# Uniform Cost Search

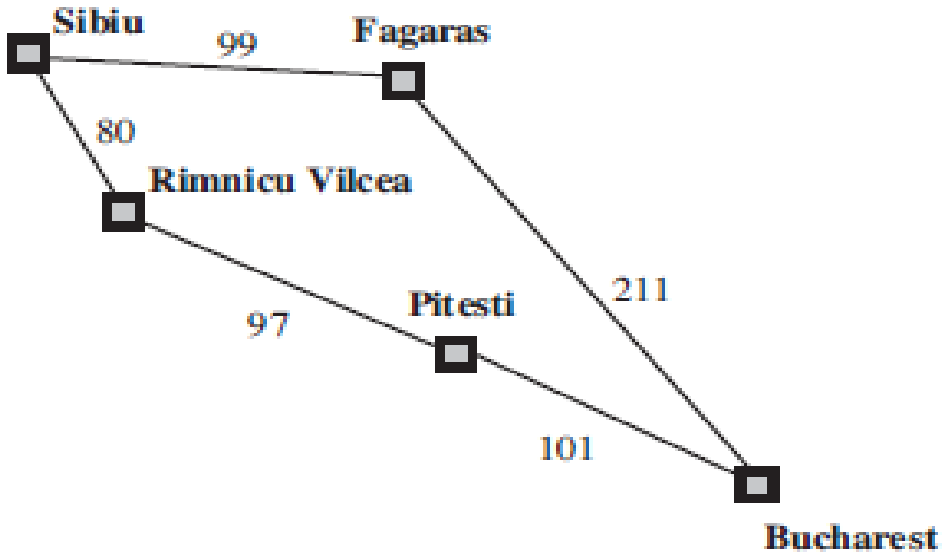
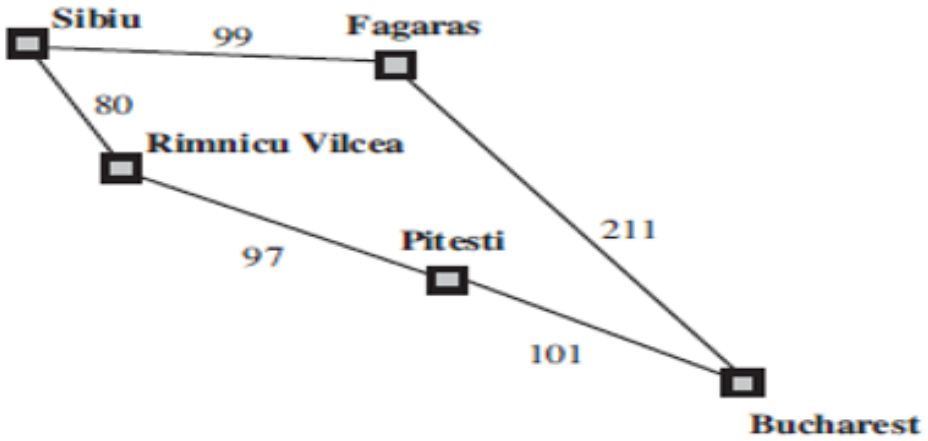



**Figure 3.15** Part of the Romania state space, selected to illustrate uniform-cost search.

Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost.

uniform-cost search does not care about the number of steps a path has,but only about their total cost.

| Fringe | Goal test | Close list | Comparison |
|---|---|---|---|
| **S** | S× | | |
| **R,F**<br>**80,99** | R× | S | S-R<S-F; 80<99; so select S-R |
| **P,F**<br>**80+97,99** | F× | S,R | S->R->P=177>S-F<br>177>99; goto F |
| **P,B**<br>**80+97,99+211** | P× | S,R,F | S->F->B=310><br>S->R->P=177; goto P<br>Though goal is reached still searches for least cost |
| **B,B**<br>**177+101,99+211** | B✓ | S,R,F,P | S->R->P->B=278 < S->F->B=310;<br>Goal reached in less cost |



```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

**Figure 3.14** Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure 3.7, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for *frontier* needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.

- Note that if all step costs are equal, this is identical to breadth-first search.
- Uniform-cost search does not care about the *number* of steps a path has, but total cost.
- Therefore, stuck in an infinite loop if it ever expands a node - zero-cost action leading back to the same state (for example, a *NoOp* action).
- **completeness** provided the cost of every step is greater than or equal to some small positive constant **c. ->**to ensure ***optimality.***
- It means that the cost of a path always increases as we go along the path. - algorithm expands nodes in order of increasing path cost.
- Eg: We recommend trying the algorithm out to find the shortest path to Bucharest.
- Uniform-cost search is guided by path costs rather than depths, so its complexity cannot easily be characterized in terms of *b* and d.
- Instead, let C* be the cost of the optimal solution, and assume that every action costs at least ***e.*** Then the algorithm's worst-case time and space complexity is $O(b^{1+[C^*/e]})$, which can be much greater than $b^d$.

Expand least-cost unexpanded node

**Implementation:**

$fringe$ = queue ordered by path cost, lowest first

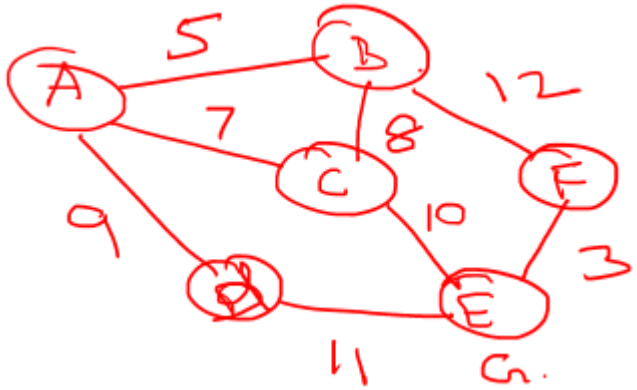
Equivalent to breadth-first if step costs all equal

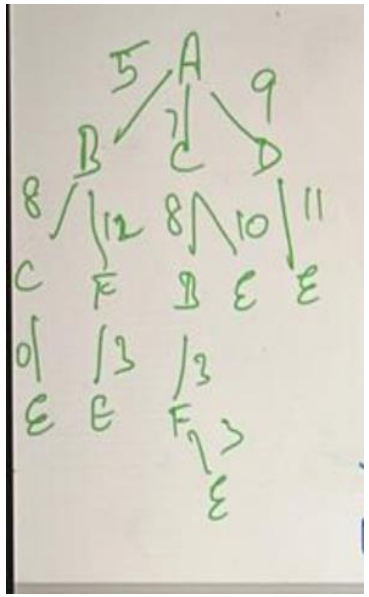
Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$




| fringe | G.T | Closed | Comp |
|---|---|---|---|
| A | A X | | |
| B, C, D<br>5, 7, 9 | B X | A | A-B < A-C < A-D<br>∴ goto A-B |
| C, D, C, F<br>7, 9, 5+8, 5+12 | C X | A, B | A-C < A-D < A-B-C<br>< A B-F<br>∴ goto A-C |
| D, C, F, B, E<br>9, 5+8, 5+12, 7+8, 7+10 | D X | A, B, C | A-D < A-B-C < A-C-B<br>goto A-D < A-B-F ≥ A-C-E |
| C, F, B, E, E<br>5+8, 5+12, 7+8, 7+10, 9+11 | | | A-B-C < A-C-B < A-B-F =<br>A-C-E < A-D-E<br>goto A-B-C |
| F, B, E, E, E<br>5+12, 7+8, 7+10, 9+11, 5+8+10 | | | A-C-B < A-B-F = A-C-E<br>< A-D-E < A-B-C-E<br>∴ goto A-C-B |
| F, E, E, E, F<br>5+12, 7+10, 9+11, 5+8+10, 7+8+3 | E V | | A-C-E is lesser than all path with 17 cost<br>This is to A-B-F. Though proceed with F, little bit cost will increase to stop |

# Depth First Search

- **-** always expands the *deepest* node in the current fringe of the search tree.
- -The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors.
- -As those nodes are expanded, they are dropped from the fringe, so then the search "backs up" to the next shallowest node that still has unexplored successors.
- - implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue, also known as a stack.

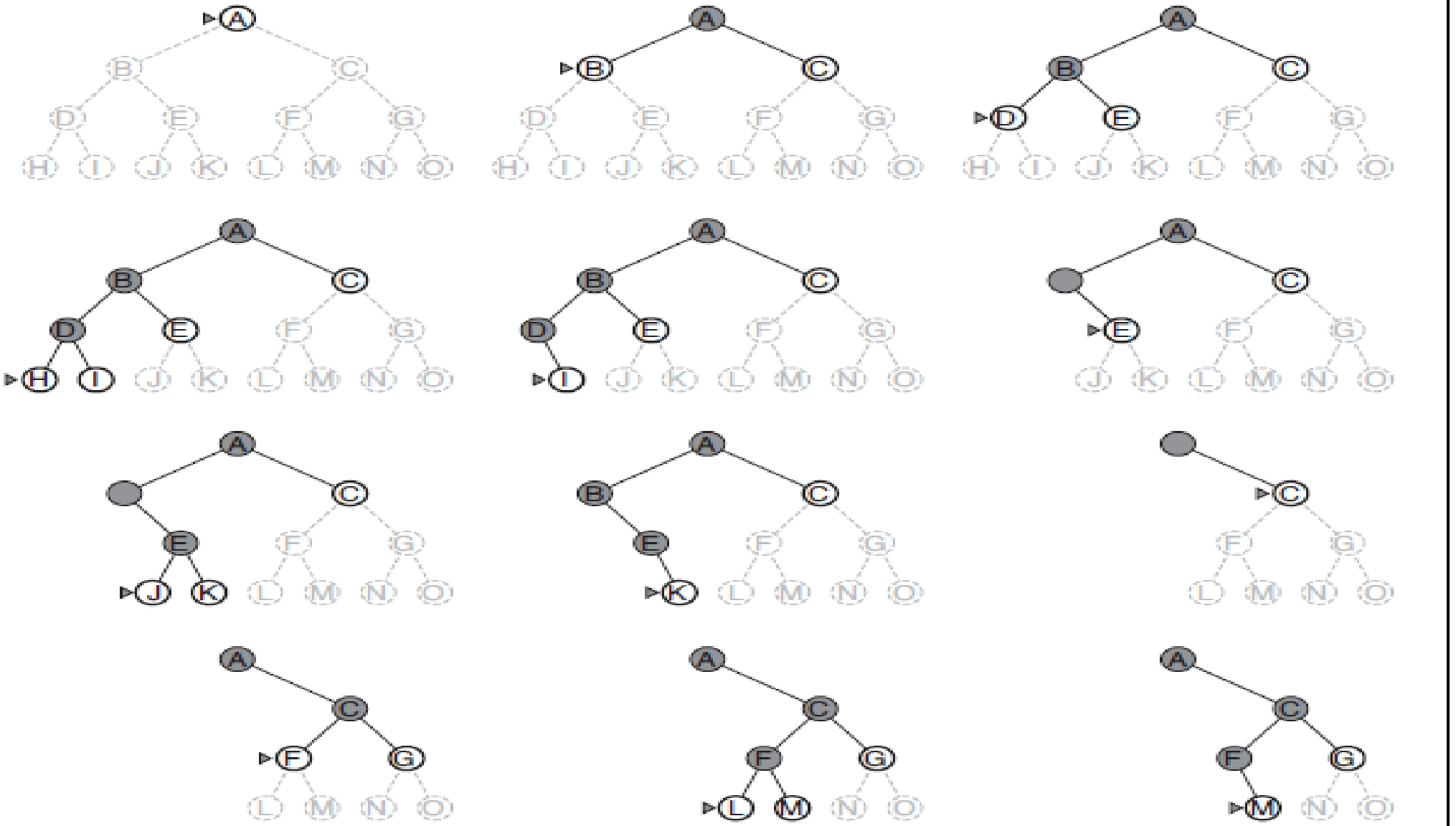
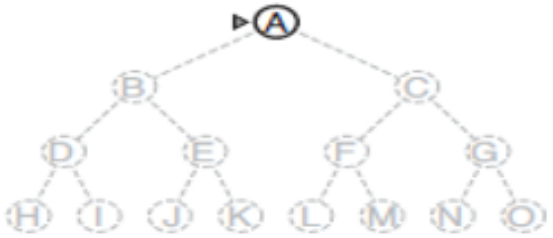
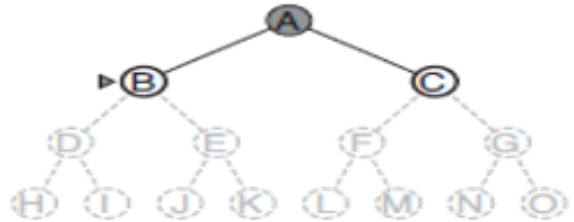
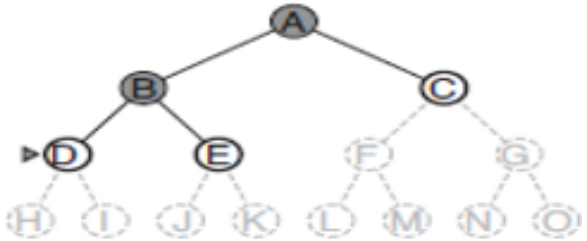
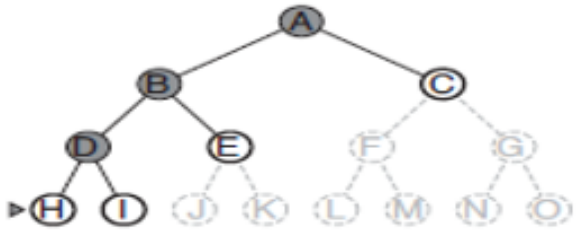



**Figure 3.16** Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and $M$ is the only goal node.

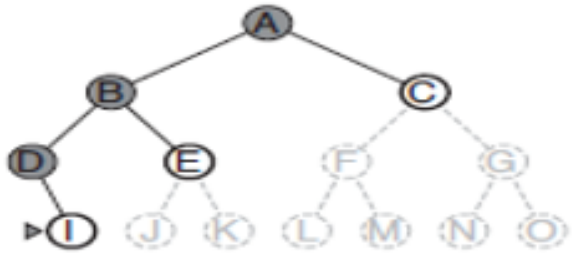


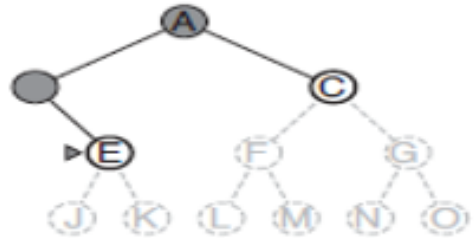



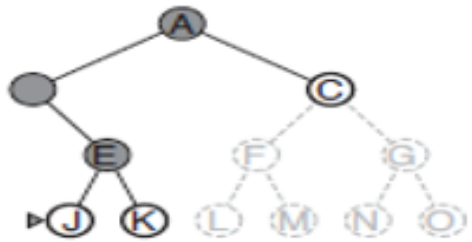



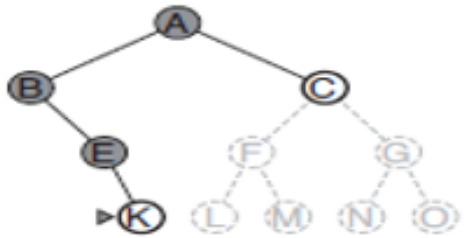




| Iteration | Fringe | Closed list (Visited) | Goal test | Removed nodes from memory | Back track to |
|---|---|---|---|---|---|
| 0 | A | | A x | | |
| 1 | C,B | A | Bx | | |
| 2 | C,E,D | A,B | Dx | | |
| 3 | C,E,I,H | A,B,D | Hx | H | D |

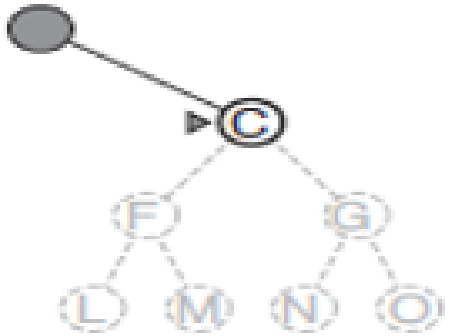


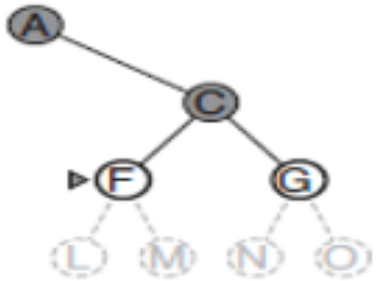



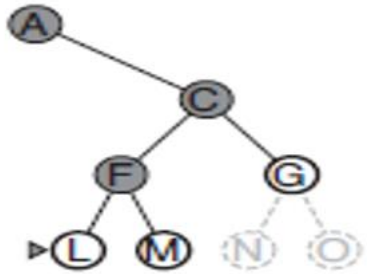



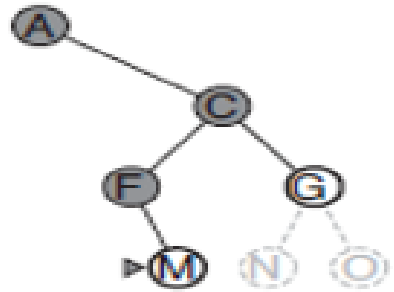




| Iteration | Fringe | Closed list (Visited) | Goal test | Removed nodes from memory | Back track to |
|---|---|---|---|---|---|
| 4 | C,E,I | A,B,D | Ix | I | D |
| 5 | C,E | A,B,D | Ex | D | B |
| 6 | C,K,J | A,B,E | Jx | J | E |
| 7 | C,K | A,B,E | Kx | K | E |



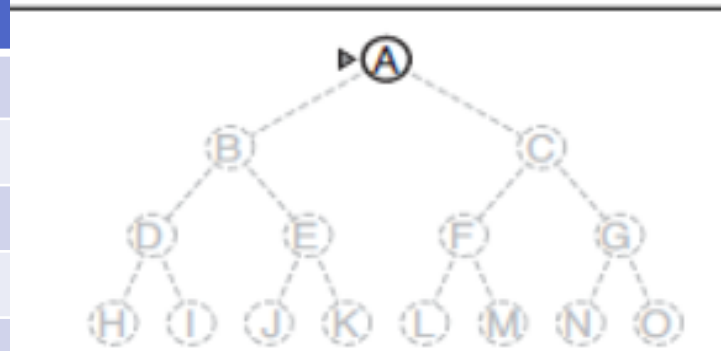




| Iteration | Fringe | Closed list (Visited) | Goal test | Removed nodes from memory | Back track to |
|---|---|---|---|---|---|
| | C | A,B | | E | B |
| | C | A | | B | A |
| 8 | C | A | Cx | | |
| 9 | G,F | A,C | Fx | | |
| 10 | G,M,L | A,C,F | Lx | L | F |
| | G | A,C,F | M- Goal reached ✓ | | |

**If 'M' is a goal :** LIFO queue

| Iteration | Fringe | Closed list (Visited) | Goal test | Removed nodes from memory | Back track to |
|---|---|---|---|---|---|
| 0 | A | | A × | | |
| 1 | C,B | A | B× | | |
| 2 | C,E,D | A,B | D× | | |
| 3 | C,E,I,H | A,B,D | H× | H | D |
| 4 | C,E,I | A,B,D | I× | I | D |
| 5 | C,E | A,B | E× | D | B |
| 6 | C,K,J | A,B,E | J× | J | E |
| 7 | C,K | A,B,E | K× | K | E |
| | C | A,B | | E | B |
| | C | A | | B | A |
| 8 | C | A | C× | | |
| 9 | G,F | A,C | F× | | |
| 10 | G,M,L | A,C,F | L× | L | F |
| | G | A,C,F | M- Goal reached ✓ | | |




**Now third column A-C-F set the path to Goal M.**

```
function Depth-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a LIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

- modest memory requirements. It needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path.
- Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.
- branching factor *b* and maximum depth *m,* - exploring nodes- **Time** $O(b^m)$
- *Memory capacity* ***O(bm)***

# The drawback –DFS

- can make a wrong choice and get stuck going down a very long (or even infinite) path when a different choice would lead to a solution near the root of the search tree.
- If node J were also a goal node, then depth-first search would return it as a solution; hence, depth-first search is **not optimal**.
- If the left subtree were of unbounded depth but contained no solutions, depth-first search would never terminate; hence, it is **not complete**.

# Depth-first search is appropriate when either..

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution;
- It is a poor method when it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or
- Solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions

# Compare BFS and DFS

| | | |
|---|---|---|
| **1.** | **BFS stands for Breadth First Search** | **DFS stands for Depth First Search.** |
| **2.** | FIFO Queue for traversal and to find goal | Uses LIFO, Stack for traversal and to find goal |
| **3.** | more suitable for searching vertices which are closer to the given source. | more suitable when there are solutions away from source. |
| **4.** | BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop. |

# Compare BFS and DFS

| | | |
|---|---|---|
| **5.** | **Here, siblings are visited before the children** | **Here, children are visited before the siblings** |
| **6.** | The Time complexity of BFS is $O(b^{d+1})$ | The Time complexity of DFS is $O(b^{m})$ |
| **7.** | Space Complexity is $O(b^{d+1})$<br>BFS has to keep track of all the nodes on the same level | Space Complexity is O(bm)<br>DFS needs less memory as it only has to keep track of the nodes in a chain from the top to the bottom |
| **8.** | Complete if branching factor is finite | DFS **can get stuck in an infinite loop**, which is why it is not "complete". Graph search keeps track of the nodes it has already searched, so it can avoid following infinite loops. "Redundant paths" are different paths which lead from the same start node to the same end node.. |
| **9.** | Optimal , if step cost are constant | **DFS is not optimal**, -the number of steps in reaching the solution, or the cost spent in reaching it is high. |