Searching Continuous State Spaces Effectively Using Critical Points

Keywords: search or optimization, multi-agent simulation

Tracking Number: A396

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. If the state boundaries are not at the right place, the results of the search will not be meaningful in the real world.

We present the idea of *critical points* as way of dynamically defining state boundaries; new states are generated as part of the process of applying operators. In addition to allowing the use of standard search and planning techniques in continuous domains, critical points enable 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.

The Problem: Defining States in Continuous Domains

Many conventional artificial intelligence techniques rely on clearly defined state spaces and operators to 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 & Nilsson 1971; 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 affect changes in state. When choosing an operator set, one is also implicitly stating that these are the actions that should be considered atomic.

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 too easy.

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. Operators are no longer atomic in terms of the state space: there can be state boundaries between the beginning and ending of an operator. Whether an operator takes a long time or a short time to complete or whether it influences a large or small spatial area need not be a concern in choosing the operator set.

The Solution: Critical Points

Our premise is that the actions an agent can take should define state boundaries. In order to be able to do this, knowing *when* and *how* an action will complete is key. It must be possible to *simulate* the most likely outcome of any action. For an action of any complexity, this requirement can be very difficult to satisfy. The classical formulation of search problems does not concern itself with the duration of an operator, only that the operator might cause the state to change. In our approach, states are not pre-defined, and will depend on what happens in the world while the operator executes. So while classical search can safely ignore the internal structure of an operator, we cannot.

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 simulate an 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.

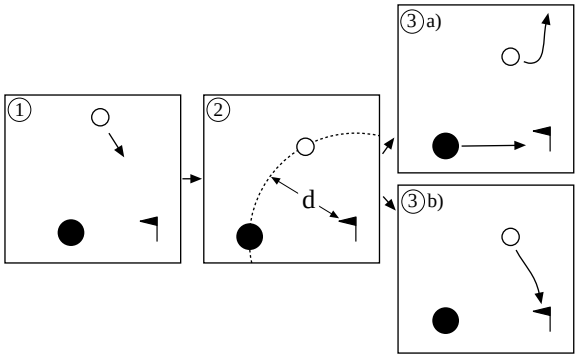

Figure 1: An example for a critical point while executing an attack action. The critical point is reached in step 2, where the attacking force must decide whether to continue or abandon the attack.

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. 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 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. This is not a large qualitative difference to the scenario that was simulated. If we had simulated without critical points, and simply completed White's action, there *would* have been a large qualitative error: The flag would have either marked as destroyed (which wouldn't have happened if Black moved in), or White would have been destroyed by the protecting black force (which would never have happened since White would have fled before it came to that).

1. Loop until all actions have completed:
   1.1 Compute the minimum critical time 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.

In order to use critical times to simulate an action, every action and plan must have two functions associated with it. The first computes the next critical time for this action. The second, **(advance world-state t)**, takes as an argument a time parameter t and will change the world state to reflect the execution of this action t time units into the future. Currently, we have no automated way of generating these functions, so they are written by the designer of the action. 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.

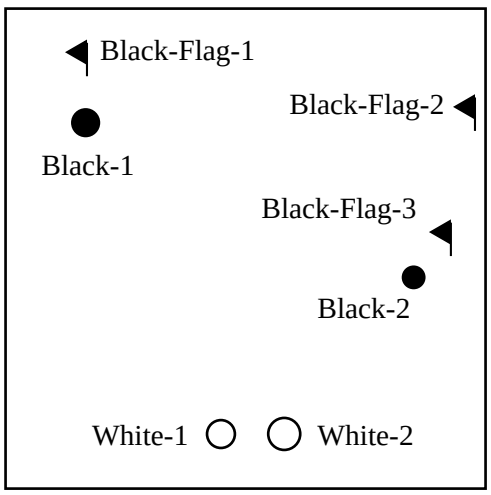

Figure 3: The initial configuration of the critical point search scenario.

Another upside is that if a critical time 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 time, at which point 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.

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 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 not unfolding as expected, and possibly retreat or regroup. There are many tactics, from attacking all-out to trying to sneak by the opponent's line of defense. In our current implementation, both players have a global view of the game; when we add limited visibility many more strategies, such as ambushes or traps, will emerge.

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: The critical point search algorithm.

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.

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 the current world state and adds the best one to action schedule. At the

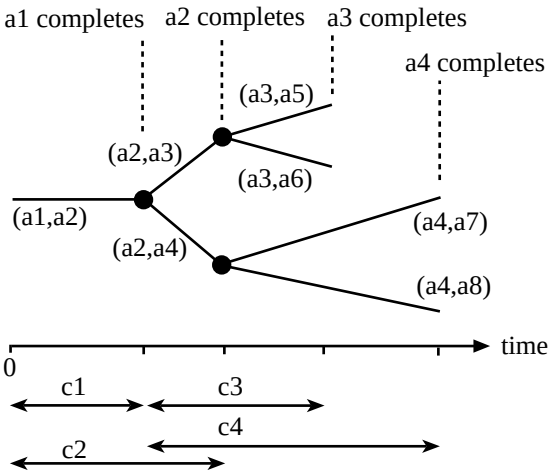

Figure 5: An illustration of how search integrates with action simulation. At time 0, two actions $a1$ and $a2$ are being simulated. They have the corresponding completion times $c1$ and $c2$ (when may themselves be the result of several critical point jumps in the inner-most simulation loop). When $a1$ 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.

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.

In effect, the world state is split 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.

search method	S-CPS	R-CPS
nodes expanded	56	50
critical points considered	115	96
est. completion time	97	96
actual completion time	122	–

Table 1: S-CPS vs. R-CPS in the CTF scenario.

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 of varying lengths. 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 the completion critical points, the critical point that predicts aborting the attack action is used.

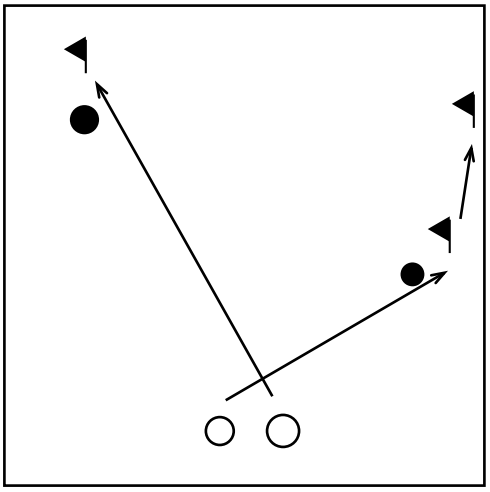

Figure 6: The best schedule found in S-CPS, using the additional critical point for aborting the attack action.

Figures 7 and 6 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. 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. "Nodes expanded" refers to the total number of actions that were considered during the search process; "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. Note however that the number of nodes and thus the number of states visited does not increase much in S-CPS. The explanation for this is that the number of states that has to be looked at depends primarily on the number of actions that have to be executed before a final state is reached, and not on the number of critical points. The number of critical points influences the efficiency of the

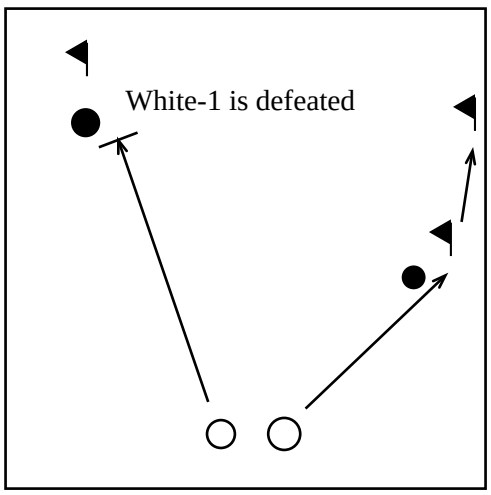

Figure 7: The best schedule found in R-CPS, using only critical points for completion. This schedule does solve the problem because during actual execution, White-1 is defeated and Black-Flag-1 is not destroyed.

simulation loop.

Overall, the results are not surprising, since R-CPS is not really suited to solving this problem; the search process is not taking an important piece of information into account, namely that the attack action may abort. However, the only way to address this problem in traditional search is 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.

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 & 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.

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.

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 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.

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. Critical points are well known in Qualitative Physics (Weld & 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 traversible 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 & 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.

For many domains, using critical points to generate states is probably overkill. Every action must have a way to estimate the next critical time as well as a model of how it effects the world. However, critical points are ideally suited for domains that have a high level of interaction between operators, where multiple agents running actions of varying length must be considered, or where the world itself changes dynamically. It is our view that it precisely the interesting domains that have these characteristics.

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

Canny, J. F. 1988. *The Complexity of Robot Motion Planning*. Cambridge, Massachusetts: MIT Press.

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

Fikes, R. E., and Nilsson, N. J. 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 Lansky, A. L. 1986. Procedural knowledge. *IEEE Special Issue on Knowledge Representation* 74(10):1383–1398.

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

Latombe, J.-C. 1991. *Robot Motion Planning*. Dordrecht, The Netherlands: Kluwer.

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

Weld, D., and deKleer, J. 1989. *Readings in Qualitative Reasoning about Physical Systems.* Los Altos, CA: Morgan Kaufmann Publishers.