Planning With Agents: An Efficient Approach Using Hierarchical Dynamic Decision Networks

William H. Turkett, Jr. and John R. Rose

Department of Computer Science and Engineering,
University of South Carolina,
Columbia, SC 29208
{turkett, rose}@cse.sc.edu

Abstract. To be useful in solving real world problems, agents need to be able to act in environments in which it may not be possible to be completely aware of the current state and where actions do not always work as planned. Additional complexity is added to the problem when one considers groups of agents working together. By casting the agent planning problem as a partially observable Markov decision problem (POMDP), optimal policies can be generated for partially observable and stochastic environments. Exact solutions, however, are notoriously difficult to find for problems of a realistic nature. We introduce a hierarchical decision network-based planning algorithm that can generate high quality plans during execution while demonstrating significant time savings. We also discuss how this approach is particularly applicable to planning in a multiagent environment as compared to other POMDP-based planning algorithms. We present experimental results comparing our algorithm with results obtained by current POMDP and hierarchical POMDP (HPOMDP) methods.

1 Introduction

The set of domains currently addressed by intelligent systems demonstrate significant complexity and will become increasingly more complex in the near future. Single agents that assume they are acting in a deterministic world are incapable of handling many of these types of domains. The domains that agents encounter in real-world problems require that agents are capable of handling partially observable states, non-deterministic actions, and multiple interacting tasks and agents. This paper introduces and provides an initial evaluation of an agent planning architecture that supports efficient and high-quality plan generation within these types of domains and that is directly amenable to use as a means of distributed planning within societies of agents.

The architecture that we have implemented has three key features. First, it takes advantage of the natural hierarchical structure of the action spaces that are seen in many domains. By employing a hierarchical structure, agents can generate plans at multiple levels of abstraction. This allows agents to consider abstract plans without having to incur the cost of refining such plans down to the concrete level until the plans are adopted. Once an abstract plan is adopted, refinement is interleaved with execution so that an unexpected event requiring a new plan does not result in significant wasted effort. A second feature of our architecture it that it uses a factored state space representation. By encoding the problem in a factored manner via decision networks, significant reductions in memory requirements for the representation of the state space are achieved. Finally, our architecture focusses on finding the best plan for the current state, rather than for all states of the world. This substantially reduces the amount of space and computation required, allowing us to address planning problems that are beyond the ability of current POMDP and HPOMDP approaches.

Our approach of planning at multiple abstract levels is also particularly applicable when planning with societies of agents. Our architecture supports true planning at abstract levels, which allows agents within a society to share knowledge and information about individual plans without having to relay the low-level details of implementation, and our architecture can be easily implemented in hierarchical organizations of agents, allowing the organization to take advantage of the inherent hierarchical structure of the action space to facilitate distributed planning. We discuss briefly some of the issues concerning the integration of planners for POMDP environments into multiagent systems and remark on where our future work is headed.

2 Background

2.1 Hierarchical Planning

Classical hierarchical planning is predominantly structured around the theory of Hierarchical Task Networks (HTNs) [1] [2]. An HTN consists of levels of task structures, where the top levels are highly abstract goals and the lower levels are progressively more concrete means of implementing those goals. HTN algorithms attempt to find an appropriate way of decomposing and grounding task structures so that a full plan to achieve the agent's goals is generated. This method is described further in *Section 3.3* during the description of the HDDN planning algorithm, as it is one of the core ideas upon which HDDN planning is built.

2.2 Decision-Theoretic Planning

For most real world problems, the assumptions made by classical planners, including classical HTN planning, 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 a belief distribution over a set of possible world states. A number of decision-theoretic planning techniques have been developed to support problem solving in these types of domains [3].

The most realistic means of handling uncertainty during planning is to pose the planning problem as a Markov Decision Problem (MDP). Boutilier, Dean, and Hanks provide a concise definition of MDPs as the process of "control of a stochastic dynamic system" [4]. In an MDP, the world is viewed as being in one of a number of well-defined states at any instance. Events trigger transitions between states and agents can perform actions, events that move the agent through the state space. A more formal definition [5] states that 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. In this formalism, a reward function *R(s)* expresses the utility of being in a given state s. This reward function can be extended to incorporate the cost of performing an action and may also be discounted over time to represent the relative importance of future rewards. 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 that an agent is in, the policy returns what action should be taken. An optimal policy Π is defined as one in which 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, such as value-iteration [6]. Optimal policies for MDPs can be found in time that is polynomial in the size of the state and action space.

Solving POMDPs [7] is a much harder problem as knowledge of the current state of the world is incomplete. Instead, the beliefs of an agent about the current state of the world are represented as a probability distribution over all possible states. A given probability distribution is called a belief state, and the set of possible distributions is the belief space. Given the 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 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 determined. This probability distribution over observations forces a probability distribution over the outcome states. This leads to the general approach for solving a POMDP problem, casting the problem as an MDP defined over a continuous belief space and using appropriately modified MDP algorithms. One approach to solving POMDPs that is space-efficient is the use of dynamic decision networks [1] [8] [9], an extension of influence diagrams to multiple timesteps. DDNs are space-efficient as they provide for a factored representation of the true state space. Planning with DDNs is described in *Section3.1* and *Section 3.2* in the description of the HDDN planning algorithm.

2.3 Hierarchical Decision-Theoretic Planning

Novel algorithms that integrate hierarchical and decision-theoretic planning have recently been introduced within the planning field. A significant amount of work has been done in hierarchical methods for solving fully-observable MDPs, including Dietterich's MAXQ learning [10], Hauskrecht's macro-actions [11], and Parr's abstract machines [12]. Pineau's work on Policy Contingent Abstraction (PolCA) [13] is an algorithm for finding approximate solutions to POMDPs through hierarchical 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 a POMDP. A policy for the overall top-level problem is built bottom-up by finding policies for each smaller problem, where transition and reward functions for an abstract action are defined by the transition and reward functions of the corresponding optimal implementing action defined at the lower level. Individual policies are generated through exact solution methods, such as pomdp-solve [14].

3 Hierarchical Dynamic Decision Network Planning Algorithm

Planning with hierarchical dynamic decision networks is a process of multi-resolution planning using standard dynamic decision networks. Thus, a description of the HDDN planning algorithm requires both an understanding of probabilistic planning with DDNs and hierarchical planning with HTNs.

3.1 Dynamic Decision Networks

A decision network (also referred to as an influence diagram) is a variant of Bayesian networks which can be defined as a directed acyclic graph with three types of nodes: chance nodes, which represent the agent's current believed state of the world; decision nodes, which represent the actions an agent can take; and utility nodes, which represent the value of being in a world state or the cost/reward for taking an action in a given state. A dynamic decision network represents these nodes in multiple timeslices, modeling the evolution of the agent and world state over time. Both the current state and the agent's action are connected to the next world state, allowing changes in state due to both the natural progression of time and outside influences as well as direct changes made by the agent. This time-slice representation of a DDN forces both a temporal ordering over the decision nodes as well as a separation of chance nodes based on time of observation. In effect, the nodes can be partitioned into sets, I and D, where I_{i-1} is the believed state of the world before the decision D_i to perform an action is made and I_i is the state of the world after the action has been taken.

A simple example of a dynamic decision network representing a taxi delivery problem is presented in Figure 1. In this domain, there are three random variables *Taxi*, *Passenger*, and *Destination* which represent taxi, passenger, and destination locations respectively. An agent has the ability to perform six primitive actions, which are represented in the decision nodes: *Pickup* to pickup a passenger, *Putdown* to let a passenger out, and *North*, *South*, *East*, and *West* to move around the world. The example decision network represents an agent that considers the current timestep *T0* and two timesteps in the future, *T1* and *T2*, with an action occuring between each timestep. At the end of all actions, the agent receives a reward based on the location of the passenger. A natural representation of the reward would have a maximum value when the passenger and destination location are the same and progressively lower rewards for the passenger being further away from the destination.

The process of finding an optimal sequence of actions in a DDN is conceptually very similar to the process of MDP value iteration [6] or solving a decision tree [9]. In value iteration, a planner starts by finding the best action to take if only one action is allowed. This is the solution for a horizon of length one. Given this knowledge and resulting value function for the world states, the planner can repeatedly apply the same process, finding solutions for longer horizons. For a decision tree, a planner starts at the end of the tree with nodes that only have leaves as children. Leaves in a decision tree are utility nodes. If the parent of a utility node is a chance node, expected utilities for being in a given belief state can be calculated. The expected utility for a belief state is calculated by summing over the products, for all states, of the likelihood of being in a given state and the utility of being in that state. If the parent is a decision node, the utility values are explicit pre-computed expected utilities for taking each action and can be copied directly from the utility

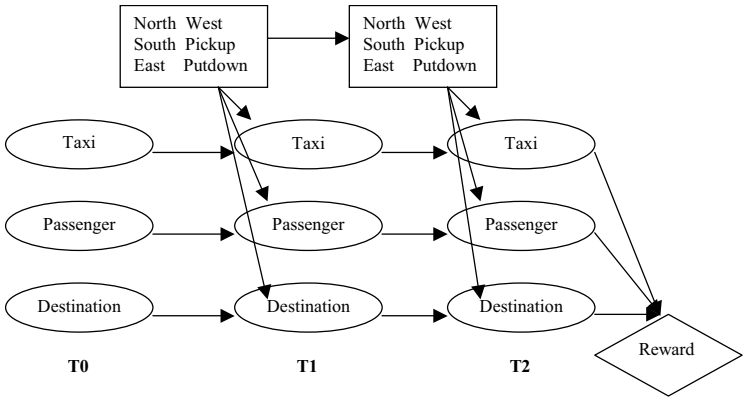


Fig. 1. A three timestep dynamic decision network representing a simple taxi delivery problem.

node. Utilities are propagated up the tree by selecting actions at each decision node that provide for the maximum utility for the agent.

While DDNs can be solved by an explicit conversion to decision trees and use of the decision tree algorithm, an alternative algorithm designed for Bayesian networks, variable elimination [15], is able to take advantage of the DDN structure for increased efficiency in finding solutions. This solution method also 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 in Bayesian network belief propagation any ordering of elimination is allowed (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 [9]. For the example dynamic decision network in Figure 1, the final *Taxi*, *Passenger*, and *Destination* nodes in timestep *T2* would be eliminated first, followed by the final decision node between *T1* and *T2*, and then repeating until all nodes in timestep *T0* are eliminated.

3.2 Planning with Dynamic Decision Networks

To begin planning with a dynamic decision network, the agent enters its evidence about the world into the initial timeslice of the network, providing information I_0. Given the policy functions calculated by the variable elimination algorithm and the current evidence concerning the state of the world, an optimal ordering of actions providing maximal utility for the agent is returned. Although the complete set of returned actions could be scheduled for execution, many decision network based planners use the advice of the planner in taking the immediate next action and then replan after execution of that action [1] [8].

An agent planning with the example dynamic decision network in Figure 1 would begin the planning process by entering evidence into the *Taxi*, *Passenger*, and *Destination* chance nodes for timestep *T0* with knowledge from its knowledge base. For fully observable variables, this entails setting the state the agent believes the variable is in to have probability 1.0 and all other states to have probability 0.0. For an agent in a partially observable domain, additional chance nodes representing observations are represented in the network. When the agent enters evidence into an observation node, it forces a probability distribution over the state of the variables representing the true state of the world dependent on the observation received. Observation nodes in a given timestep are also generally linked to the following decision node to represent that the observation is known before that decision is made. A set of optimal actions to perform is then generated by running the variable elimination algorithm. If the taxi agent using the DDN in Figure 1 has

the passenger in the taxi and is one block north of the destination, the proposed actions would most likely be *South* followed by *Putdown*. As stated above, many influence diagram based planners actually only select the first action from the plan to execute, thus choosing *South* in this instance. After the *South* action is performed, the agents new beliefs about the world are re-entered into the DDN for timestep *T0* and the planning process is restarted.

3.3 Planning with Hierarchical Task Networks

At the most general level, a planning problem is input to an HTN planner in the form of a 3-tuple: (*root task network*, *operators*, *methods*). A task network provides a set of tasks, task arguments, and task constraints, where tasks are generic actions that should be accomplished. Tasks can either be primitive, meaning they are directly performable by the agent, or abstract, meaning they can be broken down into smaller subtasks which can also be either primitive or abstract. Operators describe the results of performing a primitive action, and methods provide means for implementing abstract tasks. Methods are defined as tuples (*abstract task*, *network*) and mean that the given abstract task can be achieved if the tasks in the corresponding task network are achieved.

HTN planning is generally used to develop complete, end-to-end plans. Planning begins by instantiating the input task network. While there are abstract tasks remaining in the network, an abstract task is selected and reduced. Reduction consists of finding a second task-network which, if accomplished, achieves the higher level abstract task. If an appropriate task network is found, the tasks and constraints of the new network are incorporated into the high level network and planning continues. There may actually be many possible methods that accomplish the higher level task, and the choice of which one to use is made by a critic evaluating interactions with the other tasks in the network. When there are only primitive tasks left in the input network, the planner attempts to find a totally ordered and grounded version of the primitive tasks. If this is possible, a complete plan for the input task network is returned. There are no necessary restrictions on the form of the task network or on the manner in which abstract tasks are reduced [16]. Continuing with

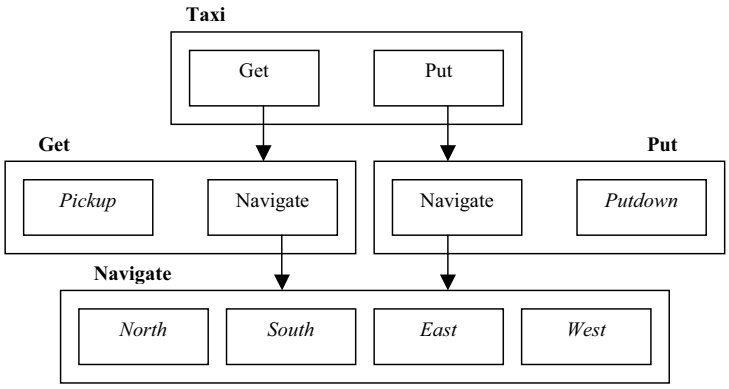


Fig. 2. An HTN representing a potential hierarchical decomposition of a taxi delivery problem.

the example taxi delivery problem, a potential HTN decomposition of the problem is represented in Figure 2. Primitive actions the agent can perform are represented in italics to contrast against abstract actions. In this figure, there are four task networks that the agent has to reason over. The *root task network* is labeled *Taxi* and is the general problem the agent is interested in solving. In this decomposition, the *Taxi* network can be fulfilled by finding appropriate means of implementing the *Get* and *Put* tasks. Recursively, a method can be found for the *Get* and *Put* tasks by ordering *Navigate* actions with the appropriate *Pickup* and *Putdown* primitive actions. Finally, if a *Navigate* action is useful for implementing the *Get* or *Put* network (the general

case when the taxi is not at the passenger or destination location), the *Navigate* task can be implemented by an appropriate ordering of the *North*, *South*, *East*, and *West* primitive actions. The final plan returned by the HTN planner would consist of only primitive actions and would be a total ordering over those actions that obeys constraints between the actions (such as the agent can't perform a putdown without a passenger in the car) and which achieves the goal of delivering the passenger to his destination.

3.4 Planning with Hierarchical Dynamic Decision Networks

Planning with hierarchical dynamic decision networks is an extension of standard DDN planning to multiple resolutions. HDDN planning ties together DDN and HTN planning by allowing the actions that an agent plans over in a decision network to now be either primitive actions, directly executable by the agent, or abstract actions, requiring further planning. An HDDN planner maintains a library of dynamic decision networks with one network for each abstract action defined for the domain. These are analagous to task networks and methods in an HTN planner. The Markovian state transitions defined in each DDN are analagous to operator definitions in HTNs, as they define the effects of actions on the states of the world over time. Each task library minimally consists of a top-level Root decision network. If this Root network is the only network present in the network library, then all actions represented within decision nodes in the network must be primitive actions and this case reduces to standard DDN planning. This case is also analagous to planning with an HTN consisting solely of primitive tasks, as the problem is reduced to finding a total ordering over the actions that accomplish the root task.

A partial view of an HDDN decomposition of the example taxi problem domain, integrating the structures of Figures 1 and 2, is illustrated in Figure 3. In this figure, the navigate decision network is not represented due to space constraints. To exploit the hierarchical structure of many action spaces, most Root networks

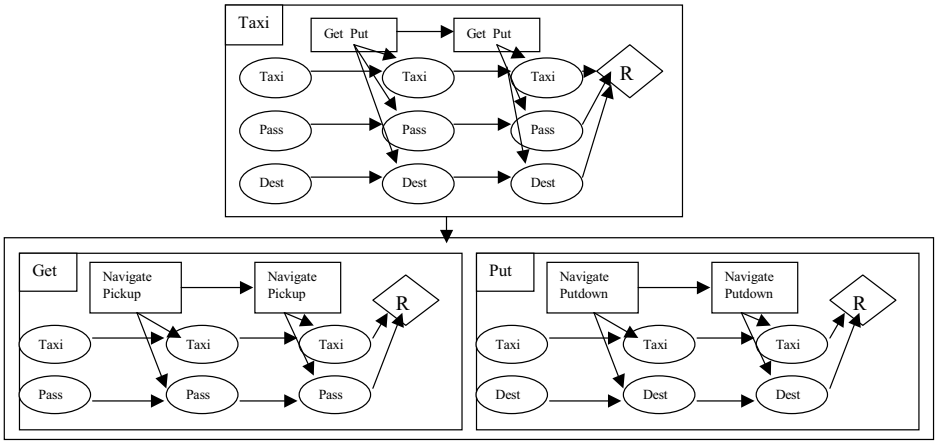


Fig. 3. A partial view of the HDDN decomposition of the taxi delivery problem.

will involve decisions over abstract actions or a mix of abstract and primitive actions. Given a hierarchical structure, an HDDN planning agent first instantiates the Root level dynamic decision network, enters current evidence about the world state, and finds an optimal sequence of actions through the standard DDN variable elimination algorithm. This sequence of actions is stored and the first action is selected to be scheduled. At this point, a minor divergence between HTN planning and HDDN planning appears. While in HTN planning there are no restrictions on the ordering of task reduction, the HDDN algorithm forces such an order, always selecting the optimal first action as the action to reduce. However, if one considers the actions defined in the root network as the set of input tasks, this approach can be cast in HTN terms as applying a critic to evaluate high level interactions and guide planning before the first reduction takes place. This approach can be also be cast in alternative HTN terms if one considers the initial task network to contain only the single

abstract task Root. The process of finding an optimal sequence of actions (via the DDN) to implement Root is the same process as finding the most appropriate method for reducing the Root action, where the set of all possible methods to select from is all possible orderings over the actions defined in the Root DDN. Finding an ordering early on is also particularly important for our agent to integrate planning and execution as it allows the planner to focus on the tasks that need to be performed first. Considering the example HDDN in Figure 3, the agent would begin planning by instantiating and planning over the root *Taxi* decision network. This would entail running the variable elimination algorithm on the network, only considering the two abstract actions *Get* and *Put* as possible actions to perform. A likely plan at this level would be *Get* followed by *Put*.

Once the agent has chosen the first action, that action is checked to determine if it is primitive or abstract. If the action is primitive, the planner stops until the action is performed and the results of execution are fed back into the planner. Alternatively, this first planned action could be an abstract action. In this case, the agent needs to refine its plan and determine a means of implementing the selected abstract action. To do this, the agent instantiates the dynamic decision network from the network library that corresponds to the new abstract action. Current beliefs are incorporated as evidence into the first timeslice of this network, and a set of new optimal actions for implementing the higher level abstract action are returned via the variable elimination algorithm. This process is recursively repeated until a primitive action is generated as the first action to perform. Once a primitive action is found and executed, planning is restarted with the Root task network. The HDDN planning algorithm as described above is shown in psuedo-code as Algorithm 1.

Algorithm 1 Hierachical 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
```

Continuing the example, the agent would select the first action from its current plan, *Get*, and since this action is not primitive, would need to find a way to implement the action. Accordingly, the agent would instantiate the *Get* DDN from its library, enter evidence on the beliefs, and find an optimal set of actions for implementing *Get*. This would most likely return a plan of *Navigate* followed by *Pickup*. Since the *Navigate* action is not primitive, the DDN for *Navigate* would be consulted to find which directional moves to make. If the *Navigation* DDN only reasons over the primitive *North*, *South*, *East*, and *West* actions, then the first action returned from this network would be executed.

This process creates multiple plans at different abstract levels, with the actions that are soon to be performed being concretely chosen and future actions represented only at abstract levels. One possible plan created by a taxi agent using the HDDN algorithm is represented in Figure 4 below. In planning with dynamic decision networks, our architecture uses two approximations to simplify the planning problem. Both approximations are reasonable to make in the context of the continual DDN-based planning algorithm we have developed. The first approximation is that the agent only reasons over a small number of timesteps in each dynamic decision network. Complete planning for finite horizons in a DDN is possible, but can quickly become expensive. To work around this problem, agents in our architecture only reason over a small sequence of actions and approximate the future value of states by using an MDP approximation (this process is described in *Section 4*). With discounted rewards, this approximation of the future values of states does not significantly [1] affect the quality of the plans that are generated.

The second approximation that is made is to limit explicit informational links between observation nodes and decision nodes to one step - between the observation node after the first action and the second decision node. Since the first timeslice represents the current state of the world, our model states that the information

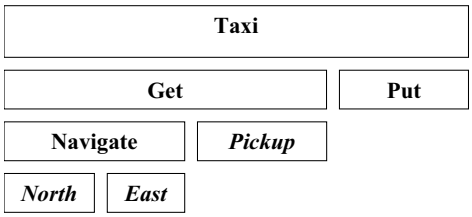


Fig. 4. An sample initial plan developed by the HDDN algorithm given the decomposition in Figure 3.

gained from the first action should be used to guide the selection of the second action. This approximation is reasonable for two reasons. 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 cost of performing tests does not decrease if delayed, so it is generally optimal to have information links early in the DDN to gather as much information as possible early on.

3.5 Benefits of HDDN Planning

While the approach of replanning down a complete task hierarchy at each time step is computationally expensive, we adopted this algorithm because it allows the agent to build better plans than the faster approach of extending current plans. In essence, our agent is continually trying to determine if another ordering over actions or method of reduction is more appropriate than the current plan. This approach allows for rapid reaction to changes in the environment and can even allow the agent to take advantage of opportunities, both of which are important in the dynamic and partially observable domains the agents will be acting in. Our agent is also able to plan efficiently in this manner because it is never finding a complete plan for the input problem but rather only finding the first action to execute and a set of higher level abstract actions that should be appropriate in the near future. Low level tasks do not need to be planned for until they are close to execution. This can save significant planning time since, by avoiding detailed planning early on, the agent does not waste time planning for situations likely to change.

A second reduction in complexity is achieved by exploitation of the fact that most of the task networks in an HDDN hierarchy deal with actions and states that are independent between networks. It is often possible to reduce the number of features that define the state space for a given decision network by ignoring information that is no longer relevant given the hierarchy. Consider the example of the previously described taxi problem. Assume that the high level goals for a taxi agent are to pickup up a passenger and deliver him to his specified location within a twenty block world. A simple model for this domain may have twenty-one locations for the passenger (any block and in the taxi), twenty locations for the taxi, and twenty locations for the destination. Using a flat decision network, the entire state space of 8400 states would need to be represented and planned over in each iteration of the planning algorithm. Now assume that the agent is working with a hierarchy of DDNs and is faced with implementing a Navigation task. In a hierarchically defined Navigation task where the taxi is only interested in how to get from one location to the other, either the passenger or the destination location can be ignored from the feature space, reducing the number of states that have to be planned over from 8400 to at most 420. Similarly, a reduction in the number of actions, a significant factor in planning complexity, can be reduced. In the taxi delivery example, an implementation of a flat DDN would need to reason over at least six primitive actions (Pickup, Putdown, North, South, East, and West) during each iteration, while in a hierarchical structure, each abstract task requires reasoning over only a subset of these actions. This state space and action space reduction can be seen by comparing Figure 1 and Figure 3 above. In Figure 1, the agent always has to plan over the six possible primitive actions and the complete state space. In the HDDN decomposition in Figure 3, however, fewer actions are considered in each of the abstract action decision networks, and in many cases, the state space considered is reduced due to irrelevance.

4 Developing a Hierarchical Domain Model

Given the HDDN algorithm for planning, one also needs a way of defining action and utility models for the dynamic decision networks that are being planned over. In our architecture, we intend for abstract actions to be thought of by the agent as if they were primitive actions. In other words, we would like each abstract action to have an associated probability distribution describing possible outcomes and the utility of those outcomes, thereby supporting abstract planning without the requirement of having to evaluate more concrete refinements of the plan. A problem, however, with modeling abstract actions in this manner is that one does not know the probability for transitions between abstract states and the rewards associated with completing an abstract action, as the problem specification only represents these values over concrete primitive actions.

Consider, as an example, the parts planning problem [21]. 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, 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. Painting 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 either rejecting a FL-BL-!PA part or shipping a !FL-!BL-PA part, and pays a cost of -1 for shipping or rejecting a part in any other state. The agent can inspect and paint the part for free.

Instead of reasoning over the complete action space, it is more efficient for the agent to reason over a smaller action space where it only needs to consider, for example, whether to *Inspect*, *Reject*, or *Process* the part, where *Process* is an abstract action that is the aggregation of *Paint* and *Ship*. If the *Process* action is chosen, the agent needs to determine its implementation by finding an optimal ordering over *Paint* and *Ship* actions. The agent's hierarchical 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. These values do not need to be computed for the actions *Inspect* and *Reject* since they are primitive actions and can be copied directly from the original domain description. Finding the correct values for the *Process* 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 [13], but these subproblems may be nearly as hard as the original problem to solve.

Our implemented approach, on the other hand, models the problem similar to the manner of HTN planning. Our model assumes that the effects of executing an abstract action are the effects of successful completion of the action. We also initially approximate the rewards and costs for taking actions and values for being in a given state 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 those actions 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 method for generating initial estimates of the true utility functions that requires time polynomial in the size of the state and action space for each abstract action. This approximation does not take into account value of information and the use of query actions, both of which are important in POMDP problems. More formally, the reward functions can be defined as follows. Let $R_{input}(A, S)$ be the reward/cost value defined for taking a given action a in a given state s in the input problem definition, and let $R_{ddn}(A, S, T)$ be the reward/cost for taking a given action a in a given state s in timestep t for a given DDN, where t starts at the value 0. Let $V_{ddn}(S)$ be the value function for being in a given final state in the current DDN (a utility function attached to the final states in the domain instead of being directly tied to an action and state). Let A_{ddn} be the set of actions available in the current DDN, and S_{ddn} be the set of states present in the current DDN. Let h be the maximal timestep t defined in the current DDN. Finally, let *MDPSolve(A,S,R)* be a function that solves a fully observable, infinite horizon MDP over a set of actions A, states S, and rewards R. For an abstract action DDN made up of only primitive actions:

$$R_{ddn}(A, S, T) = 0.95^T * R_{input}(A, S); \tag{1}$$

$$V_{ddn}(S) = 0.95^{h+1} * (MDPSolve(A_{ddn}, S_{ddn}, R_{input}(A, S))) \quad (2)$$

For an abstract action defined over abstract and primitive actions, the only change required to this definition is that rewards/costs for taking abstract actions are now represented by their MDP approximations. For primitive actions, $R_{ddn}(A, S, T)$ is still defined the same. For abstract actions,

$$R_{ddn}(A, S, T) = 0.95^{T} * V_{ddn}(S) \quad (3)$$

where $V_{ddn}(S)$ is from the DDN that implements the abstract action, calculated without the (0.95^{h+1}) factor. The final state value function is also updated to use the input rewards for primitive actions and approximated MDP rewards for the abstract actions,

$$V_{ddn}(S) = 0.95^{h+1} * (MDPSolve(A_{ddn}, S_{ddn}, R_{ddn}(A_{abstract}, S, 0) \vee R_{input}(A_{primitive}, S))) \quad (4)$$

Selection of an appropriate horizon h, the number of steps the agent looks ahead while planning, is directly correlated with both the quality of plans produced and the computation required for planning. An initial heuristic for choosing this horizon in agents planning with our approach is to include one timestep of lookahead for each possible abstract action that can be performed. Given that our agent models the effects of executing an abstract action as the effects of successful completion of the action, the agent, by planning over the maximal number of abstract actions, can generate a nearly complete plan for implementing the first selected action at each abstract level. For information intensive domains, in which there is a less formalized idea of task completion, the designer should balance computation time with plan quality. Though a longer plan length provides a value function estimate closer to the true value function, this extra modeling incurs a significantly higher cost. Selecting an appropriate hierarchy is also a difficult problem but is not the current focus of our research. When possible, we use hierarchies defined for a problem as they are specified in the current literature.

Continuing the parts painting example, we model the effects of the the *Process* abstract action as 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 estimated reward depends on a part being painted and shipped. Solving this problem as a fully observable MDP, the rewards for !FL-!BL-!PA and !FL-!BL-PA are defined as .9 and 1 respectively (both move you 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 problem where the agent reasons over primitive actions *Inspect* and *Reject* and the abstract action *Process*.

5 Distributed Planning with HDDNs

Although our initial evaluations are only in single agent domains, we believe that our algorithm is preferable to other POMDP-based algorithms for planning in agent societies. In distributed systems, it is particularly useful for agents to be able to share plans at abstract levels. The primary use of plan sharing is to allow agents to reason over how their plans interact. This principally entails determining whether conflicts exist between plans or if there are actions that may be useful or enabling for other agents' plans [17]. A significant amount of research in this area has been performed, with prominent works including Durfee's Partial Global Planning (PGP) [18] and Durfee's Hierarchical Behavior Space Search [19]. In both of these methods, agents perform distributed planning through sharing of abstract plans. In Hierarchical Behavior Space Search, agent develop local plans at multiple levels, and then attempt to develop good global plans through a recursive procedure of conflict resolution. This procedure attempts to resolve conflicts at the most abstract level first, and if the conflict can't be resolved there, moves to more concrete plan levels. In PGP, abstraction is used to allow an agent to commit to perform actions at an abstract level, while not having to commit to a particular implementation of those actions. This allows the agent to have significant flexibility at the

local level, while interacting well at the social level. Most of this work, however, deals with pre-generated or classically generated plans and there has been less focus on performing distributed planning with decision-theoretic planners.

Incorporating abstract plan sharing with POMDP and HPOMDP planning is problematic. Since POMDP and HPOMDP are both offline planners, their use precludes true interactive planning between agents. In addition, if agents were interested in sharing their current plans, the use of a POMDP policy table or a hierarchy of HPOMDP policy tables to select actions does not provide the ability for agents to model plans at a significant level of abstraction for sharing. Using POMDP tables, there is no explicit model of the actions to take for steps in the future, and all actions are modeled at the primitive level. For HPOMDP policy tables, it is possible to extract abstractions of the currently executing primitive action fairly easily, but future plan steps are not explicitly modeled. In contrast, by modeling abstract actions during planning as complete actions and reasoning over multiple plan steps, our architecture does support the generation of true multi-action plans at each level of the planning hierarchy. This significantly facilitates the integration of our architecture within coordination frameworks similar to the ones described above.

Additionally, our developed planning architecture should also be applicable in the commonly seen hierarchical model of agent organizations. Instead of embedding multiple decision networks in a single agent, the decision networks could be hierarchically distributed to an appropriate set of agents. Knowledge of the world state, abstracted to the appropriate level, would be communicated up the hierarchy, while goals are distributed down the hierarchy. We plan on devoting significant future work to evaluating these two hypotheses.

6 Experimental Results

An initial evaluation of our agent planning architecture has been performed using four different planning problems. Each planning problem differs significantly in terms of its features and was selected to evaluate the ability of our approach to work in multiple types of domains and to allow a comparison of our results with published results for POMDP and HPOMDP methods. The agents used to test our planning architecture are Java-based agents, built on top of the Zeus agent architecture. The dynamic decision network algorithms are called through the Hugin Java API. All tests on our architecture were run on a Sun dual 750MHz processor Sun-Fire-280R with 4 gigabytes of RAM. To generate the MDP approximations for abstract rewards, a Java based exact MDP solver [20] was used, solving with a discount factor of 0.95.

6.1 Parts Problem

The first problem of interest is the parts planning problem [21] that was described earlier as an example for implementing hierarchical dynamic decision networks. A complete POMDP description of the domain is available at *http://www.cse.sc.edu/~turkett/hddn/Parts.pomdp*. This problem is interesting because it is simple enough that an optimal policy can be found by an exact POMDP solver [14]. This allows a comparison of the policy generated by our algorithm with the true optimal policy. In addition, this problem has also been evaluated by Pineau using the HPOMDP algorithm [13]. The optimal policy consists of performing a single inspect action. If this action returns BL, the part is rejected, otherwise the part is painted and shipped. The policy generated by both HPOMDP and our 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, in terms of utility, this policy is as good as the optimal policy. The extra action requires more time, however, to execute. Total planning time, as measured by Pineau, for the HPOMDP algorithm, and measured locally for our HDDN algorithm and POMDP solution, are shown below in Figure 5. The total planning time for the HDDN algorithm is the sum of 4 milliseconds of MDP reward generation and 5.03 milliseconds for execution time planning. The values in the figure for HPOMDP is the value (5.84 seconds) reported in Pineau's paper [13], corrected by the speedup (2.02x) seen when running the POMDP algorithm on our faster machine (19.43 seconds versus the original Pineau POMDP time of 39.28 seconds). Note that there are orders of magnitude difference between the HPOMDP approach and the HDDN algorithm that are not attributable to machine differences.

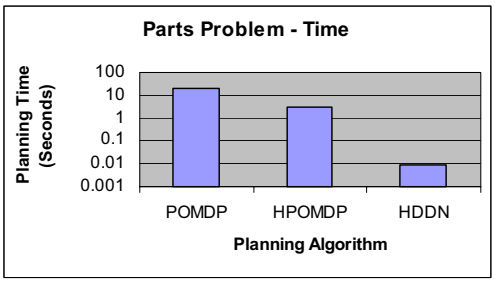


Fig. 5. Average planning time for the parts planning problem.

6.2 Cheese Taxi

The cheese taxi problem [13] was originally tested by Pineau for the HPOMDP algorithm. This problem is a combination of Dietterich's taxi problem [10] and McCallum's cheese maze problem [22]. The domain, shown in Figure 6, is small, with a total of 33 possible states produced by combining eleven taxi locations (*0, 1, 2,...,10*) with three taxi destinations (*0, 4, 10*). The taxi agent is initially placed in a random location on the map and is tasked with picking up a passenger from location *10* and carrying him to a requested destination. The taxi agent can perform seven actions: *North*, *South*, *East*, *West*, *Pickup*, *Putdown*, and *Query*. Upon performance of an action, the agent receives one of ten observations (*O1..O7, D0, D4, NULL*). If a movement action is performed, a localizing observation (one of *O1..O7*) is returned. Destination observations, *D0* and *D4*, denote the intended passenger destination and are returned after a *Query* action if a passenger has been picked up. Finally, if the agent performs a *Pickup* or *Putdown* action, the *NULL* observation is returned. The taxi agent suffers a cost of -1 for each individual action that is performed and a cost of -10 if the taxi agent attempts to pickup or putdown the passenger in the wrong place. A reward of +20 is obtained for performing a putdown action at the appropriate destination. This problem is interesting as a POMDP as there are several sources of uncertainty. The agent is initially unsure of both the taxi location and initital passenger destination, and can only disambiguate those parameters by taking the appropriate actions. To add additional uncertainty, the passenger can switch destinations if the taxi is navigating through location 2 or 6 of the map. This new location can only be discovered through the use of another *Query* action. A complete POMDP description of the domain is available at *http://www.cse.sc.edu/~turkett/hddn/CheeseTaxi.pomdp*.

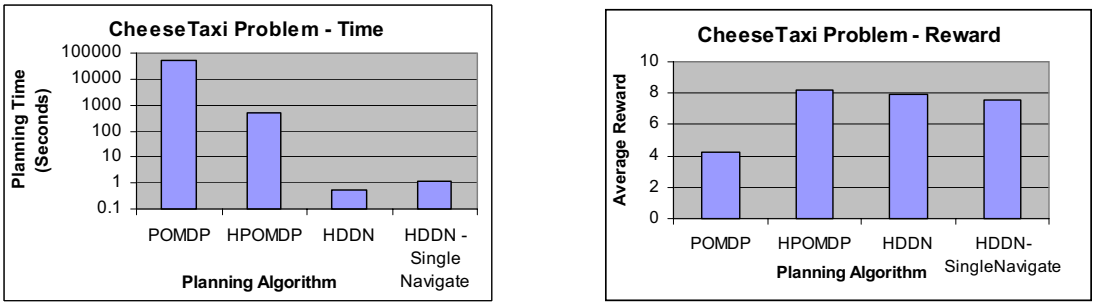
S0 **D**	**S1**	**S2**	**S3**	**S4** **D**
S5		**S6**		**S7**
S8		**S10** **P**		**S9**

Fig. 6. A map of the cheese taxi domain.

The initial hierarchical decomposition used by both HPOMDP and our algorithm consists of a top-level *Root* action that is defined over two abstract actions, *Get* and *Put*. In the HPOMDP formalism, the *Root* action is a POMDP over the two abstract actions, while in our formalism, it is a dynamic decision network that reasons over the two actions. 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 and the primitive action *Query*. Finally, each of the navigate abstract actions is defined over the primitive actions *North*, *South*, *East*, and *West*. This decomposition is very similar to the one shown in Figure 3. Pseudo-rewards of zero cost for actions in goal states, such as for being in state 0 for *NavigateS0*, were used identically to Pineau's use.

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 second 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*.

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 our planning algorithm on this problem, we ran one hundred simulations with our agent. In each simulation, starting and destination locations were chosen according to the world model distributions defined in the original problem description. While the agent was running, we recorded the total planning time, number of actions performed, and total reward gained by the agent. Pineau has similar results for the HPOMDP algorithm. A comparison of average reward and average planning time are shown in Figures 7a and 7b below. In this problem, the POMDP algorithm failed to converge and was stopped by Pineau after 24 hours and only completing five iterations of computation. The Root and Navigate tasks for Pineau's HPOMDP approach also failed to converge and were stopped after several hours of computation. In both cases, the last completed step of planning was used to guide the agents to examine accrued rewards. In contrast, the HDDN algorithm was able to find a solution in all 100 simulations. The times presented in the total planning time graph are shown using the expected 2.02x speedup factor of running the alternative algorithms on the same machine as our algorithm. For the original action hierarchy, the planning time required by our algorithm was 15 milliseconds for MDP reward generation and 510 milliseconds of execution time planning. For the alternative hierarchy, 15 milliseconds was required for MDP reward generation and 1.107 seconds for execution time planning. This increase in execution time planning is directly attributable to the more complex *Navigate* task. A comparison of average rewards earned from following policies generated by each algorithm demonstrates that our approach is capable of performing at the same level as other approximate POMDP algorithms. This is significant given the marked decrease in computation needed to find solutions with our algorithm.



(a) Average planning time.

(b) Average reward.

Fig. 7. Results for the cheese taxi domain.

6.3 Large Cheese Taxi

The large cheese taxi domain is an extension of the original cheese taxi problem, preserving 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. Scaling the state space by an order of magnitude results in a problem that exceeds the capability of current POMDP and HPOMDP methods and provides preliminary empirical evidence concerning the scalability of our approach. 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. The passenger can also change his mind between destinations 20 and 24 if navigating through locations 18, 22, or 26. Other than these changes, the rewards and transition probabilities are the same as in the original cheese taxi 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. A complete POMDP description of the domain is available at *http://www.cse.sc.edu/~turkett/hddn/LargeCheeseTaxi.pomdp*.

In this problem, a hierarchical decomposition which views navigation to each destination separately causes a significant increase in the size of the decision network for the abstract action *Put* (this problem was mentioned in the previous section). Using separate navigation decompositions, the *Put* abstract action network was unable to be compiled into a junction tree, a required step for the Hugin influence diagram variable elimination algorithms. However, a single *Navigate* abstract action, as explained for the standard cheese taxi problem, does work, although it entails a trade off of action complexity for a larger probability distribution representation.

As is the case with the standard cheese taxi problem, the state space for LargeCheeseTaxi is too large for current POMDP solvers. Consequently, the true optimal policy cannot be computed. To evaluate the HPOMDP algorithm, we began with the single *Navigate* abstract action, as all other abstract actions require the *Navigate* results. After 41 hours of computation, the exact pomdp solver was still working on iteration three, and the policy generated from iteration two only suggested *North* actions. Since the flat POMDP space requires planning over more actions than the *Navigate* abstract action, we expect that the time required for an exact solver to reach the third iteration on the flat space will be significantly larger than the forty one hours shown. To evaluate our planning algorithm on this domain, one hundred simulations of our agent were performed. Start and destination locations were selected according to the probability distributions defined in the world model. For each run, the total planning time, total number of actions, and total reward were calculated. The total planning time for our algorithm was 56 milliseconds for MDP reward generation and 89.984 seconds for execution time planning. No comparison of reward is shown, as the policy returned for the HPOMDP *Navigate* action is not sufficient to lead an agent to do anything more than perform the action *North*. Our agent, howeever, was consistently able to complete the taxi navigation task and earned an average reward of 8.44.

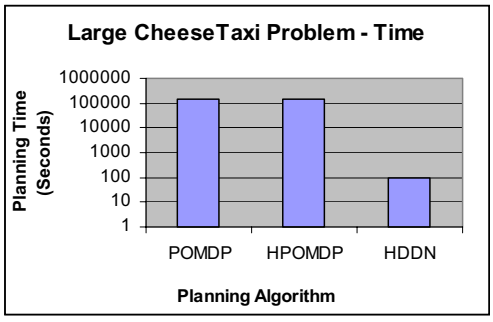
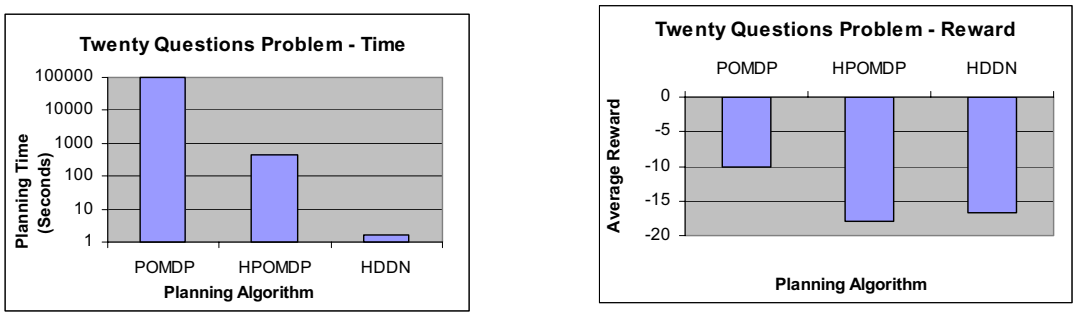


Fig. 8. Average planning time for the large cheese taxi problem.

6.4 Twenty Questions

The final problem on which our algorithm was tested was Pineau's twenty questions domain [13]. This domain is different from the previous domains in that it is completely information intensive. In this problem,

an outsider selects an object, while our planning agent asks yes or no questions to try to determine what the object is. The number of states in the domain is the number of objects that the outsider can select from. The player, in each turn, can either ask one of a set of yes/no question concerning a feature of the object, or attempt to guess the object. In the test domain, there are 12 objects (*Tomato, Apple, Banana, Potato, Mushroom, Carrot, Monkey, Rabbit, Robin, Marble, Ruby, Coal*). The player in the test domain is allowed to perform twelve *GuessX* actions corresponding to guessing one of the twelve objects and eight yes/no *AskX* questions: *AskAnimal, AskVegetable, AskMineral, AskFruit, AskWhite, AskBrown, AskRed, AskHard*. Observations are from the set (*Yes, No, Noise*). The answer to an ask type question is answered correctly 85% of the time, incorrectly 10% of the time, and as Noise 5% of the time. In addition, extra uncertainty is introduced by allowing the outsider to change objects with an 11% probability after answering a question (this leads to some disheartening changes just before the agent makes a guess) and by having objects that are ambiguous with regard to questions about features. Each question costs the asking agent -1, an incorrect guess has a cost of -20, and a correct guess has a reward of +5. A complete POMDP description of the domain is available at *http://www.cse.sc.edu/~turkett/hddn/TwentyQuestions.pomdp*. A comparison of total planning time and total reward are shown in Figures 9a and 9b below. The total time required for our planning algorithm was 13 milliseconds of MDP reward generation and 1.627 seconds of execution time planning. On this problem, we were able to run the POMDP algorithm on our local machine. The planning time required for the POMDP solution closely follow the results of Pineau, so no factor for potential speedup due to difference in machines is taken into account. The reward graph for this problem demonstrates that our HDDN algorithm is able to generate solutions at a level of quality equivalent to HPOMDP, while requiring significantly less computation.



(a) Average planning time.

(b) Average reward.

Fig. 9. Results for the twenty questions domain.

7 Conclusion

The successful application of agents to real-world problems hinges on both the ability of agents to select optimal or close to optimal actions while acting in complex environments, as well as the ability to compute those actions under reasonable time constraints. Exact POMDP solvers, capable of generating true optimal policies, unfortunately require enormous amounts of computation and space. In many domains, however, there are natural problem features, such as a hierarchically structured action space, that can be exploited to generate high quality approximate solutions with modest space requirements while achieving significant time savings.

We have developed and performed an initial evaluation of a hierarchical decision network-based planning algorithm which can be used in POMDP environments. The initial results of our evaluation are very promising. The quality of HDDN solutions in this initial set of experiments compares favorably with HPOMDP

results. However, in terms of performance, the result is superior. The HDDN solution is computed several orders of magnitude faster than the corresponding HPOMDP solution. More importantly, our HDDN method is able to find solutions to problems that are beyond the capability of current POMDP and HPOMDP methods. We believe these improvements are due to three features of our algorithm: exploitation of hierarchical action spaces, the use of a factored state space representation (by encoding the problem as a decision network), and searching only the relevant parts of the state space (by planning at run-time).

References

1. Russell, S. and Norvig, P. 1995. *Artificial Intelligence: A Modern Approach*. Englewood Cliffs, New Jersey: Prentice Hall.
2. Nau, Dana; Cao, Yue; Lotem, Amnon; and Munoz-Avila, Hector. 1999. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the International Joint Conference on Artificial Intelligence 1999*. 968-973.
3. Blythe, Jim. 1999. Decision-Theoretic Planning. *Artificial Intelligence* 20(1).
4. Boutilier, Craig; Dean, Thomas; and Hanks, Steve. 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research* 11:1-94.
5. Boutilier, Craig. 1996. Planning, Learning, and Coordination in Multiagent Decision Processes. In *Proceedings of the Sixth Conference on Theoretical Aspects of Rationality and Knowledge*. 195-201. Springer-Verlag.
6. Bellman, Robert. 1957. Dynamic Programming. Princeton, NJ: Princeton University Press.
7. Cassandra, Tony. 1999. POMDPs for Dummies. Department of Computer Science, Brown University. URL: http://www.cs.brown.edu/research/ai/pomdp/tutorial/index.html.
8. Forbes, Jeff; Huang, Tim; Kanazawa, Keiji; and Russell, Stuart. 1995. The BATmobile: Towards a Bayesian Automated Taxi. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*.
9. Jensen, Finn V. 2001. Bayesian Networks and Decision Graphs. New York, NY: Springer-Verlag.
10. Dietterich, Thomas. 2000. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition Function. *Journal of Artificial Intelligence Research* 13:227-303.
11. Hauskrecht, M.; Meuleau, N; Boutilier, C.; Kaelbling, L.; and Dean, T. 1998. Hierarchical solution of Markov decision processes using macro-actions. In *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence*. 220-229.
12. Parr, Ron and Russell, Stuart. 1997. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Machines*. 10. MIT Press.
13. Pineau, Joelle and Thrun, Sebastian. 2002. An integrated approach to hierarchy and abstraction for POMDPs. Technical Report CMU-RI-TR-02-21. Carnegie Mellon University School of Computer Science.
14. Cassandra, Tony. 1999. pomdp-solve software. Department of Computer Science, Brown University. URL: http://www.cs.brown.edu/research/ai/pomdp/code/pomdp-solve-v4.0.tar.gz
15. Madsen, Anders L, and Jensen, Finn V. 1999. Lazy evaluation of symmetric bayesian decision problems. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. 382-390.
16. Erol, Kutluhan; Hendler, James; and Nau, Dana. 1994. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*. 1123-1128.
17. Lesser, Victor R.; Decker, Keith; Wagner, Tom; Carver, N.; Garvey, A.; Horling, B.; Neiman, D.; Podorozhny, R.; Nagendra Prasad, M.; Raja, A.; Vincent, R.; Xuan, P.; and Zhang, X. Q. 2002. Evolution of the GPGP/TAEMS Domain-Independent Coordination Framework. Technical Report 02-03. Department of Computer Science, University of Massachusetts-Amherst.
18. Durfee, Edmund H. 1988. *Coordination of Distributed Problem Solvers*. Boston: Kluwer Academic Press.
19. Durfee, Edmund H. and Montgomery, Thomas A. 1991. Coordination as Distributed Search in a Hierarchical Behavior Space. In *IEEE Transactions on Systems, Man, and Cybernetics: Special Issue on Distributed Artificial Intelligence*. 21(6):1363-1378.
20. Grinkewitz, Ryan. 2001. MDP Solve software. Department of Computer Science, Drexel University. URL: http://plan.mcs.drexel.edu/courses/software/mdp/
21. Draper, Denise; Hanks, Steve; and Weld, Daniel. 1993. Probabilistic Planning with Information Gathering and Contingent Execution. Technical Report 93-12-04. Department of Computer Science and Engineering, University of Washington.
22. McCallum, R. Andrew. 1993. Overcoming Incomplete Perception with Utile Distinction Memory. In *The Proceedings of the Tenth International Machine Learning Conference (ML'93)*.