**Session-4**

**Uninformed Search Stratagies**

**Uninformed Search (**B**lind Search**), strategies have no additional information about states beyond that provided in the problem definition. All they can do is generate successors and distinguish a goal state from a non-goal state. All search strategies are distinguished by the *order* in which nodes are expanded.

Following are the various types of uninformed search algorithms:

1. Breadth-first Search
2. Uniform cost search
3. Depth-first Search
4. Depth-limited Search
5. Iterative deepening depth-first search
6. Bidirectional Search
   1. **Breadth-First Search(BFS):**

**Breadth-first search** is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then *their* successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search is an instance of the general graph-search algorithm in which the *shallowest* unexpanded node is chosen for expansion. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion.

**Algorithm:**

1. Create a variable called NODE-LIST(Frontier/Stack) and set it to initial state
2. Until a goal state is found or NODE-LIST is empty do
   * 1. Remove the first element ‘n’ from NODE-LIST and keep it in Explorer. If NODE-LIST was empty, quit
     2. For each way that each rule can match the state described in Explorer do:
        1. Apply the rule to generate a new state
        2. If the new state is a goal state, exit with the solution obtained by tracing back through the pointers
        3. Otherwise, add the new state to the end of NODE-LIST and provide pointer back to ‘n’ iff it is not in NODE-LIST and Explorer.

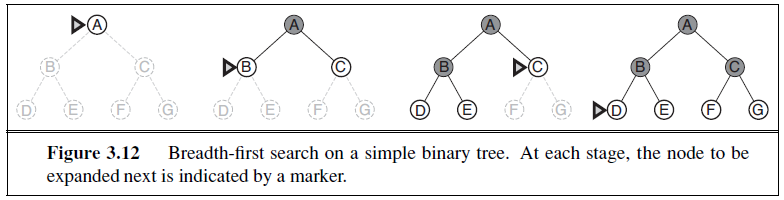
**Eg.** Apply BFS to find path from Arad to Bucharest.

1. Initially we keep source Arad into the **Frontier** (Queue) and **Explorer** as Empty.
2. Now, expand Arad so that add its successors Sibiu, Timisoara, Zerind which does not match our goal state and Explorer contains Arad.
3. Now we expand Sibiu, so that Frontier contains Timisoara, Zerind, Fagaras, Rimnicu Vilcea, Oradea and Explorer contains Arad, Sibiu. Here as Arad is present in Explorer we skip to add Arad in Frontier even there is path from Sibiu to Arad.
4. Now we expand Timisoara, so that Frontier contains Zerind, Fagaras, Rimnicu Vilcea, Oradea, Lugoj and Explorer contains Arad, Sibiu, Timisoara.
5. Now we expand Zerind, so that Frontier contains Fagaras, Rimnicu Vilcea, Oradea, Lugoj and Explorer contains Arad, Sibiu, Timisoara, Zerind
6. Now we expand Fagaras, since Bucharest is successor of Fagaras that match with our goal state we halt iteration and return solution.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  | |

**Figure:** Illustrating BFS Example.

Eg. 2:



**Advantages:**

1. BFS will provide a solution if any solution exists.
2. If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

**Disadvantages:**

1. It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
2. BFS needs lots of time if the solution is far away from the root node.

**Performance of BFS:**

**Completeness:** BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

**Optimality:** BFS is optimal if path cost is a non-decreasing function of the depth of the node.

**Time Complexity:** Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

**T (b) = 1+b2+b3+.......+ bd= O (bd)**

**Space Complexity:** Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

**NOTE:**

Time and space complexity are measured in terms of

b – branch factor which is equal to the number of successors of a state.

d – It is the depth at which solution found.

m – It is the maximum path occur in the problem state space.

1. **Uniform Cost Search (UCS):**

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs from the root node. It can be used to solve any graph/tree where the optimal cost is in demand.

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node. By a simple extension, we can find an algorithm that is optimal with any step-cost function. Instead of expanding the shallowest node, **uniform-cost search** expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g.

**Algorithm:**

1. Initialize: set OPEN = {s}, CLOSED = { }, set C(s) = 0

2. If OPEN is empty, terminate search and return failure

3. Select a minimum cost state, n, from OPEN and save n in CLOSED

4. If n is a Goal state, terminate with Success

5. Generate the successor of n

For each successor, m,

if m is not in OPEN or CLOSED :

Set C(m) = C(n) + C(n, m) and insert m in OPEN

if m is in OPEN or CLOSED:

Set C(m) = min{ C(m), C(n) + C(n, m)} and insert m in OPEN

if C(m) has decreased and m is in CLOSED, move it to OPEN

6. Loop: Go back to step 2

**Eg.** Apply uniform cost search algorithm to reach Bucharest from Sibiu in Romania.

1. Initially we keep source Sibiu into the **Open** (Priority Queue) and **Closed** as Empty and C(Sibiu) = 0.
2. Select Sibiu from Open, expand and save in CLOSED. Since Sibiu is not Goal state its successors are placed in OPEN as Fagaras with C(Fagaras)=99 and Rimnicu Vilcea with C(Rimnicu Vilcea)=80 as both are not in OPEN or CLOSED.
3. Now we select Rimnicu Vilcea as minimum cost state, expand and save in Closed. Since Rimnicu Vilcea is not Goal state its successor Piteste is placed in OPEN that contains C(Fagaras)=99 and C(Piteste)=177.
4. Now we select Fagaras as minimum cost state, expand and save in Closed. Since Fagaras is not Goal state its successor Bucharest is placed in OPEN that contains C(Piteste)=177 and C(Bucharest)=310.
5. Now we select as minimum cost state, expand and save in Closed. Since Piteste is not Goal state its successor Bucharest which is already in OPEN with lesser cost C(Bucharest)=278 update C(Bucharest)=310 to C(Bucharest)=278 and save in OPEN.
6. Now we select Bucharest as minimum cost state, expand and save in Closed. Since it is our Goal state, terminate search with success.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

**Figure:** Illustrating Uniform Cost Search example.

**Difference between BFS and UCF:**

1. The goal test is applied to a node when it is selected for expansion rather than when it is first generated. Because the first goal node that is generated may be on a suboptimal path and
2. A test is added in case a better path is found to a node currently on the frontier.

**Advantages:**

1. Uniform cost search is optimal because at every state the path with the least cost is chosen.

**Disadvantages:**

* 1. It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

**Performance of UCS:**

**Completeness:**

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

**Time Complexity:**

Let C\* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C\*/ε+1. Here we have taken +1, as we start from state 0 and end to C\*/ε.

Hence, the worst-case time complexity of Uniform-cost search is**O(b1 + [C\*/ε])/**.

**Space Complexity:**

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **O(b1 + [C\*/ε])**.

**Optimal:**

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

1. **Depth First Search (DFS):** Depth-first searchalways expands the deepest node in the current frontier 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 frontier, so then the search “backs up” to the next deepest node that still has unexplored successors.

The depth-first search algorithm is an instance of the graph-search algorithm. Depth-first search uses a LIFO queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected. As an alternative to the GRAPH-SEARCH-style implementation, it is common to implement depth-first search with a recursive function that calls itself on each of its children in turn.

**Algorithm:**

* 1. Create a variable called NODE-LIST(Frontier/Stack) and set it to initial state

1. Until a goal state is found or NODE-LIST is empty do
   * 1. Remove the first element ‘n’ from NODE-LIST and keep it in Explorer. If NODE-LIST was empty, quit
     2. For each way that each rule can match the state described in Explorer do:
        1. Apply the rule to generate a new state
        2. If the new state is a goal state, exit with the solution obtained by tracing back through the pointers
        3. Otherwise, add the new state to the beginning of NODE-LIST and provide pointer back to ‘n’ iff it is not in NODE-LIST and Explorer.

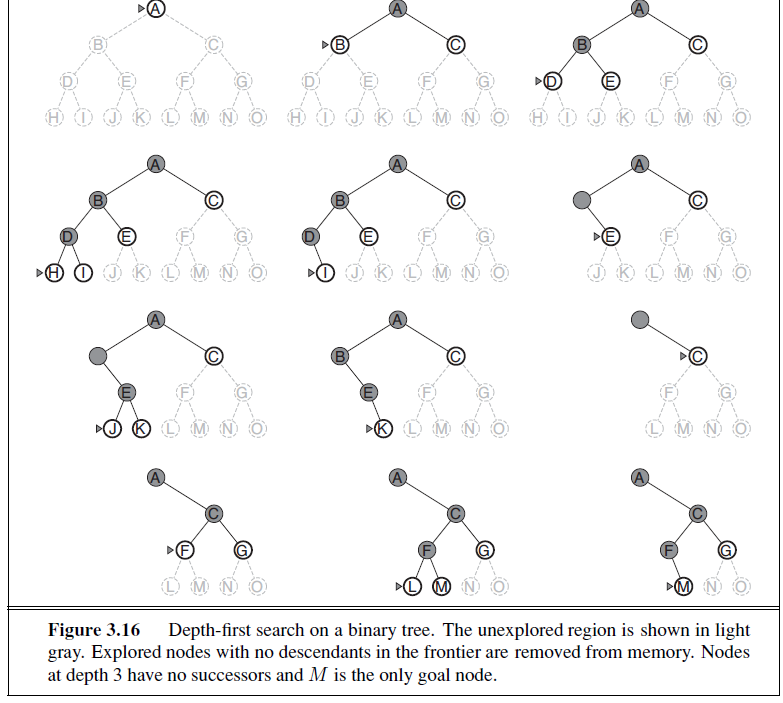
**Eg.** Apply DFS to find path from Arad to Bucharest.

1. Initially we keep source Arad into the **Frontier** (Stack) and **Explorer** as Empty.
2. Now, expand Arad so that add its successors Sibiu, Timisoara, Zerind which does not match our goal state and Explorer contains Arad.
3. Now we expand Sibiu, so that Frontier contains Fagaras, Rimnicu Vilcea, Oradea, Timisoara, Zerind and Explorer contains Arad, Sibiu. Here as Arad is present in Explorer we skip to add Arad in Frontier even there is path from Sibiu to Arad.
4. Now we expand Fagaras, since Bucharest is successor of Fagaras that match with our goal state we halt iteration and return solution.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  | |

**Figure:** Illustrating DFS example.

Eg.2:



A variant of depth-first search called **backtracking** searchuses still less memory. In backtracking, only one successor is generated at a time rather than all successors; each partially expanded node remembers which successor to generate next. In this way, only O(m) memory is needed rather than O(bm). Backtracking search facilitates yet another memory-saving (and time-saving) trick: the idea of generating a successor by modifying the current state description directly rather than copying it first. This reduces the memory requirements to just one state description and O(m) actions. For this to work, we must be able to undo each modification when we go back to generate the next successor. For problems with large state descriptions, such as robotic assembly, these techniques are critical to success.

**Advantage:**

1. DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
2. It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

**Disadvantage:**

1. There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
2. DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

**Performance of DFS:**

**Completeness:** DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

**Optimal:** DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

**Time Complexity:** Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

**T(n)= 1+ n2+ n3 +.........+ nm=O(nm)**

**Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)**

**Space Complexity:** DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

1. **Depth-Limited Search (DLS):** The embarrassing failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit ‘ℓ’. That is, nodes at depth are treated as if they have no successors. This approach is called **depth-limited search**.

The depth limit solves the infinite-path problem. Unfortunately, it also introduces an additional source of incompleteness if we choose ℓ < d, that is, the shallowest goal is beyond the depth limit likely when d is unknown.

Depth-limited search will also be non-optimal if we choose ℓ > d. Its time complexity is O(bℓ) and its space complexity is O(b\* ℓ). Depth-first search can be viewed as a special case of depth-limited search with ℓ =∞.

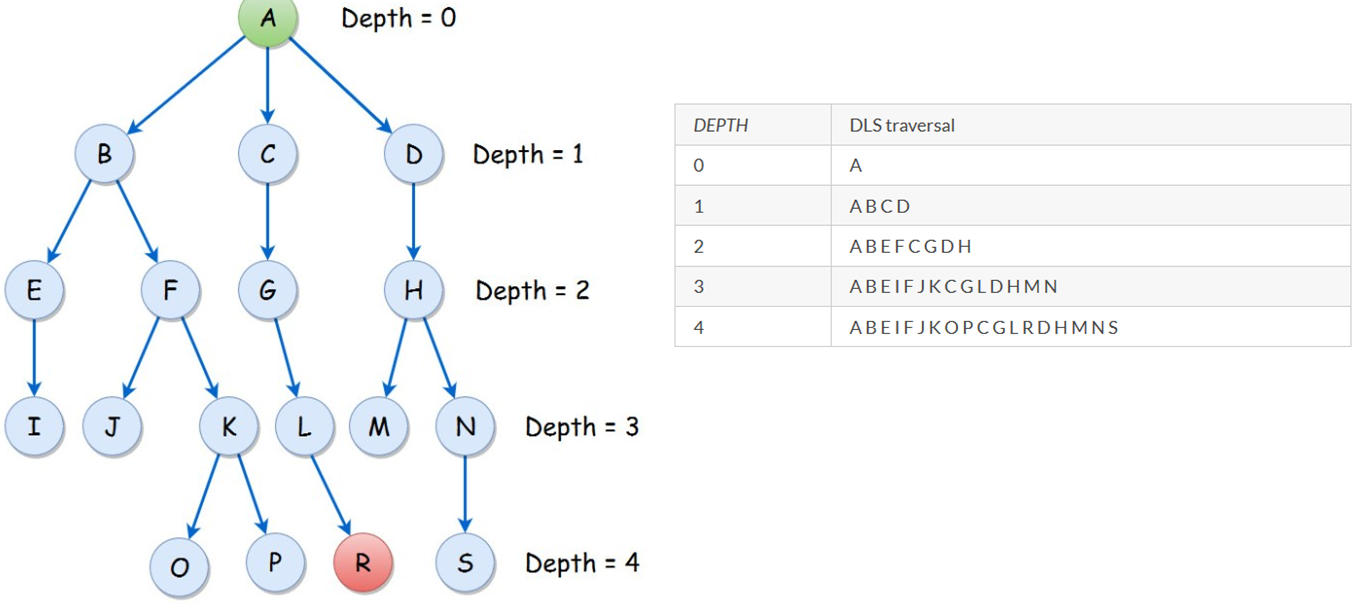
Sometimes, depth limits can be based on knowledge of the problem. For example, on the map of Romania there are 20 cities. Therefore, we know that if there is a solution, it must be of length 19 at the longest, so ℓ = 19 is a possible choice. But in fact if we studied the map carefully, we would discover that any city can be reached from any other city in at most 9 steps. This number, known as the **diameter** of the state space, gives us a better depth limit, which leads to a more efficient depth-limited search. For most problems, however, we will not know a good depth limit until we have solved the problem.

Depth-limited search can be implemented as a simple modification to the general tree or graph-search algorithm.

**Algorithm:**

1. Create a variable called NODE-LIST(Frontier/Stack) and set it to initial state
2. Until a goal state is found or NODE-LIST is empty or cut\_of\_limit occurs do
   * 1. Remove the first element ‘n’ from NODE-LIST and keep it in Explorer. If NODE-LIST was empty, quit
     2. For each way that each rule can match the state described in Explorer do:
        1. Apply the rule to generate a new state
        2. If the new state is a goal state, exit with the solution obtained by tracing back through the pointers
        3. Otherwise, add the new state to the beginning of NODE-LIST and provide pointer back to ‘n’ iff it is not in NODE-LIST and Explorer

**Eg.**



**Advantages:**

* 1. Depth-limited search is Memory efficient.

**Disadvantages:**

1. Depth-limited search also has a disadvantage of incompleteness.
2. It may not be optimal if the problem has more than one solution.

**Performance of DLS:**

**Completeness:** DLS search algorithm is complete if the solution is above the depth-limit.

**Time Complexity:** Time complexity of DLS algorithm is **O(bℓ)**.

**Space Complexity:** Space complexity of DLS algorithm is O**(bℓ)**.

**Optimal:** Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.