Module 2

Problem Solving using Search- (Single agent search)

Lesson
4

Uninformed Search

2.4 Search

Searching through a state space involves the following:

- A set of states
- Operators and their costs
- Start state
- A test to check for goal state

We will now outline the basic search algorithm, and then consider various variations of this algorithm.

2.4.1 The basic search algorithm

```
Let L be a list containing the initial state (L= the fringe)
Loop
        if L is empty return failure
        Node ← select (L)
          if Node is a goal
                  then return Node
                     (the path from initial state to Node)
        else generate all successors of Node, and
             merge the newly generated states into L
End Loop
```

We need to denote the states that have been generated. We will call these as nodes. The data structure for a node will keep track of not only the state, but also the parent state or the operator that was applied to get this state. In addition the search algorithm maintains a list of nodes called the fringe. The fringe keeps track of the nodes that have been generated but are yet to be explored. The fringe represents the frontier of the search tree generated. The basic search algorithm has been described above.

Initially, the fringe contains a single node corresponding to the start state. In this version we use only the OPEN list or fringe. The algorithm always picks the first node from fringe for expansion. If the node contains a goal state, the path to the goal is returned. The path corresponding to a goal node can be found by following the parent pointers. Otherwise all the successor nodes are generated and they are added to the fringe.

The successors of the current expanded node are put in fringe. We will soon see that the order in which the successors are put in fringe will determine the property of the search algorithm.

2.4.2 Search algorithm: Key issues

Corresponding to a search algorithm, we get a search tree which contains the generated and the explored nodes. The search tree may be unbounded. This may happen if the state space is infinite. This can also happen if there are loops in the search space. How can we handle loops?

Corresponding to a search algorithm, should we return a path or a node? The answer to this depends on the problem. For problems like N-queens we are only interested in the goal state. For problems like 15-puzzle, we are interested in the solution path.

We see that in the basic search algorithm, we have to select a node for expansion. Which node should we select? Alternatively, how would we place the newly generated nodes in the fringe? We will subsequently explore various search strategies and discuss their properties,

Depending on the search problem, we will have different cases. The search graph may be weighted or unweighted. In some cases we may have some knowledge about the quality of intermediate states and this can perhaps be exploited by the search algorithm. Also depending on the problem, our aim may be to find a minimal cost path or any to find path as soon as possible.

Which path to find?

The objective of a search problem is to find a path from the initial state to a goal state. If there are several paths which path should be chosen? Our objective could be to find any path, or we may need to find the shortest path or least cost path.

2.4.3 Evaluating Search strategies

We will look at various search strategies and evaluate their problem solving performance. What are the characteristics of the different search algorithms and what is their efficiency? We will look at the following three factors to measure this.

1. Completeness: Is the strategy guaranteed to find a solution if one exists?

2. Optimality: Does the solution have low cost or the minimal cost?

3. What is the search cost associated with the time and memory required to find a solution?
    a. <u>Time complexity</u>: Time taken (number of nodes expanded) (worst or average case) to find a solution.
    b. <u>Space complexity</u>: Space used by the algorithm measured in terms of the maximum size of fringe

The different search strategies that we will consider include the following:

1. Blind Search strategies or Uninformed search
    a. Depth first search
    b. Breadth first search
    c. Iterative deepening search
    d. Iterative broadening search
2. Informed Search
3. Constraint Satisfaction Search
4. Adversary Search

Blind Search

In this lesson we will talk about blind search or uninformed search that does not use any extra information about the problem domain. The two common methods of blind search are:

- BFS or Breadth First Search
- DFS or Depth First Search

2.4.4 Search Tree

Consider the explicit state space graph shown in the figure.

One may list all possible paths, eliminating cycles from the paths, and we would get the complete search tree from a state space graph. Let us examine certain terminology associated with a search tree. A search tree is a data structure containing a root node, from where the search starts. Every node may have 0 or more children. If a node X is a child of node Y, node Y is said to be a parent of node X.

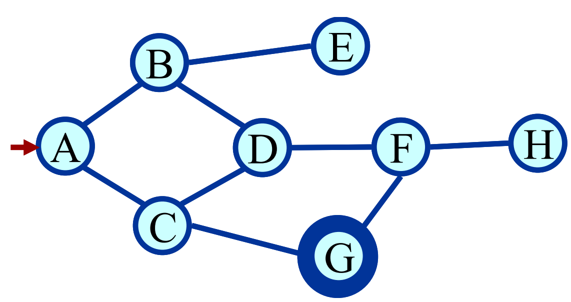


Figure 1: A State Space Graph

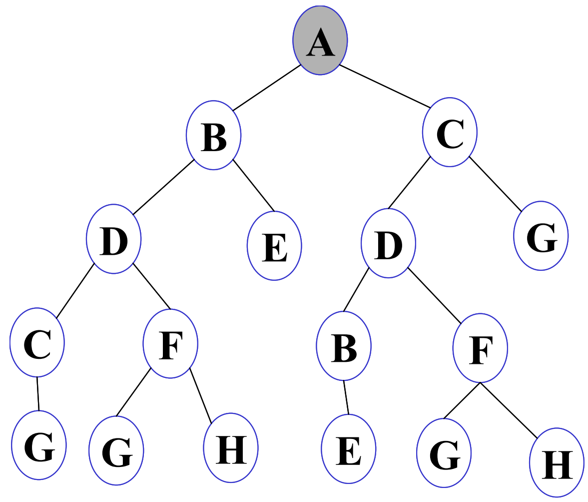


Figure 2: Search tree for the state space graph in Figure 25

Consider the state space given by the graph in Figure 25. Assume that the arcs are bidirectional. Starting the search from state A the search tree obtained is shown in Figure 26.

Search Tree – Terminology

- Root Node: The node from which the search starts.
- Leaf Node: A node in the search tree having no children.
- Ancestor/Descendant: X is an ancestor of Y is either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- Branching factor: the maximum number of children of a non-leaf node in the search tree
- Path: A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

We also need to introduce some data structures that will be used in the search algorithms.

Node data structure

A node used in the search algorithm is a data structure which contains the following:

1. A state description
2. A pointer to the parent of the node
3. Depth of the node
4. The operator that generated this node
5. Cost of this path (sum of operator costs) from the start state

The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe. Initially only the start node is in OPEN.

The search starts with the root node. The algorithm picks a node from OPEN for expanding and generates all the children of the node. Expanding a node from OPEN results in a closed node. Some search algorithms keep track of the closed nodes in a data structure called CLOSED.

A solution to the search problem is a sequence of operators that is associated with a path from a start node to a goal node. The cost of a solution is the sum of the arc costs on the solution path. For large state spaces, it is not practical to represent the whole space. State space search makes explicit a sufficient portion of an implicit state space graph to find a goal node. Each node represents a partial solution path from the start node to the given node. In general, from this node there are many possible paths that have this partial path as a prefix.

The search process constructs a search tree, where

- **root** is the initial state and
- **leaf nodes** are nodes
  - not yet expanded (i.e., in fringe) or
  - having no successors (i.e., "dead-ends")

Search tree may be infinite because of loops even if state space is small

The search problem will return as a solution a path to a goal node. Finding a path is important in problems like path finding, solving 15-puzzle, and such other problems. There are also problems like the N-queens problem for which the path to the solution is not important. For such problems the search problem needs to return the goal state only.

2.5 Breadth First Search

2.5.1 Algorithm

```
Breadth first search
Let fringe be a list containing the initial state
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            (merge the newly generated nodes into fringe)
            add generated nodes to the back of fringe
End Loop
```

Note that in breadth first search the newly generated nodes are put at the back of fringe or the OPEN list. What this implies is that the nodes will be expanded in a FIFO (First In

First Out) order. The node that enters OPEN earlier will be expanded earlier. This amounts to expanding the shallowest nodes first.

2.5.2 BFS illustrated

We will now consider the search space in Figure 1, and show how breadth first search works on this graph.

Step 1: Initially fringe contains only one node corresponding to the source state A.

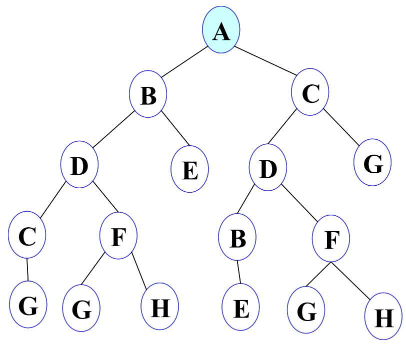


Figure 3

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

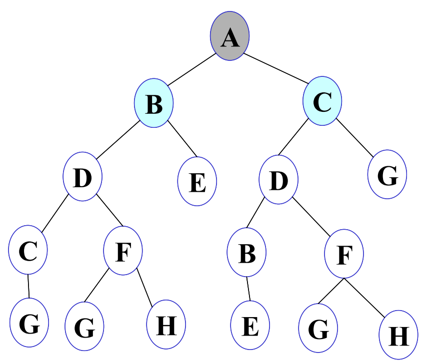


Figure 4

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

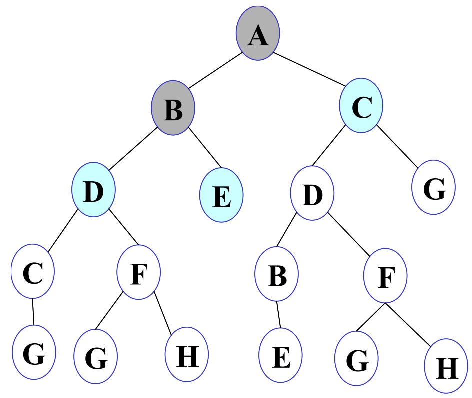


Figure 5

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

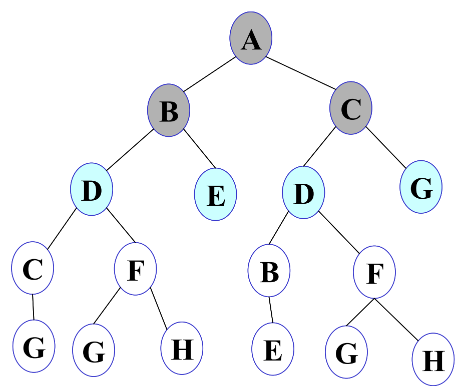


Figure 6

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

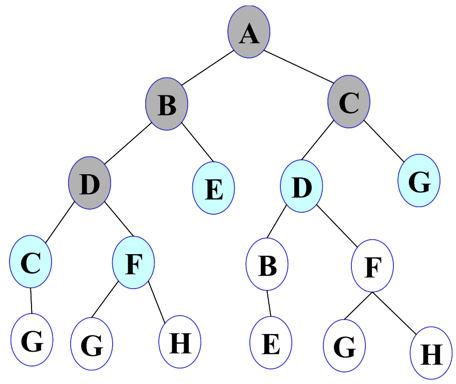
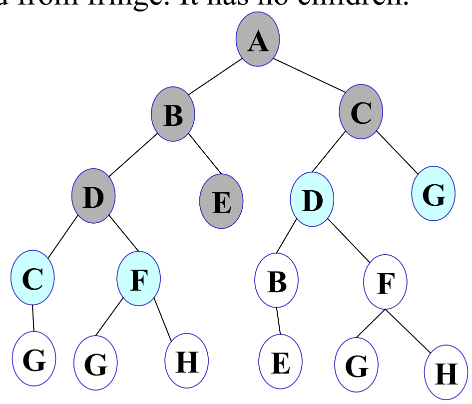


Figure 7

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.



FRINGE: D G C F

Step 7: D is expanded, B and F are put in OPEN.

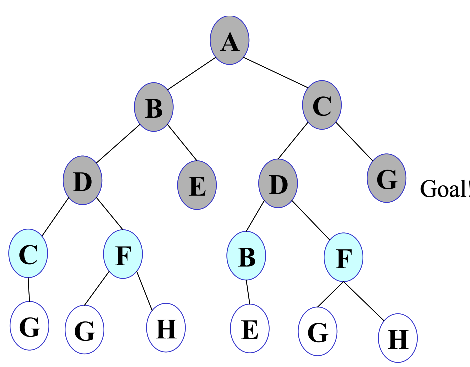


Figure 8

FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

2.5.3 Properties of Breadth-First Search

We will now explore certain properties of breadth first search. Let us consider a model of the search tree as shown in Figure 3. We assume that every non-leaf node has b children. Suppose that d is the depth o the shallowest goal node, and m is the depth of the node found first.

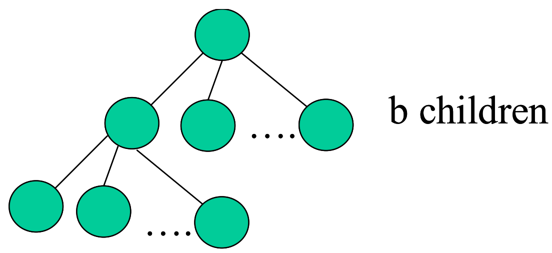


Figure 9: Model of a search tree with uniform branching factor b

Breadth first search is:

- Complete.
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- The algorithm has exponential time and space complexity. Suppose the search tree can be modeled as a b-ary tree as shown in Figure 3. Then the time and space complexity of the algorithm is O(bd) where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

A complete search tree of depth d where each non-leaf node has b children, has a total of

$$\mathbf{1 + b + b^2 + ... + b^d = (b^{(d+1)} - 1)/(b-1) \text{ nodes}}$$

Consider a complete search tree of depth 15, where every node at depths 0 to14 has 10 children and every node at depth 15 is a leaf node. The complete search tree in this case will have $O(10^{15})$ nodes. If BFS expands 10000 nodes per second and each node uses 100 bytes of storage, then BFS will take 3500 years to run in the worst case, and it will use 11100 terabytes of memory. So you can see that the breadth first search algorithm cannot be effectively used unless the search space is quite small. You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.

Advantages of Breadth First Search

Finds the path of minimal length to the goal.

Disadvantages of Breadth First Search

Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

2.6 Uniform-cost search

This algorithm is by Dijkstra [1959]. The algorithm expands nodes in the order of their cost from the source.

We have discussed that operators are associated with costs. The path cost is usually taken to be the sum of the step costs.

In uniform cost search the newly generated nodes are put in OPEN according to their path costs. This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.
Let g(n) = cost of the path from the start node to the current node n. Sort nodes by increasing value of g.
Some properties of this search algorithm are:

- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

2.7 Depth first Search

2.7.1 Algorithm

Depth First Search

Let *fringe* be a list containing the initial state
Loop
    if *fringe* is empty return failure
    Node ← remove-first (*fringe*)
     if Node is a goal
       then return the path from initial state to Node
    else generate all successors of Node, and
       merge the newly generated nodes into *fringe*
       add generated nodes to the front of *fringe*
End Loop

The depth first search algorithm puts newly generated nodes in the front of OPEN. This results in expanding the deepest node first. Thus the nodes in OPEN follow a LIFO order (Last In First Out). OPEN is thus implemented using a stack data structure.

2.7.2 DFS illustrated

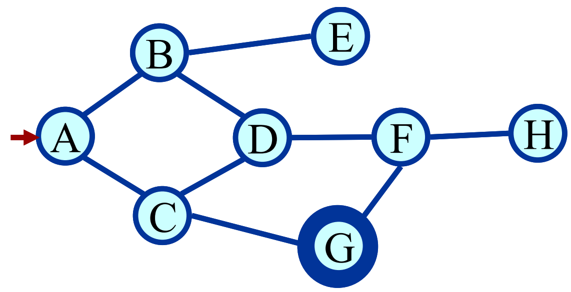


Figure 10

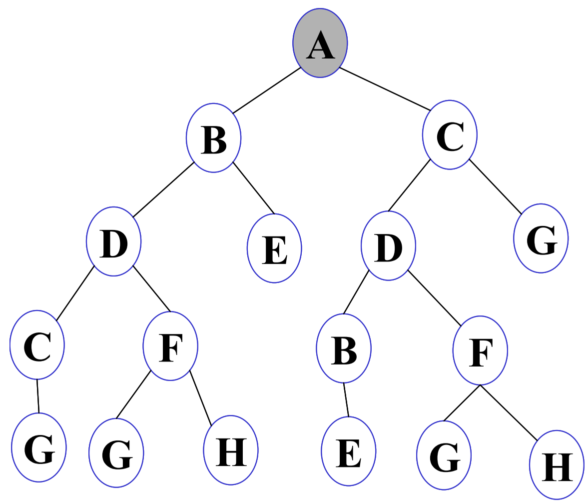


Figure 11: Search tree for the state space graph in Figure 34

Let us now run Depth First Search on the search space given in Figure 34, and trace its progress.

Step 1: Initially fringe contains only the node for A.

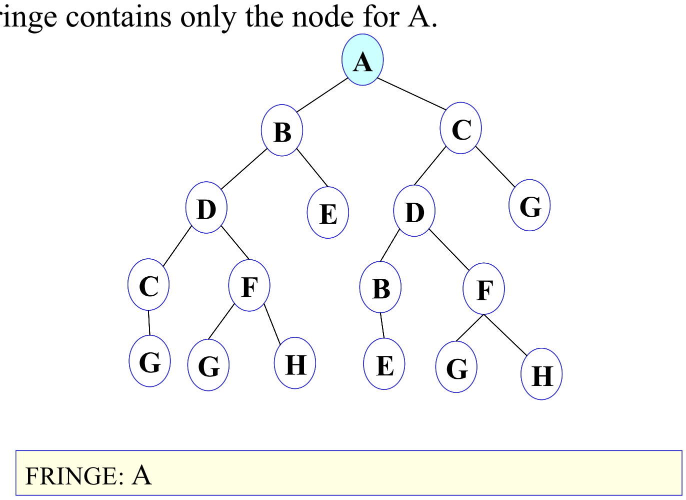
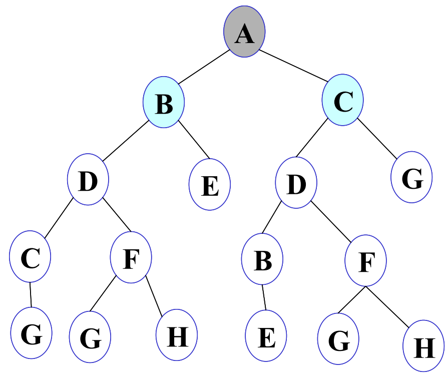


Figure 12

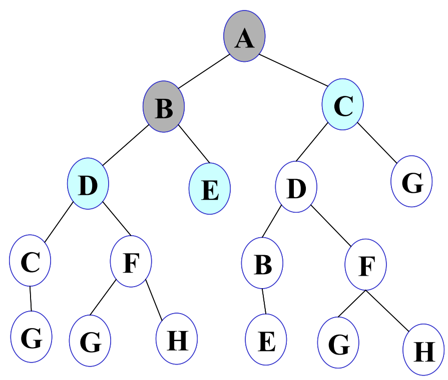
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

Figure 13

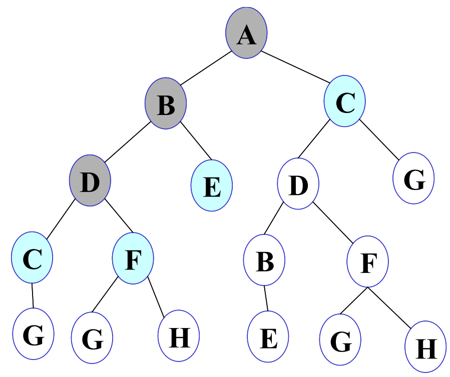
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

Figure 14

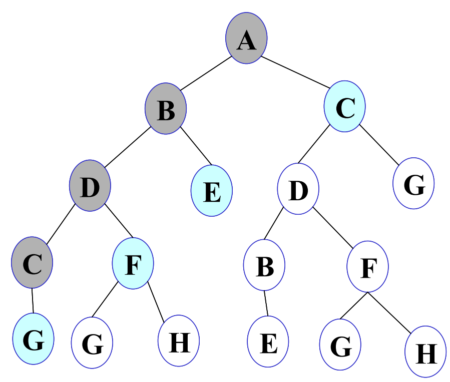
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

Figure 15

Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Figure 16

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.

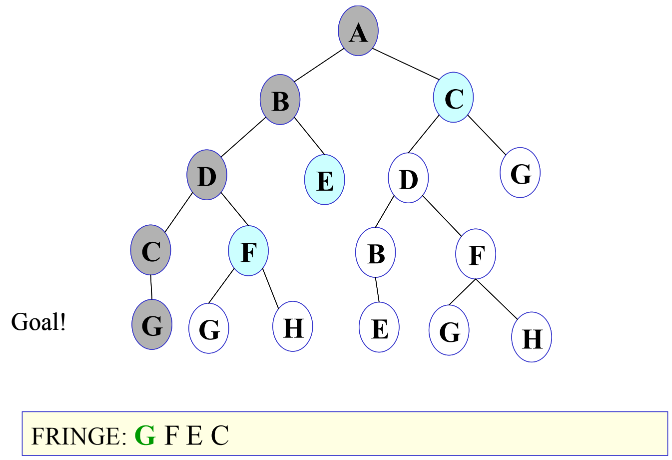


Figure 17

2.7.3 Properties of Depth First Search

Let us now examine some properties of the DFS algorithm. The algorithm takes exponential time. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$. However the space taken is linear in the depth of the search tree, O(bN).

Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus Depth First Search is not complete.

2.7.4 Depth Limited Search

A variation of Depth First Search circumvents the above problem by keeping a depth bound. Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.

Depth limited search (limit)

Let fringe be a list containing the initial state
Loop
    if fringe is empty return failure
    Node ← remove-first (fringe)
     if Node is a goal
        then return the path from initial state to Node
    else if depth of Node = limit return cutoff
    else add generated nodes to the front of fringe
End Loop

.

2.7.5 Depth-First Iterative Deepening (DFID)

First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

DFID
until solution found do *DFS with depth cutoff c* *c = c+1*

Advantage

- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth

Procedure

Successive depth-first searches are conducted – each with depth bounds increasing by 1

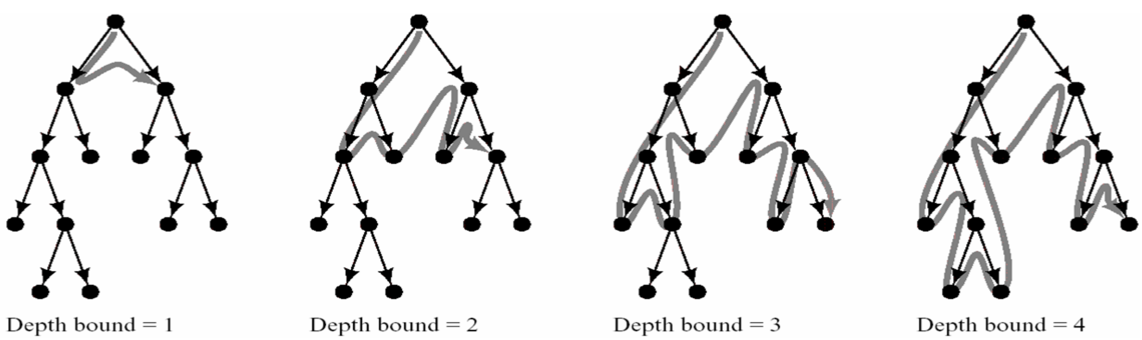


Figure 18: Depth First Iterative Deepening

Properties

For large *d* the ratio of the number of nodes expanded by DFID compared to that of DFS is given by *b/*(*b-1*).
For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative-deepening search than breadth-first search

The algorithm is
- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, $O(b^d)$

If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.
Hence $b^d + 2b^{(d-1)} + ... + db <= b^d / (1 - 1/b)^2 = O(b^d)$.

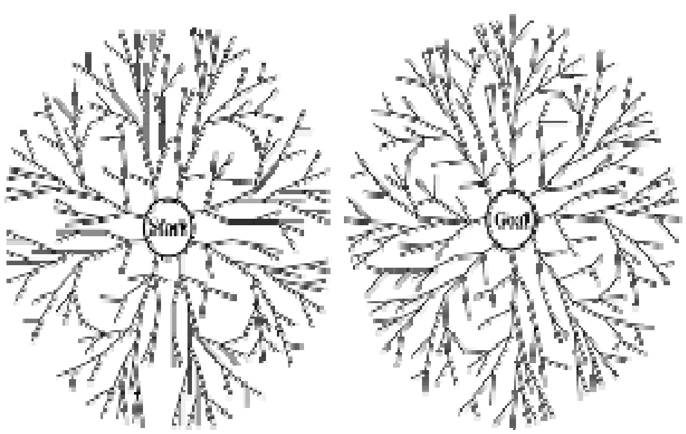
- **Linear space complexity**, O(bd), like DFS

Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly)
This algorithm is generally preferred for **large state spaces** where the **solution depth is unknown.**

There is a related technique called *iterative broadening* is useful when there are many goal nodes. This algorithm works by first constructing a search tree by expanding only one child per node. In the 2nd iteration, two children are expanded, and in the ith iteration I children are expanded.

Bi-directional search

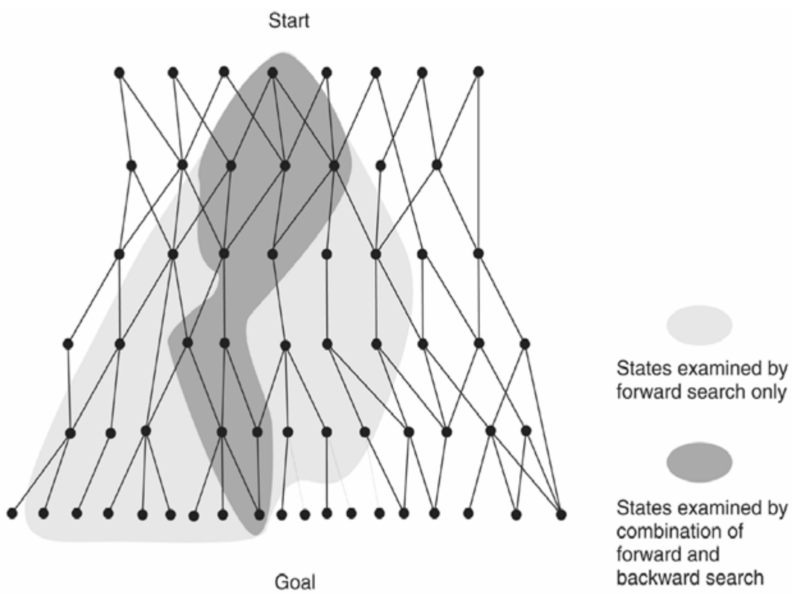
Suppose that the search problem is such that the arcs are bidirectional. That is, if there is an operator that maps from state A to state B, there is another operator that maps from state B to state A. Many search problems have reversible arcs. 8-puzzle, 15-puzzle, path planning etc are examples of search problems. However there are other state space search formulations which do not have this property. The water jug problem is a problem that does not have this property. But if the arcs are reversible, you can see that instead of starting from the start state and searching for the goal, one may start from a goal state and try reaching the start state. If there is a single state that satisfies the goal property, the search problems are identical.
How do we search backwards from goal? One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor. This is the motivation to consider bidirectional search.



Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when the frontiers intersect.

A search algorithm has to be selected for each half. How does the algorithm know when the frontiers of the search tree intersect? For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.

Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure.



Also note that the algorithm works well only when there are unique start and goal states. Question: How can you make bidirectional search work if you have 2 possible goal states?

Time and Space Complexities

Consider a search space with branching factor b. Suppose that the goal is d steps away from the start state. Breadth first search will expand $O(b^d)$ nodes.
If we carry out bidirectional search, the frontiers may meet when both the forward and the backward search trees have depth = d/2. Suppose we have a good hash function to check for nodes in the fringe. IN this case the time for bidirectional search will be $O((b^{d/2})$. Also note that for at least one of the searches the frontier has to be stored. So the space complexity is also $O((b^{d/2})$.

Comparing Search Strategies

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time Space Optimum? Complete?	b^d b^d Yes Yes	b^d bm No No	b^d bd Yes Yes	$b^{d/2}$ $b^{d/2}$ Yes Yes

Search Graphs

If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state. It is easy to consider a pathological example to see that the search space size may be exponential in the total number of states.

In many cases we can modify the search algorithm to avoid repeated state expansions. The way to avoid generating the same state again when not required, the search algorithm can be modified to check a node when it is being generated. If another node corresponding to the state is already in OPEN, the new node should be discarded. But what if the state was in OPEN earlier but has been removed and expanded? To keep track of this phenomenon, we use another list called CLOSED, which records all the expanded nodes. The newly generated node is checked with the nodes in CLOSED too, and it is put in OPEN if the corresponding state cannot be found in CLOSED. This algorithm is outlined below:

```
Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
        if fringe is empty return failure
        Node ← remove-first (fringe)
         if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
              generate all successors of Node S
              for all nodes m in S
                  if m is not in fringe or closed
                      merge m into fringe
End Loop
```

But this algorithm is quite expensive. Apart from the fact that the CLOSED list has to be maintained, the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED. Unless there is a very efficient way to index nodes, this will require additional overhead for every node.

In many search problems, we can adopt some less computationally intense strategies. Such strategies do not stop duplicate states from being generated, but are able to reduce many of such cases.

The simplest strategy is to not return to the state the algorithm just came from. This simple strategy avoids many node re-expansions in 15-puzzle like problems.

A second strategy is to check that you do not create paths with cycles in them. This algorithm only needs to check the nodes in the current path so is much more efficient than the full checking algorithm. Besides this strategy can be employed successfully with depth first search and not require additional storage.

The third strategy is as outlined in the table. Do not generate any state that was ever created before.

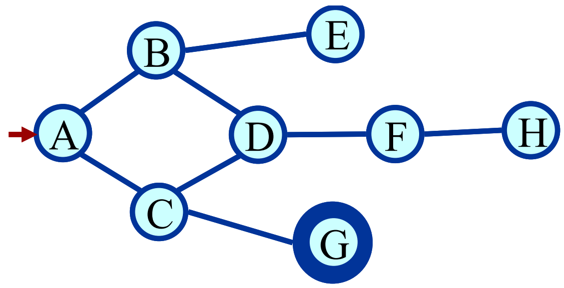
Which strategy one should employ must be determined by considering the frequency of "loops" in the state space.

Questions for Module 2

1. Give the initial state, goal test, sucessor function, and cost function for each of the following.
   Choose a formulation that is precise enough to be implemented.
   a) You have to colour a planar map using only four colours, in such a way that no two adjacent regions have the same colour.
   b) In the travelling salesperson problem (TSP) there is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city.
   c) Missionaries & Cannibals problem: 3 missionaries & 3 cannibals are on one side of the river. 1 boat carries 2. Missionaries must never be outnumbered by cannibals. Give a plan for all to cross the river.

2. Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water. You may use the following state space formulation.
   State = (x,y), where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
   Initial State = (5,0)
   Goal State = (*,1), where * means any amount
   Create the search tree. Discuss which search strategy is appropriate for this problem.

3. Consider the following graph.



Starting from state A, execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties.

4. Suppose you have the following search space:

State	next	cost
A	**B**	**4**
A	**C**	**1**
B	**D**	**3**
B	**E**	**8**
C	**C**	**0**
C	**D**	**2**
C	**F**	**6**
D	**C**	**2**
D	**E**	**4**
E	**G**	**2**
F	**G**	**8**

a) Draw the state space of this problem.
b) Assume that the initial state is **A** and the goal state is **G**. Show how each of the following search strategies would create a search tree to find a path from the initial state to the goal state:

I. Breadth-first search
II. Depth-first search
III. Uniform cost search
IV. Iterative deepening search

At each step of the search algorithm, show which node is being expanded, and the content of *fringe.* Also report the eventual solution found by each algorithm, and the solution cost.

5. Suppose that breadth first search expands N nodes for a particular graph. What will be the maximum number of nodes expanded by Iterative Deepening search ?

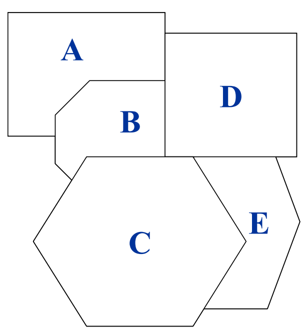
Solutions

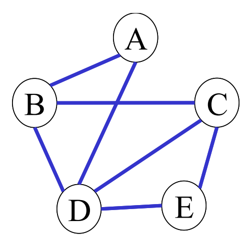
1. You have to colour a planar map using only four colours, in such a way that no two adjacent regions have the same colour.

   The map is represented by a graph. Each region corresponds to a vertex of the graph. If two regions are adjacent, there is an edge connecting the corresponding vertices.

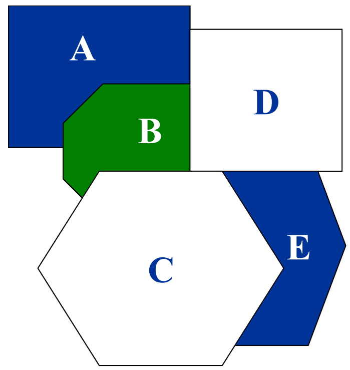
   The vertices are named <v1, v2, … , vN>.
   The colours are represented by c1, c2, c3, c4.
   A state is represented as a N-tuple representing the colours of the vertices. A vertes has colour x if its colour has not yet been assigned. An example state is:
   {c1, x, c1, c3, x, x, x …}
   colour(i) denotes the colour of si.

Consider the map below consisting of 5 regions namely A, B, C, D and E. The adjacency information is represented by the corresponding graph shown.





A state of this problem is shown below.



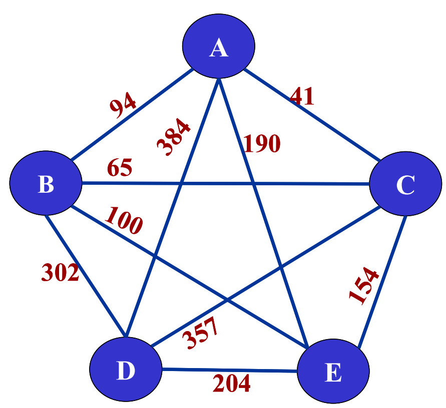
This state is represented as {blue, green, x, x, blue}.

The initial state for this problem is given as {x, x, x, x, x}
The goal test is as follows. For every pair of states s_i and s_j that are adjacent, colour(i) must be different from colour(j).

The successor functions are of the form:

- Change (i, c): Change the colour of a state i to c.

2. In the travelling salesperson problem (TSP) there is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city.



Y: set of N cities
d(x,y) : distance between cities x and y. $x,y \in Y$
A state is a Hamiltonian path (does not visit any city twice)
X: set of states

X: set of states. X =
{(x**1**, x**2**, …, x**n**)|
n=1, …, N+1,
xi $\in$Y for all I,

$x_i \neq x_j$ unless $i=1, j=N+1\}$
Successors of state
$(x\mathbf{1}, x\mathbf{2}, \ldots, x\mathbf{n})$:

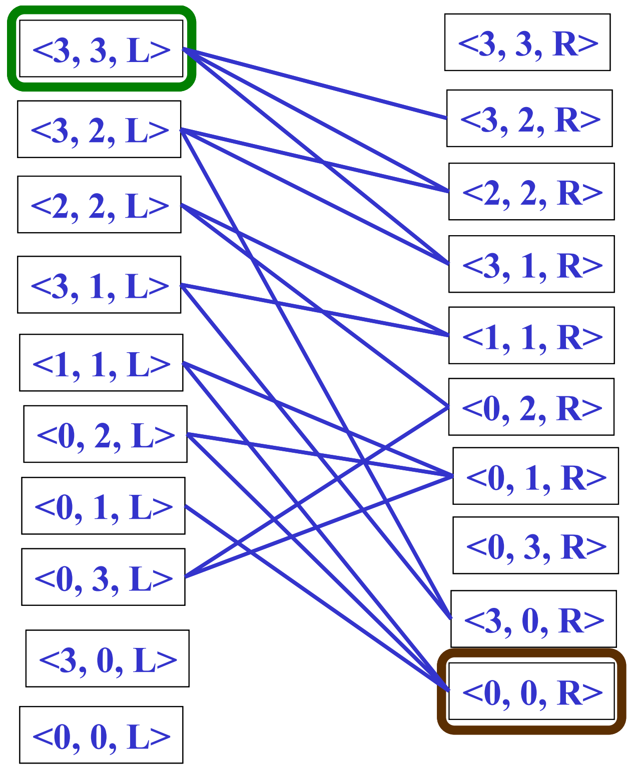
$$\delta \quad (x\mathbf{1}, x\mathbf{2}, \ldots, x\mathbf{n}) = \{(x\mathbf{1}, x\mathbf{2}, \ldots, x\mathbf{n}, \mathbf{xn+1}) \mid \mathbf{xn+1} \in \mathbf{Y}$$
$$\mathbf{xn+1} \neq \mathbf{xi} \text{ for all } 1 \leq i \leq n \}$$

The set of goal states include all states of length N+1

3. **Missionaries & Cannibals** problem: 3 missionaries & 3 cannibals are on one side of the river. 1 boat carries 2. Missionaries must never be outnumbered by cannibals. Give a plan for all to cross the river.
   State: <M, C, B>
   - ♦ M: no of missionaries on the left bank
   - ♦ C: no of cannibals on the left bank
   - ♦ B: position of the boat: L or R
4. Initial state: <3, 3, L>
5. Goal state: <0, 0, R>
6. Operators: <M,C>
   - ► M: No of missionaries on the boat
   - ► C: No of cannibals on the boat

   Valid operators: <1,0> <2,0>, <1,1>, <0,1> <0,2>

<3, 3, L>	**<3, 3, R>**
<2, 3, L>	**<2, 3, R>**
<1, 3, L>	**<1, 3, R>**
<3, 2, L>	**<3, 2, R>**
<2, 2, L>	**<2, 2, R>**
<1, 2, L>	**<1, 2, R>**
<3, 1, L>	**<3, 1, R>**
<2, 1, L>	**<2, 1, R>**
<1, 1, L>	**<1, 1, R>**
<0, 2, L>	**<0, 2, R>**
<0, 1, L>	**<0, 1, R>**
<0, 3, L>	**<0, 3, R>**
<3, 0, L>	**<3, 0, R>**
<2, 0, L>	**<2, 0, R>**
<1, 0, L>	**<1, 0, R>**
<0, 0, L>	**<0, 0, R>**



2. Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water. You may use the following state space formulation.

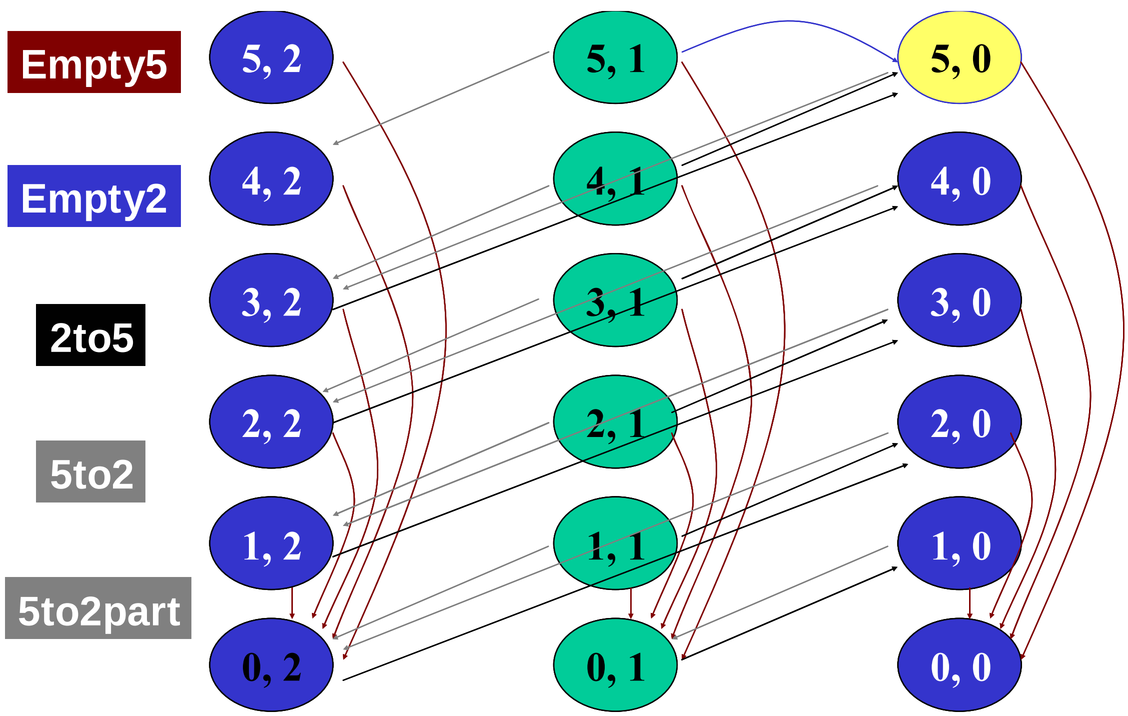
- State = (x,y), where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
- Initial State = (5,0)
- Goal State = (*,1), where * means any amount

Create the search tree. Discuss which search strategy is appropriate for this problem.

Solution:
The table below shows the different operators and their effects.

Name	**Cond.**	**Transition**	**Effect**
Empty5	–	(x,y)→(0,y)	Empty 5-gal. jug
Empty2	–	(x,y)→(x,0)	Empty 2-gal. jug
2to5	$x \leq 3$	(x,2)→(x+2,0)	Pour 2-gal. into 5-gal.
5to2	$x \geq 2$	(x,0)→(x-2,2)	Pour 5-gal. into 2-gal.
5to2part	$y < 2$	(1,y)→(0,y+1)	Pour partial 5-gal. into 2-gal.

The figure below shows the different states and the transitions between the states using the operators above. The transitions corresponding to Empty2 have not been marked to keep the figure clean.

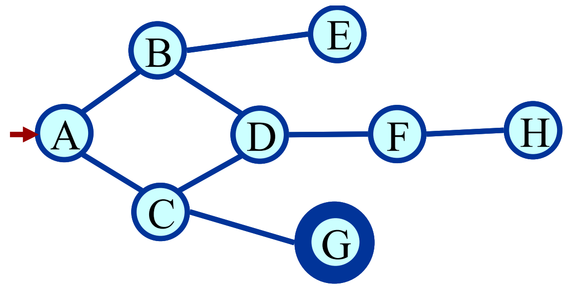


(5,0) is the initial state.
(0,1) is the goal state.
A solution to this problem is given by the path
(5,0) – (3,2) – (3,0) – (1,2) – (1,0) – (0,1).
using the operators
5to2, Empty2, 5to2, Empty2, 5to2.

Depth search is not appropriate
- The search space is a graph.
- The search tree has many repeated states

Breadth first search is appropriate

3. Consider the following graph.



Starting from state A, execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties.

Solution:

Step	Fringe	Node Expanded	Comments
1	A		
2	B C	A	
3	D E C	B	
4	F E C	D	
5	H E C	F	
6	E C	H	
7	C	E	
8	D G	C	
9	F G	D	
10	H G	F	
11	G	H	
12		G	Goal reached!