IAI : Search

1. The Importance of Search in AI
2. Properties of Search Algorithms
3. State Space Representations
4. Search Trees
5. Five Types of Uninformed Search Algorithm
6. Avoiding Repeated States
7. Comparing the Uninformed Search Algorithms
8. Informed Search

The Importance of Search in AI

It has already become clear that many of the tasks underlying AI can be phrased in terms of a ***search*** for the solution to the problem at hand.

Many ***goal based agents*** are essentially ***problem solving agents*** which must decide what to do by searching for a sequence of actions that lead to their solutions.

For ***production systems*****,** we have seen the need to search for a sequence of rule applications that lead to the required fact or action.

For ***neural network systems*****,** we need to search for the set of connection weights that will result in the required input to output mapping.

How we go about each of these searches is determined by a ***search strategy***. In this lecture we shall begin by looking at a number of ***uninformed (blind) search*** strategies, i.e. search strategies that do not use any information about the distance to the goal. Then we will have an overview of some important ***informed search*** strategies.

Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain. There are four important factors to consider:

1. ***Completeness*** – Is a solution guaranteed to be found if at least one solution exists?

2. ***Optimality*** – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?

3. ***Time Complexity*** – The upper bound on the time required to find a solution, as a function of the complexity of the problem.

4. ***Space Complexity*** – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

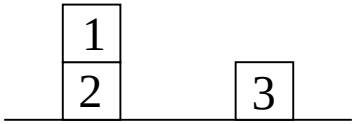
We shall start with some general theory, and then look in more detail at some specific search algorithms.

State Space Representations

The ***state space*** is simply the space of all possible states, or configurations, that our system may be in. Generally, of course, we prefer to work with some convenient ***representation*** of that search space.

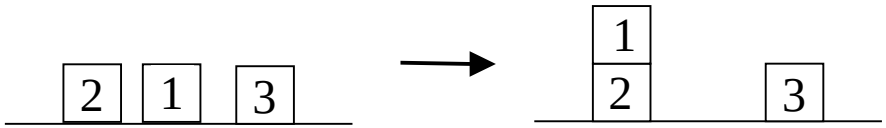
There are two components to the representation of state spaces:

1. Static States

e.g. 

((1 2) (3))

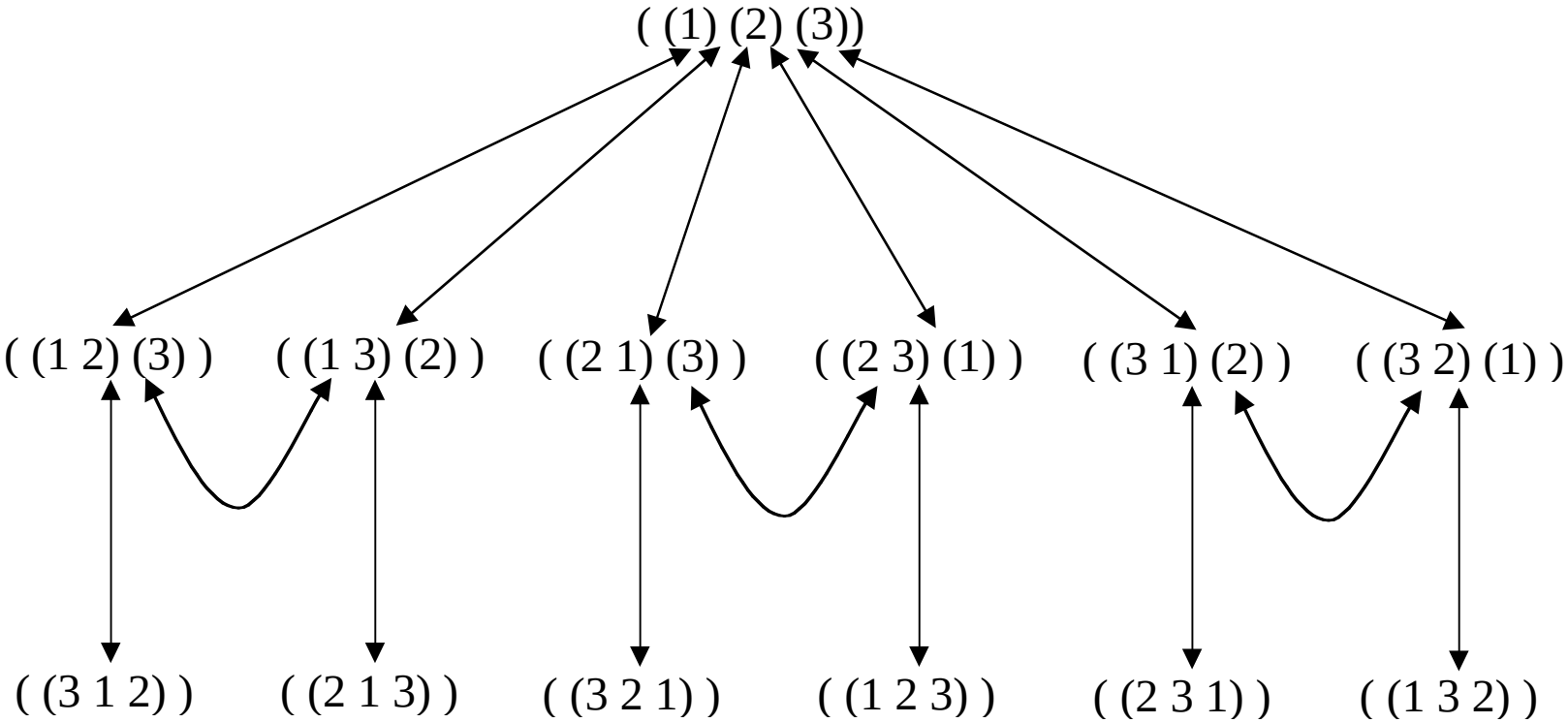
2. Transitions between States

e.g. 

move(1, 2)

State Space Graphs

If the number of possible states of the system is small enough, we can represent all of them, along with the transitions between them, in a ***state space graph***, e.g.



Exercise: Add the appropriate transition labels to each link.

Routes Through State Space

Our general aim is to search for a route, or sequence of transitions, through the state space graph from our ***initial state*** to a ***goal state***.

Sometimes there will be more than one possible goal state. We define a ***goal test*** to determine if a goal state has been achieved.

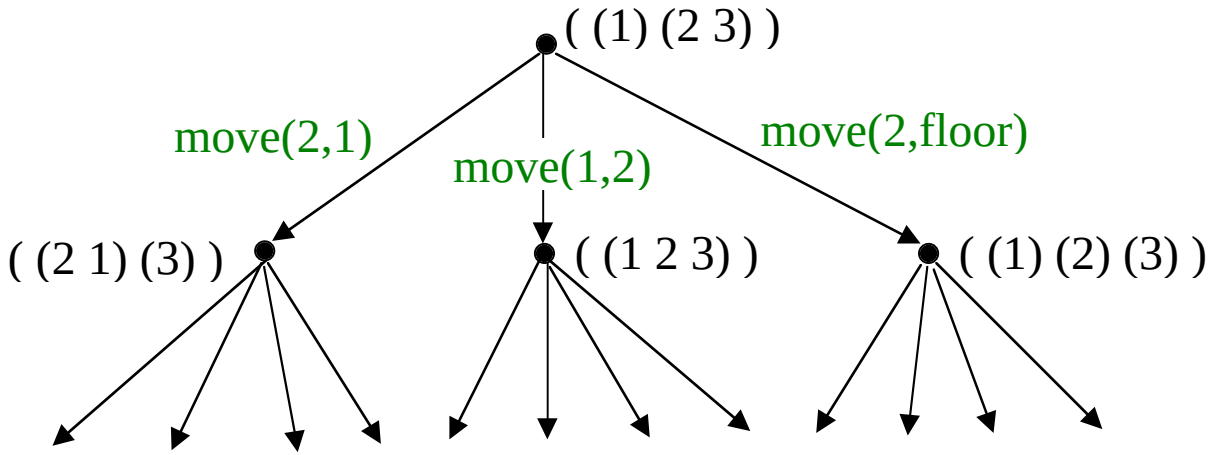
The solution can be represented as a sequence of link labels (or transitions) on the state space graph. Note that the labels depend on the direction moved along the link.

Sometimes there may be more than one path to a goal state, and we may want to find the optimal (best possible) path. We can define ***link costs*** and ***path costs*** for measuring the cost of going along a particular path, e.g. the path cost may just equal the number of links, or could be the sum of individual link costs.

For most realistic problems, the state space graph will be too large for us to hold all of it explicitly in memory at any one time.

Search Trees

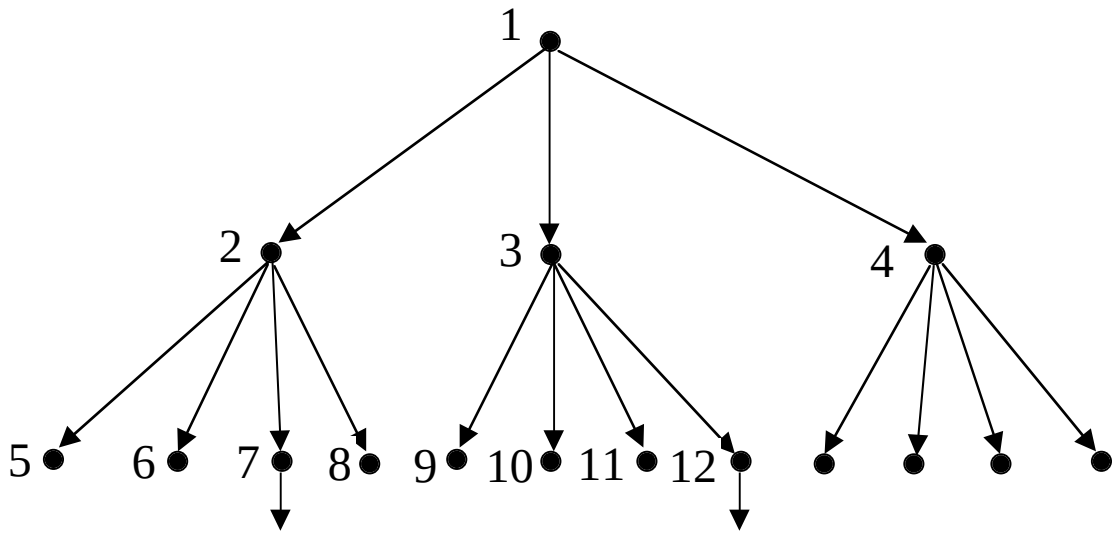
It is helpful to think of the search process as building up a ***search tree*** of routes through the state space graph. The root of the search tree is the ***search node*** corresponding to the initial state. The leaf nodes correspond either to states that have not yet been expanded, or to states that generated no further nodes when expanded.



At each step, the search algorithm chooses a new unexpanded leaf node to expand. The different search strategies essentially correspond to the different algorithms one can use to select which is the next mode to be expanded at each stage.

Breadth First Search (BFS)

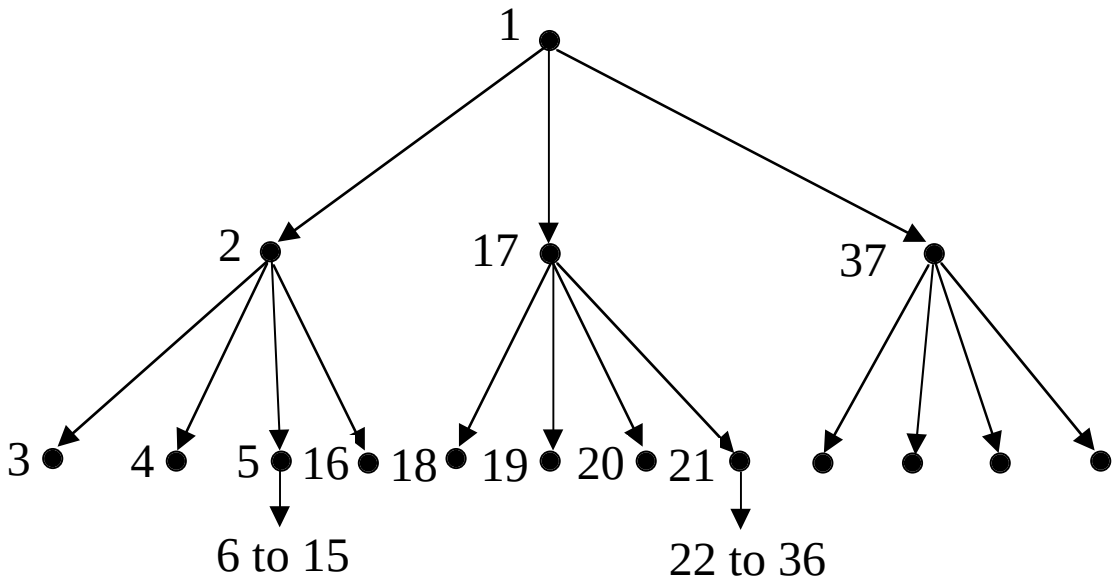
BFS expands the leaf node with the ***lowest*** path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue ("first in, first out").



This is guaranteed to find an ***optimal path*** to a goal state. It is memory intensive if the state space is large. If the typical branching factor is *b*, and the depth of the shallowest goal state is *d* – the space complexity is $O(b^d)$, and the time complexity is $O(b^d)$.

Depth First Search (DFS)

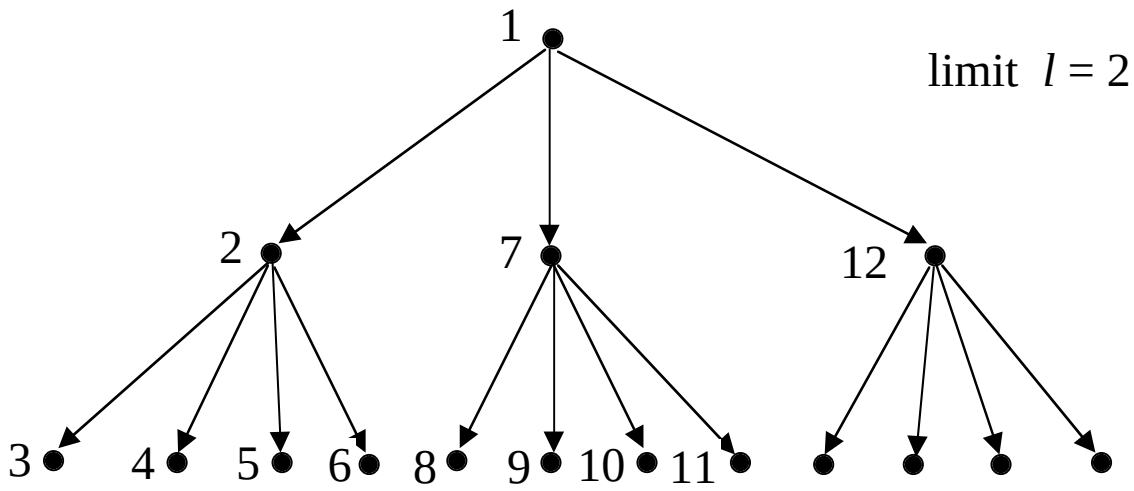
DFS expands the leaf node with the ***highest*** path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack ("last in, first out").



This is ***not guaranteed*** to find any path to a goal state. It is memory efficient even if the state space is large. If the typical branching factor is b, and the maximum depth of the tree is m (possibly ∞) – the space complexity is O(bm), and the time complexity is O(b^m).

Depth Limited Search (DLS)

DLS is a variation of DFS. If we put a limit l on how deep a depth first search can go, we can guarantee that the search will terminate (either in success or failure).



If there is at least one goal state at a depth less than l, this algorithm is guaranteed to find ***a goal state***, but it is not guaranteed to find an optimal path. The space complexity is $O(bl)$, and the time complexity is $O(b^l)$. For most problems we will not know what is a good limit l until we have solved the problem!

Depth First Iterative Deepening Search (DFIDS)

DFIDS is a variation of DLS. If the lowest depth of a goal state is not known, we can always find the best limit l for DLS by trying all possible depths $l = 0, 1, 2, 3, \ldots$ in turn, and stopping once we have achieved a goal state.

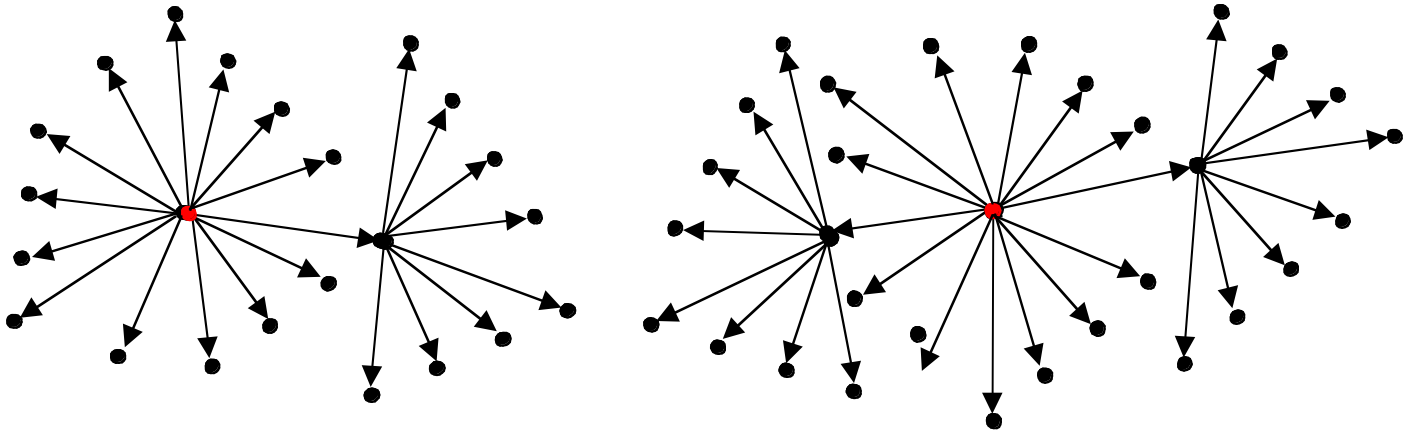
This appears wasteful because all the DLS for l less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS and BFS.

Because all the nodes are expanded at each level, the algorithm is complete and optimal like BFS, but has the modest memory requirements of DFS. **Exercise:** if we had plenty of memory, could/should we avoid expanding the top level states many times?

The space complexity is $O(bd)$ as in DLS with $l = d$, which is better than BFS. The time complexity is $O(b^d)$ as in BFS, which is better than DFS.

Bi-Directional Search (BDS)

The idea behind bi-directional search is to search simultaneously both forward from the initial state and backwards from the goal state, and stop when the two BFS searches meet in the middle.



This is not always going to be possible, but is likely to be feasible if the state transitions are reversible. The algorithm is complete and optimal, and since the two search depths are ~$d/2$, it has space complexity O($b^{d/2}$), and time complexity O($b^{d/2}$). However, if there is more than one possible goal state, this must be factored into the complexity.

Repeated States

In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the search tree.

For some problems this possibility can never arise, because each state can only be reached in one way.

For many problems, however, repeated states are unavoidable. This will include all problems where the transitions are reversible, e.g.

$$((1\ 2\ 3)) \rightarrow ((1) (2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow ((1) (2\ 3)) \rightarrow ((1\ 2\ 3)) \rightarrow \ldots$$

The search trees for these problems are infinite, but if we can prune out the repeated states, we can cut the search tree down to a finite size, We effectively only generate a portion of the search tree that matches the state space graph.

Avoiding Repeated States

There are three principal approaches for dealing with repeated states:

1. **Never return to the state you have just come from**

   The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.

2. **Never create search paths with cycles in them**

   The node expansion function must be prevented from generating any node successor that is the same state as any of the node's ancestors.

3. **Never generate states that have already been generated before**

   This requires that every state ever generated is remembered, potentially resulting in space complexity of $O(b^d)$.

Clearly these are in increasing order of effectiveness and computational overhead.

Comparing the Uninformed Search Algorithms

We can now summarize the properties of our five uninformed search strategies:

Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
DLS	If $l \geq d$	No	$O(b^l)$	$O(bl)$
DFIDS	Yes	Yes	$O(b^d)$	$O(bd)$
BDS	Yes	Yes	$O(b^{d/2})$	$O(b^{d/2})$

Simple BFS and BDS are complete and optimal but expensive with respect to space and time. DFS requires much less memory if the maximum tree depth is limited, but has no guarantee of finding any solution, let alone an optimal one. DLS offers an improvement over DFS if we have some idea how deep the goal is. The ***best overall is DFID*** which is complete, optimal and has low memory requirements, but still exponential time.

Informed Search

Informed search uses some kind of ***evaluation function*** to tell us how far each expanded state is from a goal state, and/or some kind of ***heuristic function*** to help us decide which state is likely to be the best one to expand next.

The hard part is to come up with good evaluation and/or heuristic functions. Often there is a natural evaluation function, such as distance in miles or number objects in the wrong position. Sometimes we can learn heuristic functions by analysing what has worked well in similar previous searches.

The simplest idea, known as ***greedy best first search***, is to expand the node that is already closest to the goal, as that is most likely to lead quickly to a solution. This is like DFS in that it attempts to follow a single route to the goal, only attempting to try a different route when it reaches a dead end. As with DFS, it is not complete, not optimal, and has time and complexity of $O(b^m)$. However, with good heuristics, the time complexity can be reduced substantially.

A* Search

Suppose that, for each node *n* in a search tree, an evaluation function *f(n)* is defined as the sum of the cost *g(n)* to reach that node from the start state, plus an estimated cost *h(n)* to get from that state to the goal state. That *f(n)* is then the estimated cost of the cheapest solution through *n*.

A* search**,*** which is the most popular form of best-first search, repeatedly picks the node with the lowest *f(n)* to expand next. It turns out that if the heuristic function *h(n)* satisfies certain conditions, then this strategy is both complete and optimal.

In particular, if *h(n)* is an ***admissible heuristic***, i.e. is always optimistic and never overestimates the cost to reach the goal, then A* is optimal.

The classic example is finding the route by road between two cities given the straight line distances from each road intersection to the goal city. In this case, the nodes are the intersections, and we can simply use the straight line distances as *h(n)*.

Hill Climbing / Gradient Descent

The basic idea of ***hill climbing*** is simple: at each current state we select a transition, evaluate the resulting state, and if the resulting state is an improvement we move there, otherwise we try a new transition from where we are. We repeat this until we reach a goal state, or have no more transitions to try. The transitions explored can be selected at random, or according to some problem specific heuristics.

In some cases, it is possible to define evaluation functions such that we can compute the gradients with respect to the possible transitions, and thus compute which transition direction to take to produce the best improvement in the evaluation function. Following the evaluation gradients in this way is known as ***gradient descent***.

In ***neural networks***, for example, we can define the total error of the output activations as a function of the connection weights, and compute the gradients of how the error changes as we change the weights. By changing the weights in small steps against those gradients, we systematically minimise the network's output errors.

Overview and Reading

1. We began by outlining the general properties of search algorithms, and the basic ideas of state space graph representations and search trees.
2. Then we looked at five particular uninformed (blind) search algorithms: breadth first search, depth first search, depth limited search, depth first iterative deepening search, and bi-directional search.
3. We then considered which of these algorithms was best in general.
4. We ended by looking at some important informed search algorithms.

Reading

1. Russell & Norvig: Chapters 3, 4
2. Nilsson: Chapter 8, 9
3. Callan: Chapter 3
4. Winston: Chapters 4, 5, 6
5. Rich & Knight: Chapters 2, 3