Search Algorithms in the Situation Calculus

Fangzhen Lin

Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
Email: flin@cs.ust.hk

Abstract. In this paper we consider axiomatizing AI search algorithms in the situation calculus. Proper axiomatization of such algorithms is important for understanding control knowledge which is by definition search algorithm dependent: a control knowledge can be effective with respect to one search algorithm but ineffective with respect to another. The key idea here is to view search algorithms as strict linear orders on sets of situations in the situation calculus. There are several potential advantages of viewing search algorithms this way. One is that according to it, implementing a search strategy amounts to computing the "next" relation of the corresponding linear order. We found this perspective particularly helpful in understanding various memory-bounded implementations of breadth-first and best-first search as it separates the definition of a search strategy from its implementations. A more important advantage is that according to it, a particularly simple account of search pruning is to consider it as restriction of the original search algorithms, viewed as an ordering relation, to a smaller set of situations.

1 Dedication

This is a work on using the situation calculus to axiomatize control knowledge in problem solving. I can trace the basic idea to a time when I was working with Ray at the University of Toronto, and I have no doubt that it had been influenced by Ray's work and ideas. On a more technical note, this paper would not be possible without Ray's pioneering work on foundational axioms of the situation calculus.

2 Introduction

Meta-level knowledge is knowledge about knowledge. Control knowledge is the kind of meta-level knowledge that tells a problem solver how best to use domain knowledge to solve a problem. An example of control knowledge in planning in the blocks world is:

> If a block needs to be on the table in the goal state, and it is already on the table, then don't move it.

This advice indirectly tells the problem solver what to do by eliminating those actions that would change the location of a block that is already on the table.

Although the importance of control knowledge in problem solving has long been recognized (e.g. [3,4,1,2]), there has not been much work on the subject. In particular, there has not been a widely accepted formalism for representing and reasoning about such knowledge.

In the past few years, we have been advocating using the situation calculus as such a formalism: we have applied it to axiomatizing the cut operator in logic programming [9], subgoal ordering in planning [8], and temporally extended goals [10]

In this paper we consider axiomatizing AI search algorithms in the situation calculus. Proper axiomatization of such algorithms is important for understanding control knowledge which is by definition search algorithm dependent: it can be effective with respect to one search algorithm but ineffective with respect to another. For instance, the cut operator in Prolog is used to express control knowledge about when not to backtrack, thus is useful only for backtrack oriented search algorithms such as depth-first search, and useless for others such as the breadth-first search.

Our key idea is to view a search algorithm as a strict linear order on situations in the situation calculus: a situation s is ordered after another one s' if according to the search strategy s has to be examined first for s' to be examined. To illustrate, we consider depth-first, breadth-first, and best-first search strategies. As it turns out, the strict linear orders for these algorithms can be axiomatized succinctly.

We see several advantages of viewing search algorithms this way. One is that according to it, implementing a search strategy amounts to computing the "next" relation of the corresponding linear order. This perspective is particularly helpful for understanding memory-bounded implementations of breadth-first and best-first search. For instance, we show that iterative deepening is not really a new search strategy but an implementation of breadth-first search that trades time for space. A more important advantage is that according to it, a particularly simple account of search pruning is to consider it as the restriction of the original search algorithm, viewed as an ordering relation, to a smaller set of situations.

In the following, we first review the definition of search problems and the situation calculus, and then show how a search problem can be formulated as an action theory in the situation calculus.

3 Search Problems and Algorithms

A search problem consists of a set of operators, an initial state, a goal, a cost function, and for informed (heuristic) search problem, a state evaluation function.

A typical search problem is the 8-puzzle where we have a board with 3×3 cells that can be either blank or one of the numbers between 1 and 8:

- Operators: moving the blank up, down, left, and right.
- The initial state: any given configuration of the board.
- Goal test: check to see if the current configuration of the board is the same as that in the goal state in Figure 1.
- Cost function: number of actions in a solution, i.e. each action costs one unit.

1	6	3
5		7
4	2	8

An initial state

1	2	3
8		4
7	6	5

The goal state

Fig. 1. The 8-puzzle

Given a search problem, search algorithms start at the initial state, and keep applying operators until a solution is found. Examples of search problems and search algorithms can be found in all introductory AI books (e.g. [13,5,17]). In the following we shall consider depth-first, breadth-first, and best-first search algorithms.

4 The Situation Calculus

The situation calculus (McCarthy and Hayes [12]) is a formalism for representing and reasoning about actions in dynamic domains. It is a many-sorted predicate calculus with some reserved predicate and function symbols. For example, in the blocks world, to say that block A is initially clear, we write:

$$H(clear(A), S_0),$$

where H is a reserved binary predicate and stands for "holds", and S_0 is a reserved constant symbol denoting the initial situation. As an another example, to say that after the action $stack(x, y)$ is performed, the proposition $on(x, y)$ will be true, we write:[1]

$$Poss(stack(x, y), s) \supset H(on(x, y), do(stack(x, y), s)),$$

[1] In this paper, free variables in a displayed formula are assumed to be universally quantified.

where the reserved function $do(a, s)$ denotes the resulting situation of doing the action a in the situation s, and $Poss(a, s)$ is the precondition for a to be executable in s. This is an example of how the effects of an action can be represented in the situation calculus.

More precisely, we use a many-sorted second-order language with equality for the situation calculus. We assume the following sorts: *situation* for situations, *action* for actions, *fluent* for propositional fluents such as *clear* whose truth values depend on situations, and *object* for everything else. As mentioned above, we assume that S_0 is a reserved constant denoting the initial situation, H a reserved predicate for expressing properties about fluents in a situation, *do* a reserved binary function denoting the result of performing an action, and *Poss* a reserved binary predicate for expressing action preconditions.

We assume that the space of situations is an infinite tree whose root is S_0 and whose nodes are situations obtained from S_0 by applying a sequence of actions. More precisely, we assume the following foundational axioms:

$$S_0 \neq do(a, s), \tag{1}$$

$$do(a_1, s_1) = do(a_2, s_2) \supset (a_1 = a_2 \wedge s_1 = s_2), \tag{2}$$

$$(\forall P)[P(S_0) \wedge (\forall a, s)(P(s) \supset P(do(a, s))) \supset (\forall s)P(s)], \tag{3}$$

The first two axioms are unique names assumptions for situations. The third axiom is second order induction which says that every situation has to be obtained from the initial one by repeatedly applying the function *do*. In other words, it says that a situation is either the initial situation S_0 or the result of performing a sequence of actions in the initial situation. These axioms imply that situations in the situation calculus are isomorphic to finite sequences of actions:[2] if s is a situation, then there is a unique finite sequence L of actions such that $s = do(L, S_0)$. This isomorphism is what makes the situation calculus a good formalism for representing control information because instead of talking about sequences of actions, which is often what control knowledge is about, we can then talk about situations which are objects in the situation calculus.

Frequently, instead of all situations, we only want to consider those that can be reached from the initial situation by a sequence of *executable* actions. To define these situations, we introduce a strict partial order $<$ into our

[2] Given a sequence of actions $[a_1, ..., a_n]$, we use $do([a_1, ..., a_n], s)$ to denote the resulting situation of performing the sequence of actions in s. Inductively, $do([], s) = s$ and $do([a|L], s) = do(L, do(a, s))$.

language[3], and define it inductively as follows:

$$\neg s < S_0, \tag{4}$$

$$s < do(a, s') \equiv (Poss(a, s') \wedge s \leq s'), \tag{5}$$

where $s \leq s'$ is a shorthand for $s < s' \vee s = s'$.

Using this partial order, we can then define:

$$\mathsf{Exec}(s) \equiv S_0 \leq s.$$

It can be shown that $\mathsf{Exec}(s)$ iff there is a finite sequence of actions $[a_1, ..., a_n]$ such that $s = do([a_1, ..., a_n], S_0)$ and for each $1 \leq i \leq n$, a_i is executable: $Poss(a_i, do([a_1, ..., a_{i-1}], S_0))$. Thus the set $\{s|\mathsf{Exec}(s)\}$ is the set of situations reachable from the initial one by a sequence of executable actions. One can think of the set as the space of all situations that a robot can possibly end up in beginning in S_0. If, say, action A is not executable in S_0, then the situation $do(A, S_0)$, although exists as a logical object, is not realizable. In the following, we call a situation s such that $\mathsf{Exec}(s)$ an *executable situation*.

In the following, we shall denote by Σ the set of the above axioms (1) - (5). To make the problem non-trivial, we shall assume that there is at least one executable action: $(\exists a)Poss(a, S_0)$.

5 Search Problems in the Situation Calculus

Given a search problem, we shall represent its initial state by the initial situation S_0, and each of its operators by a unique action. Under this representation, information about the initial state of a search problem will be represented by a set of situation calculus formulas about S_0, and information about its operators will be axiomatized by a situation calculus action theory.

To express goals in search problems, we introduce *fluent formulas* which are situation calculus formulas with situation terms suppressed. Formally, we have:

1. Any formula that does not mention a situation term is a fluent formula.
2. If F is a n-ary fluent, and $t_1, ..., t_n$ are object terms, then $F(t_1, ..., t_n)$ is a fluent formula.
3. If ϕ and φ are fluent formulas, x an object variable, and p a fluent variable, then $\neg\varphi$, $\varphi \vee \phi$, $(\forall x).\varphi$ and $(\forall p).\phi$ are fluent formulas. (Other connectives such as $\wedge$, $\supset$, $\equiv$, and $\exists$ can be defined as usual.)

[3] The relation $<$ was first defined in the the situation calculus by Reiter [14]. Our presentation here follows (Lin and Reiter [11]) where some of its properties such as transitivity are shown. For a thorough discussion of foundational axioms of the situation calculus, see (Reiter [15]).

For example, $A1 \neq A2$, $on(block1, block2)$, $(\forall p)(p \supset p)$ are fluent formulas, where $A1$ and A are action constants and on a binary fluent.

Given a fluent formula φ and a situation term S, we define $H(\varphi, S)$ to be a shorthand for a situation calculus formula according to the following rules:

$$
\begin{aligned}
H(\varphi, S) &\stackrel{def}{=} \varphi, \text{ provided } \varphi \text{ does not mention} \\
&\qquad \text{any situation terms} \\
H(\neg\varphi, S) &\stackrel{def}{=} \neg H(\varphi, S), \\
H(\varphi \vee \phi, S) &\stackrel{def}{=} H(\varphi, S) \vee H(\phi, S), \\
H((\forall x)\varphi, S) &\stackrel{def}{=} (\forall x) H(\varphi, S), \\
H((\forall p)\varphi, S) &\stackrel{def}{=} (\forall p) H(\varphi, S).
\end{aligned}
$$

For example,

$$
\begin{aligned}
&H(a \neq b, s) \stackrel{def}{=} a \neq b, \\
&H(on(a, b) \vee clear(b), s) \stackrel{def}{=} H(on(a, b), s) \vee H(clear(b), s).
\end{aligned}
$$

Now given a search problem, suppose that T is a situation calculus theory that captures its initial state and operators, and that the goal of the problem corresponds to fluent formula G, then testing if the goal is achieved by a sequence of actions becomes logical entailment checking: a finite sequence L of actions is a solution iff $T \cup \Sigma \models H(G, do([a_1, ..., a_n], S_0))$.

To illustrate this conceptualization of search problems in the situation calculus, consider again the 8-puzzle. We have the following fluent:

- $at(x, i, j)$ - number x is in cell $i \times j$. We denote blank by number 0. So $at(0, 1, 2)$ means that blank is in row 1 and column 2.

and the following action:

- $move(i_1, j_1, x, i_2, j_2)$ - swap 0 and x, provided that 0 is in (i_1, j_1), x is in (i_2, j_2), and that they are next to each other.

The following precondition axiom ensures that at most four instantiations of the above action can be performed in any situation:

$$
\begin{aligned}
&Poss(move(i_1, j_1, x, i_2, j_2), s) \equiv x \neq 0 \wedge H(at(0, i_1, j_1), s) \wedge H(at(x, i_2, j_2), s) \wedge \\
&\quad [i_1 = i_2 \wedge (j_1 = j_2 + 1 \vee j_2 = j_1 + 1)] \vee [j_1 = j_2 \wedge (i_1 = i_2 + 1 \vee i_2 = i_1 + 1)],
\end{aligned}
$$

where we assume that i's and j's are numbers, and the operation "+" has its usual meaning.

The effects of this action can be axiomatized by the following successor state axiom which says that the only action that has effect on at is $move$, and $move(i_1, j_1, x, i_2, j_2)$ swaps 0 and x and leaves the positions of all other numbers unchanged:

$$\begin{aligned}
&Poss(a, s) \supset H(at(x, i, j), do(a, s)) \equiv \\
&\qquad x = 0 \wedge (\exists i', j', y) a = move(i', j', y, i, j) \vee \\
&\qquad (\exists i', j', y) a = move(i, j, x, i', j') \vee \\
&\qquad H(at(x, i, j), s) \wedge \neg(\exists i_1, i_2, j_1, j_2) a = move(i_1, j_1, x, i_2, j_2)
\end{aligned}$$

Now let

- T be the set of above action precondition and successor state axioms;
- D_{S_0} the following set of axioms that capture the initial state in Figure 1:

$$\begin{aligned}
&\{H(at(1,1,1), S_0), H(at(6,1,2), S_0), H(at(3,1,3), S_0), H(at(5,2,1), S_0), \\
&H(at(0,2,2), S_0), H(at(7,2,3), S_0), H(at(4,3,1), S_0), H(at(2,3,2), S_0), \\
&H(at(8,3,3), S_0)\},
\end{aligned}$$

- D_{una} the set of following unique names axioms about actions:

$$\begin{aligned}
&move(i_1, j_1, x, i_2, j_2) = move(i'_1, j'_1, x', i'_2, j'_2) \supset \\
&\qquad x = x' \wedge i_1 = i'_1 \wedge j_1 = j'_1 \wedge i_2 = i'_2 \wedge j_2 = j'_2,
\end{aligned}$$

- G the following fluent formula that captures the goal state in Figure 1:

$$\begin{aligned}
&at(1,1,1) \wedge at(2,1,2) \wedge at(3,1,3) \wedge at(8,2,1) \wedge at(0,2,2) \wedge \\
&\qquad at(4,2,3) \wedge at(7,3,1) \wedge at(6,3,2) \wedge at(5,3,3).
\end{aligned}$$

then testing if a sequence of actions L is a solution becomes checking the validity of the following entailment:

$$T \cup \Sigma \cup D_{S_0} \cup D_{una} \models_{nat} \mathsf{Exec}(do(L, S_0)) \wedge H(G, do(L, S_0)),$$

where $\models_{nat}$ stands for logical entailment with the arguments of at and $move$ given standard number interpretation. This entailment is an instance of temporal projection, and can be checked efficiently if the initial state is complete.

6 Search Algorithms in the Situation Calculus

Under this formulation of search problems in the situation calculus, the basic idea of this paper is to view a search algorithm as a strict linear order on a set of executable situations, and search pruning as a restriction on the domain of the linear order.

Definition 1. A binary relation R is a strict linear order if it is

1. irreflexive: $\neg R(x, x)$;

2. transitive: $R(x,y) \wedge R(y,z) \supset R(x,z)$; and
3. connected: $x = y \vee R(x,y) \vee R(y,x)$,

where x, y, z range over the domain of R, which is defined to be the following set:

$$D_R = \{x \mid (\exists y)(R(x,y) \vee R(y,x))\}$$

In the following, we write $x \in D_R$ as $D_R(x)$, and if the relation R is apparent from the context, write D_R as D.

Definition 2. A relation R with domain D is called a *search algorithm* if it is a strict linear order that is consistent with $<$, expands only executable situations, and includes S_0, that is, it satisfies the following axioms:

$$D(s) \supset \neg R(s,s), \tag{6}$$

$$D(s_1) \wedge D(s_2) \wedge D(s_3) \supset \{R(s_1,s_2) \wedge R(s_2,s_3) \supset R(s_1,s_3)\}, \tag{7}$$

$$D(s_1) \wedge D(s_2) \supset s_1 = s_2 \vee R(s_1,s_2) \vee R(s_2,s_1), \tag{8}$$

consistent with $<$:

$$D(s_2) \supset [s_1 \leq s_2 \supset R(s_1,s_2)] \tag{9}$$

expand only executable situations:

$$D(s) \supset \mathsf{Exec}(s), \tag{10}$$

includes S_0:

$$D(S_0). \tag{11}$$

Notice that axiom (9) implies that if $D(s)$ and $s' < s$, then $D(s')$. Notice also that axioms (9) – (11) imply that $D(s) \supset R(S_0, s)$. This means that for any search algorithm R, S_0 is the first situation that the algorithm will examine. The axioms also entail that if there is a next situation after S_0, it must be $do(A, S_0)$ for some action A, and the next after this one must be either $do(B, S_0)$ for some action $B \neq A$ or $do(A', do(A, S_0))$ for some action A', depending on, for example, whether the algorithm uses breadth-first or depth-first search strategy.

The following proposition is immediate:

Proposition 1. *Let D be the domain of R. If R is a search algorithm, then the restriction of R on a set $K \subseteq D$:*

$$R'(x,y) \equiv K(x) \wedge K(y) \wedge R(x,y)$$

is also a search algorithm. Notice that the domain of R' is a subset of K.

Given a search algorithm R, we can define a "next" relation Next_R such that $\mathsf{Next}_R(s,s')$ holds if s' is the situation immediately after s according to the linear order R:

$$\mathsf{Next}_R(s,s') \equiv R(s,s') \wedge \neg(\exists s'').R(s,s'') \wedge R(s'',s').$$

It's clear that for any s, there is at most one situation s' such that $\mathsf{Next}_R(s,s')$. Using Next, we can "execute" a search algorithm R as follows:

1. Let $S = S_0$;
2. If S is a goal situation, then exit with success;
3. If there is a situation S' such that $\mathsf{Next}_R(S, S')$, then let $S = S'$ and go back to step 2;
4. Exit with failure.

This procedure suggests that instead of a linear order, we can also capture a search algorithm by a "next" relation. Indeed these two approaches are essentially equivalent. We have seen how to define a "next" relation given a linear order. Conversely, we can also recover the essence of a linear order from a "next" relation by taking its transitive closure. However, we prefer to start with a linear order for two reasons. One is that for those search strategies that we have investigated, it is much easier and cleaner to capture their corresponding linear orders than their corresponding "next" relations. The other, more important reason is that linear orders provide a more general framework. This is because it is much easier to investigate various variants of a search strategy through a common linear order. For instance, as we shall see, depth first search with chronological backtracking and depth first search with backjumping share a common linear order. Similarly, it is much easier to explain search control and pruning by viewing a search algorithm as an ordering relation. Having said that, there are still some occasions when it is easier to characterize a search algorithm by its corresponding "next" relation, as we shall see when we consider depth-first search with backjumping.

It is clear that the key operation of the above procedure for executing a search algorithm is the computation of the partial function Next_R. So to implement a search algorithm, what we need to do is to find a way to compute this function. A generic way of doing this is as follows. Let S be an executable situation. To compute the next situation after S, do the following:

1. Compute the following set:
$$\mathcal{S} = \{s \mid R(S, s) \wedge \neg(\exists s')(s' < s \wedge R(S, s'))\}.$$
2. Sort $\mathcal{S}$ according to the linear order R.
3. If $\mathcal{S}$ is empty, then there is no next situation: $\neg(\exists s)\mathsf{Next}_R(S, s)$; otherwise, the first element S' in $\mathcal{S}$ is the next situation: $\mathsf{Next}_R(S, S')$.

The correctness of this procedure can be easily verified: If $\mathsf{Next}_R(S, S')$, then $R(S, S')$ and there are no situation in between S and S'. In particular, by (9), there does not exist an s such that $R(S, s)$ and $s < S'$. So $S' \in \mathcal{S}$. Conversely, if S' is the first element of $\mathcal{S}$, then $R(S, S')$. Furthermore, there cannot be any situation s such that $R(S, s)$ and $R(s, S')$. Otherwise, there is an s' such that $R(S, s')$, $s' \leq s$, and $\neg(\exists s^*)(s^* < s' \wedge R(S, s^*))$. Therefore $s' \in \mathcal{S}$. By the transitivity of R, we also have $R(s', S')$, this contradicts with the assumption that S' is the first element of $\mathcal{S}$.

Notice that theoretically, the condition $\neg(\exists s').s' < s \wedge R(S, s')$ in the definition of $\mathcal{S}$ is not necessary. It is included to keep the size of $\mathcal{S}$ down:

without it, $\mathcal{S}$ would be an infinite set; with it, $\mathcal{S}$ will be finite for problems with finite number of operators.

Notice also that for this procedure to work, there must be a systematic way of generating and maintaining $\mathcal{S}$. In particular, there must be an efficient way of checking whether $R(S, S')$ holds for any given two situations S and S'. For all search algorithms that we will consider in this paper, checking whether $R(S, S')$ can be done in time and space linear to the size of S and S'.

This procedure is in many ways similar to the implementation of search algorithms using an open and a closed list (see [13]) with $\mathcal{S}$ playing the role of the open list. A well-known limitation of this implementation is that it requires a lot of memory to store the open list. However, since what we really want is the first element of $\mathcal{S}$, there is no real reason why we have to keep the entire set around. All we need is to keep the current closest situation C, and each time a new situation s in $\mathcal{S}$ is discovered, it is compared with C in the following way: if $R(C, s)$, then we do nothing; if $R(s, C)$, then we replace C by s. This seems to be the idea used by many memory-bounded implementation of best-first search algorithms ([16,7]).

Another observation that we can make about the above procedure for executing a search algorithm is that not all executable situations will necessarily be visited by a search algorithm. Let us define $\mathsf{Reachable}_R$ to be the smallest relation satisfying the following two properties:

1. $\mathsf{Reachable}_R(S_0)$ and
2. if $\mathsf{Reachable}_R(s)$ and $\mathsf{Next}_R(s, s')$, then $\mathsf{Reachable}_R(s')$.

Formally,

$$\begin{aligned}&\mathsf{Reachable}_R(s) \equiv \\ &\qquad (\forall P).\{P(S_0) \wedge (\forall s_1, s_2)(P(s_1) \wedge \mathsf{Next}_R(s_1, s_2) \supset P(s_2))\} \supset P(s).\end{aligned}$$

Then a situation s will be visited by an algorithm R iff $\mathsf{Reachable}_R(s)$. Using $\mathsf{Reachable}_R$, we can define when a search algorithm is complete, and when it is optimal:

Definition 3. Given a problem with goal G, a search algorithm R is complete for this problem if

$$(\exists s)[S_0 \leq s \wedge H(G, s)] \supset (\exists s)[\mathsf{Reachable}_R(s) \wedge H(G, s)]. \tag{12}$$

It is optimal if

$$\mathsf{First}_R(s) \supset \mathsf{Optimal}(s), \tag{13}$$

where $\mathsf{First}_R(s)$ if s is the first solution returned by R:

$$\begin{aligned}&\mathsf{First}_R(s) \equiv \\ &\quad \mathsf{Reachable}_R(s) \wedge H(G, s) \wedge \neg(\exists s')(\mathsf{Reachable}_R(s') \wedge R(s', s) \wedge H(G, s')),\end{aligned}$$

and $\mathsf{Optimal}(s)$ if s is an optimal solution:

$$\mathsf{Optimal}(s) \equiv S_0 \leq s \wedge H(G,s) \wedge \neg(\exists s')(S_0 \leq s' \wedge H(G,s') \wedge g(s') < g(s)),$$

where g is the real valued cost function of the problem, and the relation "$<$" in $g(s') < g(s)$ has its usual meaning.

7 Depth-First Search

Depth-first search always chooses a node at the deepest level for expansion until it reaches a deadend, at that point it backtracks to a node at a shallower level. Like most search strategies, depth-first search stands for a family of search algorithms, depending on the order of actions used to expand a node and on the strategies used for backtracking.

Let $\prec$ be a strict linear order on actions. One way of implementing depth-first search with chronological backtracking is as follows:

1. Construct a tree with S_0 as the only node, and let $S = S_0$.
2. If S is a goal situation, then exit with success.
3. Extend the tree by adding the situations in $\{do(a,S) \mid Poss(a,S)\}$ as the children of S, and order them from left to right according to $\prec$: if $a \prec b$, then $do(a,S)$ must be to the left of $do(b,S)$.
4. Let S be the leftmost leaf situation of the tree such that the set
$$\{do(a,S) \mid Poss(a,S)\}$$
is not empty. If there is no such situation, then exit with failure.
5. Go back to step 2.

From this procedure, we can see that according to depth-first search, a situation s_1 is visited before s_2 if

1. s_1 and s_2 are on a same branch and $s_1 < s_2$; or
2. s_1 and s_2 share a common ancestor s and the branch that s_1 is on is to the left of the branch that s_2 is on.

This leads to the following axiom that captures the strict linear order for depth-first search:

$$\mathsf{DFS}(s_1,s_2) \equiv s_1 < s_2 \vee (\exists s,a,b)(do(a,s) \leq s_1 \wedge do(b,s) \leq s_2 \wedge a \prec b). \quad (14)$$

From DFS we get depth-first search with chronological backtracking DCB by restricting the domain of DFS to the set of executable situations:

$$\mathsf{DCB}(s_1,s_2) \equiv \mathsf{Exec}(s_1) \wedge \mathsf{Exec}(s_2) \wedge \mathsf{DFS}(s_1,s_2). \quad (15)$$

The following proposition shows that DCB so defined is indeed a search algorithm.

Proposition 2. *The relation* DCB *is a search algorithm.*

Proof. We need to show that properties (6)-(11) hold for DCB. First of all, it is easy to see that the domain of DCB is Exec:

$$(\exists s')(\mathsf{DCB}(s, s') \vee \mathsf{DCB}(s', s)) \equiv \mathsf{Exec}(s).$$

From this, properties (10) and (11) follow. Property (9) is also easy to see. In the following, all situations are assumed to be executable.

Property (6):

$$\neg s < s \wedge \neg(\exists s', a, b)(do(a, s') \leq s \wedge do(b, s') \leq s \wedge a \prec b).$$

The first conjunct holds because $<$ is a strict partial order (see [11] for a proof). The second conjunct holds because $\prec$ is a strict linear order on actions, and the fact that

$$do(a, s') \leq s \wedge do(b, s') \leq s \supset a = b.$$

Property (7): Suppose $\mathsf{DCB}(s_1, s_2)$ and $\mathsf{DCB}(s_2, s_3)$. We show that $\mathsf{DCB}(s_1, s_3)$. There are three cases:

1. $s_1 < s_2$ and $s_2 < s_3$: it follows that $s_1 < s_3$, thus $\mathsf{DCB}(s_1, s_3)$.
2. $s_1 < s_2$ and for some situation s and actions a and b,

$$do(a, s) \leq s_2 \wedge do(b, s) \leq s_3 \wedge a \prec b.$$

   Either $s_1 < do(a, s)$ or $do(a, s) \leq s_1$. In the first case, we have $s_1 \leq s$, so $s_1 < s_3$, thus $\mathsf{DCB}(s_1, s_3)$. In the second case, we have

$$do(a, s) \leq s_1 \wedge do(b, s) \leq s_3 \wedge a \prec b.$$

   Thus $\mathsf{DCB}(s_1, s_3)$.
3. $s_2 < s_3$ and for some situation s and actions a and b,

$$do(a, s) \leq s_1 \wedge do(b, s) \leq s_2 \wedge a \prec b.$$

   We have

$$do(a, s) \leq s_1 \wedge do(b, s) \leq s_3 \wedge a \prec b.$$

   Thus $\mathsf{DCB}(s_1, s_3)$.

Property (8): If either $s_1 \leq s_2$ or $s_2 \leq s_1$, then it is easy to see from (15) that (8). Suppose $\neg s_1 \leq s_2$ and $\neg s_2 \leq s_1$, then there must exist some situation s and actions $a \neq b$ such that

$$do(a, s) \leq s_1 \wedge do(b, s) \leq s_2.$$

Since $\prec$ is a strict linear order, either $a \prec b$ or $b \prec a$. In the former case, we have $\mathsf{DCB}(s_1, s_2)$, and the latter $\mathsf{DCB}(s_2, s_1)$.

The following proposition shows that DCB indeed corresponds to depth-first search with chronological backtracking:

Proposition 3. *Let S be an executable situation.*

1. *If for some action A,*

$$Poss(A,S) \wedge \neg(\exists a)(Poss(a,S) \wedge a \prec A), \tag{16}$$

   then $\mathsf{Next}_{\mathsf{DCB}}(S, do(A,S))$. This is the case when S is not a deadend.
2. *If for some actions A and B, and situation S',*

$$\neg(\exists a)Poss(a,S), \tag{17}$$
$$do(B,S') \leq S \wedge Poss(A,S') \wedge B \prec A, \tag{18}$$
$$\neg(\exists a).Poss(a,S') \wedge B \prec a \prec A, \tag{19}$$
$$\neg(\exists s,a,b).do(B,S') < do(a,s) \leq S \wedge Poss(b,s) \wedge a \prec b \tag{20}$$

   then $\mathsf{Next}_{\mathsf{DCB}}(S, do(A,S'))$. This is the case when search backtracks at S.
3. *Otherwise, S has no next situation: $\neg(\exists s)\mathsf{Next}_{\mathsf{DCB}}(S,s)$.*

Proof. **Case 1** Suppose (16). Clearly $\mathsf{DCB}(S, do(A,S))$. We show that

$$\neg(\exists s).\mathsf{DCB}(S,s) \wedge \mathsf{DCB}(s, do(A,S)).$$

Suppose otherwise, i.e. for some s, $\mathsf{DCB}(S,s) \wedge \mathsf{DCB}(s, do(A,S))$. There are three cases:

1. $S < s$ and $s < do(A,S)$: there cannot be such a situation s as $s < do(A,S)$ implies that $s \leq S$.
2. $S < s$ and $(\exists s', a, b).do(a,s') \leq s \wedge do(b,s') \leq do(A,S) \wedge a \prec b$. From $do(b,s') \leq do(A,S)$, there are two cases:
   (a) $s' = S$. In this case, $A = b$ and $a \prec A$. Since s is executable and $do(a,s') \leq s$, $Poss(a,S)$. This contradicts with our assumption (16).
   (b) $do(b,s') \leq S$. In this case, we have $\mathsf{DCB}(s,S)$, a contradiction with $S < s$.
3. $\mathsf{DCB}(s, do(A,S))$ and $(\exists s', a, b).do(a,s') \leq S \wedge do(b,s') \leq s \wedge a \prec b$. It follows that $do(a,s') \leq do(A,S)$. So $\mathsf{DCB}(do(A,S),s)$, a contradiction with $\mathsf{DCB}(s, do(A,S))$.

As we can see that none of these three cases are possible, so there cannot be a situation s such that $\mathsf{DCB}(S,s)$ and $\mathsf{DCB}(s, do(A,S))$. Thus $\mathsf{Next}_{\mathsf{DCB}}(S, do(A,S))$.

Case 2 Suppose (17) – (20) hold. It is clear that $\mathsf{DCB}(S, do(A,S'))$. We show that $\neg(\exists s)(\mathsf{DCB}(S,s) \wedge \mathsf{DCB}(s, do(A,S')))$. Suppose otherwise, i.e. for some s, both $\mathsf{DCB}(S,s)$ and $\mathsf{DCB}(s, do(A,S'))$. By (17), it is impossible that $S < s$. So for $\mathsf{DCB}(S,s)$ to hold, it must be the case that for some s' and a and b,

$$do(a,s') \leq S \wedge do(b,s') \leq s \wedge a \prec b.$$

This means that $s' < S$. Since we also have $S' < S$ from (18), so either $S' < s'$, $s' < S'$, or $s' = S'$:

1. $S' < s'$. This is impossible because it would imply $do(B, S') < do(a, s') \leq S$, a contradiction with (20). (Since $\mathsf{Exec}(s)$ and $do(b, s') \leq s$, we have $Poss(b, s')$ and $\mathsf{Exec}(s')$.)
2. $s' < S'$. In this case, we also have $do(a, s') \leq S'$, since both $do(a, s') \leq S$ and $S' < S$. Therefore $do(a, s') \leq do(A, S')$, thus $\mathsf{DCB}(do(A, S'), s)$, a contradiction with our assumption.
3. $s' = S'$. In this case, $a = B$. Since $\mathsf{DCB}(s, do(A, S'))$, we also have $\mathsf{DCB}(do(b, s'), do(A, S'))$. So $\mathsf{DCB}(do(b, S'), do(A, S'))$. This means that $b \prec A$. But we also have $B \prec b$, a contradiction with (19).

Case 3 Suppose neither Case 1 nor Case 2 hold, then $\neg(\exists s)\mathsf{Next}_{\mathsf{DCB}}(S, s)$. Suppose otherwise, i.e. for some s, $\mathsf{Next}_{\mathsf{DCB}}(S, s)$. Now if $(\exists a)Poss(a, S)$, then there must be some A such that $s = do(A, S)$, and the condition (16) holds. If $\neg(\exists a)Poss(a, S)$, then there must be an S' and two actions A and B such that

$$do(B, S') \leq S \wedge do(A, S') \leq s \wedge B \prec A.$$

In this case, $s = do(A, S')$, and the conditions (18)-(20) have to be true.

Notice here that DCB backtracks only when it hits a deadend, i.e. when none of the actions are executable. However, one can often tell that a situation has no hope of reaching a goal state before it hits a deadend. For instance, suppose our goal is to drive from one city to another, and we are in a situation where our car has run out of gas and there is no way for us to refuel it, then we can conclude that we can't achieve the goal although there are still some actions, such as kicking the car, that we can do.

Of course, this information is domain and goal dependent. In general, we can axiomatize it by a predicate $\mathsf{Backtrack}(s)$, meaning that s cannot lead to a goal situation:

$$\mathsf{Backtrack}(s) \supset \neg(\exists s').s \leq s' \wedge H(G, s'),$$

where G is the given goal. For example, in constraint satisfaction, if we have two constraints G_1 and G_2 to satisfy, then we can backtrack whenever one of them becomes false, although there may still be unassigned variables:

$$\neg G_1(s) \vee \neg G_2(s) \supset backtrack(s).$$

Once we have information about this predicate, then we can restrict search to the following set:

$$\mathsf{Exec}(s) \wedge \neg(\exists s').s' < s \wedge \mathsf{Backtrack}(s').$$

We have so far considered only depth-first search with chronological backtracking. We can also capture depth-first search with backjumping using the strict linear order DFS.

Although there is no precise definition of what backjumping is, the intuition is that if S is a deadend, then search should backtrack to the deepest level in the path from S_0 to S that may lead to a goal situation, given the information available from S. More precisely, given that

$$do([a_1, ..., a_k, ..., a_n], S_0)$$

is a deadend, if we know that for any $k < i \leq n$, there is no $s > do([a_1, ..., a_i], S_0)$ such that s is a goal situation, then there is no point backtracking to any of the situations $do([a_1, ..., a_i], S_0)$, for any $k < i \leq n$. Rather the search should backjump to $do([a_1, ..., a_k], S_0)$.

To axiomatize this intuition, we assume a predicate $\mathsf{Prune}(s_1, s_2)$ whose intuitive meaning is that if one needs to backtrack at s_1, then don't backtrack to s_2. In order not to lose any solution, we require it satisfy the following axiom:[4]

$$\mathsf{Prune}(s_1, s_2) \supset [\neg H(G, s_1) \supset \neg(\exists s)(s_2 < s \wedge H(G, s))], \tag{21}$$

where G is the given goal of the problem. With this predicate, we can capture depth-first search with backjumping by axiomatizing its next relation:

$$\begin{aligned}
&\mathsf{Next}(s_1, s_2) \equiv \\
&\qquad \neg\mathsf{Backtrack}(s_1) \supset \mathsf{Next}_{\mathsf{DCB}}(s_1, s_2) \wedge \\
&\qquad \mathsf{Backtrack}(s_1) \supset \{\mathsf{DCB}(s_1, s_2) \wedge \neg(\exists s)(s < s_2 \wedge \mathsf{Prune}(s_1, s)) \wedge \\
&\qquad\quad \neg(\exists s_2')[\mathsf{DCB}(s_1, s_2') \wedge \mathsf{DCB}(s_2', s_2) \wedge \neg(\exists s)(s < s_2' \wedge \mathsf{Prune}(s_1, s))]\}.
\end{aligned}$$

In other words, if s_1 is not a backtrack point, then backjumping is the same as chronological backtracking. If s_1 is a backtrack point, then the next situation should be the first situation s_2 (according to the order of DCB) which is not a descendent of a situation that should be pruned.

Notice that for any situations s and s', if $\mathsf{Next}(s, s')$, then $\mathsf{DCB}(s, s')$; furthermore, for any situation s'' such that $\mathsf{DCB}(s, s'') \wedge \mathsf{DCB}(s'', s')$, s'' cannot be a goal situation. So backjumping will not miss any solution that can be returned by chronological backtracking. In this sense, backjumping is indeed a provably correct improvement on chronological backtracking. Of course, this does not mean that the former is always preferred over the latter as one has to take into account the cost of computing Prune.

Notice that if we use Next to define $\mathsf{Reachable}$ as in Section 6, and define

$$\mathsf{DBJ}(s_1, s_2) \equiv \mathsf{Reachable}(s_1) \wedge \mathsf{Reachable}(s_2) \wedge \mathsf{DCB}(s_1, s_2),$$

then the next relation defined using DBJ, $\mathsf{Next}_{\mathsf{DBJ}}$, is exactly the same one as the Next relation defined above. So DBJ can be considered to be DCB together with a pruning strategy that prunes some redundant branches of the search tree.

[4] One could just let $\mathsf{Prune}(s_1, s_2)$ be $\neg(\exists s)(s_2 < s \wedge H(G, s))$. But then computing Prune will be the same as deciding if the goal is achievable or not, which is a very difficult task.

8 Breadth-First Search

In contrast to depth-first search, breadth-first search expands nodes at the same level first before expanding nodes at a deeper level.

The strict linear order BFS for breadth-first search is best defined inductively. In the base case, we know that S_0 needs to be the first situation:

$$S_0 < s \supset \mathsf{BFS}(S_0, s), \tag{22}$$

$$\neg\mathsf{BFS}(s, S_0), \tag{23}$$

Inductively, for two executable situations $do(a, s_1)$ and $do(b, s_2)$, the former is before the latter iff s_1 is before s_2 or $s_1 = s_2$ and $a \prec b$:

$$\begin{aligned}\mathsf{BFS}(do(a, s_1), do(b, s_2)) \equiv{}& \mathsf{Exec}(do(a, s_1)) \wedge \mathsf{Exec}(do(b, s_2)) \wedge \\ & \mathsf{BFS}(s_1, s_2) \vee [s_1 = s_2 \wedge a \prec b].\end{aligned} \tag{24}$$

Proposition 4. BFS *is a search algorithm.*

Proof. Properties (10) and (11) hold trivially because the domain of BFS is Exec. Again, in the following, we assume that all situations are executable.

Property (6): We prove by induction. Clearly, $\neg\mathsf{BFS}(S_0, S_0)$. Inductively, suppose $\neg\mathsf{BFS}(s, s)$. We show that $\neg\mathsf{BFS}(do(a, s), do(a, s))$. This sentence is equivalently to

$$\neg(s = s \wedge a \prec a) \wedge \neg\mathsf{BFS}(s, s)$$

which follows from inductive assumption and the assumption that $\prec$ is a strict linear order. This proves the inductive case, thus (6).

Property (7): we prove it by simultaneous induction on s_1, s_2, and s_3 using the following property:

$$\begin{aligned}& s_1 = S_0 \vee s_2 = S_0 \vee s_3 = S_0 \vee \\ & \quad (\exists s_1', s_2', s_3', a, b, c).s_1 = do(a, s_1') \wedge s_2 = do(b, s_2') \wedge s_3 = do(c, s_3').\end{aligned}$$

If

$$s_1 = S_0 \vee s_2 = S_0 \vee s_3 = S_0,$$

then (7) follows from (22) and (23). Inductively, assume that

$$\mathsf{BFS}(s_1, s_2) \wedge \mathsf{BFS}(s_2, s_3) \supset \mathsf{BFS}(s_1, s_3),$$

we show that

$$\mathsf{BFS}(do(a, s_1), do(b, s_2)) \wedge \mathsf{BFS}(do(b, s_2), do(c, s_3)) \supset \mathsf{BFS}(do(a, s_1), do(c, s_3)) \tag{25}$$

Suppose $\mathsf{BFS}(do(a, s_1), do(b, s_2))$ and $\mathsf{BFS}(do(b, s_2), do(c, s_3))$:

$$[s_1 = s_2 \wedge a \prec b] \vee \mathsf{BFS}(s_1, s_2),$$

$$[s_2 = s_3 \wedge b \prec c] \vee \mathsf{BFS}(s_2, s_3)$$

we have

$$[s_1 = s_3 \wedge a \prec c] \vee \mathsf{BFS}(s_1, s_3)$$

by the inductive assumption, and the fact that $\prec$ is a strict linear order. This proves the inductive case, thus (7).

Property (8): We prove by simultaneous induction on s_1 and s_2. If $s_1 = S_0 \vee s_2 = S_0$, then (8) follows from (23). Now inductively, suppose (8) holds for s_1 and s_2, we show that it holds for $do(a, s_1)$ and $do(a, s_2)$:

$$do(a, s_1) = do(b, s_2) \vee \mathsf{BFS}(do(a, s_1), do(b, s_2)) \vee \mathsf{BFS}(do(b, s_1), do(a, s_2))$$

is equivalent to

$$\begin{aligned}&[a = b \wedge s_1 = s_2] \vee \\ &[s_1 = s_2 \wedge a \prec b] \vee \mathsf{BFS}(s_1, s_2) \vee \\ &[s_1 = s_2 \wedge b \prec a] \vee \mathsf{BFS}(s_2, s_1),\end{aligned}$$

which follows from the inductive assumption and the fact that $\prec$ is a strict linear order.

Property (9): by induction on s_1 and s_2 using the following facts about $<$:

$$\neg s < S_0 \wedge do(a, s_1) < do(b, s_2) \supset s_1 < s_2.$$

Now consider the procedure for "executing" a search algorithm given in Section 6. When applied to BFS, it is as follows:

1. Let $S = S_0$;
2. If S is a goal situation, then exit with success;
3. If there is a situation S' such that $\mathsf{Next}_{\mathsf{BFS}}(S, S')$ then let $S = S'$ and go back to step 2;
4. Exit with failure.

As we mentioned, the key operation in this procedure is the computation of $\mathsf{Next}_{\mathsf{BFS}}$. The following proposition shows that we can actually use depth-first search to compute $\mathsf{Next}_{\mathsf{BFS}}$. In the following, for any situation S, we define the *depth* of S, written $\mathsf{Depth}(S)$, to be the number of actions in the sequence of actions L such that $S = do(L, S_0)$. Obviously, $\mathsf{Depth}(S)$ is a unique finite number.

Proposition 5. *Let S be an executable situation with depth n.*

1. *If S' is the first executable situation with depth n such that* $\mathsf{DCB}(S, S')$*:*

$$\mathsf{Depth}(S') = n \wedge \mathsf{DCB}(S, S'),$$
$$\neg(\exists s).\mathsf{Depth}(s) = n \wedge \mathsf{DCB}(S, s) \wedge \mathsf{DCB}(s, S'),$$

then $\mathsf{Next}_{\mathsf{BFS}}(S, S')$.

2. *If there is no such S', and S'' is the first executable situation of depth $n+1$ after S according to* DCB*:*

$$\mathsf{Depth}(S'') = n+1 \wedge \mathsf{DCB}(S, S''),$$
$$\neg(\exists s).\mathsf{Depth}(s) = n+1 \wedge \mathsf{DCB}(S, s) \wedge \mathsf{DCB}(s, S''),$$

then $\mathsf{Next}_{\mathsf{BFS}}(S, S'')$.

3. *Otherwise, there is no next situation after S:* $\neg(\exists s)\mathsf{Next}_{\mathsf{BFS}}(S, s)$.

Proof. We do induction on n, the depth of S. If $n = 0$, this means that $S = S_0$. It is easy to see that $\mathsf{Next}_{\mathsf{BFS}}(S_0, S')$ iff for some action A,

$$S' = do(A, S_0) \wedge Poss(A, S_0) \wedge \neg(\exists a)(Poss(a, S_0) \wedge a \prec A)$$

iff $\mathsf{Next}_{\mathsf{DCB}}(S_0, S')$. From these, it is easy to see that the result holds for S_0.

For the inductive case, suppose that the results hold for n. We show that they hold for $n+1$ as well. First of all, we have

$$(\exists a)(A \prec a \wedge Poss(a, S)) \supset \{\mathsf{Next}_{\mathsf{BFS}}(do(A, S), do(B, S')) \equiv$$
$$S = S' \wedge Poss(B, S) \wedge A \prec B \wedge \neg(\exists a)(A \prec a \prec B \wedge Poss(a, S))\}.$$

and

$$\neg(\exists a)(A \prec a \wedge Poss(a, S)) \supset \{\mathsf{Next}_{\mathsf{BFS}}(do(A, S), do(B, S')) \equiv$$
$$\mathsf{Next}_{\mathsf{BFS}}(S, S') \wedge Poss(B, S') \wedge \neg(\exists a)(Poss(a, S') \wedge a \prec B)\}.$$

So

$$\mathsf{Next}_{\mathsf{BFS}}(s, s') \supset \mathsf{Depth}(s') = \mathsf{Depth}(s) \vee \mathsf{Depth}(s') = \mathsf{Depth}(s) + 1.$$

From this, and the following two facts:

$$\mathsf{BFS}(s, s') \supset \mathsf{Depth}(s) \leq \mathsf{Depth}(s'),$$
$$\mathsf{Depth}(s) \leq \mathsf{Depth}(s') \wedge \mathsf{DCB}(s, s') \supset \mathsf{BFS}(s, s'),$$

the results follow.

If we use this proposition to compute $\mathsf{Next}_{\mathsf{BFS}}$, then we'll get essentially iterative-deepening search. So under our interpretation of search algorithm, iterative-deepening is not really a new search strategy, but a way of implementing breadth-first search. It trades time for space.

9 Best-First Search

Given a search problem, let h be its heuristic function, g its path cost function, and f an evaluation function on situations composed of h and g. The idea behind best-first search is rather simple: at any point, the search always expands a situation with the smallest f value. Notice that sometimes there may be more than one situations all having the same smallest f value. In this case, a "tie-breaking" rule is needed. In the following, we assume that Pref is a strict linear order on situations used as a "tie-breaker".

Let's denote by Heur the linear order corresponding to best-first search. Since by definition, Heur needs to be transitive and consistent with $<$, so for any situations s_1 and s_2, if there is a situation s such that $s < s_2$ and $\mathsf{HeurEq}(s_1, s)$, then $heur(s_1, s_2)$, where for any s' and s'', $\mathsf{HeurEq}(s', s'')$ is a shorthand for $\mathsf{Heur}(s', s'') \vee s' = s''$. If there is no such s, then $\mathsf{Heur}(s_1, s_2)$ if either $f(s_1) < f(s_2)$ or $f(s_1) = f(s_2)$ and s_1 is before s_2 according to the tie-breaking rule:

$$\begin{aligned}\mathsf{Heur}(s_1, s_2) \equiv\ & \mathsf{Exec}(s_1) \wedge \mathsf{Exec}(s_2) \wedge \\ & (\exists s).s < s_2 \wedge \mathsf{HeurEq}(s_1, s) \vee \\ & \neg(\exists s).s < s_1 \wedge \mathsf{HeurEq}(s_2, s) \wedge \\ & \quad [f(s_1) < f(s_2) \vee (f(s_1) = f(s_2) \wedge \mathsf{Pref}(s_1, s_2))]\end{aligned} \tag{26}$$

Proposition 6. Heur *is a search algorithm.*

Proof. The proof of this proposition is very tedious, and is omitted here for lack of space.

Recall that according to Definition 3, given a search problem with goal G and cost function g, a search algorithm R is optimal if the first solution returned by it is always an optimal solution:

$$\mathsf{First}_R(s) \supset \mathsf{Optimal}(s).$$

For best-first search, the following proposition shows that it is optimal if it always explores goal situations with less costs.

Proposition 7. *If*

$$\mathsf{Exec}(s_1) \wedge \mathsf{Exec}(s_2) \wedge H(G, s_1) \wedge H(G, s_2) \wedge g(s_1) < g(s_2) \supset \mathsf{Heur}(s_1, s_2)$$

then Heur *is optimal.*

Proof. Let s be any situation such that $\mathsf{First}_{\mathsf{Heur}}(s)$. We show that $\mathsf{Optimal}(s)$ holds as well. Suppose otherwise, then by the definition of $\mathsf{Optimal}$ in Definition 3, there must be a situation s' such that

$$H(G, s') \wedge S_0 \leq s' \wedge g(s') < g(s).$$

By the assumption in the proposition, $\mathsf{Heur}(s', s)$. Since $\mathsf{First}_{\mathsf{Heur}}$, we have $\mathsf{Reachable}(s)$. Thus $\mathsf{Reachable}(s')$, a contradiction with $\mathsf{First}_{\mathsf{Heur}}(s)$.

The following proposition says that if Heur is complete, i.e. always returns a solution when one exists, then the condition in the above proposition is also a necessary condition for Heur to be optimal for all search problems.

Proposition 8. *If* Heur *is complete:*

$$(\exists s)(S_0 \leq s \wedge H(G,s)) \supset (\exists s)\mathsf{First}_{\mathsf{Heur}}(s),$$

then Heur *is optimal for all search problems iff*

$$\mathsf{Exec}(s_1) \wedge \mathsf{Exec}(s_2) \wedge H(G,s_1) \wedge H(G,s_2) \wedge g(s_1) < g(s_2) \supset \mathsf{Heur}(s_1,s_2).$$

Proof. The sufficiency of this condition has been proved in last proposition. Now suppose the condition is not true, i.e. there are two situations s_1 and s_2 such that

$$S_0 \leq s_1 \wedge S_0 \leq s_2 \wedge H(G,s_1) \wedge H(G,s_2) \wedge g(s_1) < g(s_2) \wedge \neg\mathsf{Heur}(s_1,s_2).$$

By the connectedness of Heur, we have have $\mathsf{Heur}(s_2,s_1)$. Now define a new goal G' such that $H(G',s) \equiv s = s_1 \vee s = s_2$. By the completeness of Heur, for this new search problem, $\mathsf{First}(s_2)$. This means that Heur is not optimal for this new problem.

Using the first proposition, we can show that if the cost function is additive and the heuristic function h is admissible, then the resulting Heur with evaluation function $f(s) = g(s) + h(s)$, the traditional A^* algorithm, is optimal.

We can also show that among all complete and optimal Heur algorithms for admissible problems, A^* expands the least set of situations. We will not give precise proofs here as these results are not new and can be found in [6].

Finally, just like iterative deepening can be thought of as an implementation of breadth-first search, many memory bounded A* variants such as iterative deepening A* can be considered to be special strategies for implementing A*. In fact, Korf [7] shows that if f is monotonically increasing along every path, then iterative deepening A* expands nodes in best-first order. Korf also shows that his recursive procedure called RBFS always expands nodes in best-first order as well.

10 Conclusions

We have presented a situation calculus account of search algorithms by treating them as strict linear ordering relations. Although AI search is a more mature area of AI, we think it can still benefit from such a formal account. In particular, we believe this account will be good for proving properties of various search methods, for comparing them, for finding new ways of implementing them, and for studying various search control and pruning strategies such as backjumping and dependency-based backtracking.

Acknowledgements

This work was supported in part by the Research Grants Council of Hong Kong under Competitive Earmarked Research Grants HKUST6091/97E and HKUST6145/98E.

References

1. A. Bundy and L. Sterling. Meta-level inference: two applications. *Journal of Automatic Reasoning*, 4(1):15–27, 1988.
2. K. Currie and A. Tate. O-Plan: the open planning architecture. *Artificial Intelligence*, 51(1):49–86, 1991.
3. R. Davis. Meta-rules: Reasoning about control. *Artificial Intelligence*, 15(3):179–222, 1980.
4. R. Davis and B. G. Buchanan. Meta-level knowledge: overview and applications. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI–77)*, pages 920–927, 1977.
5. T. Dean, J. Allen, and Y. Aloimonos. *Artificial intelligence : theory and practice.* Menlo Park, Calif. : Addison-Wesley Pub. Co., 1995.
6. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of ACM*, pages 505–536, 1985.
7. R. E. Korf. Linear-space best-first search. *Artificial Intelligence*, 62(1):41–78, 1993.
8. F. Lin. An ordering on subgoals for planning. *Annals of Mathematics and Artificial Intelligence*, 21:321–342, 1997.
9. F. Lin. Applications of the situation calculus to formalizing control and strategic information: the prolog cut operator. *Artificial Intelligence*, 103:273–294, 1998.
10. F. Lin. On measuring plan quality. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 224–233, 1998.
11. F. Lin and R. Reiter. State constraints revisited. *Journal of Logic and Computation, Special Issue on Actions and Processes*, 4(5):655–678, 1994.
12. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, Edinburgh, 1969.
13. N. J. Nilsson. *Principles of Articifial Intelligence.* Morgan Kaufmann, Los Altos, CA., 1980.
14. R. Reiter. Proving properties of states in the situation calculus. *Artificial Intelligence*, 64:337–351, 1993.
15. R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamic Systems.* In preparation, 1997.
16. S. Russell. Efficient memory-bounded search methods. In *Proceedings of ECAI'92*, pages 1–5, 1992.
17. S. J. Russell and P. Norvig. *Artificial intelligence : a modern approach.* Englewood Cliffs, N.J. : Prentice Hall, 1995.