# AITA : Uninformed 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 Search Algorithms

# 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.

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

In terms of ***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 look at a number of ***uninformed (blind) search*** strategies, i.e. search strategies that do not use any information about the distance to the goal. Next semester you will learn about more advanced 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 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.

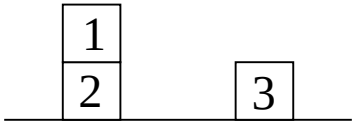
We shall start with some general theory, and then look in more detail at some specific search algorithms.

# 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.

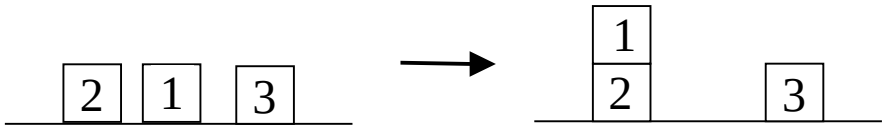
There are two components to the representation of state spaces:

## 1. Static States

e.g. 


( (1 2) (3) )

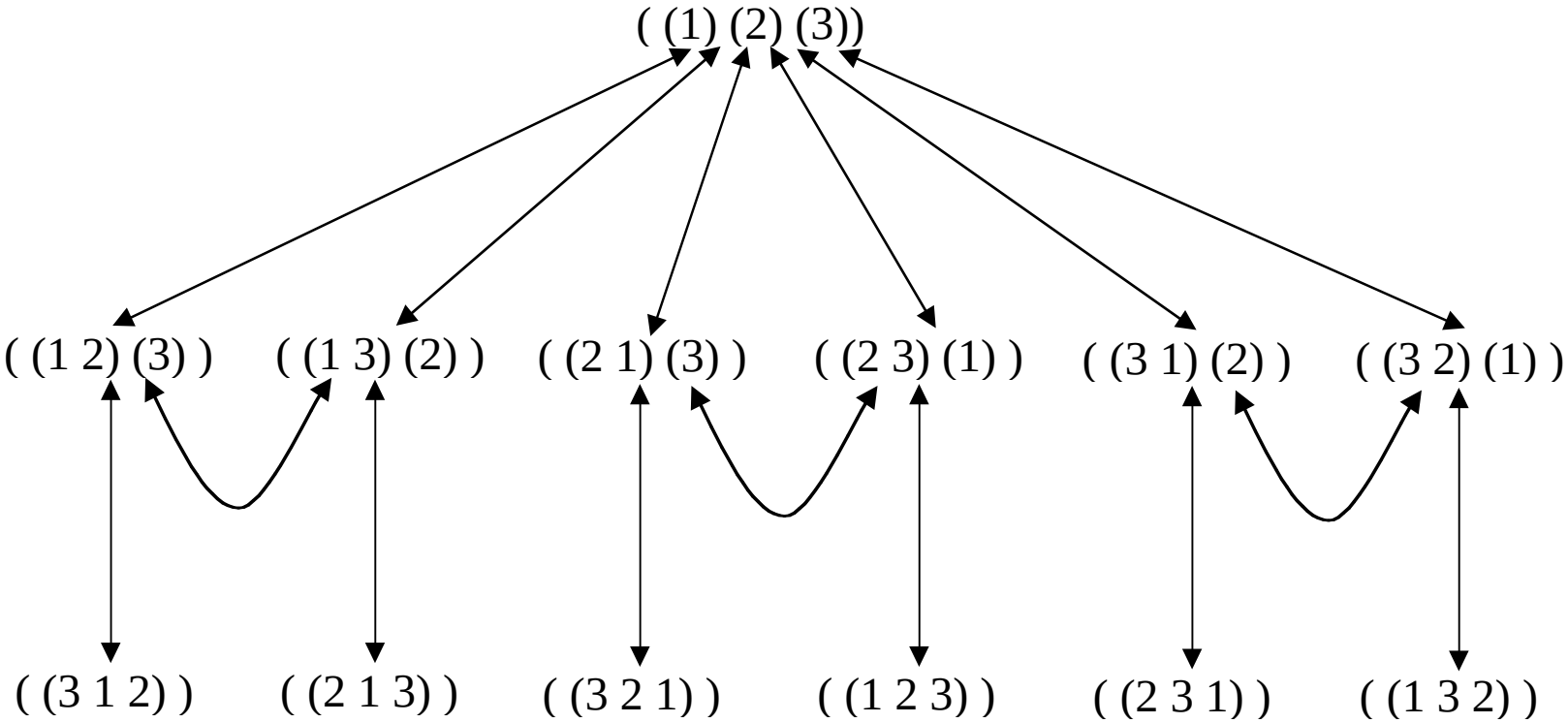
## 2. Transitions between States

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***.

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

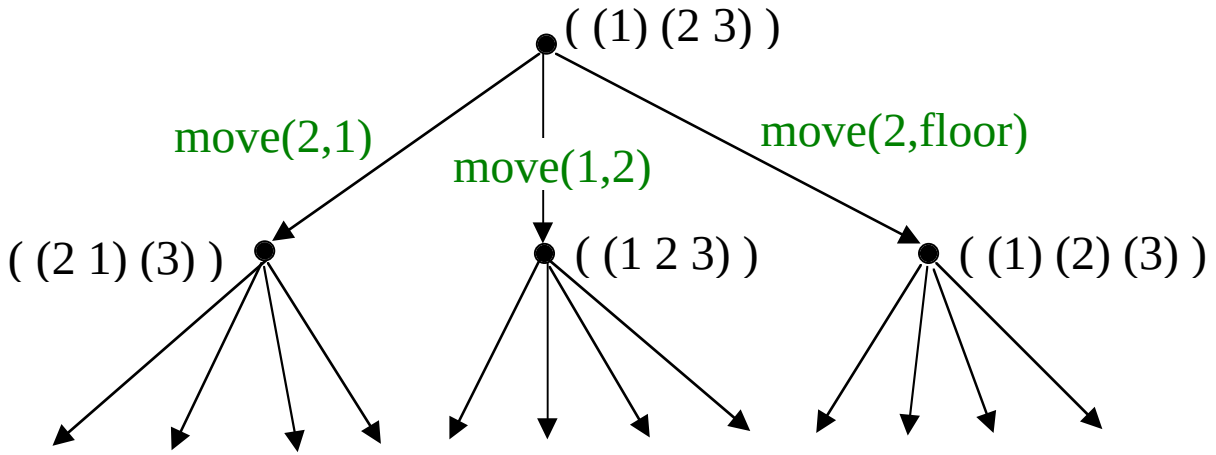
The solution can be represented as a sequence of link labels (or transitions) on the state space graph. Note that the labels depend on the direction moved along the link.

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

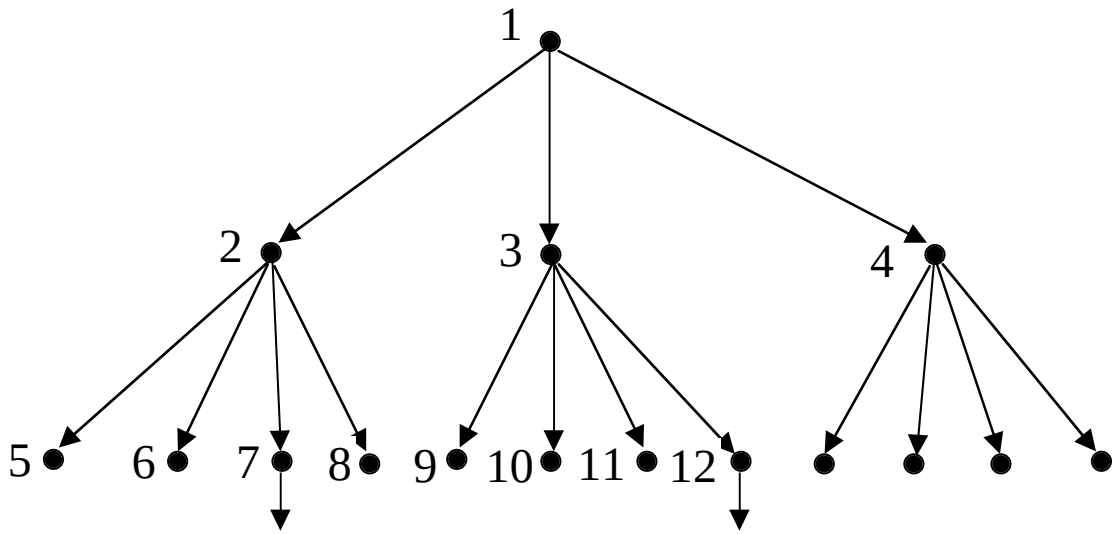
It is helpful to think of the search process as building up a ***search tree*** of routes through the state space graph. The root of the search tree is the ***search node*** corresponding to the initial state. The leaf nodes correspond either to states that have not yet been expanded, or to states that 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)

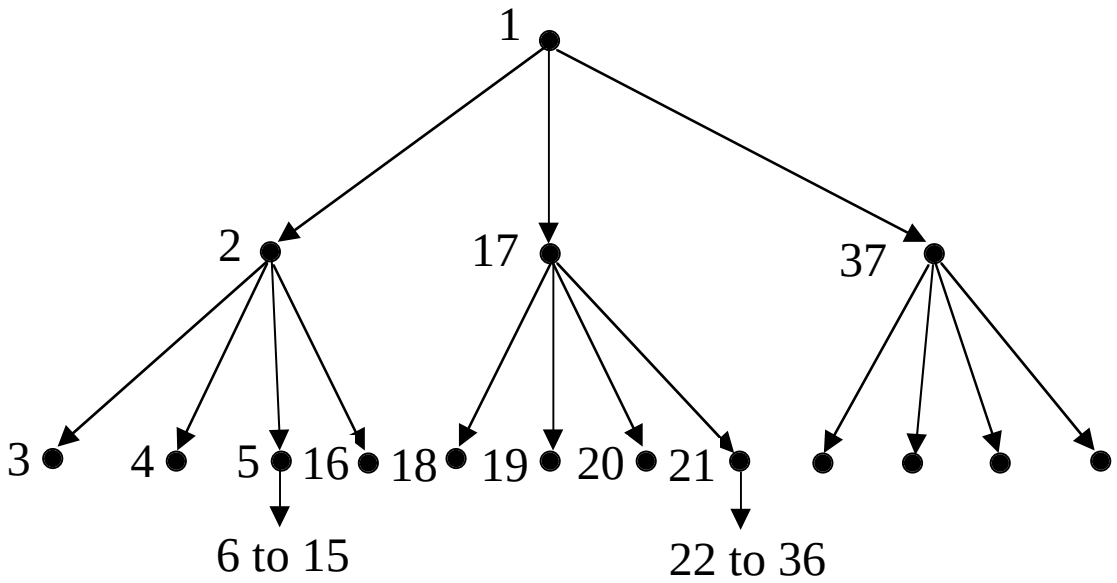
BFS expands the leaf node with the ***lowest*** path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue ("first in, first out").




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)

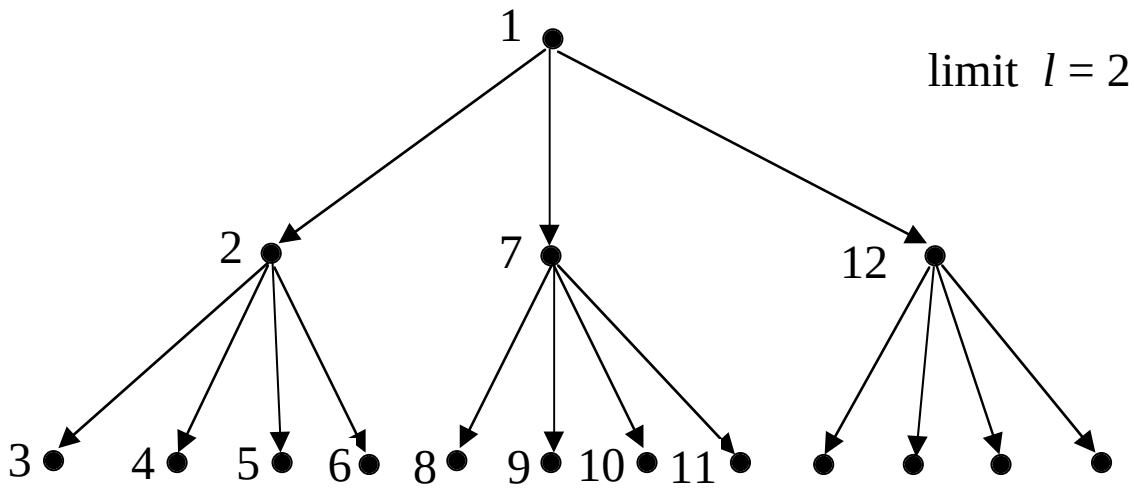
DFS expands the leaf node with the ***highest*** path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack ("last in, first out").




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 $\infty$) – 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.

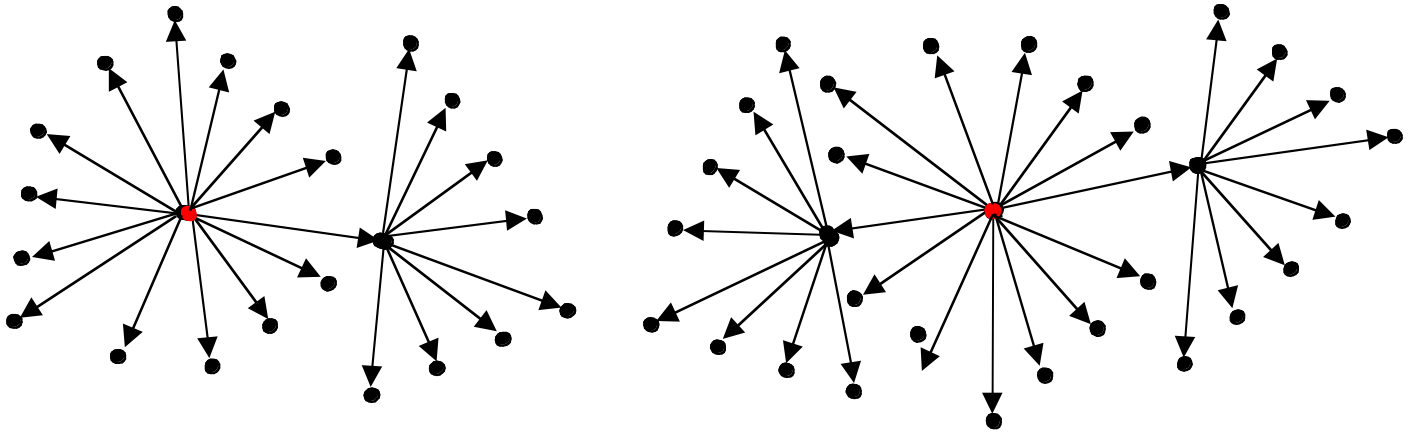
This appears wasteful because all the DLS for $l$ less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS and BFS.

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 simultaneously search 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 necessarily possible, but is likely to be feasible when the state transitions are reversible. There will clearly be problems if there is more than one goal state. However, 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($b^{d/2}$).

# Repeated States

Up until now we have ignored an important complication that often arises in search processes – the possibility of wasting time by expanding states that have already been expanded before somewhere else on the search tree.

For some problems this possibility never arises, 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 some of the repeated states, we can cut the search tree down to a finite size, generating only the portion of the tree that spans the state space graph.

# Avoiding Repeated States

There are three principal ways to deal with repeated states:

1. **Do not return to the state you just came from**

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

2. **Do not create 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. **Do not generate any state that was ever 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 Search Algorithms

We end with a comparison of the five uninformed search strategies we have looked at:

| 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 there is 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.

# Overview and Reading

1. We began by outlining the general properties of search algorithms, and the basic ideas of state space representation 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 the problem of repeated states and how to avoid them.
4. We ended with a comparison of our five uninformed search algorithms.

## Reading

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