Extending Search for Hybrid Systems: Action Timing

Todd W. Neller*
Knowledge Systems Laboratory, Stanford University
neller@ksl.stanford.edu

Abstract

Artificial Intelligence tree search algorithms are designed for discrete systems. In order to extend tree search algorithms for hybrid systems, there are two new decisions the search algorithm must make: choice of actions from a continuum, and choice of action timing. In this paper, we focus on principled ways to choose when to branch the search tree.

We introduce anytime, depth-first, iterative-refinement algorithms for satisficing and real-time decision-theoretic goals. Further, we discuss iterative adjustment techniques. Finally, we briefly discuss the use of information-based optimization for action timing choices.

1 Introduction

Discrete state-space search techniques have been central to problem solving in computer science and operations research for decades. Such techniques can be thought of as optimizing action sequences towards achieving some goal state while minimizing a cost function. While this general problem description is relevant to much of discrete/hybrid control, most of the search techniques of artificial intelligence have not been applied to control problems. Optimal control has made creative use of many different forms of optimization, so one might expect that AI search techniques should have found its place alongside of global optimization, (non-)linear programming, and dynamic programming in the control engineer's essential problem solving toolset. Curiously, this has not been the case.

Perhaps the most important reasons that many state-space search techniques have not been applied to control problems are that (1) the computational complexity of search severely restricts the class of problems to which it can be applied, and (2) it is not obvious how one might reformulate many control problems such that they fall within that class. Regarding (1), most nodes examined by searches are irrelevant. Human game players focus on relevant lines of play. We desire efficient methods for metalevel relevance reasoning for search.

Regarding (2), extending discrete search to hybrid system search introduces two new decisions in optimization: action discretization and action timing discretization. In this paper we choose to address the latter decision: How could a search algorithm choose when to branch the search tree and consider possible actions? We will thus assume that continuous action spaces are already discretized. From the perspective of the search algorithm, action discretizations are static, i.e. the search cannot affect the discretization. However, from the perspective of the search algorithm, action timing discretizations are dynamic, i.e. branching points are chosen by the search algorithm. For this reason, we will call such searches "SADAT searches" as they have Static Action and Dynamic Action Timing discretization.

Formally, a SADAT search is defined as a 6-tuple

$$\{S, s_0, t_0, A, l, d\}$$

where

- S is the state space,
- $s_0 \in S$ is the initial state,
- $t_0 \in \Re^+$ is the initial time,
- A is the discrete action space, with each action $a : S \to S \times \Re$ a mapping from state to resulting state and the utility/cost of the action,
- $l : S \to \{a_1, \ldots, a_n\} \in A$ is a function mapping state to a finite set of legal actions which may be executed in that state,
- $d : S \times \Re^+ \to S \times \Re$ is a function mapping a state and time delay to the resulting state and the utility/cost of the trajectory segment.

The total utility of a finite trajectory is computed at the sum of the trajectory action and segment utilities. In this paper, we assume a limited time horizon t_f for search. Furthermore, the algorithms seek to iteratively improve a trajectory from t_0 to t_f, so we assume the existence of default actions. Thus we assume that we are given $l_0 : S \to a_i \in \{a_1, \ldots, a_n\}$, a function mapping state to a default action which would be executed

* This work was supported by the Stanford Lieberman Fellowship, and NASA Grant NAG2-1337. Author's address: Knowledge Systems Laboratory, Gates Building 2A, Stanford University, Stanford CA 94305-9020.

in that state and time. We require that d evolves the system according to l_0. Thus, in this paper we consider time-bounded SADAT searches with default actions: $\{S, s_0, t_0, t_f, A, l, l_0, d\}$

2 Global branching decisions: depth first

In most discrete searches, it is not an option to search all possible branches of the tree to all depths. The first question in search algorithm design concerns how one will "grow" the tree. The simplest approaches predefine the search tree to be the set of all nodes to a given cutoff depth. One difficulty of such approaches is that one may not know a proper cutoff depth. For this reason, Slate and Atkin developed iterative-deepening search (Slate & Atkin 1977) which tries successively deeper and deeper depth-first searches while time permits.

We face an analogous problem when seeking to perform a time-bounded SASAT search. Although we have a time bounds, we do not know how to branch the tree within those bounds. In this section, we make a restrictive assumption: Every branch of the tree shares the same branching times. In other words, we make global action timing decisions for search. As with cutoff depth in discrete searches, it is not clear how many or which branching points we should consider within the time bound. In addressing how many, we take the iterative-deepening idea, and reapply it in this domain as *iterative-refinement*: For each iteration, we search our tree using a list of decision times (called a tlist), choose an additional decision time expected to be most beneficial, and repeat as time permits. A sketch of this top level procedure for such a search is given in Algorithm 1.

Algorithm 1 Top level of Iterative Refinement Search

```
procedure IterativeRefinement(node)
  bestAction := node.defaultAction()
  tlist := {}
  while not terminationTest() do
    t_new := max_{t∈[t_0,t_f]}(ExpUtilOfDFS(tlist ∪ {t} ))
    tlist := tlist ∪ {t_new}
    bestAction := DFS(node, tlist)
```

ExpUtilOfDFS is a function which computes the expected utility of the action that would be recommended if we were to perform depth-first search according to the given tlist. In other words, we wish to know which additional branching point or decision time would most likely greatly benefit our overall decision. Discussion of how one might form such expectations is deferred to Section 2.3. Note that each successive tree subsumes the previous tree, so bestAction will monotonically increase in utility and the algorithm is anytime.

2.1 Satisficing approach

Suppose our goal is to satisfice, i.e. we want to find an action which will reach a given utility threshold U_{goal}. The anytime top level procedure for such a search is given in Algorithm 2. Note that here we're not interested in picking a new time point which maximizes expected utility. Rather, we wish to pick a time point which has the greatest probability of satisficing. In (Neller 1998), we reformulated a hybrid system initial safety refutation as a satisficing optimization and introduced a form of information-based optimization which was well suited to that task. We propose that it would be beneficial to this satisficing task as well.

Algorithm 2 Top level of Satisficing Iterative Refinement Search

```
procedure SatisficingIterativeRefinement(node, U_goal)
  bestAction := node.defaultAction()
  tlist := {}
  while not satisfices(bestAction) do
    t_new := max_{t∈[t_0,t_f]}(Prob(ExpUtilOfDFS(tlist ∪ {t})
                               ≥ U_goal))
    tlist := tlist ∪ {t_new}
    bestAction := DFS(node, tlist)
```

2.2 Decision-theoretic approach

Let us now consider the real-time use of search for hybrid system control. One would expect such search to be most successfully applied to problems which share two types of characteristics, environmental characteristics and informational/topological search characteristics. The environment must be accessible, deterministic, nonepisodic, static (only effected by the controller), and SADAT hybrid. In sampling possible system trajectories from the current state, the search tree formed must contain sufficient information within a limited depth, given a low branching factor, to indicate intelligent action.

Let us assume that our control architecture includes a module which is continually performing search to inform an intelligent control action. A simple approach might place a fixed time bound on each search and periodically alternate between search and action. A more interesting decision-theoretic approach would repeatedly make a principled decision whether to search further, or commit to the action recommended by search thus far and begin searching for the best next action. The anytime top level procedure for such a search is given in Algorithm 3.

To compute ExpectedGain, we subtract utility of the current bestAction[1] and the *expected computational time cost* from the expected utility of the future bestAction[2] gained by additional search. As before,

[1]assuming it is committed to at the current time

[2]assuming it is committed to after additional search

we defer discussion of computing expectations to Section 2.3. Utility of the current bestAction should be stored from the previous search. Computing the "cost" of computation time is a far more interesting problem, and is one of the key pursuits of research in rational metareasoning. If we have a measure of expected value of our computation time, and we can further estimate the computational time cost of further search, then we have a rational basis for search termination: when the expected action utility gained by computation is outweighed by expected utility lost through computational time cost.

That computation has a time cost is obvious for fast real-time applications. For example, the utility of deciding whether or not to take the next exit on the highway drops to zero when one passes the exit. However, computation has a cost for any application. The computer performing the computation has a real cost in terms of its base hardware and software, support staff, etc. We also might base computational "cost" on a statistical expectation of average utility of computation over time. It would be desirable to have some rational basis for distribution of computational resources among a series of searches over time.

One approach is to uniformly distribute the computational time cost over the relevant period of usage. Another approach is to empirically determine an appropriate measure of time cost through experimentation. Yet another approach is to apply an intuitive problem-specific measure, as demonstrated in the context of playing Othello, where Russell and Wefald "set the time cost to be an appropriately parameterized inverse function of the average time remaining per move"(Russell & Wefald 1991, p. 95). Of these three, the statistical empirical approach is appealing in the context of controlling hybrid systems in real-time using search. Since we assume that we are given a simulation engine, it would make sense to simulate the working of such a system and let experimental data from search form the basis for future search expectations.

Algorithm 3 Top level of Decision-Theoretic Iterative Refinement Search

procedure DTIterativeRefinement(node)
  bestAction := node.defaultAction()
  tlist := {}
  while (**true**) **do**
    $\mathrm{t_{new}} := \max_{t\in[t_0,t_f]}(\mathrm{ExpectedGain}(\mathrm{tlist}\cup\{t\}))$
    if $(\mathrm{ExpectedGain}(\mathrm{tlist}\cup\{t_{\mathrm{new}}\})\leq 0)$ **break**
    $\mathrm{tlist} := \mathrm{tlist}\cup\{t_{\mathrm{new}}\}$
    bestAction := DFS(node, tlist)

2.3 Computing expectations

Until this point, we have deferred discussion of how one might compute expectations of search utility. Indeed such discussion is difficult in a general context. Specific problems can often suggest specific heuristics or "rules of thumb" for forming expectations.

Algorithm 4 Top level of Decision-Theoretic Iterative Adjustment Search

procedure DTIterativeAdjustment(node)
  bestAction := node.defaultAction()
  tlist := {}
  while (**true**) **do**
    $\mathrm{t_{new}} := \max_{t\in[t_0,t_f]}(\mathrm{ExpectedGain}(\mathrm{tlist}\cup\{t\}))$
    $\mathrm{tlist_1} := \mathrm{tlist}\cup\{t\}$
    $[\mathrm{t_{del}},\mathrm{t_{add}}] :=$
      $\max_{t_{\mathrm{del}}\in\mathrm{tlist},t_{\mathrm{add}}\in[t_0,t_f]}(\mathrm{ExpectedGain}((\mathrm{tlist}-\{t_{\mathrm{del}}\})\cup\{t_{\mathrm{add}}\}))$
    $\mathrm{tlist_2} := (\mathrm{tlist}-\{t_{\mathrm{del}}\})\cup\{t_{\mathrm{add}}\}$
    if $(\mathrm{ExpectedGain}(\mathrm{tlist_1}) > \mathrm{ExpectedGain}(\mathrm{tlist_2}))$
      $\mathrm{tlist} := \mathrm{tlist_1}$
    else $\mathrm{tlist} := \mathrm{tlist_2}$
    if $(\mathrm{ExpectedGain}(\mathrm{tlist}))\leq 0)$ **break**
    bestAction := maxUtility(bestAction, DFS(node, tlist))

In the context of general use, it makes sense to first look at what information we can readily gain from the computation we perform. Specifically, we can perform additional computation during depth-first search to assess the contribution of any subset of the tlist to the utility of the whole. Since we have default actions, we may perform simple bookkeeping during depth-first search such that we in parallel assess the utility the action recommended by a subtree defined by a tlist subset. Specifically, for each time on the tlist, we could compute the effect on the decision utility if it were not on the tlist. Doing this for each time point multiplies the complexity of each iteration by the length of the tlist.

In computing the relative utility gain of a tlist time relative to all others, we gain simple but useful information for metalevel search control. First, we can a clearer global picture of how frequency of decisions affects overall utility within a fixed time frame. After many successive training runs, one would gain a statistical picture of the relationship between decision frequency and utility gain. As a decision time point approaches another, the utility gain of that decision time point will approach zero. Simulated training runs can provide a statistical basis for global expectations of where and how much utility can be gained between two existing time points.

Furthermore, such information combined with the data from a single iterative refinement/adjustment run can allow more accurate interpolation of values. Given that the relationship between utility gain and frequency of action will vary depending on the problem and what parts of the problem's state space we're dealing with, we can combine global understanding of how utility gain tends to decrease with frequency with local measurements to give a better estimate of expected utility gain.

If one is willing to face the "curse of dimensionality", one can do even better. Consider the task of navigating a robot through a narrow passage. Clearly, the utility gain for increased frequency of action is higher within the narrow passage. By conditioning expectations on the state space itself, we gain richer information at the cost of the exponential dimensional complexity of state-space-based methods.

In short, many statistical methods may be brought to bear on the problem of forming expectations for these searches. Since excessive complexity in metalevel reasoning can easily defeat the efficiency gains purposed by such reasoning, such methods must necessarily be simple. The key observation to be made here is that expensive base-level computation merits intelligent metalevel computation to reduce the need for base-level computation. If the information we can gain easily from search can help us form more accurate expectations of utility of possible future searches, then such additional metalevel computation is justified.

3 Discussion and Conclusion

In this paper, we have given a top level view of a few approaches to extending search for SADAT hybrid hystems. In all cases we assume global uniformity of action timing along all branches, and discuss the application of statistics and information-based optimization to such timing choices.

Although depth-first search has the advantage of low computational space complexity, we need not necessarily take a depth-first approach. Best-first approaches along the same lines are possible. In order to improve best-first computational time complexity, Baum and Smith (Baum & Smith 1997) seek to acheive complexity proportional to the size of the final tree by employing a "gulp trick", whereby they expand the tree in a series of "gulps". A gulp is a fraction of the best possible tree-expansions, so each gulp is linear in the size of the tree. The difficulty of applying such gulps in the context of SADAT search is that we need to make sure each pair of expansions within a single gulp are (1) independent with respect to time cost estimation, and (2) independent with respect to post-search expected utility estimation. The first is easily achieved by forcing all expansions to introduce single branches to the time horizon, i.e. for all new (non-default) branches, no further branching occurs between the expanded node and the time horizon t_f. The second is more difficult given that time-dependent utility is computed with respect to a single computational action, so such utilities do not sum. A decision-theoretic basis for deciding whether or not to perform a gulp is an open research question.

References

Baum, E., and Smith, W. 1997. A bayesian approach to relevance in game-playing. *Artificial Intelligence* 97(1–2):195–242.

Neller, T. W. 1998. Information-based optimization approaches to dynamical system safety verification. In Henzinger, T. A., and Sastry, S., eds., *LNCS 1386: Hybrid Systems: computation and control, First International Workshop, HSCC'98, Proceedings.* Berlin: Springer. 346–359.

Russell, S., and Wefald, E. 1991. *Do the Right Thing: studies in limited rationality.* Cambridge, MA, USA: MIT Press.

Slate, D., and Atkin, L. 1977. Chess 4.5 - the northwestern university chess program. In Frey, P., ed., *Chess Skill in Man and Machine.* Berlin: Springer. 82–118.