Satisficing Strategies for Resource-Limited Policy Search in Dynamic Environments

Dmitri Dolgov and Edmund H. Durfee

Computer Science and Engineering
University of Michigan, EECS Department
Ann Arbor, MI 48109

ddolgov@umich.edu, durfee@umich.edu

ABSTRACT

In this work, we examine the problem of searching for schedulable real-time control policies for resource-limited agents acting in dynamic environments. The dynamic properties of the environment and resource limitations of the agent render the problem of solving for an optimal policy infeasible. We therefore limit our search to a satisficing, rather than an optimal policy. We view the policy search as a search for a state reachability graph, with an action assigned to each of the states in the graph. In the search algorithm, we exploit properties of the reachability graphs to propagate failure conditions from inherent failure states to other states in the reachability graph, which allows us to exploit constraint satisfaction techniques to quickly remove some unacceptable policies from consideration. Our analysis and experiments show that, under certain conditions, such as when the “safe” states in the reachability graph are separated from the failure states by a relatively small set of states, we can use backtracking and memoization techniques that significantly improve the efficiency of the search algorithm.

Categories and Subject Descriptors

I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search – *Backtracking, Plan execution, formation, and generation.*

General Terms

Algorithms, Performance, Design, Experimentation

1. INTRODUCTION

A resource-limited agent, operating in a dynamic environment, faces complex tradeoffs when formulating its control policy. On the one hand, because the world can change while the agent is deciding what to do, it needs to rapidly attend to the important cues in its environment that demand responses. On the other hand, because the agent is resource limited, resources that it devotes to monitoring some cues mean that other cues get less frequent attention. The problem of dynamic construction of real-time control

plans is addressed by the Cooperative Intelligent Real-Time Control Architecture (CIRCA) [8], [9].

To appeal to intuition, consider a (resource-limited) person who is driving in a (dynamic) city environment. The driver wants to avoid collisions while finding a destination, and will cycle through a number of activities toward this end such as looking at the road ahead, watching traffic on either side, checking the speedometer, glancing at the rear-view mirror, reading names of cross streets, etc. Depending on the observations, the driver can accelerate, brake, turn, etc. Time spent on one activity (e.g., looking at building numbers) takes attention from another activity (e.g., watching the road ahead), which in turn can lead to failure (e.g., rear-ending a car ahead).

In this paper, we are looking at the analogous problem of finding a schedulable control policy for a computational agent operating in a dynamic domain. In some respects, the problem resembles a Markov Decision Problem (MDP). However (as we shall discuss in more detail later), the time costs for determining the current state, coupled with the resource limitations that mean that the choice of how to respond to a particular state can impact the responses that can be chosen for another state, make a dynamic programming solution strategy impossible. The set of computationally feasible (schedulable) control policies is a subset of all control policies, and typically will not include the optimal policy based on unlimited resources.

The challenge, as outlined more fully in Section 2, is to search through the space of control policies to find one that is schedulable and that satisfies the constraints of avoiding failure states and (eventually) reaching a goal state. To do this, we adopt plan-space search techniques for searching through the space of reachability graphs, along with ideas for solving constraint satisfaction problems in order to quickly prune away infeasible portions of the search space (Section 3). We overview the control policy search strategy in Section 4, and show how techniques can be incorporated for dependency-directed backtracking (Section 5) and no-good processing (Section 6). Using analysis and experiments, in Section 7 we show the potential benefits of these improvements, and then conclude with a look towards future work (Section 8).

2. THE PROBLEM

We adopt the following description of the problem. The world is characterized by a set of *world features* F , where each feature $f \in F$, has an associated domain of possible values V . Each state in the world is described by an assignment of particular values to features. A *node* in the state space consists of a reference to a state

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AAMAS '02, July 15-19, 2002, Bologna, Italy.

Copyright 2002 ACM 1-58113-480-0/02/0007...\$5.00.

and some extra information about that particular instance of the state (for example, probability of reaching the state).

A *reachability graph* consists of one or more nodes connected by transitions representing which states can arise from which other states. There are two types of transitions that we model. *Action transitions* are explicitly controlled by the agent. Each state can have only one action assigned to it. Events outside of the agent's control are modeled as *temporal transitions*. A temporal transition (*tt*) that leads to a failure state is labeled as a *tff*. There can be multiple temporal transitions leading from a state, corresponding to the various events that can occur if the system remains in that state long enough.

Each transition (temporal or action) is defined by a set of *preconditions* that specify feature value assignments that have to be satisfied in order for the transition to be valid, and a set of *postconditions* that specify feature value assignments that hold after the transition is executed. Each temporal transition also has the following characteristics: a probability, which describes how likely that particular transition is to occur, and a *minimum delay*, which specifies the minimum time until the transition may occur. These transition probabilities specify the base values and are normalized during planning so that the total probability of all transitions leading from a state is 1. Each action has a worst-case execution time (*wcet*) that is the time between when it starts being considered and when its changes to the world go into effect. During planning, an action is also assigned a *period* that dictates the maximum elapsed time between when the action is considered. Summed together, the *period* and the *wcet* determine the *maximum delay* between when a state arises for which the action has been planned and when the agent notices this and completes the action. A transition to a failure state can be *preempted* by an action whose *maximum delay* is less than the *minimum delay* of the *tff*, and therefore the preempted failure state is not part of the reachability graph.¹

The objective of the control policy search is therefore to find a reachability graph beginning with a set of initial states, such that the graph includes one or more states that satisfy the goal test and are likely enough to be reached. The graph also should not include any failure states, or if it does then the probability of reaching them should be so low that they can be safely ignored. The reachability graph is *complete* when each state in the graph has been expanded, and thus has an action (or a no-op) assigned to it, where the *period* of the action is specified. A reachability graph is *feasible* when all of its actions with their *wcets* and *periods* are schedulable. The control policy search has succeeded when a feasible, complete reachability graph has been found.

To illustrate these concepts, consider the following Unmanned Combat Aerial Vehicle (UCAV) problem. The objective of the search is to find a schedulable control policy that lets the plane fly from location A to B without being hit by an enemy missile. In order to accomplish that, the agent should come up with an action for each situation that it might encounter such that these actions will let it avoid failure. In the process, it will also need to determine the periods for all actions used to preempt failures. These periods will determine whether the solution is feasible, i.e. whether the corresponding control policy can be scheduled.

¹ To keep discussion simple, this characterization does not involve more complex probabilistic representations [6], although the algorithms in this paper will work with those as well.

In this example, suppose that a solution reachability graph satisfies the failure test if it does not contain failure states that are reachable with probability greater than $\epsilon=0.07$ and if it includes a goal state that is reachable with probability greater than $\delta=0.7$. Figure 1 gives a complete description of the domain.

Figure 2 shows a solution to the above problem. Notice that there is a failure state in the graph, but the probability of reaching it ($p=0.06$) is less than ϵ , so the reachability graph successfully passes the failure test. The probability of reaching the goal state is greater than δ , so the goal test is satisfied as well. Furthermore, all state nodes in the graph are expanded. Therefore, we consider the reachability graph in Figure 2 to be a satisfactory solution assuming that the actions are all schedulable. Methods for generating satisficing reachability graphs are examined in the next sections.

World Features:	
location:	{A, B, C}
decoy:	{yes, no}
flightMode:	{onGround, normal, evasive, hit}
threat:	{yes, no}
decoyDeployed:	{yes, no}
Initial state:	
(flightMode=onGround, decoy=no, location=A, threat=no, decoyDeployed = no)	
Goal:	
(flightMode = onGround) AND (location = B)	
Actions:	
<u>LoadDecoy</u>	
pre:	(flightMode=onGround) AND (location=A) AND (decoy=no)
post:	(decoy = yes), <i>wcet</i> =1, <i>period</i> =1000 (no ttf to preempt)
<u>HeadingToC</u>	
pre:	(flightMode = onGround) AND (location = A)
post:	(flightMode=normal) AND (location=C), <i>wcet</i> =1, <i>period</i> =1000 (no ttf to preempt)
<u>HeadingToB</u>	
pre:	(flightMode=onGround) AND (location=A OR location=C)
post:	(flightMode = normal) AND (location=B)
<i>wcet</i> =1, <i>period</i> =1000 (no ttf to preempt)	
<u>Land</u>	
pre:	(flightMode = normal) AND (location=B)
post:	(flightMode = onGround), <i>wcet</i> =2, <i>period</i> =1000 (no ttf to preempt)
<u>EvasiveManeuver</u>	
pre:	(flightMode=normal) AND (threat=yes)
post:	(flightMode = evasive), <i>wcet</i> =1, <i>period</i> =9
<u>DeployDecoy</u>	
pre:	(flight = evasive) AND (threat = yes) AND (decoy = yes)
post:	(decoyDeployed = yes), <i>wcet</i> =1, <i>period</i> =11
<u>ResumeFlight</u>	
pre:	(flight = evasive) AND (threat = no)
post:	(flightMode = normal), <i>wcet</i> =2, <i>period</i> =10
Temporal transitions:	
<u>MissileThreat</u>	
pre:	(flightMode = normal) AND (threat = no) AND (location= B)
post:	(threat = yes), <i>p</i> = 0.5, <i>min-delay</i> = 0
<u>NormalHit</u>	
pre:	(flightMode = normal) AND (threat = yes) AND (decoyDeployed = no)
post:	(flightMode = hit), <i>p</i> = 0.9, <i>min-delay</i> =10
<u>EvasiveHit</u>	
pre:	(flightMode = evasive) AND (threat = yes) AND (decoyDeployed = no)
post:	(flightMode = hit), <i>p</i> = 0.5, <i>min-delay</i> =12
<u>DecoyHit</u>	
pre:	(threat=yes) AND (decoyDeployed=yes)
post:	(flightMode = hit), <i>p</i> = 0.01, <i>min-delay</i> =12
<u>MissileMiss</u>	
pre:	(threat = yes)
post:	(threat = no), <i>p</i> = 0.1, <i>min-delay</i> =0

Figure 1 UCAV domain description

3. SOLUTION CONCEPTS

An obvious place to start in developing computational mechanisms for formulating a control policy is to model the problem as a Markov Decision Problem (MDP) and solve it in a straightforward way using a dynamic programming method such as value or policy iteration [3]. Such formulations typically assume that the world is accessible (the agent can know what state it is in) and the world is stationary (nothing changes until the agent executes an action). Some modifications to this kind of problem make different assumptions. For example, incomplete access to the world state leads to techniques for Partially Observable MDPs [1], [13].

Our problem, as we have described it, assumes full accessibility, but there is a hitch: it takes time on the part of the agent to know which state it is in. Furthermore, because the world is not stationary, it is possible that the world will change states while the agent is in the process of determining the current state. Because we assume that there are fewer actions an agent can take than states that it can be in, we assume an agent will adopt a control technique of cycling through its possible responses (where some critical responses might appear more often in each cycle), and classifying the current state only so far as to determine whether it fits the response. In other words, the agent only determines enough about the state to decide whether to take the action in question.

Figure 2: Reachability graph for the UCAV domain. A state has features <location, decoy, flightMode, threat, decoyDeployed>. The number next to a state is the probability that it will *eventually* be reached (see [6]). Dark arrows indicate action transitions; temporal transitions are indicated by light arrows. The gray state is a failure, and the dark outline state is a goal. The number next to a transition is the probability that it will occur, given its parent state.

An agent can exercise the most control when it is constantly monitoring for a type of situation and then reacting faster than the dynamics of the world. This is the idea of preemption described in the previous section. Unfortunately, the more frequently the agent checks for one type of situation, the less frequently it can check for others due to its resource limitations. Therefore, even if an agent can formulate an optimal control policy, that policy will usually not be feasible to execute. As mentioned earlier, typically MDP techniques assume that dynamic programming will work because a policy decision for one state does not affect the choices for other states and, therefore, a globally optimal policy can be constructed from locally optimal action choices. For a resource-limited agent, this local optimality principle fails to hold, leading to an intractable optimization problem. In this paper, we focus instead on finding a control policy that is satisficing: it is good enough for avoiding failure and achieving the goal, while being executable by the agent.

3.1 Plan-Space Search

The planning community has developed a variety of strategies for finding satisficing plans that are assured of achieving goals. For example, the Partial Order Planning (POP) algorithm, as de-

scribed in [11], characterizes the search for a plan as a process of adding steps and constraints to a partial plan until a plan that is consistent and complete (where all conditions have been established and are not threatened) is formed. Depending on various choice strategies (the order of conditions to establish, the selection of operators to establish conditions, etc.), different plans might be formed. Any plan that is complete and consistent constitutes a satisfactory solution. Whenever an inconsistency arises, the search can backtrack to a previous choice point and try an alternative.

For our problem, we can analogously consider the search for a control policy as the iterative construction of a reachability graph. The graph begins with the initial state(s), and is enlarged as states are expanded with action and temporal transitions. The search is complete when a graph is formed that is complete² (when all states in the graph have been expanded) and consistent (no failure states are included and a goal state is). On top of this, we need to add a requirement that the actions incorporated in the graph need to be schedulable. The details of our approach are presented in Section 4.

3.2 Constraint-Satisfaction Search

Implicit in the POP algorithm are concepts from searching in constraint satisfaction problems (CSPs). The goal of POP is to assign to each condition (in CSP parlance, variable) an establishing step (value) such that all constraints are satisfied. Furthermore, it is clear that a partial plan that has an inconsistency can never be expanded into a complete and consistent plan. Thus, portions of the potential solution space can be pruned quickly when inconsistencies arise.

Our problem can similarly be viewed as a constraint satisfaction problem, since, in essence, the objective of the search is to come up with action assignments for every state. In CSP terminology, states correspond to variables, action choices to assignment of values to the corresponding variables, and the failure and goal tests can be easily modeled as constraints. If a subgraph of the reachability graph contains a failure state, the graph as a whole cannot be extended to a graph that does not contain the same failure state. In other words, if a partial graph has no viable policy, any graph that includes it also has no policy. Similarly, the order in which values are assigned to variables does not affect the solution, since each variable must eventually have a value assignment. The ordering certainly affects the search path, and some orderings of variables (state expansions) can result in far more efficient search paths than others, but the resulting solution is the same. For example, a CSP search often involves remembering “nogood” partial assignments so that if an assignment, containing a nogood subset is encountered at a later point, this search path can be immediately pruned away. In our problem, because of the structure of the reachability graph where once a state is added to the graph its descendants follow, nogoods can be reduced down to single states. That is, in the course of the search, states can be identified that can be labeled as “nogood” because they inevitably lead to failure. Identifying and remembering such states can help prune away large portions of the reachability graph search space, as is detailed later.

² Here, we use the term *complete* in the planning sense, and it should not be confused with a completely-connected graph.

4. SEARCH FOR SATISFICING, EXECUTABLE POLICIES

4.1 Reachability Graph Search

In this section, we present an algorithm for searching for a satisficing reachability graph. Figure 3 describes the algorithm in pseudo-code. The search starts with a sub-graph that contains only the initial states. It then expands nodes one by one until the graph contains no unexpanded nodes. If at some point it encounters a subgraph that is not “viable,” it backtracks. For now, let us define “viable” as passing the failure test. We will extend this definition later, when we include the notion of nogood lists. Notice that there are a couple of ways that we can influence the search:

- *choose_node_to_expand(graph)* determines the order in which we expand nodes. This corresponds to variable ordering heuristics in CSPs, such as most-constrained variable first.
- *choose_best_action(node)* determines the order in which we consider action choices for states. This corresponds to value selection heuristics in CSPs, such as least-constraining value first.
- *choose_backtrack_node()* determines the node where the search backtracks from when it finds a dead-end subgraph, i.e. a sub-graph that contains no unexpanded nodes, but does not satisfy the goal test. A good heuristic would backtrack in such a way as to minimize the changes to the old graph incurred by the search for a goal state.

Notice, that the search algorithm utilizes two different types of backtracking. One is invoked when a “non-viable” sub-graph is encountered. This backtracking procedure tries to remove all transitions to failure nodes from the graph. The other type is invoked when the search hits a “dead-end”, i.e. when it finds a completely expanded sub-graph that does not satisfy the goal test. In that case, the backtracking procedure undoes some previous action choices and makes alternative decisions.

4.2 Checking Policies

As we are dealing with resource-limited agents, not all policies might be executable (we use the terms “schedulable” and “executable” interchangeably), since some of them might require more resources than are available to the agent. To accommodate for that fact, we have to check whether the policy captured in a reachability graph is executable. This can be accomplished by embedding the algorithm from Figure 3 inside of a loop that verifies that the policy returned by *search_reachability_graph()* is schedulable. Figure 4 contains the pseudo-code for such an algorithm.

4.3 Checking Policies

As we are dealing with resource-limited agents, not all policies might be executable, since some of them might require more resources than are available to the agent. To accommodate for that fact, we have to check whether the policy captured in a reachability graph is executable. This can be accomplished by embedding the algorithm from Figure 3 inside of a loop that verifies that the policy returned by *search_reachability_graph()* is schedulable. Figure 4 contains the pseudo-code for such an algorithm.

Notice that *policy_search()* can modify the parameters that are passed to *reachability_graph_search()*. For example, if *reachability_graph_search()* fails to find a policy given the current goal

and failure tests, *policy_search()* can relax the problem by raising the failure probability threshold (ϵ) or lowering the goal probability threshold (δ).³ Because this loop could iterate numerous times, it is important that *reachability_graph_search()* avoids unnecessary search. Techniques for making *reachability_graph_search()* more efficient are the subjects of the next two sections.

```
function reachability_graph_search
input:  list of initial states
        failure test
        goal test
output: a reachability graph that includes all
        initial states, satisfies the goal test,
        passes the failure test, and all states in
        the graph are expanded or an empty
        graph if no satisficing graph is found
{
  graph ← {all initial states}
  while (graph does not satisfy goal_test) {
    while (graph contains unexpanded nodes) {
      current_node ← choose_node_to_expand(graph)
      apply temporal transitions to current_node
      action ← choose action for current_node
      apply action transition to current_node
      if (graph is not viable) {
        backtrack so that graph passes failure_test
        if (cannot backtrack) return 0
      }
      if (graph does not satisfy goal_test) {
        backtrack_node ← choose_backtrack_node(graph)
        backtrack from backtrack_node so that
          graph has unexpanded nodes
      }
    }
  }
}
```

Figure 3 Pseudo-code for generating satisficing reachability graphs

```
function policy_search
output: an executable policy
{
  do {
    initial_states ← determine_initial_states()
    goal_test ← determine_goal_test()
    failure_test ← determine_failure_test()
    policy ← reachability_graph_search(initial_states,
                                      failure_test, goal_test)
    schedulable ← schedule_policy(policy)
  } while (NOT schedulable)
}
```

Figure 4 Pseudo-code for the executable policy search algorithm

5. Backtracking

As partial reachability graphs are generated, the search algorithm has to be able to backtrack if it hits a search node (partial graph) that cannot or should not be expanded further. In this section we examine some common backtracking methods.

5.1 Chronological Backtracking

One of the simplest methods of backtracking is to reconsider previous choices of actions in reverse chronological order, i.e. in the order opposite to which actions were chosen and nodes expanded. However, if the search process chooses which unexpanded node to expand next in an erratic way, chronological backtracking could do a very significant amount of unnecessary and useless backtracking by reconsidering portions of the reachability graph that have nothing to do with reasons for backtracking. This has been noted by many others (e.g., [5]), such as for map-coloring CSPs where backtracking to loosely-connected portions of the constraint graph is unlikely to help resolve a constraint violation in a different portion, and undoes probably perfectly good sets of bindings for those loosely-connected portions.

³ Of course, other outer loops are possible, such as starting with very easy thresholds and iteratively making them harder. Such an approach can lead to an interruptible “any-time” search [14].

Figure 5: Search tree for problem in Figure 1 as generated by the algorithm in Figure 3. Some nodes are omitted for brevity.

The tree in Figure 5 would be generated by a search that uses chronological backtracking. One can notice that whenever the search encounters a failure reachability graph, it backtracks by reconsidering actions in reverse chronological order. For example, when the forward expansion search encountered the first failure reachability graph (bottom left), it tried to choose another action for the last expanded state (state “6”). That state had no action alternatives, so the search backtracked to the state that was expanded just before that (state “3”). It then tried a different action for that state (*land*, instead of *evasive*), just to find another failure.

Now imagine that there is another transition from the initial state that leads the search into a completely non-overlapping state space. An example of such a transition would be an engine failure before takeoff. It is clear that this portion of the state space does not affect the portion that deals with the plane flying around B. Further imagine that, after expanding state “3”, the forward search decided to explore the state space that is reachable via the engine failure transition. In fact, it might explore a huge portion of that state space before getting back to states that involve flying around B. Therefore, when the search finally expands state “6”, encounters a failure, and decides to backtrack chronologically, it will have to try different actions for all states that it has expanded in the engine-failure portion of the space before reconsidering state “3”. Moreover, when it finally gets to state “3”, it has collapsed the engine-failure part of the space, which will have to be regenerated later. Even worse, if there are many unavoidable failures in the portion of the space that deals with flying around B, the search might collapse and then re-explore the whole engine-

failure sub-space between every forward and backwards step it takes in the “flying around B” portion of the reachable state space.

5.2 Dependency-Directed Backtracking

Dependency-directed backtracking is more informed than chronological backtracking. Instead of going to the last expanded state, it makes use of the constraints to focus backtracking on decisions connected to the failure. In the context of a reachability graph, a failure state arises because of choices made in its ancestor states, so dependency-directed backtracking can simply be implemented as working backward along parent-child relationships. Therefore, if we consider the problem from the last section, it would not make any difference for dependency-directed backtracking whether another portion of the space has been explored between the expansions of states “3” and “6”. The algorithm would backtrack to state “3” right away, without undoing any actions in unrelated portions of the state space.

6. NOGOOD LISTS

A *nogood* is a CSP term that refers to a (partial) assignment of values to variables that violates some constraint(s) or otherwise cannot be extended to a solution. A *minimal nogood* is a nogood of which no subset is a nogood ([4], [10], [12]). We can apply the notion of nogoods to our problem, but will push on it further. Rather than remembering (many) nogood combinations of state-action pairs, a more minimal set of nogoods will simply remember states that, while not failure states, are equivalent to failure states because each unavoidably leads to failure no matter what action is

taken in that state. Such failure equivalent states should be “black-listed” in the terminology of [7].

During the search in Figure 5, for example, we saw that taking the action “Land” in state “6” introduced a failure state that can be reached with sufficiently high probability to make the reachability graph inconsistent with the safety constraints. Since state “6” has no other matching action choices, our extensions for nogood memoization treat state “6” as a no-good: whenever that state is added to any future graph, search along that path should end and backtracking should begin. Similarly, once all possible expansions of state “3” have failed to avoid failure, state “3” is added to the nogood list, as ultimately is state “2”. The bottom line is that the plane should not leave without loading the decoys.

A search tree generated by the nogood-memoizing algorithm would be different from the one in Figure 5. In particular, whereas the search in Figure 5 had to redo a lot of search in the middle branch (corresponding to getting to the same states as the left branch via a different route), the search performed by a nogood-memoizing algorithm would be truncated once it is determined that the middle branch leads inexorably to nogood states.

In our previous analysis, we have ignored state probabilities. However, introducing state probabilities and a failure probability threshold ϵ , as described in Section 2, adds a complication to the problem. With that change, it is no longer enough to store just the states in the nogood list. It is necessary to include the probability that this state had when it was labeled as a nogood. Clearly, if we reach a known nogood state with a probability higher than before, the new state must also be a nogood, because any transitions to failure that were applicable to the old state are applicable to the new state, resulting in higher probabilities of failure states. In this case, we can say that the state in the nogood list *dominates* the current state. On the other hand, if the current probability is lower than the probability of the state stored in the nogood list, we can not make any assumptions about the current state, because its new lower probability might cause the probabilities of failure states to fall below the threshold, thus effectively removing the failure. In that case, the state in the nogood list does not dominate the new state, and it is necessary to re-expand the new state.

Another complication arises when we consider the determination of minimal nogoods. If the failure probability threshold is zero ($\epsilon=0$), any reachability graph that contains failure states is not a solution and cannot be extended to one. In this case, the minimal nogoods are reduced to single states as described above. However, when the probability threshold is nonzero ($\epsilon>0$), the minimal nogood has to include the *combination of states* that unavoidably lead to failure. For example, if $\epsilon=0$, each of the parents of a particular failure (or failure-equivalent) state can be labeled as a nogood, provided that the parent has no action assignment that preempts reaching the failure (or failure-equivalent) state. On the other hand, if $\epsilon>0$, only the combination of parents (with their respective probabilities) of a failure state can be labeled as a nogood, since all of the parents contribute to the probability of the failure state. That is, reaching one of the parents with the same or higher probability in some other context will not necessarily imply that the failure state is reached with sufficiently high probability, because other routes to it might be less probable.

7. PERFORMANCE ANALYSIS

7.1 Backtracking

The behavior of chronological backtracking is dependent on the order in which states are expanded during forward search, because chronological backtracking reconsiders actions in reverse chronological order. Dependency-directed backtracking, on the other hand, does not depend on the order of forward expansion. Therefore, the two backtracking methods might behave very similarly or very differently, depending on the order of forward expansion.

To drive this point home consider a simple analogy of a search tree. If the forward expansion performs a depth-first search, chronological and dependency-directed backtracking will behave in exactly the same way. If, however, forward expansion performs a breadth-first search, chronological backtracking will “erase” all nodes below the cause of the problem, whereas dependency-directed backtracking will erase the nodes on the path from the cause of the problem to the current node. In fact, if the problem is discovered at depth d and its cause is at depth p , chronological backtracking will collapse approximately b^d states, whereas dependency-directed backtracking will only collapse b^{d-p} states.

To get a better sense of what these differences mean empirically, we performed some experiments. Clearly, the performance of the techniques, and more generally the challenges in finding complete and consistent reachability graphs, depend on the domain, including how large the state space is and how interconnected the states are. To avoid the bias that could arise from testing on a small number of hand-crafted cases, we instead implemented a system for generating “random” knowledge bases (that specify the state features, initial state(s), and action and temporal transitions). The random knowledge base generation process accepts several parameters that influenced the expected features of the resulting reachability space, including the number of features in the world, the average number of actions and temporal transitions per state. It was also possible to control the locality and generality of actions and temporal transitions. The locality is a measure of how much the transitions change the state to which they are applicable, and generality is a measure of how widely applicable the transitions are. Figure 6 and Figure 7 show some experimental results using randomly generated knowledge bases, where the search heuristic was simply to expand the most probable state next. In each case, 2000 knowledge bases with the same average size of the reachable state space were generated, and for each knowledge base the search was conducted with chronological and with dependency-directed backtracking. In the figures, the number of expansions needed for chronological backtracking is usually greater than those needed for dependency-directed backtracking. Figure 6 was generated from knowledge bases with a high average number of temporal transitions per state, while Figure 7 was generated from knowledge bases with a low average number of temporal transitions. This difference in the corresponding knowledge bases explains the difference in the graphs. In the case of few temporal transitions (Figure 7), forward expansion resembled a depth-first search, and therefore, dependency directed backtracking behaved very similarly to chronological backtracking. In the other case, where there were many temporal transitions (Figure 6), forward expansion performed more of a breadth-first search, and it is evident that dependency-directed backtracking was more efficient. It is worth noting that the points on the diagrams include a large number of points that correspond to problems that required no

backtracking whatsoever. Obviously, in that case both algorithms behaved the same way.

Another point worth observing is that in some (relatively rare) cases, chronological backtracking outperformed dependency directed backtracking. This is due to the fact that, in a strongly connected graph, changing a portion of the graph that is seemingly unrelated to the failure may change the probabilities of states around the failure in such a way that the probability of reaching a failure state drops below the failure probability threshold, thus serendipitously solving the problem.

Figure 6 Chronological vs. dependency directed backtracking. Forward expansion is very similar to BFS. The line is $x=y$ (no speedup).

Figure 7 Chronological vs. dependency directed backtracking. Forward expansion is very similar to DFS. The line is $x=y$ (no speedup).

7.2 Nogood Memoization

Obviously, nogood memoization reduces the number of state expansions in some cases, and the nogood-memoizing algorithm never expands more states than the one without nogood memoization. This is accomplished by the making the classic tradeoff between time and space complexity. In this section we present some experimental results of nogood memoization efficacy, and discuss some properties of the reachable state space that affect the performance.

The space of all reachable states can be roughly broken up as shown in Figure 8. In this diagram, F is the set of all failure states, i.e. states that are themselves inherently bad. B is the set of all “bad” states, which consists of states that are either inherently bad (F) or have non-preemptable transitions to other bad states. The

states in $B - F$ are the failure equivalent states, which are stored in the nogood list.⁴ S is the set of all “safe” states, i.e. states that have transitions that can avoid failure or bad states. D is a subset of S representing the set of all “dangerous” states, which are the states that contain transitions to bad states but those can be preempted by some action. In other words, from some state in D it is possible to go to a safe state or a bad state, depending on the choice of action. B' is a subset of B that consists of states that are immediately reachable from states in D . In other words, to get to a state in B , it is necessary to go through at least one state in B' . Note that B' can include both inherent failure states and failure equivalent states, though this is not explicitly shown in Figure 8.

Figure 8 Reachable state space

Let us consider a qualitative hypothesis as to the number of states explored by an algorithm that memoizes nogood states, relative to the number of states explored by an algorithm that performs no memoization. We will consider the worst-case scenario, in which all possible bad reachability graphs will be explored before the search finds a satisfying reachability graph or determines that no such graph exists.

An algorithm that memoizes nogoods will only have to explore the states in B once, whereas an algorithm that performs no memoization might have to explore the set B multiple times. We can consider how effective nogood memoization will be, depending on the relative sizes of the sets described above.

If $|S| \gg |D|$, the problem is underconstrained, and it is relatively easy to find a satisfying solution. Therefore, memoizing nogoods will not greatly improve the efficiency of the algorithm. Nogood memoization is far more useful when $|S| \sim |D|$.

If $|B| \sim |F|$, there is little point in remembering nogoods because there are few. This is due to most of the bad states being inherently bad (known to be nogood *a priori*). Nogood memoization is much more effective when $|F| \ll |B|$.

If $|D| \sim |B'|$, memoizing nogoods will not be very effective since different states in D will have different corresponding bad states in B' , and the search will likely explore a new subset of B every time it enters it. Thus, the chances of reusing the nogood information will not be very high. The speedup from using nogood memoization occurs when $|D| \gg |B'|$.

Hence, we hypothesize that nogood memoization is most useful when $|B| \gg |S| \sim |D| \gg |B'|$. In other words, it is most useful for solving highly constrained ($|B| \gg |S|$) problems, where the implied bad states are separated from the safe states by a relatively small set of states ($|D| \gg |B'|$). In that case the improvement in efficiency, which can be expressed as the ratio of the number of state expansions performed by an algorithm that does no memoization to the number of state expansions performed by an algo-

⁴ Note that, whereas failure states are such due to their features and the probability that they are reached, a state is “failure equivalent” with respect to the set of action transitions. That is, if an agent is given new capabilities, it might be able to preempt a transition to failure that it previously could not, and then a state that was failure equivalent in the context of one knowledge base is not, given a different knowledge base.

rithm that memoizes nogoods, will be on the order of:

$$\frac{|S| + \frac{|D|}{|B'|}|B|}{|S| + |B|} \equiv \frac{|D|}{|B'|}$$

We have tested this hypothesis and confirmed that these conditions are indeed necessary for speedup. They are not sufficient, however, to ensure substantial speedup because the algorithm is extremely sensitive to the structure of the reachability space⁵.

Memoizing nogoods obviously incurs a space penalty, which can be rather large in some cases. One approach to reducing the space penalty is to have a fixed size nogood list, and replace old, rarely-used nogoods with new ones as the list fills up.

We now present some experimental data, which was obtained by generating 150 random knowledge bases (as described before) and running two reachability space search algorithms on them, where one memorized nogoods, and the other did not. Both algorithms used dependency directed backtracking. Figure 9 gives a plot of the number of states expanded by an algorithm that performs no memoization vs. the number of states expanded by a nogood-memoizing algorithm.

Figure 9 No memoization vs. nogood memoization.
The line is $y=x$ (no speedup).

There are a few things about Figure 9 that are worth noting. Firstly, one can observe that there is a plenitude of solutions that require a small number of state expansions by both algorithms. This is due to the nature of the experiment and the way the random knowledge bases were generated. This shows that the majority of the problems that we have generated were either easily solvable ($|S| \gg |D|$) or it could be easily determined that no solution exists ($|B| \sim |F|$, $|D| \sim |S|$). However, as we get into the more difficult cases, where the answer to a problem is not immediately obvious, we begin to see the usefulness of nogood memoization.

8. SUMMARY AND FUTURE WORK

We have described the problem of searching for a satisficing policy for resource-limited agents in dynamic environments, and have presented an algorithm for performing such a search. We have also shown how we can exploit some properties of the search space to improve the efficiency of the search. Our analysis of these techniques explains in a qualitative way the conditions un-

der which they can be expected to make a difference, and our preliminary experimental results support these claims.

We have demonstrated that dependency-directed backtracking usually outperforms chronological backtracking by a significant margin. However, it has its own limitations. For example, it might take dependency-directed backtracking a long time to get to the root of the problem if it occurred in an ancestor state that is much higher up the tree. In that case, it might be possible to use a more informed heuristic to make a better estimate of where the root of the problem is, and to backtrack there right away.

To date, we have made a number of limiting assumptions in this work. We have assumed that temporal transitions have constant probabilities. A number of interesting complications arise if we allow non-constant (and non-monotonic) probability distributions.

In this work, when we talk about nogood subgraphs, we only consider nogoods that are due to a failure to find a satisficing reachability graph. Actually, a subgraph can be labeled as a nogood if it is not schedulable due to the agent's resource limitations, and might lead to further pruning of the search space.

9. ACKNOWLEDGMENTS

This material is based upon work supported by DARPA/ITO and the Air Force Research Laboratory under contract F30602-00-C-0017 as a subcontract through Honeywell laboratories. The authors would like to thank their colleagues Haksun Li and Kang Shin for their valuable contributions to this work.

10. REFERENCES

- [1] Åström, K.J.: *Optimal control of Markov decision processes with incomplete state estimation*. *J. Math. Anal. Appl.*, (1965)10, 174-205
- [2] Atkins, E.: *Plan Generation and Hard Real-Time Execution with Application to Safe, Autonomous Flight*. Ph.D. diss., Dept. of Electrical Engineering and Comp. Science, The Univ. of Michigan (1999)
- [3] Boutilier C., Dean T., Hanks S.: *Decision-Theoretic Planning: Structural Assumptions and Computational Leverage*, *Journal of Artificial Intelligence Research* 1 (1999), 1-94
- [4] Dechter R.: *Learning while searching in constraint satisfaction problems*, *Proceedings AAAI'86* (1986)
- [5] Ginsberg M.: *Dynamic backtracking*, *Journal of Artificial Intelligence Research* 1 (1993), 25-46
- [6] Li H., Atkins E., Durfee E., Shin K.: *Practical State Probability Approximation for a Resource-Limited Real-Time Agent*. *Proceedings of the IJCAI-01 Workshop on Planning with Resources*, (2001)
- [7] Musliner D., *CIRCA: The Cooperative Intelligent Real-Time Control Architecture*, PhD dissertation, Computer and Engineering in the University of Michigan (1993)
- [8] Musliner D., Durfee E., Shin K. *CIRCA: A Cooperative Intelligent Real-Time Control Architecture*, *IEEE Transactions on Systems, Man, and Cybernetics* (1993)
- [9] Musliner D., Durfee E., Shin K. *World Modeling for the Dynamic Construction of Real-Time Control Plans*. *Art Intelligence* pp 83-127 (1995)
- [10] Richard T., Richard B.: *Nogood learning for Constraint Satisfaction*, IC Parc Imperial College London
- [11] Russell S., Norvig P.: *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ (1995)
- [12] Schiex T., Verfaillie G.: *Nogood Recording for Static and Dynamic Constraint Satisfaction Problems*, *Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence* (1993)
- [13] Smallwood R., Sondik E.J.: *The optimal control of partially observable Markov processes over a finite horizon*. (1973)
- [14] Zilberstein S, Russell, S.J. *Anytime sensing, planning and action: A practical model for robot control*. In *Proc. 19th International Joint Conference on Artificial Intelligence*, Chambéry, France (1993)

⁵ For analysis and experiments regarding the effects of reachable state space structure on nogood memoization efficacy, see http://www.eecs.umich.edu/~ddolgov/papers/direct_study.pdf