Dipartimento di elettronica e informazione
Politecnico di Milano

Artificial Intelligence 2010-11

5. Search strategies

5.1 How search strategies work

Search strategies[1] are strategies for finding solutions of state space problems. They typically work by building a *search tree*; every *node* of the search tree represents a *state*, plus additional information that includes:

- a reference to the *parent node* in the tree;
- a symbol identifying the *action* that produced the state represented by the node;
- a real number representing the *total cost* of the path from the root to the current node (when costs are identically equal to 1, then the total cost coincides with the *depth* of the node in the tree).

Informally, a search strategy builds a search tree as follows:

- the *root node* of the tree represents the initial state of the problem (the reference to the parent node and the symbol identifying the action that produced the state are null, and the total cost is zero);
- the actions defined in the state space are used to *expand* nodes belonging to the *frontier* of the search tree (the frontier is the list of all the *leaves* of the current search tree, that is, of all nodes that have been generated but not yet expanded); to expand a node *n* means to generate all its *successor nodes*, that is, the nodes that represent the results of executing all applicable actions on the state represented by *n*;
- when a leaf of the tree is found that represents a goal state, a solution of the problem is generated by following the unique path connecting the root to such a leaf.

Tree vs. graph search

States should never be confused with nodes: every node represents exactly one state, but in the same search tree the same state may be represented by different nodes. When they generate a new node, the simplest search strategies do not check whether other nodes representing the same state have already been processed; such strategies are called *tree-search strategies*. Strategies that do check (and take appropriate action) are called *graph-search strategies*. Note, however, that both tree-search and graph-search strategies produce *search trees* (not search graphs!).

Parallel vs. sequential strategies

The whole idea of search is based on the fact that, at every state, typically *more than one action is applicable*: this is why a search process produces a tree. However, there are two basically different ways of producing a tree: one may proceed *in parallel*, carrying on the search process in all possible directions at the same time, or *sequentially*, carrying on the search process in one direction at a time, and resuming the search along other directions only when it is necessary.

---

[1] The reader is referred to the recommended textbook, Sections 3.3 to 3.5, for details on the most significant search strategies.

Uninformed vs. informed strategies

Every search strategy must have access at least to the *description of a problem* (in the technical sense defined in the Section 4). Any strategy that does not rely on further information to solve the problem is said to be *uninformed*.

Some search strategies access further information, which is not part of the description of the problem, but may help to find a solution. In such cases the strategy is said to be *informed*. Note that, in general, such additional information is exploited to achieve *secondary optimisation*, not *primary optimisation*; in other words, the additional information is used *to optimise the process of finding a solution*, not *to optimise the solution itself*.

Classifying some search strategies

A strategy may have both a tree-search and a graph-search version. As far as the other two dimensions are concerned, the most important search strategies can be classified as follows:

	parallel	*sequential*
uninformed	BF (breadth first) UC (uniform cost)	DF (depth first) BT (backtracking) ID (iterative deepening)
informed	A*	GBF (greedy best first)

5.3 The basic search strategies

In what follows, the components of a state space problem are underlined. We first define a node of the search tree as a structure:

```
struct  node  =  [currentState: a state, i.e., a data structure representing an element of States;
                  pastAction: an action, i.e., a symbol identifying an element of Actions;
                  parentNode: a node, i.e., a reference to an instance of this structure;
                  totalCost: a number];
```

We need functions to manipulate lists of elements:

```
function isEmpty(list: a list of elements of type T)
returns (a Boolean) {
    if (list has no element)
        true
    else false;
}
```

```
function insertFirst(x: an element of type T; list: a list of elements of type T)
returns (a list of elements of type T) {
    add x to list as the first element;
    return list;
}
```

```
function insertLast(x: an element of type T; list: a list of elements of type T)
returns (a list of elements of type T) {
    add x to list as the last element;
    return list;
}
```

```
function insertList(list1, list2: two lists of elements of type T)
returns (a list of elements of type T) {
    add all the nodes of list1 to list2 according to a policy to be specified;
    return list2;
}
```

```
function removeFirst(list: a list of elements of type T)
returns (an element of type T) {
    vars   x: an element of type T;
    x = the first element of list;
    remove the first element of list;
    return x;
}
```

Tree search

```
function treeSearch(initialState: a state; Actions: a list of actions; GoalStates: a set of states)
returns (a list of actions or 'failure') {
    vars {
        n: a node;
        frontier: a list of nodes (initially empty);
    }
    n = new node(initialState,null,null,0);
    insertFirst(n,frontier);
    loop {
        if isEmpty(frontier) return 'failure';
        n = removeFirst(frontier);                                  \\ comment 1
        if (n.currentState in GoalStates) return buildSolution(n);  \\ comment 2
        insertList(expandNode(n,Actions),frontier);                 \\ comments 3 and 4
    }
}
```

```
function buildSolution(n: a node) returns a list of actions {
    vars { a: an action;
           solution: a list of actions (initially empty);
    }
    while (n != null) {
        insertFirst(n.pastAction,solution);                         \\ comment 5
        n = n.parentNode;
    }
    return solution;
}
```

```
function expandNode(n: a node; Actions: a list of actions) returns a list of nodes {
    vars { newNode: a node;
           a: an action;
           successorList: a list of nodes (initially empty);
    }
    for (a in Actions) {
        if applicable(a,n.currentState) {
            newNode = new node(result(a,n.currentState), a, n, n.totalCost+cost(a,n.state));
            insertLast(newNode,successorList);                          \\ comments 6 and 7
    }
    return successorList;
}
```

Comments

1. All the burden of implementing different search strategies is placed on the policy by which newly generated nodes are *inserted* in the frontier. This allows us to extract nodes always from the same side of the frontier (from the beginning, in our case).
2. To implement different search strategies, insertList will have to implement different insertion policies (see Section 5.4).
3. A node is tested for being a goal *when it is extracted from the frontier to be expanded*, and not when it is generated. This is required in order for *some* search strategies to be optimal (i.e., to return an optimal solution). Other search strategies are optimal also if a node is tested for being a goal when it is generated, before it is inserted in the frontier; in such cases, as we shall see, the search process becomes more efficient if the function is modified so that a node is tested for being a goal *immediately after it is generated*.
4. What happens if the expanded node has no successors? Obviously, no new node will be added to the frontier; less obviously, the expanded node will now be a "stray node," in the sense that no node of the search tree refers to it (as its parent node). Given that the node has also been removed from the frontier, there is now no way to access it; at this point, it is reasonable to destroy the node and free the corresponding memory space (this operation has not been explicitly represented in the pseudo-code).
5. The use of insertFirst allows us to generate an action list in which the order of action executions will be preserved.
6. We use the list frontier to store the set of all nodes that have already been generated, but not yet expanded. But in what data structure is the whole search tree stored? The answer is, we do not need any additional data structure: the structure of the search tree is represented by the set of all references to parent nodes. This means that the search tree is represented as a *reverse tree*, that is, as a tree where references point *bottom up* (from successors to their parents) and not *top down* (from parents to their successors). But this is just what we need.
7. The use of insertLast allows us to generate a node list that preserves the order of creation of the successors of n (this may be important for implementing certain strategies).

Graph search

Additions to the treeSearch function are shown in *italics*.

```
function graphSearch(initialState: a state; Actions: a list of actions; GoalStates: a set of states)
returns (a list of actions or 'failure') {
    vars {
        n: a node;
        frontier: a list of nodes (initially empty);
        explored: a set of states (initially empty);
    }
    n = new node(initialState,null,null,0);
    insertFirst(n,frontier);
    loop {
        if isEmpty(frontier) return 'failure';
        n = removeFirst(frontier);
        addToSet(n.state,explored);
        if (n.currentState in GoalStates) return buildSolution(n);
        insertList(graphExpandNode(n,Actions,explored),frontier);
    }
}
```

```
function graphExpandNode(n: a node; Actions: a list of actions; explored: a set of states)
returns a list of nodes {
    vars { newNode: a node;
           a: an action;
           successorList: a list of nodes (initially empty);
    }
    for (a in Actions)
        if applicable(a,n.currentState) {
            newNode = new node(result(a,n.currentState),a,n,n.totalCost+cost(a,n.state));
            if !(newNode.state in explored)
                insertLast(newNode,successorList);
        }
    return successorList;
}
```

5.3 Some facts about trees

To understand the complexity of search strategies it is necessary to remember a few facts about trees. The two factors determining the number of nodes in a tree T are its *depth* d (i.e., the number of *levels* in T) and its *average branching factor* b (i.e., the mean number of successors of every internal node). Of course, b is typically greater than 1 (if is equal to 1, the tree degenerates into a linear list).

Levels are counted from 0 (the root's level). If we assume that all levels of T are complete (i.e., they contain all nodes that can belong to that level), then level k contains b^k nodes. Therefore (see Figure 5.1), the total number of nodes (internal nodes + leaves) is

$$Total(b,d) = \sum_{k=0}^{k=d} b^k = \frac{b^{d+1}-1}{b-1}$$

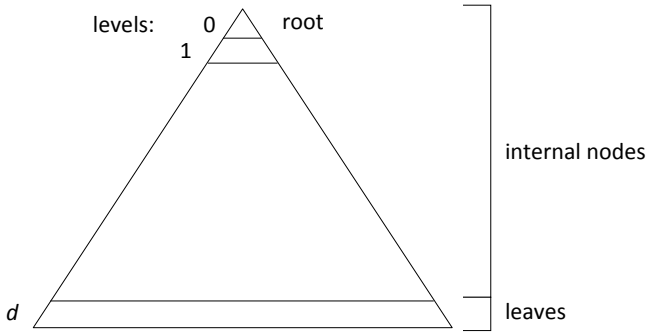

Figure 5.1 A tree of depth *d*.

This formula can be derived as follows:

(1) $$\sum_{k=0}^{k=d+1} b^k = b^0 + b \cdot \sum_{k=0}^{k=d} b^k = 1 + b \cdot \sum_{k=0}^{k=d} b^k$$

(2) $$\sum_{k=0}^{k=d+1} b^k = \sum_{k=0}^{k=d} b^k + b^{d+1}$$

From (1) and (2):

(3) $$\sum_{k=0}^{k=d} b^k + b^{d+1} = 1 + b \cdot \sum_{k=0}^{k=d} b^k$$

From (3):

$$\sum_{k=0}^{k=d} b^k = \frac{b^{d+1}-1}{b-1}$$

The total number of internal nodes is

$$Internal(b,d) = \sum_{k=0}^{k=d-1} b^k = \frac{b^d-1}{b-1}$$

and the total number of leaves is

$$Leaves(b,d) = b^d = (b-1) \cdot Internal(b,d) + 1$$

The last formula is particularly interesting: neglecting the "+ 1" term, it says that expanding one more level of a tree costs $b-1$ times as much as expanding *all previous levels*. Thus for $b>2$ the number of leaves is greater than the number of internal nodes; for example, with $b = 5$ and $d = 10$ we have:

$$Total(5,10) = 12{,}207{,}031$$
$$Internal(5,10) = 2{,}441{,}406$$
$$Leaves(5,10) = 9{,}765{,}625$$

Clearly, this implies that when generating a tree all measures should be taken to expand as few levels as possible. Finally, note that the ratio of the total number of nodes to the number of leaves is

$$\frac{b^{d+1}-1}{b-1} \cdot \frac{1}{b^d} = \frac{b}{b-1} - \frac{1}{b^d(b-1)}$$

which, for sufficiently large values of *d*, is approximately equal to

$$\frac{b}{b-1}$$

This means that the total number of nodes in a tree has (approximately) a constant ratio to the number of leaves, and such a ratio becomes smaller for larger values of the branching factor. For example, with $b = 2$ this ratio is 2; for $b = 5$ (see the previous example) the ratio is 1.25; and for $b = 11$ the ratio is 1.1.

5.4 Implementing uninformed search strategies

Different search strategies can be implemented by adopting the basic algorithms and specifying the insertion policies of function insertList. In many cases, this will not deliver a fully optimised version of the strategy. For example, in some cases a node can be tested for representing a goal state immediately after it is generated, instead of testing it when it is picked up from frontier for expansion; when this is possible, fewer leaves are generated. From a conceptual point of view, however, this is a marginal point: after choosing a search strategy it is not difficult to modify the search function so that the number of generated nodes is minimised (as an example, we show how to do this for the BF strategy below).

We now present the most important uninformed search strategies (see Section 6 for an informed strategy).

Breadth-first search

A breadth-first search strategy (BF) generates the search tree level by level, that is, all nodes belonging to the level k of the search tree are expanded before a node of level k+1 is expanded.

To implement BF, the insertList function has to follow a FIFO policy: given that nodes are removed from the *beginning* of the frontier, this policy is achieved by inserting the successor nodes to the *end* of the frontier; that is, the frontier is managed as a *queue*.

BF can be implemented in the tree-search or graph-search version. Both strategies are:

- *complete*, because any solution (if it exists) will have some finite length d, and therefore will be found when the strategy expands the nodes belonging to level d of the search tree;
- *optimal* if all costs are identical, because in such cases: (i), the total cost of a solution is equal to its length, and thus the optimal solutions coincide with the shortest solutions; and (ii), the solution returned by BF is a shortest solution.

Note that BF is *not* optimal if the action costs are not all identical: in such a case, an optimal solution need not be a shortest solution: but BF always returns a shortest solution. The *uniform cost strategy* (UC, see below) is a modified version of BF that is optimal also when costs are not identical.

As far as complexity is concerned, let us assume that d is the length of a shortest solution. Remember that according to our strategy a goal can be detected when a node is *expanded*, not when it is *generated*. In the average, therefore, half of level d of the search tree will have to be expanded before a goal is detected, and this implies that, again in the average, half of level d+1 of the search tree will have to be generated. Given that generating a new node takes most of the time and memory necessary to run the strategy, both the time and the memory required by BF are roughly proportional to b^{d+2} (see Section 5.3):

$$\text{time(BF)} \propto b^{d+2},$$
$$\text{memory(BF)} \propto b^{d+2}.$$

When costs are all identical, however, it is possible to modify the search algorithm and check for goal nodes when nodes are *generated*, instead of waiting for them to be *expanded*. This will cut down the time and memory required by BF to

$$\text{time(BF)} \propto b^{d+1},$$
$$\text{memory(BF)} \propto b^{d+1}.$$

It should be noted that with other search strategies this modification may prevent us from finding an optimal solution (see for example strategy UC below).

Depth-first search

A depth-first search strategy (DF) generates the search tree by expanding the most recently generated nodes first. To implement DF, the insertList function has to follow a LIFO policy: given that nodes are removed from the *beginning* of the frontier, this policy is achieved by inserting the successor nodes to the *beginning* of the frontier. That is, the frontier is managed as a *stack*.

DF can be implemented in the tree-search or graph-search version. Both strategies are:

- *incomplete* if the search tree contains infinite branches, but *complete* if this is not the case (if the search tree does not contain infinite branches, the whole search tree is finite and therefore has a finite maximum depth, *m*);
- *not optimal* even when they are complete: in general, nothing guarantees that the first solution found by the strategy is optimal, not even if all costs are identically equal to 1 (see Figure 5.2 for an intuitive counterexample).

There is, however, a remarkable class of problems for which DF is both complete and optimal. Indeed, it is sufficient to assume that:

- the search tree is finite (and therefore DF is complete);
- all action costs are identical;
- all solutions have the same length.

These conditions may appear unrealistic, but in fact they are satisfied by an important class of problems, known as *constraint satisfaction problems* (we shall deal with these problems later on in the course). When DF is complete, and therefore the search tree has maximum depth *m*, the time complexity of DF is:

$$\text{time(DF)} \propto b^{m+1}.$$

As far as the memory complexity of DF is concerned, let us first consider the case of tree search (without the explored set). In such a case,

$$\text{memory(DF)} \propto bm,$$

that is, the memory requirements are linear in *m*. This result assumes that every node to which no other node refers is destroyed, and the corresponding memory space is freed (see Section 5.2, Comment 3). On the other hand, if we use graphSearch we have to take into account the memory required to store the explored set; but this is not easy to estimate, because it depends on the frequency with which states are multiply generated.

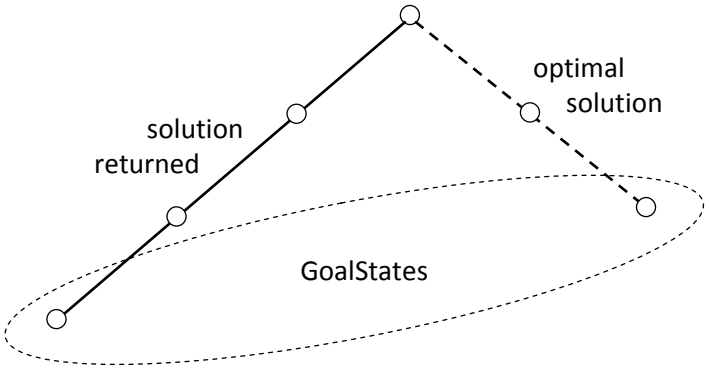

Figure 5.2. DF is not optimal.

Just to give a concrete example, let us consider the 8-puzzle. The total number of states is 9! = 362,880, but the state space turns out to be made up of two disconnected components, each containing 9!/2 = 181,440 states. This means that starting from any state, only 181,440 different states can be reached. Now take a problem which admits of an optimal solution of length 20 (i.e., the goal state can be reached by making 20 moves from the initial state). Given that the average branching factor of the 8-puzzle is $b = 2.7$, building a search tree of depth 20 will generate 673,271,337 nodes. However, using DF at most $2.7 \cdot 20 = 54$ nodes will have to be stored to represent the search tree. Now, what about the size of explored? This is difficult to estimate, but in any case it cannot exceed 181,440 states, because this is the maximum number of different states that can be reached from any initial state.

Of course, we know that for a case like the 8-puzzle DF is neither complete nor optimal. Later on we shall see how this difficulty can be dealt with by using a more sophisticated strategy (ID).

Backtracking

Backtracking (BT) is a slight variation of DF. When a node is expanded, instead of applying all possible actions to generate all successors, only one action is executed and one successor is generated; a reference is left to all further actions, should it be necessary to execute them later. This can be implemented by providing each node with an additional field (furtherActions), which stores the list of actions that have not yet been executed. BT behaves much as DF, but its memory requirements are:

$$\text{memory(DF)} \propto m.$$

Therefore, BT is a better choice than DF when the branching factor is high.

Uniform cost

As we have already remarked, BF is not optimal when the costs of the different action executions are not all identical. To achieve optimality, the nodes in the frontier must be expanded not according to level, but according to total cost. This only requires that the frontier is managed as a *priority queue*, ordered by non-decreasing values of totalCost. The *uniform cost* strategy (UC) does exactly this. To implement UC, it is sufficient that insertList inserts the successor nodes into the frontier so that this is kept ordered by non-decreasing values of totalCost. To resolve possible ambiguities, we assume that every newly generated node *n* is inserted in the frontier *immediately after* any node that already exists in the frontier and has the same totalCost as *n*.

It is not immediate to see that UC is optimal: after all, search strategies always return the *first* solution they find; how can we guarantee that this is always an optimal solution? Indeed, it is possible to prove that, under suitable hypotheses, when UC expands a node *n* representing a state *s*, then the total cost of *n* is *the cost of an optimal path from the root node to any node representing state s*. This guarantees that the first solution found by UC is optimal, provided that *a node is tested for being a goal when it is removed from* frontier *for expansion*. (Note that UC would not be optimal if the node were tested when it is generated!)

The 'suitable hypotheses' mentioned above are the following ones:

- every node of the search tree has a finite number of successors (but this is guaranteed by the fact that the set of possible actions if finite);
- there is a positive lower bound ε to the costs of actions; that is, for every action *a* and state *s*:
  $cost(a,s) \geq \varepsilon$, with $\varepsilon > 0$.

In other words, costs cannot be arbitrarily small: an assumption that is readily verified in all realistic applications.

Finally, note that when all action costs are identical UC produces exactly the same search tree as BF.

Iterative deepening

Let us now concentrate on problems where the cost of all actions is identically equal to 1. What search strategy should we adopt? If the conditions are satisfied for DF to be complete and optimal, this is the best choice, because the memory complexity of DF is linear in the maximum depth of the search tree. BT would be an even better choice, if the branching factor and depth of the search tree are large enough to justify the corresponding overhead. But if the conditions for the completeness and optimality of DF are not satisfied, it seems that we have to adopt BF. This is bad news, because an exponential memory complexity is likely to cripple even a computer with a very large RAM.

There is, however, the possibility of combining BF and DF into a new strategy (called *iterative deepening*, ID) that preserves the best features of both, at an acceptable price, because:

- like BF, it is complete and optimal;
- like DF, has linear memory complexity.

To specify ID, we first produce a *limited depth* variant of `treeSearch` or `graphSearch`: this is obtained by checking that a node is expanded only if it has not yet reached a predefined *depth limit* (remember that with all costs equal to 1, the total cost of a node represents its depth in the search tree). Then specify ID as a loop of limited depth DF searches:

- first generate the root and check it for being a goal;
- if the root is not a goal, perform a limited depth DF search with depth limit equal to 1;
- if no goal is found at level 1, increment the depth limit to 2 and perform another DF search;
- and so on.

Note that at every cycle a complete search tree is produced, re-starting from the root (which is generated only once, at the beginning of the procedure). If d is the length of a shortest (and therefore optimal) solution, then ID will produce search trees for *all* limited depths h, such that $1 \le h \le d$. Let us consider a concrete example. Suppose that $b = 5$, and $d = 10$. Then:

the node at level 0 will be generated once

the 5 nodes at level 1 will be generated 10 times, for a total of 50 nodes

the 25 nodes at level 2 will be generated 9 times, for a total of 225 nodes

...

the 9,765,625 nodes at level 10 will be generated once.

This gives us a grand total of 15,258,776 generated nodes. How does this figure compare with the number of nodes generated by BF? These are 12,207,031, therefore ID generates approximately 1.25 times more nodes than BF. If the search is pushed further, for example down to level 20, ID will still generate 25% more nodes than BF. Indeed it can be proved that once branching factor b is fixed, the ratio of the number $\mathrm{ID}_b(d)$ of nodes generated by ID to the number of nodes $\mathrm{BF}_b(d)$ generated by BF down to any predefined level d is approximately a constant, given by

$$\frac{b}{b-1}$$

With $b = 5$, as we have already seen, $c = 5/4 = 1.25$. The derive this constant it is convenient to assume that at every iteration a new root node is generated (i.e., $d+1$ root nodes are assumed to be generated, instead of just 1), and then subtract d from the total:

$$\begin{aligned}\mathrm{ID}_b(d) &= \sum_{h=0}^{h=d}\sum_{k=0}^{k=h} b^k - d = \sum_{h=0}^{h=d}\frac{b^{h+1}-1}{b-1} - d = \frac{1}{b-1}(b\cdot\sum_{h=0}^{h=d} b^h - \sum_{h=0}^{h=d} 1) - d \\ &= \frac{b}{b-1}\sum_{h=0}^{h=d} b^h - \frac{d+1}{b-1} - d = \frac{b}{b-1}\mathrm{BF}_b(d) - \frac{d+1}{b-1} - d \approx \frac{b}{b-1}\mathrm{BF}_b(d)\end{aligned}$$

To conclude, the time required to find a solution with ID is only slightly more than the time required to find a solution with BF if the branching factor is large enough (e.g., if $b = 11$, then $b/(b–1) = 11/10 = 1.1$). Given that at every iteration ID uses a DF strategy, the memory complexity is still linear in d; for this reason, ID is the preferred search strategy whenever DF is not complete and optimal.

5.5 General comments

State space search (in all its versions) has two fundamental features. The first feature is that it allows to explore a state space in search of a solution *without explicitly representing the whole search space*. As state spaces are often large or even infinite, this is obviously a necessary condition for the actual implementation and use of search strategies. The second feature is less obvious, but it is important to understand it in order to distinguish state space search from planning, which we shall consider in the final part of the course. The main point, here, is *how states and actions are represented*.

Actions are represented procedurally

As we have seen, a state space has a finite set A of action (of cardinality $\#A \geq 2$). In state space search, every action is represented by a procedure, implemented in a suitable programming language, which can be applied to the representation of a state. Suppose for example that the procedure representing action a is applied to the representation of state s. Typically, the search program will first check whether action a can be executed in s; if action a can be executed in state s, the procedure will then produce the representation of a new state, s', which results from the execution of a in s. State s' is then further processed according to the search procedure (i.e., it is inserted in frontier, and so on).

The important point is to understand what the search strategy knows about the action: it only knows *how* to produce state s' from state s. Using the jargon of AI, this is a case of *procedural knowledge*. What gives a search agent some degree of rationality is the ability to 'simulate' the execution of actions in its memory, without actually performing them in the real world: a state space search system simulates the execution of an action by running the corresponding procedure on a representation of a state, without physically executing the action in the environment. Of course, this ability to simulate the execution of an action is required by any kind of search: if the agent had to physically perform the actions in its real environment, then no search would be possible, because in general real environments do not allow an agent to undo its actions!

A first consequence of these remarks is that the best way to represent a state is the one which *makes it most efficient to simulate the execution of an action*. Iconic representations are very good at this, and this is why state space systems typically use iconic representations.

A further consequence, however, is that an agent cannot *reason on the structure of its own actions*. For example, an agent could not compare two actions, a_1 and a_2, and conclude that a_1 should never be performed before a_2, because a_2 cannot possibly be executed in a state produced by a_1. The reason is that the only representations the agent has of its actions are the procedures that simulate the performance of such actions; these procedures can be executed, but they cannot be 'inspected' statically, for example to compare two actions. In other words, an agent cannot use a state space representation to infer by reasoning some general property of its own actions. To allow an agent to do so we need a different approach, which is typical of *planning*.