Incorporating Decision-Theoretic Planning in a Robot Architecture

G. L. Peterson and D. J. Cook

Department of Computer Science and Engineering
The University of Texas at Arlington
P.O. Box 19015
Arlington, TX 76019-015
{gpeterso,cook}@cse.uta.edu

Keywords: robot control architecture, decision theory, planning

Abstract

The goal of robotics research is to design a robot to fulfill a variety of tasks in the real world. Inherent in the real world is a high degree of uncertainty about the robot's behavior and about the world. We introduce a robot task architecture, DTRC, that generates plans with actions that incorporate costs and uncertain effects, and states that yield rewards.

The use of a decision-theoretic planner in a robot task architecture is demonstrated on the mobile robot domain of miniature golf. The miniature golf domain shows the application of decision-theoretic planning in an inherently uncertain domain, and demonstrates that by using decision-theoretic planning as the reasoning method in a robot task architecture, accommodation for uncertain information plays a direct role in the reasoning process.

1. Introduction

Robot task architectures represent an assemblage of algorithms that interpret user goals and sensor readings, in order to determine robot motor actions that will reach the user-specified goals. Many current approaches to robot task architectures use symbolic planning to perform high-level reasoning. This reasoning generates a sequence of actions for problems that transition the robot agent from an initial condition to a goal condition. In classical symbolic planning problems, the agent is given all relevant information about the world and executed actions are both deterministic and always assumed to succeed.

The use of a symbolic planner in a robot task architecture overlooks the inherent uncertainty in robotic domains. The uncertainty stems from the existence of multiple possible action effects, unreliable sensors, and incomplete domain information. Instead of having the planner address these issues, current task architectures choose to have other parts of the architecture handle problems arising from uncertainties or to create domain-specific and robot-specific workarounds that allow the robot to complete the given task.

To meet the goal for robots to complete tasks in the real world, the robot must deal with the uncertainty associated with actions, sensors, and information. Additionally, actions in the real world have associated costs. To perform this reasoning within a robot architecture, we have developed the Decision-Theoretic Robot Controller (DTRC).

Our robot task control architecture contains three main elements: the DT-Graphplan planner, the robot skills, and the execution monitor. The DT-Graphplan planner is a decision-theoretic planner that generates a satisficing plan for domains with incomplete information, stochastic action effects, state reward conditions, and action costs. DT-Graphplan performs high-level reasoning over the domain, generating action sequences to transition the agent from the initial condition to the goal condition. The robot skills are a collection of low-level base building blocks that handle motor and sensor control and integration, essentially the actions the planner uses to assemble a plan. The execution monitor communicates between DT-Graphplan and the robot behaviors by maintaining current domain information for the planner and activating behaviors based on the plan provided. DTRC's execution monitor also detects plan failure and triggers replanning from the current condition.

We illustrate DTRC on a robot miniature golf domain. For robot miniature golf, the task the robot must address is to get the golf ball into the cup with the fewest number of strokes. The course has a number of stationary and non-stationary obstacles to avoid. The robot has three methods of moving the ball, and each action has a different stroke penalty (cost), and outcome. Our results are generated from an implementation of this domain using a Pioneer 1 robot. This domain demonstrates the application of decision theory in a robot architecture, shows how changes in the domain result in changes in the preference of plans, and describes how the robot architecture sustains a probabilistic representation of the domain state and generates initial conditions for replanning.

This work introduces the DT-Graphplan planner and our robot control architecture, illustrating the application of the planner to imprecise and uncertain domains such as robotics. Section 2 briefly covers related robot control architecture work. Section 3 discusses the interactions of the elements of the task control architecture, and design of DT-Graphplan to handle decision theory instead of using a strictly Bayesian method. Section 4 contains the results of applying the architecture to the task of robot miniature golf, and the required elements to convert to the robot soccer domain and how altering the domain description changes the overall behavior of the robot. We finish with concluding remarks and possible extensions of this work.

2. RELATED WORK

The standard robot task architecture consists of a planner, a sequencer, and a set of robot behaviors or skills. The robot architectures divide the entire robotic planning process into planning, sequencing, and basic skills so that the robot can reason about actuator or control errors while also dealing with a changing set of interacting goals. In the basic sense, we abstract the planning phase away from the robot skill level to increase planning speed and success.

Two of the best-known control architectures are 3T [6], and the Task Control Architecture TCA [12]. Both systems use the symbolic planner PRODIGY [14] for high-level task planning. Both 3T and TCA use a large set of skills to perform tasks that are not specifically part of the planning problem such as navigation, path planning, and item searching. The sequencer used by 3T is the RAPS reactive plan execution system that is specifically designed to accept plans and goals at a high level of abstraction and expand them into detailed actions at run time [9]. The most visible application of TCA is the Xavier robot that wanders the halls of CMU fulfilling tasks sent over the Internet (http://www.cs.cmu.edu/~Xavier/).

In a project reported by Aylett, et al. [2], plans generated by the planner were executed directly by activating the behaviors on two mobile robots. They found that in order for the architecture to work, the planner must consider execution effects. Our architecture differs from this approach in that plan execution is continuously monitored, and if the plan execution deviates from expectation, a new plan is generated to achieve the goal from the current state. The tradeoff was made in favor of replanning over exploring contingent plans.

Another approach to robot architectures has been to use a symbolic planner in a hierarchical method, abstracting the robot task into three levels [11]. In this architecture, planners were used at varying levels of abstraction. Each planner generates a plan at its level of abstraction that is passed to the next lower-level planner to refine. The lowest-level planner generates a plan in which the actions represent the robot behaviors themselves. These actions are then executed. This system essentially uses a planner in place of a sequencer. This approach has the benefit in that while switching or adding tasks, the operator does not have to add and change multiple sequences. The only changes necessary are to the planning domains. We adopt this approach with the exception that instead of a symbolic planner we use DT-Graphplan to generate the high-level plans.

3. THE ROBOT ARCHITECTURE

The Decision-Theoretic Robot Controller (DTRC) consists of DT-Graphplan as the high-level reasoning system, a set of robot skills as the low behavioral level, and an execution monitor which passes information between the two. These three elements provide a robust method for handling uncertain robotic environments. This section discusses the execution monitor, the skills layer, and how these two elements interact with DT-Graphplan.

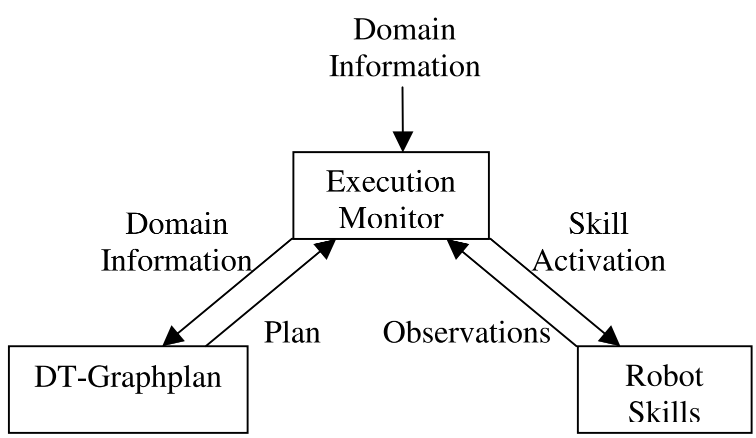


Figure 1. The DTRC Robot Architecture.

Figure 1 shows an overview of the robot architecture. The user supplies the domain information to the execution monitor, which consists of the robot actions, the robot goals, the reward conditions, a state representation, and the utility threshold for the domain. The execution monitor maintains current state information, requests a plan from DT-Graphplan, and notifies the skill layer of which skill to activate next as specified by the generated plan. The execution monitor maintains the state information in a Bayes network. A Bayes network is a collection of propositions connected as a directed graph. For each connected pair of propositions, one proposition influences the other based on the graph connection. The proposition being influenced is the child; the influencing proposition is the parent. Each proposition has a conditional probability table that represents the parent proposition's effect on the child. The state is updated with the action outcomes and observations of the state made by the robot skills. These updates are entered into the Bayes network as findings or evidence.

3.1. Decision-Theoretic Graphplan (DT-Graphplan)

The concept behind DT-Graphplan is similar to that of Graphplan [4]. In particular, DT-Graphplan creates a graph containing all possible combinations of actions that reach a goal condition from an initial condition. Once the graph is generated, it is searched in a backward-chaining fashion for a complete plan. DT-Graphplan extends Graphplan and allows for probabilistic propositions and stochastic actions, reasoning about costs and rewards. DT-Graphplan's objective is to locate a plan that meets a time-independent threshold on expected utility.

3.1.1. The Graphplan Planning Algorithm

The Graphplan algorithm, written by Blum and Furst, performs symbolic planning, reducing the search space by performing search from both the initial and the goal conditions [4]. The Graphplan algorithm plans by alternately expanding a planning graph and extracting a plan solution. The plan graph is a series of layers alternating between proposition nodes and action nodes. The first layer consists of proposition nodes that represent the initial plan condition. For each action node, directed edges lead from the proposition nodes that represent the preconditions of the action to the action node, and then from the action node to the proposition nodes that represent the effects of the action.

During graph building, the graph retains binary mutual exclusion information (referred to as *mutex relations*). Mutex information speeds search by tracking the propositions that interfere with each other and cannot exist simultaneously (in the same state), thus reducing the search space by ignoring impossible states. The mutex relation also preserves state information: two propositions that are mutex cannot exist simultaneously. Two action instances at a level are mutex if they interfere. An interference exists if one action deletes a precondition or effect of another, or if the actions demonstrate competing needs – the actions contain preconditions that are mutually exclusive at the previous level. Two propositions at a level are mutex if all ways of achieving the propositions (actions generating the propositions) are mutex.

Graph expansion halts on a proposition layer when each element of the goal condition is present and none are pairwise mutex. Graphplan then searches the plan graph for a plan solution using a backward chaining search. The search results in a path from the set of goals to the initial condition consisting of only non-mutex actions. If the search finds no such plan, then an additional layer is added to the plan graph searched in an iterative fashion.

Because the Graphplan algorithm performs both forward and backward search, the number of plans that must be examined is reduced. The forward search phase occurs during graph building, in which only those states reachable from the initial condition are added to the graph. The backward search phase is a backward-chaining search of the planning graph. The reduction in the size of the search space motivated the selection of Graphplan as the basis for the decision-theoretic planner used in this work.

The Graphplan algorithm has been extended into a probabilistic planner, PGraphplan planner [5] which produces a contingent probabilistic plan like C-Buridan [10]. PGraphplan extends the Graphplan algorithm. PGraphplan's methodology differs from Graphplan in that instead of searching in a backward-chaining manner, search is a forward-chaining process to find an optimal contingent plan. PGraphplan uses the forward-chaining search phase to both propagate probabilities and locate a plan.

Propositional reasoning offers savings over state space reasoning as a few propositions can be used to represent a set of states instead of explicitly representing all possible states. As a result, propositions only need to be included in the graph when their value is changed or they represent a precondition of one of the actions. They can be excluded from all other situations. By limiting the search space to only the affected propositions, the space that must be searched is reduced, and the memory required is reduced. The graph used in DT-Graphplan only includes elements that play a role in the agent's task, and so states with unnecessary elements are excluded in favor of states that lead to task completion.

Additionally, to counteract the exponential increase in overhead encountered by adding decision-theoretic reasoning, DT-Graphplan prunes non-promising portions of the plan space. Propositions associated with low expected utility states are tagged as non-promising and are not expanded at the following step or graph level. If later it becomes apparent the pruned space may be important, the tagged propositions may be revisited.

In DT-Graphplan, a situation is characterized as a probability distribution over possible states. An action causes a transition from one situation to another. A plan is a sequence of actions ending in at least one state in which the goal condition is true.

3.1.2. Definitions and Problem Description

This section discusses the definition of the planning problem solved by DT-Graphplan. Here we define states, actions, rewards, problems, and problem solutions.

3.1.2.1. State

DT-Graphplan assumes a finite state space $S = \{s_1, s_2, \ldots s_n\}$, in which a state is described using a set of propositions. However, the planner does not include complete information about the world and so each state has an associated probability in the plan space. The probability of the agent being in a state is represented by associating a probability with each proposition that comprises the state description. The propositions are assumed to be independent unless specified by the user. Let n represent the number of propositions that describe the world. The probability of a state is:

$$P[s] = \prod_{i=1}^{n} P(p_i \in s \mid p_1 \in s, p_2 \in s, \ldots, p_{i-1} \in s, p_{i+1} \in s, \ldots, p_n \in s) \quad (1)$$

Proposition dependence information is initially input by the user in the form of a Bayes Network. Throughout planning, the Bayes Network updates each proposition's probability that can be extracted by the planner to provide the values shown in equation 3.1. For example, on a miniature golf course, the robot's success at putting the ball into the cup depends on whether the course slopes toward or away from the hole. Instead of assuming that propositions are independent, or that the propositions do not affect each other, which means in this case that the slope of the course does not affect the robot's success, DT-Graphplan queries the Bayes Network to obtain information about proposition interdependence. Using this method, if the propositions are dependent and represented in the network, then the probability of the state depends on the associations between propositions and the network evaluation. Details of the network construction and usage are provided in section 3.1.3.

3.1.2.2. Actions

The current state changes based on the occurrence of events. One type of event is an action, an event caused by the agent. DT-Graphplan does not handle observation actions nor exogenous events. At each state, it is possible to apply the feasible set of actions, which is the set of all actions applicable at a state. DT-Graphplan limits this selection by only applying actions possible based on the action's preconditions

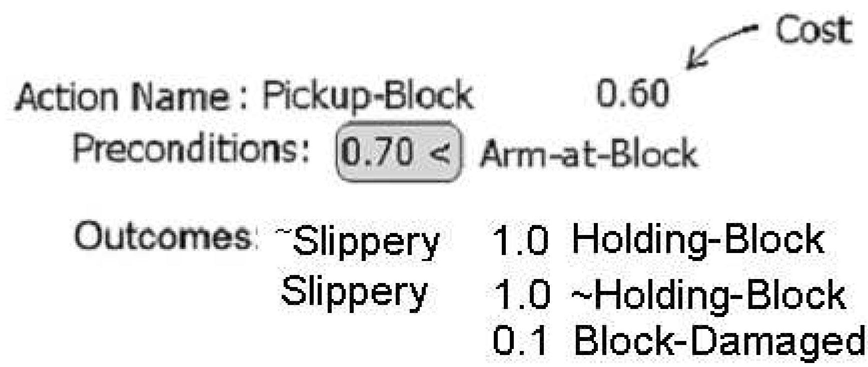


Figure 2. Example action.

An example action 'Pickup-Block' is shown in figure 2. An action represents a transition from one set of states to another set of states. An action consists of a set of outcomes and a set of preconditions. Associated with each outcome are a number of propositions called the effects. There may only be one outcome, in which case there is one distribution over effects that is applied each time the action is executed. The reason for using both preconditions and conditions is that through the use of preconditions, if an action has only one conditional effect then the states in which the action can execute are reduced, lessening the work the planner must perform.

Actions in DT-Graphplan consist of a set of preconditions ('Arm-at-Block') that must be met for the action to be applied, and a set of outcomes. Each outcome contains a conditional ('Slippery', '~Slippery'), and associated probability distribution over effects ('~Holding-Block' 1.0, 'Block-Damaged' 0.10). The action description of DT-Graphplan is assumed to be non-destructive unless specified otherwise by the user. By assuming the actions are non-destructive, the planner saves the user from adding the condition to each action that if an effect exists prior to action execution then it will exist after the action executes.

The grayed area in the precondition of the “Pickup-Block” action represents a precondition test. The precondition propositions (in this example, Arm-at-Block) must meet the threshold probability for the action to be executed. The precondition test reduces the number of states in which the action can execute, in effect pruning plans with little advantage. The precondition test can be set to 0.0, in which case, the action is applied to all states in which the preconditions hold.

The evidence of an action is the product of the probabilities of the preconditions of the actions. This product represents the probability of the situation the action is applicable to. This evidence, the current propositions preconditions, and the probability of the outcome dictates the probability of new effect propositions.

The ‘Pickup-Block’ action has two outcomes dependent on the ‘Slippery’-ness of the arm gripper. The condition of the first outcome is for a non-slippery gripper that has one effect, ‘Holding-Block’, with a corresponding probability of 1.0. The condition for the second outcome for a slippery gripper includes a ‘Holding-Block’ effect with a probability of 0.0, as well as a ‘Block-Damaged’ effect with probability of 0.1. If the block is not held, the block may be damaged from a fall when executing the action. Each action has an associated cost. For the ‘Pickup-Block’ action, the cost of the action is 0.60. The cost is administered as a negative reward and is applied to the state resulting from the action execution.

3.1.2.3. Rewards

A reward is an assignment of a real number to a state, representing the desirability of the state. In DT-Graphplan, reward functions are defined based on the reward functions specified by users for specific propositions. DT-Graphplan assumes reward independence; the reward for one proposition has no effect on the rewards for other propositions. The reward for a state may thus be calculated as the sum of the rewards for the individual propositions comprising the state.

$$R(s) = \sum_{p_i \in s} R(p_i) \tag{2}$$

The rewards in DT-Graphplan are time-separable. The reward received is additive and the reward does not depend on the time at which it is received. The reward function of DT-Graphplan makes use of a shifting reward scale. Once a proposition exists, reward for maintaining that proposition is 0. For example, if the robot received a reward for holding a block, it would not get additional reward for holding the block at the next plan step or dropping it and picking it up again.

3.1.3. The DT-Graphplan Algorithm

The DT-Graphplan algorithm functions in two phases: plan graph expansion and backward-chaining search. Planning alternates between the two phases until a plan is found or DT-Graphplan determines that no plan can be generated.

DT-Graphplan is sent a collection of propositions with probabilities representing the initial condition, a set of propositions representing the goal conditions, the expected utility threshold, a set of reward propositions with reward amounts, and a Bayes network that represents any needed dependence between propositions of the state. The Bayes network also includes any queries that dictate how to propagate the dependence information.

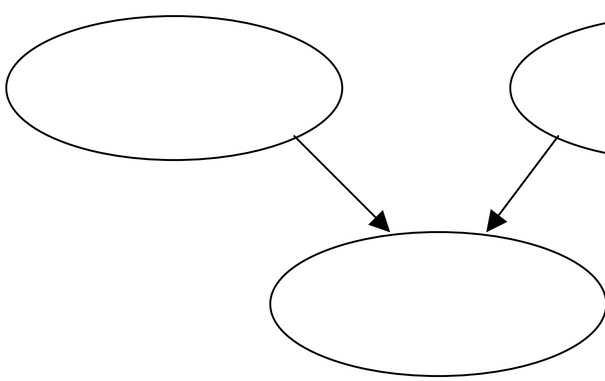
DT-Graphplan begins by generating a single plan graph. Plan graph expansion continues until the goal conditions are present and the expected utility of the state they represent exceeds the expected utility threshold. At this point, all acceptable sets of propositions representing states that meet the goal condition (goal lists) are collected from the final level of the graph. These goal lists are then searched to determine if they can be reached from the initial condition.

The search phase of DT-Graphplan differs from Graphplan only in the action selection. In Graphplan, the last action that adds a goal proposition is the first action that is searched. In DT-Graphplan, the algorithm chooses the action to search based on the expected utility prior to the action, selecting the highest expected utility first. Searching the plan graph for a connection from the goal condition to the initial condition.

If one of the goal lists can be reached from the initial condition, then a plan is found and the algorithm ends. If none of the goal lists lead to a complete plan, then the algorithm tests to make sure that a plan may exist. This test is performed by comparing the last two graph levels. If the last two levels are the same, then no plan exists. If the last two levels are not identical then the algorithm continues.

The algorithm then adds an additional action and proposition layer, effectively extending the graph one step deeper. Once the new layer is added, the algorithm returns to generating and searching the goal lists.

DT-Graphplan assumes that all propositions are independent. If a domain requires propositions to be dependent, the dependencies must be included in a Bayes network representing the state. The Bayes network represents the state in conjunction with a set of user-defined queries informs the planner about node dependencies, about which nodes represent evidence, and about which nodes should be queried. For example, in the “bomb in the toilet” domain there is a bomb in one of two packages, and the agent must defuse the bomb. There is a 50 percent chance that package 1 is unsafe and package 2 is safe, and a 50 percent chance of the opposite. Each package’s condition depends upon the other, thus the propositions are not independent.



Bomb in		
p1	p2	P(D)
T	T	1.00
T	F	0.50
F	T	0.50
F	F	0.00

Figure 3. Bomb and Toilet Network.

To reason about the possible states inherent in this domain, the input to DT-Graphplan consists of the probability that each of the packages is safe, in this case 0.5 (or 50 percent), and the probability that the bomb is defused, which here is 0.0. After one of the packages is dunked in the toilet, the probability that the bomb is defused becomes 50 percent. In order for DT-Graphplan to correctly calculate this probability, a Bayes network representing the state is included in the domain containing the information that the 'Defused' proposition is dependent on the state of both of the packages, as shown in figure 3.

DT-Graphplan makes use of the Netica API to maintain the Bayes network, enter findings, and run queries. Before initiating planning, the input file of the Bayes network is converted into a Netica network by sending each node and probability table to Netica.

3.1.4. Plan Graph Expansion

Plan graph expansion occurs as in Graphplan, adding all of the actions possible given the propositions that hold at the current plan step. DT-Graphplan associates a probability value and reward functions with each of the propositions in the graph. The probability value represents the probability that the proposition is currently true. One minus this value represents the probability the proposition is currently false. The reward function is the rewards earned and costs incurred for this proposition. This function also carries rewards and costs prior to reaching the proposition, for calculating the expected utility of the plan. The expected utility of a plan is a combination of the proposition's reward functions for the propositions in the end state.

During graph expansion, action effects lead to additional, possibly new propositions. Each action inserted into the graph generates new propositions that are added to the graph at the next plan step. In addition, 'noops' carry existing proposition from the current plan step to the following step. As a result, multiple occurrences of a proposition occur at the same level in the graph, with different probabilities and reward functions, resulting from different action sequences. The graph generated by DT-Graphplan represents all possible action combinations and resulting propositions.

Because of the possible multiple proposition occurrences, an additional mutex rule is applied to maintain state information and speed search. The additional mutex rule specifies that each of the propositions with the same name but different probability or reward functions are mutex with each other. DT-Graphplan also retains the existing action mutex relations of interference of effects and competing needs from the original Graphplan algorithm, and the proposition mutex relation from Graphplan, whereby two propositions are mutex if all ways of reaching the propositions are mutex.

Once all actions are applied to the graph and 'noops' are added, Bayes network queries are executed. The query node probability calculation is performed by the Netica API. The probability of the evidence propositions are entered into the Bayes network representing the state as findings, one at a time. The Bayes network is then queried for the query proposition, and the returned value is used as the probability of the new proposition inserted into the plan graph at the next step. For example, if the probability of 'Bomb in p1' is 0.10 and of 'Bomb in p2' is 0.40, then the probability of 'Defused' returned from Netica is 0.75.

3.1.5. Calculating Expected Utilities

The probability calculation for an action's effects depends on a conjunction of the precondition probabilities (evidence), the action's conditions, and the conditional effects. The evidence calculation assumes the precondition propositions are independent and computes the product of the probabilities over all corresponding precondition propositions. For preconditions that are dependent on each other, the dependent propositions must be represented by an additional proposition that represents the joint effect of the two propositions on the state. This additional proposition is used as the action precondition.

An effect's probability is the evidence of the action multiplied by the conditional probability of the action. The evidence of the action is the product of the probabilities of the preconditions. The conditional probability of the action for an effect is the probability of the sum over all of the conditions that possess the effect of the probability of the effect given the condition. For example, the 'Pickup Block' action from figure 2 has an condition for '~Slippery' that the action results in the positive effect of the robot holding the block with probability of 1.0. If the probability of 'Slippery' is 0.20, then the probability of 'Holding-Block' for the condition is (1.0-0.20)*1.0= 0.80. This is added to the condition for 'Slippery', which is 0.2*0.0= 0.0. The resulting 80 percent probability is multiplied with the precondition probabilities to generate the probability that the robot is holding the block.

The rewards earned during planning are stored in a reward list, associated with each proposition. Each reward in the reward list consists of the reward name, and the reward amount. When adding effects of the actions, the reward lists are compared, combining only those that exist once. This combined reward list is passed on to all of the effect propositions and represents all of the reward earned in the plan prior to applying the current action. If the current proposition meets a reward condition, this reward is added to the reward list for the proposition.

Costs in DT-Graphplan are represented as negative rewards. The costs of the actions performed during planning are stored in a cost list, associated with each proposition.

After generating the reward and cost lists, and calculating the probability value for an action effect, the effect proposition is added to the graph. For each effect, the effect is checked to see if a reward is earned, and if so the reward is added to the reward list, and the cost of the current action is added to the cost list. If a version of the proposition does not exist with the new probability value for this action effect, graph expansion inserts a proposition with the new probability that is mutex with the other instantiations of the same proposition.

At any point the expected utility of a plan can be computed. Any set of propositions that are non-mutex can potentially represent an end situation of a plan. The expected utility of the situation is calculated by merging the reward lists and cost list the same as for action preconditions. The utility of the situation is the sum of the reward amount of all the rewards in the generated reward list and the cost amount for each cost in the generated cost list. The utility of the situation times the probability of the situation results in the expected utility of the plan. It is also possible to perform the same calculation with a single proposition, getting the effect the proposition has on the expected utility of a plan.

3.1.6. Pruning

DT-Graphplan restricts the creation of an exponential number of world states by reasoning about individual propositions. However, the propagation of proposition information, reward lists, and cost lists, increases the computation time at each step. To reduce this burden, DT-Graphplan prunes the plan graph during construction. A proposition is pruned based on its expected utility value. If the expected utility of the proposition falls below a user-defined threshold, then the proposition is not expanded during graph generation.

At each plan step, the current minimum and maximum expected utility values are determined for each set of propositions at a graph level with the same name. The pruning threshold is calculated based on the minimum and maximum expected utility values for the set, and a user-specified percentage.

$$prune\_threshold = node\_utility_{\max} + (node\_utility_{\max} - node\_utility_{\min}) * percentage \quad (3)$$

If the expected utility value of a proposition falls below the pruning threshold, then it is not expanded during the following iteration. For example, if the best expected utility for proposition X generated at this level of the graph is 10 and the lowest is 2, and the user wants to retain the best 20 percent of propositions, then the pruning threshold would be (10 - ((10 - 2) * 0.20)) = 8.4. During graph expansion for the next plan step, each proposition X with an expected utility below 8.4 is marked and is not expanded.

3.2. The Execution Monitor

The execution monitor maintains state information, interprets sensor input, and triggers actions at the other layers of the architecture (such as initiating a robot skill or generating a new plan). The execution monitor interprets the current state and plan, and decides whether to continue the plan or request a new plan.

On startup, the execution monitor receives the initial state, a Bayes network describing dependencies between all of the environment variables, and the action set. The execution monitor also receives a set of reward conditions for the robot task. The execution monitor passes the current state, the Bayes network describing the state, the reward conditions, the action descriptions, and the utility threshold to the DT-Graphplan planner. Once DT-Graphplan returns a plan, the execution monitor activates the correct skill for that plan step. The skill will return a value to the execution monitor stating whether it has succeeded or failed, along with any observation information gained by the skill. This and the remainder of the Execution Monitor main loop is shown in figure 4.

The cycle the execution manager performs is to execute a plan step, add the action effect information to the state, receive observation information from the robot, and add this information to the state network, and then check to make sure the current plan is still valid. If the state the robot is in does not match the state need for the next step or there is a change in the goal conditions, replanning is performed. If replanning fails, assistance is requested. If still no plan is found, the Execution Monitor aborts the task.

```
Function : Execution Monitor
Return Value : BOOLEAN
Input :
  fact_list goal_facts
  bayes_list state_nodes
Variables :
  BOOLEAN plan_failed = done = FALSE
  int time_step = 0
  vertex_list plan_step = NULL

Begin
  found_plan = dtgp( goal_facts, time )
  InitialiseState( state_nodes )

  while( !done )
     If the planner found no plan from the initial condition notify the user and end.
    if( !found_plan )
      print "I found no initial plan! HELP!"
      return (FALSE)
     Execute the first step in the plan. If there is a drastic hardware failure, as in no
     communication with the robot, this function returns TRUE.
    done = ExecutePlanStep( time_step, time, plan_step )
    Update the Bayes network representing the state with the outcomes of the action.
    UpdateStateAction( state_nodes, plan_step, time )
    Check for drastic hardware failure.
    if ( done )
      print "ERROR! Plan step failure, most likely hardware error. ABORTING."
      end.
    Have the robot observe the current state
    Observe( state_nodes )
    Now add the observations to the state
    SendStateFindings( state_nodes )

    Now verify the next action with the current state, make sure that the preconditions exist.
    AND check  to see if the user added any additional goals, if additional goals OR the
    preconditions of the next  action do not match the current state, return TRUE.
    plan_failed = VerifyPlanWithState( time_step, time, goal_facts, done )
    if( plan_failed AND !done )
      Failed to match next action. REPLANNING.
      GetInitialConditionFromState( state_nodes )
      found_plan = dtgp( goal_facts, time )
      If no plan was located ask for operator assistance, in our case, the robot and elements of
      the  environment  are adjusted and a new observation is made to plan from.
      if ( !found_plan )
        Did not locate feasible plan, request operator assistance
      else
        time_step = 0
end while
  return (TRUE)
end.
```

Figure 4. The Execution Monitor main loop.

To maintain an accurate probability distribution over variables representing the current state, the execution monitor makes use of a Bayes network to represent the state. This same network is supplied as an input to DT-Graphplan. DTRC makes use of the Netica API to generate the Bayes network and enter findings and run queries. Before initiating planning, the input file of the Bayes network is converted into a Netica network by sending each node and probability table to the Netica API.

During execution of a plan step, the execution monitor updates the Bayes network representing the current state using the outcome probabilities of the plan step. This is accomplished by entering findings into the Bayes network representing the state. Each set of findings are sent to Netica to be applied to the network representing the state. For each outcome of an action, corresponding propositions have an associated probability, representing the probability that the proposition is observed, and the outcome of the action is applied to the current state as a finding. In order to reduce the memory of the Bayes network in regards to previous steps, during plan execution the findings decay. As the execution monitor is executing an action, the effects of previous findings on the state decay. The findings in DTRC decay by 0.50, we selected this decay rate because the state changes after each action, propositions in the domain shift drastically from being true to not true. If the findings did not decay, findings from previous steps would limit the importance of the latest action or observation finding on the state. It is this most recent information the robot needs more than information a number of steps prior.

After a step is executed, the execution monitor takes the observation set from the skills and applies the changes to the state. This observation also includes any additional reward conditions added during the step's execution. In this manner, a probability distribution representing the current state is always available. This maintains a representation of the current state for an initial condition if replanning is required, and verifies that the plan is still valid by comparing the state with the preconditions for the next action.

If an observation made during execution shows that a reward condition is met, the reward is removed from the list of reward conditions. The exception to this update occurs when the reward condition represents maintenance goals, which are regenerated by the execution monitor. Once a maintenance goal is fulfilled, the execution monitor adds the goal back into the domain description for the planner.

Replanning occurs when the current plan does not match the current state and the plan cannot be continued, or a new reward condition is added to the domain. Replanning occurs in the same manner as the initial plan generation, with the exception that instead of using the initial state information, the current state information from the Bayes network is used.

Skill failures occur for many reasons and in any case in which the current state differs from the preconditions for the next plan step, or when a reward is added, replanning occurs. Replanning occurs when a new reward condition is added to the domain, because the new reward condition may be important enough to stop what the robot is doing now and plan to receive the reward, or it may be possible to fulfill elements of the new reward condition while fulfilling current reward or goal conditions.

3.3. Robot Skills

The robot skills send commands to the actual robot hardware and integrate the motors and sensors of the robot in use. The robot skills are written to perform simple tasks well, and combine several small skills together in order to generate more complex skills. In robotics, skills handle situations as simple as reading a single sensor and as complex as moving across an inhabited room. Skills succeed because they are inherently simple, being close to a knee-jerk type of reaction to the world. Skills that perform more advanced functions such as path planning or mobile robot navigation either use multiple smaller skills cooperating together or a decision-theoretic method [10,31]. These methods do not provide an ability to assemble complex tasks into sequences in order to complete a number of potentially-competing goals that a planner provides.

The use of skills for the lowest level of a robot architecture allows the robot to robustly complete simple tasks. When skills are turned on and off according to a goal-oriented plan, the outcome is a robot controller that completes complex tasks.

3.4. Integration

Integration of the DTRC components is illustrated using the mail/coffee robot delivery domain [7]. The mail/coffee delivery robot domain features a robot that must obtain and deliver mail to the user, satisfy coffee requests, and keep the lab tidy. The domain has six propositions: location of the robot, lab tidiness level, presence of mail, presence of a coffee request, robot holding mail, and robot holding coffee. The position of the robot can be one of five possible locations: mailroom (M), coffee room (C), user's office (O), hallway (H), or laboratory (L.) The tidiness level of the lab has five possible values: 0 (messiest) to 4 (tidiest). The five locations are arranged in a loop, clockwise from the mailroom are the coffee room, laboratory, user's office, and hallway. The result is the robot has two location changing actions of moving clockwise and counterclockwise, to navigate around the loop.

The robot has eight actions it can execute. The actions include move clockwise (Clk), move counterclockwise (CClk), tidy lab (TL), pickup mail (PUM), get coffee (PUC), deliver coffee (DelC), deliver mail(DelM), and wait (Wait). All include, as part of their effect representation, the transition model for the exogenous events of mail arrival and coffee request. These exogenous events occur at any state with a probability of 0.20. The move-clockwise and move-counterclockwise operators result in a probability of 0.90 of moving the robot to the next location and a probability of 0.10 of not moving the robot at all. The cost of all actions, with the exception of the wait operator, is 1.0. The deliver-mail and pickup-mail actions are certain to deliver mail to the office if held, and to obtain mail from the mailroom if present. The deliver-coffee action can be performed anytime the robot has coffee. If it is performed in the office, it has a 5% chance of spilling the coffee trying to deliver it, and a 95% chance of delivering it successfully. If the robot attempts to deliver the coffee in any other location, there is a 70% chance the coffee is taken from the robot but not by the intended person, resulting un an unfulfilled request because the wrong person would have taken the coffee. The tidy action has no uncertainty and improves the lab tidiness by one.

There are three exogenous events: mail arrival, coffee request, and untidy the lab. The exogenous events are modeled with an implicit-event model. An implicit-event model represents events beyond the robots control as part of the effects of the robots actions. An example of an exogenous event is that there is a 20% chance that mail arrives or a coffee request is submitted at a given time step. Because DT-Graphplan is non-observant, the actual event is not sensed or observed by the planner.

For this example, the goal of the robot is to not have any mail and not have any mail in the mailroom. This is a maintenance goal, and one that may not be initially satisfiable. The reward for not holding mail and no mail being present in the mailroom is 10.0. This is the only reward, so the largest possible expected utility for any plan is 10.0.

For this domain and reward function, and an initial condition that consists of the robot being in the mailroom, the plan that DT-Graphplan returns is to do nothing, or an empty plan. This is because at the initial condition there is no mail in the mailroom. The execution monitor, having no plan, simply waits. After a period of time, the skills return an observation. If the observation shows that there is no mail, then the current plan is still valid and the robot waits again. If the observation shows that there is mail in the mailroom, the plan is not valid. An exogenous event occurred which changed the state. Since the new state does not match the state of the plan from DT-Graphplan, the execution monitor requests a new plan.

The execution monitor generates a new initial state for the planner, one that includes the information that the mail is in the mailroom. Since the reward conditions have not changed, the utility threshold or maximum possible utility is 10.0. The plan that is generated is to pick up the mail and go to the office and deliver the mail, a three-step plan.

The execution monitor activates the skill to pick-up the mail, and updates the state to represent the possible conditions after picking up the mail. Once the skill finishes, the observations that the robot is holding the mail and that there is no mail in the mailroom are made. These observations are added to the state description.

The execution monitor then activates the move-to-office command. During execution, the execution monitor again updates the probabilities for success as well as the probability that mail has been delivered. After the skill finishes, since the robot is no longer in the mailroom and cannot sense if there is mail or not, the probability of mail remains the same, until it decays following the next action.

If during execution, the user makes a coffee request, a new reward condition is added to the reward set. Because of this change, the execution monitor requests a new plan. Ideally, the new plan could be generated while the robot is completing its current plan, but this is not implemented in the current version of the architecture.

The reward for filling a coffee request or for delivering coffee when requested is 50.0. This reward has the effect that the maximum possible expected utility is now 10.0+50.0=60.0. The reward also emphasizes that coffee is more important than mail. The plan generated by DT-Graphplan directs the robot to get the coffee and then deliver both the coffee and the mail.

After completing both tasks, the coffee request reward condition, being a single execution reward, is removed from the reward set. Also, the robot is now in the office, and since it has been away from the mailroom for a while there is a high probability that there is now mail in the mailroom. Since the reward condition for not having mail in the mailroom still holds, and the other reward has dropped off, a new plan is generated. Given the state that there is a high probability of mail in the mailroom, the plan is to go to the mailroom and obtain and deliver the mail.

Once the robot reaches the mailroom, if mail is present, it already has a plan to obtain and deliver it and so just continues. However, it may find that there is no mail. In this case, the robot aborts the plan and generates a new plan. This basically returns the robot to the initial waiting condition.

Using the DTRC architecture described in this paper, a robot can generate and execute a plan for an uncertain domain. This is shown in the following section discussing the implementation of robot miniature golf.

4. ROBOT MINIATURE GOLF

Integration of the systems into the DTRC architecture is illustrated using the application of a real robot on the robot miniature golf domain. Through this example we provide details of the DTRC implementation, component interaction, and handling of plan failures. The domain does not demonstrate maintenance goals, or replanning based on receiving new goals while executing a plan.

For the robot miniature golf domain, the robot used is an Activmedia Pioneer 1 with a gripper and camera. A neon orange tennis ball takes the place of a golf ball, and a two-foot circular foam disc with a neon green tennis ball on a pole six inches above the center of the hole represents the flag and the cup. The course covers a 12’x12’ area. Three obstacles are included that affect the ball trajectory with a probability less than 1.0. The three obstacles are ‘The Funoodle Surprise’, ‘The Land of the Legobots’, and a small ridge. ‘The Funoodle Surprise’ is a five-foot foam cylinder suspended over the golf course that swings back and forth, sometimes knocking the ball off course and into a neighboring region. ‘The Land of the Legobots’ is a six-foot square region of the course in which a Lego robot wanders hoping to strike and knock the ball off course. The small ridge is a impediment between two regions of the golf course that can prevent the ball from making it to the next region.

Uncertainty is present in the outcome of all the actions to move the ball. This uncertainty stems from the robot not being certain about its location, the ball’s location, motor failings, and the obstacles.

Figure 5 shows the layout of the robot golf course. There are a number of paths along which the robot can navigate and move the ball. The space is divided into these paths to facilitate reasoning at a high level and to avoid performing extensive navigation computations. This breaks the domain into eleven regions with off-the-course being one region. From each region, the robot can move the ball or itself in one of four directions: north, south, east, or west. If there is a solid obstacle in the way or the path would take the robot off the course, the corresponding action is not included in the domain. If the ball happens to roll off the course, the robot will locate and retrieve the ball, placing it in the center of the closest region if possible.

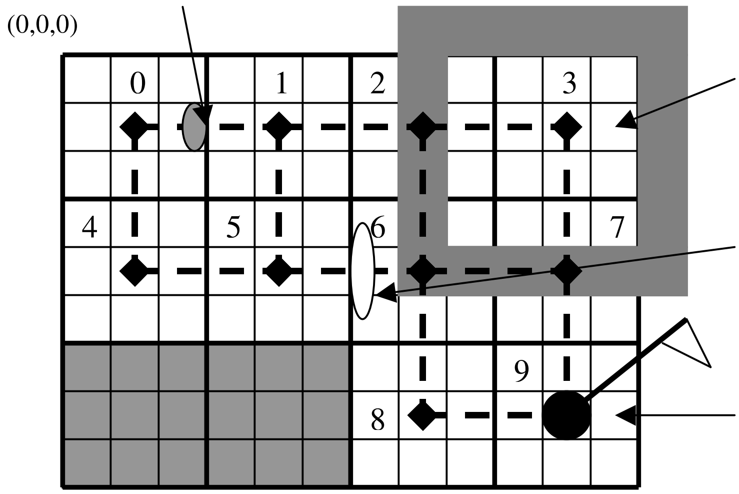


Figure 5. The Robot Miniature Golf Domain.

The actions available for the robot are to push the ball from one region to a neighboring region, pick up and carry the ball to a neighboring region, strike the ball to a neighboring region, attempt to putt the ball into the cup, and pickup and place the ball in the cup. Striking the ball has a cost of 0.50, pushing the ball has a cost of 1.00, and the pick-up-and-move action has a cost of 2.50. The only reward condition is for the robot to get the ball into the hole, and the reward amount is 20.0. There are 64 actions to move the ball from region to region, and into the cup.

The probability distributions for the actions were calculated by executing the push, strike, and carry actions ten times each and generating the probability distributions for the average cases. The probability distributions for the cases that go over the 'Small Ridge' or around 'The Land of the Legobots', 'The Funoodle Surprise', and the into the cup, were generated in the same way.

The values of the rewards and action costs were generated by setting preferences for the actions. We wanted the robot to incur less cost on strike than pushing, and pushing than carrying the ball. Using these, we set a cost of 0.50 for the strike action and just multiplied this by two and three for the other two actions. The reward value was calculated based on the action cost, and the expected length of the plan. The reward needed to be more that the cost of the plan to provide enough incentive for the robot to try and get the ball in the cup.

4.1. Robot Miniature Golf Skills

The Pioneer 1 robot that plays miniature golf has a single camera that can be trained to recognize objects of three different colors. Once trained, the camera will return the center of a blob in image coordinates of the trained color in an image. If the color is not present in the image, the coordinate of 0,0 is returned. Since the camera does not have pan, tilt, or zoom capabilities, we experimentally derived a function to translate the image coordinates returned into a relative position of the ball from the robot. The same procedure was performed for the cup, because the neon green tennis ball representing the flag over the cup was not on the floor, but six inches above it. The function relied on the angle of the camera to the floor, and would not work near the edges of the camera lens because of the lens distortion.

There are five high-level skills. They are carry, strike, push, putt_to_cup, and put_ball_in_cup. These five high-level skills represent actions that are referenced by the execution monitor and DT-Graphplan. Each skill is made up of multiple smaller skills that assist in completing the task. There can be as many skills and layers of skills as required to complete a task.

The carry skill locates the ball, picks it up, moves with the ball to a specific region and then drops the ball. The strike skill is similar: locate the ball, the robot then calculates where the robot should be in relation with the ball to hit the ball to the destination and then the robot moves there (position_robot_on_ball), hit the ball, and move to the next region. Push differs from strike by having a low-level push_the_ball skill that just pushes the ball slowly instead of hitting it. If the robot cannot locate the ball, then the robot will perform a square search pattern for the ball.

All of the high level skills follow the same basic pattern for triggering lower skills. This pattern is to locate_ball, calculate the balls position in the universe (ball_in_universe), move to strike/push/carry/putt the ball (position_robot_on_cup), then perform the action that differentiates the skills (strike/push/carry/putt), and then move_to_next_region. To assist, there are skills to find_the_ball, locate the cup (locate_cup), calculate destinations (calculate_distance_from_ball), and angles (turn_angle).

In the locate_ball skill, the robot scans for the ball with the camera and determines the center of the ball in the image, then translates the x, y image coordinates into physical distance in millimeters from the robot. The locate_cup skill is identical except instead of calculating the position of the ball it calculates the position of the cup. The difference, as outlined earlier, is necessary because the ball representing the cup is six inches off the floor.

The ball_in_universe skill translates the position of the ball from x, y distance in relation to the robot, to x, y position in relationship to the origin of the golf course, which is in the upper right of region 0. The robot's position and orientation are represented in universe coordinates, so the only needed calculation is to rotate the ball's position into the universe's reference frame. This is shown in the following formula, with B_X , B_Y representing the ball's position in relation to the robot, R_θ the robot's angle, and U_X, U_Y, and U_θ the ball's position rotated into the universe frame. U_θ will always be 0.0 because the ball has no rotation angle. The next step is to move the ball from being centered on the robot to being centered at the universe origin. This is done by adding the robot's x, y position to the new ball's x, y position. This results in the ball's position with respect to the origin of the golf course.

$$\begin{bmatrix} U_X \\ U_Y \\ U_\theta \end{bmatrix} = \begin{bmatrix} \cos R_\theta & -\sin R_\theta & 0 \\ \sin R_\theta & \cos R_\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_X \\ B_Y \\ 0 \end{bmatrix} + \begin{bmatrix} R_X \\ R_Y \\ 0 \end{bmatrix} \tag{4}$$

The calculate_distance_from_ball skill uses the ball's position in the universe (U_X, U_Y), and a destination position (D_X, D_Y), in the universe to calculate a position and angle from the ball to locate the robot. The destination position is either the center of a region or the location of the cup. The points of the ball and destination represent a line in the universe; the calculate_distance_from_ball function calculates a point on this line 500 millimeters away from the ball on the side of the line away from the destination. The skill also calculates the angle of the line from the ball to the destination. The equations to calculate the destination position and angle are shown:

$$x = B_x - \left(\frac{(D_X - U_X)*500}{\sqrt{(D_X - U_X)^2 + (D_Y - U_Y)^2}} \right) \tag{5}$$

$$y = B_Y - \left(\frac{(D_Y - U_Y)*500}{\sqrt{(D_X - U_X)^2 + (D_Y - U_Y)^2}} \right) \tag{6}$$

$$\theta = \sin^{-1} \left(\frac{(D_Y - U_Y)}{\sqrt{(D_X - U_X)^2 + (D_Y - U_Y)^2}} \right) \tag{7}$$

Skill position_robot_on_ball uses the calculation from calculate_distance_from_ball as a destination position for the robot. The skill receives the location and moves the robot to the position and angle calculated. This sets the robot in a position to move the ball to the destination, because the robot is facing the ball and the destination.

The skill move_to_next_region has the robot follow the ball to the next region. If the ball made the trip to the next region, the robot will be facing the ball. If at any time the robot loses sight of the ball, the robot will attempt to find_the_ball. The find_the_ball skill has the robot perform a square meter search around where the ball should be, turning 360 degrees at each of the corners of the square to search. If the robot locates the ball outside of the bounds of the golf course, it will retrieve the ball and place it in the center of the closest region.

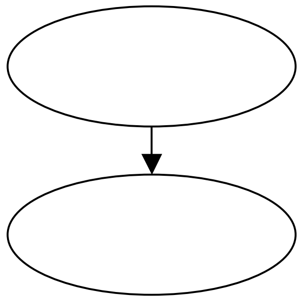
The putt and put-ball-in-cup skills use the strike and carry movements, and reference the cup. The carry skill picks up the ball and then moves forward the distance to the destination and drops the ball. The strike skill hits the ball at 600mm/s for 500 + d^2 milliseconds, where d is the distance from the ball to the destination position. The push skill is similar to strike, the robot hits the ball at 200mm/s for the distance to the destination position.

4.2. Robot Miniature Golf Domain Description

The domain description for robot miniature golf consists of the location of the robot, and the ball, the actions that move the ball to a region and the ball into the cup, and the Bayes network representing the state, shown in figure 6. The possible locations are numbered from 0 to 9, as shown in figure 6. The robot and the ball start in region 0 and the cup is in region 9.

The initial condition starts the ball in region 0, the goal is that the ball should be in the cup, the reward is 20.0 for getting the ball into the cup, and the expected utility threshold is 8.0. The expected utility threshold value is calculated as the reward minus the six steps that it takes to at least reach the goal times two for a higher than average action cost, 20.0 – 6*2 = 8.0. There are 64 actions, two that get the ball in the cup are: putting the ball into the cup and placing the ball in the cup. The remaining 62 actions strike, push, and carry the ball between regions.

To offer the greatest amount of flexibility for altering the course layout and changing probabilities, the actions are stored in an action transition matrix that has the start and end regions, and the probability distributions for the ball ending in a specific region. The



T	0.01
F	0.99

P(ball_at)		
location	ball_in_cup	~ball_in_cup
0	0.0	0.1
1	0.0	0.1
2	0.0	0.1
3	0.0	0.1
4	0.0	0.1
5	0.0	0.1
6	0.0	0.1
7	0.0	0.1
8	0.0	0.1
9	0.0	0.1

Figure 6. Miniature Golf State Representation.

execution monitor converts each line of the array into an action for the planner. For example, a push action from region 0 to 1 looks like:

0 1 push 1.00 0.05 0.95 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00

and represents a push from region 0 to region 1 with a cost of 1.00 to execute. The probability distribution for the ball after executing this push is 0.05 that the ball will still be in region 0, 0.95 that the ball will be in region 1, and 0.00 that the ball will be in any of the other seven regions. The push action is pretty reliable. The reason that there is a 0.05 probability of the ball remaining in region 0 is because of the small ridge between region 0 and region 1. The generated action for DT-Graphplan is represented as:

```
push_ball_0_to_1 1.00
:v
:p 0.70 > ball_at_0()
:e + 0.05 ball_at_0()
  + 0.95 ball_at_1()
```

Two of the three uncertain obstacles, the ‘Funoodle Surprise’, and ‘The Land of the Legobots’ can be seen in Figure 7. The ‘Funoodle Surprise’, is a large foam cylinder five feet high that is weighted and hangs from a string. The end of the foam hangs a half-inch from the floor and we supply a force that causes the foam cylinder to sway and rotate around the area between locations 5 and 6. The effect of this obstacle is that during a strike, the ball has a

0.05 probability of ending up in the regions 1 and 2 or not moving, and a 0.85 probability of ending up in the desired region. The 'Funoodle Surprise' has no effect on the push and carry actions. This is because the robot pushes the foam obstacle out of its way and away from the ball.

```
strike_ball_5_to_6 1.00
:v
:p 0.70 > ball_at_5()
:e + 0.05 ball_at_1()
   + 0.05 ball_at_2()
   + 0.05 ball_at_5()
   + 0.85 ball_at_6()

push_ball_5_to_6 1.00
:v
:p 0.70 > ball_at_5()
:e + 0.03 ball_at_5()
   + 0.97 ball_at_6()
```

Figure 7. The 'Funoodle Surprise', and 'The Land of the Legobots'.

The small ridge, shown in figure 8, has a similar effect as the 'Funoodle Surprise'. However, instead of the ball having a tendency to head to the left or right of the destination, the ball tends to stay in the same location by not making it over the ridge and rolling back. After a strike, the ball will end in the start region (region 0) or the region below it, region 4 with a 0.15 probability and in the destination region with a 0.70 probability. The ridge does have some effect on the push action, as occasionally the robot does not make it over the hump and stops at the top. The result is that the ball will roll back into the start region with a 0.05 probability and end in the desired region with a 0.95 probability.

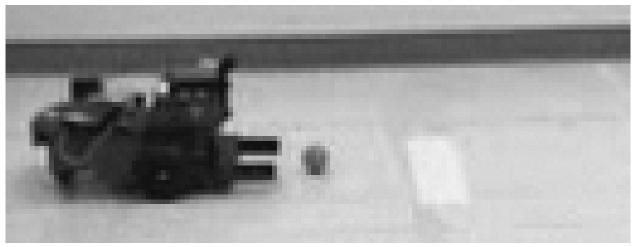


Figure 8. Small ridge with the Pioneer 1 robot.

The ‘Land of the Legobots’, towards the back of figure 7, is an area of the floor containing a Lego robot that randomly moves through the area. The Legobot predominately affects the strike action by disturbing the ball while the ball is rolling. The effect of this uncertainty is that the ball, when struck by the Legobot, could end up with fairly uniform probability in any one of regions 2, 3, 6, or 7.

4.3. Robot Miniature Golf Empirical Results

We implemented DTRC for this domain of robot miniature golf with the Pioneer 1 robot. This implementation employed the execution monitor, the skill set described in section 4.1, and the DT-Graphplan decision-theoretic planner. Having the robot complete the task of getting the ball into the hole without user intervention verifies that DT-Graphplan can perform in realistic domains in the presence of uncertain and noisy information.

DT-Graphplan generates a plan from the initial condition to a goal condition with an expected utility greater than 8.0 in an average of 109 seconds on a 300Mhz Pentium II with 97 MB of memory running Windows 2000. The pruning strategy is set to retain the best 60% of propositions. The generated plan is push the ball from region 0 to region 1, strike the ball from region 1 to 2, push from region 2 to 6, push to region 7, push the ball to region 9. The robot then attempts to putt the ball into the cup twice. This plan has an expected utility of 13.05.

Once a plan is found, the execution monitor begins to execute the plan. The execution monitor activates the push skill, to push_ball_from_0_to_1 and updates the current state. The state representation after this action is

```
ball_in_cup   0.000  1.000
ball_at   0.851 0.135 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.002
```

After observing that the ball is actually in region 1, the state representation becomes

```
ball_in_cup   0.000  1.000
ball_at   0.059 0.940 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

During execution, the plan generally fails at least once, spurring replanning. One of the most common reasons for plan failure is on the strike from region 1 to 2, the ball has a tendency to go too far, because the floor in the lab slopes down a little, something that was not noticed in generating the domain. The result is that the ball ends in region 3 instead of region 2. After executing the action, the state representation becomes

ball_in_cup 0.000 1.000
ball_at 0.001 0.850 0.129 0.001 0.001 0.008 0.001 0.001 0.001 0.001 0.001 0.001

If the ball ends in region 2 as it should, the state distribution is

ball_in_cup 0.000 1.000
ball_at 0.000 0.062 0.937 0.000 0.000 0.001 0.000 0.000 0.000 0.000 0.000 0.000

If, however, the ball does not end in region 2 but region 3, the observation does not change the state representation. This is because the result is not expressed in the domain description as a result of the 'strike_ball_1_to_2' action. Since it is not represented, the action's outcome distribution and the observation distribution are at odds with each other - the findings from the observation distribution are high when the action distribution has indicated that they would be low. During replanning, a plan with an expected utility greater than 8.0 is not found. The execution monitor then asks for user assistance. We choose to make no intervention and have the execution monitor use the last observation for replanning. Using just the current observation as the initial condition, DT-Graphplan locates a plan to get the ball into the cup. The plan generally has the robot take the ball through the edge of 'The Land of the Legobots' from region 3 to region 7 and then to region 9, by pushing the ball. If, however, we choose to move the ball back into region 2, where it should have ended after the strike, the DT-Graphplan generates a plan identical to the remaining steps of the original plan with an expected utility of 14.43.

After a series of failures of this type, we modified the planning domain to add a 0.10 probability of being in region 3 and lower the probability of ending in region 2 for the strike_from_region_1_to_2 action, here:

```
strike_ball_5_to_6 1.00
:v
:p 0.70 > ball_at_5()
:e + 0.05 ball_at_1()
   + 0.80 ball_at_2()
   + 0.10 ball_at_3()
   + 0.05 ball_at_6()
```

The result is the planner opts for a more expensive but safer plan by choosing to push the ball from region 1 to 2 instead of striking. The result plan has an expected utility of 12.86.

The state representations after each action and observation are shown in figure 9. This figure shows the actions nearing the goal state. The robot attempts to putt the ball into the cup, then loses sight of the ball, and the plan fails.

The second most common reason for replanning is due to difficulties in getting the ball into the cup. The reason for replanning here is similar to the strike from region 1 to 2, in that the probability distribution representing the effect of the action does not properly represent the action's outcome. The 'putt_to_cup' action, shown in figure 10, misses the cup more often than makes it into the cup. The result is the ball ends up not just in the current region but also in any of the regions surrounding the cup. If the ball remains in region 9, the action effect and the observation are similar enough that DT-Graphplan can generate a new plan, to putt the ball into the cup. The 'put_ball_in_cup' is fairly reliable, with respect to interpreting the image data and position information of the robot, the ball, and the cup.

By modifying the planning domain to more closely represent the outcome of 'putt_to_cup', the resulting plan makes no use of the action, instead attempting to execute 'put_ball_in_cup' twice instead. The resulting plan has an expected utility of 12.35.

```
Executed Action : push_ball_2_to_6
  ball_in_cup   0.000  1.000
  ball_at   0.002 0.002 0.787 0.002 0.002 0.002 0.193 0.002 0.002 0.002 0.002 0.002
After Observation
  ball_in_cup   0.000  1.000
  ball_at   0.000 0.000 0.039 0.000 0.000 0.000 0.960 0.000 0.000 0.000 0.000 0.000
Executed Action : push_ball_6_to_7
  ball_in_cup   0.000  1.000
  ball_at   0.002 0.002 0.002 0.002 0.002 0.002 0.824 0.160 0.002 0.002 0.002 0.002
After Observation
  ball_in_cup   0.000  1.000
  ball_at   0.000 0.000 0.000 0.000 0.000 0.000 0.049 0.950 0.000 0.000 0.000 0.000
Executed Action : push_ball_7_to_9
  ball_in_cup   0.000  1.000
  ball_at   0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.733 0.003 0.242 0.002 0.002
After Observation
  ball_in_cup   0.000  1.000
  ball_at   0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.029 0.000 0.970 0.000 0.000
Executed Action : putt_to_cup
  ball_in_cup   0.001  0.999
  ball_at   0.009 0.009 0.009 0.009 0.009 0.009 0.009 0.009 0.009 0.900 0.009 0.010
After Observation
  ball_in_cup
  0.000  1.000
  ball_at   0.090 0.090 0.090 0.090 0.090 0.090 0.090 0.090 0.094 0.010 0.090 0.090
 Failed to match next action. REPLANNING.
```

Figure 9. State representation during plan execution.

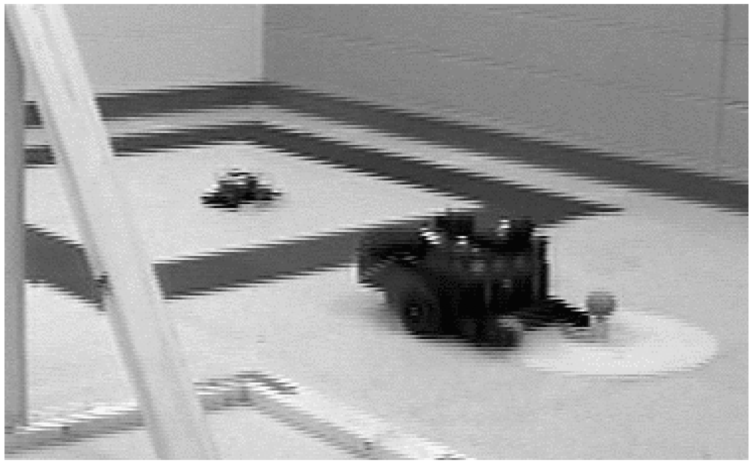


Figure 10. The robot attempting to putt the ball.

The implementation of the robot miniature golf domain demonstrates the integration of DT-Graphplan with the robot task architecture on an uncertain domain. For the robot miniature golf domain, the plan generation from initial condition to goal condition took 109 seconds. The plan execution of the plan was only successful on 20% of the attempts. On the failed attempts replanning took an average of twenty seconds, and the robot reached the goal. The replanning takes less time because the plans were on average a step or two long. This demonstrates the feasibility of the single-shot decision theoretic plan with replanning in a robot architecture.

The downside to using a decision theoretic planner for planning in a robot architecture is the additional work required to generate the probability distributions of the action outcomes, through experimentation or estimation. The user must also generate reward conditions, and an expected utility threshold. The benefits of the systems is that planning does take into consideration the uncertainty of the actions, choosing more certain actions over less certain actions. DT-Graphplan also incorporates action costs and orders goals based on rewards.

4.4. Robot Soccer

This section describes the changes that must be made to the skills set and the planning domains to switch the robot from the miniature golf to the soccer domain. The reason for selecting the robot soccer is that this domain, like robot miniature golf, is centered on moving a ball around a field and into a goal to score points. At the same time, the domains are sufficiently distinct to cause the robot skills to react differently. We discuss the tasks required to switch domains, and underline the concept that by using a decision-theoretic planner, the work required to switch tasks is less than generating a new controller to handle the decision-theoretic elements, for a system using symbolic planning.

There are two approaches to take in converting from Robot Miniature Golf to Robot Soccer. The first is to make the most use of the existing skills and structure. This entails representing the soccer field as a grid world just as in miniature golf. The second approach is to let the robot behave more reactively in the world and not superimpose a grid on the soccer field. The second approach is similar to the RoboCub robot soccer tournament [1].

Although both approaches have merit, the approach of turning the soccer field into a grid world emphasizes the requirements of switching domains of a single robot more than the second. The reason for this is that the first approach places more emphasis on planning, and is geared more toward a single robot soccer team. This correlates more with the miniature golf domain, than the second approach, mostly because we are trying to apply to a domain that has planning. For robot soccer, it is possible to play with a completely reactive agent, but this does not demonstrate the architecture. The skills set is similar to the skills used for robot miniature golf. In order to play, the robot needs to be able to locate and move the ball up the field and score a goal. These abilities already exist as miniature golf skills of locating the ball, moving to a location, searching an area for the ball, striking the ball, and scoring a goal (putt). The only additional skill is to guard the goal, which resembles a wait or noop operation. There is a new constraint on the strike skill, that if the opponent robot is in the region to which the ball is being hit, the uncertainty associated with the ball's destination increases.

The playing field is rectangular and divided into grid regions. The robot strikes the ball to move it from region to neighboring region and, once adjacent to the goal, attempts to score. This is a similar format to robot miniature golf. The differences between the domains are the initial and reward conditions, the action uncertainties, and interpreting sensor information to generate observations.

The initial condition for the domain positions the ball on the centerline with the robots facing each other on opposite sides of the centerline. One robot will be designated to kick the ball off; the receiving robot will be one region away from the ball. Both robots know the location of the ball and the other robot, and the locations of their goal and their opponent's goal.

The reward condition for the domain is as simple as a large reward for getting the ball into the opponent's goal. However, because of the adversarial nature of soccer, the reward conditions can include a degree of player psychology. For example, a robot can take an aggressive approach in actively seeking out the ball and attempting to score a goal at any cost (a single high reward for scoring a goal). Alternatively, the robot can take a defensive approach and try and guard the goal, only going for the ball when it comes near the goal, in this way attempting to prevent the other robot from scoring (the robot would receive a negative reward for losing a point, and possibly a small reward for being in states near the goal). An additional approach the robot could take is to play 'robot on robot', following the opponent robot around until the opponent finds the ball and then attempting to steal the ball and score (a reward for scoring, and an action description change). For each of these three approaches, the reward conditions and domain description the robot receives dictate the robots' overall behavior. By using reward in the domain, the robot acts on different strategies, this has also been shown using Q-Learning to learn individual strategies in a team given local and global rewards [2].

For observation information each robot relies on the vision system as in miniature golf. For the vision system, color is used to identify the ball and opponent robot when visible.

Given this observation information, if the ball or opponent robot is in the visual range of the robot, the observing robot has a good idea of the locations of both. Since the vision system uses only color, the robot cannot determine the orientation of the opponent. This would require recognizing the form of the object. Velocity could be estimated by taking multiple readings for the position of the opponent or ball.

In the robot miniature golf domain, the robot was trying to achieve a single specific goal. Here, in the robot soccer domain, as in the coffee/mail delivery robot domain, the goal is a maintenance goal. For as long as the game lasts, the robot is attempting to score a goal. Each time the robot scores a point the plan goal of scoring a point is not removed but maintained in the list of plan goals. The consequence of this is that as soon as the ball is kicked off again, the planner will again plan to score another goal.

With the exception that the objective of the robot is a maintenance goal, planning in the robot soccer domain is very similar to the robot miniature golf domain. Since the robot can count on having difficulties moving the ball from region to region when the opponent is around; the uncertainty of an action is higher near the opponent. Since the planner is atemporal, the planner cannot determine where the opponent may move in the future. There are two approaches to handling the effect the opponent has on the uncertainty of an action. The first is to assume that the opponent only affects actions near where it currently resides, for the entire length of the plan. The second is that an exogenous event is attached to the opponent's location that imposes a uniform probability distribution of the opponent's location over the entire soccer field. In essence after a number of plan steps, the probability of the opponent's location is uniform for all locations in the domain.

The first approach is simpler to implement, not having to add an exogenous event to every action. However, the second approach represents the knowledge the planner has about the opponent better. The second approach requires that a time function for the opponent's location be added to the outcome of each action, to represent the probability of the exogenous event. The ability to model an exogenous event is possible in the planner; however, the ability to have the function change over time is not possible. This addition to the planner is being left for future work.

The alterations to the skills and the new planning domain are the extent of the changes required to DTRC to convert from robot miniature golf to robot soccer. For comparison, the 3T system makes use of a symbolic planner to perform planning. In order to convert the robot from miniature golf to soccer using 3T, the same skills changed for DTRC need to be changed for 3T. For planning, less work would be required for 3T, as no uncertainties need to be provided for action outcomes, no costs provided for actions, and no reward conditions generated. However, in order to convert 3T from miniature golf to robot soccer, the RAPS layer must be changed [9].

In 3T, the RAPS layer serves a similar purpose as the execution monitor, receiving a plan from the planner, executing the skills and monitoring the outcome. RAPS differs by not maintaining a current state representation, and in handling of failure conditions. RAPS attempts to handle plan and skill failures, and attempts to rectify the failure to continue with the existing plan. RAPS follows a user defined set of rules in order to do this. The user is responsible for foreseeing any problem and writing a rule to bring the robot back to the state it needs to be in. If this is not possible, then replanning occurs, but because 3T does not maintain a state, the user must define rules for RAPS to generate an initial condition.

This section shows how the robot architecture can be switched from one domain to another. Many of the skills from the miniature golf domain are reused with little alteration in the robot soccer domain. And changes to the planning domain are shown as straightforward as the two domains are similar variations on path planning. Also discussed are domain change differences between DTRC and the 3T robot architecture.

Additionally, this section has discussed the application of DT-Graphplan in a robot task control architecture to an uncertain domain. The DT-Graphplan planner working in conjunction with the execution monitor handles the uncertainty of the domain, replanning when the plan fails using expected information from the last action and information from the current observation to generate a new initial condition for the planner. Failures that occur in the plan have been linked to errors in the effect distributions of the actions, which were not generated empirically. Modifying the domain to correct for these errors is detailed here, as is the effect on the plan generated and the expected utility of those plans. Unfortunately, the robot does not currently learn from the plan failures or perform automatic updates of the planning domain description.

5. CONCLUSIONS

The goal of robotics-related research is for robots to complete complex tasks in the real world. However, in the real world there is a great deal of uncertainty associated with actions, sensors, and environment information. This paper discusses DTRC, a robot control architecture, capable of interpreting uncertainty associated with real world domains and generating acceptable plans of action. DTRC consists of a low-level robot skill layer, an execution monitor, and DT-Graphplan. In DTRC DT-Graphplan generates a high level plan to complete a task, using the structure of planning domains and the propositional referencing of Graphplan to limit the space of plans searched, while reasoning about plan utilities, action uncertainties and resource constraints. The execution monitor triggers the robot skills according to the plan and maintains a representation of the current state using a Bayes network. The robot skills then control the actions of the robot and return observations to the execution monitor. Given this structure and using decision-theory at higher levels, there exists a straightforward method to generate an initial condition for replanning exists and additionally, a more general robot controller that simplifies converting the robot to a new task.

By implementing DTRC on the task of robot miniature golf, and discussing the subsequent requirements of conversion to the domain of robot soccer, this work shows that using decision theory at higher reasoning level has three benefits: (1) The amount of work required to convert a robot from one task to another is reduced, (2) the controller can numerically evaluate one plan against another, and (3) by maintaining a current state representation in a Bayes network, the controller has a straightforward method of generating initial conditions for replanning on plan failure or on the addition of new goals.

Extensions to DTRC include applying the controller to a continuous task with maintenance goals and move the robot into a less structured environment. We also hope to add a learning element, so the robot can learn to adjust the action effect probabilities when the estimated probability distribution does not accurately reflect the actual effect probability distribution.

References

[1] Asada, M., Birk, A., Pagello, E., Fujita, M., Noda, I., Tadokoro, S., Duhaut, P., Stone, P., Veloso, M., Balch, T., Kitano, H, Thomas, B., (2000). Progress in RoboCup Soccer Research in 2000. In *Proceedings of the 2000 International Symposium on Experimental Robotics*.

[2] Aylett, R. S., Coddington, A. M., Barnes, D. P., and Ghanea-Hercock, R. A., (1997). What does a planner need to know about execution? In Recent Advances in AI planning, eds S.Steel & R.Alami, 26-38, Springer.

[3] Balch, T., (1997). Learning Roles: Behavioral Diversity in Robot Teams. In *1997 AAAI Workshop on Multiagent Learning*.

[4] Blum, A. L., Furst, M. L., (1995). Fast Planning Through Planning Graph Analysis. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 1636-1642.

[5] Blum, A. L., Langford, J. C., (1999). Probabilistic Planning in the Graphplan Framework. In *The 5th European Conference on Planning* (ECP'99).

[6] Bonasso, R. P., Kortenkamp, D., (1996). Using a layered control architecture to alleviate planning with incomplete information. In *Proceedings of the AAAI Spring Symposium*. “Planning with Incomplete Information for Robot Problems”. 1-4.

[7] Boutilier, C., Dean, T., and Hanks, S., (2000), Decision-Theoretic Planning: Structural Assumptions and Computational Leverage*, Journal of Artificial Intelligence Research* 1, 1-93.

[8] Draper, D., Hanks, S., Weld, D., (1994). Probabilistic Planning with Information Gathering and Contingent Execution. *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems,* 31-36.

[9] Firby, J. R., (1996). Modularity Issues in Reactive Planning. *In Proceedings of the Third International Conference on AI Planning Systems*, Edinburgh Scotland, 78-85.

[10] Kushmerick, N., Hanks, S., Weld, D., (1994). An Algorithm for Probabilistic Planning, *Artificial Intelligence* 76, 239-286.

[11] Nourbakhsh, I., (1997) "Interleaving Planning and Execution for Autonomous Robots.” PhD thesis. Also available as technical report STAN-CS-TR-97-1593, Department of Computer Science, Stanford University.

[12] Simmons, R., Goodwin, R., Haigh, K. Z., Koenig, S., and O'Sullivan, J., (1997). A Modular Architecture for Office Delivery Robots, in Autonomous Agents 1997 ACM. 245-252.

[13] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Haehnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, (2000). Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva*. International Journal of Robotics Research*, 19(11). 972—999.

[14] Veloso, M., Carbonell, J., Perez, A., Borrajo, D., Fink, E., and Blythe, J., (1995). Integrating planning and learning: The prodigy architecture. *Journal of Experimental and Theoretical AI* 7, 81-120.