University of Essex

School of Computer Science & Electronic Engineering

CE213 Artificial Intelligence – Lecture 3

Blind Search Strategies

We now have a way of representing problems:

Formulate them as state spaces

To solve them we need:

Some way of searching for a sequence of transitions that will take us from the initial state to a goal state

Strategies for Searching State Spaces

The objective of a **search strategy** is to systematically explore the state space by constructing a **search tree**.

Key question in search tree construction:

Which state (or node) to be expanded first and how?

There are many ways in which this can be done, such as **blind search** and **heuristic search**.

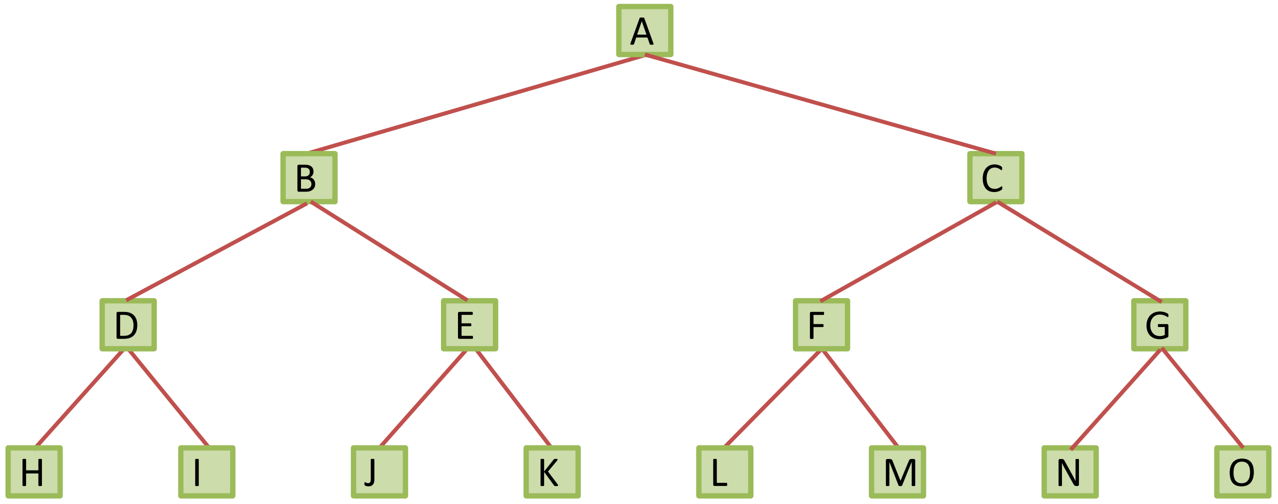
Note: The search space may be very large, possibly infinite.
Exhaustive search may be unrealistic.

A Binary Search Tree

We first consider the simplest possible search state space:

One in which there are only two possible transitions from each state.

This will lead to a binary search tree:



A search strategy determines the order in which the tree is constructed.

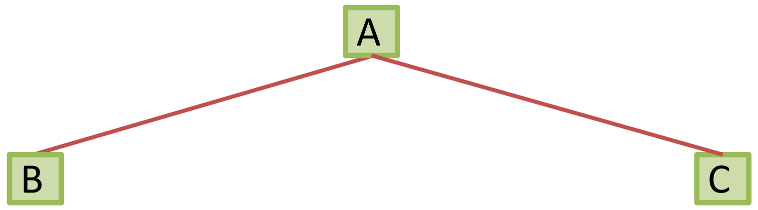
Node Generation and Node Expansion

Basic elements in a search tree:

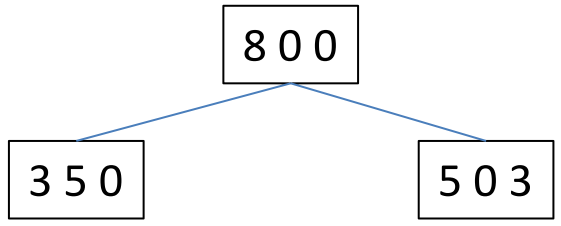
A ***node*** represents a state and its related information such as cost, heuristic value and parent node; A ***branch*** represents an action or move.

A node is ***generated*** when it is added to the tree.

A node is ***expanded*** when its successors (children) are added to the tree.



e.g. for the 3 jugs problem



At this point:

Node A has been expanded.

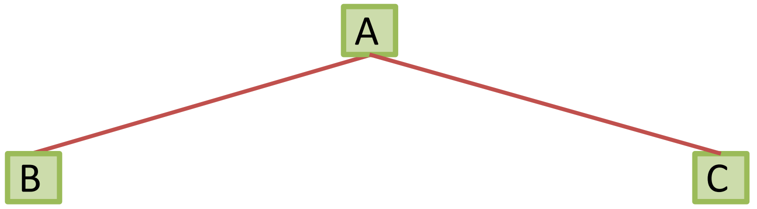
Nodes B and C have been generated but not expanded.

Breadth First Search

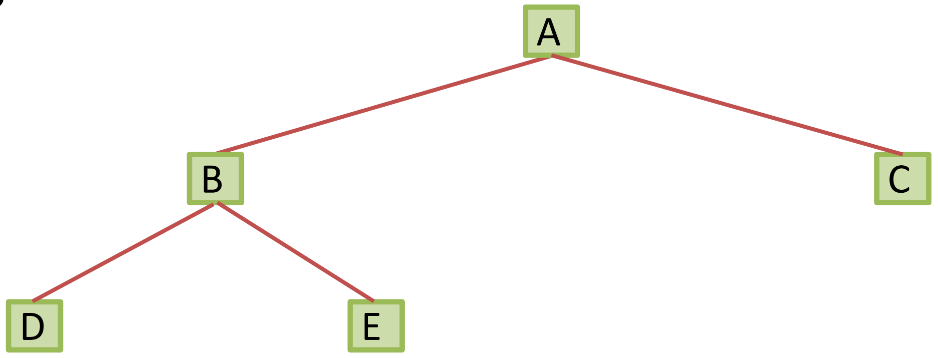
In ***breadth first search,*** all the nodes at depth n from the root must be expanded before any node at depth n+1.

In other words, the ***least recently generated*** unexpanded node is chosen for expansion.

So we begin by expanding Node A:

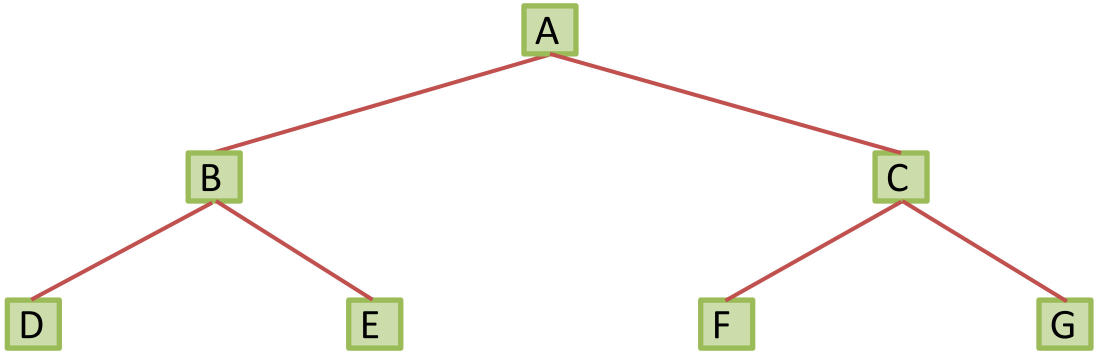


Then Node B

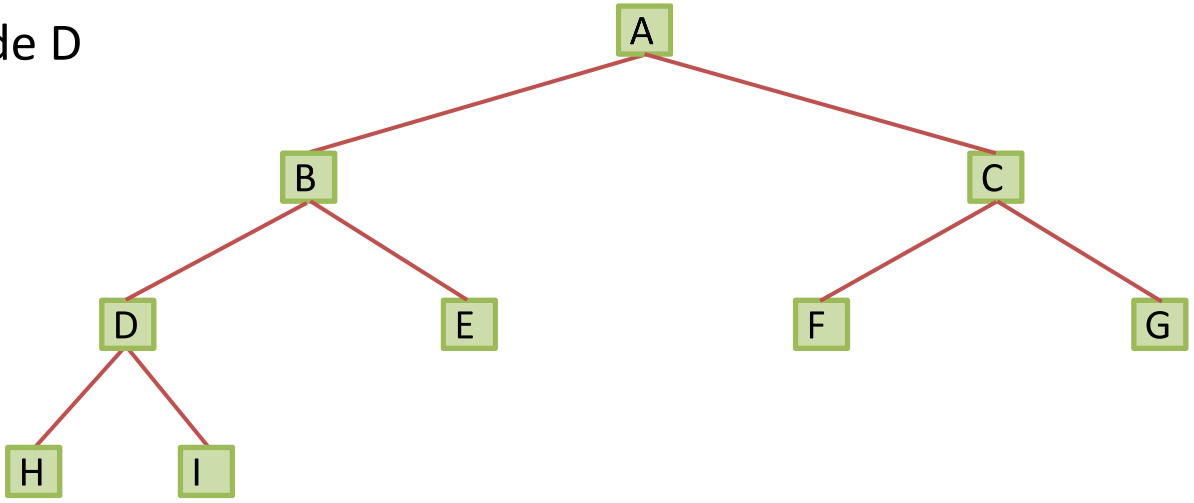


Breadth First Search (2)

Then Node C

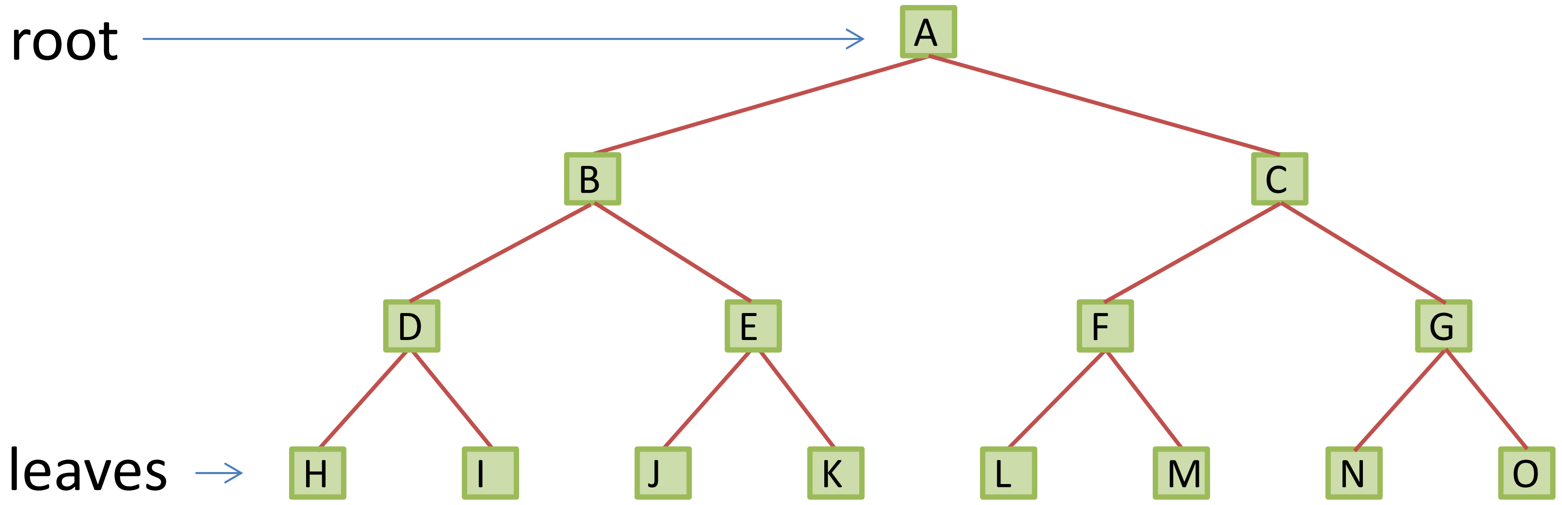


Then Node D



Breadth First Search (3)

And then Node E, then Node F, then Node G to produce:



A search strategy is therefore simply a rule for deciding which node to expand next.

Note that the generated but unexpanded nodes are the leaves of the tree (H, I, J, K, L, M, N, O are leaves of the above search tree).

Pseudo Code for Breadth First Search

```
Create an empty list UnexpNodes 1);
{Holds list of unexpanded nodes}
Add initial node to UnexpNodes;
WHILE (UnexpNodes not empty)
    N = First item of UnexpNodes 2)
    Remove N from UnexpNodes
    IF N is goal
    THEN Produce Solution 3) and Return(Success);
    Expand N to produce list of successors of N 4);
    Add successors to tail of UnexpNodes;
Return(Failure)
```

Would it be better to check whether any successor is goal here?

1) A node may contain state, cost, parent, etc.

2) Which node to be expanded next? – depending on search strategy adopted

3) How to produce a solution? – backtracking the parents from the goal node found

4) How to expand a node? – depending on the problem, e.g., possible operators

Pseudo Code for Breadth First Search (2)

It would be possible to check for the goal state immediately after a new node is generated by expanding its parent.

This would save (a little) time.

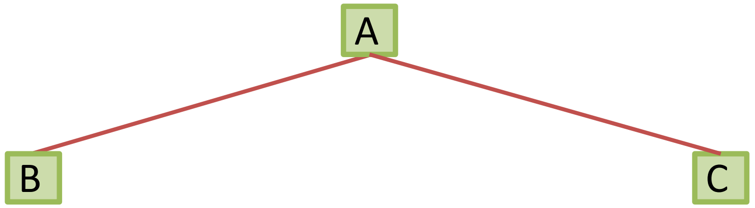
However, waiting until the node is chosen for expansion is important in some of the searches we will consider later.

e.g., uniform cost search, to ensure the solution found is the optimal one.

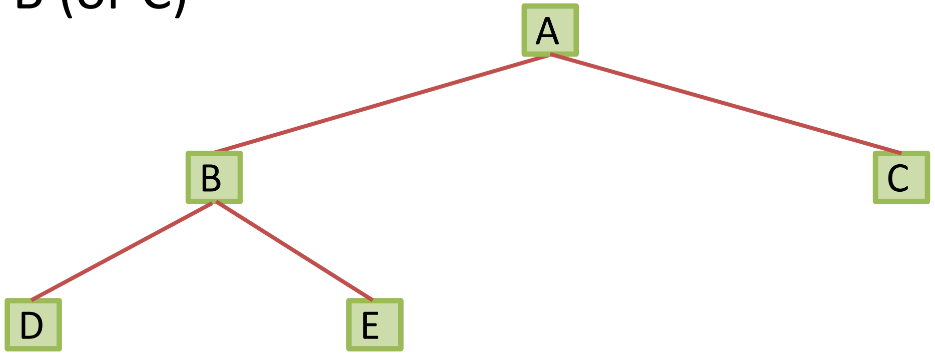
Depth First Search

In ***depth first search,*** the ***most recently generated*** unexpanded node is chosen for expansion.

Again we begin by expanding Node A:

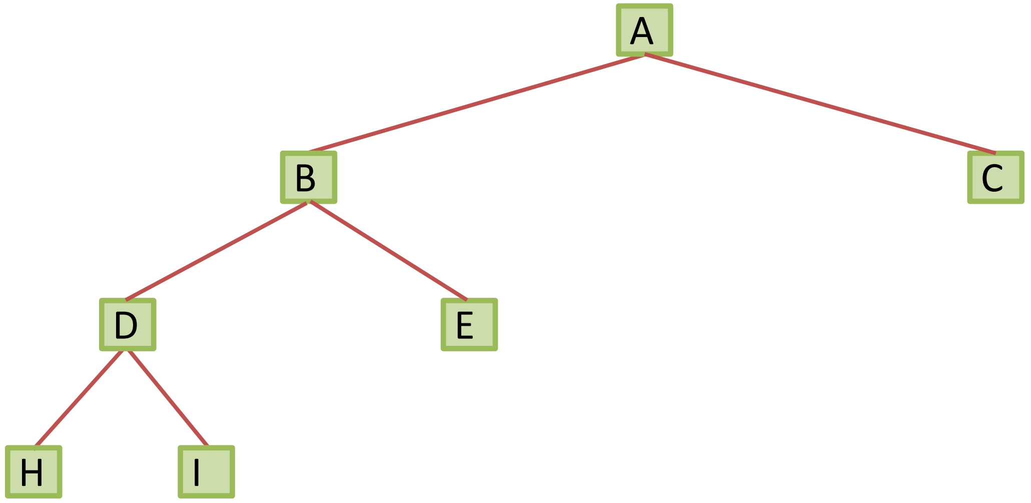


And then Node B (or C)



Depth First Search (2)

Next Node D (or E) (not C) is chosen for expansion:

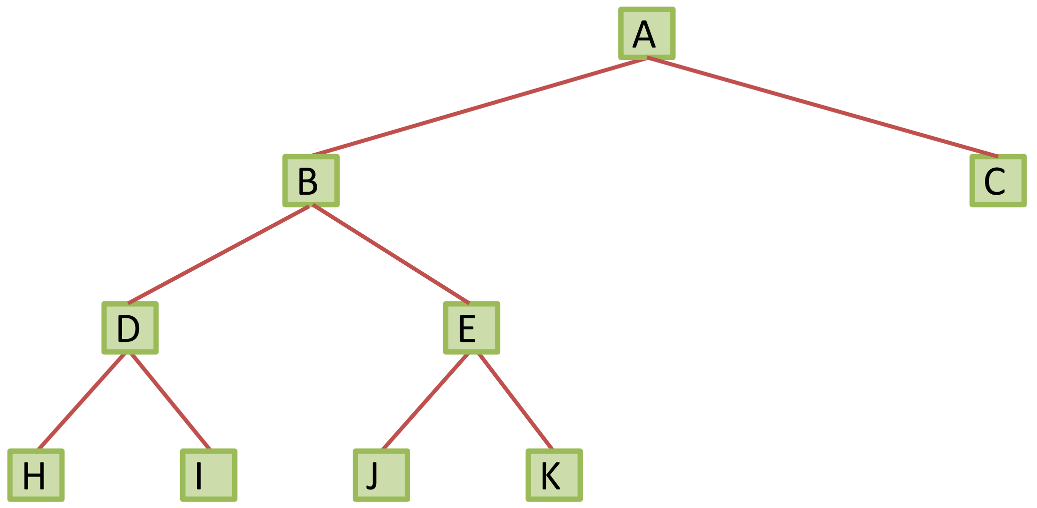


Assuming Nodes H and I have no successors, we cannot expand these. So the search process **backtracks** (very important concept in this search).

Depth First Search (3)

The most recently generated unexpanded node is Node E.

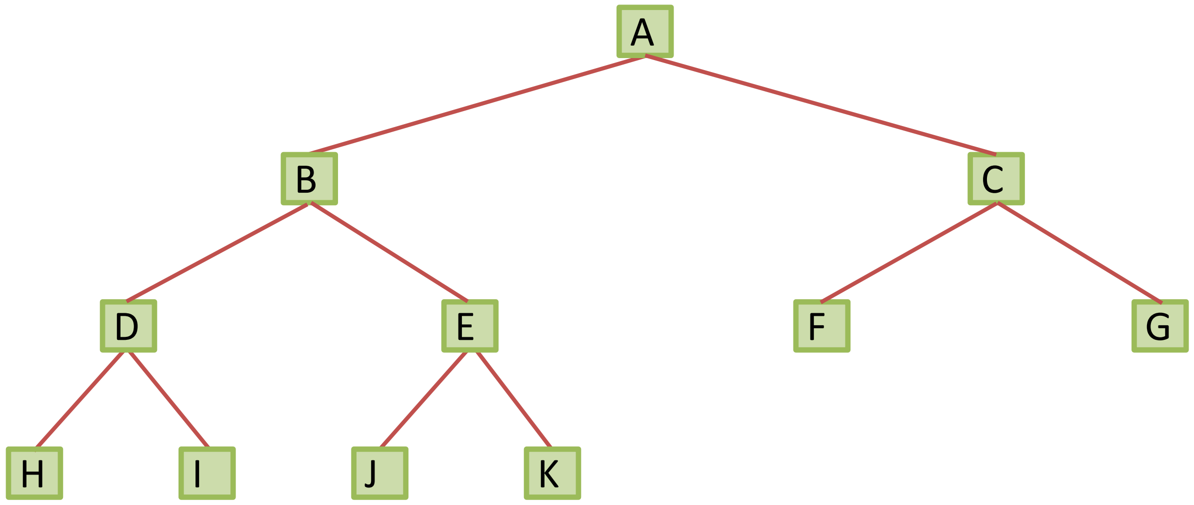
So this is chosen for expansion:



Note that Nodes H and I were removed from UnexpNodes before backtracking (see pseudo code later).

Depth First Search (4)

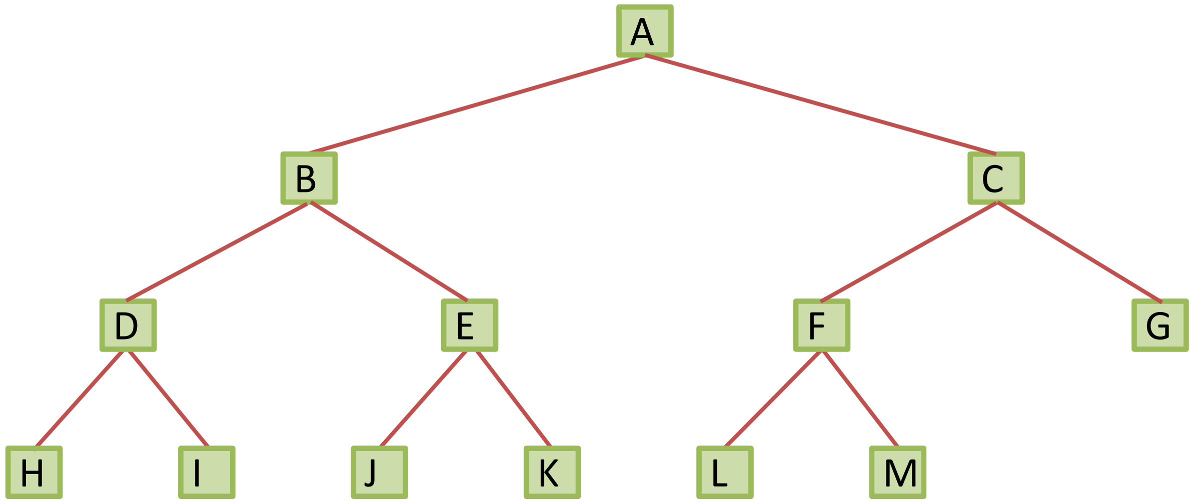
Assuming Nodes J and K have no successors. Once again we have to backtrack: Node C is now the only unexpanded node so this must be chosen next.



Note that Nodes J and K were removed from UnexpNodes before backtracking (see pseudo code later).

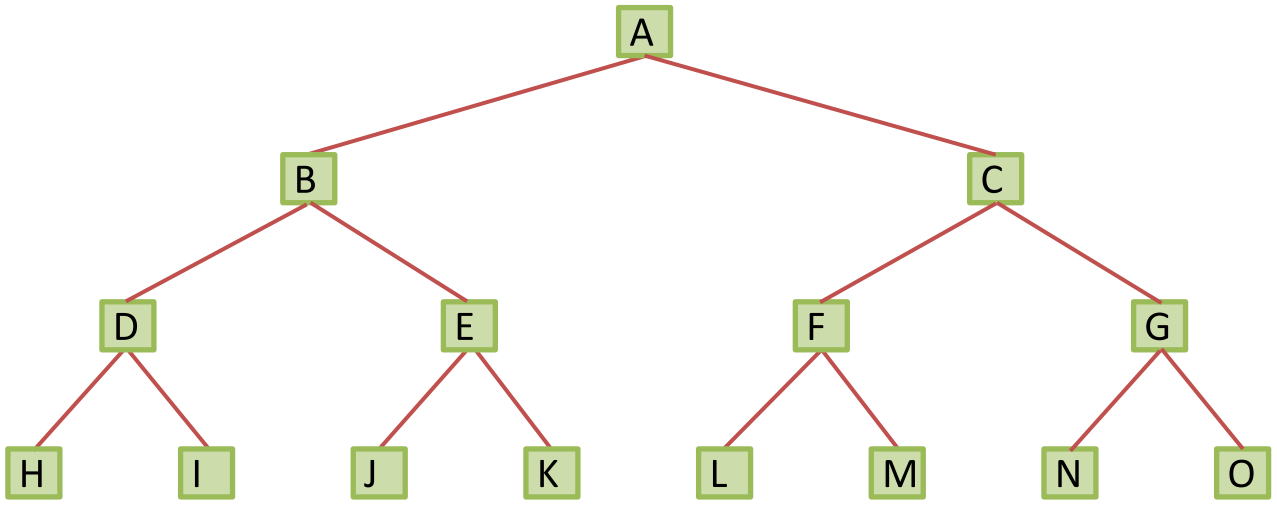
Depth First Search (5)

Next Node F is chosen:



Depth First Search (6)

Finally we backtrack again and expand Node G:



Note that Nodes L and M were removed from UnexpNodes before backtracking (see pseudo code later). Nodes N and O have no successor.

Pseudo Code for Depth First Search

```
Create an empty list UnexpNodes;
{Holds list of unexpanded nodes}
Add initial node to UnexpNodes;
WHILE (UnexpNodes not empty)
   N = Last item of UnexpNodes //only change from BFS
   Remove N from UnexpNodes
   IF N is goal
   THEN Produce Solution and Return(Success);
   Expand N to produce list of successors of N;
   Add successors to tail of UnexpNodes;
Return(Failure)
```

Blind Searching

Depth first search and breadth first search are called ***blind search*** strategies

They use no additional information in selecting which node to expand next

Blind search is also called ***uninformed search***.

Comparing/Evaluating Search Strategies

We now have two search strategies – Which one is better?

Four Criteria:

Completeness

Is the strategy certain to find a solution if there is one?

Optimality

If there is more than one solution, does the strategy find the best one?

(Clearly this may depend on what is meant by 'best'.)

Time Complexity

How long does the strategy take to find a solution?

Space Complexity

How much memory does the strategy require?

Completeness

Breadth First

Complete.

Suppose a goal state exists at depth *d* from the root.

The program will eventually expand all nodes at depth *d*.

Depth First

Not complete, if the state space is infinite.

If the search space is infinite the program may never explore a branch containing a goal state.

However, if the search space is finite and the program checks for duplicate states on the current path then this method is complete.

Optimality

Let a best solution be the one with a minimal number of moves.

Breadth First

Optimal.

First solution encountered clearly has minimum number of moves.

Depth First

Not optimal.

It depends entirely on which branches the program happens to explore first.

Time Complexity

The time complexity and space complexity depend upon the size of the tree to be searched.

We define the size of a search tree as follows:

The branching factor for every node is *b*.

The goal state lies at depth *d* from the root.

The maximum depth is *m*.

For complexity, we usually consider the worst case scenario.

Time complexity is about the time required for expanding nodes, depending on *the number of nodes expanded*.

Time Complexity (2)

Breadth First

The number of nodes expanded will be at most (worst case scenario):

$$1 + b^1 + b^2 + \ \dots + b^d$$

Hence the time complexity is $O(b^d)$.

Thus the time complexity increases *exponentially* as the branching factor is increased.

(If you are not familiar with the Big O notation, please visit https://en.wikipedia.org/wiki/Big_O_notation)

Depth First

In the worst case scenario the solution could be found only when the last node is reached, so all the nodes may have to be expanded.

Hence the time complexity is $O(b^m)$. $m \geq d$

Note, however, that if there are a large number of possible solutions, the actual time may be much less, and depth first search may be faster then breadth first search.

Space Complexity

Space complexity is about the memory required for storing nodes, depending on *the number of nodes unexpended*.

Breadth First

It is necessary to keep all of the nodes at depth n in memory just prior to expanding the nodes at depth $n + 1$.

So the space complexity is also $O(b^d)$. (the same as its time complexity)

Depth First

When the program backtracks it discards the nodes it has already visited.

So it need only store the unexpanded nodes on the path from the root to the current node and their immediate successors.

At each depth, at most b unexpected nodes need to be stored.

Hence the space complexity is only $O(bm)$. This is the only advantage of depth first search, compared to breath first search.

The practical effect of exponential complexity

Assume: Branching factor b = 10,
1 million nodes expanded per second,
Each node needs 1000 bytes of memory.

Depth	Nodes	Time	Memory
2	110	0.11 millisecs	107 Kbytes
4	11,110	11 millisecs	10.6 Mbytes
6	10^6	1.1 seconds	1 Gbytes
8	10^8	2 minutes	100 Gbytes
10	10^{10}	3 hours	10 Terabytes
12	10^{12}	13 days	1 Petabytes
14	10^{14}	3.5 years	100 Petabytes
16	10^{16}	350 years	10 Exabytes

More Sophisticated Blind Search Strategies

- Depth Limited Search
- Iterative Deepening Search
- Uniform Cost Search
  (also know as Dijkstra’s algorithm)

Basic ideas: Limiting the depth in depth first search or introducing a specific cost rather than the number of moves

➔ ***Better than breadth first or depth first search?***

Depth Limited Search

If depth first search makes a wrong choice near the start, it may waste a great deal of time exploring ever deeper in a portion of the tree that contains no solutions.

In the case of an infinite search space it will never terminate.

Depth Limited Search addresses this problem.

A maximum limit, *m*, is placed on the depth that can be explored before backtracking occurs.

So, in the worst case, the program searches the entire tree to depth *m*.

Time complexity $O(b^m)$

Space complexity $O(bm)$.

Provided there is a solution with path length less than *m,* it will be found.

Iterative Deepening Search

If you cannot work out in advance what the maximum depth should be. You cannot safely use depth-limited search.

Iterative Deepening Search provides a simple solution to this problem:

```
m = 1;
REPEAT
    Do depth limited search to depth m;
    m = m + 1;
UNTIL Solution Found;
```

Clearly, like breadth first, this strategy will eventually find an optimal solution.

So iterative deepening search is

complete

optimal

low space complexity: $O(bm)$,

high time complexity (repetition): $O(b^m)$

An attractive search strategy

Uniform Cost Search

So far we have assumed that an optimal solution is one that has the shortest path length, i.e., has the minimum number of moves/operations/actions.

What happens if some operators have higher costs than others?

Uniform Cost Search takes operator costs into account. (It is not breath-first or depth-first search!)

Nodes are selected for expansion as follows:

Compute the cost of a node as ***the sum of the costs of the operations leading to it from the root***.

Always expand the cheapest node next.

Provided costs are never negative, the solution found will be the cheapest (optimal!).

Pseudo Code for Uniform Cost Search

```
Create an empty list UnexpNodes;
{Holds list of unexpanded nodes}
Add initial node to UnexpNodes;
WHILE (UnexpNodes not empty)
    N = Lowest cost item of UnexpNodes
    Remove N from UnexpNodes
    IF N is goal
    THEN Produce Solution and Return(Success);
    Expand N to produce list of successors of N;
    Compute costs of reaching successors by adding
    transition costs to cost of N.
    Add successors with costs to tail of UnexpNodes; 1)
Return(Failure)
```

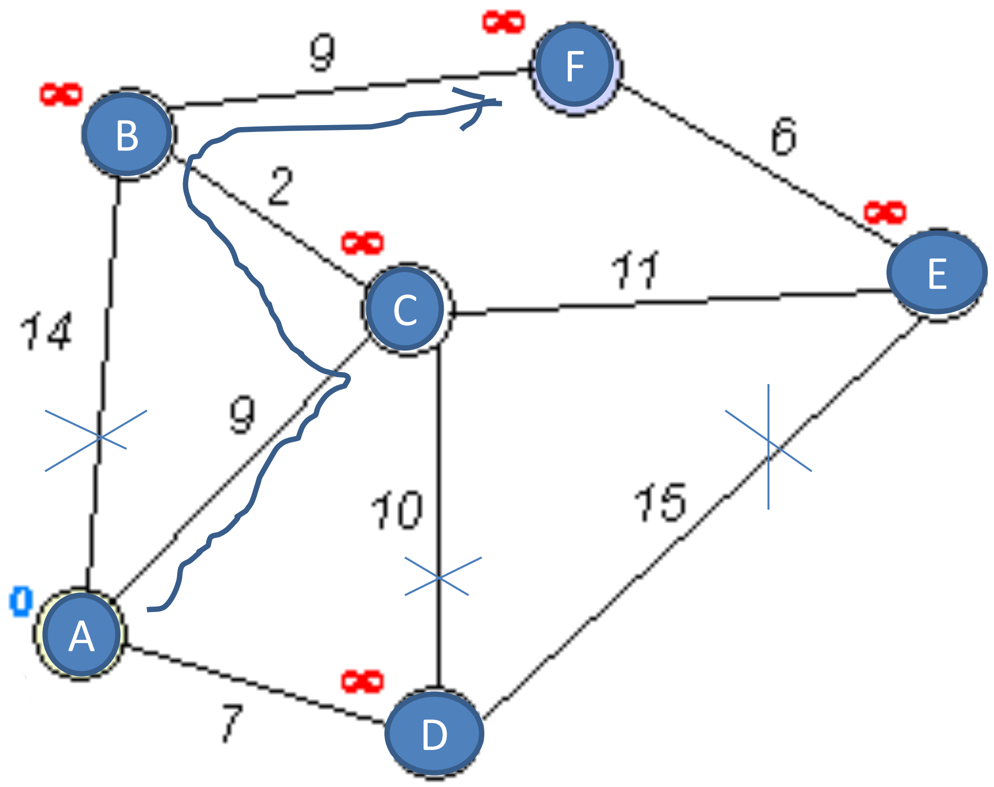
Will do exercises with toy problems in the class, which will help you a lot in understanding the pseudo code!

Note 1: If a cheaper route to a node in UnexpNodes has been found, replace its costlier parent node with the cheaper one.

An Example of Uniform Cost Search

There are 6 towns: A, B, C, D, E, and F. The numbers alongside the lines indicate the distances between the towns. Find the shortest road from town A to town F using uniform cost search:

Expand node A: produce new nodes B, C, and D.
Choose a node with minimum cost from nodes B, C, D to expend.
Expend D: produce a new Node E.



Choose a node from nodes B, C, E to expand.
Expand C: produce no new node, but update costs of nodes B (14->11) and E (22 -> 20).
……

Summary

Blind (Uninformed) Search Strategies

- Breadth First Search
- Depth First Search
- Depth Limited Search
- Iterative Deepening Search
- Uniform Cost Search

Comparison of Search Strategies

- Completeness
- Optimality
- Time Complexity
- Space Complexity