an example a Mars rover agent on an exploration mission. Increasing the speed of the rover to meet a deadline for arrival is valid, as long as it doesn't damage the rover and make it useless later on in the mission. If meeting the deadline is critical to the mission success, however, the damage from its speedy travel may be warranted. Agents should also be willing to perform a sub-optimal or more costly action if it prevents harm at a higher level. This is similar to the classic mattress-in-the-road example. If an agent performs the extra action of removing a mattress that has fallen off a truck from the road, it costs that agent just a little more, but saves everyone else significantly down the line.

4.4.2 Generating A Utility Function

To develop an agent that reasons with the proposed principles, the agent designer must determine, for each agent, an appropriate utility function representing the preference function for states of the world as determined by the proposed principles. The

strong-safety utility function states the agent should always select the action that is most likely to achieve the social goal states. The problem can then be specified as finding a social goal achievement utility function which, when combined with the individual goal achievement utility function, brings about an agent whose rational actions are those that are most likely to ensure social goal achievement.

In theory, one would like to be able to arbitrarily set goal rewards, where the maximum utility is provided for states that achieve social and individual success, the next highest for social success, the next for individual success, and then finally a zero reward for neither social or individual success. Arbitrary selection of utility values does not always support the selection of the most useful social actions because of *expected utility*. As an example, consider, an agent that is deciding between two actions - one that has a 80% probability of moving to a desired individual goal state and 45% probability of moving to a desired social goal state, and another action which has a 40% probability of moving to a desired individual goal state and a 60% chance of reaching a desired social goal state. If the individual reward is 100 and the social reward is 200, then the rational choice is to take the first action, with an expected utility of 170, over the second action with expected utility of 160. However, if success is particularly determined by achievement of social goals, then the second action, which is more likely to move to the social achievement states, should be chosen. Even though the utility function weights social rewards over individual rewards, the weight is not enough to drive the agent to move towards social success. Essentially, the agent designer has not found an appropriate value function which assures the agent is putting the social interests ahead of selfish interests.

The methodology for finding such a social utility function requires several assumptions about the environment. First, the agent must be working in a fully observable, discrete state, Markovian environment. Secondly, one must be able to specify individ-

ual goal states and social goal states. While this precludes some types of Markovian environments where there are not clearly discernible end states, it is a reasonable assumption for many realistic, team-based tasks. It is also assumed that the individual goal achievement value function is known and that the transition functions for all states on all actions is known (no learning involved).

To determine an appropriate socially-oriented agent utility function, six steps are taken:

1) Determination of states in the agent's state space that indicate individual goal achievement.

2) Determination of states in the agent's state space that indicate social goal achievement.

3) Analysis of probability of accomplishing individual goal achievement.

4) Analysis of probability of accomplishing social goal achievement.

5) Calculation of appropriate social achievement utility function.

6) Determination of appropriate utility values for all states taking into account individual and social level achievement utility functions.

It is assumed that the states that define social goal achievement and individual goal achievement will be provided in the domain description. These states are marked as such and recorded to complete the first two steps in defining the social utility function. To ensure that agents value social success and prevention of harm over individual goal interests (to ensure agents always select actions that provide the highest probability of social level success), one must then consider, for each state, the possible outcomes of taking every action in that state and the probability of goal achievement. This process is implemented via MDP value iteration, which is a polynomial time algorithm. In the worst case, a single MDP process costs $O(|S|^2|A| + |S|^3)$ per iteration, with polynomial iterations. For the implemented algorithm, value iteration to convergence

is performed three times. Standard techniques for speeding up MDP value iteration, such as using a factored state representation, can be used in this process.

The first MDP process, implementing step 3, is performed for the individual goal level. A MDP is defined in which the value for states that are individual goal achievement states (specified in step 1 of our process) are are set to 1 and the states marked as absorbing, while all other states are given a reward of 0. By performing value iteration on this MDP and returning the Q function, one can find the likelihood of entering an individual goal achievement state from any state and for any action in the environment. By specifying the individual goal achievement state reward as 1, the expected utility of any other state is just the likelihood of entering one of the goal achievement states from the current state given that optimal actions are taken in the future.

For step 4, a second MDP is defined for which the value for states that are social goal achievement states are set to 1 and those states are marked as absorbing. All other states are given a value of 0. By performing value iteration on this MDP, one can find the likelihood of entering a social goal achievement state from any state and for any action in the environment given that optimal actions are taken in the future.

Now that one has the likelihoods for individual goal achievement and social goal achievement, one can move on to step 5 of the process. These likelihood values can be used with the individual goal achievement value function to determine the desired social achievement value function. This process is first demonstrated for an agent that has only two actions to select from, and then extended to demonstrate the process for an arbitrary number of actions.

Assume that an agent is currently in *StateA* and has two actions it can perform: *ActionOne* and *ActionTwo*. *ActionOne* leads the agents to one state from a set of possible states *StateSetA* and *ActionTwo* will put the agent in one of a second set of

states *StateSetB*. Assume that, for a given state, the two applications of value iteration described above return different actions as optimal: the individual achievement policy returns *ActionOne* and the social goal achievement policy returns *ActionTwo*. If the same action is returned, this state does not need to be considered any further as the individual goal achievement value function drives the agent to do what one desires for social goal achievement. Given two different actions, the agent designer's task is to determine what utility is required for social goal achievement to drive the agent to always select the action most useful in the social context. In this case, it is desired that the agent choose *ActionTwo*, as it is more likely to lead the agent to a social goal achievement state. Thus, for this given state and the returned optimal actions, the following inequality is required to hold:

$$ExpectedUtility(ActionTwo) >$$
$$ExpectedUtility(ActionOne)$$

Let PI be the probability of individual goal achievement, PS be the probability of social goal achievement, IR be the individual goal achievement reward, and SR be the social goal achievement reward. From the definition of expected utility and the assumption of additive individual and social rewards, this inequality is the same as:

$$(PI(ActionTwo) * IR + PS(ActionTwo) * SR) >$$
$$(PI(ActionOne) * IR + PS(ActionOne) * SR)$$

Let there be two components to the social reward SR: the engineered component, ESR, which is the utility to be derived to ensure selection of the highest likelihood social achievement action, and an additional reward component, ASR, which is due to preferences over social goal states. The inequality can now be represented as:

$$(PI(ActionTwo) * IR + PS(ActionTwo) * (ESR + ASR)) >$$
$$(PI(ActionOne) * IR + PS(ActionOne) * (ESR + ASR))$$

In the worst case, the ASR for the state entered from *ActionTwo* would be 0, while the ASR for the state entered from *ActionOne* would be maximal, determining that:

$$(PI(ActionTwo) * IR + PS(ActionTwo) * ESR) >$$
$$(PI(ActionOne) * IR + PS(ActionOne) * (ESR + ASR))$$

which is equivalent to:

$$(PS(ActionTwo) * ESR) - (PS(ActionOne) * (ESR + ASR)) >$$
$$(PI(ActionOne) - PI(ActionTwo)) * IR$$

To save space, define *PS(ActionTwo) - PS(ActionOne)* as *SocialDifference*. Also, define *IndividualDifference* as *PI(ActionOne) - PI(ActionTwo)*. Then, the above inequality is equal to:

$$(SocialDifference * ESR) - (PS(ActionOne) * ASR) >$$
$$IndividualDifference * IR$$

Finally, by setting IR as the maximum attainable individual goal achievement reward and ASR as the maximum social goal achievement reward from preferences over goal achievement states and rearranging, the desired inequality to satisfy to obtain the engineered social goal achievement reward component that will lead the agent to select socially useful actions from a given state is:

$$ESR > \frac{((IndividualDifference * IR) + (PS(ActionOne) * ASR))}{SocialDifference}$$

For the given state of interest, this computed value of ESR, if applied, will ensure the agent selects the action most likely to achieve the social goals. However, this value may not be appropriate over the entire state space to bring about the desired results. Thus, this value needs to computed for all states where the actions that

bring about the maximum likelihood of individual achievement and social achievement (from steps 3 and 4) are different, with the maximum value across all states selected as the minimum desired social goal achievement reward.

For an example of this process, assume that an individual goal achievement reward is 100, that there are no preferences over goal achievement states so the ASR is 0, and that *ActionOne* has an 80% chance of individual goal success and 30% chance of social goal success, and *ActionTwo* has a 40% chance of individual success and 60% chance of social success. The social achievement reward can be calculated as:

$$ESR > \frac{((0.80 - 0.40) * 100) + (0.30 * 0)}{0.60 - 0.30} = 133$$

To extend to more than two actions, one can no longer determine the needed utility from just the action that has the highest social goal achievement likelihood and the action that has the highest individual goal achievement likelihood. Instead, one must consider the maximum likelihood of individual goal achievement (call this action *ActionOne*) and both the second highest (*ActionTwo*) and highest (*ActionThree*) social goal achievement likelihoods. The final social reward needs to ensure the maximum likelihood social achievement action is taken. If the second highest likelihood is not taken into account, another action may generate the overall highest expected utility after combining individual and social rewards. Consider the three actions *ActionA*, *ActionB*, and *ActionC*. Assume that the individual goal achievement reward is 100, that all social goal achievement states are equally preferred so ASR = 0, and that *ActionA* has an 70% chance of individual goal success and 40% chance of social goal success, *ActionB* has a 60% chance of individual success and 70% chance of social success, and *ActionC* has a 40% chance of individual success and 80% chance of social success. The social achievement reward using the old equation is calculated as:

$$ESR > \frac{((0.70 - 0.40) * 100) + (0.40 * 0)}{0.80 - 0.40} = 75$$

Using a social reward of 75.1 (just higher than the threshold 75), *ActionA* has an expected utility of 100.04, *ActionB* has an expected utility of 112.57, and *Action C* has an expected utility of 100.08. Thus *ActionB* would be chosen. *ActionB* is not the most likely action to achieve the social goals however.

By using the action with the second highest probability of social achievement, the rewards returned are guaranteed to select the maximum likelihood action. ESR is calculated for more than two actions according to

$$ESR > \frac{((PI(ActionOne) - PI(ActionThree)) * IR) + (PS(ActionTwo) * ASR)}{(PS(ActionThree) - PS(ActionTwo))}$$

For the previous example, the threshold value is now computed as

$$ESR > \frac{((0.70 - 0.40) * 100) + (0.70 * 0)}{0.80 - 0.70} = 300.$$

and the computed expected utilities using a reward of 301 are *ActionA* 190.4, *ActionB* 270.7, and *ActionC* as 280.8.

A proof of the fact that only the maximum likelihood individual achievement action and the two highest likelihood social achievement actions are needed in this computation is as follows:

To guarantee selection of the maximum likelihood social achievement action, the expected utility of that action must be higher than any other available action. As previously stated, this requirement can be written as the following inequality, where MLSAction represents the maximum likelihood social achievement action and AnyOtherAction represents any other action:

$$ExpectedUtility(MLSAction) >$$

$$ExpectedUtility(AnyOtherAction)$$

This inequality is equivalent to the following inequality under the definition of utility for the problems of interest (additive individual and social utility):

$$(PI(MLSAction) * IR + PS(MLSAction) * SR) >$$

$$(PI(AnyOtherAction) * IR + PS(AnyOtherAction) * SR)$$

Taking preferences over social goal states into account, this inequality is then written as:

$$(PI(MLSAction) * IR + PS(MLSAction) * (ESR + ASR)) >$$
$$(PI(AnyOtherAction) * IR + PS(AnyOtherAction) * (ESR + ASR))$$

A potentially overestimated but guaranteed upper bound for the value of the right hand side of this inequality is derived from maximizing each of the additive components. First allow ASR for the MLSAction on the left hand side to be zero, while ASR for the right hand side to be the maximal utility for a goal achievement state. The maximum value contributed for the individual achievement component will be from the action that has the highest probability of individual goal achievement, since the value IR is fixed to the maximum attainable individual achievement reward. Similarly, the maximum value for the social component of the right hand side is generated from the action that has the highest probability of social goal achievement. Since the true maximum social achievement likelihood is already being used on the left hand side (it is the action the designer wants the agent to take), only the second highest likelihood action could contribute maximally to to this value. Labeling the actions as before, with the maximum likelihood of individual goal achievement action called *ActionOne*, the second highest social goal achievement action called *ActionTwo* and the highest social goal achievement action called *ActionThree*, the inequality with a maximal right hand side is derived:

$$(PI(ActionThree) * IR + PS(ActionThree) * ESR) >$$
$$(PI(ActionOne) * IR + PS(ActionTwo) * ESR + PS(ActionTwo) * ASR)$$

which is equivalent to the previously defined inequality for computing SR:

$$ESR > \frac{((PI(ActionOne) - PI(ActionThree)) * IR) + (PS(ActionTwo) * ASR)}{(PS(ActionThree) - PS(ActionTwo))}$$

Any other choice of probabilities will either be less likely to achieve individual goals or less likely to achieve social goals. By maximizing the right hand side and solving the inequality, the generated utility function will always select the action with the maximum likelihood of social goal achievement in a given state.

To ensure that an agent selects actions that correspond with the highest likelihood of social goal achievement, an agent could use just the Q function derived in step 4. This Q function explicitly represents the probabilities of social achievement and could save the designer the time of going through the processes in step 5 and 6. The use of the entire process is justified by the fact that the social goal achievement states, though equivalent in the sense that they are goal states, are not necessarily equivalent in preference. Thus, while step 4 determines the actions that are most likely to move the agents into social goal achievement states, steps 5 and 6 are used to take into account preferences among social goal achievement states when two or more such states have the same probability of being entered. The same differentiation is also made at the individual achievement level during steps 5 and 6. The use of the MDP method in step 6 takes advantage of a concise and fast algorithm to merge the social and individual achievement values functions.

The agent designer is now at step 6, in which the final value function is found for each state in the environment by integrating the individual goal achievement rewards (specified in the problem) and the social level rewards that were just calculated. A MDP is defined in which states that are individual goal achievement states are given their true individual achievement reward, states that are social goal achievement states are given the basis social goal achievement reward plus an appropriate offset representing the agents preferences over the social goal states, states that are both are given the sum, and all other states are given a reward of 0. By performing value iteration on this MDP and returning the Q function, one has the expected utility of

taking any action in any state for the problem. Maximization of this expected utility leads to guaranteed selection of the actions probabilistically most likely to lead to goal achievement, while also allowing for individual optimization when possible.

The proposed model works towards, but does not guarantee, a solution to the multiagent coordination problem when communication is not allowed. As an example, consider the problem of robots moving large items, where the large items require multiple agents to synchronize lift and move actions. If the social goal is defined as successful movement of the large items, individual policies, since they are directed towards achievement of social goals, should lead the agents to move towards the large items and lift. Given that agents are performing distributed planning in the proposed model, however, there is no mechanism to ensure that multiple agents select the same individual policy when there are multiple individual policies that are equally socially useful. Just as with humans, once two agents have an item lifted, they may miscoordinate on which direction to move if there are two routes to take that are equally optimal, at the social level, in reaching the drop-off point for the item. If the actions are truly equal in utility to both agents (there are no individual preferences for one action over another), such problems can be easily worked around if there are coordination mechanisms added to the system. One such mechanism is to enforce a system-wide ordering over actions so that if two actions are truly equally useful, the agents consult the ordering to determine which one to take. Boutilier discusses this ordering mechanism, along with others, in his work on MMDPs (Boutilier, 1999). When individual utilities are taken into account, an agent may prefer one of the multiple socially-optimal actions for personal reasons. In this case, more advanced coordination mechanisms outside of the MDP framework, such as negotiation between agents, will be required.

4.4.3 Embedding Principles in an Agent

Once a complete MDP value function is generated over the agent's state space, the value function can be used to allow an agent to make decisions. In fully observable domains, the optimal policy based on the utility function is found by taking the action with the maximum expected utility at each state. The agent could then select actions to perform by looking up the appropriate action from a policy table once it determines its current state. The value function should also be usable as a heuristic for partially-observable variations of a given problem under the architecture described within this work, as agents under the described architecture work in partially observable environments by projecting forward a limited number of steps using the fully-observable value function as an estimate of the true POMDP value function.

Chapter 5

Architectural Design Issues

The algorithms introduced in the previous chapter are general techniques that are applicable to any DDN-based algorithms for POMDPs. This chapter describes five parameters for those algorithms that an agent designer can optimize for domains of interest. These parameters affect both the quality of the policies that are generated by the developed algorithms as well as the computational requirements. A small number of experiments evaluating the effects of these parameters have been run and the results are presented in this chapter and in Chapter 6. These results are intended to provide support for the parameter settings used in the experiments validating the developed algorithms as well as provide guidance for design choices when used in other domains.

5.1 Observations and Lookahead

The complexity of planning with dynamic decision networks is directly related to the size of the largest clique (effectively, the largest joint probability table) that must be generated. The size of this table grows exponential in the action and observation space, dependent on the number of timesteps that are being considered. Essentially,

planning with a DDN is the same as projecting forward all possible action and observation sequences and selecting the one that provides the highest utility to the agent.

If an agent is continuously replanning at each timestep and the optimal observation for the first timestep is not dependent on observations in the future, the exponential factor for the observations can be reduced to just the cardinality of the observation space in the formula for DDN complexity. Since the first timeslice represents the current state of the world, this model says that the information gained from the first action (the observation in the second timeslice) should be used to guide the selection of the second action (the second decision node). This greedy approximation is often called a myopic approach, as it can only take into account a single observation in computing value of information. In domains where there is not a dependence between multiple tests, a myopic approach is reasonable. Since the agent is always replanning after each action, the agent is always planning with as much evidence as possible and can always consider information gathering actions as the next step. Second, in many domains, the inherent cost of performing tests does not decrease if delayed. In such a context, placing the observation taken early in the decision network provides for gathering of information as early as possible. Additional computation can be saved since the observation nodes for all future timeslices can also be left out of the DDN. While the myopic assumption can provide significant reductions in computation time, it also can affect the quality of the policy generated by a POMDP solver.

The number of timesteps that are being considered is another key contributor to the size of the probability table for the largest clique, as it provides the exponential factor. Accordingly, dynamic decision network algorithms can save computation time by reducing the number of states for which lookahead is performed. In particular, the quality of the value function estimate is essential in determining the amount of

lookahead that is required. If this estimate is the true value function, the agent only needs to look ahead one step. If the estimate is different from the true value function, a larger lookahead is required to generate useful policies. Russell and Norvig (Russell and Norvig, 2002) state that a short lookahead is usually enough to generate high quality solutions, particularly when the discount rate is not extremely close to 1.

The effects of observation reduction and lookahead steps on required computation time and solution quality were measured on a set of POMDP test domains to examine tradeoffs in the described architecture. These results are summarized in Tables 5.1-5.4. The first and third quartile endpoints for each statistic are also presented in brackets where appropriate. These allow for an idea of both the centering and the variance of the data, as they represent the bounds for the middle 50% of the data. Histograms representing the collected data are presented in Appendices A and B. In addition to the planning time and reward statistics, the size of the largest clique generated from the Hugin decision network compilation algorithm is shown in the tables. This metric allows one to examine how the different levels of observation and lookahead affect the probability propagation datastructures. The size of the largest clique is directly correlated with computation time. A Mann-Whitney test was used to test the hypothesis that the planning time distributions or reward distributions for different algorithms are from the same population. If the Mann-Whitney test returns a p-value < 0.05 (the chosen critical point for this work), the presented times or rewards can be considered statistically significantly different. For each test domain, three different agent control mechanisms were used. Each control mechanism represented all possible actions and all possible states in the domain. The first type of agent control used was a one-step lookahead network. In this mechanism, no observations were taken into account as only one forward action is considered. The second control mechanism was the use of a multi-step lookahead dynamic decision network, only

taking into account the first possible observation. The third type of agent control used was a multi-step lookahead dynamic decision network which takes into account all observation sequences possible for the number of steps of lookahead used. For each control mechanism, the average computation time required for planning a single action and the average reward earned per trial were computed. An entry of *NC* indicates that the domain could not be compiled within the Hugin system under the memory constraints of the system (1024 megabytes of memory were allocated to the process).

The Mann-Whitney statistical test results consider all differences in planning times to be statistically significant ($p < 2.2e\text{-}16$ for all comparisons on all problems). For all problems except the parts problem, only the one step lookahead reward was considered statistically significant from the other reward distributions ($p < 4.103e\text{-}12$ in all cases). The null hypothesis that there are no reward differences between other approaches cannot be rejected given the results returned by the Mann-Whitney test. For the parts problem, where there are only +1 and -1 rewards that are earned, a test of proportions (proportion of +1 rewards earned, which is equivalent to the proportion of successful of shipments) was used to compare the alternative planning techniques. This test returned p-values above the critical threshold for all comparisions, meaning the null hypothesis that the rewards generated by the different algorithms are from the same population can't be rejected for the parts problem.

5.2 Cache Size Limits

If agents are planning at runtime and storing actions in a cache, as described in the previous chapter, the agent designer needs to consider the tradeoff between memory space required for the cache and computation time. If the cache is unlimited in size, the agent can effectively store an action for every possible belief state that it encoun-

Table 5.1: Effects of Observations and Lookahead - Parts Problem

Lookahead	Max Clique	Avg Time	Avg Reward
1-Step	64	1.30 [1,1]	0.41
2-Step	128	2.08 [2,2]	0.6
3-Step, One Obs	512	3.11 [3,3]	0.63
3-Step, All Obs	1024	3.65 [3,4]	0.57
4-Step, One Obs	2048	5.45 [5,6]	0.47
4-Step, All Obs	8192	14.50 [14,15]	0.6

Table 5.2: Effects of Observations and Lookahead - Small Cheese Taxi Problem

Lookahead	Max Clique	Avg Time	Avg Reward
1-Step	2541	3.64 [3,4]	4.5 [3,8]
2-Step	16170	19.29 [19,19]	8.36 [8,10]
3-Step, One Obs	113190	227.44 [165,276]	8.17 [8,10]
3-Step, All Obs	1131900	4702.57 [4292,5108]	8.59 [8,10]

Table 5.3: Effects of Observations and Lookahead - Hallway Problem

Lookahead	Max Clique	Avg Time	Avg Reward
1-Step	18000	9.19 [8,9]	-53.91 [-100,-12]
2-Step	31500	43.40 [41,42]	-16.41 [-21.5,-10]
3-Step, One Obs	157500	275.65 [190,316]	-16.01 [-21,-10]
3-Step, All Obs	3307500	22038.09 [21861,24206]	-16.00 [-21,10]

Table 5.4: Effects of Observations and Lookahead - Large Cheese Taxi Problem

Lookahead	Max Clique	Avg Time	Avg Reward
1-Step	63000	35.75 [34,36]	3.18 [1,6]
2-Step	426300	981.11 [758,1179]	7.98 [6,10]
3-Step, One Obs	2984100	13951.93 [13130,15107]	8.40 [7,11]
3-Step, All Obs	86538900	NC	NC (NA)

ters during execution. By storing all previously seen belief points, the computation time required in the future will be significantly reduced as the agent re-encounters the same states. In realistic deployments, however, there will likely be a constrained amount of memory available to the agent which will force recomputation of optimal actions for some belief points. This problem is partially mitigated by the algorithm in use for plan caching and by features of many domains. The implemented plan caching algorithm updates an entry timestamp whenever the cache is accessed. When the cache has reached its upper limit on entries, the least recently used (LRU) belief point is removed from the cache. By using a LRU replacement algorithm, the most commonly used belief points are maintained in the cache. Since these points are commonly visited and are kept in the cache, future runs will still see significant speedup. The dynamics of the problem environment also play a role in the effectiveness of the cache. Those environments that are fairly deterministic and localize well, such as navigation domains, consistently move through the same parts of the belief space and don't overfill the cache.

An evaluation of the effect of cache size on computation times was performed on several POMDP test domains. Starting with a cache size of zero, increasingly larger cache sizes were considered. The average computation time required for planning individual actions was computed for each cache size. These results are shown in Chapter 6 in the section discussing evaluation of caching algorithms.

5.3 Grid Cache Threshold

If an agent is planning with a grid-based cache, only a subset of the encountered belief points are stored in the cache. This can reduce the memory requirements compared to an exact-match cache, particularly if belief points tend to be located in close regions of the belief space. The threshold used for grid-based caching has

both a direct effect on the size of the cache that is required and the quality of the policies that are generated. Threshold values can be between 0 and 2, representing the maximum shift in the probability distribution allowed when comparing belief points. A threshold of zero is equivalent to an exact-match cache. Accordingly, every belief point encountered will need to be stored in the cache. A zero threshold cache also provides the highest quality policy available when using a grid-based cache as the optimal action will be computed using the dynamic decision network for any new belief state. A threshold of two allows for a complete shift in beliefs, and only requires one entry in the cache. Thresholds greater than zero have the potential of introducing errors into the policies as the optimal actions may differ between the center point and those points within the threshold region.

An analysis of the tradeoffs between the grid threshold required computation time and solution quality was performed on several POMDP test domains. Thresholds close to zero were the focus of this analysis as they are the most likely to maintain solution quality. For each grid-threshold, the average computation time required for planning individual actions and the average reward earned per trial were computed. These results are shown in Chapter 6 in the evaluation of the grid-cache algorithm.

5.4 Hierarchical Design

When developing hierarchies of dynamic decision networks, the abstractions selected can affect both the quality of policies that are generated and the computational requirements. Computational requirements are affected as different abstractions may make different portions of the state space irrelevant or equivalent. The depth of the tree representing the hierarchy also plays a role in the computational costs since a different dynamic decision network must be evaluated at each level of the hierarchy. The quality of a generated policy can be affected by the loss of information due to ab-

straction. Correctness of cost estimates for abstract actions also play a role in policy quality, as an incorrect estimate can lead an agent to make incorrect choices. Automated techniques for evaluating appropriate abstractions have not been extensively developed, and this problem is not the focus of the current work. The hierarchies used in experiments in this work are those seen in the literature or that appear to be most natural for the problem. The results of a comparison of two different hierarchies for one of the experimental domains are presented in Chapter 6.

Chapter 6

Experimental Analysis

In this chapter, the proposed algorithms are compared through tests on the sample domains presented in Chapter 3. The primary perforance measures of interest are average planning time per action, a measure of the computational requirements of an algorithm, and average reward earned, a measure of the quality of the policies that are generated. By performing multiple simulations for a given domain, distributions of planning times and rewards are generated. Histograms representing the collected raw planning time data are presented in Appendix A, while histograms for the reward data are presented in Appendix B. These distributions are non-normal. To compare two such distributions to determine if a technique for structural or run-time knowledge exploitation produces a statistically significant difference in planning time or reward, a non-parametric test, the Mann-Whitney (2-sample Wilcoxon rank sum) test was used. The null hypothesis for this test is that the two sets of samples are drawn from the same population. The test computes whether the overlap between the distributions is less than expected by chance given the null hypothesis. Significant differences according to this test statistic are highlighted in the text, with a critical value of 0.05 selected. Large numbers of ties can affect the Mann-Whitney statistic, so the analysis of significance for caching tests, where multiple zero planning times

are common, should be read with more caution. The first quartile and third quartile endpoints are also presented in brackets for each statistic computed for the DDN algorithms. These values provide both an idea of the center of the data and of the variability of the data.

6.1 Exploiting Hierarchical Structure

The first problem tested was the parts planning problem (Draper et al., 1993). This problem is interesting because it is simple enough that an optimal policy can be found by an exact POMDP solver (Cassandra, 1999a). In addition, this problem has also been evaluated by Pineau using the HPOMDP algorithm (Pineau and Thrun, 2002). The hierarchical decomposition used in this problem is to have a *Root* action that is an abstraction over the primitive actions *Inspect*, *Reject*, and the abstract action *Process*. *Process* is an abstraction over the primitive actions *Paint* and *Ship*. In the HDDN formulation, both the *Root* and *Process* decision networks used two action lookahead with an observation link between the first observation and the second action. The optimal policy, as found by the exact solver, consists of performing a single *Inspect* action. If this action returns *BL (blemished)*, the part is rejected, otherwise the part is painted and shipped. The policy generated by both HPOMDP and the developed HDDN algorithm is suboptimal, but only slightly. The policy generated from both hierarchical planners first performs a single *Inspect* action. If the part is blemished, it is rejected. Otherwise, the part is painted twice and then shipped. Since painting is a zero cost action, this policy is as good as the optimal policy in terms of utility. The extra action requires more time, however, to execute. The policy generated by a flat DDN using three steps of lookahead and one observation also performs an extra *Paint* action. A flat DDN with one step lookahead is the worst performer. This DDN randomly chooses between *Inspect* and *Reject* as the first action, leading the

mechanism to arbitrarily discard a much larger portion of parts than appropriate. For each DDN-based mechanism, 75 trials (execution from an initial state to an end state, consisting of multiple actions) were used to compute average per-action planning times.

Pre-execution and per-action planning times are presented in Table 6.1 for the Parts problem. A description of the policy developed by each mechanism is also presented. POMDP and HPOMDP are pre-execution policy generators. The entire value iteration process must be executed before the agent can begin execution. The DDN-based mechanisms are run-time planners. A comparison between these two types of approaches allows for an analysis of the cases where a good policy takes extraordinarily long times to develop using an exact algorithm. The per-timestep planning time is also important to provide a measure of the reactiveness of an agent if operating in the real world. The time presented for POMDP and HPOMDP is the time reported by Pineau, executed on a different hardware platform than the DDN based mechanisms. Pre-execution times for the DDN based approaches are the computation times required to generate MDP cost estimates using an available MDP solver (Grinkewitz, 2001). These times are approximate, taken from a small set of runs with times varying somewhat dependent on the load on the computer. All DDN pre-execution times were less than a second, so small variations should not affect the comparison versus pre-execution policy generators. Table 6.2 represents the amortized cost per action for POMDP, HPOMDP, and HDDN over the 75 trials that were performed for the parts problem. Since POMDP and HPOMDP are based on pre-execution, the amortized cost will approach zero over a large number of trials. Because of the low per-action costs of the HDDN approach, however, the number of trials required for POMDP and HPOMDP to reach an amortized cost equivalent to that of the HDDN approach is high. The Mann-Whitney test indicates that there is

Table 6.1: Hierarchical Approximations to Parts Problem

Control	Pre-Execution	Per Action Execution	Policy
POMDP	19.45 s	0 ms	Optimal
HPOMDP	5.84 s	0 ms	+1 Paint
HDDN	4 ms	2.53 ms [2,3]	+1 Paint
Flat DDN [3,1]	7 ms	3.11 ms [2,3]	+1 Paint
Flat DDN [1,0]	7 ms	1.30 ms [1,1]	Random Initial

Table 6.2: Amortized Planning Costs for Parts Problem - 75 trials

Control	Amortized Cost Per Action
POMDP	66.16 ms
HPOMDP	19.86 ms
HDDN	2.54 ms
Flat DDN [3,1]	3.13 ms

a statistically significant difference ($p < 2.2e\text{-}16$) between the distribution of planning times for the HDDN as compared to both the flat three-step, one-observation DDN approach and the flat one-step DDN approach.

The cheese taxi problem (Pineau and Thrun, 2002) was originally tested by Pineau for the HPOMDP algorithm. The initial hierarchical decomposition used by both HPOMDP and the developed algorithm consists of a top-level *Root* action that is defined over two abstract actions, *Get* and *Put*. The *Get* abstract action is similarly defined over the abstract action *NavigateS10* and the primitive action *Pickup*, while the *Put* abstract action is defined over the abstract actions *NavigateS0*, *NavigateS4*, and *Query*. Finally, each of the navigate abstract actions are defined over the primitive actions *North*, *South*, *East*, and *West*. Pseudo-rewards of zero cost for actions in goal states, such as for being in state *0* for *NavigateS0*, were used as appropriate and along the lines of the rewards used by Pineau.

Although using distinct navigate actions for each destination allows for significant reductions in state space for these actions, an additional cost is incurred by the

increased number of actions that must be reasoned over. Since there are only three destinations in this domain, this does not present a major problem. However, in larger domains, the cost of reasoning over a navigation action for each destination becomes prohibitive.

Along these lines of reasoning, an alternative hierarchy was developed in which only a single *Navigate* action exists in the hierarchy. This *Navigate* action reasons over all taxi locations and destinations. In this decomposition, the *Root* and *Get* abstract actions are modeled exactly as they are in the original decomposition. Since the decision of which destination to navigate towards is now made within the *Navigate* task, the *Query* action is moved from the *Put* abstract action into the *Navigate* abstract action, leaving the *Put* abstract action to reason only over *Navigate* and *Putdown*.

In the HDDN formulation, the *Root* and *Get* decision networks used two action lookahead with an observation link between the first observation and the second action, while the *Put* and *Navigate* decision networks used three action lookahead with a single observation link between the first observation and the second action.

The cheese taxi problem is large enough that a true optimal policy cannot be found by an exact POMDP solver. To evaluate the quality of the developed planning algorithm on this problem, two hundred simulations were executed. In each simulation, starting and destination locations were chosen according to the world model. Pineau has similar results for the HPOMDP algorithm. A comparison of average reward and average per-action planning time is shown in Table 6.3 below. Table 6.4 presents the amortized planning costs over the 200 trials run for the small cheese taxi problem. In this problem, the POMDP algorithm failed to converge and was stopped by Pineau after 24 hours and completion of five iterations of computation. To examine accrued rewards in this case, the last completed step of planning was

Table 6.3: Hierarchical Approximations to Small Cheese Taxi Problem

Control	Pre-Execution	Per Action Execution	Average Reward
POMDP	∼100000 s	0 ms	4 (NA)
HPOMDP (Multi Nav)	∼1000 s	0 ms	8 (NA)
HDDN (Multi Nav)	60 ms	41.04 ms [17,65]	7.43 [8,10]
HDDN (Single Nav)	60 ms	82.48 ms [78,86]	6.98 [8,10]
Flat DDN [3,1]	30 ms	227.44 ms [165,276]	8.17 [8,10]
Flat DDN [1,0]	30 ms	3.64 ms [3,4]	4.5 [3,8]

used to guide the agents. The increase in execution time planning between the two HDDN approaches is directly attributable to the more complex *Navigate* task. Along with the comparisons to POMDP and HPOMDP, the HDDN approach was evaluated relative to two other dynamic decision network based approaches - a flat DDN representation of the problem and a flat DDN with one-step lookahead. A flat DDN should not lose any optimality caused by abstractions, but may require more computation time due to modeling of larger state, observation, and action spaces in the flat DDN. The times for POMDP and HPOMDP are those presented by Pineau (Pineau and Thrun, 2002), computed on a different hardware platform. The distributions of planning times for the different HDDN tests are all singificantly different ($p < 2.2e\text{-}16$ for each combination of distributions). The Mann-Whitney test demonstrated a significant difference in reward distributions only for comparisons of the flat one-step lookahead HDDN approach with the alternative HDDN approaches ($p < 2.2e\text{-}16$). p-values for other distribution comparisons were all greater than 0.1.

The large cheese taxi domain was also used to compare the HDDN algorithm versus other hierarchical and flat approaches. For hierarchical approaches, three abstractions were used. The *Taxi* abstract task was an abstraction over the actions *Get* and *Put*. *Get* was an abstraction over the primitive action *Pickup* and the abstract action *Navigate*. Similarly, *Put* was an abstraction over the primitive action

Table 6.4: Amortized Planning Costs for Small Cheese Taxi Problem - 200 trials

Control	Amortized Cost Per Action
POMDP	35945.36 ms
HPOMDP (Multi Nav)	359.45 ms
HDDN (Multi Nav)	41.06 ms
HDDN (Single Nav)	82.51 ms
Flat DDN [3,1]	227.44 ms

Putdown and abstract action *Navigate*. Finally, *Navigate* was an abstract action over the five primitive actions *North*, *South*, *East*, *West*, and *Query*. In the HDDN formulation, the *Root* and *Get* decision networks used two action lookahead with an observation link between the first observation and the second action, while the *Put* and *Navigate* decision networks used three action lookahead with a single observation link between the first observation and the second action. Like the Cheese Taxi domain, this problem was also too large to generate the true optimal policy with an exact POMDP solver. Neither POMDP or HPOMDP had generated a usable policy after 41 hours of work and were terminated. Since the policies generated were unusable, the average reward results for these algorithms is labeled as Not Available *(NA)*. In the DDN-based tests, 250 simulations were run. Start and destination locations were selected according to the probability distributions defined in the world model during simulations. Tables 6.5 represents the pre-execution planning time, per-action average planning time, and average reward for the Large Cheese Taxi problem, while Table 6.6 presents the amortized planning costs for the problem. All planning time distributions for the HDDN approaches are significantly different per the Mann-Whitney test ($p < 2.2e\text{-}16$). The reward distributions for the HDDN approach and flat, three-step lookahead approach were found to be significantly better in performance than the one-step lookahead DDN ($p < 2.2e\text{-}16$), The p-value for the test between the HDDN approach and the three-step lookahead DDN was 0.1220, which is high enough to not

Table 6.5: Hierarchical Approximations to Large Cheese Taxi Problem

Control	Pre-Execution	Per Action Execution	Average Reward
POMDP	$\gg$ 41 hrs	0 ms	NA
HPOMDP	$\gg$ 41 hrs	0 ms	NA
HDDN	130 ms	4671.02 ms [4103,5951]	7.91 [6,10]
Flat DDN [3,1]	230 ms	13951.93 ms [13130,15107]	8.40 [7,11]
Flat DDN [1,0]	230 ms	35.75 ms [34,36]	3.18 [1,6]

Table 6.6: Amortized Planning Costs for Large Cheese Taxi Problem - 250 trials

Control	Amortized Cost Per Action
POMDP, HPOMDP	$>$ 43437.32 ms
HDDN	4671.06 ms
Flat DDN [3,1]	13952 ms

allow rejection of the null hypothesis that the rewards are from the same population.

The fourth problem on which the HDDN algorithm was tested was the Twenty Questions domain (Pineau and Thrun, 2002). The hierarchical representation of this problem begins with a *TwentyQuestions* task, which is an abstract action over three primitive actions: *askAnimal*, *askVegetable*, and *askMineral*, and three corresponding abstract actions: *subtaskAnimal*, *subtaskVegetable*, and *subtaskMineral*. *subtaskAnimal* is an abstract action over the primitive tasks of *askWhite*, *askRed*, *askBrown*, *askHard*, *guessMonkey*, *guessRabbit*, and *guessRobin*. *subtaskMineral* is an abstract action over the primitive actions of *askWhite*, *askRed*, *guessMarble*, *guessRuby*, and *guessCoal*. *subtaskVegetable* serves to differentiate fruits and vegetables. It is an abstraction over the primitive action of *askFruit* and two abstract actions, *subtaskFruit* and *subtaskRealVegetable*. *subtaskFruit* is an abstraction over the primitive actions *askWhite*, *askRed*, *askHard*, *guessTomato*, *guessApple*, and *guessBanana*. Finally, *subtaskRealVegetable* is an abstraction over the primitive actions of *askWhite*, *askBrown*, *askHard*, *guessPotato*, *guessMushroom*, and *guessCarrot*. In the HDDN formulation,

Table 6.7: Hierarchical Approximations to Twenty Questions Problem

Control	Pre-Execution	Per Action Execution	Average Reward
POMDP	∼100000 s	0 ms	∼-11 (NA)
HPOMDP	∼1000 s	0 ms	∼-18 (NA)
HDDN	12 ms	114.73 ms [72,152]	-17.54 [-29,-3]
Flat DDN [3,1]	0ms	488.22 ms [331,595]	-100 [-100,-100]
Flat DDN [1,0]	0ms	3.79 ms [3,4]	-100 [-100,-100]

Table 6.8: Amortized Planning Costs for Twenty Questions Problem - 250 trials

Control	Amortized Cost Per Action
POMDP	24606.30 ms
HPOMDP	246.06 ms
HDDN	113.90 ms
Flat DDN [3,1]	488.22 ms

all decision networks used four action lookahead with a single observation link between the first observation and the second action.

A comparison of required per-action planning time and average reward is shown in Table 6.7, while Table 6.8 presents amortized planning costs over the set of trials run. The times presented for POMDP and HPOMDP policy generation are the times reported by Pineau(Pineau and Thrun, 2002), computed on a different hardware platform. In the HDDN-based tests, 250 simulations were run to compute average planning times and average rewards. For the flat DDN tests, only 75 simulations were recorded, as the agent never did better than just randomly asking questions. All planning time distributions were found to be significantly different per the Mann-Whitney test ($p < 2.2e\text{-}16$ in all cases). The Mann-Whitney test was not run to compare reward distributions due to the extremely poor performance of the flat DDN approaches.

These tests provide evidence for the assertion that the HDDN approach is a viable technique for run-time planning in POMDP environments. In each domain tested,

the quality of the policies generated by the HDDN algorithm are comparable to that returned by the HPOMDP algorithm. Given this, it is believed that the quality of the policy is limited by the abstractions in these cases, instead of by other potential factors such as limited lookahead or poor reward estimates. Additionally, the pre-execution planning time is limited for all DDN approaches as only a fully-observable MDP solution needs to be computed. This compares favorably to the HPOMDP and exact POMDP approaches. Most importantly, there are domains, such as the Large Cheese Taxi problem, where the exact and HPOMDP approaches were not able to find an executable policy within a reasonable time period. The HDDN approach on the Large Cheese Taxi problem was always able to find a solution during its simulation trials. The HDDN approach also compares favorably with the multi-step flat DDN algorithm. Per action planning times are reduced significantly, while plan quality is still relatively high.

6.2 Exploiting Limited Belief States

The effects of adding a standard plan cache were evaluated in several of the experimental domains. Since the use of a standard plan cache does not affect the quality of the plans generated, computation time was the key metric examined. Computation time is presented in this section as the average planning time required for a single action (with multiple actions making up a complete trial). Since the cache is keyed on the belief state of the agent, the size of the cache can also provide a measure of the range of belief states that an agent encounters. A third metric, directly related to both the computation time needed per action for the agent and the range of the belief space, is the time required to saturate the cache with the needed belief-action pairs. To allow for an easy comparison between alternative exact cache techniques, the trials used for a given problem preserve start states, transitions, and observations across

simulations for different techniques. This ensures that the agents encounter the same parts of the state space when using each different caching implementation. Identical use of start states, transitions, and observations is possible since exact caching does not affect the policy that is generated (agents will always select the same action in a particular belief state).

Table 6.9: Effects of Adding a Plan Cache on Required Planning Time - Cheese Taxi, Hierarchical Control

Available Cache Size	Average Per-Action Time
None	82.55 ms [78,85]
500	1.51 ms [0,1]

Table 6.10: Effects of Adding a Plan Cache on Required Planning Time - Large Cheese Taxi, Hierarchical Control

Available Cache Size	Average Per-Action Time
None	4730.61 ms [4176,5996]
500	237.78 ms [0,1]

Table 6.11: Effects of Adding a Plan Cache on Required Planning Time - Hallway, Dynamic Flat Control

Available Cache Size	Average Per-Action Time
None	174.34 ms [47,240]
500	137.38 ms [0,195.5]
1000	139.55 ms [0,216.5]
Unlimited	134.57 ms [0,195]

The effectiveness of an exact match belief-state cache is directly related to the determinism of the domain the agent is acting in. This determinism can be provided by actions that primarily work correctly and sensors that tend to return correct results. Given the same initial belief state, an agent's future belief states will be

Table 6.12: Effects of Adding a Plan Cache on Required Planning Time - Twenty Questions, Hierarchical Control

Available Cache Size	Average Per-Action Time
None	114.28 ms [72,153]
500	89.27 ms [0,113]
1000	86.15 ms [0,112]
Unlimited	87.19 ms [0,102]

Table 6.13: Effects of Adding a Plan Cache on Required Planning Time - Robot Tag, Dynamic Flat Control

Available Cache Size	Average Per-Action Time
None	2471.36 ms [1890,3105]
500	1617.90 ms [0,2841]
1000	1181.13 ms [0,2500]
Unlimited	921.85 ms [0,2094]

similar to those of the past and the agent can take advantage of the experiences it has already had and the parts of the policy it has already computed.

Tables 6.9 through 6.14 report for different problems and different cache sizes the metrics of average planning time, maximum cache usage, and the number of trials for which DDN planning was required. The number of trials which require planning metric provides a measure of how quickly the agent covers the important parts of the state space and can rely on just cached information. The Cheese Taxi problem and Large Cheese Taxi problem both have deterministic actions and correct observations and are able to take significant advantage of the exact-match belief state cache. For both the Cheese Taxi and Large Cheese Taxi problems, the difference in planning time distributions is statistically significant, with p-values from the Mann-Whitney test for both problems reported as $< 2.2e\text{-}16$ when comparing cache-utilizing times against non-cache times. On the other hand, the Twenty Questions domain is very variable. The object being guessed can change across any time step and answers to

Table 6.14: Cache Metrics Related to Belief State Usage - Unlimited Cache Size

Problem	Maximum Cache Use	Trials Requiring Planning
CheeseTaxi	35	7 of 200
Hallway	1877	187 of 200
LargeCheeseTaxi	98	92 of 250
TwentyQuestions	2225	220 of 250
RobotTag	1691	107 of 250

questions are not always correct or useful. This leads the agent to enter into a much larger set of belief states. Accordingly, the agent needs alot more experiences and a larger cache to cover the set of belief states it experiences. The consequences of the larger belief state set for the Twenty Questions problem are the higher usage of cache slots and higher average planning time. For the TwentyQuestions problem, the planning time distributions are significantly different when non-cached versus cached algorithms are compared, with a p-value of $< 2.2e\text{-}16$ returned. When the planning time distributions are compared across caching algorithms with different cache sizes, the differences detected by the Mann-Whitney test are still significant, but not as strongly as for cached versus non-cached. In particular, the p-value for a comparision between planning times for a cache of size 500 and a cache of size 1000 is 2.488e-4, and the p-value for a comparison between a cache of 1000 and an unlimited size cache is 1.51e-06. The Hallway problem is noisy in both actions and observations and cache results similar to those of the Twenty Questions problem are seen. Fewer trials are completely cache based, more cache space is needed, and the per-action planning time does not show as large a drop in required computation time (compared to the more deterministic domains). The Mann-Whitney test suggests there is a significant difference between the planning time distributions generated for cache-based system and non-cache based systems in this domain ($p < 2.2e\text{-}16$ for comparisons between no cache and a cache of any size). The p-values returned when different size caches are

compared are larger than 0.05, however, and the null hypothesis that the planning time observations are from the same distribution can't be rejected in these cases. Finally, for the Robot Tag domain, all planning time distributions returned from each different cache implementation are considered significantly different, with all p-values $<$ 4.441e-15.

The graphs in Figures 6.1 through 6.5 demonstrate how a decrease in planning time is correlated with the number of simulations that are executed. An agent, as it experiences more and more of the domain, needs to rely on computing optimal actions less. The graphs are particularly striking for the taxi navigation problems. Each graph shows the average per-action planning time required, computed for windows of 100 actions. Since agents perform many actions in a set of trials, an average over a 100 action window provides for snapshots of how well the agent is able to take advantage of its previous experiences as execution progresses. Figures 6.1 and 6.2 represent the cheese taxi problems and demonstrate how quickly the reachable belief space is encountered and evaluated. Figure 6.5 shows that the Robot Tag belief space takes approximately 3000 actions to explore. After this time, the agent is able to take significant advantage of its experiences, particularly if the size of the cache is unbounded. The cache performance for size-limited caches is hampered by two factors: additional overhead to implement the LRU replacement algorithm and additional planning time to compute optimal actions for belief states that may have once been in the cache but were removed. By viewing the contents of the cache, the LRU replacement algorithm does tend to preserve those states that are commonly encountered. As an example, the most commonly exploited belief states in the cache for the Robot Tag problem are the initial state and the in-the-same-location state (where the optimal action is for the agent to tag the opponent).

By using a similarity-based cache, it is possible to take advantage of precomputed

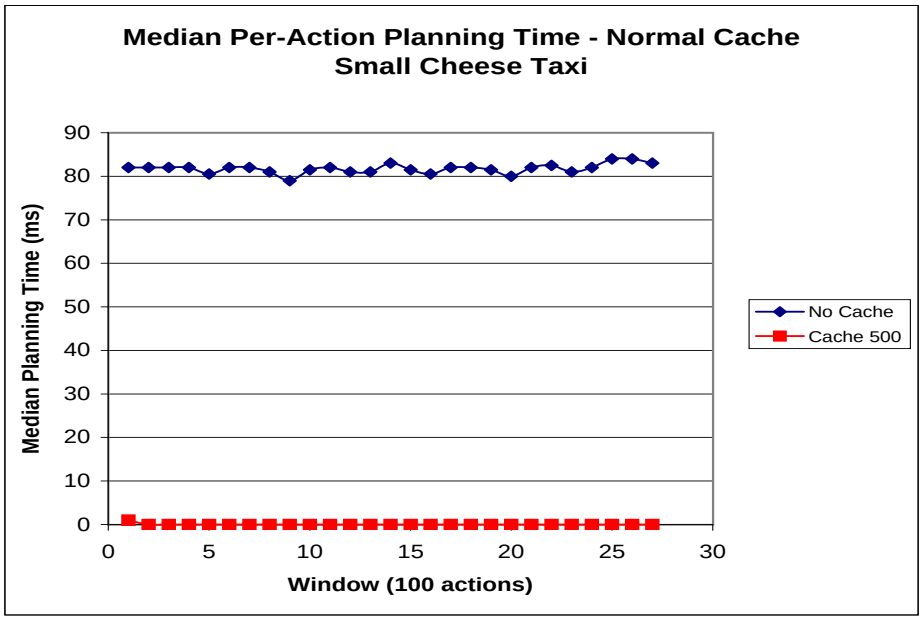

Figure 6.1: Sliding median planning time required for Small Cheese Taxi with normal cache.

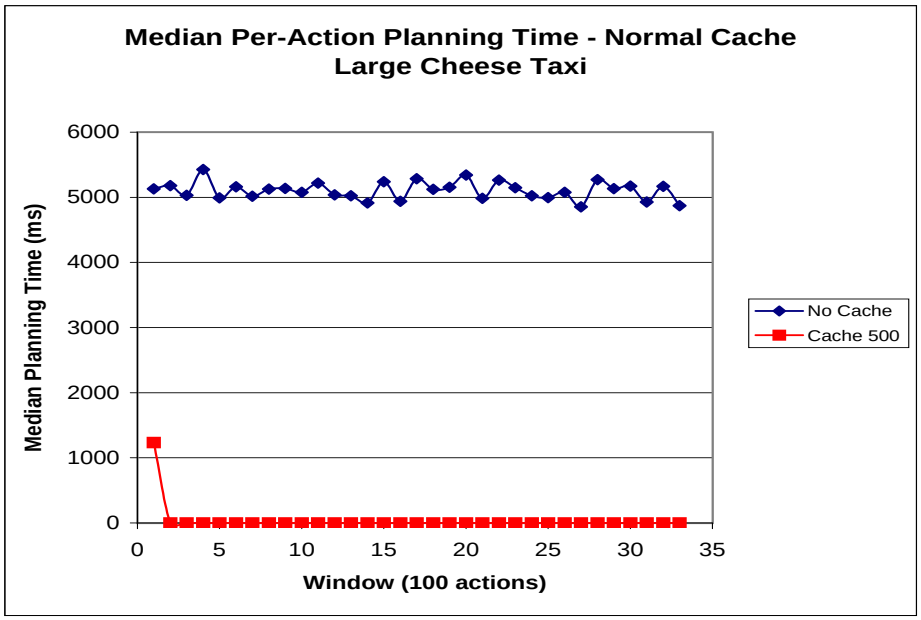

Figure 6.2: Sliding median planning time required for Large Cheese Taxi with normal cache.

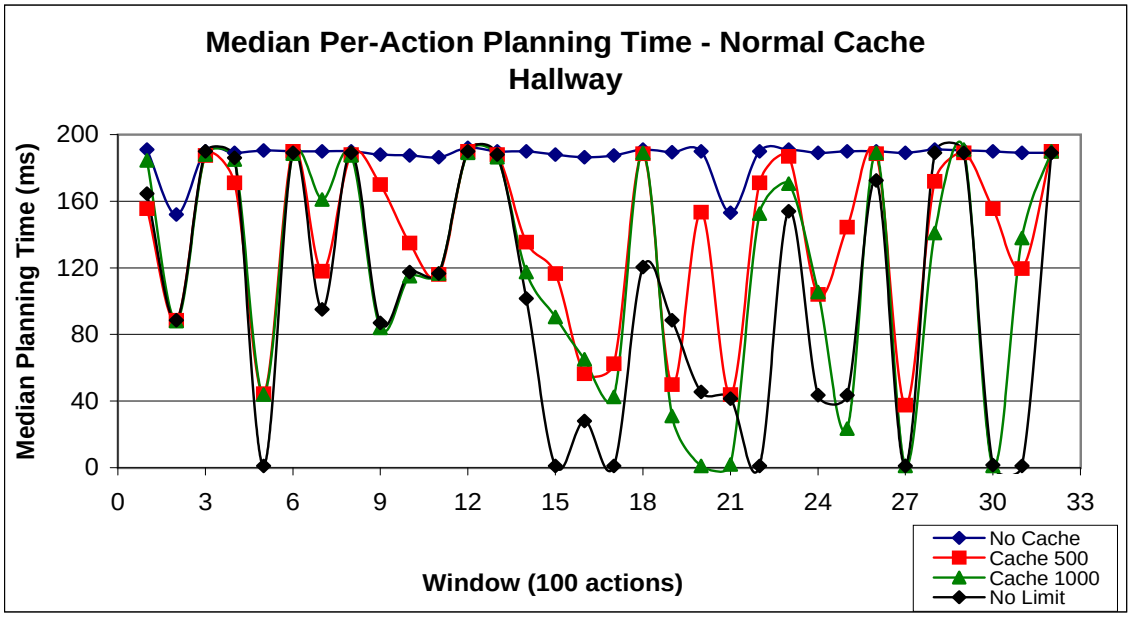

Figure 6.3: Sliding median planning time required for Hallway with normal cache.

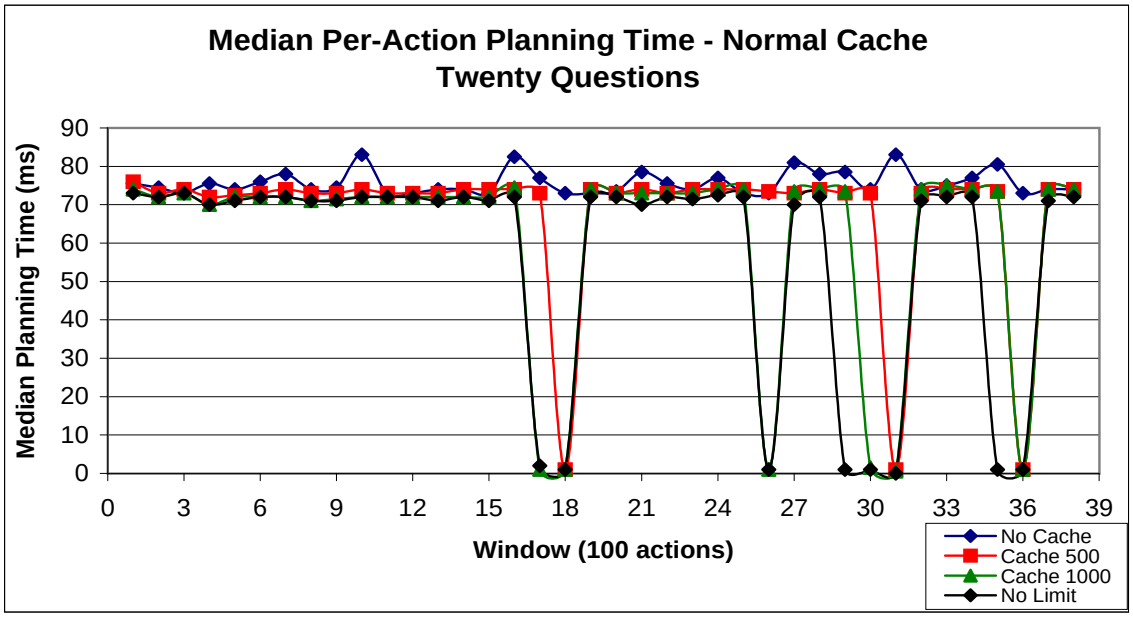

Figure 6.4: Sliding median planning time required for Twenty Questions with normal cache.

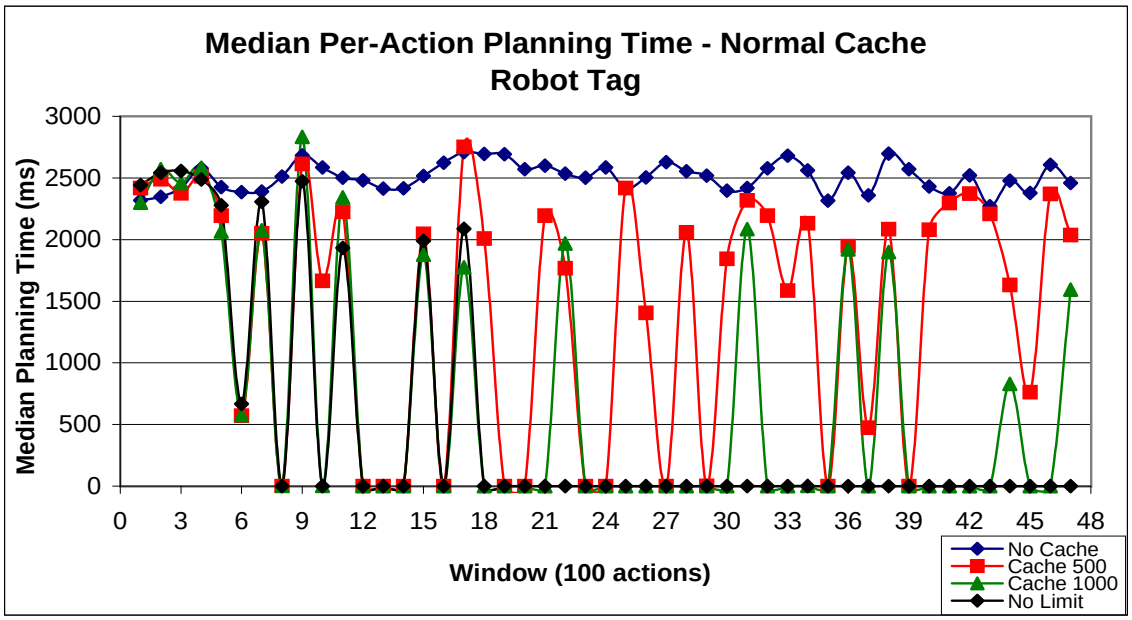

Figure 6.5: Sliding median planning time required for Robot Tag with normal cache.

cache entries for similar belief states. This type of cache can introduce a decrease in plan quality, however, as the action for a similar belief state may not be the optimal action for the true belief state. The domains of interest for examining the similarity-based cache are the Hallway, Twenty Questions, and Robot Tag problems. An agent using a similarity-based cache mechanism does not necessarily generate the same policy as an agent using no cache or an exact match cache. Accordingly, it is not possible to use identical trials to compare implementations of similarity-based caches. To test alternative similarity-based cache techniques, identical start states were used across different techniques while transitions and observations were allowed to vary according to domain dynamics. The start states used were the same start states selected for testing implementations of exact-match caches.

For each problem, improvements in computational requirements are seen when using a similarity-based cache. Tables 6.15 through 6.18 report the metrics of average planning time, average reward, and maximum cache usage, while Figures 6.6 through

6.8 detail average per-action planning time required over windows of 100 actions. First quartile and third quartile endpoints are also represented in brackets for each table entry. For all cases, reductions in average planning times and number of stored belief states are seen with generally minimal loss in policy quality (particularly for small thresholds).

For the Hallway problem, the Mann-Whitney test suggests that there are significant differences in planning time distributions when one moves from no-cache to any type of cache ($p < 2.2e\text{-}16$ in all comparisons) and when one moves from an unlimited size exact cache to an unlimited size epsilon cache with epsilon value > 0.05 ($p < 3.016e\text{-}4$ for these cases). Per the Mann-Whitney test, differences between the various epsilon thresholds are not large enough to warrant rejection of the null hypothesis. When reward distributions are compared across all different cache implementation comparisons, the Mann-Whitney test returns p-values of > 0.05. This means that it is not possible to reject the null hypothesis that these rewards are from the same distribution.

For the Twenty Questions problem, all planning time distributions are deemed significantly different from each other by the Mann-Whitney test, with the largest p-value returned being 2.905e-3. When rewards are compared, only the 0.20-epsilon threshold reward distribution is considered significantly different from the others ($p < 1.552e\text{-}10$). The effect of epsilon values on policy quality appears to be domain-dependent, as the Twenty Questions problem shows a significant drop in policy quality for an epsilon-value of 0.20, while the same level has no effect on the the average policy quality for the Hallway problem and only a small effect on the average for the Robot Tag problem.

Finally, for the Robot Tag problem, there are significant differences in the planning time distributions when comparing a non-cache implementation versus any cache

Table 6.15: Effects of Adding a Grid-Cache on Planning Time and Quality - Hallway, Dynamic Flat Control

Cache Size	Threshold	Action Time	Reward
None	0.00	174.34 ms [47,240]	-15.55 [-20,-9]
Unlimited	0.00	134.57 ms [0,195]	-15.55 [-20,-9]
Unlimited	0.01	102.06 ms [1,201]	-15.75 [-20,-10]
Unlimited	0.05	88.06 ms [1,157]	-16.16 [-21,-9]
Unlimited	0.10	77.50 ms [1,76]	-16.43 [-21,-9]
Unlimited	0.20	71.25 ms [2,65]	-15.16 [-19,-9]

implementation and when comparing the unlimited size exact-cache implementation to any epsilon cache implementation ($p < 2.2e\text{-}16$ for all of these cases). Note that the version of the Mann-Whitney test used only checks for differences (a two-sided test), not for the direction of difference. In this case, the difference may not be an improvement (though that has been the general case up to this point as averages, medians, and quartile endpoints have shown improvements with new techniques). In particular, per-action average planning times for the Robot Tag problem are worse when using a grid-cache with a small epsilon value than when using an exact belief state cache. This is likely due to the additional overhead needed in the grid-cache algorithm to examine the unfactored state space representation for shifts in the belief probability distribution. For the Robot Tag problem, this unfactored representation is fairly large (870 states) and, thus, computationally expensive to analyze. The planning time distribution for the epsilon 0.01 cache is significantly different when compared to the distributions from other epsilon caches ($p < 4.795e\text{-}11$ in all of these cases). Mann-Whitney comparisons between the other epsilon caches return p-values above the critical threshold of 0.05, which does not allow rejection of the null hypothesis that these planning times may be from the same distribution. Mann-Whitney comparisons of reward distributions all return p-values > 0.05, meaning that the null hypothesis of no reward difference cannot be rejected.

Table 6.16: Effects of Adding a Grid-Cache on Planning Time and Quality - Twenty Questions, Hierarchical Control

Cache Size	Threshold	Action Time	Reward
None	0.00	114.28 ms [72,153]	-18.5 [-29,-4]
Unlimited	0.00	87.19 ms [0,102]	-18.5 [-29,-4]
Unlimited	0.01	68.13 ms [2,85]	-18.42 [-30,-3]
Unlimited	0.05	47.78 ms [3,74]	-17.57 [-30,-3]
Unlimited	0.10	34.69 ms [3,29]	-19.45 [-30,-3]
Unlimited	0.20	10.42 ms [4,10]	-34.98 [-54,-10]

Table 6.17: Effects of Adding a Grid-Cache on Planning Time and Quality - Robot Tag, Dynamic Flat Control

Cache Size	Threshold	Action Time	Reward
None	0.00	2471.36 ms [1890,3105]	-8.30[-14,0]
Unlimited	0.00	921.85 ms [0,2094]	-8.30 [-14,0]
Unlimited	0.01	1270.92 ms [1,1998]	-8.23 [-14,0]
Unlimited	0.05	1035.17 ms [1,1596.5]	-8.39 [-13,0]
Unlimited	0.10	883.31 ms [1,1160.5]	-8.72 [-16,0]
Unlimited	0.20	705.10 ms [49,844]	-9.03 [-16,0]

Table 6.18: Normal vs Grid Cache Usage - Unlimited Cache Size

Problem	Cache Type	Threshold	Cache Used
Hallway	Normal	0.0	1877
Hallway	Grid	0.01	1156
Hallway	Grid	0.05	932
Hallway	Grid	0.10	729
Hallway	Grid	0.20	572
TwentyQuestions	Normal	0.0	2225
TwentyQuestions	Grid	0.01	1774
TwentyQuestions	Grid	0.05	1010
TwentyQuestions	Grid	0.10	685
TwentyQuestions	Grid	0.20	220
RobotTag	Normal	0.0	1691
RobotTag	Grid	0.01	1042
RobotTag	Grid	0.05	901
RobotTag	Grid	0.10	704
RobotTag	Grid	0.20	498

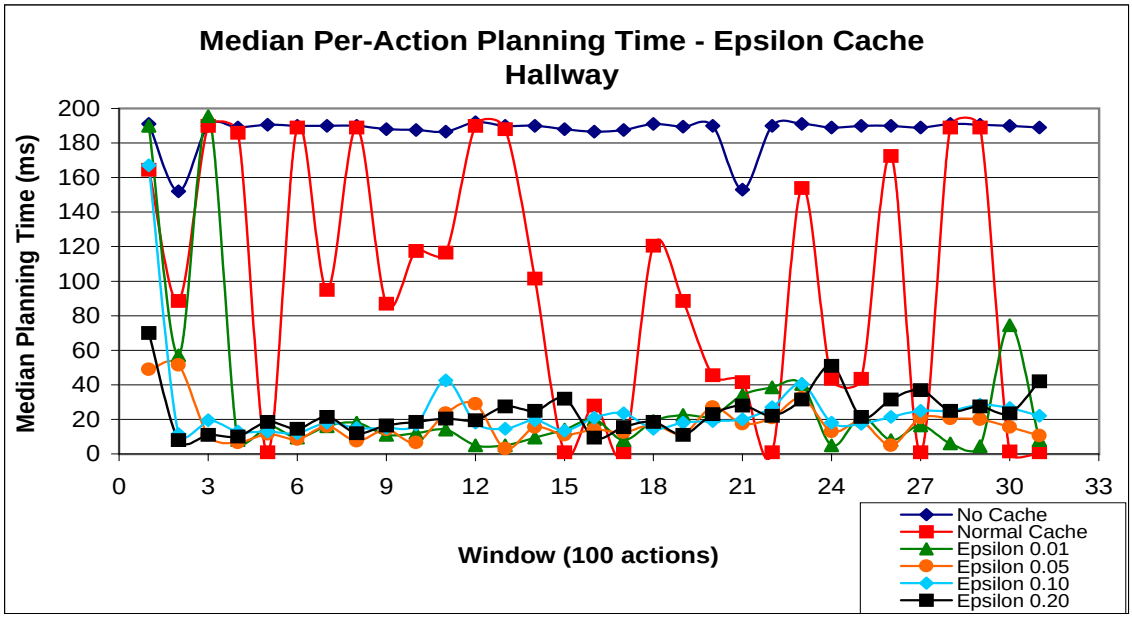

Figure 6.6: Sliding median planning time required for Hallway with grid-based cache.

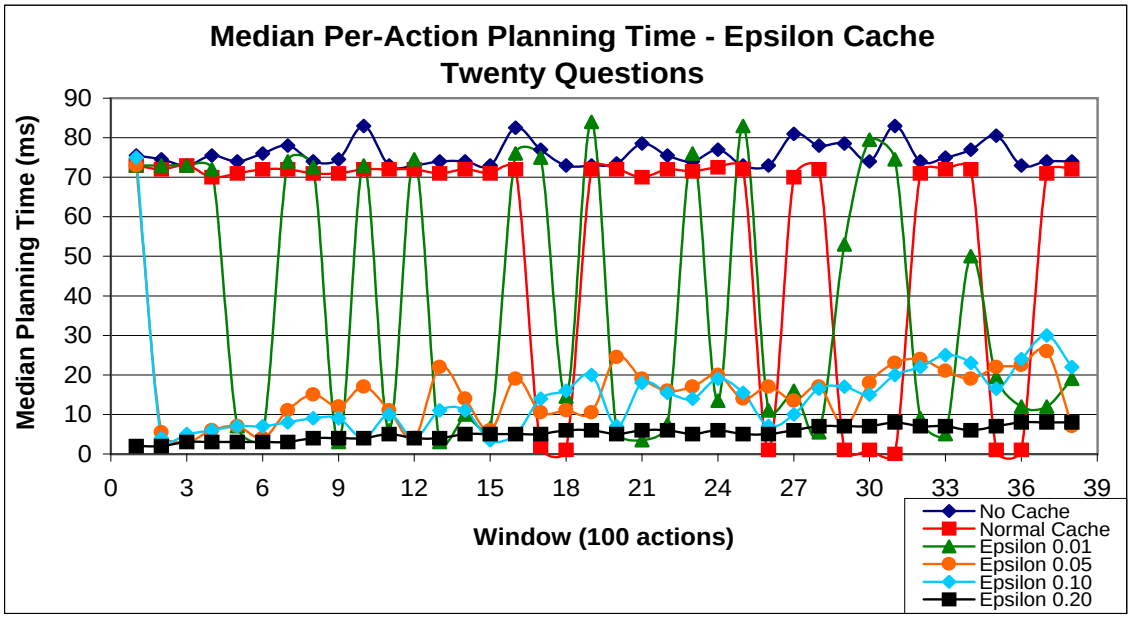

Figure 6.7: Sliding median planning time required for Twenty Questions with grid-based cache.

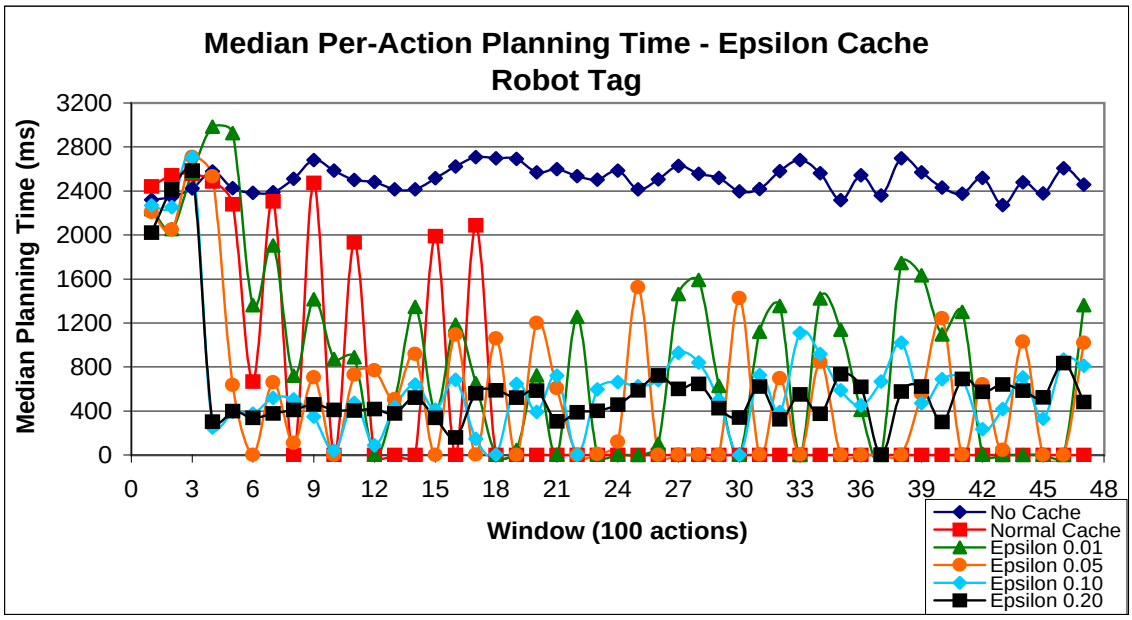

Figure 6.8: Sliding median planning time required for Robot Tag with grid-based cache.

6.3 Exploiting Run-Time Knowledge through Dynamic Construction

The effects of dynamic construction of decision networks on computation time was evaluated in three POMDP domains. The Large Cheese Taxi, Hallway, and Robot Tag problems were evaluated as they represent problems that are computationally expensive even when considering hierarchical representations and caching. Additionally, they represent two classes of problems for which different dynamic construction techniques are useful. The Large Cheese Taxi problem, due to its near deterministic action effects and observations, quickly converges to using just a small part of the belief space. On the other hand, the Robot Tag problem contains a mix of variables, some that are always fully observable and others that never converge into a small set of beliefs. The Hallway problem lies in the middle of these two classes, as it has landmark observations that can be used to find the exact state, but is fairly noisy

when not at a landmark. The Twenty Questions problem was not used for evaluation as the dynamics of the problem force the consideration of every state during each timestep (as the object being guessed can change to any other object at anytime). Thus, Twenty Questions is not an effective problem for the application of dynamic construction techniques. Approximate methods ignoring parts of the state space that are highly unlikely may be effective for such a problem, however.

For the Large Cheese Taxi problem and Hallway problem, the dynamically constructed decision networks used the same number of lookahead steps and observation links as their respective statically built networks. For all three problems, the average planning time required for a single action and the average time required to build a decision network for a single action were computed. For the Hallway problem, dynamic construction was more expensive than use of a prebuilt DDN if dynamic construction was always used. The reported results constrain dynamic construction to be used only when the number of states with probability greater than zero drops to 20 or less (one-third of the original states). The pre-execution time was recorded for each problem and consists of the time required for MDP reward generation, computation of the potential states needed for each node in the network, and construction of the network structure at the start of execution. 250 trials were run for the Large Cheese Taxi and RobotTag problems and 200 trials were run for the Hallway problem. For the flat algorithm on the Hallway problem, a total of 27 trials were run as each trial took upwards of twenty minutes.

Results are shown for dynamic construction of decision networks in Tables 6.19-6.24. Three tables present average build and plan times, pre-execution time, and the corresponding first quartile and third quartile endpoints where appropriate. The other three tables present amortized planning costs for each of the problems and algorithms tested. For all three problems, the planning time distributions generated

Table 6.19: Effects of Using Dynamic Construction - Large Cheese Taxi Problem

Control	Per-Action Plan	Per-Action Build	Pre-Execute
Hier	4671.02 ms [4103,5951]	0 ms	130 ms
Dyn Hier	56 ms [10,15]	79.48 ms [27,41]	8982.6 ms

Table 6.20: Amortized Planning Costs for Large Cheese Taxi Problem - 250 trials

Control	Amortized Cost Per Action
Hierarchical	4671.06 ms
Dynamic Hierarchical	138.53 ms

from the non-dynamic construction algorithm are considered significantly different ($p < 2.2e\text{-}16$ from the Mann-Whitney test in all cases) when compared to the planning time distributions generated from the dynamic construction algorithms. When a Mann-Whitney test was used to compare the planning times for the fully-observable dynamic build algorithm against those from the highly compressed dynamic build algorithm on the Robot Tag problem, the difference was also considered significant ($p < 2.2e\text{-}16$). The most striking results are for the Large Cheese Taxi problem, showing a large drop in per-action required time over the static DDN approach. Another interesting result was that the network built for the Robot Tag problem using the fully observable state algorithm, which takes advantage of extra precomputed information when building, required longer average build and plan times than when the highly compressed building technique was used. The networks that were built by the highly compressed technique allowed for enough compression to make up for the extra building overhead. Dynamic construction of decision networks as implemented is an exact approach, and its use does not affect the quality of the policies that are generated.

Overall, these results demonstrate that the developed approximation techniques for DDN-based planning in POMDP domains can be useful across a variety of do-

Table 6.21: Effects of Using Dynamic Construction - Hallway Problem

Control	Per-Action Plan	Per-Action Build	Pre-Execute
Flat	249.15 ms [190,297]	0 ms	155.4 ms
Dyn Flat	149.75 ms [14,217]	24.51 ms [0,24]	1510.4 ms

Table 6.22: Amortized Planning Costs for Hallway Problem - 200 trials

Control	Amortized Cost Per Action
Flat	249.20 ms
Dynamic Flat	174.80 ms

Table 6.23: Effects of Using Dynamic Construction - Robot Tag Problem

Control	Per-Action Plan	Per-Action Build	Pre-Execute
Flat	108.05 s [106,110.4]	0 s	1.829 s
Dyn Flat	0.594 s [0.332,0.832]	1.927 s [1.434,2.399]	12.709 s
FO-Dyn Flat	0.846 s [0.483,1.123]	2.132 s [1.588,2.673]	14.628 s

Table 6.24: Amortized Planning Costs for Robot Tag Problem - 250 trials

Control	Amortized Cost Per Action
Flat	108.054 s (27 trials)
Dynamic Flat	2.524 s
Dynamic Flat, FO State	2.981 s

mains. The most interesting result of these tests is the fact that certain domains, such as Large Cheese Taxi, can easily be handled by DDN-based planners, even if they are unsolvable under reasonable time constraints by other POMDP approaches. These domains have actions that perform close to expected and observations that are useful for information-gathering. These domain characteristics minimize the set of belief states that are reached by the agent. Because the observations are useful and the agent quickly localizes, very small dynamic decision networks are needed for reasoning. Caching is also very effective under these same conditions. Since these DDNs are small, both network construction and probability propagation are fast. Finally, since the agent is localizing fairly quickly, the fully-observable MDP estimates are very close to the true value function for the POMDP, so the quality of the generated policies can remain very high.

6.4 Embedding Principled Behaviors

Given the proposed principles and approach to defining a social utility function described in Chapter 4, experimental validation can demonstrate whether an engineered utility function does in fact allow reliable achievement of social goals as well as demonstrate how much the strict safety approach hurts individual agent achievement. The defined approach has been evaluated on variations of two multiagent problems seen in the literature - Tambe's Helo Domain and the Tragedy of the Commons cow grazing problem.

Table 6.25: Policy Comparisons for Helo Domain

Metric	Greedy	Social	FO - Ind	FO - Central
Transport Deadline	87.14%	100%	100%	100%
Avg. Escort Utility	5.495 [3,8]	4.968 [3,7]	5.368 [3,8]	5.017 [3,7]
Avg. Transport Utility	4.183 [2,6]	4.683 [2,7]	4.454 [2,6]	5.224 [3,7]

Given the definition of the Helo Problem from Chapter 3, and using the six step process defined in *Section 4.4*, the mission achievement reward for the Helo Problem is just required to be greater than the maximum individual reward. 1205 start states for the agents were generated to test the socially-oriented agent policies. These states are the set from which the agents should be able to reach the mission goal state through some combination of actions. This set of states was taken from the set of permutations of all escort and transport states (except destroyed and end), all possible timesteps left, and all possible radar locations in front of the escort. Results for this test are shown in Table 6.25. This table presents the percentage of trials in which the transport reaches the endpoint before the deadline as well as the average utility earned by the transport and escort, which is equivalent to the average time spent at the endpoint for each helicopter. The first quartile and third quartile endpoints are also presented to show the bounds of the middle 50% of the rewards. A Mann-Whitney comparison of utilities was not performed due to the small number of possible utilities that could be attained. A small reward set is likely to lead to a large number of rank ties, decreasing the strength of the Mann-Whitney test. As expected, the escort agent was willing to support the mission by sacrificing itself when needed. This happened in 69 cases, all allowing the transport to reach the end state on time. A performance comparison was also made against agents acting under greedy policies. The greedy escort policy just flies directly to the destination without helping the transport, so this policy can be used as an upper bound on the escort local utility. Comparisons were also made to results from two fully observable instances of the problem. In the fully observable instances, the *Communicate* action is no longer needed as both agents can see the true state of the world. The first version of the fully observable problem used individual utility maximization. In this case, the escort agent would only destroy the radar if needed to ensure the mission success. Otherwise, it would fly

straight to the end base to maximize its own reward. The other version of the fully observable problem used global utility maximization. In this case, the escort agent would destroy the radar at times to increase the earned utility of the transport and the combined utility. While the average individual utility gained by the agents under the social utility function is not as high as other approaches, the approach is able to consistently achieve mission goals. The greedy and social approaches were also more realistic instances of the problem, as the FO implementations allowed for complete knowledge of the world state by both agents.

According to the proposed principles, the agents in the Tragedy of the Commons cow grazing problem should be willing to lose money at market as long as it ensures that the common pasture continues growing. Given this definition of the problem, and using the six step process defined in *Section 4.4*, the social goal achievement reward for this problem is required to be approximately 100,000. This relatively high value (compared to the individual rewards) comes from one state in particular for which the likelihood of social success is high for both actions, with *MoveFromCommon* being slightly higher, while the likelihood of individual success moves from 0.70 for the *MoveToCommon* action to 0 for the *MoveFromCommon* action. In this case, since the *MoveFromCommon* approach provides for more likely social success, it is chosen, even when it is guaranteed to not allow for individual goal achievement.

Given this social reward function, the corresponding principle-based, socially-biased policy was generated and the performance of agents acting under this policy was compared against those acting under a purely greedy policy. In the purely greedy policy, the individual agents have no regard for the amount of grass left in the common pasture. They solely prefer states where they have weights of 10 or more. The greedy policy was edited to select socially useful states when an agent achieved its maximum weight, moving off the common field when the current weight is 12. A third

policy was also generated, which assumes a centralized planner with full knowledge of the world-state (weights of all cows). This approach gives the optimal policy for the three-cow grazing problem, but is significantly more computationally expensive than the distributed greedy and social approaches.

Ideally, the tests should demonstrate that the principle-based policy does preserve the common pasture, while only showing a minor loss in profit from sales. All 25920 non-trivial start states were generated and were used to test both policies. These 25920 states are all possible common grass units from 1 to 15, and all possible weights for all three cows from 0 to 11. The most interesting of these cases are those where the common pasture has little grass, and the cows all need to put on alot of weight.

Table 6.26: Policy Comparisons for Cow Grazing Domain

Test	Metric	Greedy	Social	FO Optimal
NonTrivial	Trials with Grass Left	75%	100%	100%
NonTrivial	Sellable Cows	95.4%	93.4%	99.46%
NonTrivial	Average Weight	11.86	11.75	11.92
Difficult	Trials with Grass Left	41%	100%	100%
Difficult	Sellable Cows	82%	84.6%	96.25%
Difficult	Average Weight	11.52	11.41	11.51

Key results for this test are shown in Table 6.26. For the greedy policy, 75% of the time the agents leave grass on the common pasture, with 95.4% of the cows being above weight and an average weight of 11.86. For the socially-oriented policy, 100% of the time the agents leave grass on the common pasture. At the same time, 93.4% of the cows achieve a weight of 10 or more to be sold, and the average weight is 11.75 units. This is positive for the use of the socially oriented social utility function in this domain, as the agents were always able to achieve the social goal, with little setback on individual goals (cows sold and weight). For a more interesting comparison, one can also look at just the cases for which the agents were in a difficult situation. 1875

cases were examined in which the weight of all cows started at 4 or less. With a weight of 4, an individual cow can graze just on the local pasture and gain enough weight to be sold. With weights below four, the agents must share the common pasture to even make weight, much less make a large profit. In these cases, using the greedy policy produced 40% of the trials with common grass above zero, 82% of the cows above weight, and an average weight of 11.52. The socially-oriented policy once again achieved a rate of 100% common grass above zero, while also achieving 84.6% cows above weight and an average weight of 11.41. Due to the constrained set of possible rewards for this problem (weights of 10 through 12), a Mann-Whitney test is inappropriate for this data as almost all data points in a comparision will be tied in rank with a majority of the other points. First quartile and third quartile endpoints have also not been presented, as the majority of weights earned were the maximal weight of 12.

These results suggest that the social goal of grass conservation also produced positive feedback within the system, allowing the agents to support each others weight growth, even when constrained in how much grass to eat and in how much knowledge of the other agents was available. As Turner (Turner, 1993) suggests for combatting the tragedy of the commons in cooperative domains, this approach appears to have successfully – "taken pains to ensure that there are shared goals of preserving the commons..." (defining social goals), "... and that these have priority over individual goals" (defining the social goal reward function appropriately).

Chapter 7

Conclusions and Future Work

7.1 Summary of Contributions

This research work has produced a set of implemented algorithms that can exploit structure and run-time knowledge in POMDP domains. These approximate algorithms demonstrate improvements in scalability for POMDP planning, while also maintaining the ability to generate high quality policies. In particular, a novel algorithm for POMDP planning with hierarchical dynamic decision networks and a novel algorithm for runtime construction of dynamic decision networks have been introduced. In addition, an analysis of runtime exact and grid-based plan caching algorithms has been performed. Scalability in problem solving and quality of generated policies for the implemented algorithms have been compared against alternative state-of-the-art exact and approximate approaches for POMDP planning.

This work has also contributed to the field of distributed planning a principle-based approach for developing local policies for cooperative agents in MDP domains. This algorithm is designed for a constrained but applicable set of domains. The approach has been shown to allow for achievement of critical social goals while supporting local optimization in two test environments. A comparison of the quality of

policies developed under the principle-based approach has been made against strictly selfish local agent policies, as well as policies developed by fully-observable, centralized approaches.

7.2 Future Work

Several lines of research extend naturally from this work. One area of continuing research is improvements to the theoretical and practical aspects of the implemented algorithms. While the algorithms developed in this research have been demonstrated to generate plans of comparable quality to other approximate planners under reasonable time constraints, there is no guarantee on the quality of the policies that will be generated. Examining theoretical guarantees for the approaches of abstraction and grid-caching is important to fully understand the applicability of such approaches.

The approach to building a hierarchical domain model used in this work assumes that abstract actions will successfully complete. This requires a model of state completion from the agent designer. The approach also uses fully observable MDP solutions to estimate the costs of abstract actions. The application of learning approaches, such as expectation-maximization (EM) and reinforcement learning, could be made use of to improve models of abstract actions. Since the accuracy of these models and cost estimates affect an agent's high-level decision making, improved models should lead to improvements in the quality of decisions that an agent makes.

Improvements to the grid-cache algorithm should be focused on more efficient ways to store and search the belief points represented in the cache. Another area of potential research that bridges both distributed planning and caching of plans is the ability to share cached local knowledge at the social level. This approach stems from a humanistic view of agents. People rarely plan completely from scratch for a new situation. Instead, people tend to fit plans that they have used before to the

current situation or ask others for useful actions to take. If agents can share cached plans, there is potential for significant cost savings for plan generation and increased performance in execution.

Related to planning in realistic (POMDP) domains is the problem of plan recognition - inferring what another agent may be attempting to achieve and predicting next actions by observing current states and actions. Hierarchical plan recognition approaches should allow for a reduction in uncertainty in determining which policies an agent is pursuing. Hierarchical plan recognition should also prove useful in distributed domains for modeling group level strategies, given that group level actions are modeled as abstractions of individual actions. Significant work in using Bayesian networks for plan recognition has already been performed (Huber and Durfee, 1993). Such approaches could benefit from techniques such as run-time construction of belief networks for increasing the responsiveness of plan recognition algorithms.

A third area of research is in the continued analysis of principled decision making. While the current work has evaluated the principles of harm and goal achievement, other principles could be examined within this framework such as efficiency and conventions. The current approach is designed for fully observable Markov Decision Processes but should also be evaluated in partially observable domains. An initial approach for using the implemented principled utility functions in POMDPs is to use the generated utility function as the final state reward estimates in the decision networks used for POMDP planning. Finally, analysis of appropriate situations in which relaxation of the strong safety approach is applicable would be useful. Such analysis would allow for development of agents that can still perform well socially, albeit without a strict guarantee of highest probability of social goal achievement. Agents acting under such a mechanism would be able to optimize better at the individual level than those executing under the current algorithm.

Bibliography

Asimov, I. (1952). *Foundation and Empire*. Gnome Books.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Bernstein, D., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of markov decision processes. *Mathematics of Operation Research*, 27(24).

Blum, A. L. and Furst, M. (1997). Fast planning through planning graphy analysis. *Artificial Intelligence*, (90):281–300.

Blum, A. L. and Langford, J. C. (1999). Probabilistic planning in the graphplan framework. In *Proceedings of the Fifth European Conference on Planning*. Springer Verlag.

Blythe, J. (1999). Decision-theoretic planning. *Artificial Intelligence*, 20(1).

Bonet, B. (2002). An epsilon-optimal grid-based algorithm for partially observable markov decision processes. In *Proceedings of the 19th International Conference on Machine Learning*, pages 51–58. Morgan Kaufmann.

Boutilier, C. (1996). Planning, learning, and coordination in multiagent decision

processes. In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*, pages 195–201. Springer-Verlag.

Boutilier, C. (1997). Correlated action effects in decision-theoretic regression. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, pages 30–37.

Boutilier, C. (1999). Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conference On Artificial Intelligence*.

Boutilier, C., Brafman, R., and Geib, C. (1997). Prioritized goal decomposition of markov decision processes: Towards a synthesis of classical and decision theoretic planning. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*.

Boutilier, C., Brafman, R., Hoos, H., and Poole, D. (1999a). Reasoning with conditional ceteris paribus preference statements. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intellligence*.

Boutilier, C., Dean, T., and Hanks, S. (1999b). Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94.

Cassandra, T. (1999a). pomdp-solve software. Available at: http://www.cs.brown.edu/research/ai/pomdp/code/pomdp-sovle-v4.0.tar.gz. Brown University Department of Computer Science.

Cassandra, T. (1999b). Pomdps for dummies. Available at: http://www.cs.brown.edu/research/ai/pomdp/tutorial/. Brown University Department of Computer Science.

Chien, S., Sherwood, R., Rabideau, G., Castano, R., Davies, A., Burl, M., Knight, R., Stough, T., Roden, J., Zetocha, P., Wainwright, R., Gaasbeck, J. V., Cappelaere, P., and Oswald., D. (2002). The techsat-21 autonomous space science agent. In *Proceedings of the International Conference on Autonomous Agents*.

Dearden, R. and Boutilier, C. (1997). Abstraction and approximate decision-theoretic planning. *Artificial Intelligence*, 89:219–283.

desJardins, M., Durfee, E., Charles Ortiz, J., and Wolverton, M. (1999). A survey of research in distributed, continual planning. *AI Magazine*, 20(4):13–22.

Dietterich, T. (2000). Hierarchical reinforcement learning with the maxq value function decomposition function. *Journal of Artificial Intelligence Research*, 13:227–303.

Draper, D., Hanks, S., and Weld, D. (1993). Probabilistic planning with information gathering and contingent execution. Technical Report 93-12-04, University of Washington Department of Computer Science.

Durfee, E. (1999). Distributed problem solving and planning. In *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, pages 121–164. MIT Press.

Erol, K., Hendler, J., and Nau, D. (1994). Htn planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1123–1128.

Estlin, T., Castano, R., Anderson, R., Gaines, D., Fisher, F., and Judd, M. (2003). Learning and planning for mars rover science. In *Proceedings of the IJCAI Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World Modeling, Planning, Learning, and Communicating*.

Fitoussi, D. and Tennenholtz, M. (2000). Choosing social laws for multi-agent systems: Minimality and simplicity. *Artificial Intelligence*, 119:61–101.

Forbes, J., Huang, T., Kanazawa, K., and Russell, S. (1995). The batmobile: Towards a bayesian automated taxi. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.

Grinkewitz, R. (2001). Mdp solve software. Available at: http://plan.mcs.drexel.edu/courses/software/mdp. Drexel University Department of Computer Science.

Ha, V. and Musliner, D. (2003). Balancing safety against performance: Tradeoffs in internet security. In *Proceedings of the 36th Hawaii International Conference on System Sciences*.

Haddaway, P. and Hanks, S. (1993). Utility models for goal-directed decision-theoretic planners. Technical Report 93-06-04, Department of Computer Science and Engineering, University of Washington.

Hansen, E. and Zhou, R. (2003). Synthesis of hierarchical finite state controllers for pomdps. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling*.

Hauskrecht, M. (2000). Value function approximations for pomdps. *Journal of Artificial Intelligence Research*, 13:33–94.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA.

Huber, M. J. and Durfee, E. H. (1993). Observational uncertainty in plan recognition among interacting robots. In *Proceedings of the International Joint Conference on AI Workhop on Dynamically Interacting Robots*, pages 68–75.

Jensen, D. and Lesser, V. (2002). Social pathologies of adaptive agents. In *Safe Learning Agents: Papers from the 2002 AAAI Spring Syposium*, volume TR SS-02-07.

Jensen, F. (2001). *Bayesian Networks and Decision Graphs*. Springer-Verlag, New York, NY.

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

Lesser, V., Decker, K., Wagner, T., Carver, N., Garvey, A., Horling, B., Neiman, D., Podorozhny, R., Prasad, M. N., Raja, A., Vincent, R., Xuan, P., and Zhang, X. (2002). Evolution of the gpg/taems domain-independent coordination framework. Technical Report 02-03, University of Massachusetts-Amherst Department of Computer Science.

Lesser, V. R. (1991). A retrospective view of fa/c distributed problem solving. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(6):1347–1362.

Lin, S.-H. and Dean, T. (1995). Generating optimal policies for high-level plans with conditional branches and loops. In *Proceedings of the Third European Workshop on Planning*, pages 187–200.

Littman, M. L. (1994). The witness algorithm: Solving partially observable markov decision processes. Technical Report CS-94-40, Department of Computer Science, Brown University.

Littman, M. L., Cassandra, A. R., and Kaelbling, L. P. (1995a). Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370.

Littman, M. L., Dean, T. L., and Kaelbling, L. P. (1995b). On the complexity of solving markov decision problems. In *Proceedings of the Eleventh Annual Conference on Uncertainty in Artificial Intelligence*, pages 394–402.

Marichal, J.-L. and Roubens, M. (2000). Determination of weights of interacting criteria from a reference set. *European Journal of Operational Research*, 124(3):641–650.

McCallum, A. R. (1993). Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Machine Learning Conference*.

Muscettola, N., Nayak, P., Pell, B., and Williams, B. (1998). Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103:5–48.

Nair, R., Tambe, M., Yooko, M., Pynadath, D., and Tambe, M. (2003). Taming decentralized pomdps: Towards efficient policy computation for multiagent settings. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Nau, D., Cao, Y., Lotem, A., and Munoz-Avila, H. (1999). Shop: Simple hierarchical ordered planner. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 968–973.

Ngo, L., Haddaway, P., Krieger, R., and Helwig, J. (1997). Efficient temporal probabilistic reasoning via context-sensitive model construction. *Computers in Biology and Medicine*, 27(5):453–476.

Nicholson, A. and Brady, M. (1994). Dynamic belief networks for discrete monitoring. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(11):1593–1610.

Nicholson, A. and Russell, S. (1993). Techniques for handling inference complexity in dynamic belief networks. Technical Report CS-93-31, Department of Computer Science, Brown University.

Pineau, J., Gordon, G., and Thrun, S. (2003). Point-based value iteration: An anytime algorithm for pomdps. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Pineau, J. and Thrun, S. (2002). An integrated approach to hierarchy and abstraction for pomdps. Technical Report CMU-RI-TR-02-21, Carnegie Mellon University School of Computer Science.

Provan, G. (1993). Tradeoffs in constructing and evaluating temporal influence diagrams. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*.

Pynadath, D. and Tambe, M. (2002). The communicative multiagent team decision problem: Analyzing teamwork theories and models. *Journal of Artificial Intelligence Research*, 16:389–423.

Rose, J. and Huhns, M. (2001). Philosophical agents. *IEEE Internet Computing*, 5(3):104–106.

Rose, J., Huhns, M., Roy, S. S., and Turkett, W. (2002). An agent architecture for long-term robustness. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1149–1156.

Russell, S. and Norvig, P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, second edition.

Shoham, Y. and Tennenholtz, M. (1995). On social laws for artificial agent societies: Off-line design. *Artificial Intelligence*, 73:231–252.

Shoham, Y. and Tennenholtz, M. (1997). On the emergence of social conventions: Modeling, analysis, and simulations. *Artificial Intelligence*, 94:139–166.

Theocharous, G. (2002). *Hierarchical Learning and Planning in Partially Observable Markov Decision Processes*. PhD thesis, Michigan State University, East Lansing, MI.

Turner, R. M. (1993). The tragedy of the commons and distributed ai systems. In *Proceedings of the Twelfth International Workshop on Distributed Artificial Intelligence*, pages 379–390.

Weld, D. and Etzioni, O. (1994). The first law of robotics (a call to arms). In *Proceedings of the Twelfth National Conference on AI*.

Xuan, P. and Lesser, V. (2002). Multi-agent policies: From centralized ones to decentralized ones. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*.

Zhang, N. L. and Liu, W. (1996). Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Hong Kong University of Science and Technology.

Zhang, X., Podorozhny, R., and Lesser, V. (2000). Cooperative, multistep negotiation over a multi-dimensional utility function. In *Proceedings of the IASTED International Conference, Artificial Intelligence and Soft Computing*, Banff, Canada. ASTED/ACTA Press.

Appendix A

Histograms of Planning Time Data

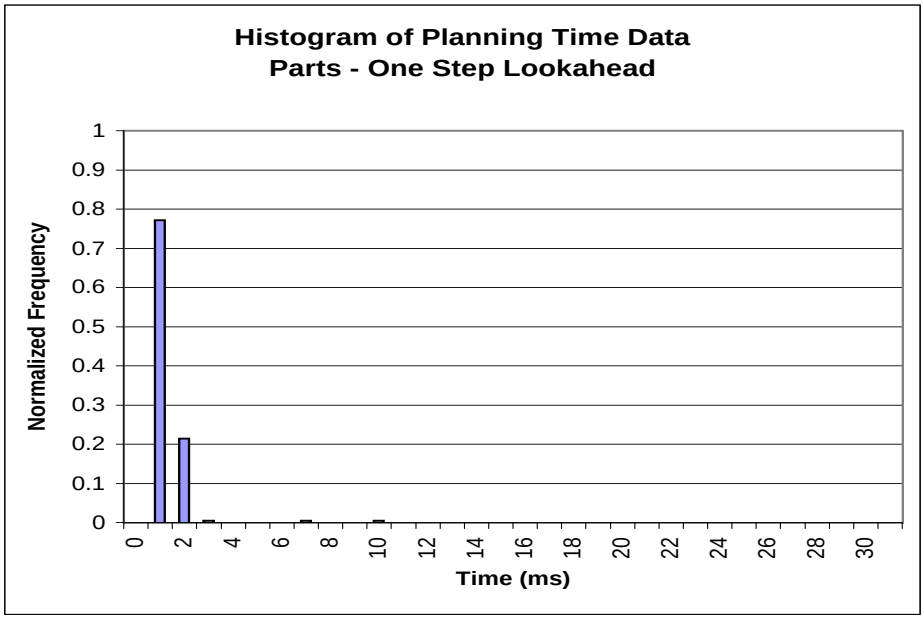

Figure A.1: Planning Time Data - Parts - One Step Lookahead

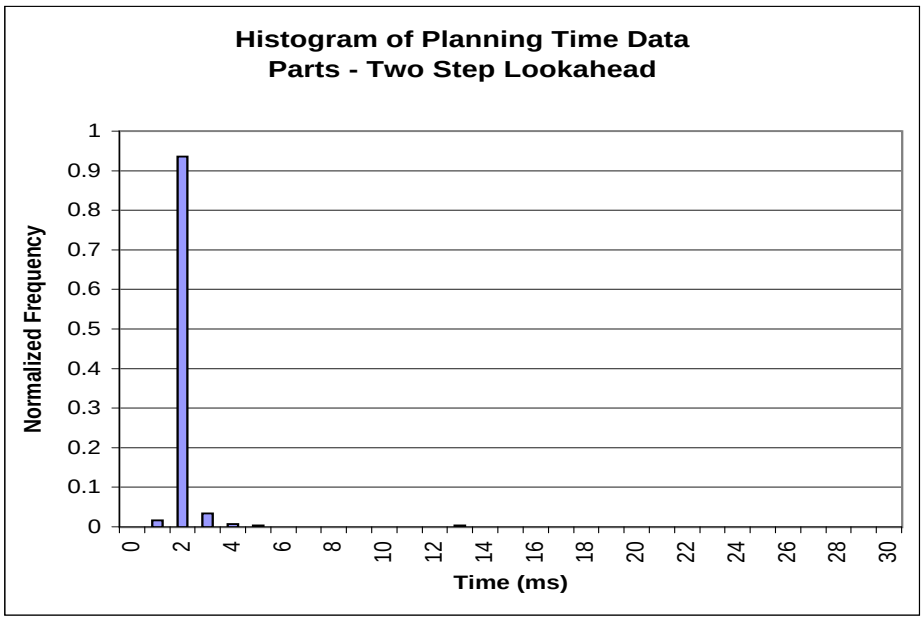

Figure A.2: Planning Time Data - Parts - Two Step Lookahead

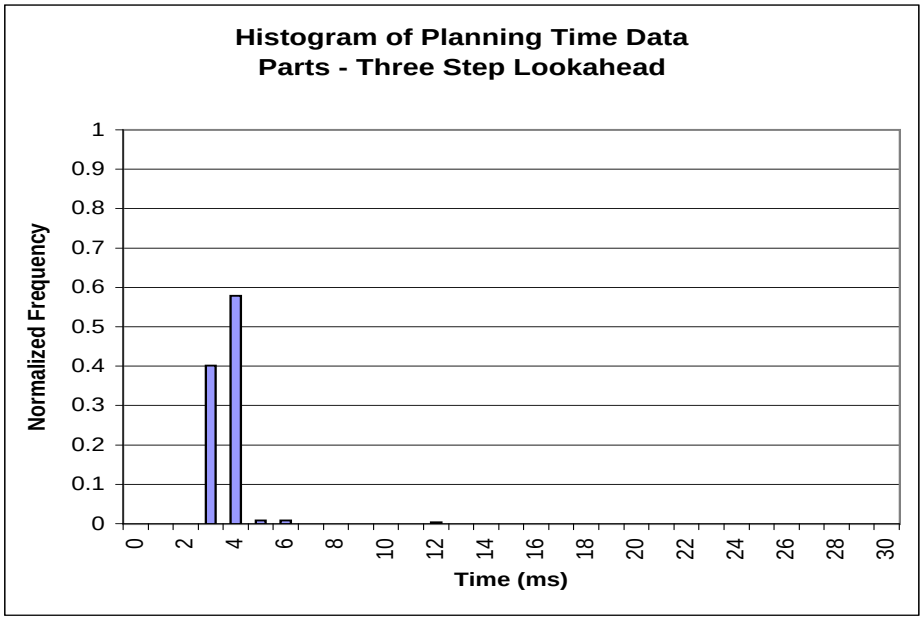

Figure A.3: Planning Time Data - Parts - Three Step Lookahead

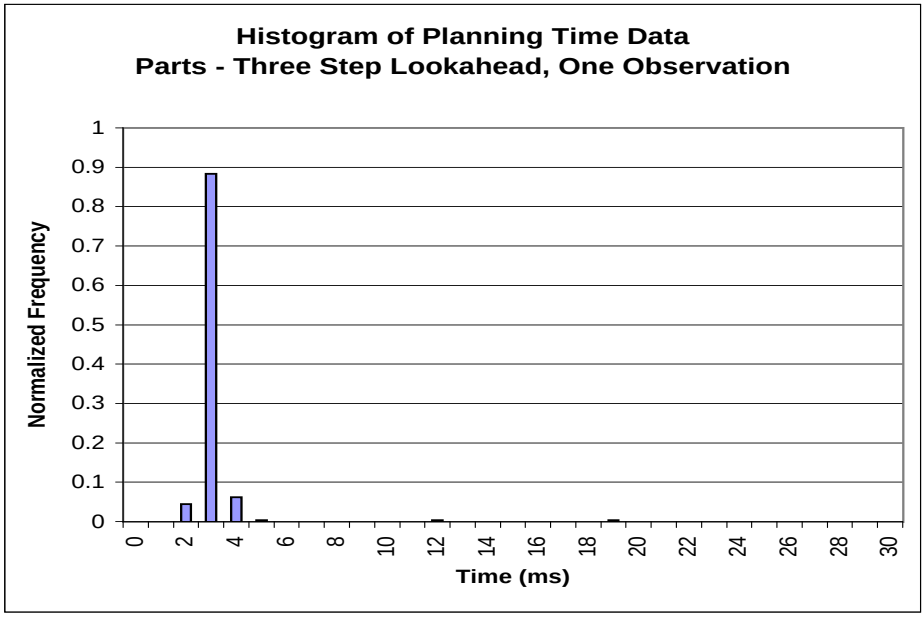

Figure A.4: Planning Time Data - Parts - Three Step Lookahead, One Observation

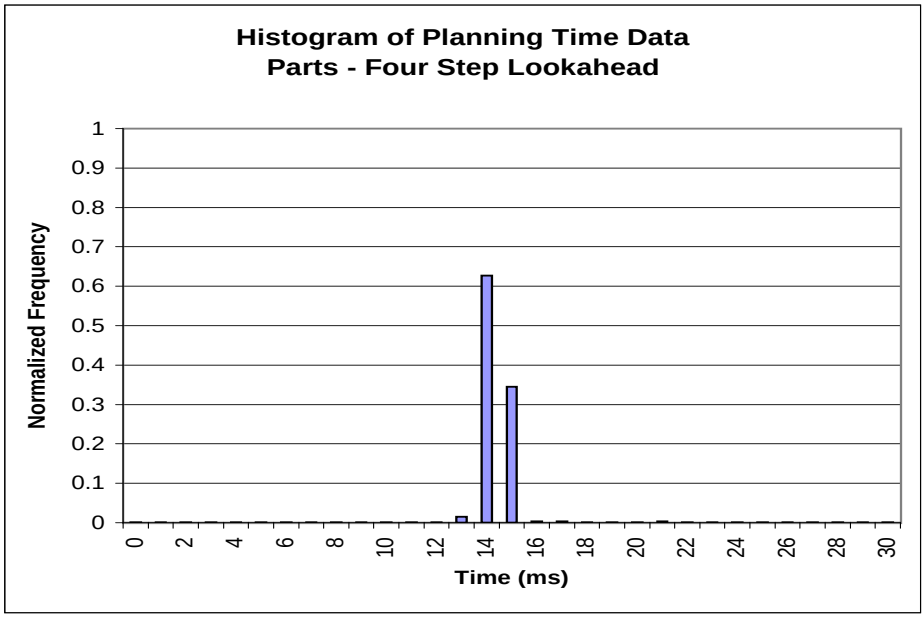

Figure A.5: Planning Time Data - Parts - Four Step Lookahead

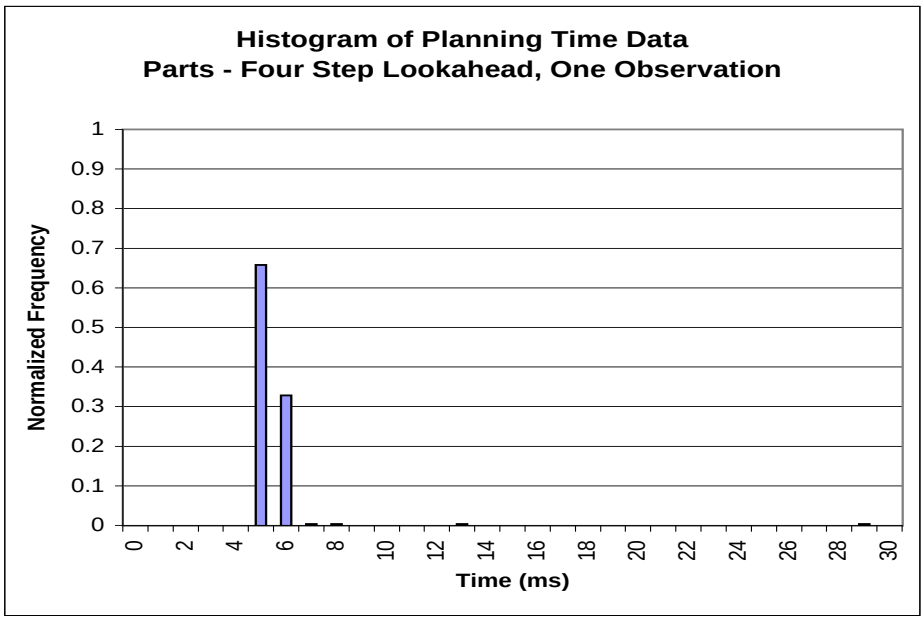

Figure A.6: Planning Time Data - Parts - Four Step Lookahead, One Observation

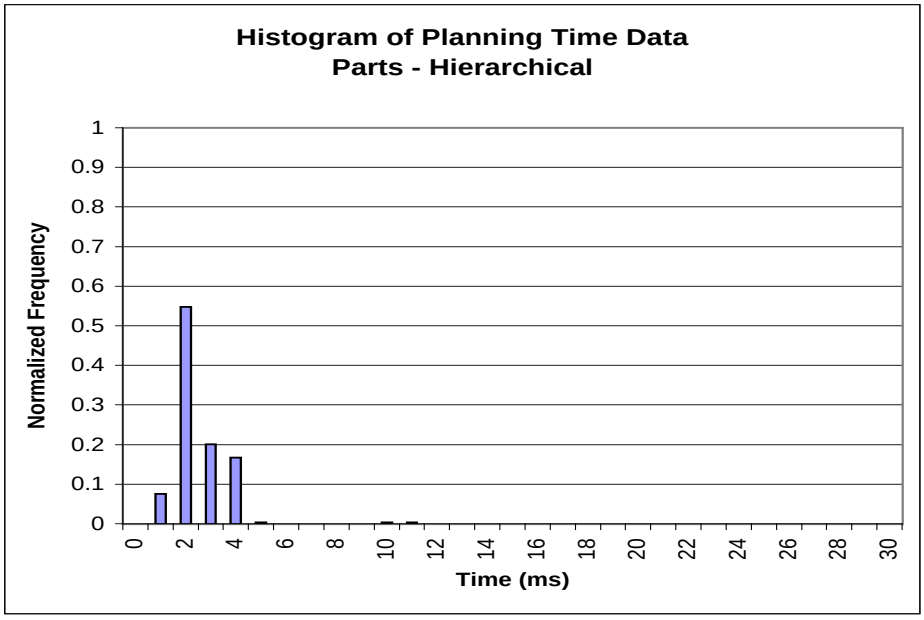

Figure A.7: Planning Time Data - Parts - Hierarchical

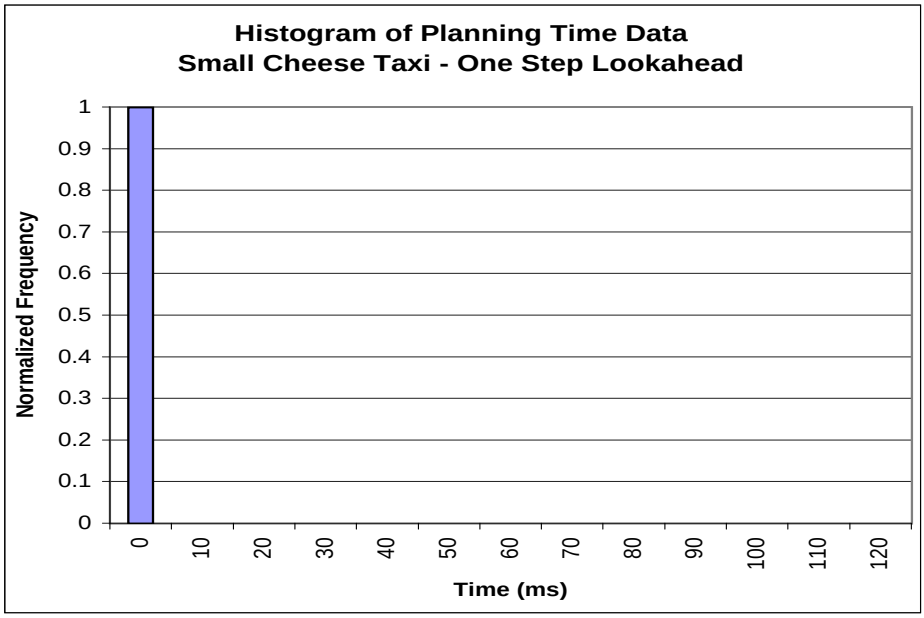

Figure A.8: Planning Time Data - Small Cheese Taxi - One Step Lookahead

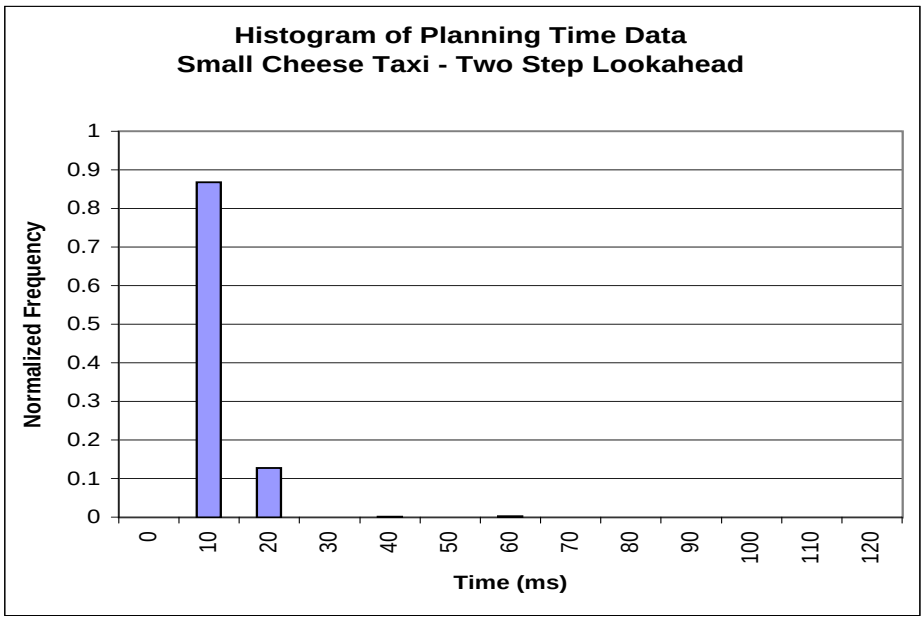

Figure A.9: Planning Time Data - Small Cheese Taxi - Two Step Lookahead

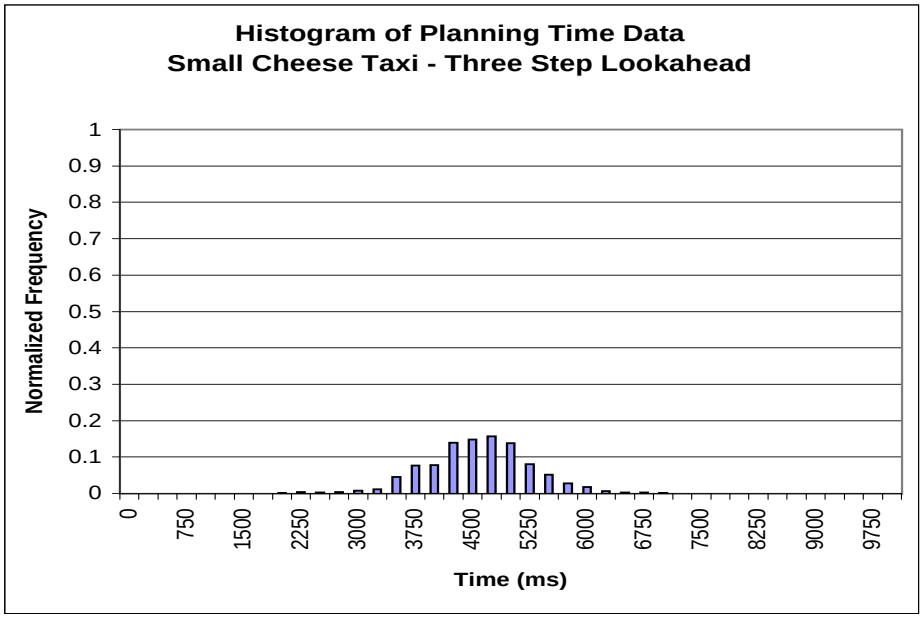

Figure A.10: Planning Time Data - Small Cheese Taxi - Three Step Lookahead

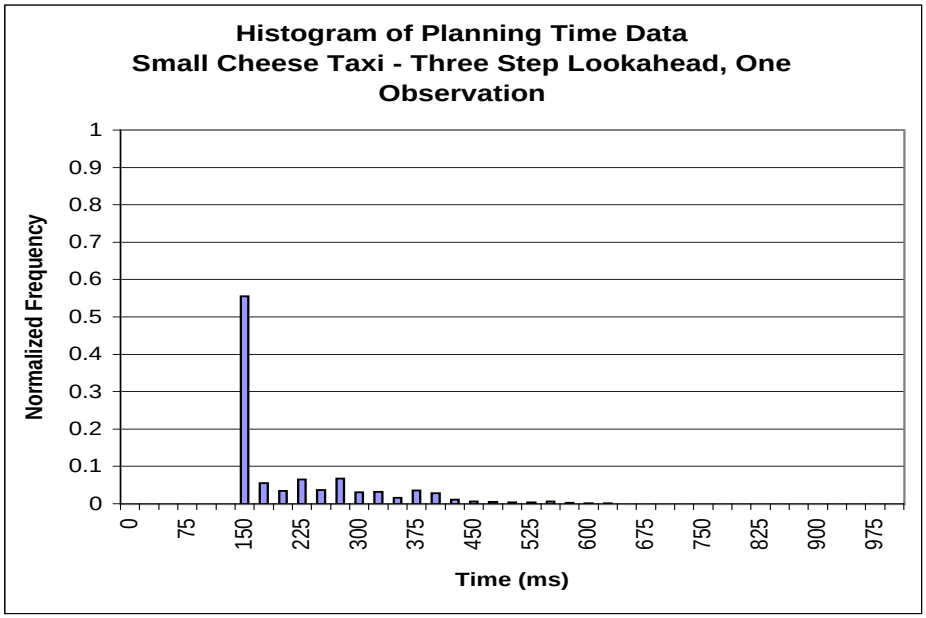

Figure A.11: Planning Time Data - Small Cheese Taxi - Three Step Lookahead, One Observation

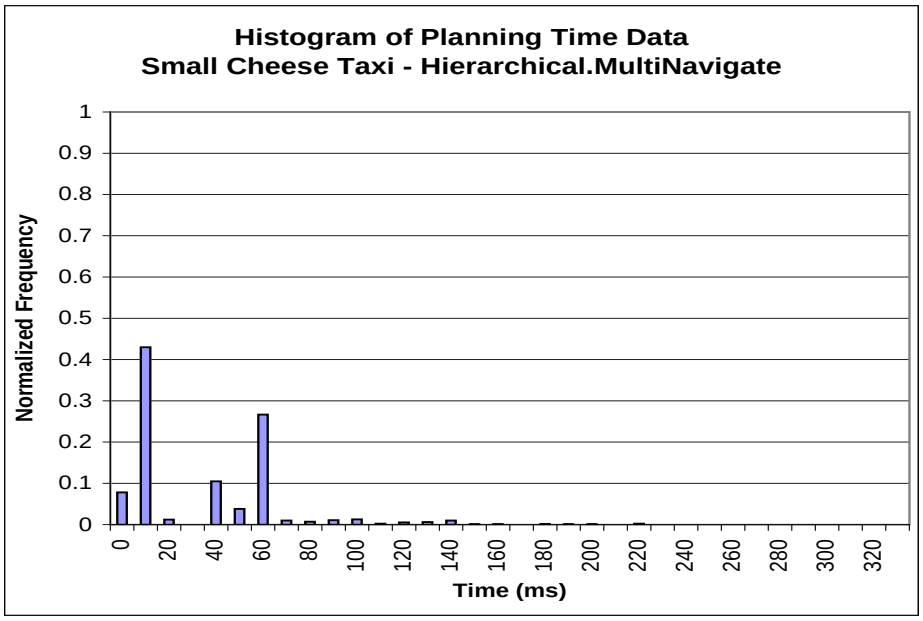

Figure A.12: Planning Time Data - Small Cheese Taxi - Hierarchical (MultiNavigate)

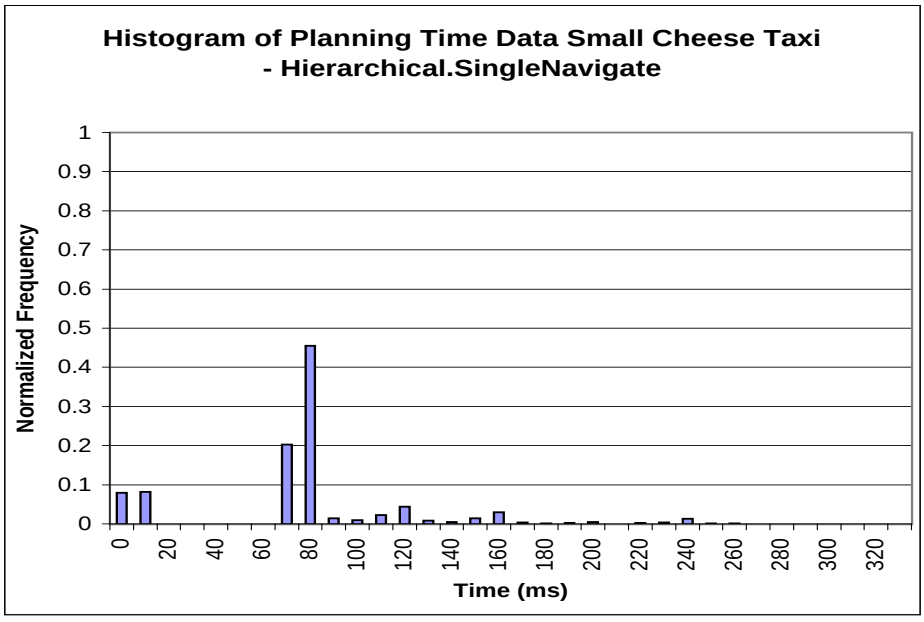

Figure A.13: Planning Time Data - Small Cheese Taxi - Hierarchical (SingleNavigate)

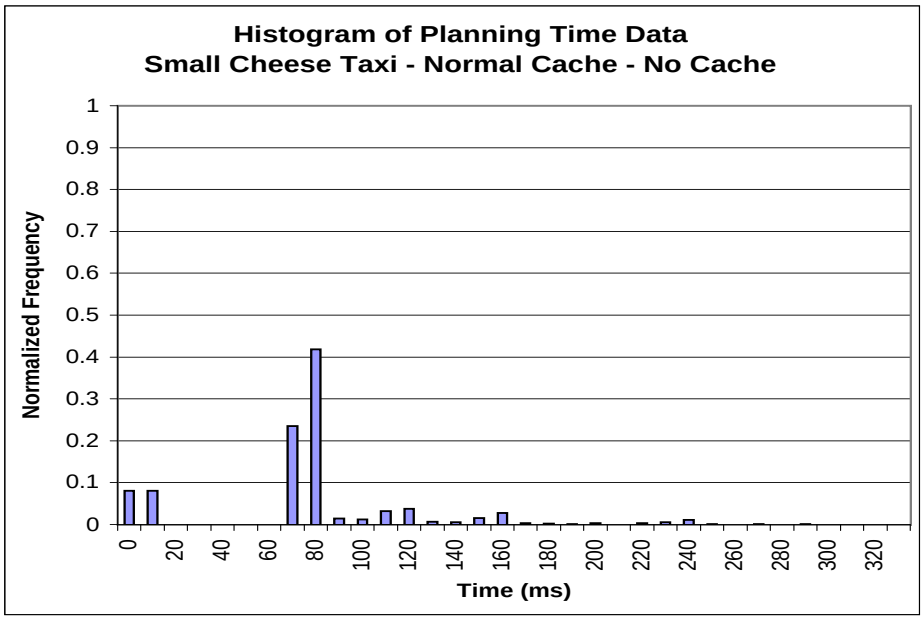

Figure A.14: Planning Time Data - Small Cheese Taxi - Normal Cache - Cache Size = 0

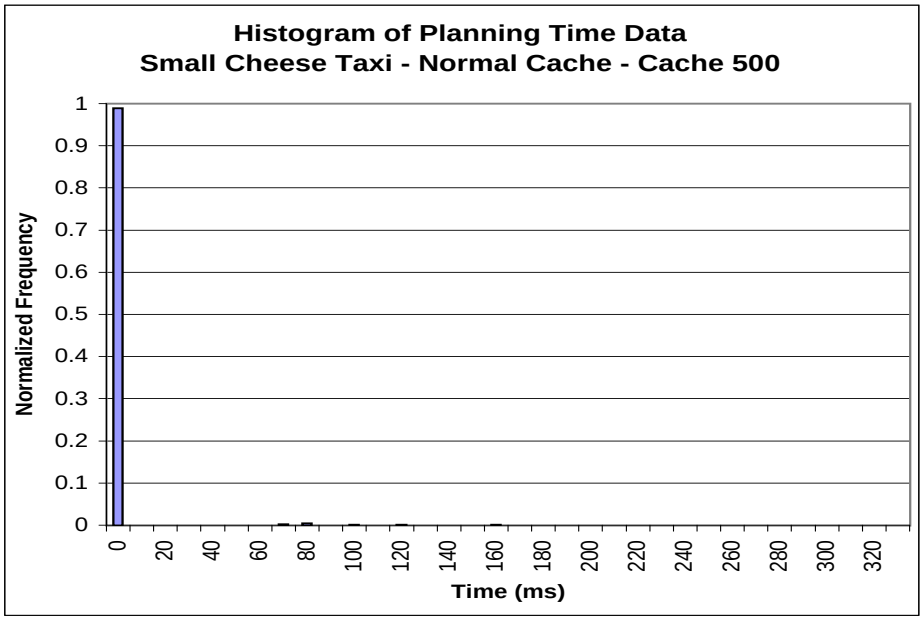

Figure A.15: Planning Time Data - Small Cheese Taxi - Normal Cache - Cache Size = 500

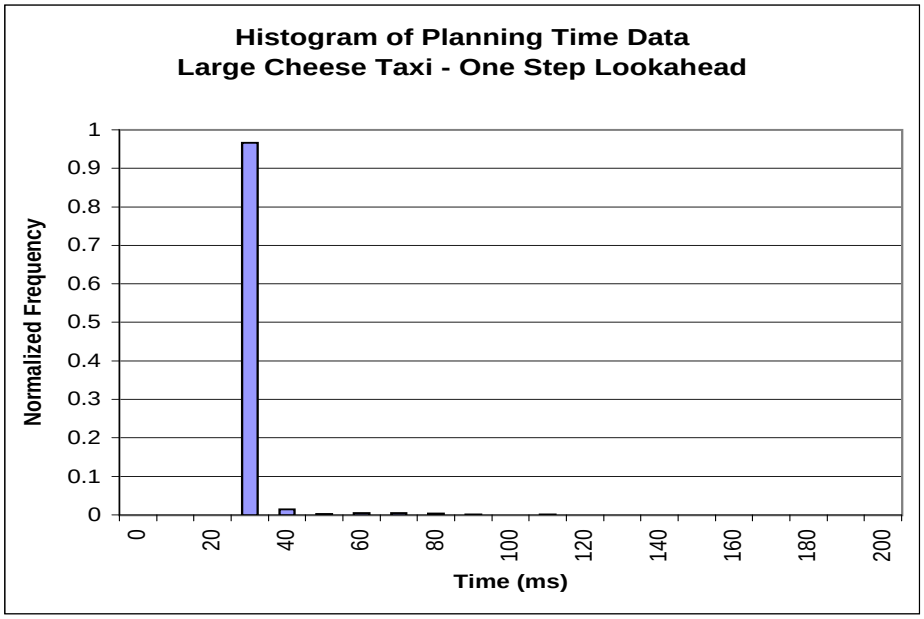

Figure A.16: Planning Time Data - Large Cheese Taxi - One Step Lookahead

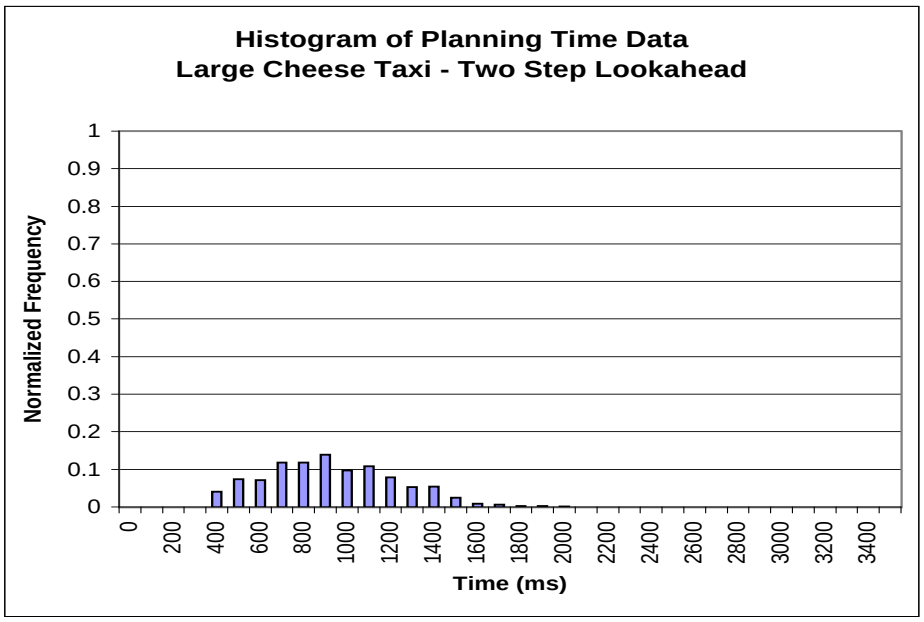

Figure A.17: Planning Time Data - Large Cheese Taxi - Two Step Lookahead

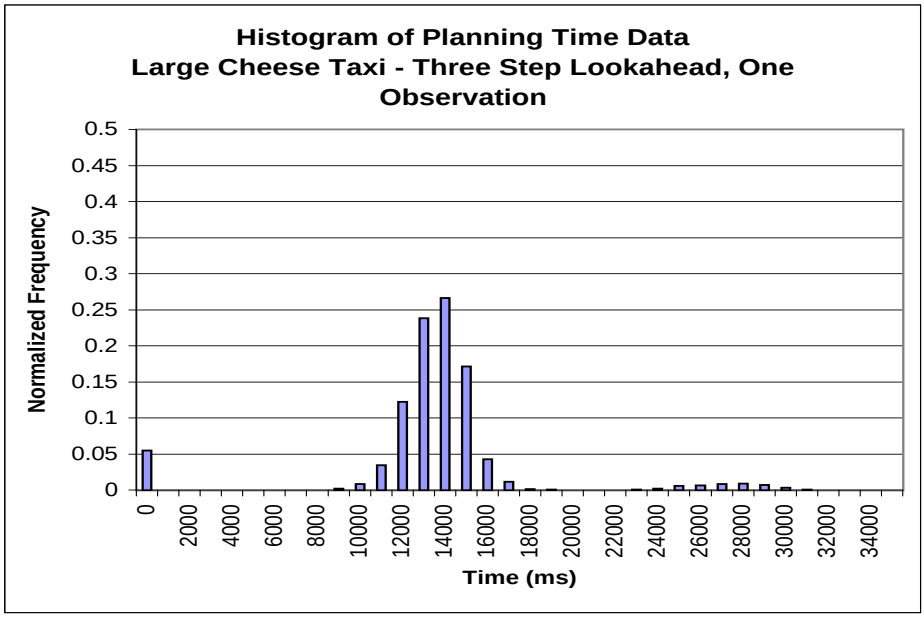

Figure A.18: Planning Time Data - Large Cheese Taxi - Three Step Lookahead, One Observation

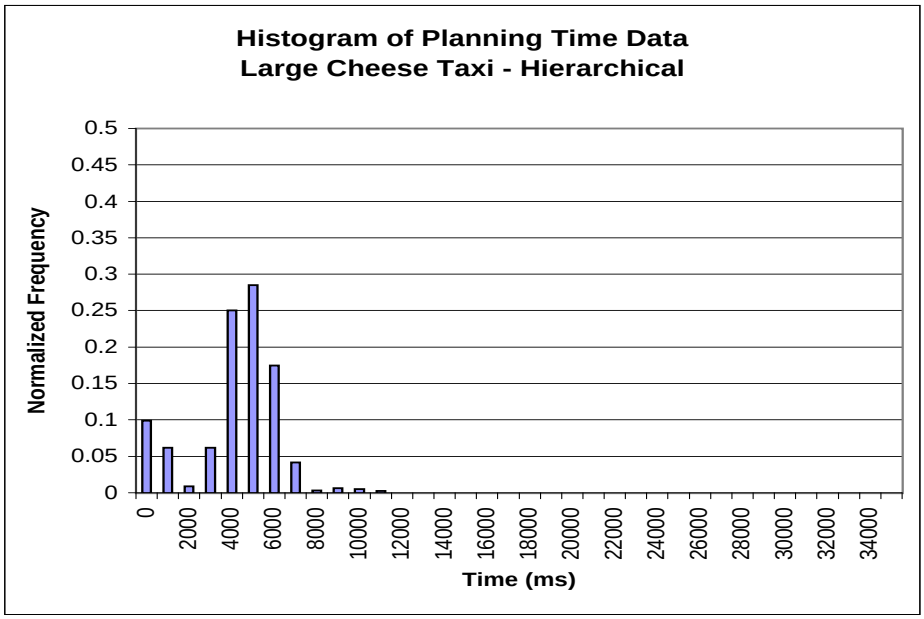

Figure A.19: Planning Time Data - Large Cheese Taxi - Hierarchical

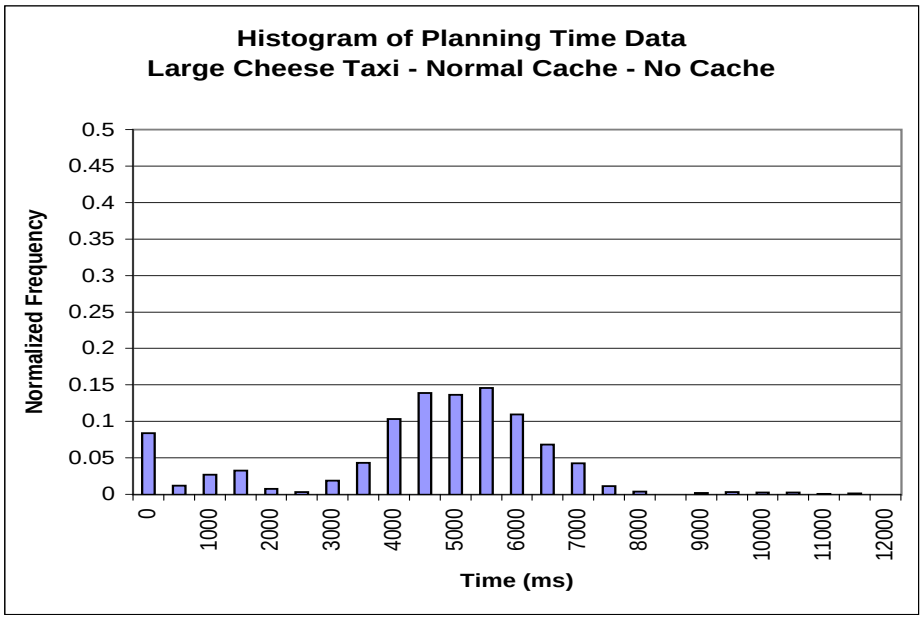

Figure A.20: Planning Time Data - Large Cheese Taxi - Normal Cache - Cache Size = 0

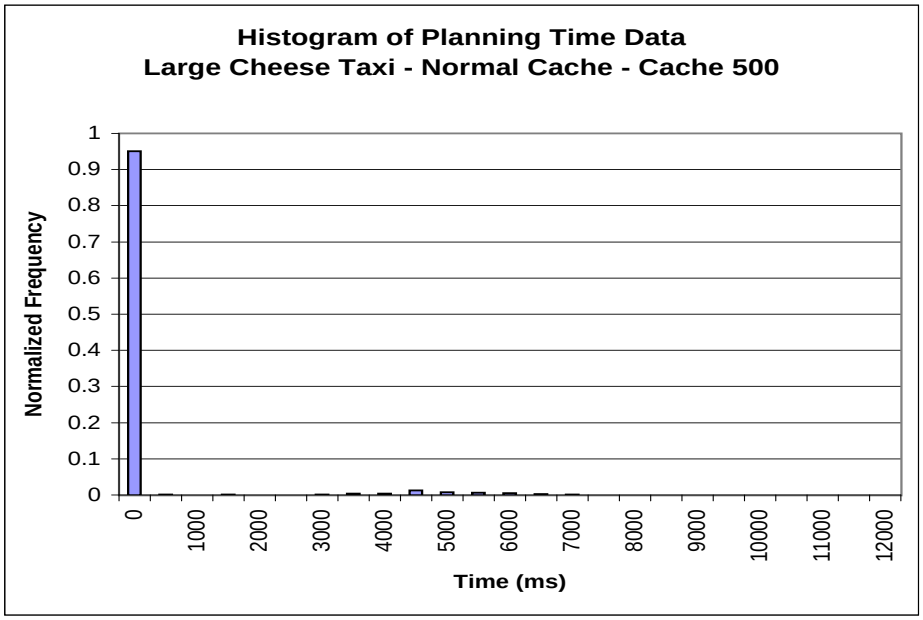

Figure A.21: Planning Time Data - Large Cheese Taxi - Normal Cache - Cache Size = 500

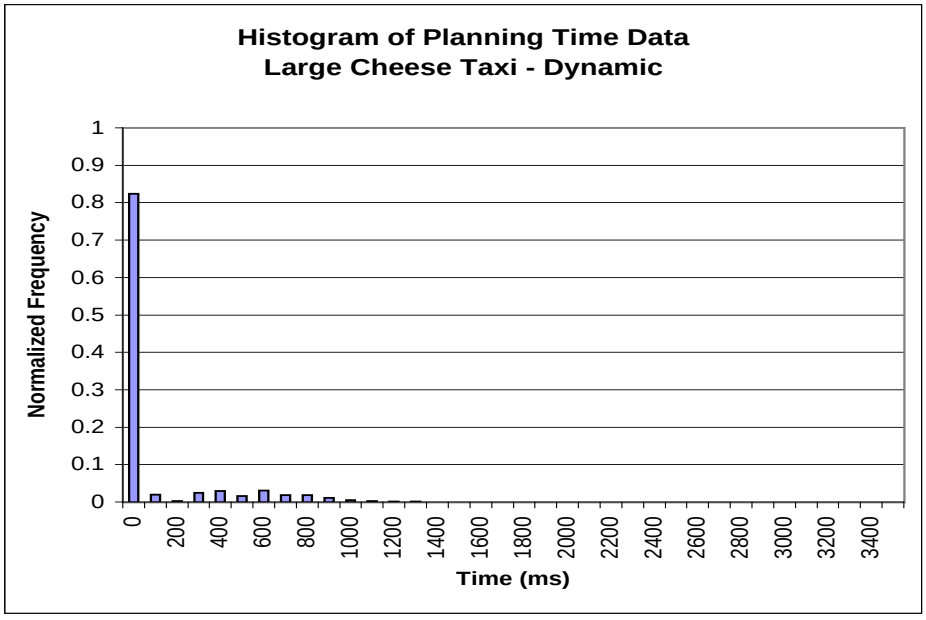

Figure A.22: Planning Time Data - Large Cheese Taxi - Dynamic Build

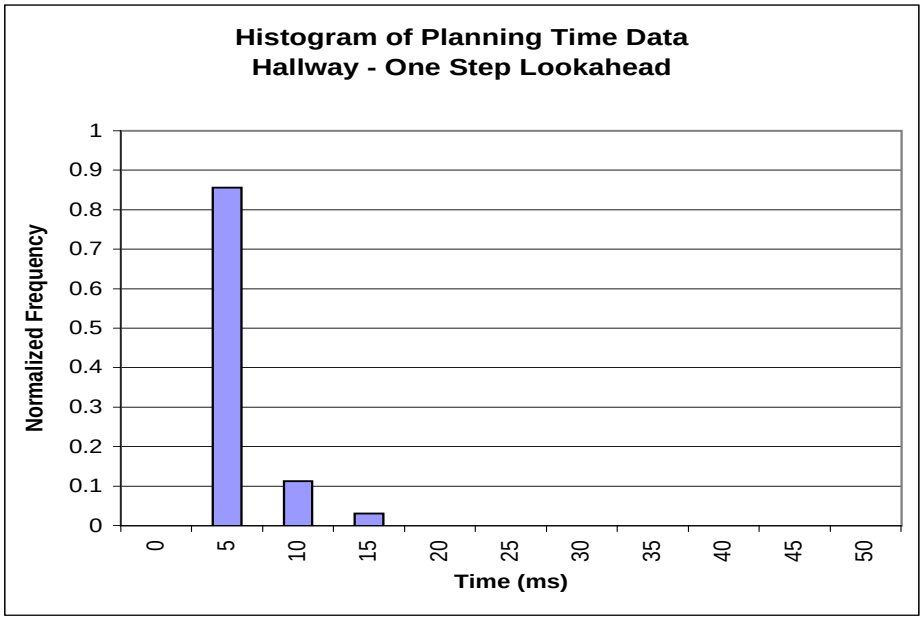

Figure A.23: Planning Time Data - Hallway - One Step Lookahead

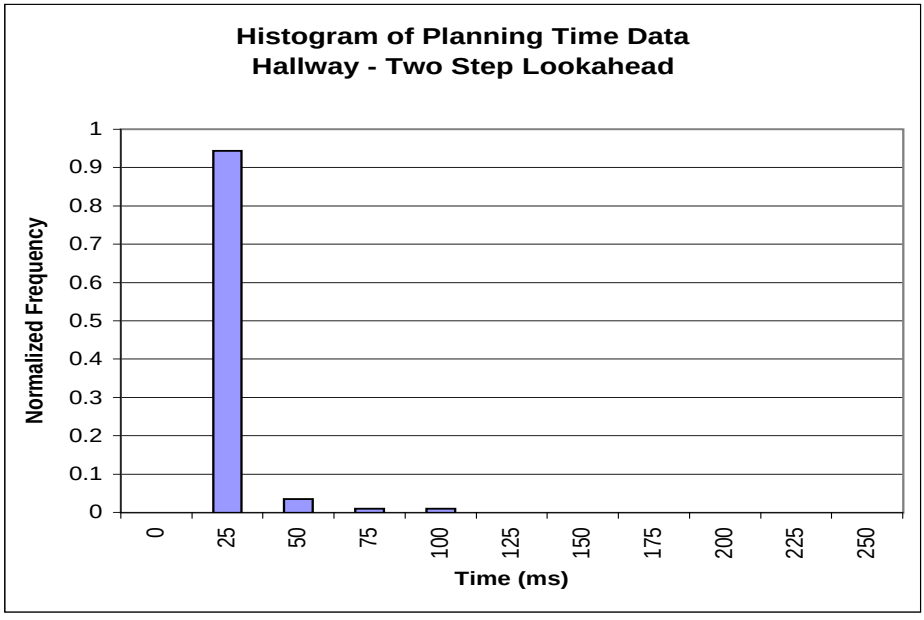

Figure A.24: Planning Time Data - Hallway - Two Step Lookahead

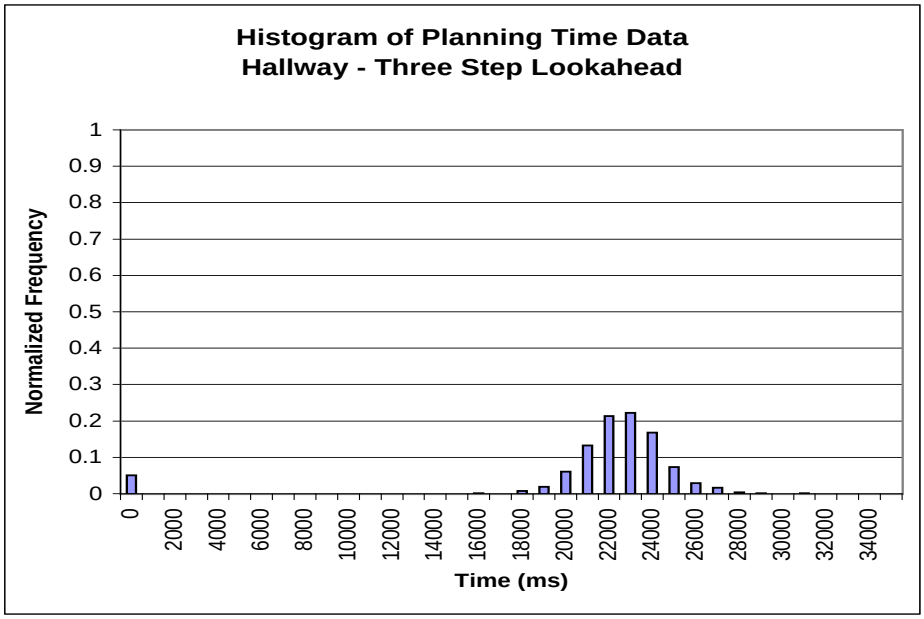

Figure A.25: Planning Time Data - Hallway - Three Step Lookahead

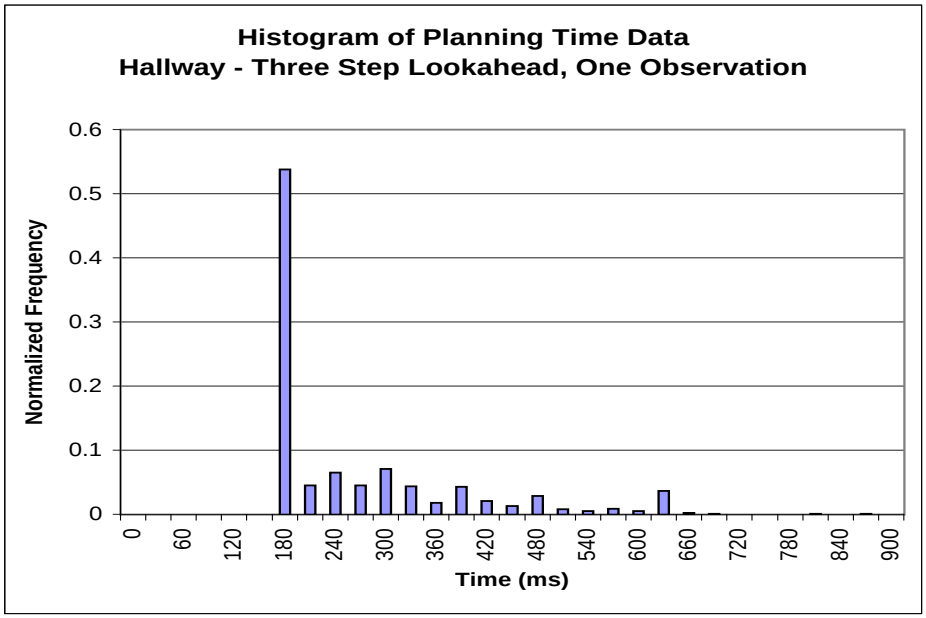

Figure A.26: Planning Time Data - Hallway - Three Step Lookahead, One Observation

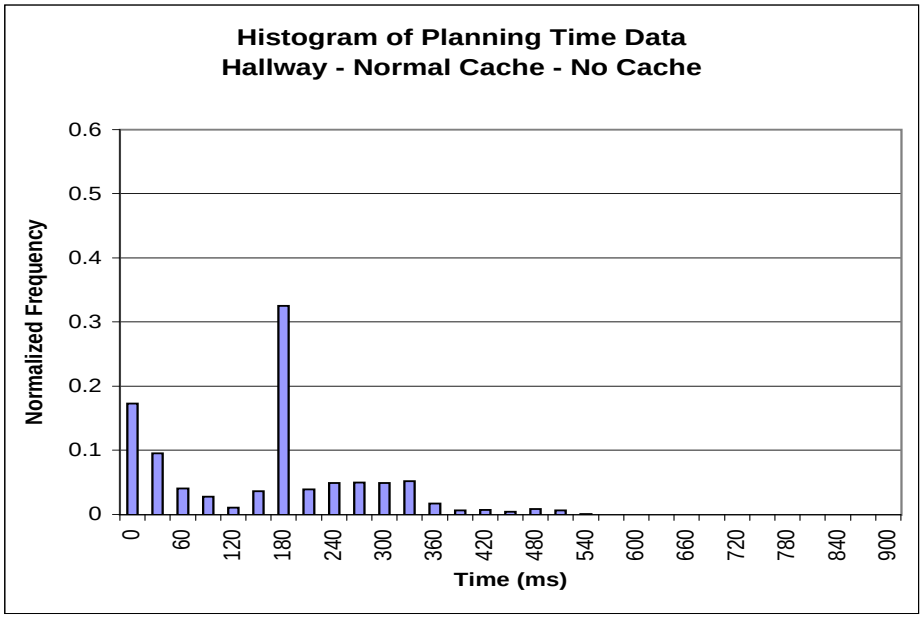

Figure A.27: Planning Time Data - Hallway - Normal Cache - Cache Size = 0

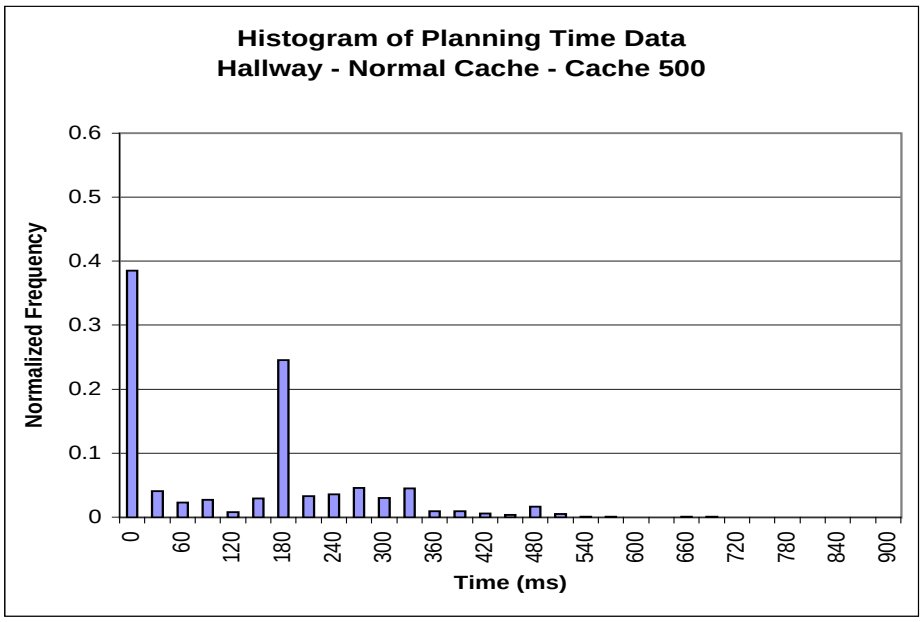

Figure A.28: Planning Time Data - Hallway - Normal Cache - Cache Size = 500

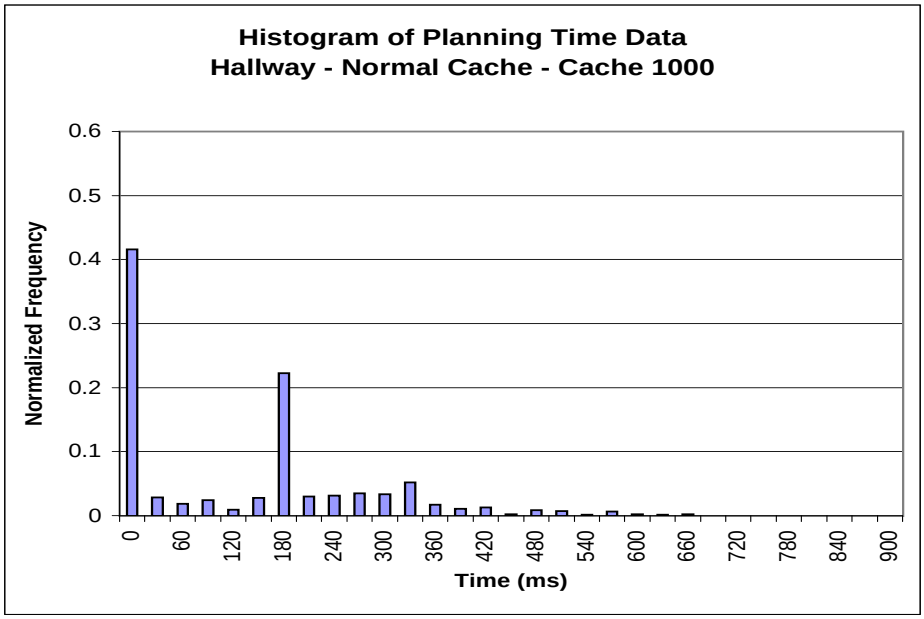

Figure A.29: Planning Time Data - Hallway - Normal Cache - Cache Size = 1000

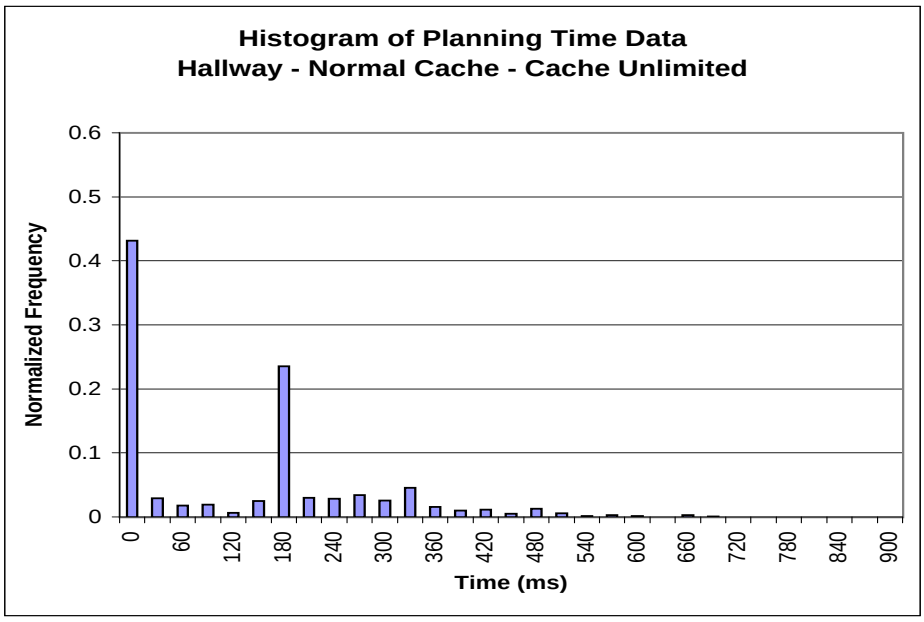

Figure A.30: Planning Time Data - Hallway - Normal Cache - Cache Size = Unlimited

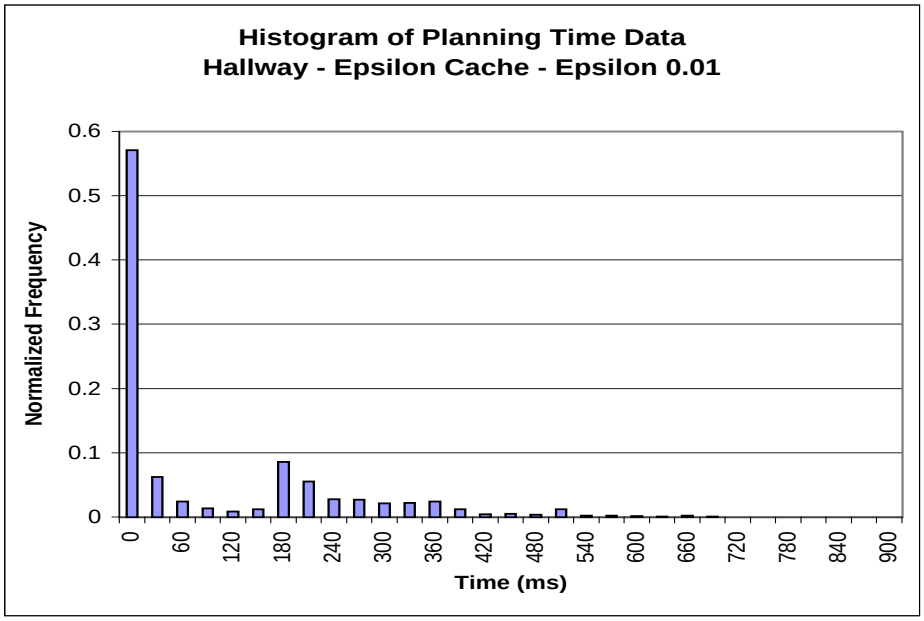

Figure A.31: Planning Time Data - Hallway - Epsilon Cache - Epsilon = 0.01

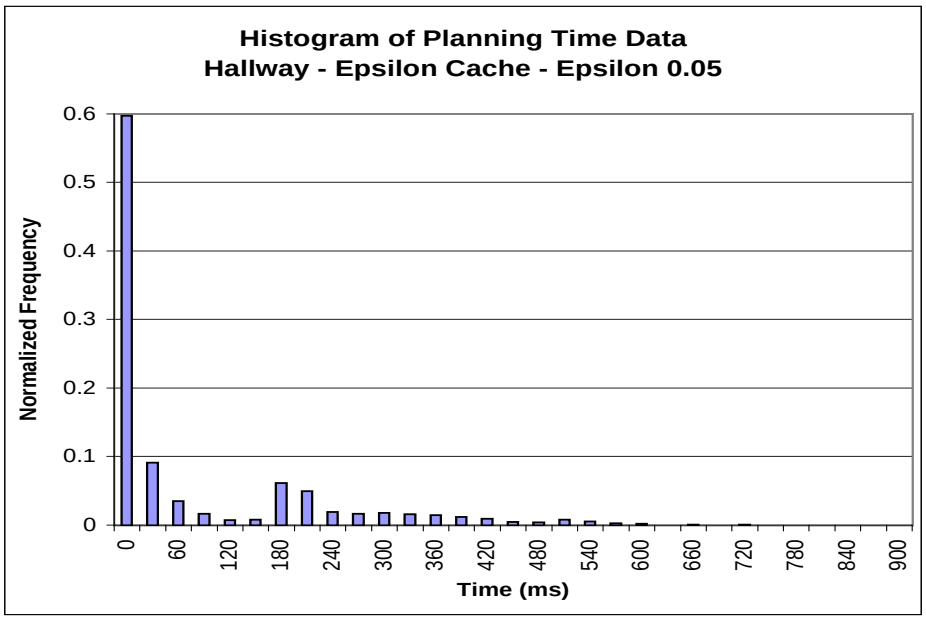

Figure A.32: Planning Time Data - Hallway - Epsilon Cache - Epsilon = 0.05

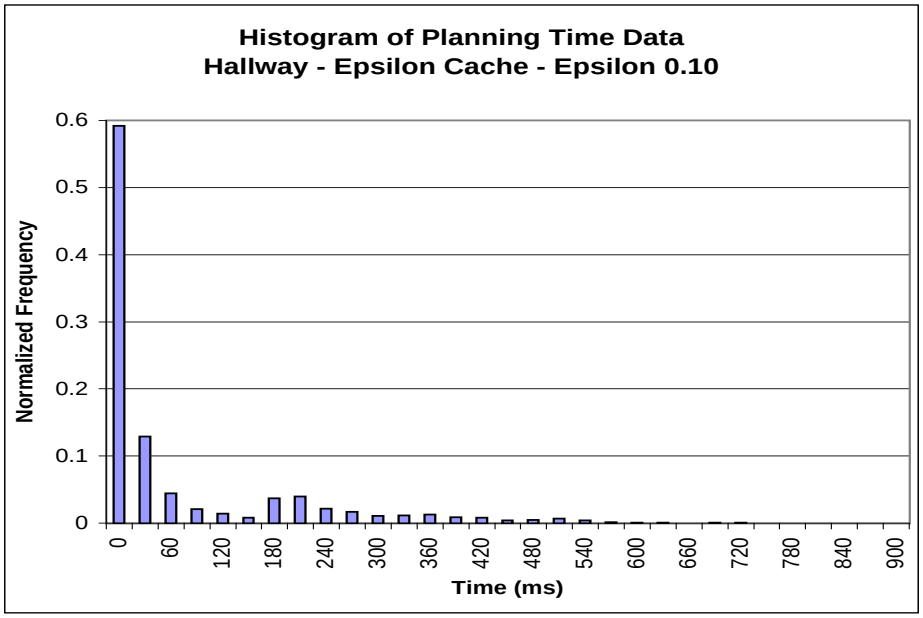

Figure A.33: Planning Time Data - Hallway - Epsilon Cache - Epsilon = 0.10

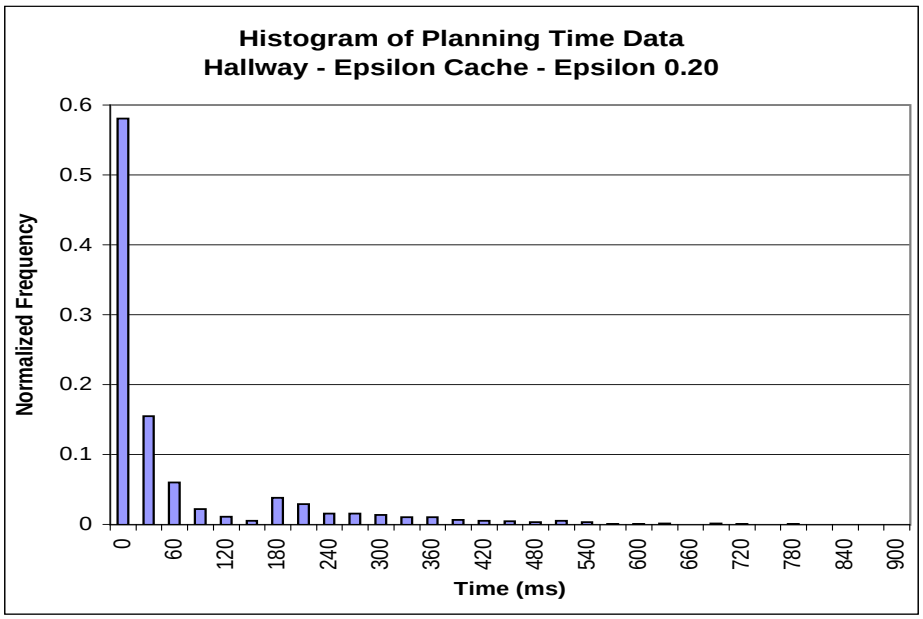

Figure A.34: Planning Time Data - Hallway - Epsilon Cache - Epsilon = 0.20

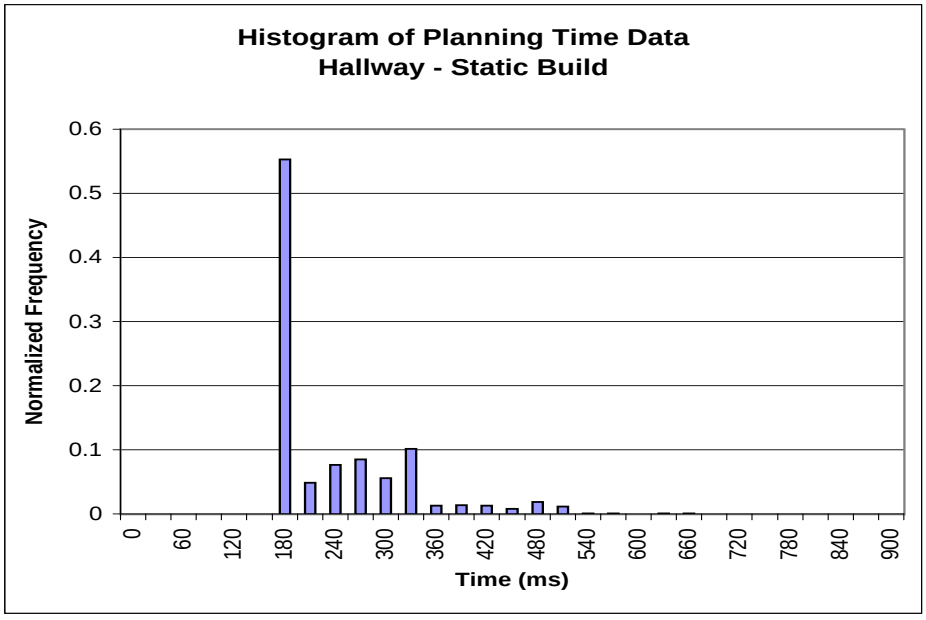

Figure A.35: Planning Time Data - Hallway - Static Build

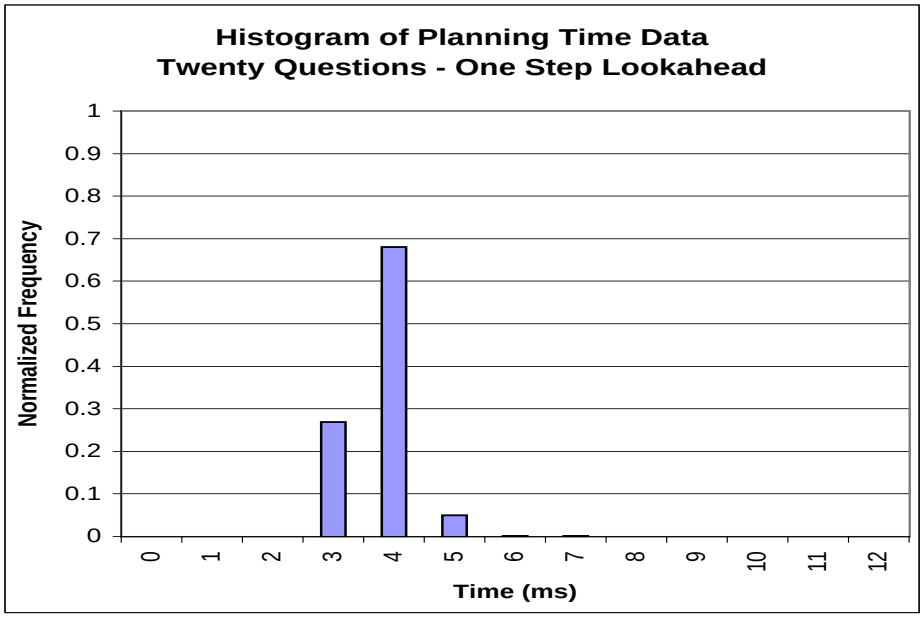

Figure A.36: Planning Time Data - Twenty Questions - One Step Lookahead

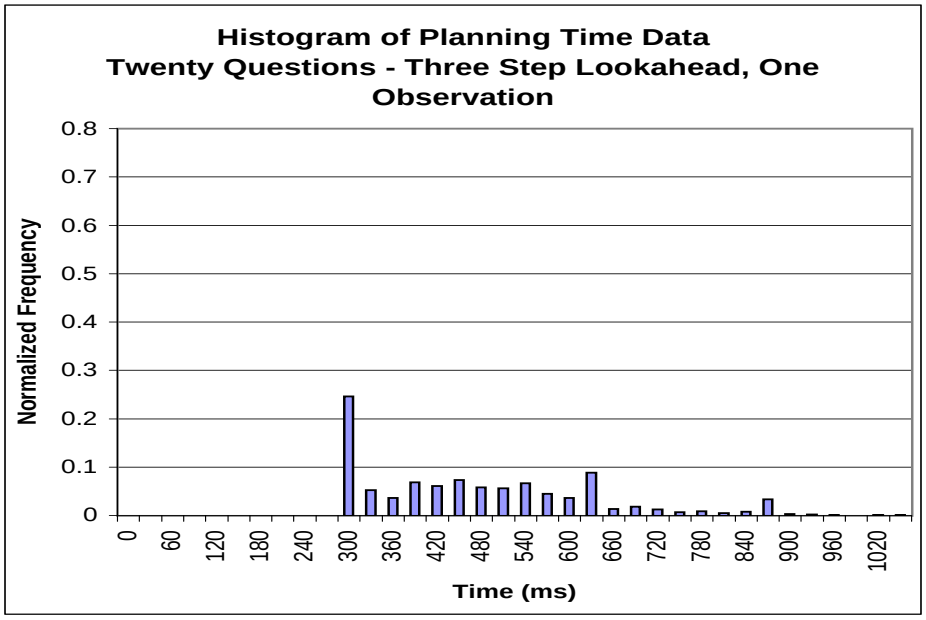

Figure A.37: Planning Time Data - Twenty Questions - Three Step Lookahead, One Observation

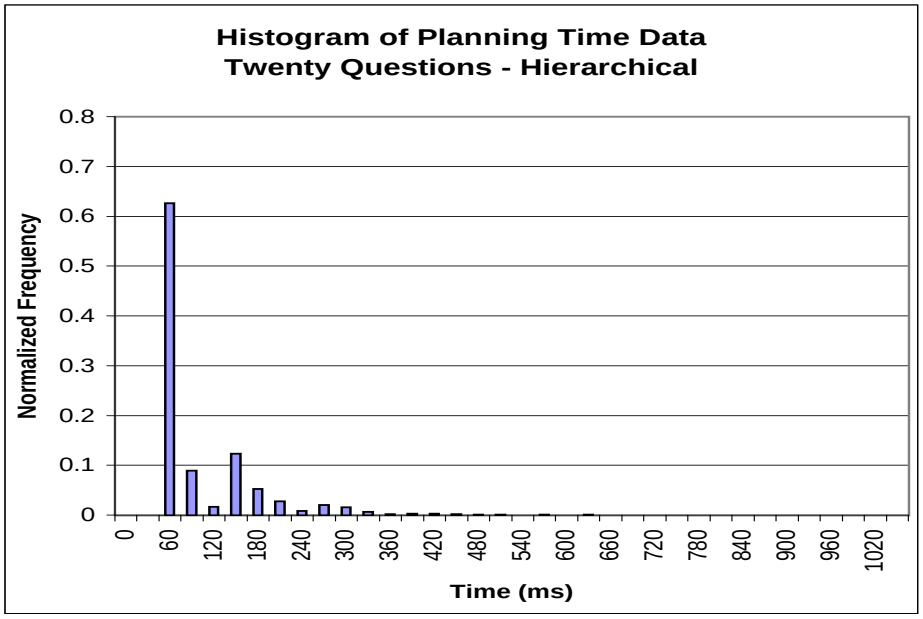

Figure A.38: Planning Time Data - Twenty Questions - Hierarchical

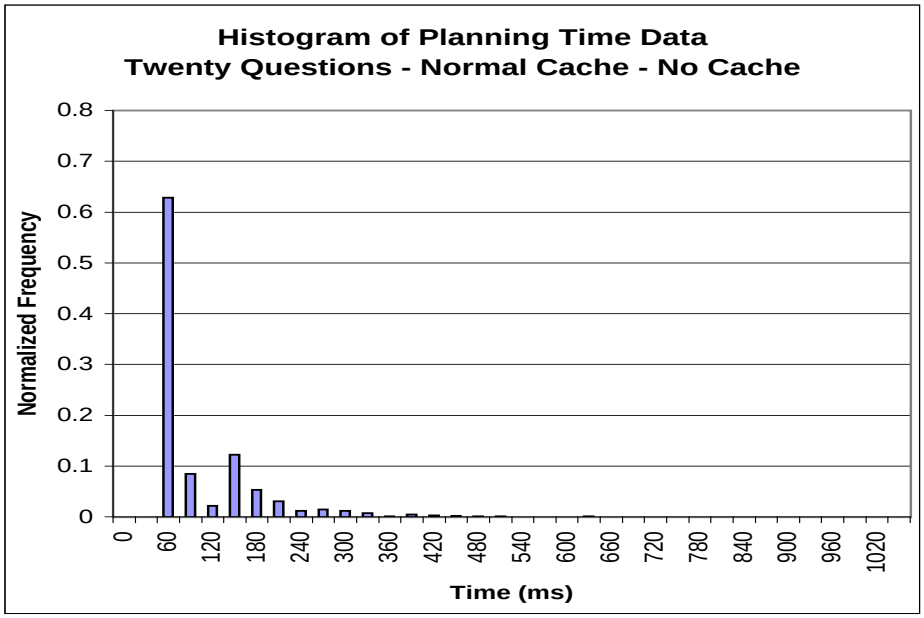

Figure A.39: Planning Time Data - Twenty Questions - Normal Cache - Cache Size = 0

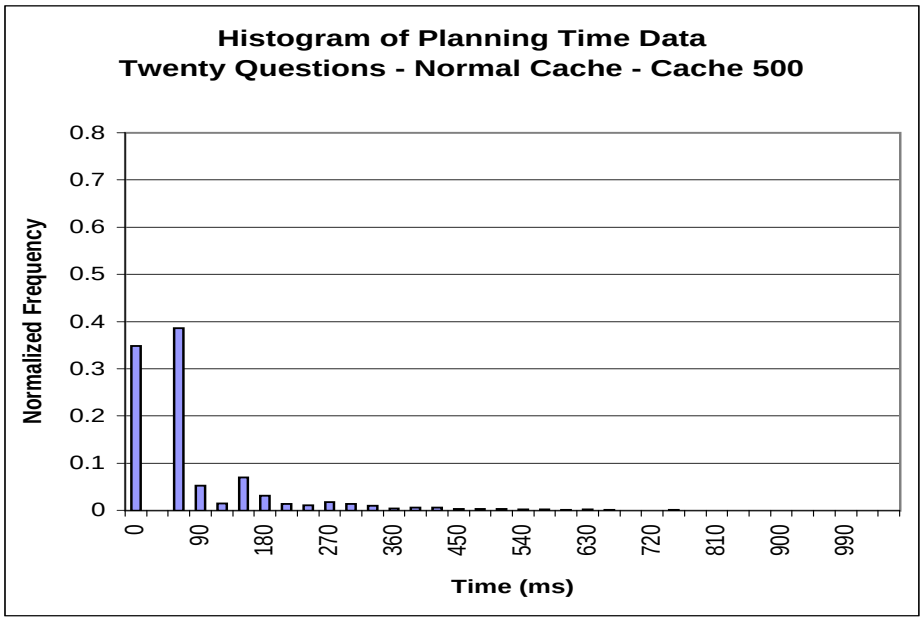

Figure A.40: Planning Time Data - Twenty Questions - Normal Cache - Cache Size = 500

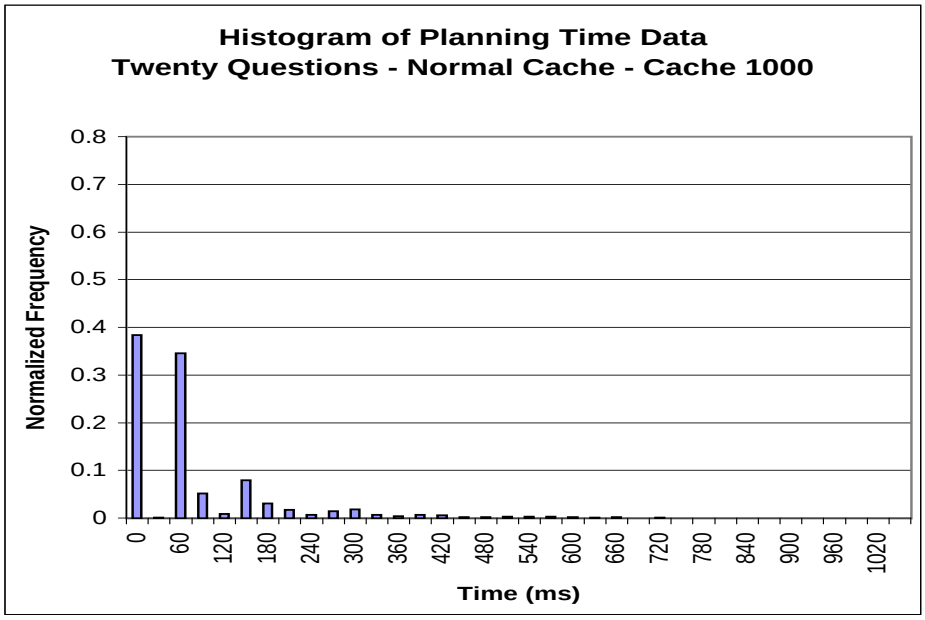

Figure A.41: Planning Time Data - Twenty Questions - Normal Cache - Cache Size = 1000

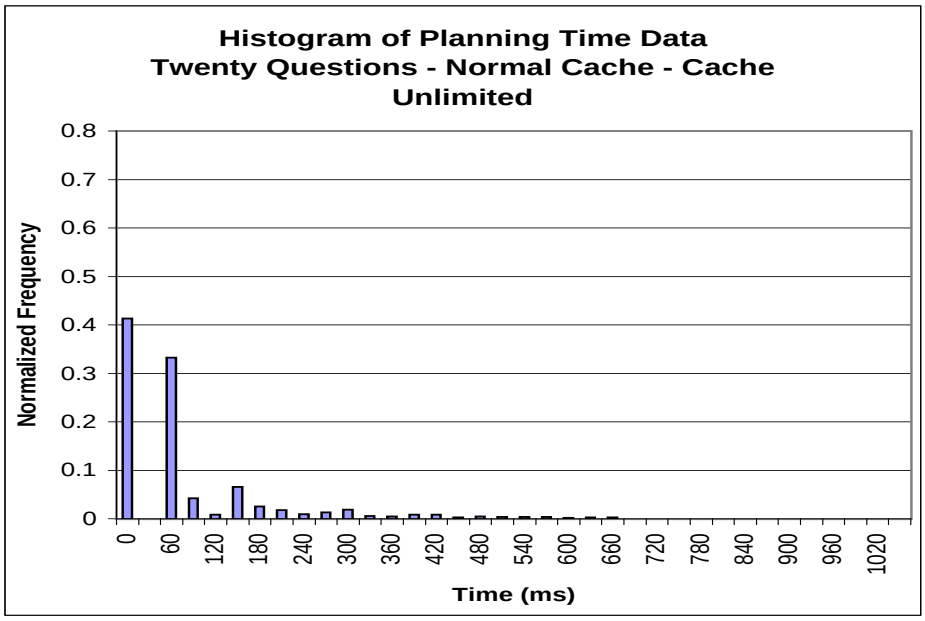

Figure A.42: Planning Time Data - Twenty Questions - Normal Cache - Cache Size = Unlimited

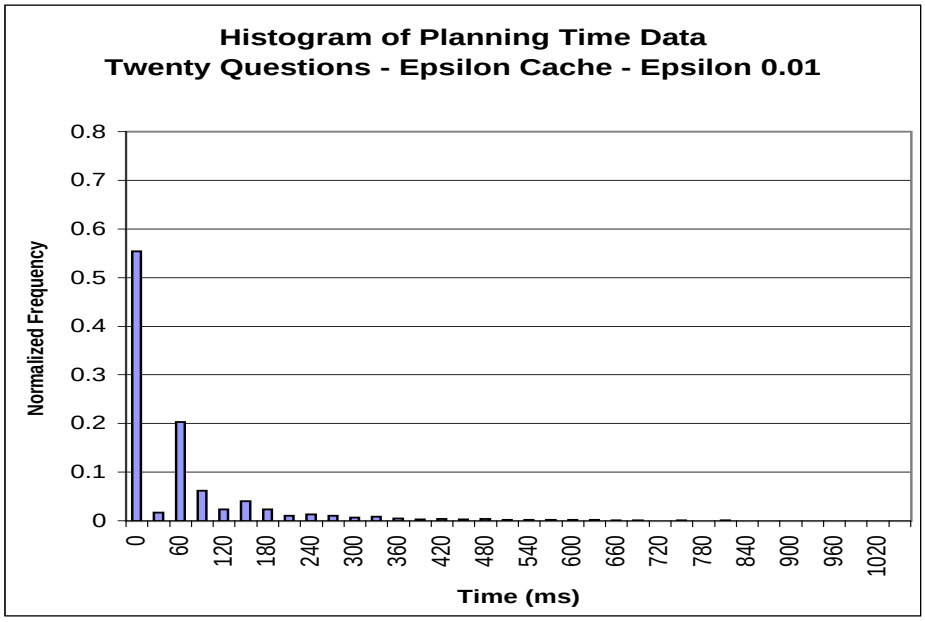

Figure A.43: Planning Time Data - Twenty Questions - Epsilon Cache - Epsilon = 0.01

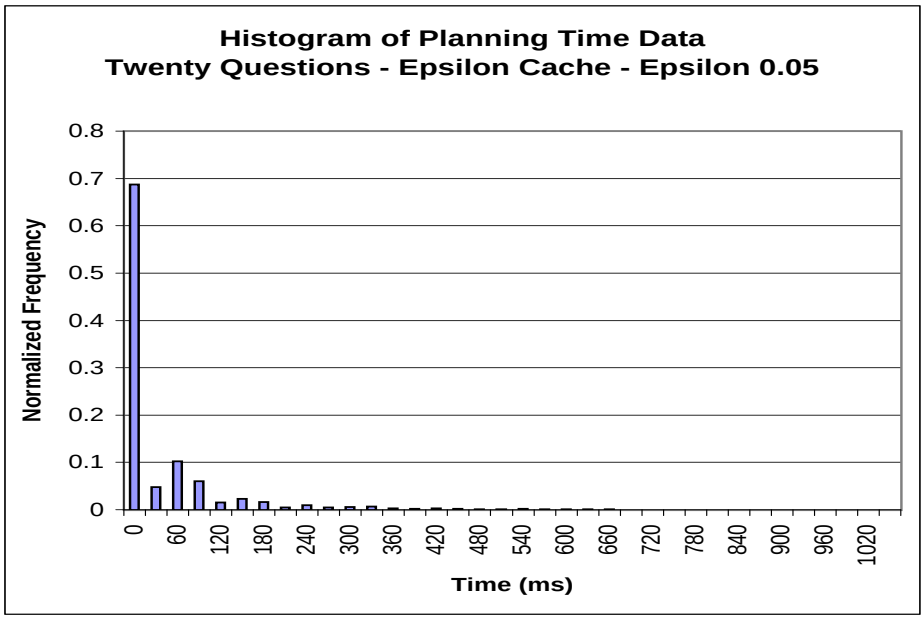

Figure A.44: Planning Time Data - Twenty Questions - Epsilon Cache - Epsilon = 0.05

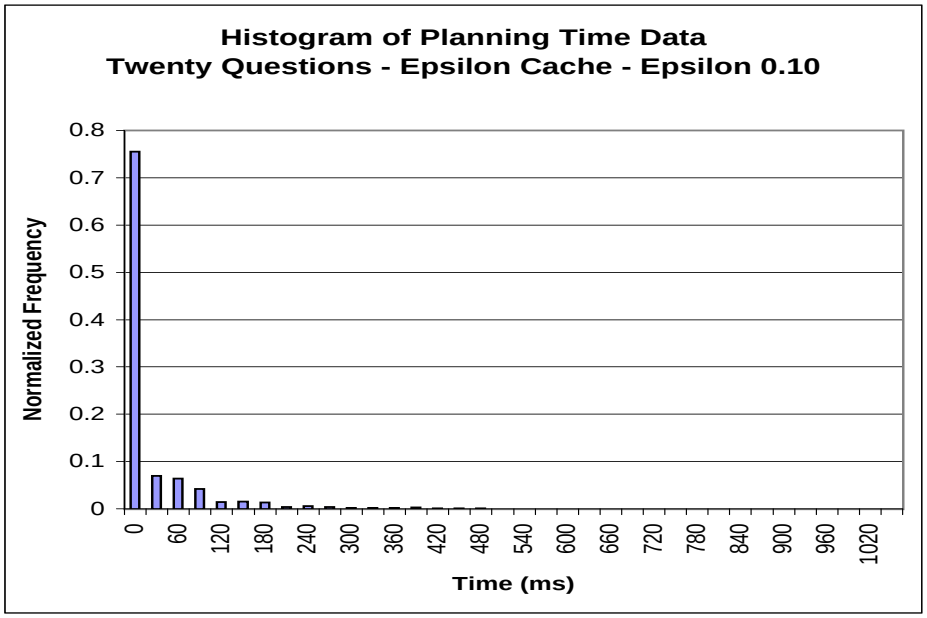

Figure A.45: Planning Time Data - Twenty Questions - Epsilon Cache - Epsilon = 0.10

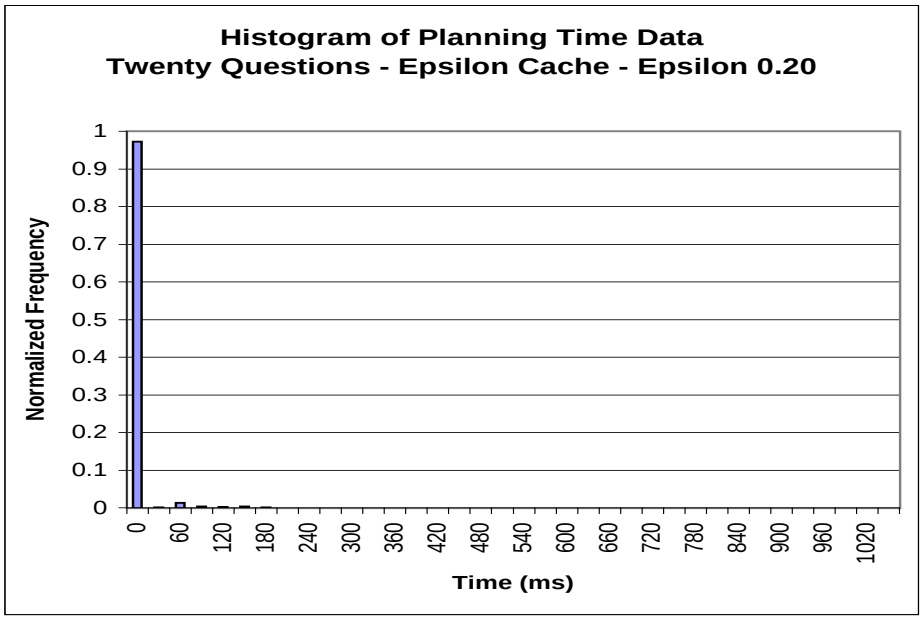

Figure A.46: Planning Time Data - Twenty Questions - Epsilon Cache - Epsilon = 0.20

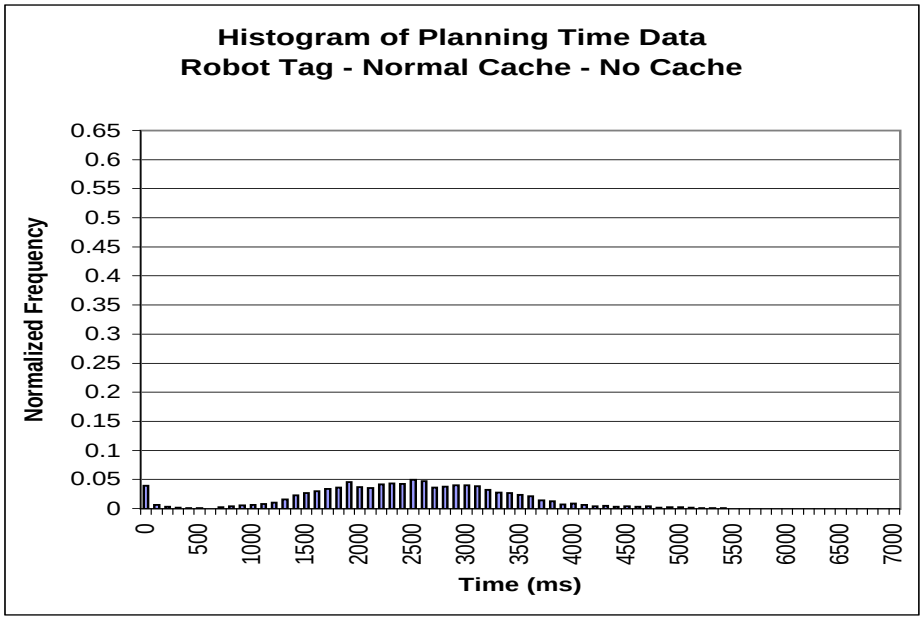

Figure A.47: Planning Time Data - Robot Tag - Normal Cache - Cache Size = 0

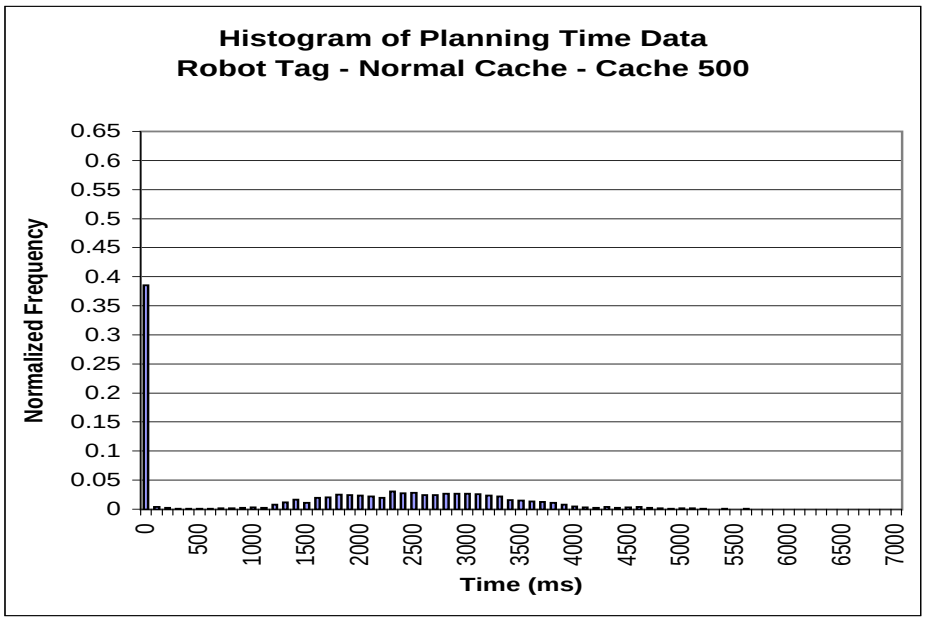

Figure A.48: Planning Time Data - Robot Tag - Normal Cache - Cache Size = 500

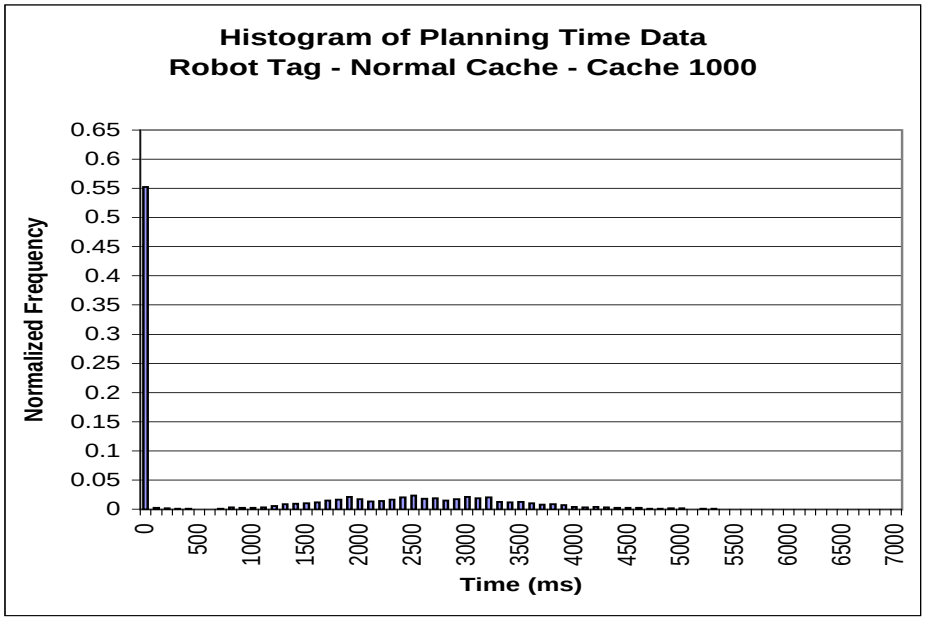

Figure A.49: Planning Time Data - Robot Tag - Normal Cache - Cache Size = 1000

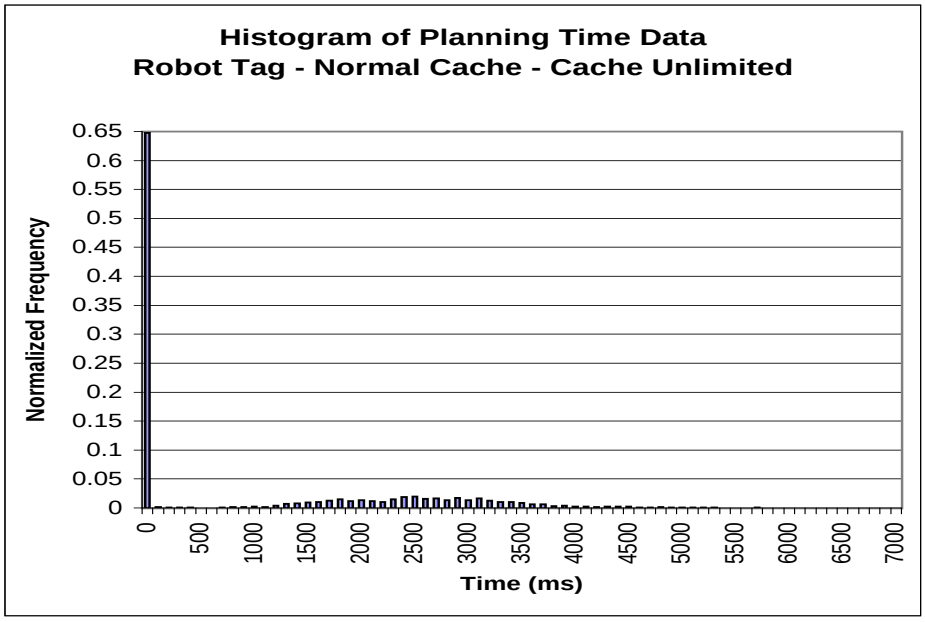

Figure A.50: Planning Time Data - Robot Tag - Normal Cache - Cache Size = Unlimited

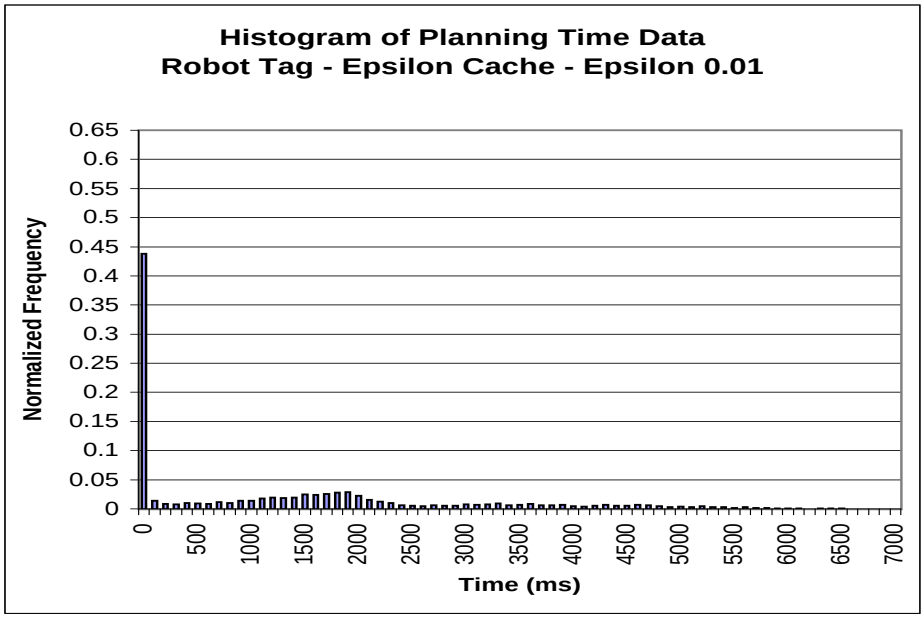

Figure A.51: Planning Time Data - Robot Tag - Epsilon Cache - Epsilon = 0.01

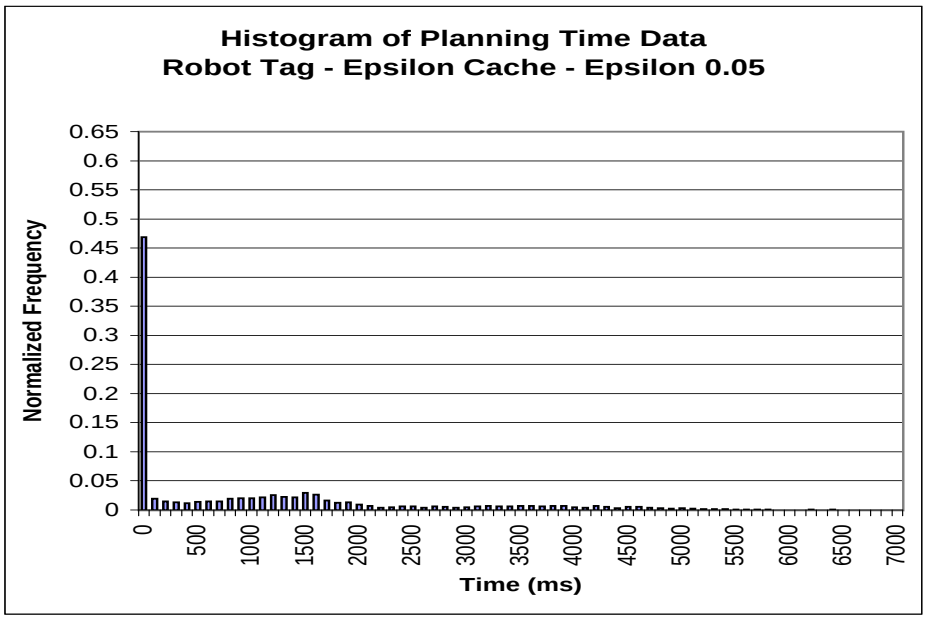

Figure A.52: Planning Time Data - Robot Tag - Epsilon Cache - Epsilon = 0.05

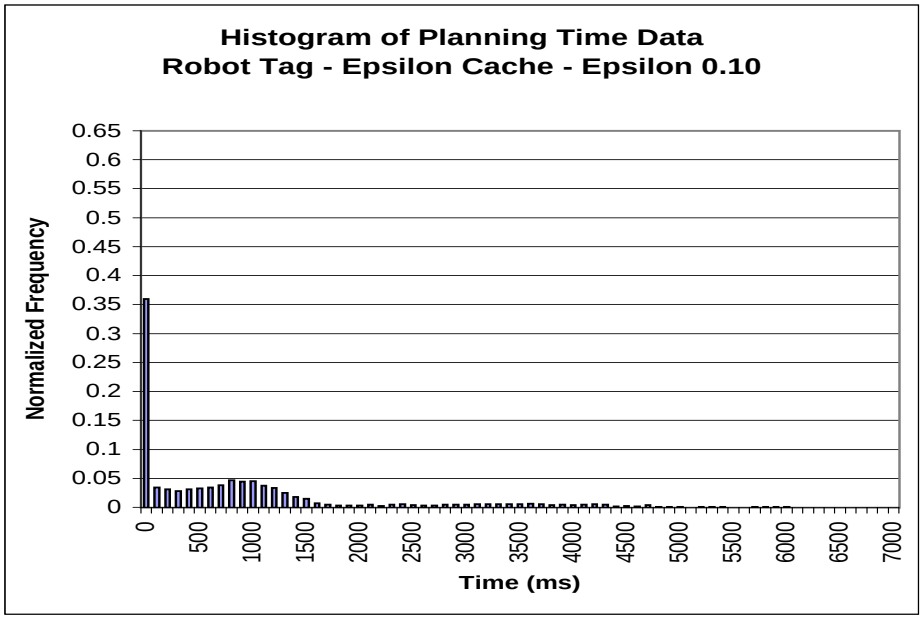

Figure A.53: Planning Time Data - Robot Tag - Epsilon Cache - Epsilon = 0.10

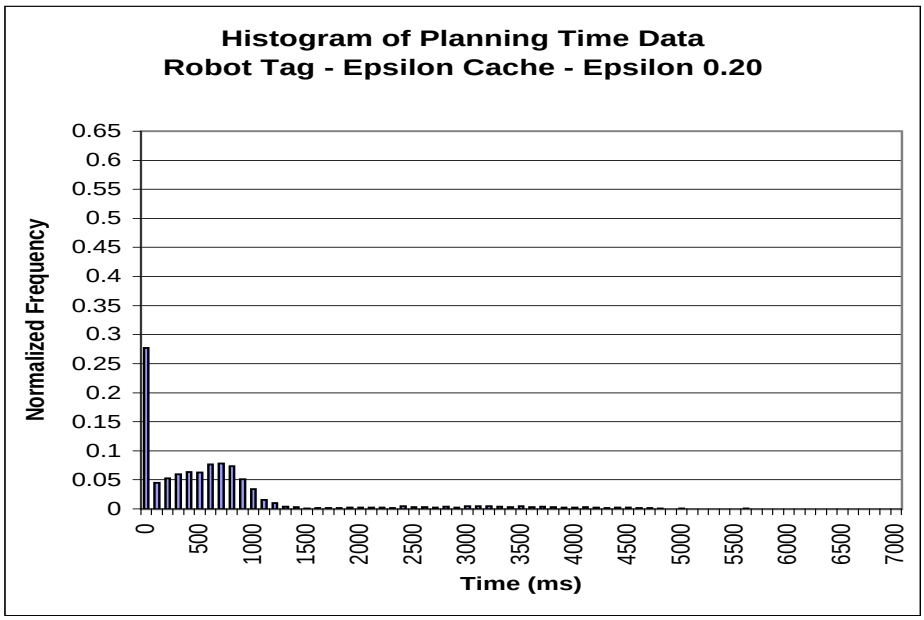

Figure A.54: Planning Time Data - Robot Tag - Epsilon Cache - Epsilon = 0.20

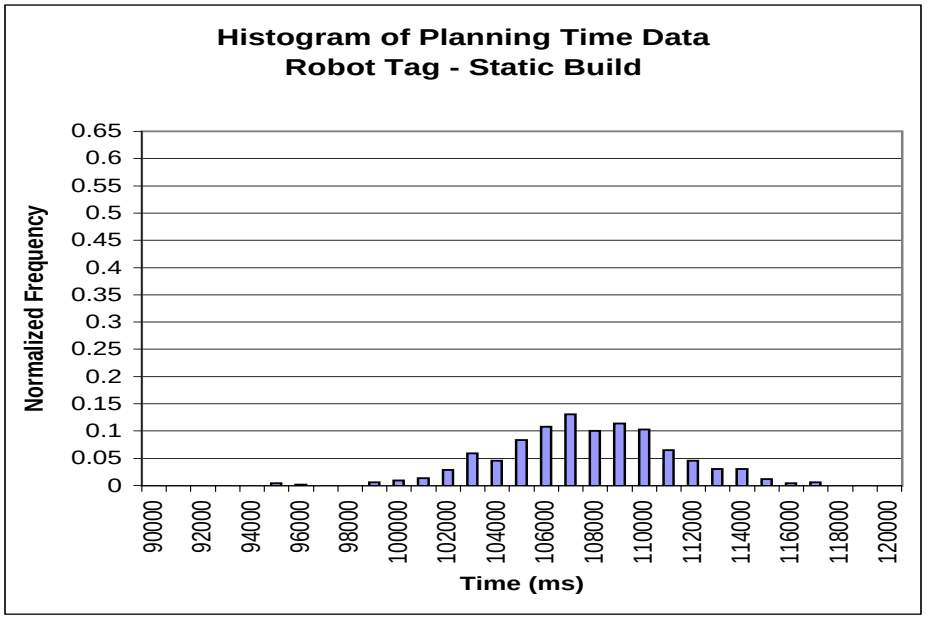

Figure A.55: Planning Time Data - Robot Tag - Static Build

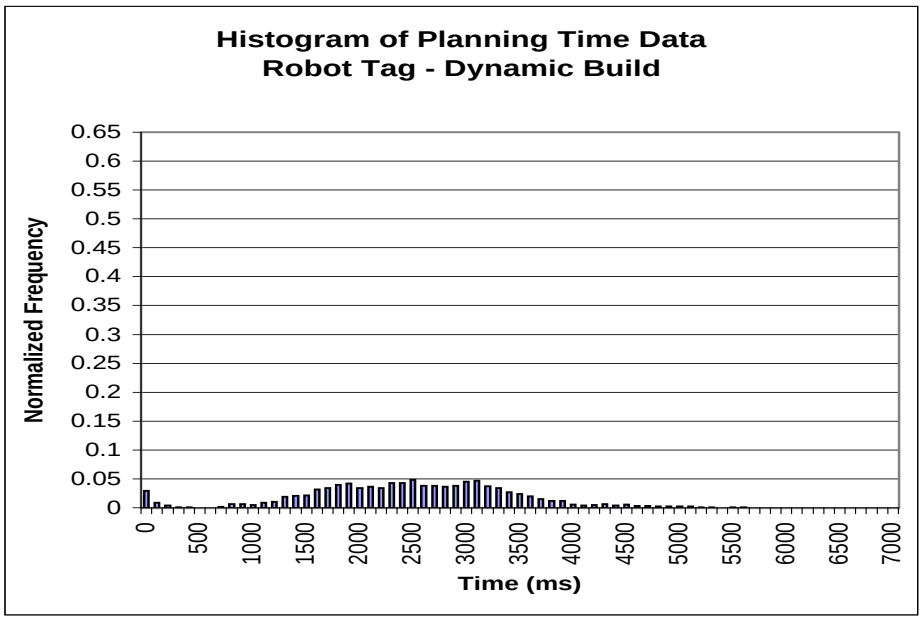

Figure A.56: Planning Time Data - Robot Tag - Dynamic Build

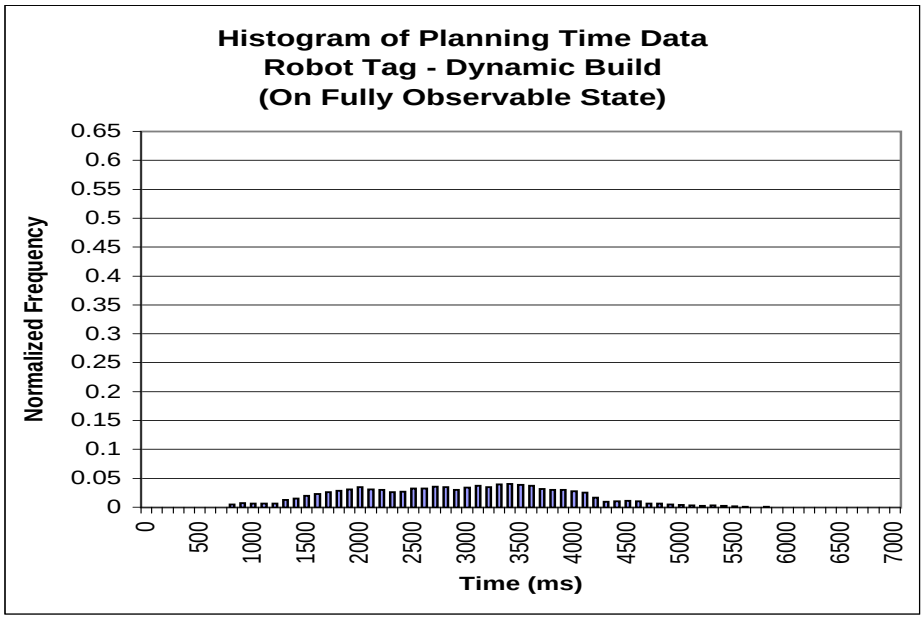

Figure A.57: Planning Time Data - Robot Tag - Dynamic Build (On Fully Observable State)

Appendix B

Histograms of Reward Data

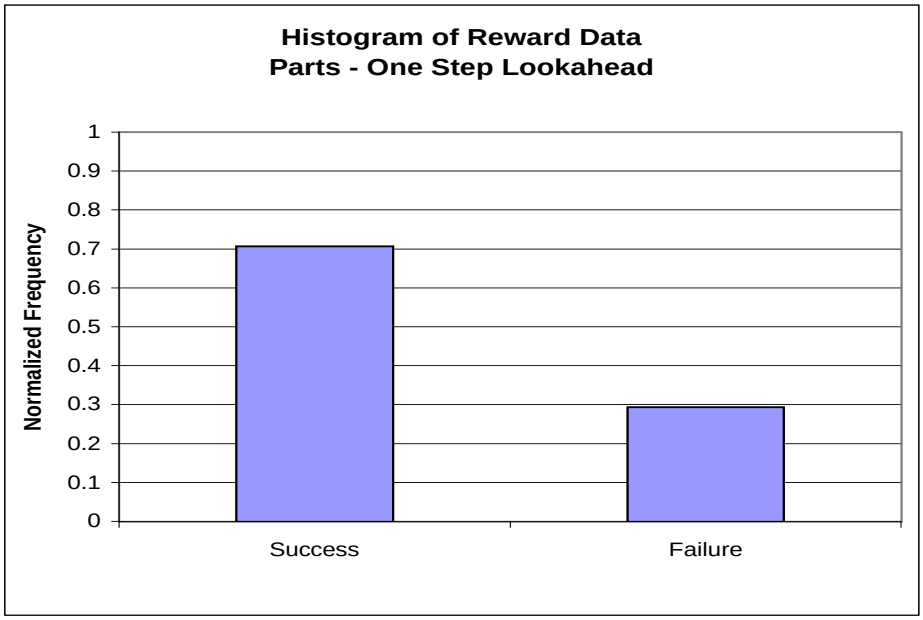

Figure B.1: Reward Data - Parts - One Step Lookahead

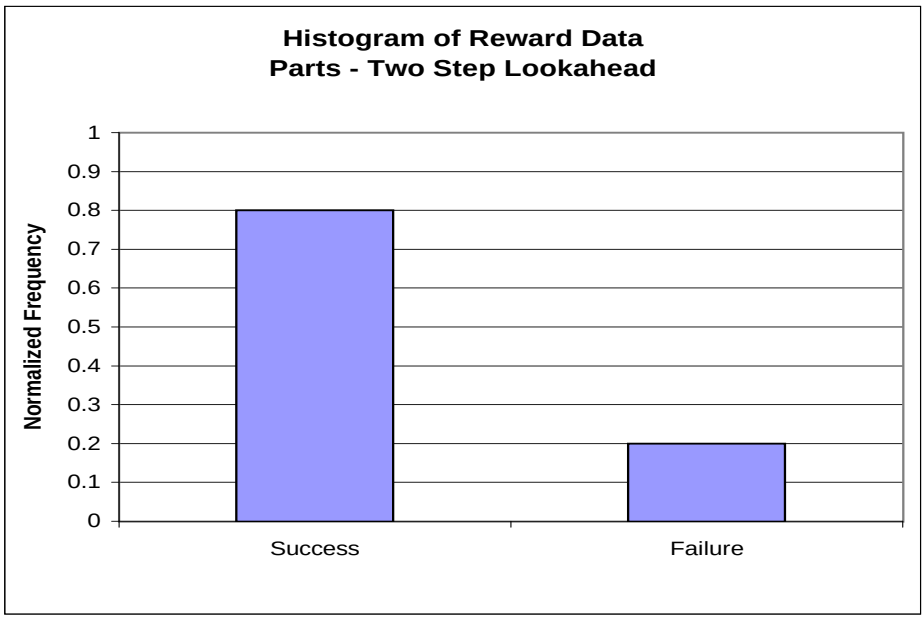

Figure B.2: Reward Data - Parts - Two Step Lookahead

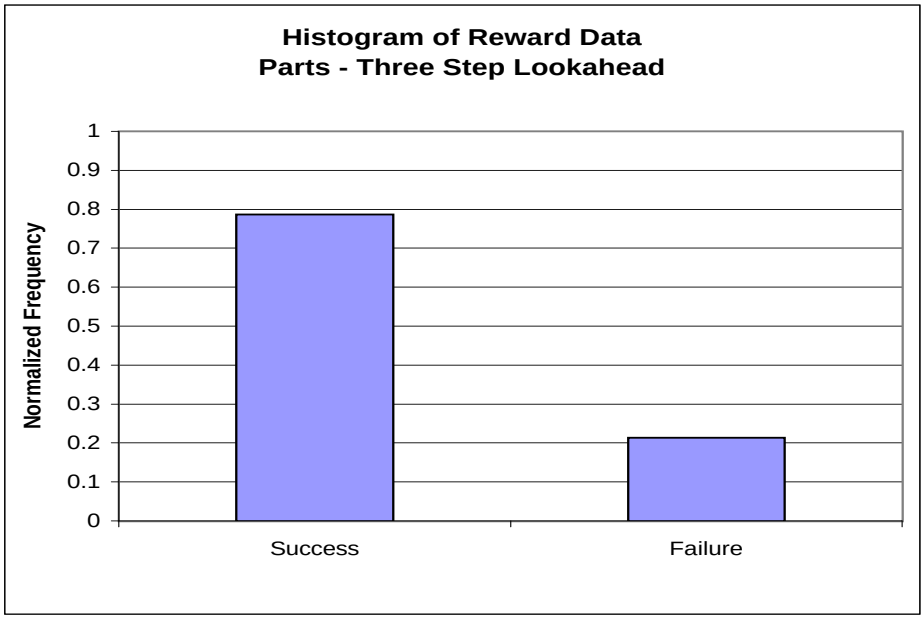

Figure B.3: Reward Data - Parts - Three Step Lookahead

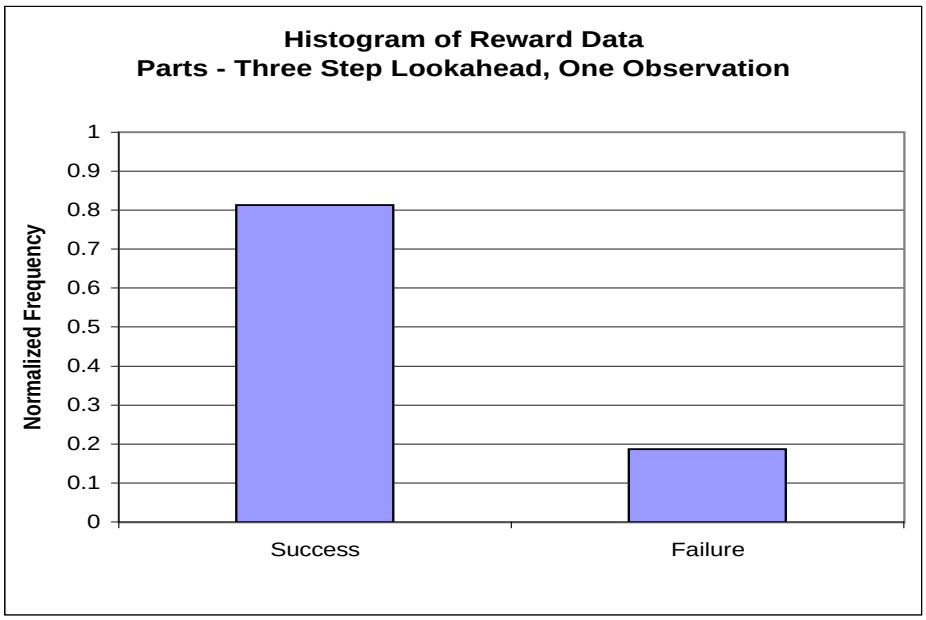

Figure B.4: Reward Data - Parts - Three Step Lookahead, One Observation

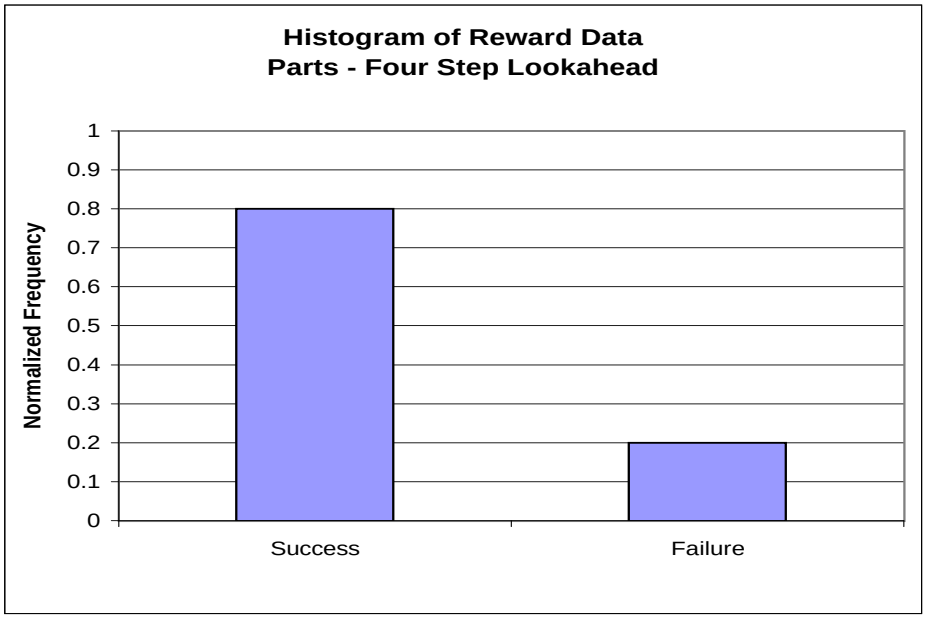

Figure B.5: Reward Data - Parts - Four Step Lookahead

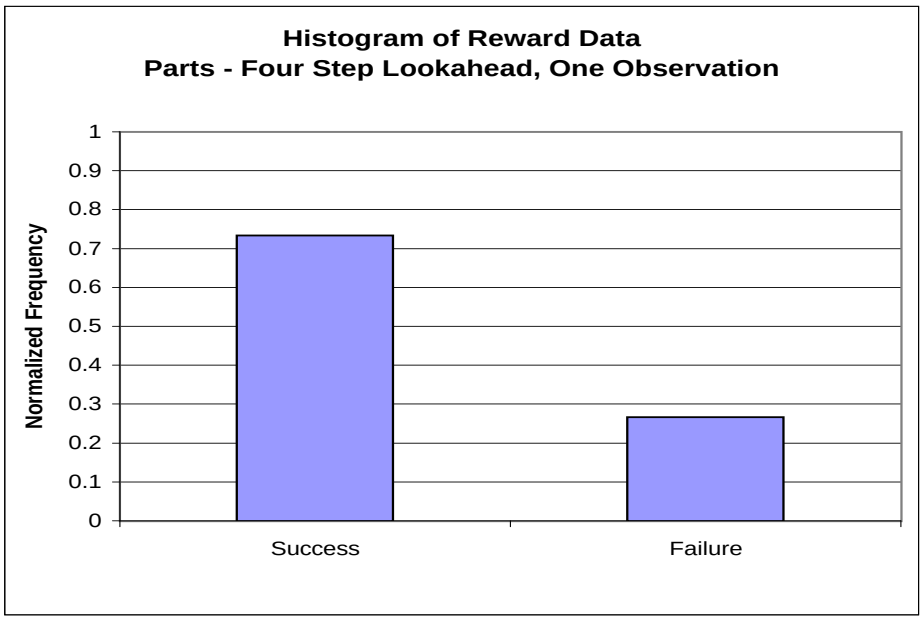

Figure B.6: Reward Data - Parts - Four Step Lookahead, One Observation

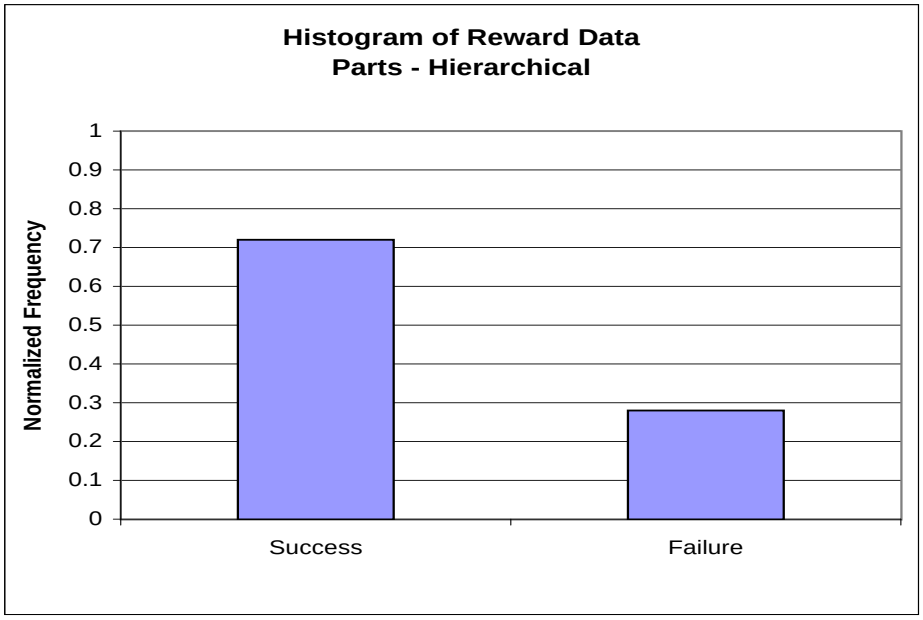

Figure B.7: Reward Data - Parts - Hierarchical

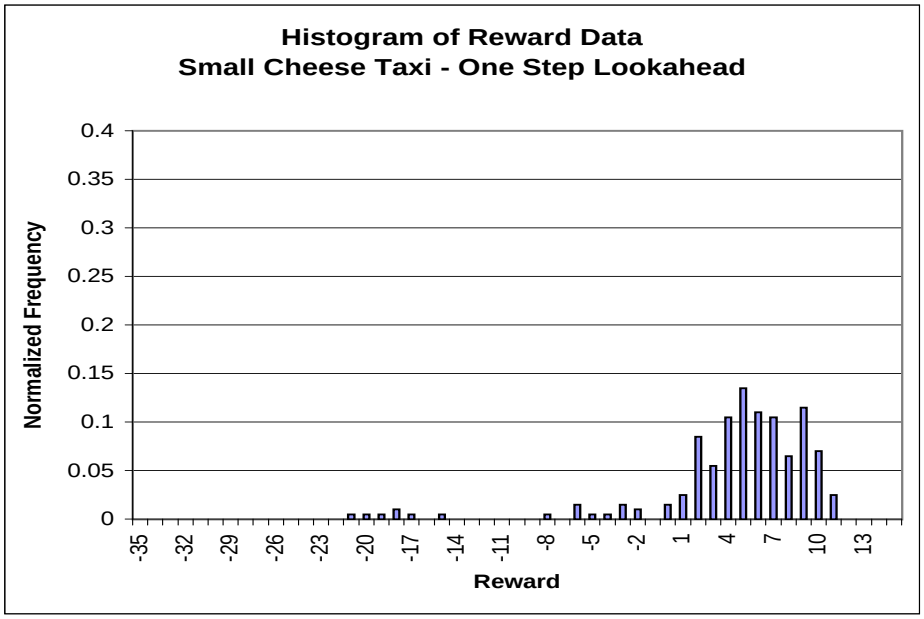

Figure B.8: Reward Data - Small Cheese Taxi - One Step Lookahead

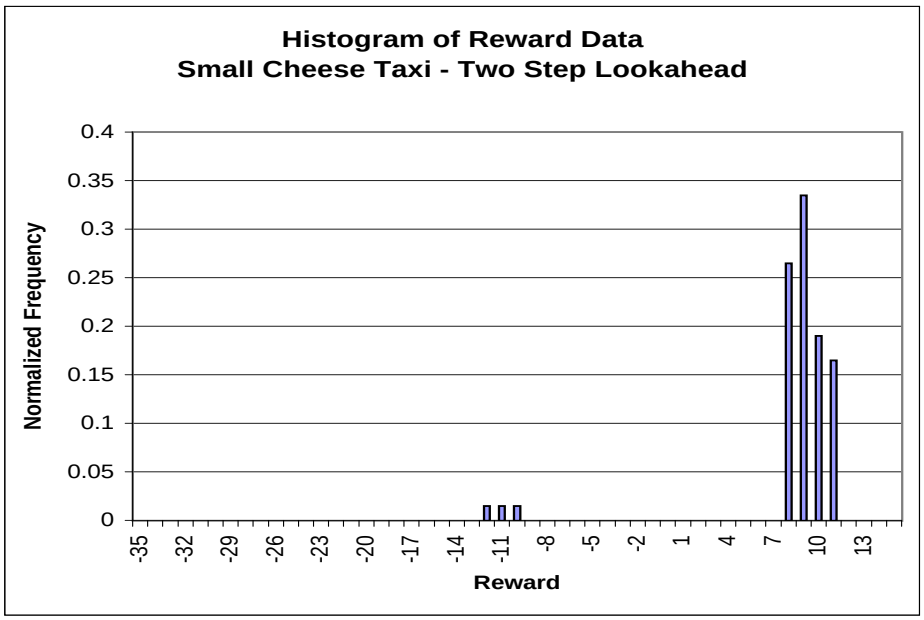

Figure B.9: Reward Data - Small Cheese Taxi - Two Step Lookahead

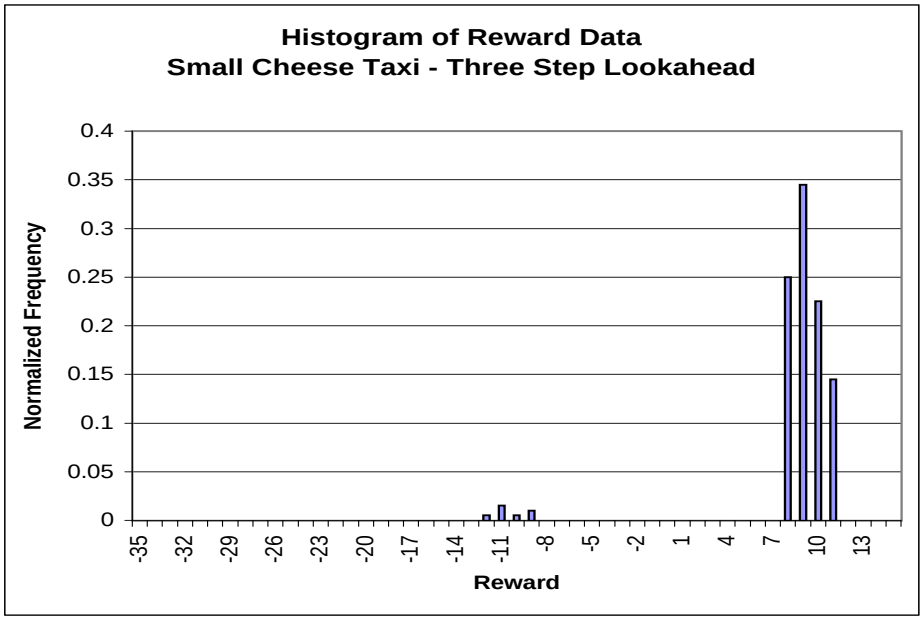

Figure B.10: Reward Data - Small Cheese Taxi - Three Step Lookahead

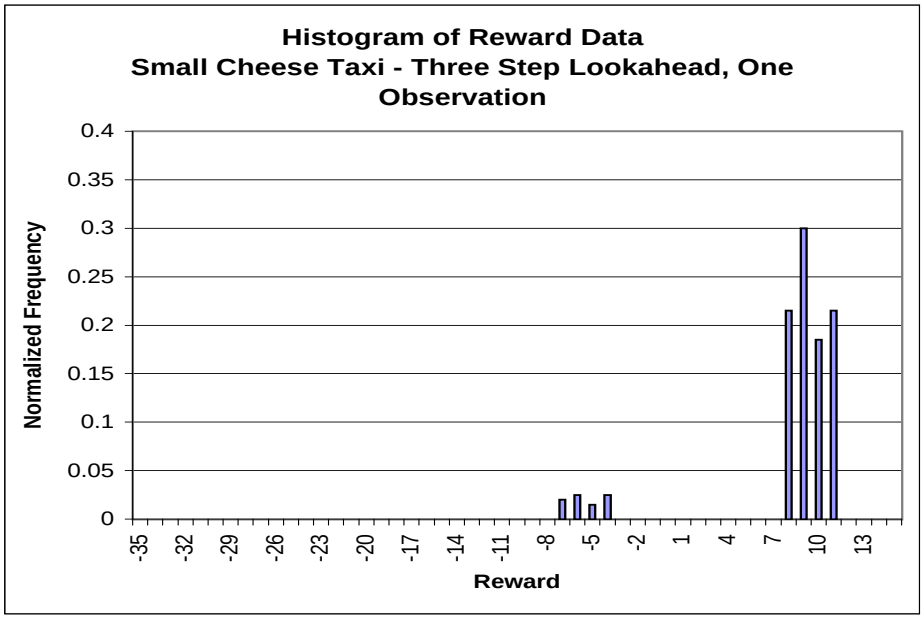

Figure B.11: Reward Data - Small Cheese Taxi - Three Step Lookahead, One Observation

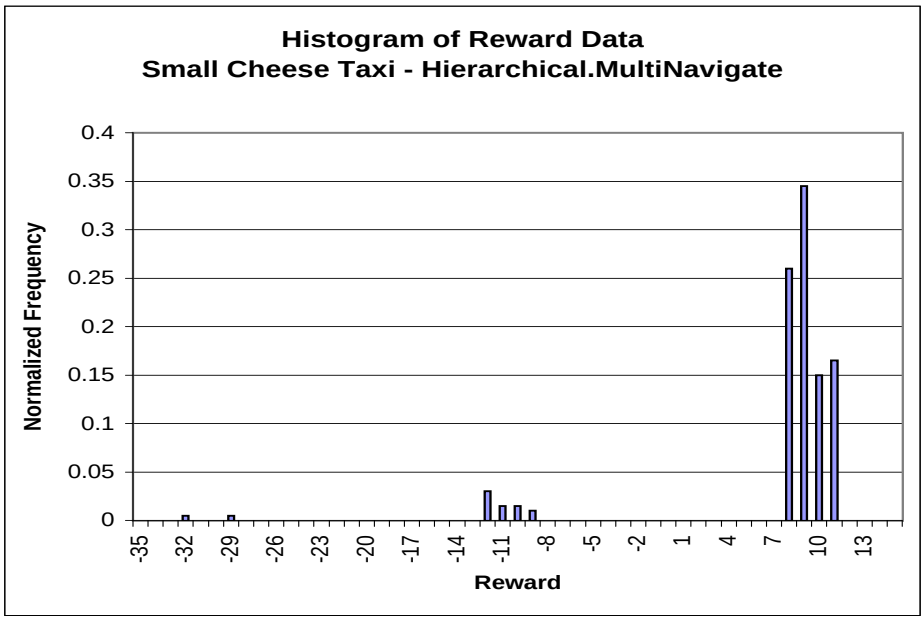

Figure B.12: Reward Data - Small Cheese Taxi - Hierarchical (MultiNavigate)

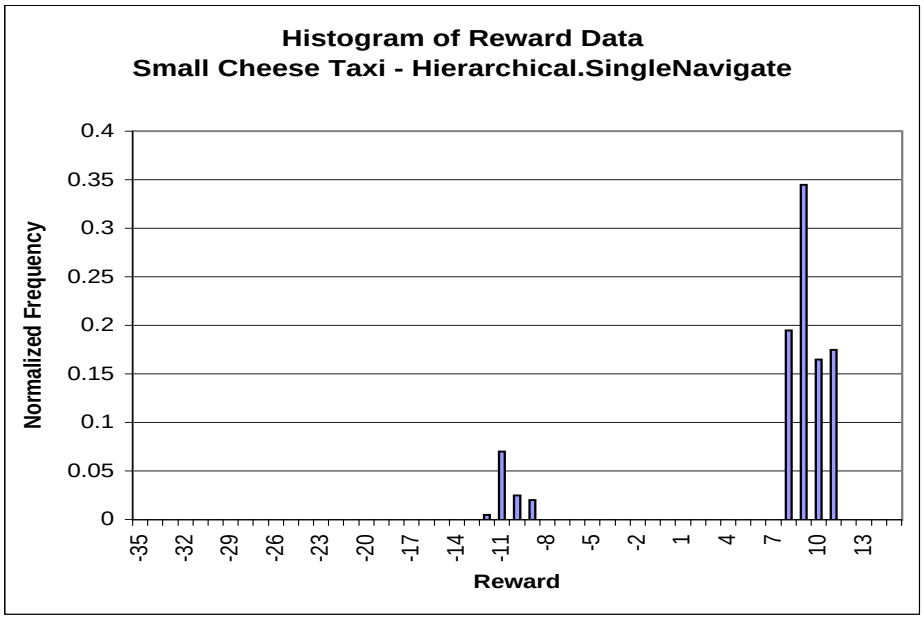

Figure B.13: Reward Data - Small Cheese Taxi - Hierarchical (Single Navigate)

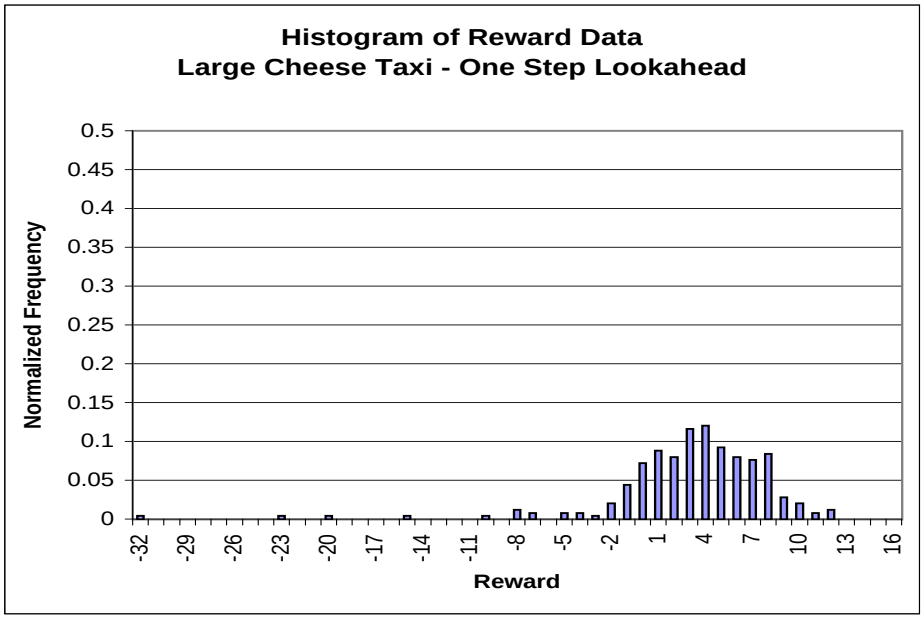

Figure B.14: Reward Data - Large Cheese Taxi - One Step Lookahead

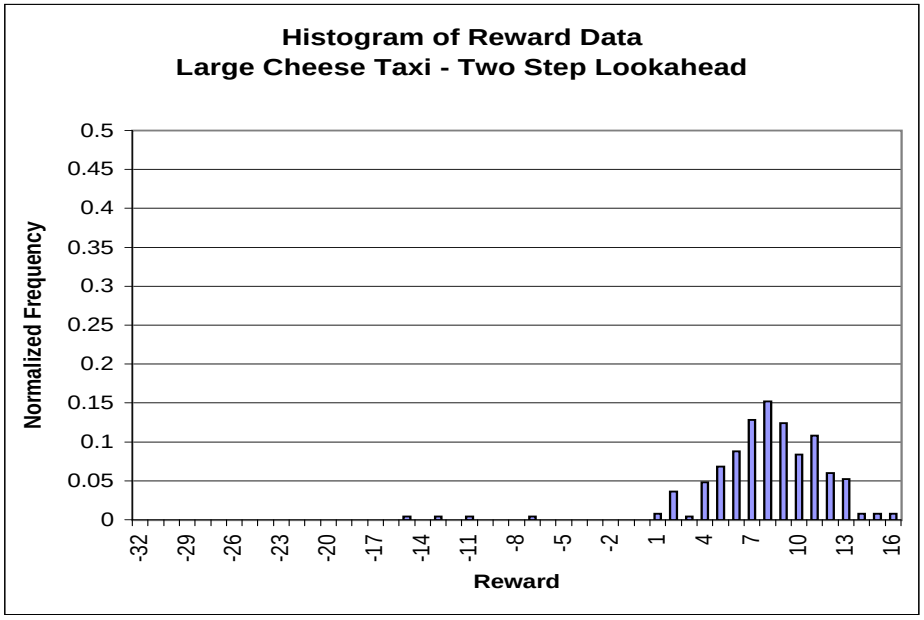

Figure B.15: Reward Data - Large Cheese Taxi - Two Step Lookahead

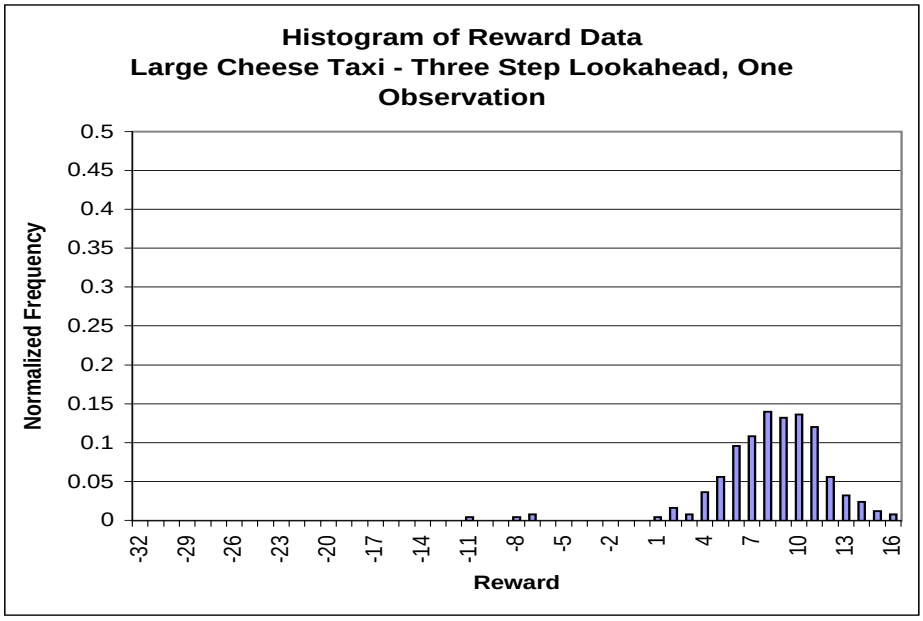

Figure B.16: Reward Data - Large Cheese Taxi - Three Step Lookahead, One Observation

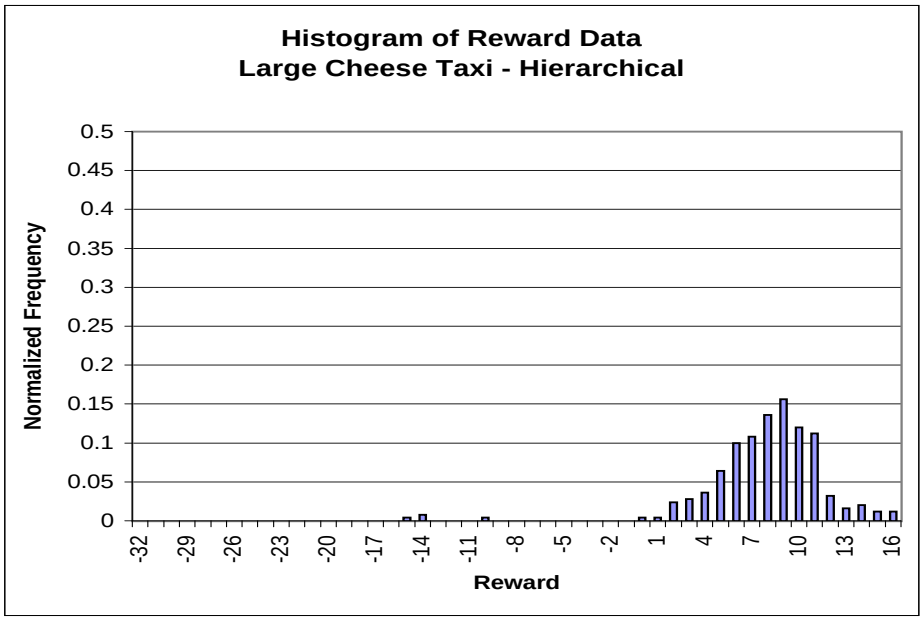

Figure B.17: Reward Data - Large Cheese Taxi - Hierarchical

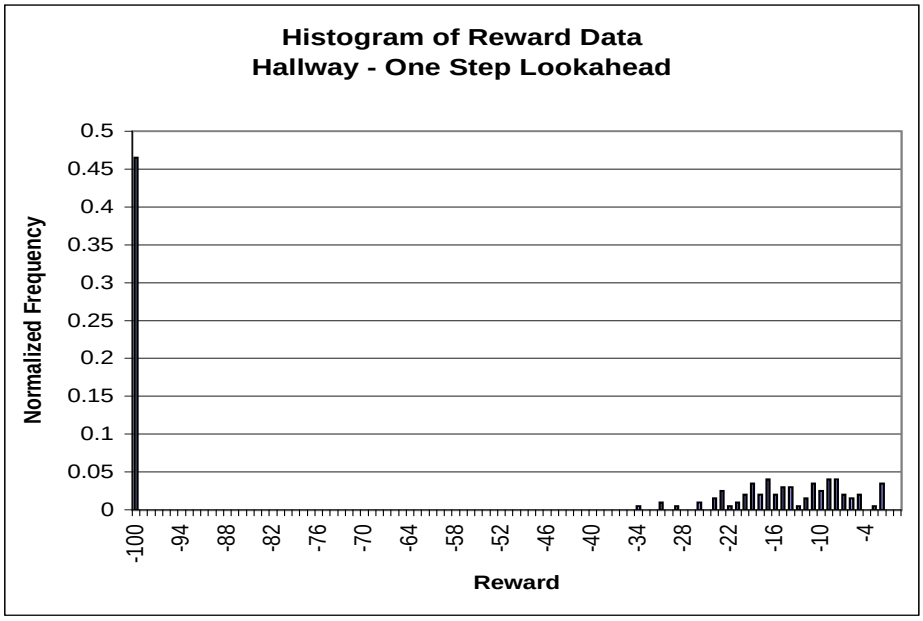

Figure B.18: Reward Data - Hallway - One Step Lookahead

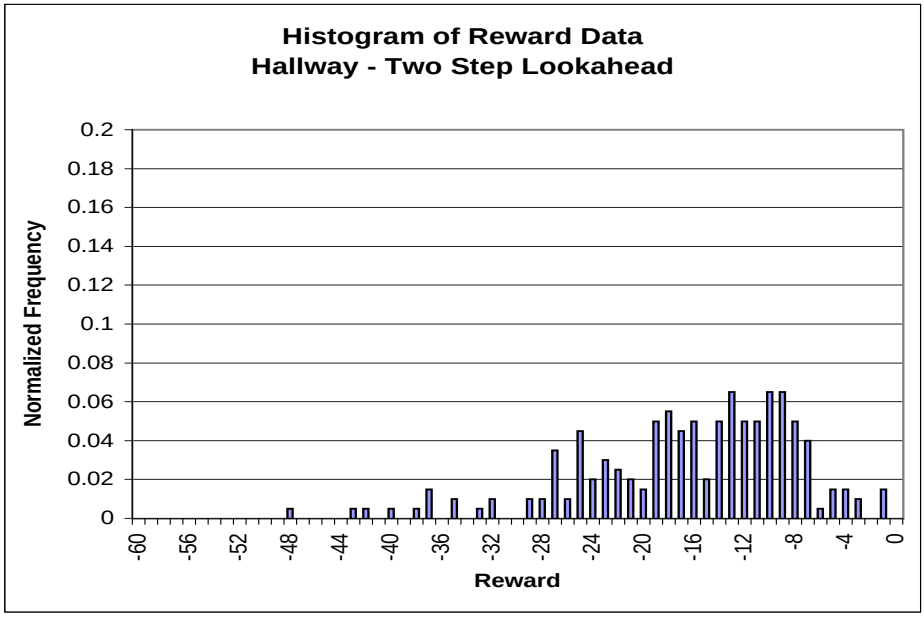

Figure B.19: Reward Data - Hallway - Two Step Lookahead

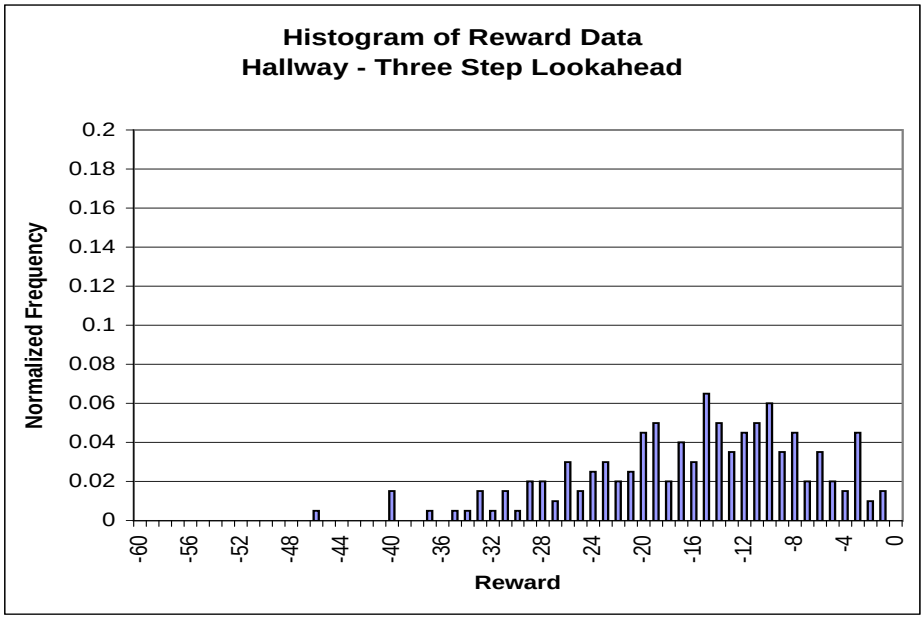

Figure B.20: Reward Data - Hallway - Three Step Lookahead

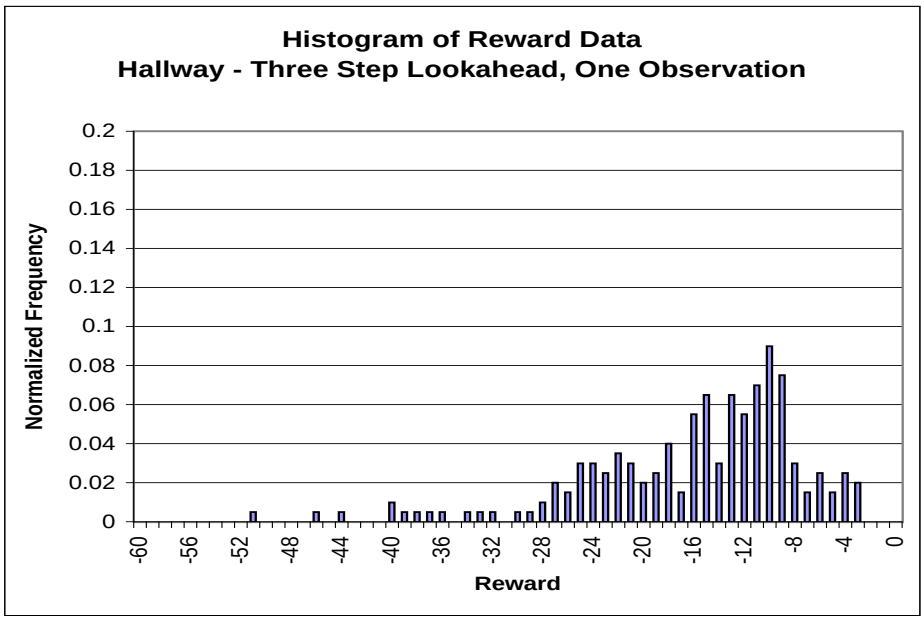

Figure B.21: Reward Data - Hallway - Three Step Lookahead, One Observation

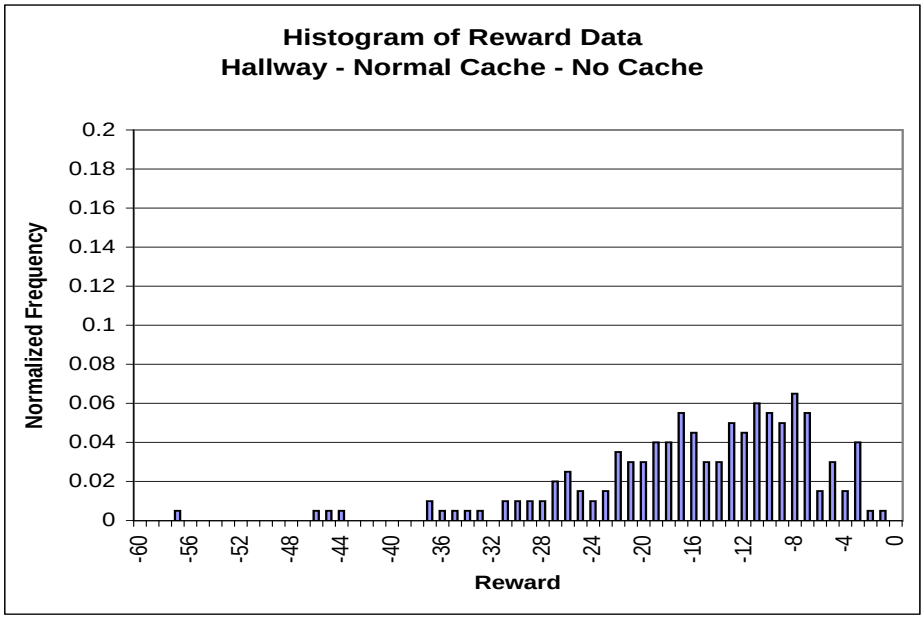

Figure B.22: Reward Data - Hallway - Normal Cache - Cache Size = 0

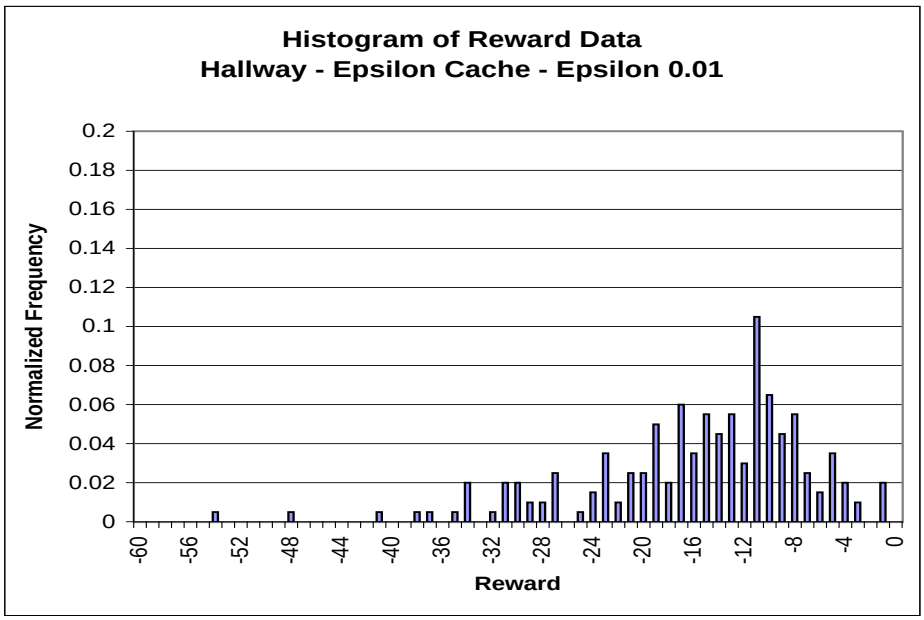

Figure B.23: Reward Data - Hallway - Epsilon Cache - Epsilon = 0.01

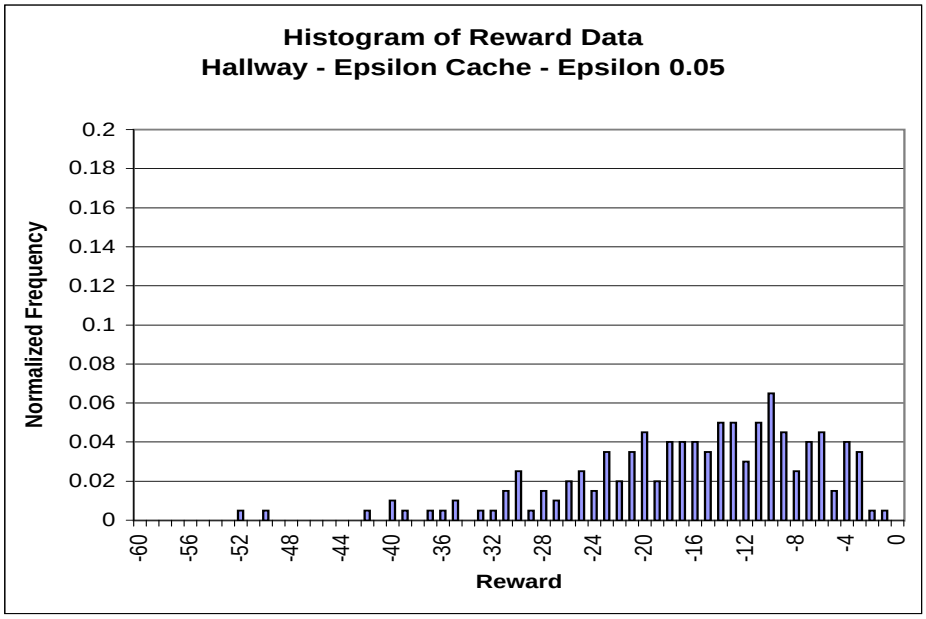

Figure B.24: Reward Data - Hallway - Epsilon Cache - Epsilon = 0.05

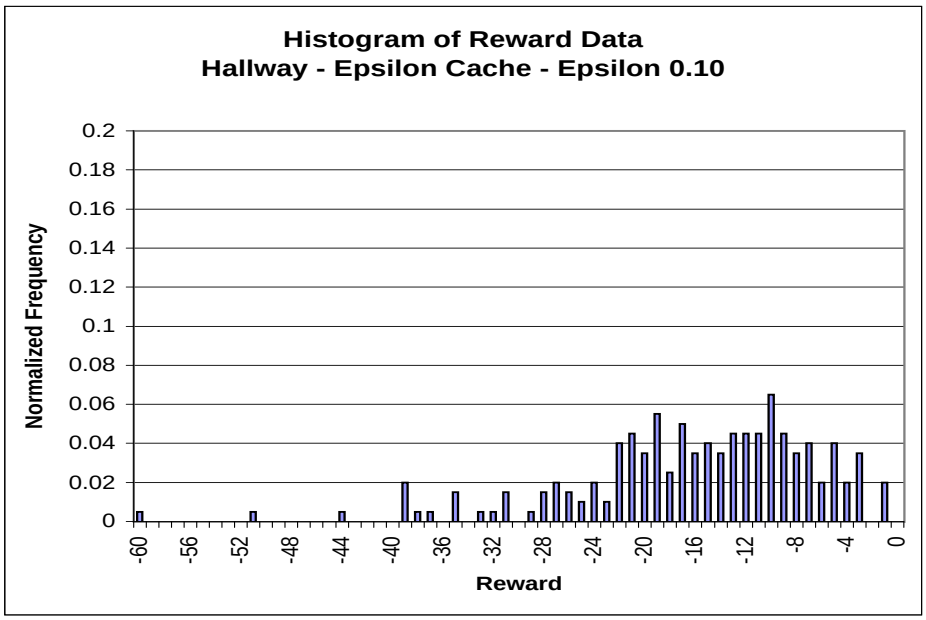

Figure B.25: Reward Data - Hallway - Epsilon Cache - Epsilon = 0.10

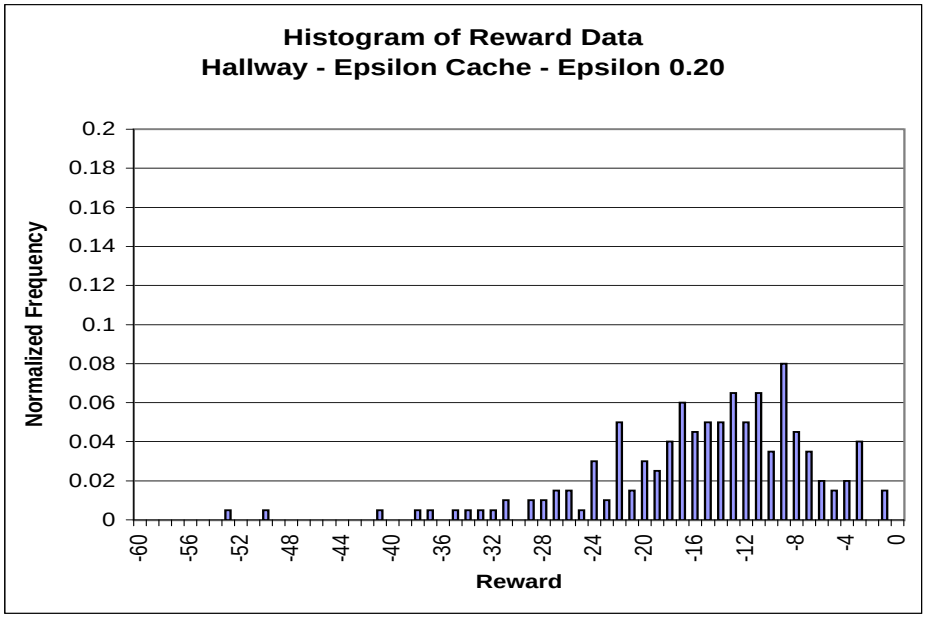

Figure B.26: Reward Data - Hallway - Epsilon Cache - Epsilon = 0.20

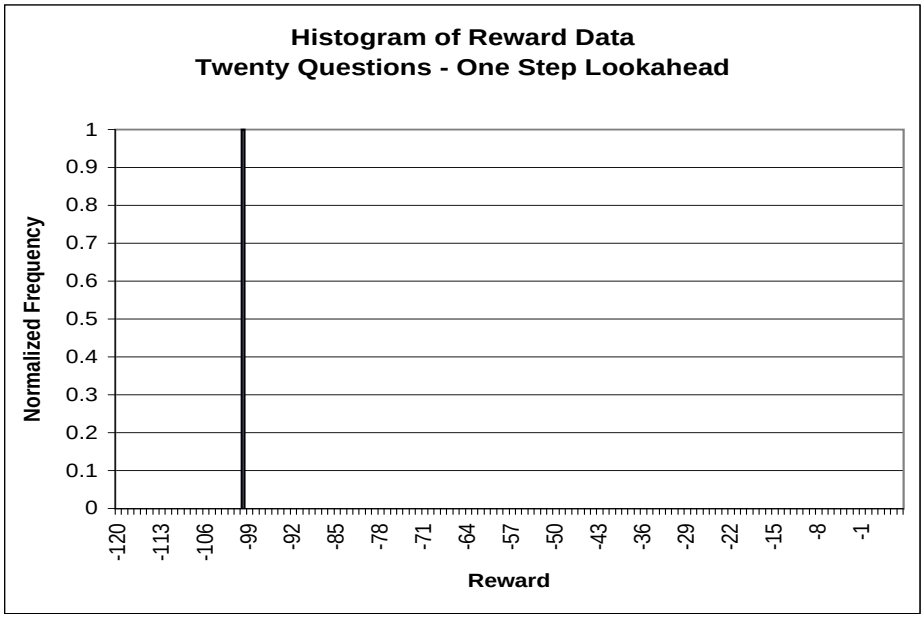

Figure B.27: Reward Data - Twenty Questions - One Step Lookahead

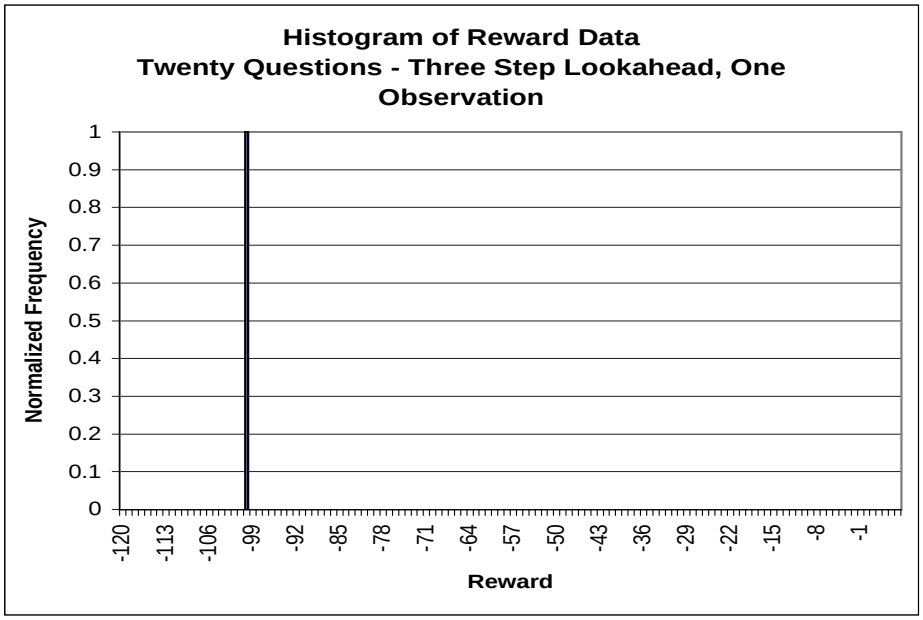

Figure B.28: Reward Data - Twenty Questions - Three Step Lookahead, One Observation

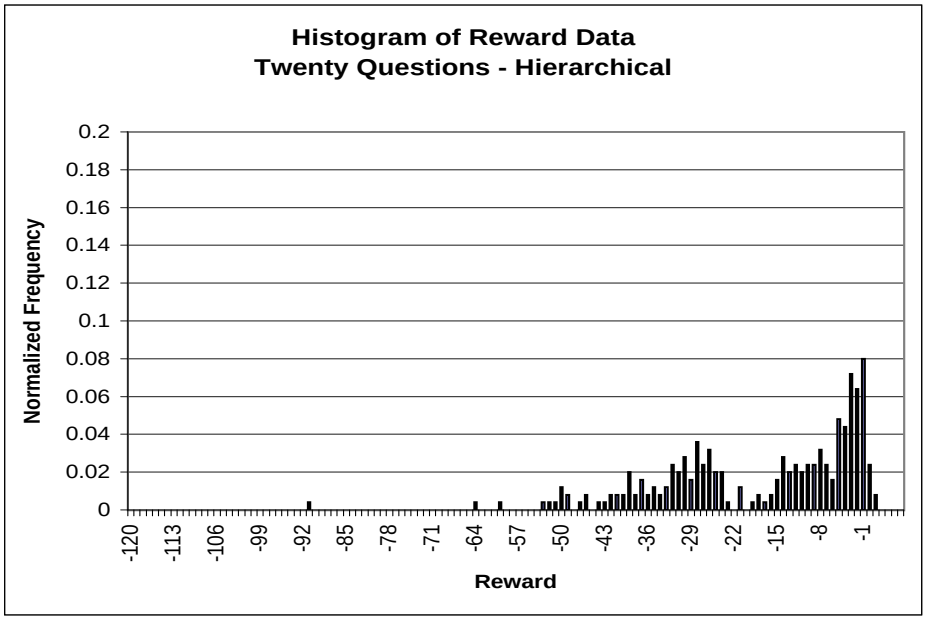

Figure B.29: Reward Data - Twenty Questions - Hierarchical

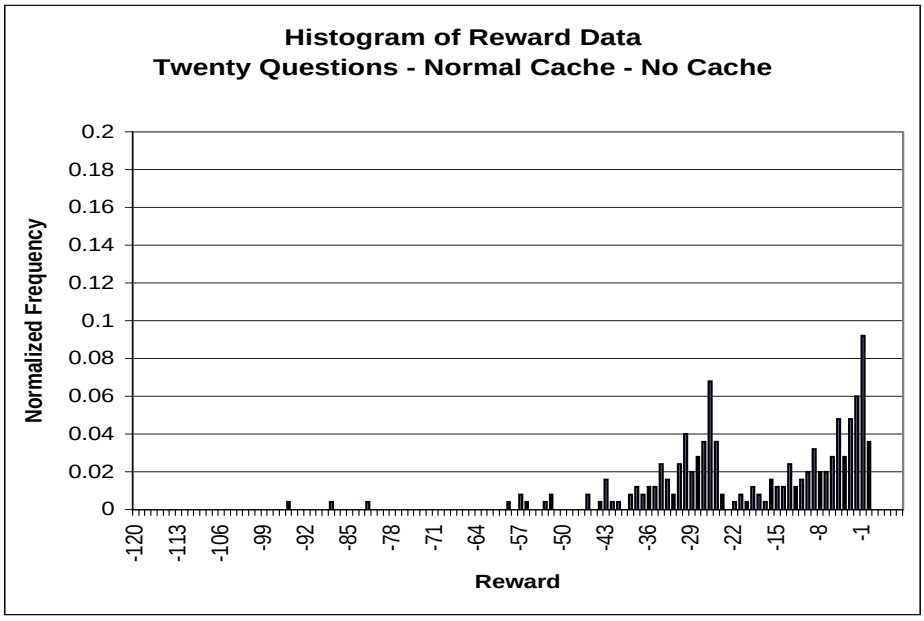

Figure B.30: Reward Data - Twenty Questions - Normal Cache - Cache Size = 0

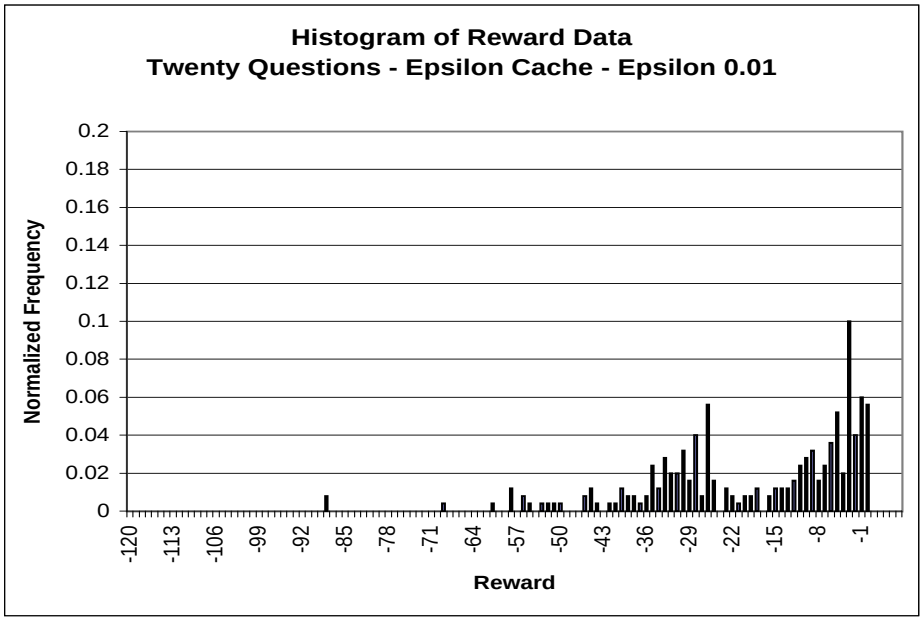

Figure B.31: Reward Data - Twenty Questions - Epsilon Cache - Epsilon = 0.01

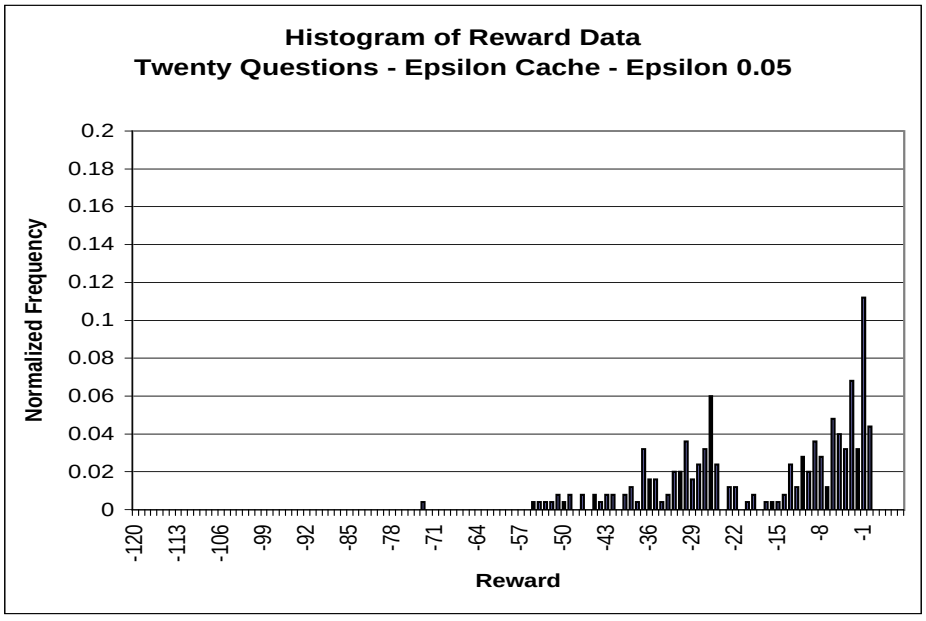

Figure B.32: Reward Data - Twenty Questions - Epsilon Cache - Epsilon = 0.05

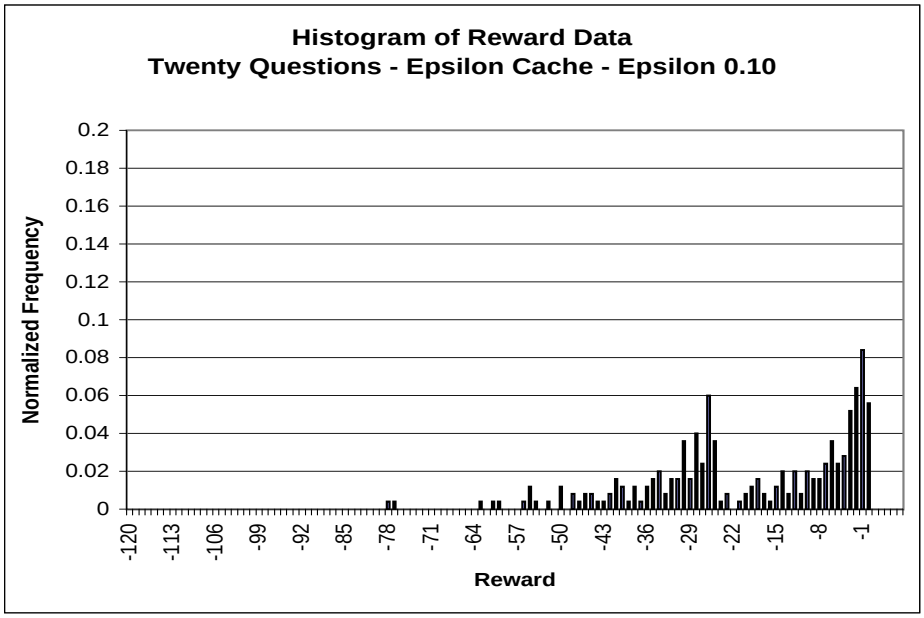

Figure B.33: Reward Data - Twenty Questions - Epsilon Cache - Epsilon = 0.10

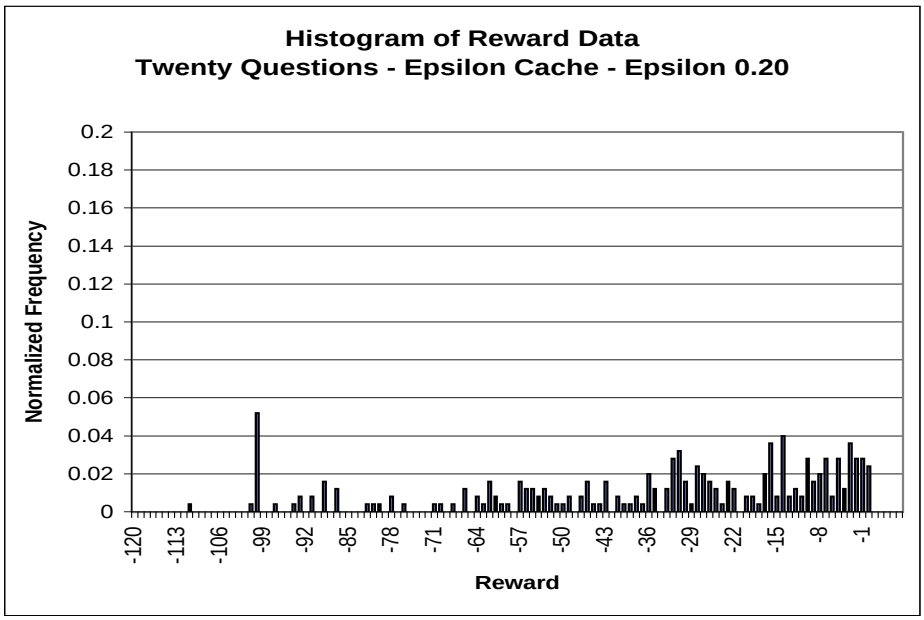

Figure B.34: Reward Data - Twenty Questions - Epsilon Cache - Epsilon = 0.20

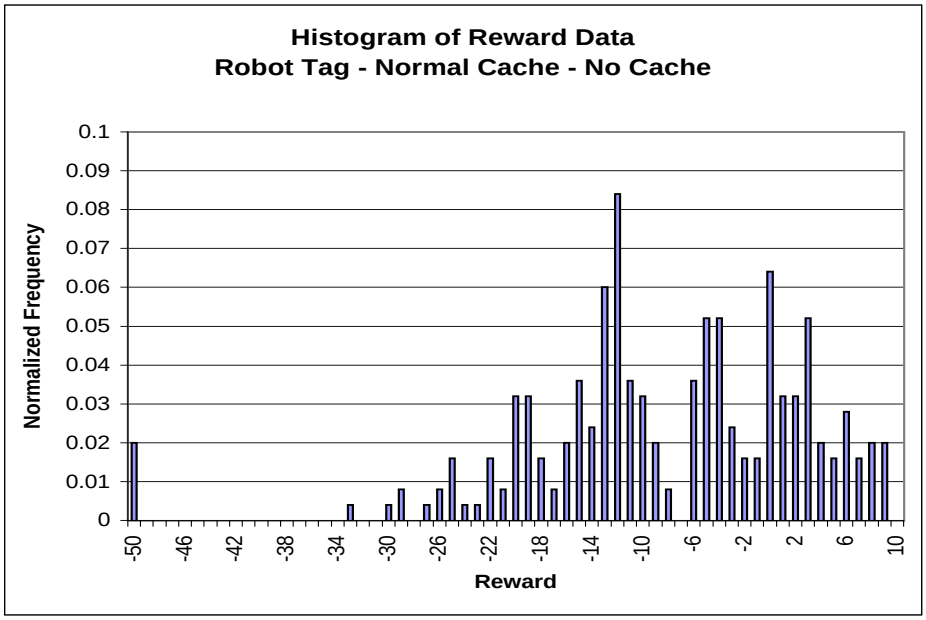

Figure B.35: Reward Data - Robot Tag - Normal Cache - Cache Size = 0

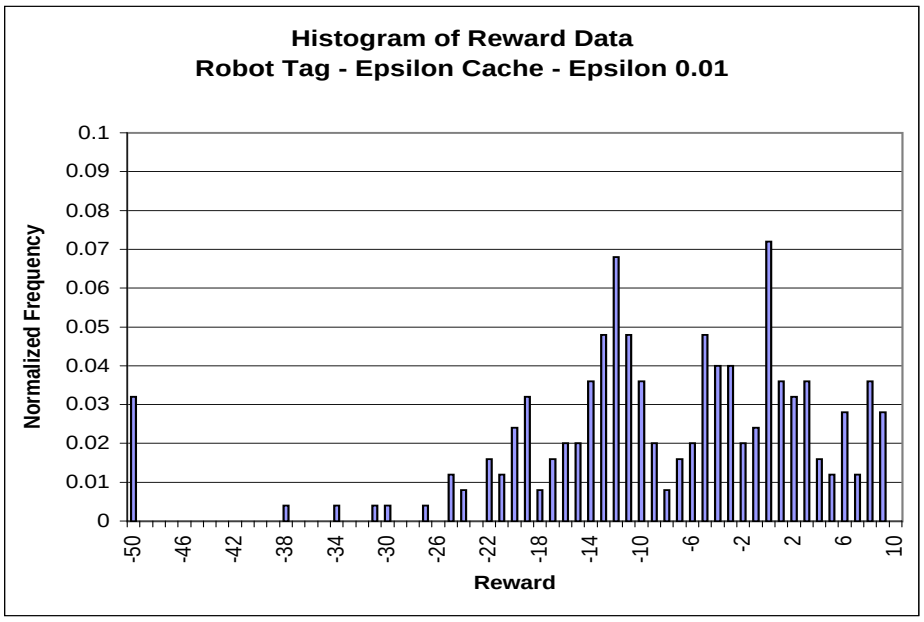

Figure B.36: Reward Data - Robot Tag - Epsilon Cache - Epsilon = 0.01

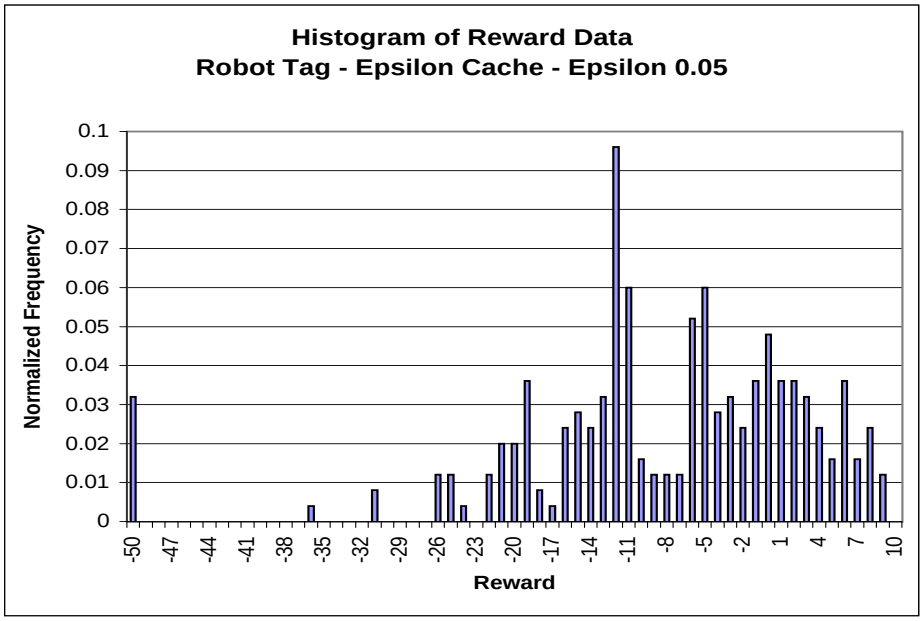

Figure B.37: Reward Data - Robot Tag - Epsilon Cache - Epsilon = 0.05

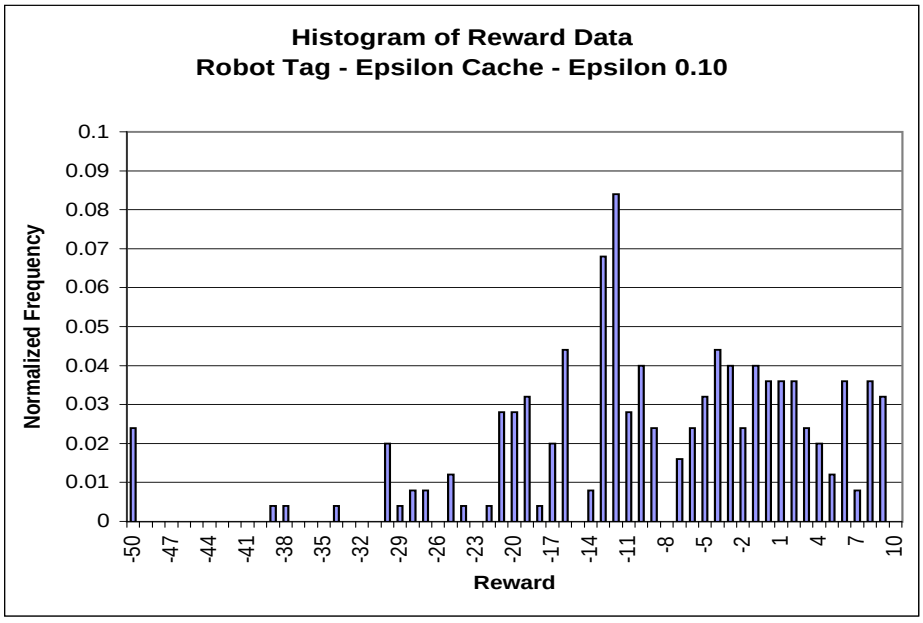

Figure B.38: Reward Data - Robot Tag - Epsilon Cache - Epsilon = 0.10

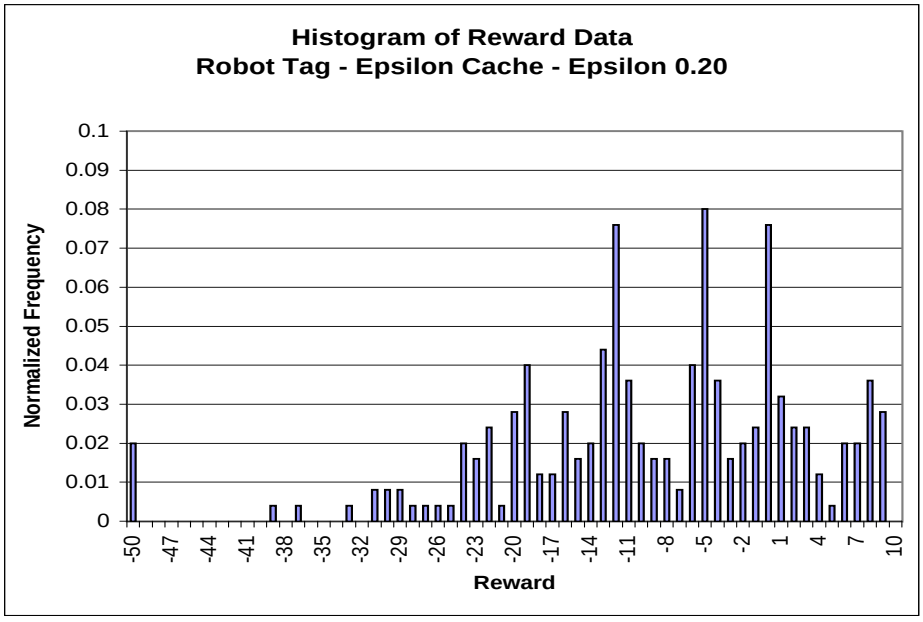

Figure B.39: Reward Data - Robot Tag - Epsilon Cache - Epsilon = 0.20

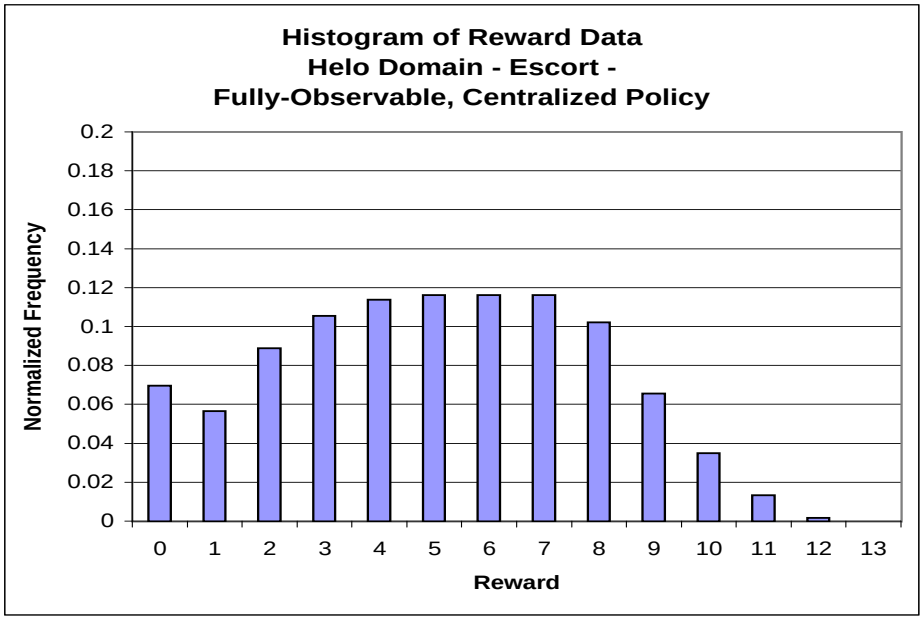

Figure B.40: Reward Data - Helo Domain - Escort - Fully Observable, Centralized Policy

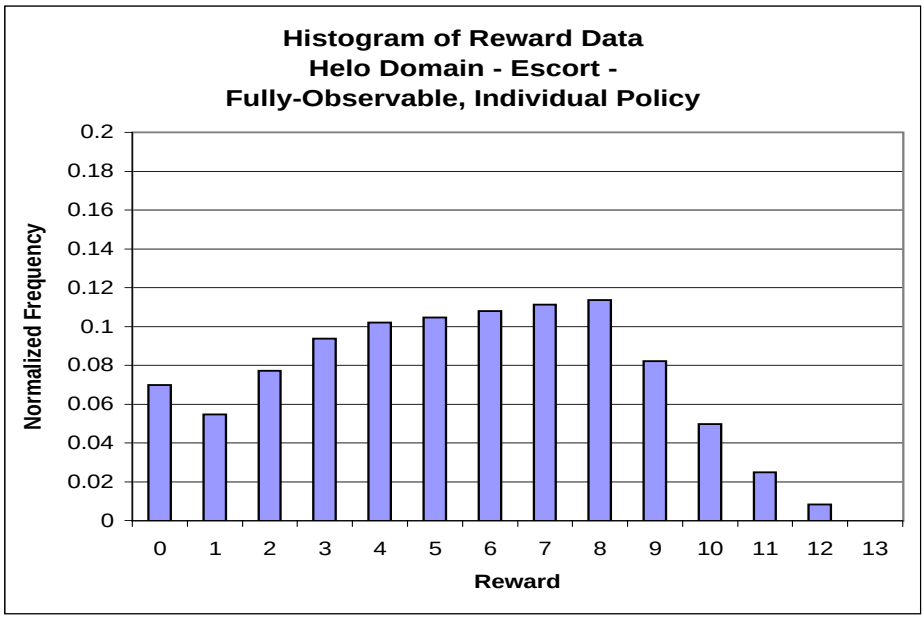

Figure B.41: Reward Data - Helo Domain - Escort - Fully Observable, Individual Policy

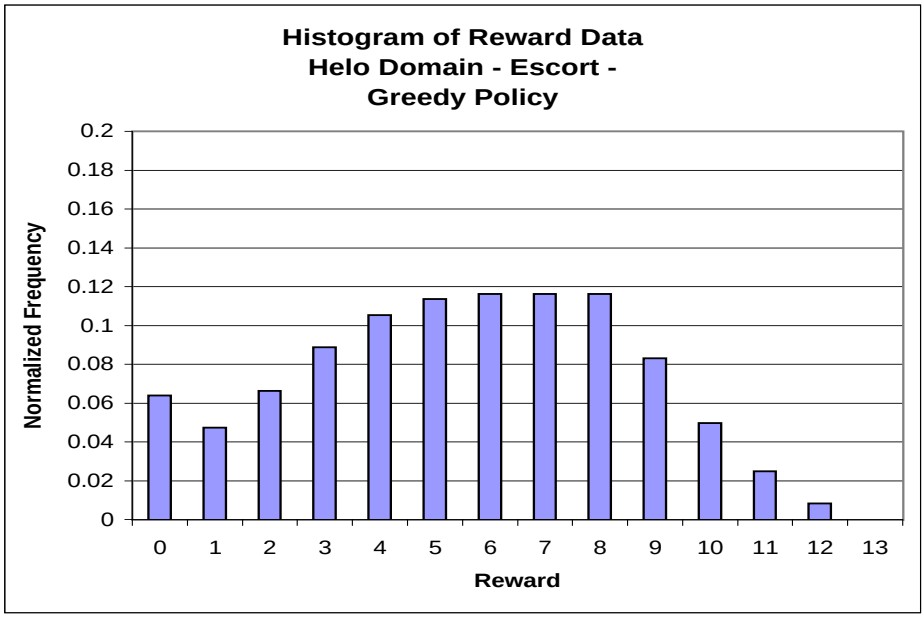

Figure B.42: Reward Data - Helo Domain - Escort - Greedy Policy

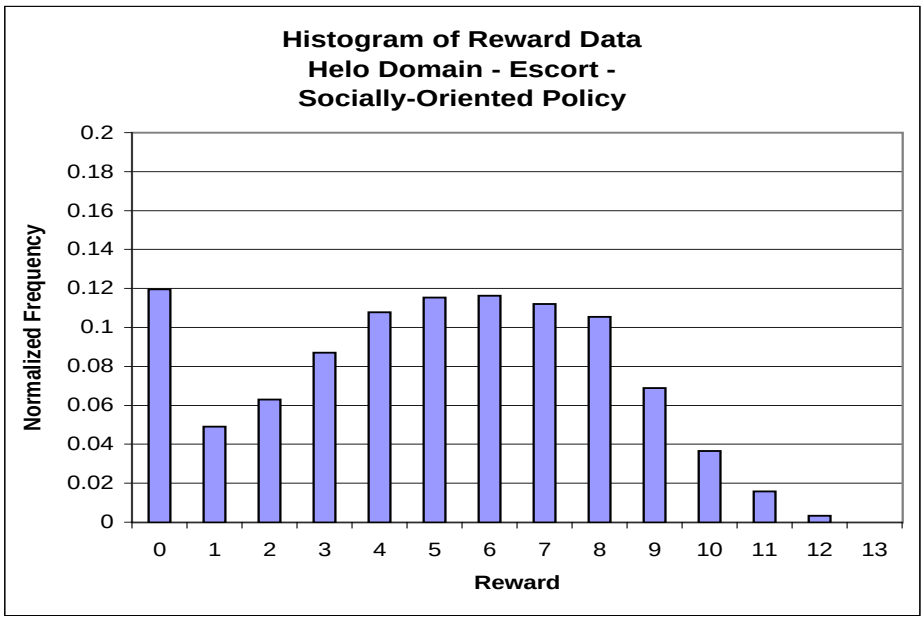

Figure B.43: Reward Data - Helo Domain - Escort - Socially-Oriented Policy

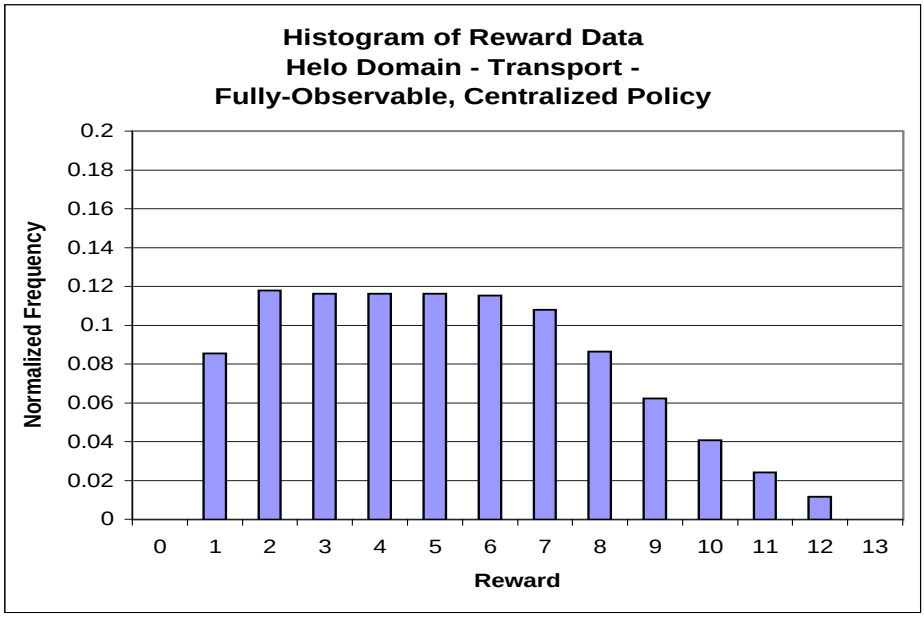

Figure B.44: Reward Data - Helo Domain - Transport - Fully-Observable, Centralized Policy

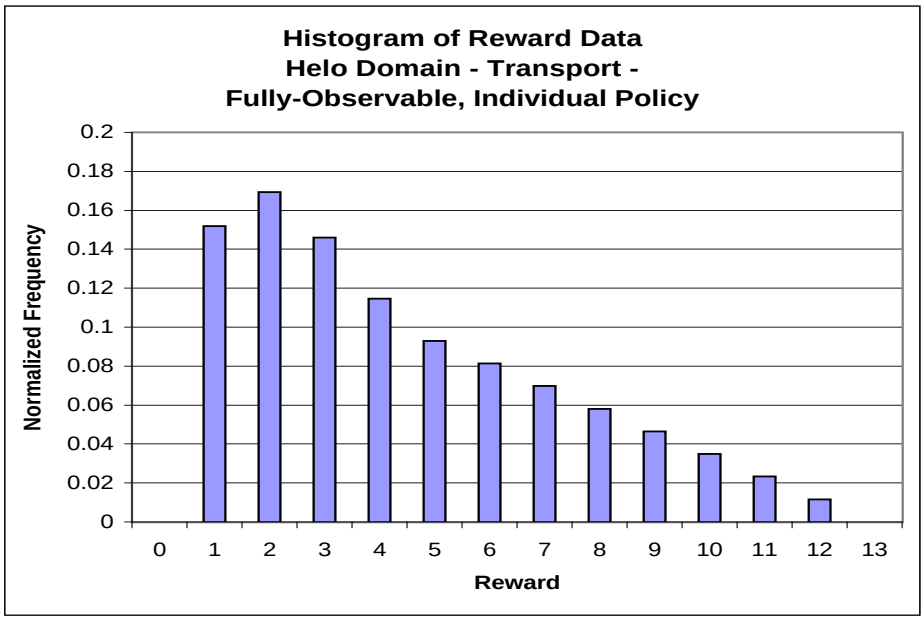

Figure B.45: Reward Data - Helo Domain - Transport - Fully-Observable, Individual Policy

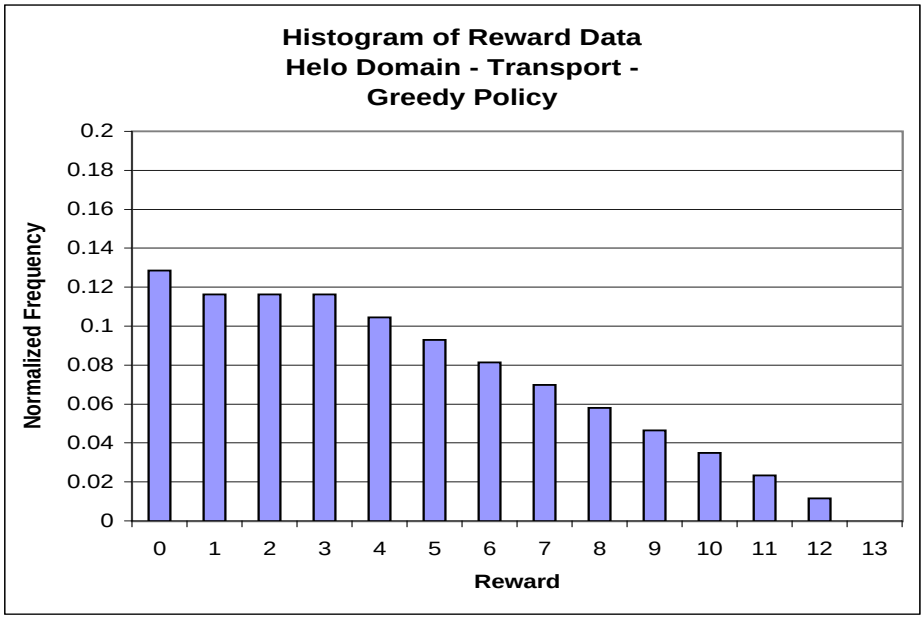

Figure B.46: Reward Data - Helo Domain - Transport - Greedy Policy

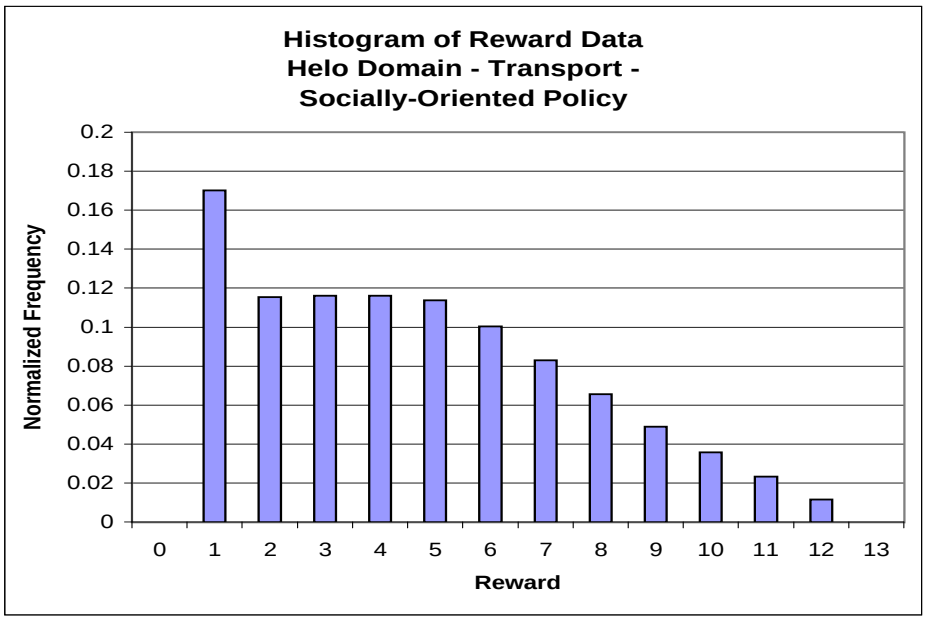

Figure B.47: Reward Data - Helo Domain - Transport - Socially-Oriented Policy

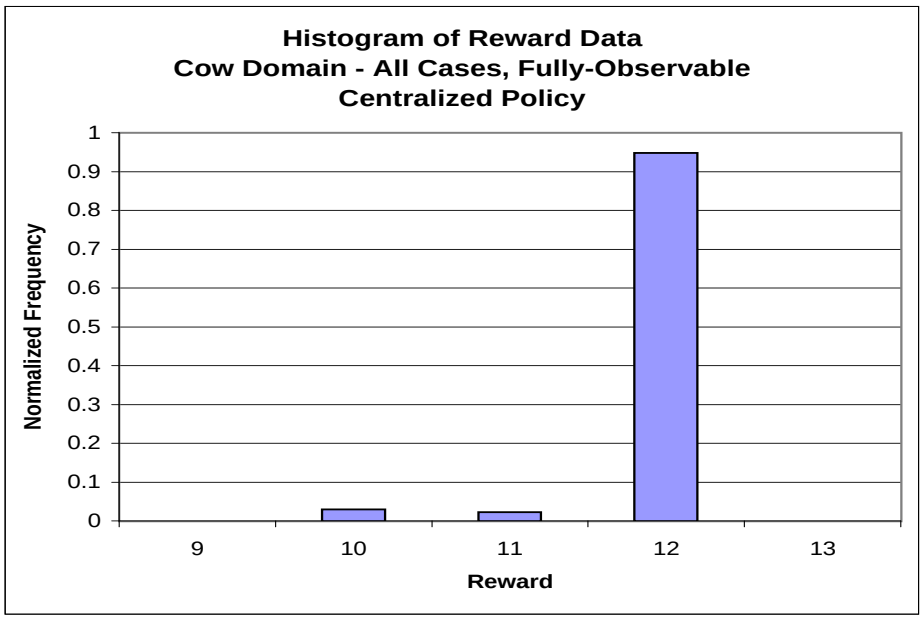

Figure B.48: Reward Data - Cow Domain - All Cases - Fully-Observable, Centralized Policy

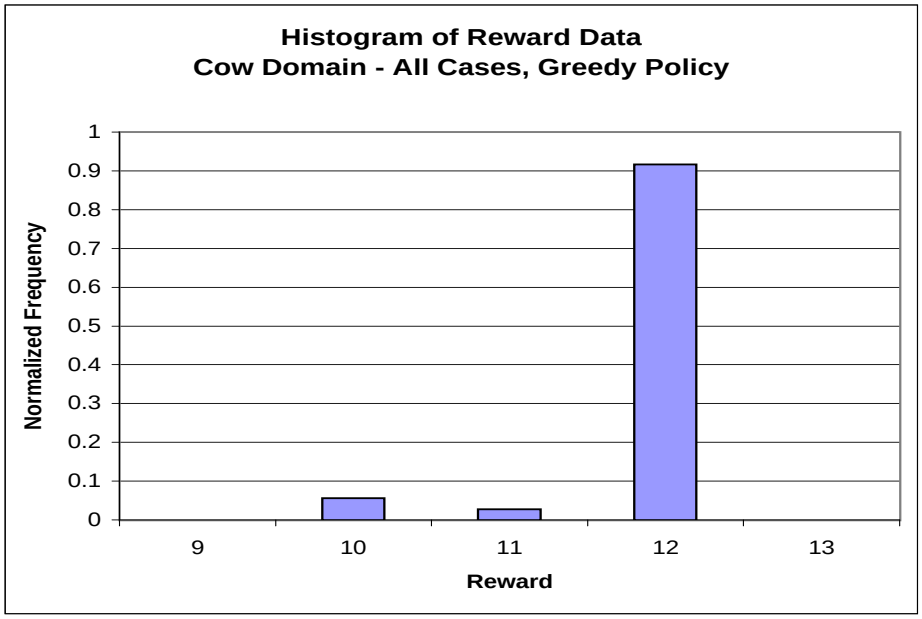

Figure B.49: Reward Data - Cow Domain - All Cases - Greedy Policy

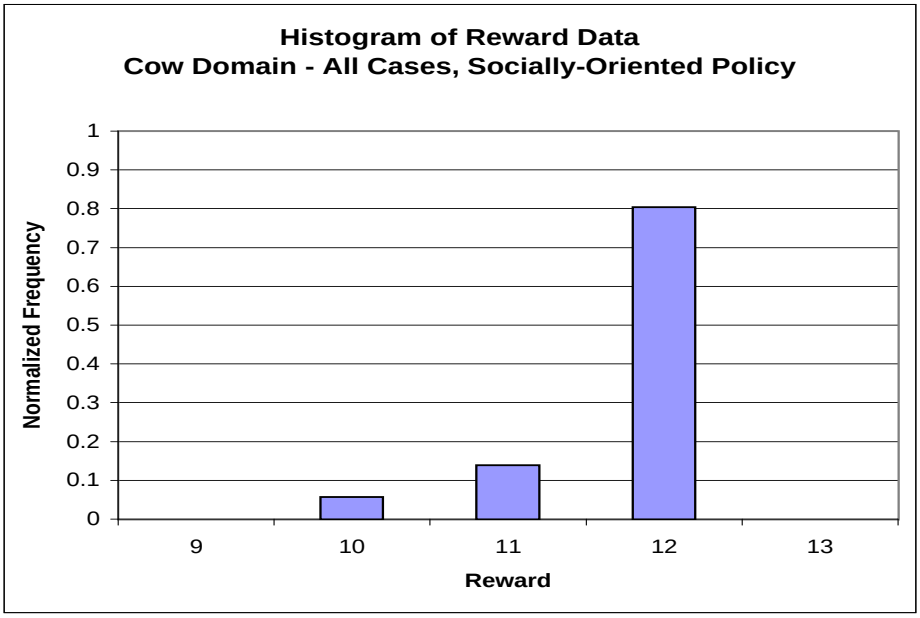

Figure B.50: Reward Data - Cow Domain - All Cases - Socially-Oriented Policy

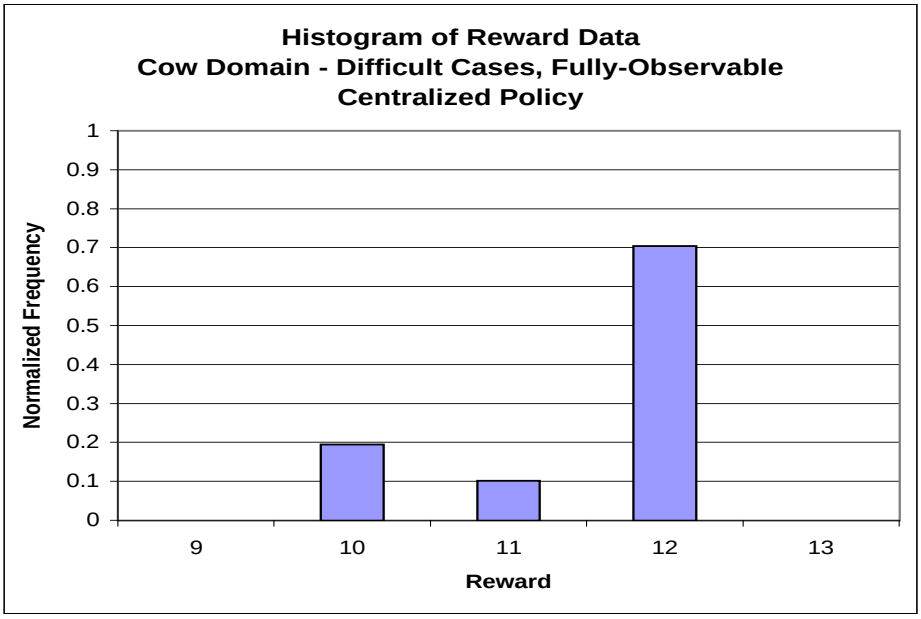

Figure B.51: Reward Data - Cow Domain - Difficult Cases - Fully-Observable, Centralized Policy

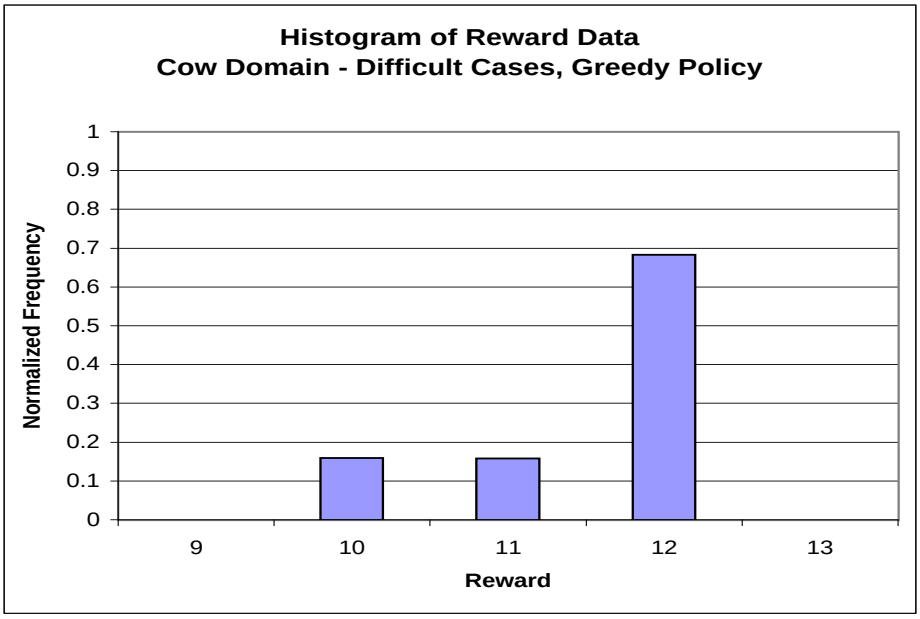

Figure B.52: Reward Data - Cow Domain - Difficult Cases - Greedy Policy

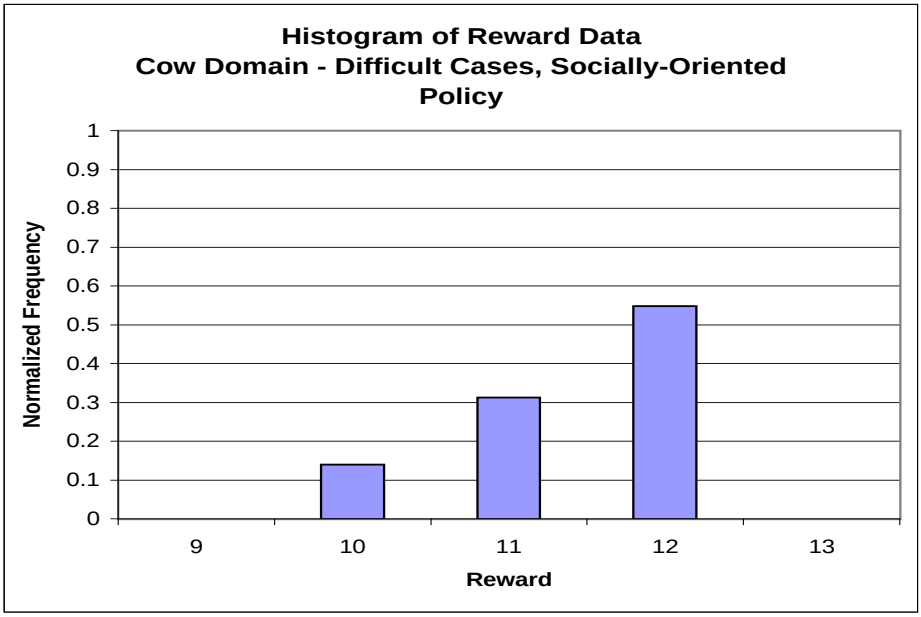

Figure B.53: Reward Data - Cow Domain - Difficult Cases - Socially-Oriented Policy