Reaction-First Search

Article · May 2003

Source: CiteSeer

CITATIONS
19

READS
50

4 authors, including:

Mark Drummond
Apple Inc.
66 PUBLICATIONS **1,234** CITATIONS
SEE PROFILE

John Bresina
NASA
93 PUBLICATIONS **1,773** CITATIONS
SEE PROFILE

Rich Levinson
NASA
24 PUBLICATIONS **468** CITATIONS
SEE PROFILE

Reaction-First Search

Mark Drummond Sterling Software
Keith Swanson NASA
John Bresina Sterling Software
Richard Levinson Recom Technologies

AI Research Branch, Mail Stop: 269-2
NASA Ames Research Center
Moffett Field, CA 94035-1000 USA

Abstract

This paper presents Reaction-First Search (RFS), an incremental planning algorithm that produces plans for execution by a reactive system. The reactive system is independently competent in the sense that it is able to produce behavior in its environment without a plan. This reactive system has some prior probability of satisfying a given goal, and plans found by RFS serve to increase the probability of goal satisfaction above that prior. RFS is usefully incremental in that any partial plan it produces can be executed by the reactive system. While any given partial plan may or may not increase the reactive system's goal satisfaction probability, the expected performance of RFS is to monotonically increase the goal satisfaction probability as a function of time spent planning.

1 Introduction

Most general-purpose planning algorithms cannot be used incrementally. Since initial steps in a partial plan are not necessarily part of a solution, planning algorithms generally do not produce any plan for execution until a complete plan has been found. Such non-incremental algorithms fail to address the needs of an agent that must act before a complete plan is available.

This paper presents a general-purpose planning algorithm that *is* incremental. The algorithm, Reaction-First Search (RFS), generates partial plans in service of a given goal. RFS is used by a situated planner to provide guidance to an independently competent reactive system. Without a plan, the reactive system executes a human-provided program, and by so doing, stands some chance of producing a behavior that satisfies a given goal. The partial plans found by RFS help the reactive system handle difficult situations that were not foreseen by the programmer. RFS reasons about the reactive system's behavior, giving it plans to help satisfy the goal. We cannot guarantee that the single plan found by RFS after an arbitrary amount of computation increases the reactive system's chances of satisfying the goal. However, averaged over a number of trials, we can guarantee that RFS monotonically increases the reactive system's goal satisfaction probability as a function of computation time. Thus, the *expected* performance of RFS is to monotonically improve the reactive system's goal satisfaction probability – it is this expected improvement property that makes RFS usefully incremental.

2 Background

A situated agent often needs to take action within bounded time. To facilitate this, we suggest that whenever possible, a *prior reactive competence* should be hand-coded by a programmer. This approach avoids expensive automatic planning whenever the prior competence is capable of solving a given problem. Our general architecture is based on this idea and includes two connected components: a *planner* and a *reactor*. For details see Bresina and Drummond [1990], Drummond, *et al.* [1991], and Bresina, *et al.*, [*in press*].

Bresina and Drummond [1990] introduced the *principle of independent competence* to ensure that the reactor is able to take action *with* or *without* a plan. When the reactor is told to act, it executes the currently available plan and then falls back on a reactive program defined by the programmer. If a complete plan is available then the execution of that plan will cause the reactor to solve the problem. If absolutely no plan is available then the reactor executes only the programmer-provided reactive program. When there is a *partial* plan available, the plan takes the reactor part of the way, and the reactive program takes over at the plan's end.

These intermediate cases lead to the following problem. If the reactor executes a partial plan before that plan is known to lead to a solution, how can we be sure that this will not impair the reactor's performance? Indeed, the entire reason that *search* is an essential part of problem-solving is that there is no purely "local" way to guarantee that any given partial plan can be extended to a solution. This is the problem that RFS solves. The expected performance of RFS is to monotonically improve the goal satisfaction probability of the reactor, no matter how much time is spent searching. RFS accomplishes this by reasoning about possible behaviors of the reactor and constraining them as appropriate. RFS can be used incrementally, releasing partial plans computed in the time available.

3 Basic definitions

This section defines the terms used throughout the rest of the paper. Most of these terms are reasonably standard. The planner is covered first, the reactor second.

The planner

Briefly, a *state* is a structure that describes a situation in the application domain. An *operator* describes an executable action and is considered *enabled* in a state when the action denoted by the operator can be executed in the corresponding situation. An operator can be *applied* to a state in which it is enabled, producing a successor state that describes the effects of the action denoted by the operator. The STRIPS operator language [Fikes and Nilsson, 1971] is an example of this but is actually a special case; more general languages are possible. A *trajectory* is a sequence of operator applications, and any given trajectory models one possible behavior of the reactor.

A *search tree* is a tree of trajectories rooted at a common state. A *level* in a search tree is a set of states that are the same distance from the root. A *goal* is a function[1] that maps a trajectory into the set $\{t, f\}$. If the goal function returns t for a given trajectory, then the trajectory *satisfies* the goal. A *problem* is a specific initial state, a set of operators (defined in terms of operator enablement and application functions), and a goal.

A *prefix partial plan* (or just *prefix plan*) for a given problem is a trajectory constructed from the problem's operators, rooted in the problem's initial state. Such a plan is a "prefix" since it specifies the first few actions in a trajectory which may or may not lead to the eventual satisfaction of the problem's goal. A *solution plan* for a given problem is a trajectory that satisfies the goal.

The reactor

A *policy* is a function that maps a state into a set of enabled operators and is provided by a programmer to specify how the reactor should behave in a given set of situations. Thus, a policy defines a prior reactive competence for the reactor. For each possible state S and for a given policy po, if $|\, po\,(S)\,| \leq 1$, we say that the policy is *deterministic*; otherwise, if there exists a possible state S for which $|\, po\,(S)\,| > 1$, we say that the policy is *nondeterministic*. Since a policy is used to define the reactor's prior competence, it is largely counterproductive to allow the policy to encode a general purpose problem solving mechanism (a planner, for instance). Thus, we insist that any given policy be computable within time and space that are bounded by constants.

A policy is executed by the reactor as follows. First, a state S is created to describe the agent's current situation. Next, the policy is evaluated on S, producing a disjunctive set of recommended operators, $\{O_1, O_2, \ldots, O_n\}$. If this set is empty then the reactor halts. Otherwise, one particular O_i is randomly selected and executed.

In terms of the planner's search space, the *subtree defined by the policy from a state S* is the tree rooted at S defined by recursive application of all policy-recommended operators. Thus, the states reachable from S under the policy are a subset of all possible states reachable from S. A *policy state* is simply a state for which the policy returns a non-empty set of operators.

We measure the reactor's *prior competence* for a given problem instance as the probability that the reactor can satisfy the goal using only the programmer-defined policy. This probability can be explicitly measured, for example, by repeatedly placing the reactor in the problem's initial state and allowing it to attempt to satisfy the goal by executing its policy. The number of samples required to obtain high statistical confidence is a function of the nondeterminism in the policy. If the policy is deterministic then a single run suffices. If there is a large degree of nondeterminism in the policy, then a large number of runs might be required.

The plan execution model of the reactor is as follows. At each point in time there is a prefix plan defined by the trajectory that terminates in the planner's "current" search state. If the reactor is told to act, it first executes this prefix plan and then executes its policy until the goal is satisfied or until the policy returns the empty set. If the current prefix plan causes the reactor to produce a behavior that satisfies the goal, then the policy is not executed. If the current prefix plan is null, then the policy is executed immediately.

This paper makes some assumptions regarding the execution environment. We assume that no errors occur during plan execution, and we also assume that sensing is free and provides enough information to define a search tree that includes a solution (if one exists). These assumptions merely simplify the discussion; they are easily relaxed, but space precludes a detailed discussion. Finally, note that RFS does not address the question of how long to plan – it simply uses whatever computation time is available.

4 The problem and a solution

Assume that a timer, t, is started when a problem is presented to the planner and reactor. At $t = 0$, the reactor can be told to run and, by executing its policy, stands some chance of solving the problem. If the reactor were to sit idle until the planner released a prefix plan for execution at $t = 1$, how would the reactor's performance be affected? Again, what if the reactor were to sit idle until the planner released a prefix plan at $t = 2$? RFS guarantees that whenever the planner is stopped, $t = 1, 2, 3, \ldots$, the expected performance of the reactor is never worse than its performance at $t = 0$. Indeed, as we show below, the reactor's performance can be significantly improved.

The three principles of Reaction-First Search

RFS is a randomized search algorithm that requires the planner to first reason about what the reactor would do without a plan. RFS uses the policy to define a search-space bias, exploring the policy-defined subtree before any other part of the search space is examined. The policy defines a set of possible reactor behaviors. The planner, guided by the policy, searches through the set

[1] We disallow goal functions that permit the growth of infinite length trajectories.

of trajectories that model those behaviors, looking for one that satisfies the given goal.

There are three principles that differentiate RFS from traditional state-space search. These principles, together with a definition of state-space search, constitute a definition of RFS. The three principles of RFS are:

1. All states in the policy-defined subtree must be expanded before all other states in the search space;
2. After selecting a level in the search tree by any means, a policy state in that level must be *randomly* chosen for expansion;
3. In any given policy state, policy-recommended operators must be *randomly* chosen for application.

These three principles are all that is required by RFS; any state-space search procedure that follows them faithfully implements our algorithm and consequently has expected performance that guarantees monotonic improvement of the reactor's goal satisfaction probability. The proof of monotonic improvement [Drummond, *et al.*, 1992] hinges on the observation that the probability of goal satisfaction at each level of the policy-defined subtree is the same. RFS samples from these levels in a randomized manner, and this ensures that the probability of goal satisfaction obtained at each point in time is, on average, always greater or equal to that obtained by the policy at time zero.[2] The incrementality of RFS comes from the fact that it can be stopped whenever it is about to expand a state: the trajectory that connects the root to this current state defines a plan that can be executed by the reactor.

The three principles of RFS are not as restrictive as they might first appear. In fact, it is easy to implement RFS so that it explores the policy subtree in a variety of ways. We have experimented with standard depth-first and breadth-first search, as well as iterative sampling [Minton, *et al.*, 1992; Langley, 1992]. Other implementations are also possible; for instance, iterative deepening [Korf, 1985] and iterative broadening [Ginsberg and Harvey, 1990]. One need only bias the search according to the three principles of RFS.

Completeness and monotonic improvement

RFS first explores the subtree defined by the policy, starting with the root state. If the search strategy used for this exploration is complete (for instance, depth-first search with chronological backtracking), and if a solution exists in this subtree, RFS will find it. If a non-systematic search strategy such as iterative sampling is used then RFS is only asymptotically complete. In essence, with respect to the policy subtree, RFS is only as complete as the underlying search strategy.

If RFS discovers that there is no solution in the policy subtree, it selects an arbitrary open state and applies an arbitrary enabled operator, creating a new search state. The choice of which particular open state and enabled operator to choose is not dictated by RFS; the algorithm works no matter how the state and operator are chosen. The only requirement, and this is important, is that the subtree rooted at the new state must now be explored in the manner dictated by RFS. By repeatedly generating new open states in this manner, RFS is clearly complete, again assuming that a complete search strategy is used to explore each policy-defined subtree.

Our hypothesis regarding Reaction-First Search

While RFS can be expected to monotonically improve the performance of the reactor, there is clearly a simpler approach: don't let the planner release prefix plans! With the principle of independent competence, the reactor's goal satisfaction probability would hold steady at the point determined by its policy until sufficient time had elapsed to allow a complete solution to be found. We call this alternate approach *non-incremental*, since it does not issue preliminary advice to the reactor.

We hypothesize that RFS' release of prefix plans can have an advantage over the non-incremental approach. Specifically, we hypothesize that the partial plans found by RFS can help the reactor avoid dead-ends that might otherwise trap its policy. A dead-end for the policy is a state for which the policy returns the empty set and which fails to satisfy the goal. A *critical choice point* [Drummond, 1989] is a state from which there is an action that leads to possible success and from which there is also an action that leads to necessary failure. In the context of RFS, a critical choice point is a state from which the probability of goal satisfaction (under the policy) is non-zero, but from which there exists a policy-recommended action that leads to a state from which the goal satisfaction probability *is* zero. We hypothesize that RFS can have an advantage over the non-incremental approach since its prefix plans can provide guidance to the reactor at these critical choice points and thus help the reactor avoid dead-ends during execution.

Figure 1 sketches our hypothesis regarding the general relationship between the non-incremental approach and RFS. Assume that the reactor's prior probability of goal satisfaction is P. Thus, at time 0, there is a probability, P, that the reactor will satisfy the given goal. The non-incremental approach ensures that the reactor's goal satisfaction probability holds steady at P until the earliest time at which a solution is found. This point is indicated on the computation time axis by "ES" (earliest solution). The average performance of the reactor will then increase until it reaches a maximum at the planner's latest solution ("LS") time, at which point the planner always has a solution, if one exists. The precise rate at which the reactor's average performance increases between ES and LS is a function of the search strategy and the search space.

The important aspect of our hypothesis is the performance difference it predicts. The reactor's performance profile under RFS initially tracks that of the non-incremental approach, starting with a value of P at time 0. Whenever RFS identifies a critical choice point, the expected goal satisfaction probability should increase (there are two such increases sketched in the figure). The

[2] Please note that RFS does not actually calculate the policy's goal satisfaction probability. The monotonic improvement property is an objective statement about RFS: the algorithm exhibits expected monotonic improvement without having to explicitly calculate any probabilities.

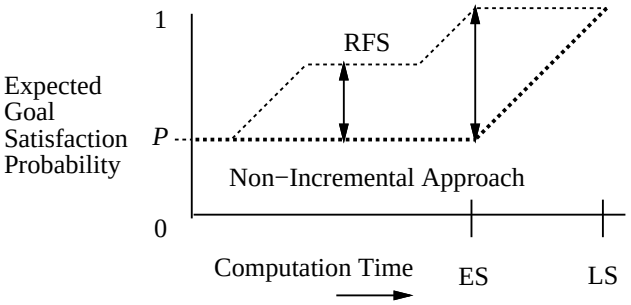

Figure 1: Hypothesis: Non-Incremental *vs* RFS.

precise size and shape of the difference is again a function of the underlying search strategy and search space. It is important to note, however, that the expected performance of RFS is *never* worse than that obtained by the non-incremental approach and, according to our hypothesis, can sometimes be significantly better. The next section experimentally explores this hypothesis.

5 Empirical results

We have implemented RFS on a telescope scheduling problem [Drummond, *et al.*, *in press*], where the reactive policy is defined by a dispatch scheduler written by the company that builds the telescope controller. While we have a working version of RFS for this scheduling problem, we have not yet defined and carried out any experiments. Thus, in this paper, we present the results from a detailed study of RFS at work in a simulated domain, the NASA TileWorld [Philips and Bresina, 1991]. The goal of the experiment reported here is to evaluate the hypothesis outlined in the previous section.

The problem class for this experiment is couched in terms of a two-dimensional grid of cells. The task is to retrieve a tile from one room, B, and to carry it back to a particular location in another room, A. As shown in Figure 2, room A occupies the lower left portion of the grid and room B occupies the upper right portion. The goal is satisfied if the agent, while grasping the tile, is in the lower-left corner cell of room A. Different problem instances in this class vary in terms of the obstacles present *in* the rooms and *between* the rooms. For all problem instances considered, the agent is initially in the lower-left corner of room A and the tile to be retrieved is initially in the top-right corner of room B. The figure shows this common problem structure, illustrating all prior information available to the (human) designer of the reactor's policy.

The policy we devised for these experiments is a simple hill-climber that uses a traditional Manhattan distance calculation. The policy operates in three stages: first, move the agent from room A to room B; second, grasp the tile; third, move the agent back. The movement from room A to room B is itself implemented in three stages: first, move the agent from its initial location to one of the doors of room A; second, move the agent to one of the doors of room B; third, move the agent next to the tile. Any one of these subtrips can be halted by entrapment in a dead-end, since the policy uses Manhattan distance to

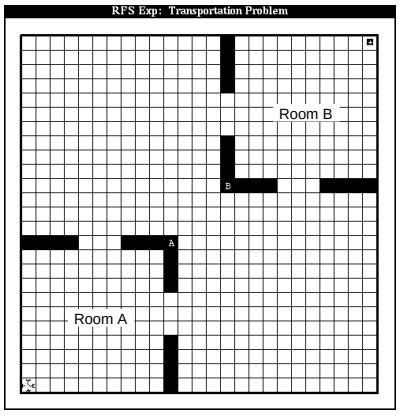

Figure 2: All prior information.

suggest moves, and while this calculation provides useful guidance, it is not guaranteed to find a solution. The policy returns *all* moves suggested by the Manhattan distance calculation and hence is nondeterministic. Once the agent has grasped the tile, the return trip is also implemented in three stages. As for the outward journey, each of these stages can be halted by entrapment in a dead-end. Since the policy is not capable of physically backtracking there is no guarantee that it will solve the problem. Even so, it works well for many problems and satisfies the requirement of computability in time and space that are bounded by constants.

The dependent variable for this experiment is reactor performance, measured as goal satisfaction probability, and the independent variable is planning time. Intuitively, planning time should be measured in some metric units; however, since we carried out this experiment in Common Lisp, running on a Unix system, elapsed time is only weakly correlated with the amount of time actually spent planning. To overcome this, we observed that the time spent by the planner applying an operator to a state can be bounded by a constant. This is true since the evaluation of the policy takes constant time, as do all other operations involved with expanding a single state. While the precise value of this constant depends on a variety of factors, the important aspect is that a bounded amount of time is spent applying an operator to a state. We use the term *step* to refer to such an operator application. Thus, instead of measuring actual metric time we measure the number of planner steps taken.

To facilitate data collection, we designed an algorithm for calculating the probability that the policy will lead the agent into a dead-end. The procedure accepts an initial and goal location and returns the probability that the agent will become trapped in a dead-end via the execution of the policy. This procedure allows us to determine the probability of goal satisfaction without explicitly executing the policy.

Figures 3 and 4 show some of our results, and more results are reported in Drummond, *et al.*, [1992]. The left side of each figure shows a particular problem instance, and the right side shows the reactor's goal satisfaction probability as a function of planning steps. We used a

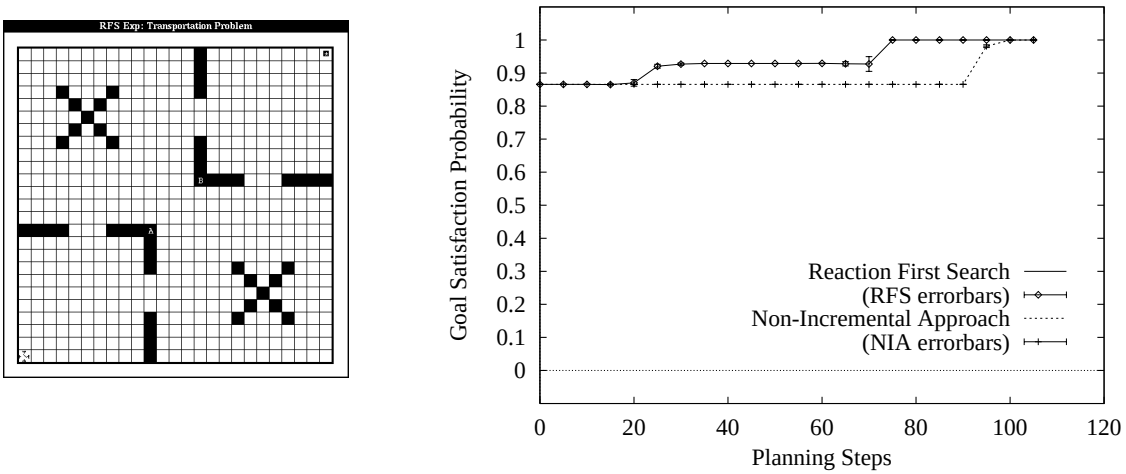

Figure 3: Problem instance and expected RFS performance.

depth-first chronological backtracking version of RFS for these experiments. Since the policy is nondeterministic, we ran RFS 500 times on each problem instance, and the graphs in each figure show the mean probability value obtained together with errorbars characterizing the 95% confidence interval.

Each problem instance is consistent with the prior information available to the programmer (Figure 2), but each adds some number of obstacles. Obstacles have the general shape of an "X" and are located either in the rooms or between the rooms. The arms of these "X"s define dead-ends that can trap the hill-climbing policy, and this is where planning can help provide guidance. Note, had this information been available in advance, we might have been able to formulate a better policy.

The graph in each figure shows the reactor's goal satisfaction probability under RFS and the same measure for the non-incremental approach discussed in section 4. Since the non-incremental approach only releases complete plans for execution, the reactor's performance remains at the level defined by the policy until the planner terminates. As we predicted, there is some variation between the earliest and latest times at which the planner terminates. More importantly, our prediction of a performance difference between RFS and the non-incremental approach is clearly demonstrated by these experimental results.

Consider Figure 4: the reactor's prior goal satisfaction probability is about 0.5, and within 10 steps the probability increases to almost 0.8. The reason for this early and significant improvement is clear from the problem instance. There are two dead-ends defined by the single "X" in room A. On its way out of the room, the agent can easily become trapped in the west or south side of the "X". When RFS gets the agent past the critical choice points that define the entries to these dead-ends, the probability of goal satisfaction increases dramatically. Of course, once past *all* dead-ends, the reactor's policy is able to solve the problem, without a plan. This is shown by the probability of goal satisfaction reaching 1.0 before the planner actually terminates.

It is interesting to note the points of greatest variance in the goal satisfaction probability. In Figure 3, for instance, the variance is greatest around 70 planner steps. This is a result of the fact that there are a number of different plans that RFS might find to move the agent back to room A. Each of these plans takes the agent past the two "X"s, but there is some variability in how close these plans come to the various critical choice points. This gives rise to some variation in the reactor's goal satisfaction probability that finally settles down once all plans guide the reactor past all critical choice points.

This experiment clearly supports the hypothesis of section 4. The results show how prefix plans released incrementally by RFS can help the reactor avoid dead-ends that might otherwise trap its policy.

6 Related work

Simmons' [1988] "Generate-Test-Debug" technique has partly inspired our definition of RFS. RFS starts with a specification of the reactor's behavior, reasons about how it may or may not satisfy the goal in a given situation, and "debugs" the behavior as appropriate. There is a clear opportunity here for more sophisticated approaches to debugging than are currently employed by RFS.

There are other systems that require a programmer to manually code a prior reactive competence in order to avoid or reduce the need for automatic planning. In this regard there is work by Beetz and McDermott [1992], Brooks [1986], Firby [1987], Kaelbling [1988], and Georgeff and Lansky [1987]. Of these, only the system of Beetz and McDermott includes a planner. This planner, described in detail by McDermott [1992], uses a number of heuristic transformations that improve a given reactive program's goal-achieving properties. We feel that it should be possible to use RFS in this transformational planner, but more work is required to demonstrate this.

There are several new architectures for the integration of plan generation and plan execution. One such architecture, RoboSoar [Laird and Rosenbloom, 1990], uses planning to resolve execution impasses, and this is similar to our view that planning is used to guide the reactor out of dead-ends defined with respect to its policy. Our

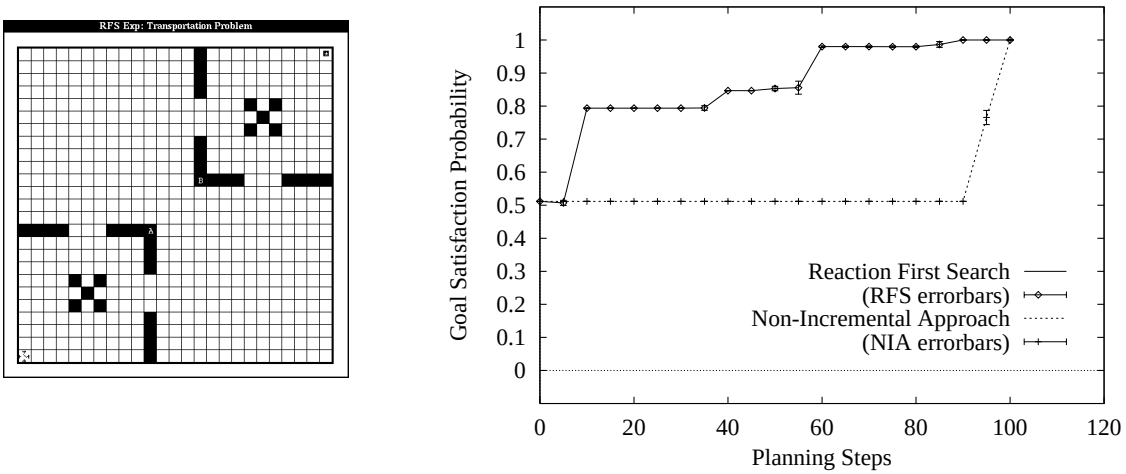

Figure 4: Problem instance and expected RFS performance.

results are quite different, however. RFS can be used incrementally, since it guarantees monotonic improvement, and no such guarantees exist for the planning component of RoboSoar. It should be easy to make the planning component of RoboSoar operate according to the three principles of RFS. If so, then RoboSoar should be able to exhibit the property of monotonic performance improvement. Lyons and Hendriks [1992] and Godefroid and Kabanza [1992] describe architectures for integrating planning and execution, both of which are concerned with incremental planning. Since these architectures use a projector–reactor model similar to our own, we hypothesize that they can be easily extended to employ RFS.

An *anytime algorithm* [Dean and Boddy, 1988] is one that can be terminated and restarted at any time, where the results computed by the algorithm improve in some "well-behaved manner" as a function of time. RFS is statistically anytime, in the sense that its expected performance satisfies the anytime algorithm requirements. However, the guarantee of monotonic improvement is stronger than that required of an anytime algorithm, and it is this guarantee that makes RFS usefully incremental. Boddy [1991] studies anytime algorithms in more detail.

In terms of general-purpose search, Georgeff's [1983] strategy-first search technique could be profitably integrated with RFS. Strategy-first search is a technique that allows for the use of "heuristic strategies" in search control. Were we to allow a policy to return a set of operator *sequences*, we would be able to implement Georgeff's strategy-first search technique as an instance of RFS. The same incremental planning result would obtain, with the only difference being the expressive power of the policy as a programming mechanism. This is an excellent area for further research.

Korf's [1990] work on RTA* was also motivated by the problem of limited computation time, based on observed drawbacks of the A* [Hart, *et al.*, 1968] and IDA* [Korf, 1985] algorithms. As Korf noted: "a related drawback of A* and IDA* is that they must search all the way to a solution before making a commitment to even the first move in the solution." [Korf, 1990, p. 191]. While RTA* *is* guaranteed to make locally optimal choices, it is *not* guaranteed to monotonically improve the performance of a plan executor. That is, RTA* cannot ensure that the prefix plans it produces over time will not adversely impact the expected performance of a system that executes those plans. This is a natural result of the fact that RTA* was not designed to find plans for an independently competent reactor. As with the planning component of RoboSoar, however, it should be relatively easy to implement a version of RTA* that adheres to the three principles of RFS. With this done, RTA* would also exhibit the monotonic performance improvement property.

7 Summary

Reaction-First Search is a general *incremental* planning algorithm that produces partial plans for execution by an independently competent reactor. The expected performance of RFS is to monotonically increase the reactor's goal satisfaction probability as a function of time spent planning. To our knowledge, RFS is the first planning algorithm that guarantees this property. While the property is statistical, this itself is rather interesting, since few average-case guarantees are made regarding planning algorithms. We have empirically supported our hypothesis that there can be significant value associated with the early release of prefix plans. Our experiments demonstrated that the prefix plans found by RFS improve the reactor's performance by helping it avoid dead-ends defined with respect to its policy.

Acknowledgments

Discussions with John Allen, Peter Cheeseman, Phil Laird, and Drew McDermott have been extremely useful. We would like to thank Peter Friedland, Rich Keller, Amy Lansky, Nathalie Mathe, Andrew Philpot, David Thompson, and Richard Washington for helpful comments on a draft of this paper. Thanks also to Carlos Salinas for help with the PostScript figures.

References

[Beetz and McDermott, 1992] Beetz, M., McDermott, D. Declarative Goals in Reactive Plans. *Proc. of First International Conference on Artificial Intelligence Planning Systems*, pp. 3–12, University of Maryland, 1992. J. Hendler, ed. Morgan Kaufman.

[Boddy, 1991] Boddy, M. Anytime Problem Solving Using Dynamic Programming. *Proc of 9th National Conference on AI*, pp. 738–743, Anaheim, CA, 1991.

[Bresina and Drummond, 1990] Bresina, J., and Drummond, M. Integrating Planning and Reaction: A Preliminary Report. *Working Notes of the 1990 AAAI Spring Symposium Series, Session on Planning in Uncertain, Unpredictable, or Changing Environments.* Reprinted as SRC TR 90-45, Systems Research Center, University of Maryland, 1990. J. Hendler, ed.

[Bresina, *et al.*, *in press*] Bresina, J., Drummond, M., and Kedar, S. Reactive, Integrated Systems Pose New Problems for Machine Learning. *Machine Learning Methods for Planning*, S. Minton, ed. Morgan-Kaufmann, *in press.*

[Brooks, 1986] Brooks, R. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, Vol. 2, No. 1, 1986.

[Dean and Boddy, 1988] Dean, T., and Boddy, M. An Analysis of Time-Dependent Planning. *In Proc. of AAAI-88*, pp. 49–54, St. Paul, MN, 1988.

[Drummond, 1989] Drummond, M. Situated Control Rules. *Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning*, pp. 103–113, Toronto, Canada, 1989.

[Drummond, *et al.*, 1991] Drummond, M., Bresina, J., & Kedar, S. The Entropy Reduction Engine: Integrating Planning, Scheduling, and Control. *SIGART bulletin*, Vol. 2, No. 4 (August), 1991. ACM Press.

[Drummond, *et al.*, 1992] Drummond, M., Levinson, R., Bresina, J., and Swanson, K. Reaction-First Search: Incremental Planning with Guaranteed Performance Improvement. NASA Technical Report TR-FIA-92-34, Code FIA, NASA Ames Research Center, Moffett Field, CA, 1992.

[Drummond, *et al.*, *in press*] Drummond, M., Swanson, K., and Bresina, J. Robust Scheduling and Execution for Automatic Telescopes. *Heuristic Scheduling Systems*, M. Fox & M. Zweben, eds. Morgan-Kaufmann, *in press.*

[Fikes and Nilsson, 1971] Fikes, R., and Nilsson, N. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence Journal*, Volume 2, pp. 189–208, 1971.

[Firby, J., 1987] Firby, J. An Investigation into Reactive Planning in Complex Domains. *Proc. of AAAI-87*, pp. 202–206, Seattle, WA, 1987.

[Georgeff, 1983] Georgeff, M. Strategies in Heuristic Search. *Artificial Intelligence*, Number 20, pp. 393–425, 1983.

[Georgeff and Lansky, 1987] Georgeff, M., and Lansky, A. Reactive Reasoning and Planning. *Proc. of AAAI-87*, pp. 677–682, Seattle, WA, 1987.

[Godefroid and Kabanza, 1991] Godefroid, P., and Kabanza, F. An Efficient Reactive Planner for Synthesizing Reactive Plans. *Proc. of AAAI-91*, Anaheim, CA. pp. 640–645, 1991.

[Ginsberg and Harvey, 1990] Ginsberg, M., and Harvey, W. Iterative Broadening. In *Proc. of AAAI-90*, pp. 216–220, Boston, MA, 1990.

[Hart, *et al.*, 1968] Hart, P., Nilsson, N., and Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.* Vol. 4. pp. 100–107, 1968.

[Kaelbling, 1988] Kaelbling, L. Goals as Parallel Program Specifications. *Proc. of AAAI-88*, pp. 60-65, St. Paul, MN, 1988.

[Korf, 1985] Korf, R. Depth-First Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 27, pp. 97–109, 1985.

[Korf, 1990] Korf, R. Real-Time Heuristic Search. *Artificial Intelligence*, Vol. 42, pp. 189–211, 1990.

[Laird and Rosenbloom, 1990] Laird, J. E., and Rosenbloom, P.S. Integrating Execution, Planning, and Learning in Soar for External Environments. In *Proc. of AAAI-90*, pp. 1022–1029, Boston, MA, 1990.

[Langley, 1992] Langley, P. Systematic and Nonsystematic Search Strategies. *Proc. of First International Conference on Artificial Intelligence Planning Systems*, pp. 145–152, University of Maryland, 1992. J. Hendler, ed. Morgan Kaufman.

[Lyons and Hendriks, 1992] Lyons, D. and Hendriks, A. A Practical Approach to Integrating Reaction and Deliberation. *Proc. of First International Conference on Artificial Intelligence Planning Systems*, pp. 153–162, University of Maryland, 1992. J. Hendler, ed. Morgan Kaufman.

[McDermott, 1992] McDermott, D. Transformational Planning of Reactive Behavior. Yale University, Department of Computer Science, Technical Report YALEU/CSD/RR#941, December, 1992.

[Minton, *et al.*, 1992] Minton, S., Drummond, M., Bresina, J., and Philips, A. Total Order *vs.* Partial Order Planning: Factors Influencing Performance. In *Proc. of the Third International Conference on Principles of Knowledge Representation and Reasoning.*

[Philips and Bresina, 1991] Philips, A., & Bresina, J. NASA TileWorld Manual. NASA Technical Report TR-FIA-91-11, Code FIA, NASA Ames Research Center, Moffett Field, CA, 1992.

[Simmons, 1988] Simmons, R. A Theory of Debugging Plans and Interpretations. *Proc. of AAAI-88*, pp. 94–99, St. Paul, MN, 1988.