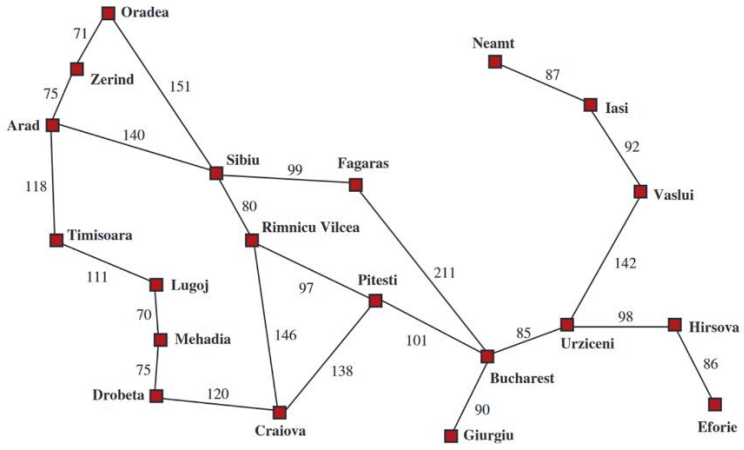
Introduction to Artificial Intelligence

A ***blind search*** (also called an ***uninformed search***) is a search that has no information about its domain. The only thing that a blind search can do is distinguish a non-goal state from a goal state.

Consider the following, simplified map of Romania. Assume you are now in the city of Arad and want to get to Bucharest. What is the optimal path from the city Arad to Bucharest



In blind searches we are about to consider only differ in the order in which nodes are expanded.

For every graph or tree search the generalized data structure terminology *Fringe* or *Frontier* is used. For searching we need to maintain two lists: OPEN list (for traversal/to be explored / to be visited) and CLOSED list (Visited/ Explored/traversed). An OPEN list that keeps track of the current 'immediate' nodes available for traversal/to be explored/ to be visited and a CLOSED list that keeps track of the nodes already traversed/explored/visited.

Tree Search *vs* Graph Search

Both tree and graph searches produce a tree *(from which you can derive a path) while exploring the search space, which is usually represented as a graph*.

Differences (Tree Search vs Graph Search)

- *In the case of a graph search, we use a list, called the* ***closed list*** *(also called* ***Explored / Visited set****), to keep track of the nodes that have already been visited and expanded, so that they are not visited and expanded again.*

- *In the case of a* ***tree search****, we do* ***not*** *keep this closed list. Consequently, the same node can be visited multiple (or even infinitely many) times, which means that the produced tree (by the tree search) may contain the* ***same node multiple times****.*

The only difference between tree search and graph search is that tree search does not need to store the explored set, because we are guaranteed never to attempt to visit the same state twice.

Generic Graph Search:

TREE-SEARCH(problem):

Initialize the frontier using the initial state of problem

Loop forever:

If frontier is empty, return failure

Choose a leaf node from the frontier and remove it (from the frontier)

If the node contains a goal state, return the corresponding solution

If the node did not contain a goal state, expand the chosen node, adding the resulting nodes to the frontier

GRAPH-SEARCH(problem):

Initialize the frontier using the initial state of the problem

Initialize the explored set to empty

Loop forever:

If frontier is empty, return failure

Choose a leaf node from the frontier and remove it (from the frontier)

If the node contains a goal state, return the corresponding solution

Add the node to the explored set

If the node did not contain a goal state, expand the chosen node, adding the resulting nodes to the frontier, but only if the resulting node is not already in the frontier or the explored set

For a finite graph without cycles, it will eventually find a solution no matter which order you select paths on the frontier. A solution is a path from a start node to a goal node.

Tree Search Algorithm	Vs	Graph Search Algorithm
- We might repeat some states - But we do NOT need to remember states		- We remember all the states that have been explored - But we do NOT repeat some states

Breadth-First Search (BFS)

Breadth-first search: Run the generic graph search algorithm with the frontier stored as a (LIFO) queue

In breadth-first strategy we expand the root level first and then we expand all those nodes (i.e. those at level 1) before we expand any nodes at level 2. OR to put it another way, all nodes at level *d* are expanded before any nodes at level *d*+1. This is because the breadth first search use *fringe/frontier* as a QUEUE , i.e. First In First Out (FIFO) that adds expanded nodes to the end of the queue and removes the nodes from front end.

The following observations about breadth-first searches:

- It is a very systematic search strategy as it considers all nodes (states) at level 1 and then all states at level 2 etc.
- If there is a solution breadth first search is *guaranteed* to find it.
- If there are several solutions then breadth first search will always find the shallowest goal state first and if the cost of a solution is a non-decreasing function of the depth then it will always find the cheapest solution.

Properties of Breath First Search (BFS):

Complete: YES

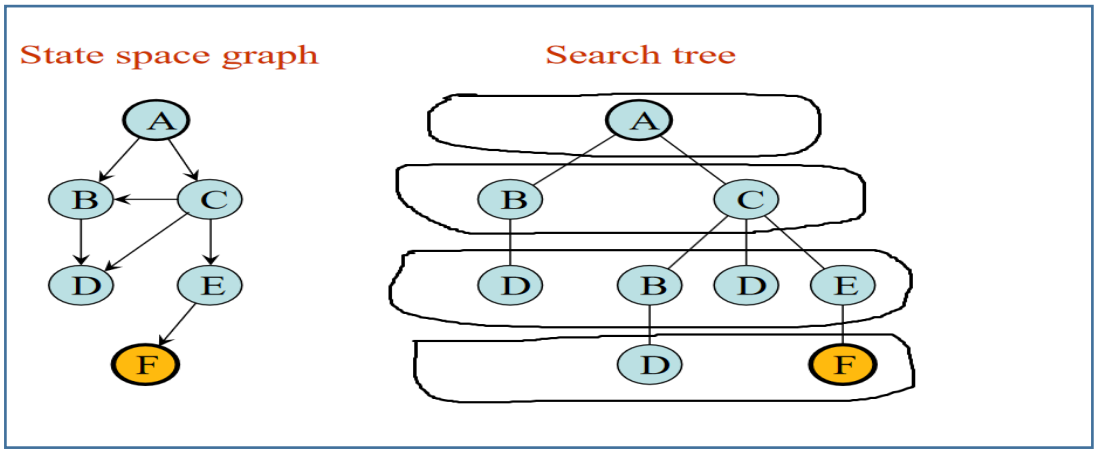
Optimal: YES

Time Complexity: $O(b^d)$

Space Complexity: $O(b^d)$

Note: Space is more of a factor to breadth first search than time

Example 1: BFS using Tree Search Algorithm (NO need to maintain visited or expanded list or CLOSED list), Repeated visit of some nodes.

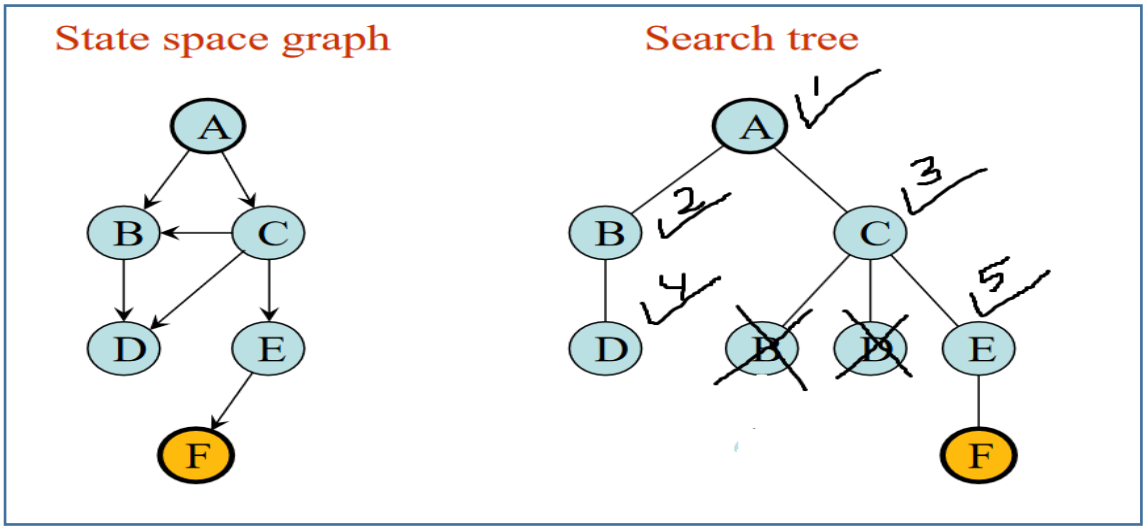


Visited Sequence: A B C D B D E D F

Path (Solution) : A — B – C –D –E—F

Number of nodes expanded: 5, Number of nodes tested (for GOAL): 9

Example 2: BFS using Graph Search Algorithm (Need to maintain visited or expanded list or CLOSED list), Repeated visit of nodes NOT allowed.



Visited Sequence: A B C D C E F

Path (Solution) : A — B – C –D –E

Number of nodes expanded: 4, Number of nodes tested (for GOAL): 5

Uniform Cost Search (UCS)

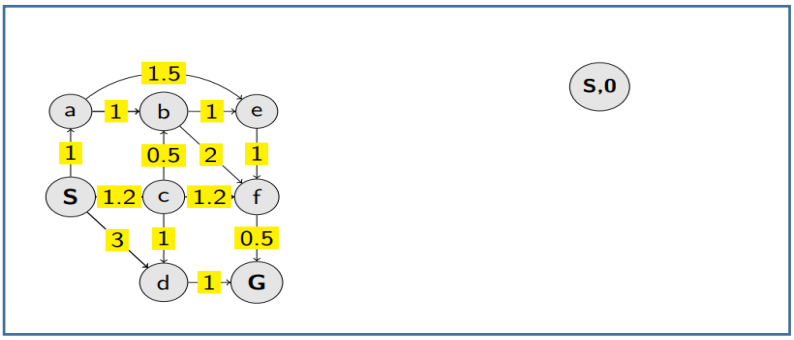
It works by always expanding the lowest cost node on the fringe, where the cost is the path cost, g(n).

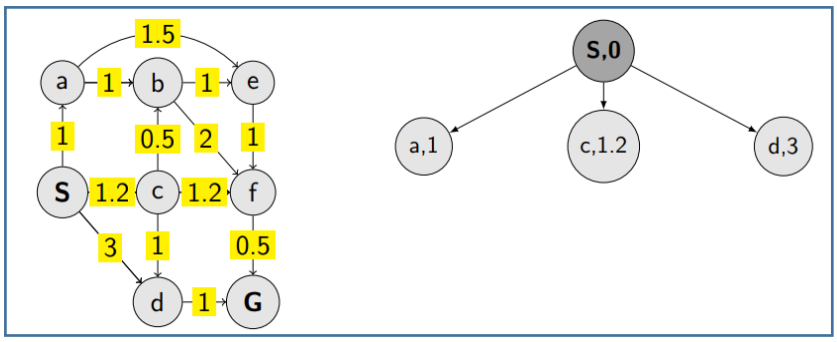
Breadth first search (BFS) is a uniform cost search with *g(n) = DEPTH(n).*

The UCS graph search

```
function UCS_GRAPH_SEARCH(env) return a solution or failure
    node ← env.observe()
    frontier ← priority_queue(node)
    explored ← set()
    while frontier not empty do
        node ← frontier.pop()
        if node contains Goal then return node
        end if
        explored.add(node.state)
        child_nodes ← env.expand(node.state)
        for all child_nodes do
            if child_node.state not in explored and not in frontier then
                frontier.insert(child_node)
            else if child_node.state in frontier with higher cost then
                replace that node with the child_node
            end if
        end for
    end while
end function
```

****In summary, uniform cost search will find the cheapest solution provided that the cost of the path never decreases as we proceed along the path.*

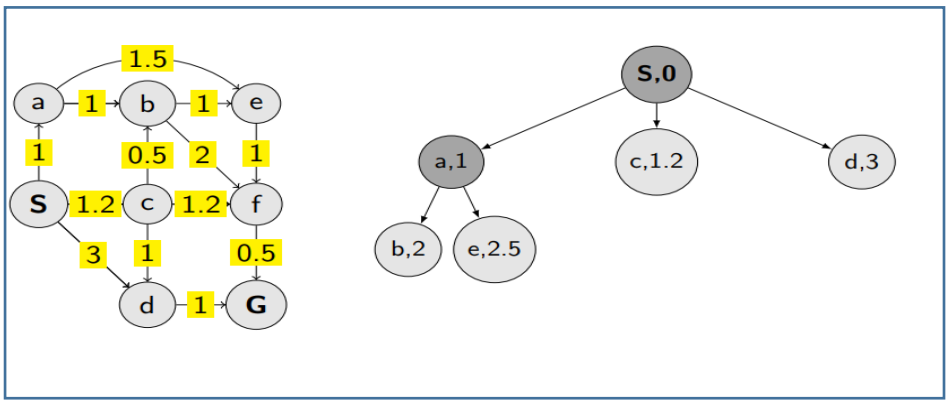


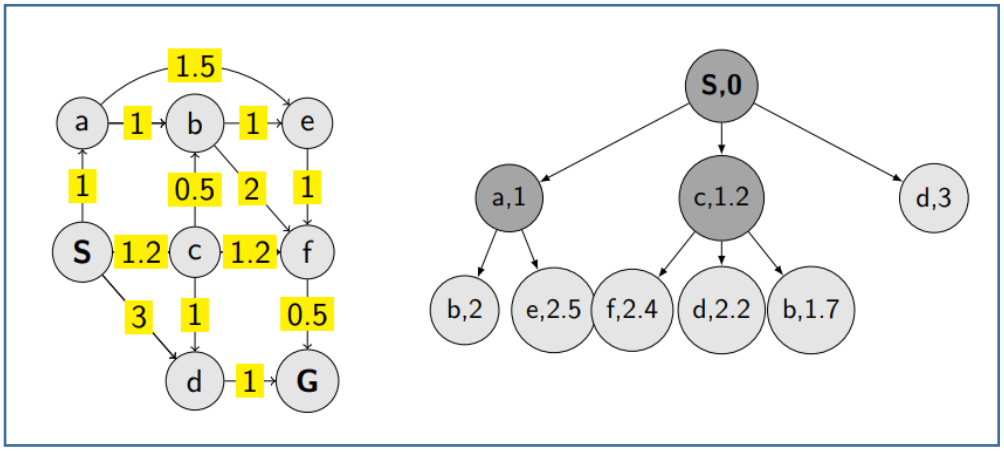


Remember: In every step always expand the lowest cost node on the fringe (Priority Queue)

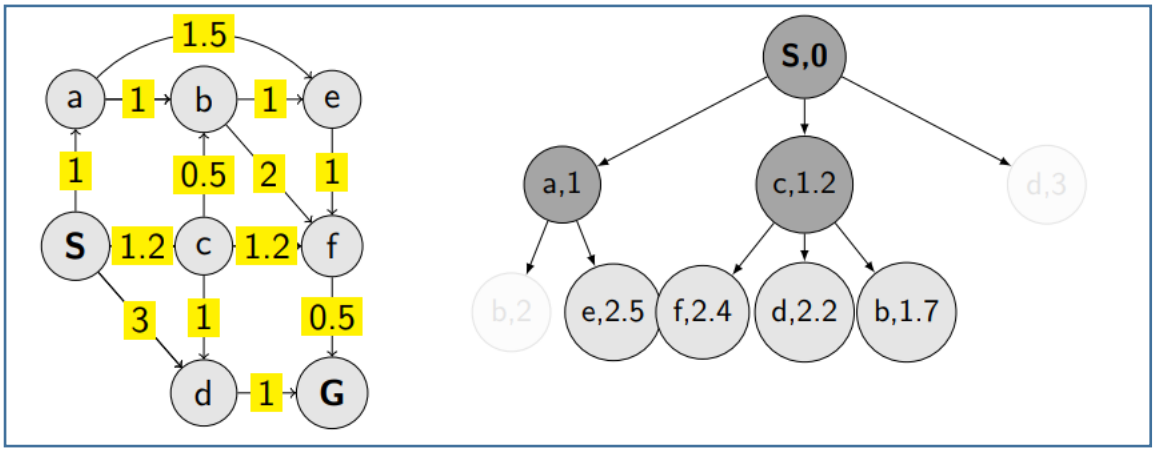
OR

Expand lowest cost leaf node (Not visited) from the tree

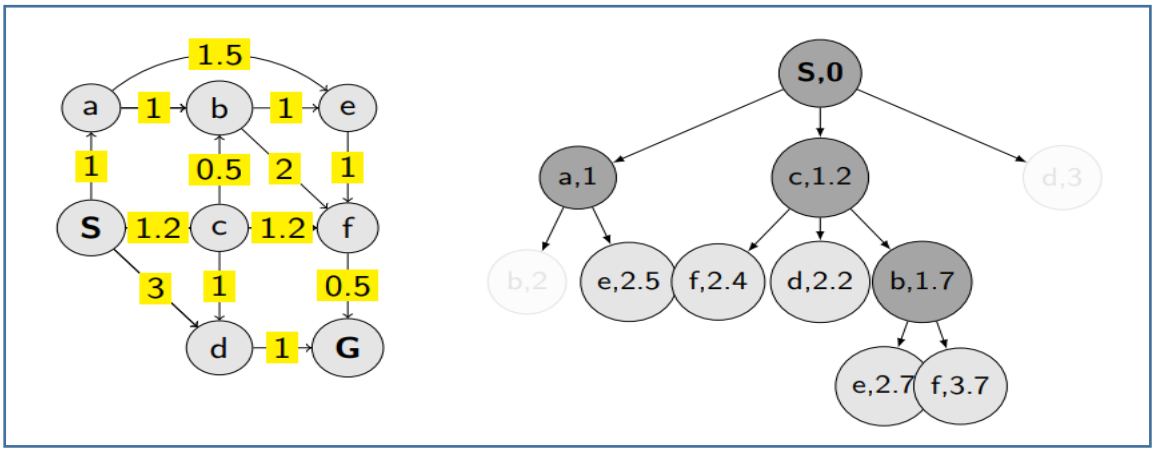




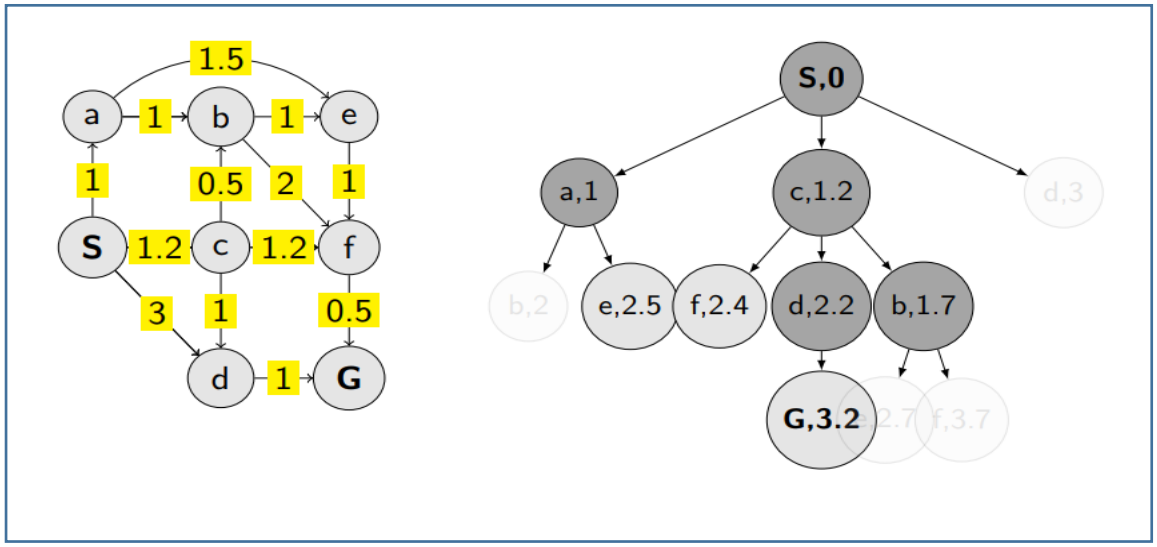
Always expands the lowest cost node from the priority queue or from the leaf nodes (not visited) of the tree in every step.



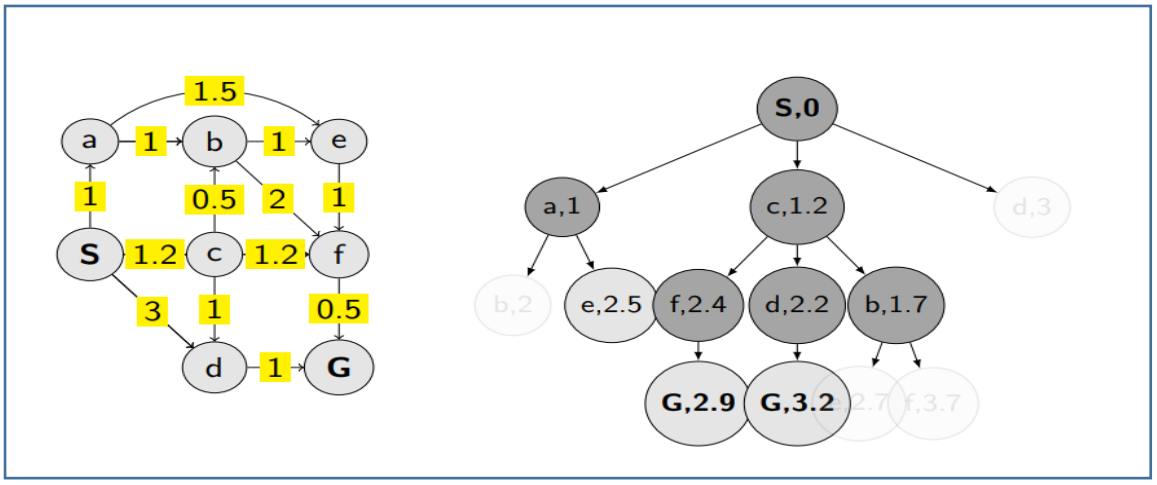
(b,2) disappears as (b,1.7) appears update with lower cost. Similarly, (d,3) disappears as (d,2.2) appears with lowest cost.



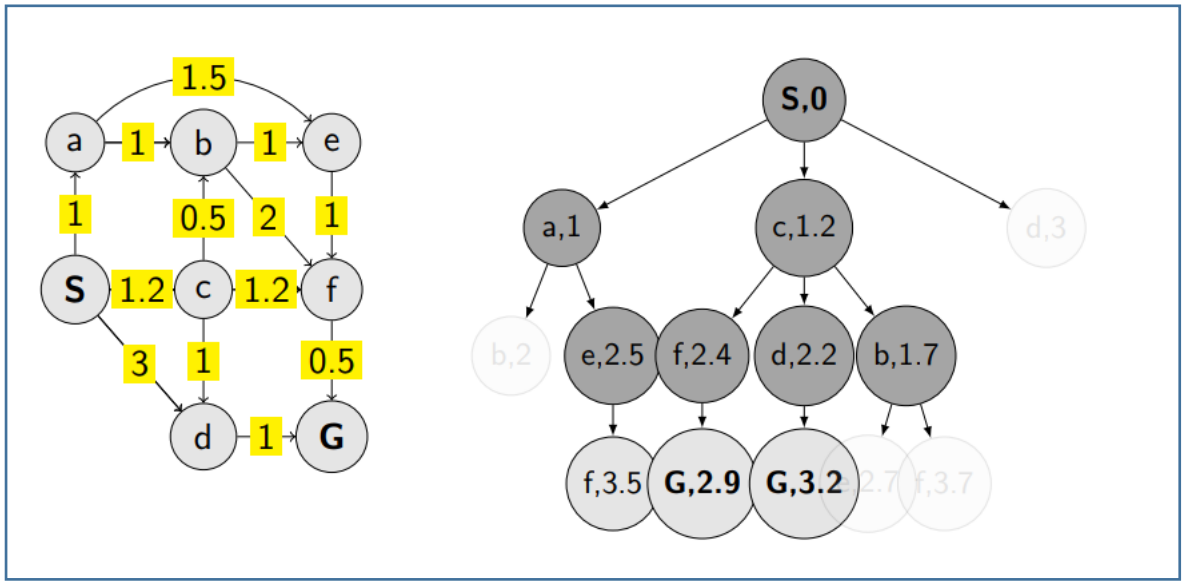
(e,2.7) and (f,3.7) appear to immediately disappear again - their cost is higher than already available for those states.

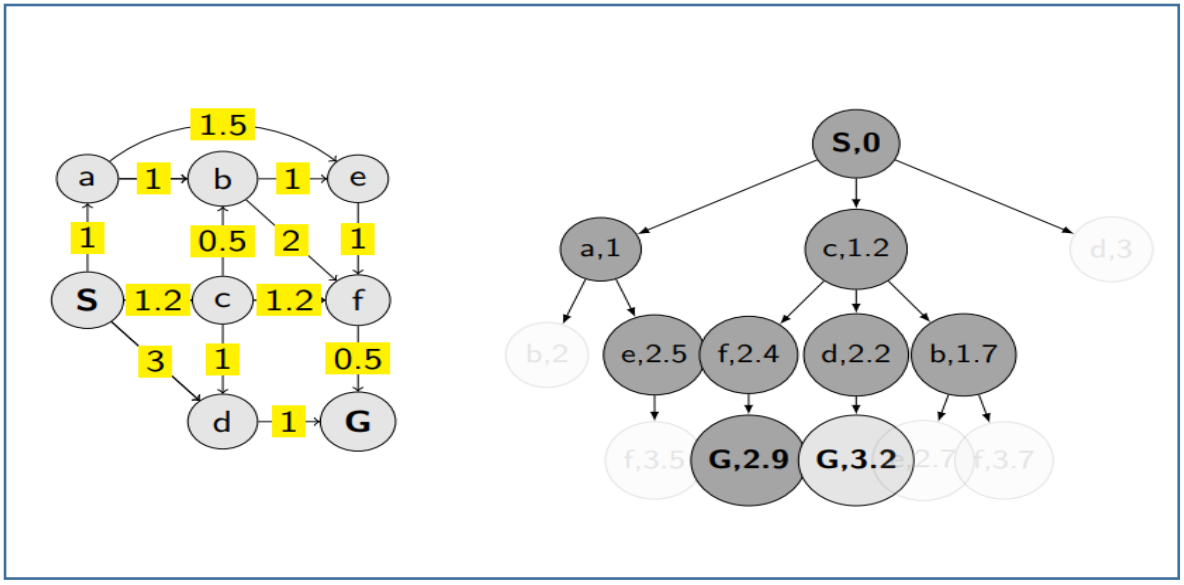


Always expands the lowest cost node from the priority queue or from the leaf nodes (not visited) of the tree in every step.



Always expands the lowest cost node from the priority queue or from the leaf nodes (not visited) of the tree in every step.





Here, there are three leaf nodes (f,3.5), (G,2.9), (G,3.2) and (G,2.9) is the lowest cost leaf node among the leaf nodes, so selected for expansion and the node is tested as a GOAL node so terminate process

CLOSED LIST/ VISITED SEQUENCE LIST: S A C B D F E

PATH (SOLUTION): S—C—F —G

PATH COST: (1.2 + 1.2 + 0.5)=2.9

Depth-First Search (DFS)

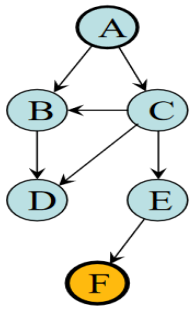
Depth-first search**: Run the generic graph search algorithm with the frontier stored as a (LIFO) stack**

Depth first search explores one branch of a tree before it starts to explore another branch. It can be implemented by adding newly expanded nodes in a STACK. i.e. Last In First Out (LIFO)

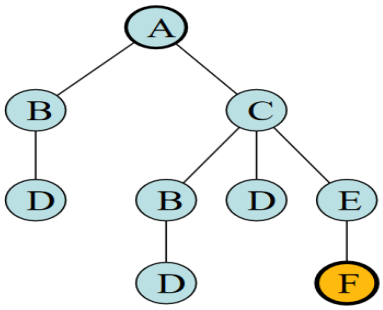
- It only needs to store the path from the root to the leaf node as well as the unexpanded nodes. For a state space with a branching factor of *b* and a maximum depth of *m*, depth first search requires storage of *bm* nodes.
- The time complexity for depth first search is b^m in the worst case
- If depth first search goes down a infinite branch it will not terminate if it does not find a goal state. If it does find a solution there may be a better solution at a higher level in the tree. Therefore, depth first search is *neither complete nor optimal*.

Example 1: DFS using Tree Search algorithm (NO Need to maintain Visited/Closed List)

State space graph



Search tree

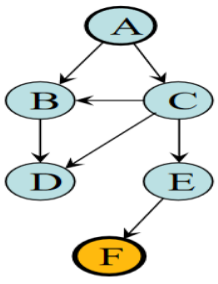


Queue

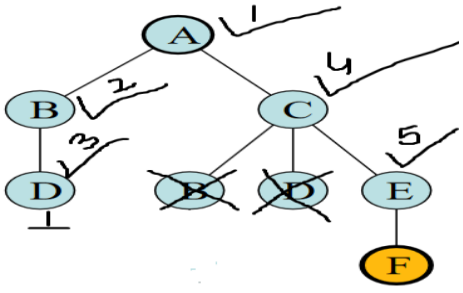
(A)
(B C)
(D C)
(C)
(B D E)
(D D E)
(D E)
(E)
(F)

Example 2: DFS using Graph Search algorithm (Below figure) (Never visit a node twice) TRY and Verify it?

State space graph

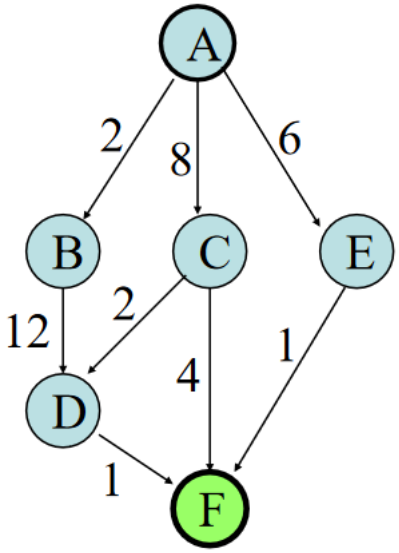


Search tree



(Closed / Visited/Explored sequence) List: A B D C E F
Path(solution): A—C–E –F

Example: Try Breadth-first and Uniform Cost Search



BFS: (TREE SEARCH ALGORITHM)

Visiting Sequence: A B C E D D F

BFS: (GRAPH SEARCH ALGORITHM)

Visiting Sequence: Find?

UCS: (Never visit nodes twice)

Visiting Sequence: A B E F

Path(Solution) : A—E –F

Cost: 7

Depth-Limited Search (DLS):

- The problem with depth first search is that the search can go down an infinite branch and thus never return.
- Depth-limited search avoids this problem by imposing a depth limit which effectively terminates the search at that depth.
- The algorithm can be implemented using the general search algorithm by using operators to keep track of the depth.

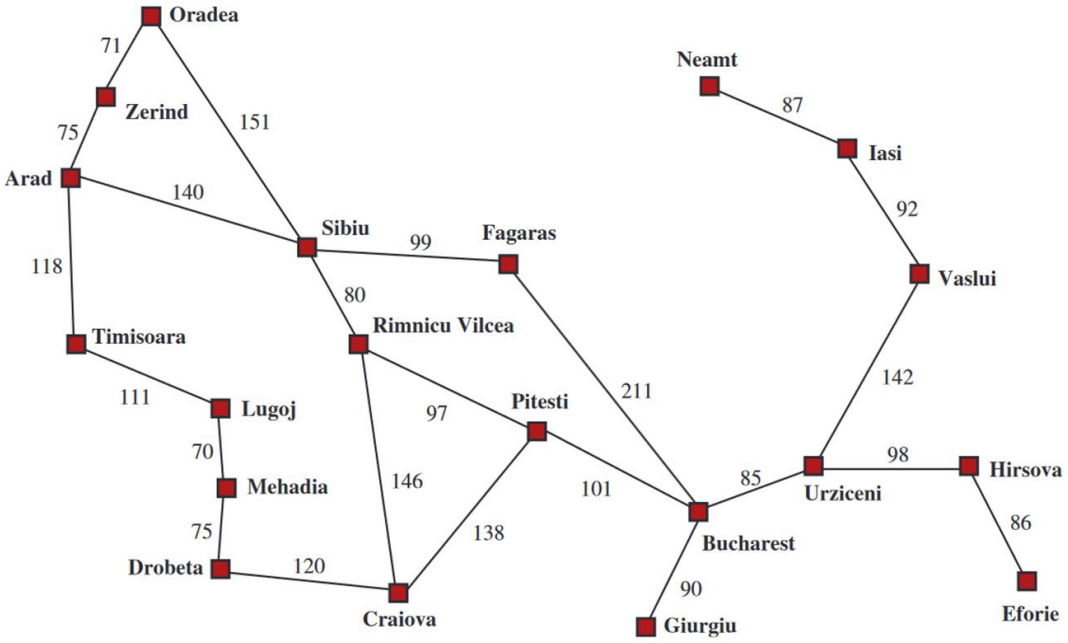
The choice of the depth parameter can be an important factor. Choosing a parameter that is too deep is wasteful in terms of both time and space. But choosing a depth parameter that is too shallow may result in never reaching a goal state.

As long as the depth parameter, l (this is lower case L, not the digit one), is set "deep enough" then we are guaranteed to find a solution if it exists. Therefore, it is complete as long as l >= d (where d is the depth of the solution). If this condition is not met then depth limited search is not complete.

The space requirements for depth limited search is O(bl).
The time complexity is $O(b^l)$
This algorithm is NOT complete, Therefore NOT optimal

Iterative Deepening Depth First Search (IDS/IDDFS)

The problem with depth limited search is deciding on a suitable depth parameter. If you look at the Romania map you will notice that there are 20 towns. Therefore, any town is reachable from any other town with a maximum path length of 19.
But, closer examination reveals that any town is reachable in at most 9 steps. Therefore, for this problem, the depth parameter should be set as 9. But, of course, this is not always obvious and choosing a parameter is one reason why depth limited searches are avoided.



To overcome this problem there is another search called iterative deepening search. This search method simply tries all possible depth limits; first 0, then 1, then 2 etc., until a solution is found. What iterative deepening search is doing is combining breadth first search and depth first search**.**

The time complexity for iterative deepening search is $O(b^d)$
The space complexity being O(bd).
******For large search spaces where the depth of the solution is not known iterative deepening search is normally the preferred search method.***

Checking for Repeated States

Check if the same state will be generated more than once. If we can avoid this then we can limit the number of nodes that are created and, more importantly, stop the need to have to expand the repeated nodes.

There are three methods we can use to stop generating repeated nodes.

1. Do not generate a node that is the same as the parent node. Or, to put it another way, do not return to the state you have just come from.
2. Do not create paths with cycles in them. To do this we can check each ancestor node and refuse to create a state that is the same as this set of nodes.
3. Do not generate any state that is the same as any state generated before. This requires that every state is kept in memory (meaning a potential space complexity of $O(b^d)$).

Summary:

This is a summary of the five search methods that we have looked at. In the following table the following symbols are used.

Criteria	BFS	UCS	DFS	DLS	IDDFS	Bidirectional
Frontier	Queue (FIFO)	Priority Queue(Order min. cost first)	Stack (LIFO)	Stack (LIFO)	Stack (LIFO)	
Completeness	YES [a]	YES [a,b]	Tree search-No (Cycle) Graph Search—Yes (finite) /No (infinite)	No	YES	YES
Optimality	YES*	YES [a,b]	No	No	No	YES
Time	$O(b^d)$	$O(b^{c^*/\varepsilon})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{c^*/\varepsilon})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$

b = Branching factor
d = Depth of solution
m = Maximum depth of the search tree
l = Depth limit

a - when b in finite
b – when every edge cost positive and ε > 0
ε- minimum cost between two nodes
C^*-is the cost of optimal solution

Remember:

Note 1: For all types of search problem by default Graph Search Algorithm is used by maintaining OPEN and CLOSED list. NEVER VISIT a node which has already been visited.

Note 2: When the problem specifically mentioned use Tree Search Algorithm, Then NO NEED to maintain CLOSED list only maintain OPEN list. In this case some nodes may be visited multiple times.

****
****

Practice Questions: Uninformed Search