Proceedings of the 2000 Winter Simulation Conference
J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

USING SIMULATION AND CRITICAL POINTS TO DEFINE STATES IN CONTINUOUS SEARCH SPACES

Marc S. Atkin
Paul R. Cohen

Experimental Knowledge Systems Laboratory
Department of Computer Science, 140 Governor's Drive
University of Massachusetts / Amherst
Amherst, MA 01003-4610, U.S.A.

ABSTRACT

Many artificial intelligence techniques rely on the notion of a "state" as an abstraction of the actual state of the world, and an "operator" as an abstraction of the actions that take you from one state to the next. Much of the art of problem solving depends on choosing the appropriate set of states and operators. However, in realistic, and therefore dynamic and continuous search spaces, finding the right level of abstraction can be difficult. If too many states are chosen, the search space becomes intractable; if too few are chosen, important interactions between operators might be missed, making the search results meaningless. We present the idea of simulating operators using *critical points* as a way of dynamically defining state boundaries; new states are generated as part of the process of applying operators. Critical point simulation allows the use of standard search and planning techniques in continuous domains, as well as the incorporation of multiple agents, dynamic environments, and non-atomic variable length actions into the search algorithm. We conclude with examples of implemented systems that show how critical points are used in practice.

1 THE PROBLEM: DEFINING STATES IN CONTINUOUS DOMAINS

Many conventional artificial intelligence techniques rely on a clear mapping from the target domain to a state space. Operators transition from one state to the next. Classical search and problem solving algorithms (see (Korf 1988) for a survey), theorem provers, and STRIPS-based planners (Fikes and Nilsson 1971, Fikes and Nilsson 1993) share the assumption that the world can naturally be divided into states, and that what happens as we move from one state to the next is something that can safely be abstracted away.

For simple and discrete problems, defining the state space is indeed often quite straightforward. However, realistic continuous spaces pose a much more difficult problem. In a continuous search space, for example the domain of robot path planning, there is in principle an infinite number of ways to partition the search space into states. The choice of states is intimately linked to the choice of operators, since the operators effect changes in state.

The formulation of the problem space gives rise to an interesting trade-off. If the operators are too primitive, and correspondingly the state space large, the solution to a given problem will involve a deeper search through the space than if the state space were smaller. If the operators become too abstract, however, they start to gloss over all the interactions between operators and the world that made the problem worth solving in the first place.

Consider path planning as an example: Assume a robot on a 2D plane has a `move` operator that will allow it to move a certain distance d. If d is chosen too small, the problem quickly becomes intractable because there are too many possible paths to be considered. If d is too large—larger than some of the obstacles on the map—the robot might jump over an obstacle during the search process, something that it cannot do in the real world. There are better solutions to this particular problem, but that is not the point. The point is that by defining the state space and operator set a priori, one can make the problem unnecessarily hard or overly simplified.

In this paper, we will discuss an approach that avoids this dilemma. The state space is not specified a priori, instead it is generated dynamically as operators are executed. The operators themselves define state boundaries. A simulation model is used to determine the effects of operator application. Operators are no longer atomic in terms of the state space: there can be state boundaries between the beginning and ending of an operator.

2 THE SOLUTION: CRITICAL POINTS

During search, operators move you into different states depending on their outcome. It only makes sense to distinguish state A from state B if the consequences of being in state A differ from those of being in state B. Consequences might be the set of operators that is now applicable, or a difference in an arbitrarily defined reward function. If—as was stated in the previous section—we no longer assume that operators take equally long to execute, or that a state space is defined a priori, then we must be given some sort of additional information about the operator in order to establish what state should be entered upon the operator's completion. This information must be sufficient to allow us to compute *how* (with what outcome) and *when* the operator completes. In effect, we require enough information to be able to *simulate* this operator.

Simple actions, such as moving from point A to point B over unobstructed terrain, have completion times that are easily estimated given the terrain type and the agent's typical movement speed. The type of completion is also easy to predict: without any obstacles, a `move` will always complete successfully. More complex actions make *internal decisions* that influence the action's completion time and type. In the general case, these decisions are conditional on the state of the world *at the time the decision is made*. So we are faced with a problem: to compute an action's outcome (to simulate the action), we need to know what the state of the world every time the action has to make decision.

In order to tackle this problem, we introduce the concept of a *critical point*:

Definition:
A critical point is a time during the execution of an action where a decision might be made, or the time at which it might change its behavior. If this decision can be made at any time during an interval, it is the latest *such time.*

For actions without internal decisions, such as the aforementioned `move` or instantaneous actions such as pushing a button, the only critical point is the completion time. More complicated actions have larger critical point sets.

The attack action depicted in Figure 1 makes a *decision* during its execution: it will abandon the attack if the target is protected by an agent stronger than the attacker. In this example, a white force is attacking a black flag and there is a large black force nearby. The critical point is the time at which the white force is closer to the flag than the black force is now (Step 2). This is the latest point in time at which Black could interfere with the attack action. If Black has started moving to the flag by this time, White will abandon the attack. If Black has remained stationary or

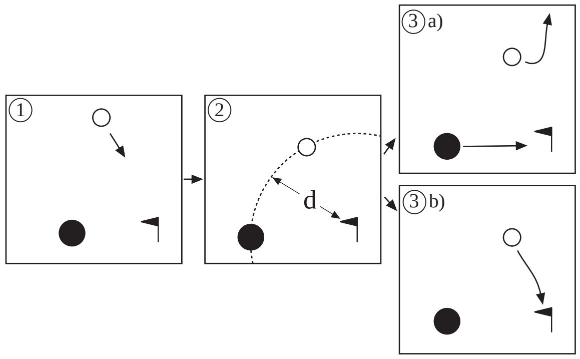

Figure 1: An Example for a Critical Point while Executing an Attack Action

gone somewhere else, the attack will be successful and the flag will be destroyed.

Note that critical points are only bounds, they are not the exact times at which a decision will be made. In the above example, the black force might move to protect its flag right away, in which case White will abandon the attack sooner than the critical time predicted. In this case the real world will differ quantitatively, but not qualitatively, from the outcome we arrived at through critical point simulation. In contrast, if we had simulated without critical points and simply completed White's attack action in the face of an approaching black defender, there would have been a large qualitative difference between the simulated and the real world: The simulation would have generated the outcome that White attacked and was subsequently destroyed by Black, whereas in reality White fled before it came to that. This kind of discrepancy arises exactly in those cases where in simulation we skip over times when important events happen that should have been taken into consideration.

In order to use critical points to simulate an action, every action and plan must have two functions associated with it. The first computes the next critical point for this action. The second, **(advance** *t***)**, takes as an argument a time parameter *t* and changes the world state to reflect the execution of this action *t* time units into the future. These functions are currently written by the designer of the action. Critical point estimations are local to the action; they are based on what this action is likely to do based on the *current state* of the world and predictions that can be made from it. In the case of the attack action shown in Figure 1, the decision about whether or not to abort depends on whether the white force can get closer to the black flag than any black force. A simple approximation of the critical point would be the time it will most likely take, given the terrain, to get as close to the black flag as the closest black force is now. A more accurate approximation would take the current velocities of all black forces into account, since some might be moving towards the flag.

Figure 2 shows the basic algorithm for simulating a group of actions. Simulation time has to be advanced to the minimum of all critical points. To understand why this is necessary, let us consider action T, the action with the minimum critical time t. The decision that T must make at time t depends on the state of the world at that time. The state of the world is affected by all the other actions that are executing. If the world had not been advanced by the minimum of all critical times, T might not make the same decision in simulation as it would have if it had actually executed. The downside of having to take the minimum is that forward simulation will take shorter and shorter jumps as the number of simulated actions increases. This makes sense, though, since the more actions you have, the more possible interactions there might be. (One way to alleviate this problem is to prune the number of actions by eliminating those that will most likely have no effect on the action being evaluated.) The upside is that no action has to concern itself with the critical point computation of any other action.

```
1.   Loop until all actions have completed:
     1.1   Compute the minimum critical point
           $t$ of all actions being simulated.
     1.2   Advance all actions by $t$ time units;
           update world state.
```

Figure 2: The Basic Action Simulation Algorithm using Critical Points.

If a critical point is very hard to compute, it is acceptable for it to be underestimated. This will cause the simulated world state to advance to a time sooner than the actual critical point, and the action will have another chance to estimate it correctly. In the extreme case, an action can report the smallest time increment possible. The evaluation process for this action will degenerate into tick-based simulation.

3 USING CRITICAL POINTS FOR STATE-BASED SEARCH

Critical points were motivated by the need to estimate how and when an action completes, but in effect they are defining a set of interesting states that will occur during the execution of this action. In this capacity, they can be exploited by traditional AI search techniques.

We have been developing a continuous, dynamic, and adversarial domain in which to test our ideas on critical points. This domain is based on the game of “Capture the Flag” (CtF). In CtF there are two teams; each has a number of movable units and flags to protect. Their number and starting locations are randomized. They operate on a map which has different types of terrain. Terrain influences movement speed and forms barriers. A team wins when it captures all its opponent’s flags. A team can also go after its opponent’s units to reduce their strength and effectiveness. This game is deceptively simple. The player must allocate forces for attack and defense, and decide which of the opponent’s units or flags he should go after. The player must react to plans that do not unfold as expected, and possibly retreat or regroup. We model limited visibility and inaccurate sensor data. This leads to additional strategies involving feints, sneak attacks, and ambushes.

In order to demonstrate how critical points might be used in conjunction with search, we created a reduced CtF scenario depicted in Figure 3. There are two units on either team. Black’s goal is to defend its three flags, White’s is to destroy them. In order to keep things simple, Black behaves reactively in this scenario, meaning that White need not consider alternative actions for Black. The purpose of the search algorithm is to generate a schedule of actions for White that will destroy the flags in the shortest possible time.

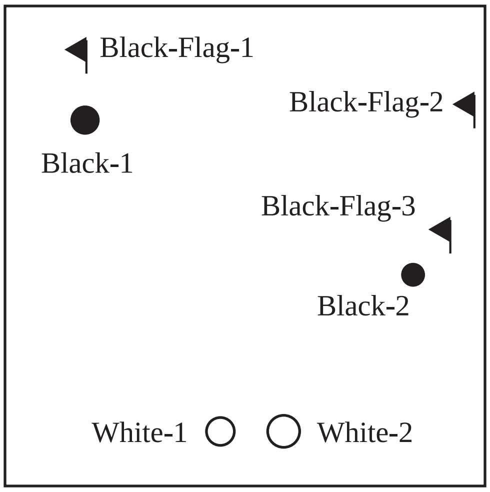

Figure 3: The Initial Configuration of the Critical Point Search Scenario.

White’s operator set is basic: It can only attack enemy flags. Since there are three possible targets for White and two White blobs, this results in 3^2 possible action combinations for White—a branching factor of 9. The attack action has two critical points: The first is its estimated completion time, and second one the time at which it will be closer to the target than any enemy unit is now. The attack action aborts if its target is being protected by an enemy that it cannot defeat.

Figure 4 outlines the basic algorithm used. For sake of simplicity, we use depth-first search, but in principle any state-based search algorithm could be used. The function **cp-search** has two arguments: the first is *world-state*, the state of the world at the time the **cp-search** is called; the second is *schedule*, which contains the best sequence of actions for every agent in the world-state starting at the current time until the game ends. The function **cp-search()** evaluates all combinations of actions that are applicable in

```
Define function cp-search(world state, schedule):
  1.   Let $A$ be the set of all possible action combinations that can be executed in this world state. Loop over all $a \in A$:
       1.1.  In simulation, loop until game end conditions are met or some agent's action has completed:
             1.1.1  Compute the minimum critical time $t$ of all actions being simulated.
             1.1.2  Advance all actions by $t$ time units; generate a new-world-state.
       1.2.  If game is over, evaluate new-world-state; return this value as the score for $a$.
       1.3.  If an action has completed, recursively call cp-search(new-world-state, schedule); return the score of the returned schedule as the score for $a$.
  2.   Add the action combination with highest score to the schedule.
  3.   Return the schedule.

Main Body:
  1.   Call cp-search(initial-world-state, nil)
  2.   Execute the actions in the returned schedule sequentially for every agent.
```

Figure 4: A Search Algorithm Based on Critical Points

the current world state and adds the best one to the schedule. At the very heart of **cp-search()** is the familiar loop which advances the world ahead to the next critical time. The world state is advanced until an action completes (or the game ends), at which point **cp-search()** is recursively called on the updated world state. Time is modeled explicitly—time always advances to the next critical point. The search branches when more than one operator (action) can be executed.

In effect, the world state branches every time an agent completes (or aborts) its action. State boundaries are being created dynamically depending on the execution of the actions being simulated. When an agent goes idle, every new possible action is considered for it. Figure 5 illustrates this process for a hypothetical tree with branching factor 2: At time 0, two actions $a1$ and $a2$ are being simulated. They have the corresponding completion times $c1$ and $c2$ (when may themselves be the result of several critical point jumps in the inner-most simulation loop). When $a1$ completes, two actions, $a3$ and $a4$ are considered to replace it. Action $a2$ is still present in both search paths, and when it completes, it too is replaced by one of two possible alternative actions. Since both branches of the initial tree are now simulating distinct sets of actions, their completion times no longer line up.

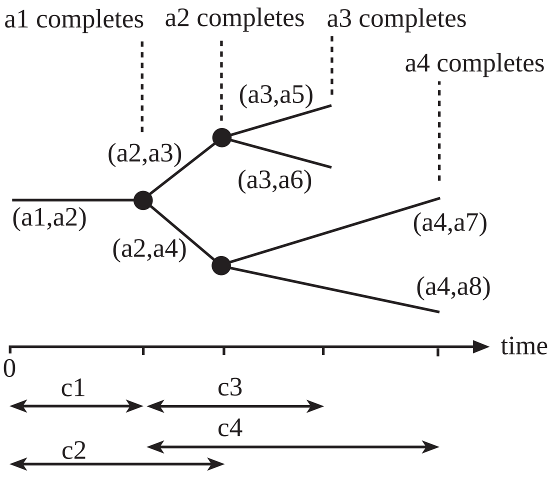

Figure 5: An Illustration of How Search Integrates with Action Simulation

4 COMPARISON BETWEEN CLASSICAL AND CRITICAL POINT SEARCHES

It is instructive to investigate how a classical depth-first search compares to the critical point search in the CtF scenario. In practice, this is not easy to do, since a standard state-based search has trouble dealing with concurrent actions that take varying amounts of time to execute. The next best thing, however, is to compare a standard critical point search with a search involving only the minimal number of critical points. For ease of reference, call the standard search `s-cps` and the reduced version `r-cps`. `r-cps` uses only the critical points that estimate an action's completion time, `s-cps` uses all critical points. In our simple scenario, this means that in addition to completion critical points, the critical point that predicts aborting the attack action is used.

Figures 6 and 7 show the best schedule found by the two versions of the algorithm, respectively. Since Black-Flag-1 is being defended by a unit with greater strength (indicated by the larger size) than White-1, White-1 cannot successfully take this flag. `r-cps` does not take this into account during the search process and consequently generates a schedule which is not executable (during execution, White-1 is defeated and Black-Flag-1 is not destroyed). `s-cps` on the other hand, generates a schedule which takes slightly longer, but ensures that White can win the engagements it gets involved in.

Table 1 summarizes how each algorithm performed on this trial run. "Critical points considered" is the total number of jumps that occurred within the inner-most simulation loop (bullet 1.2 in Figure 4). Understandably, `s-cps` considers a larger number.

`r-cps` is not taking an important piece of information into account, namely that the attack action may abort. The only way to address this problem in traditional search is

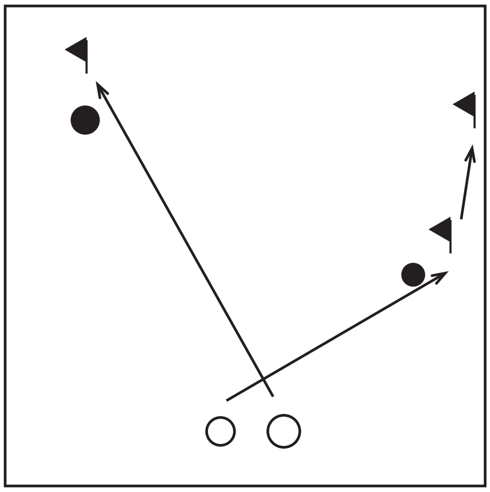

Figure 6: The Best Schedule Found in `s-cps`, using the Additional Critical Point for Aborting the Attack Action

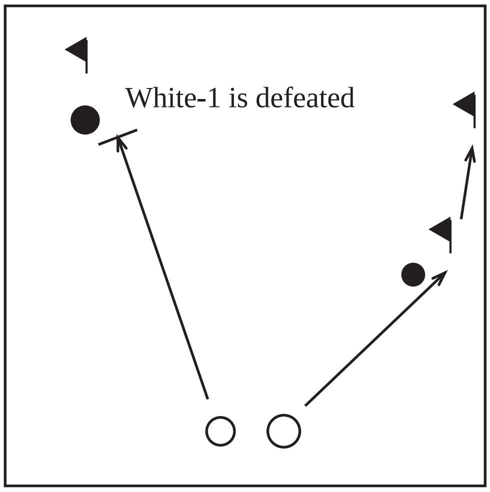

Figure 7: The Best Schedule Found in `r-cps`, using Only Critical Points for Completion

to redefine the operator set, effectively splitting `attack` into two operators `move-past-defense-line` and `attack-with-greater-numbers`. Increasing the operator set will greatly increase the number of nodes that have to be searched, something that *does not* happen when a critical point is added to an action. The number of critical points influences the efficiency of the simulation loop, not the branching factor.

Table 1: `s-cps` vs. `r-cps` in the CtF Scenario

search method	`s-cps`	`r-cps`
critical points considered	111	88
est. completion time	110	104
actual completion time	120	–

5 USING CRITICAL POINTS FOR PLAN EVALUATION

Critical points have a wider applicability than just search. The full version of CtF has many agents and flags on each side; any generative planning solution would have to face an enormous branching factor since many possible action combinations can be executed at any given time. To cope with this problem, we rely on a partial hierarchical planner (Georgeff and Lansky 1986), which retrieves plans from a set of pre-compiled skeletal solutions, and uses heuristics to allocate resources in a reasonable way (for example, an `attack` plan will never attack a target with a smaller force than the force defending it).

When several plans apply, partial hierarchical planners typically select one according to heuristic criteria. Military planners will actually play out a plan and determine how the opponent might react to it. A wargame is a qualitative simulation. The CtF planner does the same: it simulates potential plans at some abstract level, then applies a static evaluation function to select the best plan. The static evaluation function incorporates such factors as relative strength and number of captured and threatened flags of both teams, to describe how desirable this future world state is.

Simulation is a costly operation, and in order to do it efficiently, CtF must be able to jump ahead to times when interesting events take place in the world. Again we face the problem of having to impose "states" on a continuous domain. Critical points are essential for plan evaluation in the CtF planner, since they are used to guide forward simulation. The basic idea behind forward simulation is that instead of advancing the world tick by tick, which is time-consuming, we jump right to the next critical point. Forward simulation proceeds as outlined in Figure 8.

This application of critical points is different from the one in the previous sections in that no search need be conducted. One plan is given, and the goal is to determine what the world state would look like if this plan were to execute. What makes this interesting is that CtF is an

```
1.  Add the plan $P$ to be evaluated to all the
    actions currently ongoing in the simulator.
2.  In simulation, loop either until a fixed time
    in the future or until too many errors have
    accumulated in the simulation:
    2.1  Compute the minimum critical time $t$
         of all actions being simulated.
    2.2  Advance all actions by $t$ time units;
         update world state.
3.  Evaluate the resulting world state; return this
    value as the score for the plan $P$.
```

Figure 8: The Plan Evaluation Algorithm.

adversarial domain. In lieu of a detailed opponent model, we simply assume the opponent would do what we would do in his situation. During forward simulation, the action list also contains opponent actions. When CtF starts plan evaluation, it simply puts the top-level goal **win-the-game** for the opponent into the action list. The opponent action's critical times are computed just like ours, and they are advanced in the same way. Whereas our side evaluates all plans and chooses the best one, the opponent chooses the worst one (for us). This is a form a minimax search, with the two sides executing their plans in parallel.

This brings up another point: in our previous example we only had to simulate fairly simple actions like `move` and `attack`, but now we have to generate critical points for complex plans such as `win-the-game`. As actions get more complex, their critical point computations become more complex as well. This problem is mitigated, however, by the fact that actions and plans are organized hierarchically in CtF. Just as actions lower in hierarchy can be used as building blocks to achieve some goal of a higher level action, so can critical points of more complex actions base their computations on simpler ones. Consider as an example the action `follow`, which repeatedly schedules `move` to chase a moving target. Let us assume `follow` periodically checks the position of the target and redirects the currently executing `move` if need be. `follow` will also schedule a new `move` if the the current one aborts for some reason. `follow`'s critical points are simply the union of `move`'s critical points and the target check period. These are the times at which `follow` has to make a decision about changing course.

6 DISCUSSION AND RELATED WORK

To the best of our knowledge, the idea of using critical points to make any continuous search space suitable for classical AI methods has not been put forth in this general form before. That is not to say that intellectual precedents don't exist, however. Others have used simulation to evaluate (Lee and Fishwick 1994) or test plans (Beetz and McDermott 1994, Hammond 1990, Lesh, Martin, and Allen 1998). Not all planning approaches represent state in the same way, and there is indeed an entire subfield of planning that seeks to reason about continuously changing processes (e.g. (Dean and Wellman 1991, Penberthy and Weld 1994)).

Critical points are well known in Qualitative Physics (Weld and deKleer 1989, Forbus 1984). Roboticists, in particular those dealing with motion planning (Canny 1988, Latombe 1991), have long had to face the problem of continuous search spaces. Many approaches for quantizing these search spaces exist, here we will only touch on the most common: Cell decomposition methods overlay the continuous space with a finite number of often regularly shaped cells. Conventional search algorithms are used to plan a path from a cell to any other. Skeletonization methods, for example those used to generate Voronoi diagrams, collapse the infinite number of possible points in the traversable space to a roadmap that defines safe paths between obstacles. The roadmap is a graph, and graph search methods can be used for path planning. Note that while all these approaches are general, they impose an a priori state decomposition on the search space, unlike critical points, which generate state boundaries based on the action set. If one were to do path planning with critical points, the `move` *action* would report, given the size of the agent and the direction it was going, where the next decision point would have to be.

One of the biggest open issues is *how*, given an action, one should go about defining its critical points. We have stated that this process involves estimating, through experience or insight, at which times a decision has to be made within the action. Critical points are the times at which an action might take a different course depending on the state of the environment. We do want to emphasize that this problem is indeed easier than partitioning the complete search space into states. When estimating critical points, one can look at one action in isolation, to partition the state space one has to decide for *all* possible actions what characteristics of the space are relevant.

Finding critical points can be compared to the problem of learning planning operators. They are one more thing that has to be specified in addition to pre- and postconditions when designing operators. It might even be feasible to *learn* critical points. Recent work in nonlinear dynamics (Rosenstein and Cohen 1998) has shown how it is possible to cluster conceptually related actions based time series of associated sensor readings. We have the hypothesis that in such clusters, critical points are the points at which groups of time series diverge.

While critical points can be used to generate dynamic state boundaries, they do not by themselves reduce the potentially high branching factor of searches in continuous domains. The reason we used a partial hierarchical planner in the Capture the Flag domain, and not a search algorithm, is precisely due to the high branching factor.

ACKNOWLEDGMENTS

This research is supported by DARPA/USAF under contract numbers N66001-96-C-8504, F30602-97-1-0289, and F30602-95-1-0021. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Defense Advanced Research Projects Agency/Air Force Materiel Command or the U.S. Government.

REFERENCES

Beetz, M. and D. McDermott 1994. Improving robot plans during their execution. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 7–12.

Canny, J. F. 1988. *The complexity of robot motion planning*. Cambridge, Massachusetts: MIT Press.

Dean, T. and M. Wellman 1991. *Planning and control*. Morgan Kaufmann.

Fikes, R. E. and N. J. Nilsson 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2*(2): 189–208.

Fikes, R. E. and N. J. Nilsson 1993. STRIPS, a retrospective. *Artificial Intelligence 59*(1-2): 227–232.

Forbus, K. D. 1984. Qualitative process theory. *Artificial Intelligence 24*: 85–168.

Georgeff, M. P. and A. L. Lansky 1986. Procedural knowledge. *Proceedings of the IEEE Special Issue on Knowledge Representation 74*(10): 1383–1398.

Hammond, K. J. 1990. Explaining and repairing plans that fail. *Artificial Intelligence Journal 45*: 173–228.

Korf, R. E. 1988. Optimal path finding algorithms. In L. N. Kanal and V. Kumar (Eds.), *Search in Artificial Intelligence*, Chapter 7, 223–267. Springer Verlag.

Latombe, J.-C. 1991. *Robot motion planning*. Dordrecht, The Netherlands: Kluwer.

Lee, J. and P. A. Fishwick 1994. Simulation-based planning for computer generated forces. *Simulation 63*(5): 299–315.

Lesh, N., N. Martin, and J. Allen 1998. Improving big plans. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 860–867. AAAI Press.

Penberthy, J. and D. S. Weld 1994. Temporal planning with continuous change. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1010–1015. Menlo Park, CA: AAAI/MIT Press.

Rosenstein, M. and P. R. Cohen 1998. Concepts from time series. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 739–745. AAAI Press.

Weld, D. and J. deKleer 1989. *Readings in qualitative reasoning about physical systems*. Los Altos, CA: Morgan Kaufmann Publishers.

AUTHOR BIOGRAPHIES

MARC S. ATKIN is a graduate student at the Experimental Knowledge Systems Laboratory at the University of Massachusetts. He did his undergraduate work at Karlsruhe University in Germany, and received his Masters in Computer Science from the University of Massachusetts. He is now pursuing a Ph.D. in artificial intelligence at the same institution. His research has focussed on the design of intelligent embedded agents. Currently, his interests include domain-general planning and techniques for efficient control of agents in real-time, continuous, and adversarial domains. His email and web addresses are `<atkin@cs.umass.edu>` and `<eksl.cs.umass.edu/˜atkin>`.

PAUL R. COHEN received his Ph.D. from Stanford University in 1983. He is a Professor of Computer Science at the University of Massachusetts, and Director of the Experimental Knowledge Systems Laboratory. He edited the “Handbook of Artificial Intelligence,” Volumes III and IV with Edward Feigenbaum and Avron Barr. Cohen was elected in 1993 as a Fellow of the American Association for Artificial Intelligence, and served as Councillor of that organization, 1991–1994. His research concerns the design principles for intelligent agents and the acquisition of conceptual structures. His email and web addresses are `<cohen@cs.umass.edu>` and `<eksl.cs.umass.edu/˜cohen>`