CS249 – ARTIFICIAL INTELLIGENCE - 1

3rd Week Slide/Notes

- 2201AI12 Hari Om Kumar
- 2201AI13 Harpranav Singh Uppal
- 2201AI14 Harsh Kumar
- 2201AI15 Harshit Tomar
- 2201AI16 Himani Yadav

PROLEM SOLVING BY SEARCH

The Farmer, Wolf, Goat, and Cabbage puzzle is a simple one player “game” in which the player attempts to determine how a farmer can transport a wolf, a goat, and a cabbage across a river using a small boat that can only hold the farmer and one other object. What makes the puzzle somewhat challenging is that if the farmer leaves either the wolf alone with the goat or the goat alone with the cabbage, one will be eaten by the other. If the farmer is present, then he will keep the participants from eating one another.

State description:

(<side for farmer>, <side for wolf>, <side for goat>, <side for cabbage>)

Initial state:

(w, w, w, w) – All participants begin on the west bank of the river.

Final (goal) state:

(e, e, e, e) – All participants end on the east bank of the river.

Loss (dead) states:

(e, w, w, e) – The wolf eats the goat.

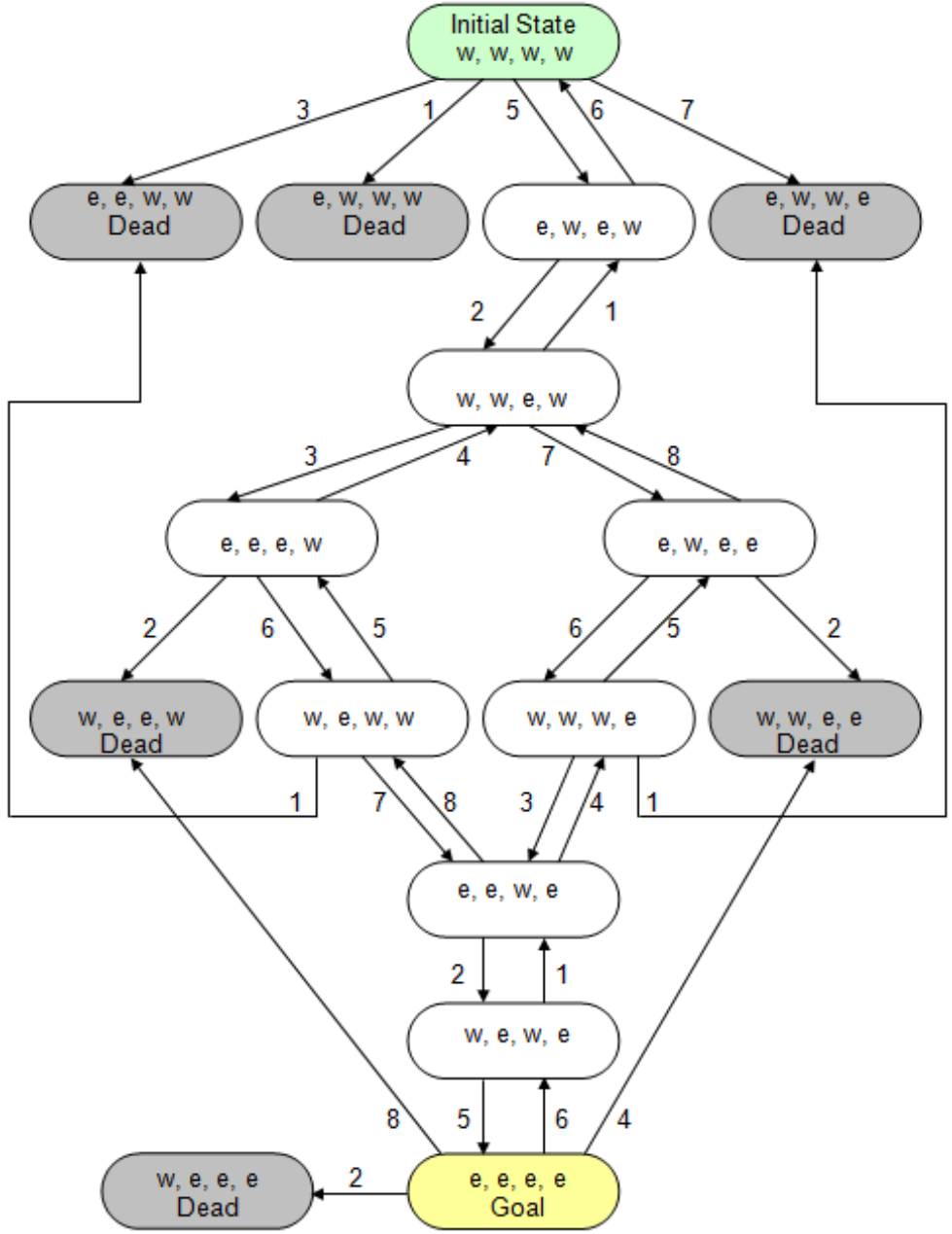
(w, e, e, w) – The wolf eats the goat.

(e, e, w, w) – The goat eats the cabbage.

(w, w, e, e) – The goat eats the cabbage.

(e, w, w, w) – The goat eats the cabbage, and the wolf eats the goat.

(w, e, e, e) – The goat eats the cabbage, and the wolf eats the goat.



Transitions:

- (w, LocWolf, LocGoat, LocCabbage) ⇒ (e, LocWolf, LocGoat, LocCabbage)
- (e, LocWolf, LocGoat, LocCabbage) ⇒ (w, LocWolf, LocGoat, LocCabbage)
- (w, w, LocGoat, LocCabbage) ⇒ (e, e, LocGoat, LocCabbage)
- (e, e, LocGoat, LocCabbage) ⇒ (w, w, LocGoat, LocCabbage)
- (w, LocWolf, w, LocCabbage) ⇒ (e, LocWolf, e, LocCabbage)
- (e, LocWolf, e, LocCabbage) ⇒ (w, LocWolf, w, LocCabbage)
- (w, LocWolf, LocGoat, w) ⇒ (e, LocWolf, LocGoat, e)
- (e, LocWolf, LocGoat, e) ⇒ (w, LocWolf, LocGoat, w)

SEARCHING STATE SPACE

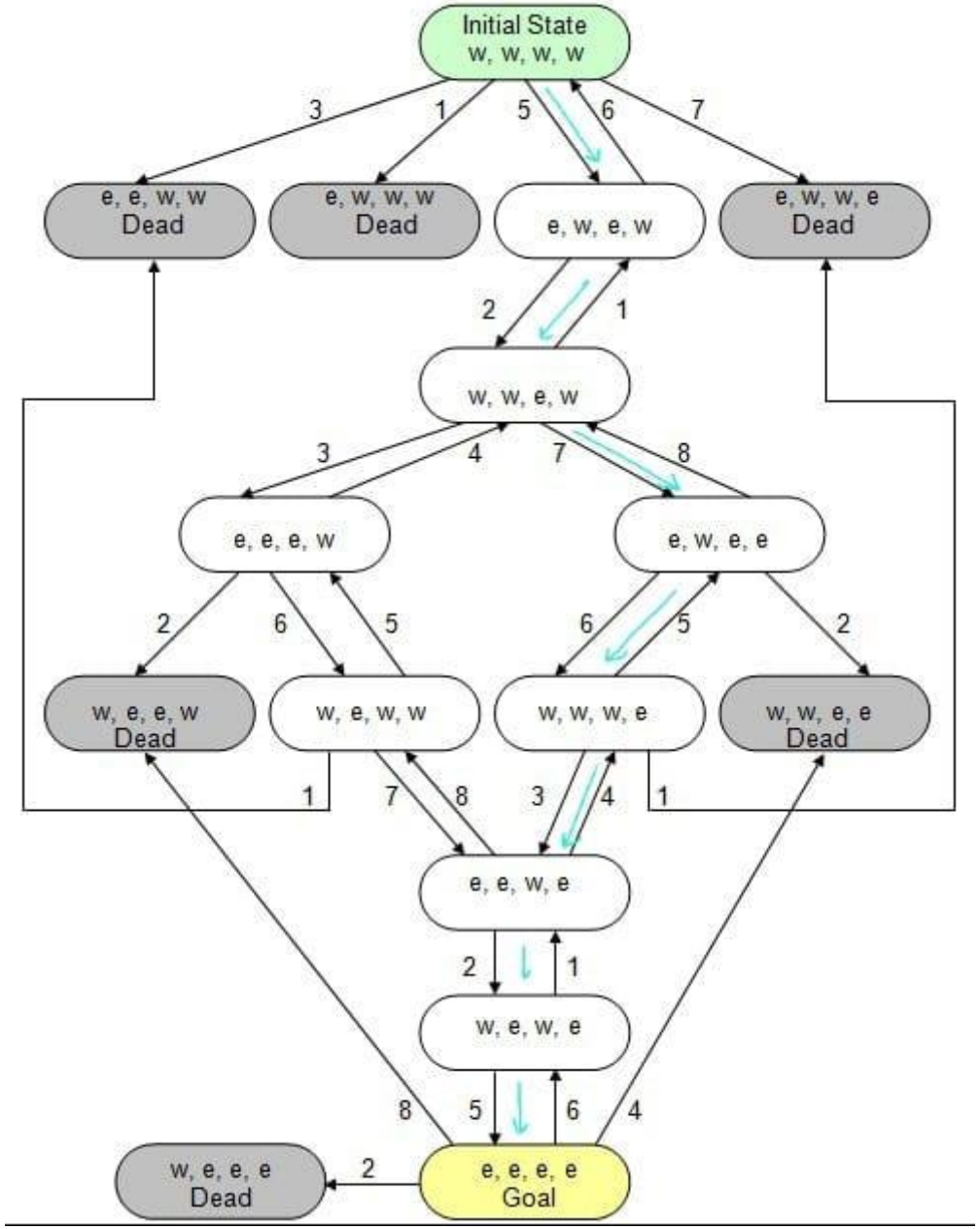
There are two common procedures for searching a state space. These are breadth-first search and depth-first search. Beginning at the initial state, **breadth-first** search considers all paths of length one, then all paths of length two, then three, etc., until it encounters the goal. The breadth-first strategy can be thought of as exploring all paths of equal length "in parallel". **Depth-first** search, on the other hand, will explore a single path, corresponding to a single line of reasoning, or chain of events, until either the goal is reached, or failure is certain. Only when it can determine that a path leads to failure does the algorithm back up and try an alternate route.

Figure on next slide illustrates how breadth-first and depth-first searches of a sample state space would be conducted. The state space graph begins at an initial state from which three valid transitions could be applied. The left and right transitions lead to states that each support two additional transitions. The middle transition from the initial state leads to a state that supports three more transitions. All paths of length two lead to dead states, with the exception of the rightmost path, which leads to the goal.

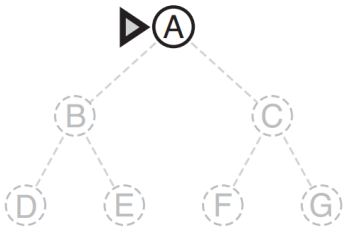
ONE OF THE SOLUTION:

s = (w, w, w, w)
-> (e, w, e, w)
-> (w, w, e, w)
-> (e, w, e, e)
-> (w, w, w, e)
-> (e, e, w, e)
-> (w, e, w, e)
-> (e, e, e, e)

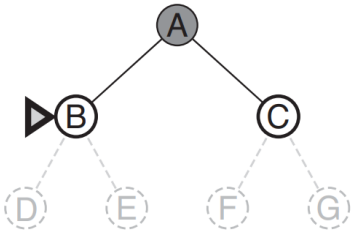
Total 7 transitions needed.



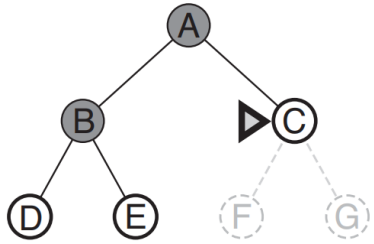
BREADTH FIRST SEARCH



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.

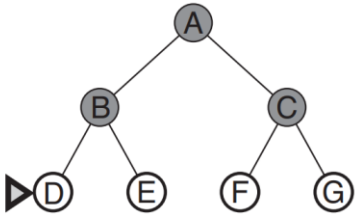


- Uses Queue (FIFO) as the main Data Structure
- Optimal and Complete Algorithm
- Time Complexity – $O(b^d)$ because it is an Uninformed Search Algorithm
- Space Complexity – $O(b^d)$ as it keeps a record of all its brother nodes across the whole tree

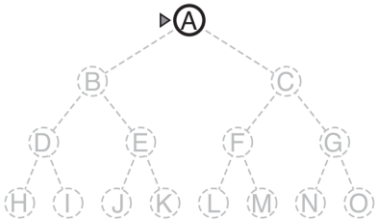


Here b is the branching factor of the state space

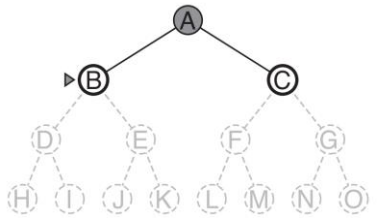
d is the minimum depth of a goal state



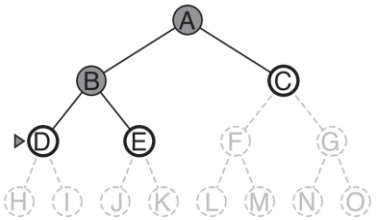
DEPTH FIRST SEARCH



Depth-first search algorithm expands the root-node first, then the first successor of the root node is expanded next, then its first successor, and so on. In other words, a node and all its successors are expanded to the maximum depth in the search tree before the expansion of its brother node.

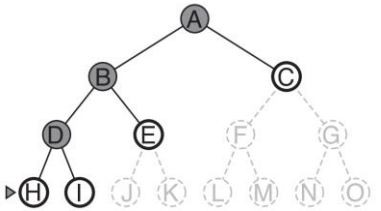


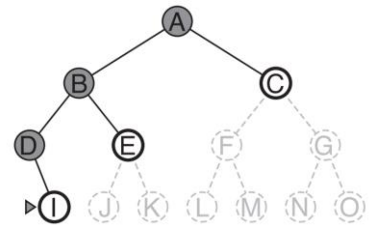
- Uses Stack (LIFO) as the main Data Structure
- Not-Optimal and Incomplete Algorithm
- Time Complexity – $O(b^m)$ as it is an Uninformed Search Algorithm
- Space Complexity – O(bm) because for m nodes down the path, we need to store b nodes extra for each of the m nodes.



Here b is the maximum branching factor of the search tree

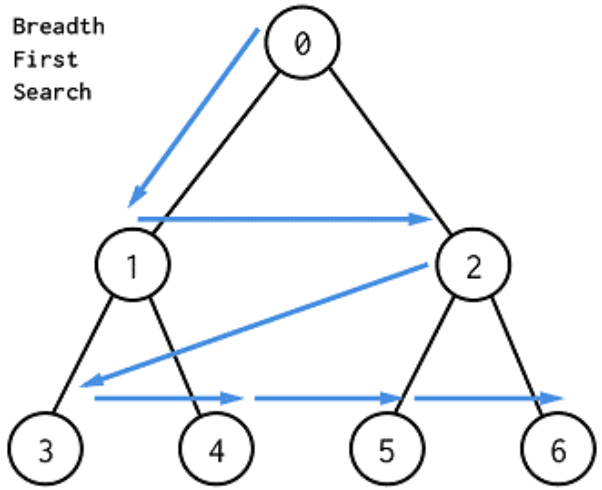
m is the maximum depth of the state space

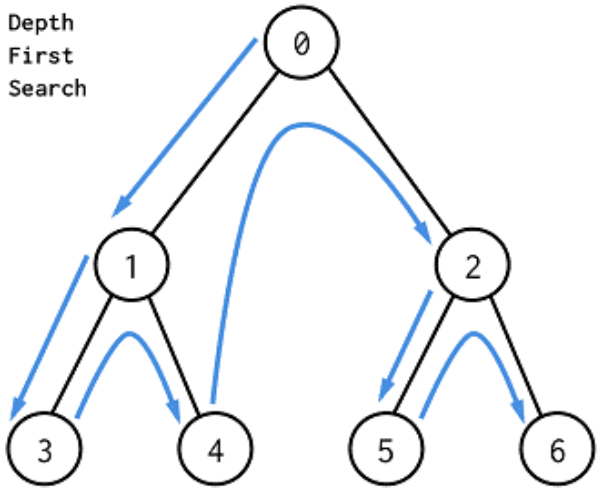




BFS vs DFS

	Breath-first Search	Depth-first Search
Time Complexity	$O(b^d)$	$O(b^d)$
Space Complexity	$O(b^m)$	O(bm)
Optimality	Yes	No
Completeness	Yes	No



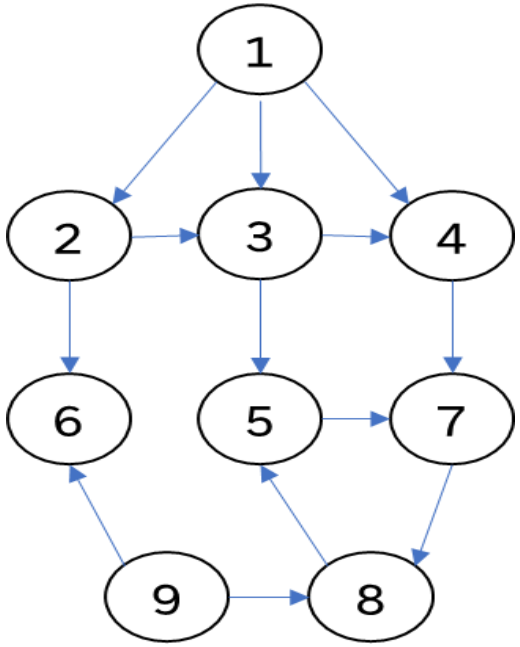


Source-Goal using DFS

Stack-Representation

				7
			4	4
		3	3	3
	2	2	2	2
1	1	1	1	1

Source **Goal**



Adjacency list:

1: {2,3,4}
2: {3,6}
3: {4,5}
4: {7}
5: {7}
6: { }
7: {8}
8: {5}
9: {6,8}

Source-Goal using BFS

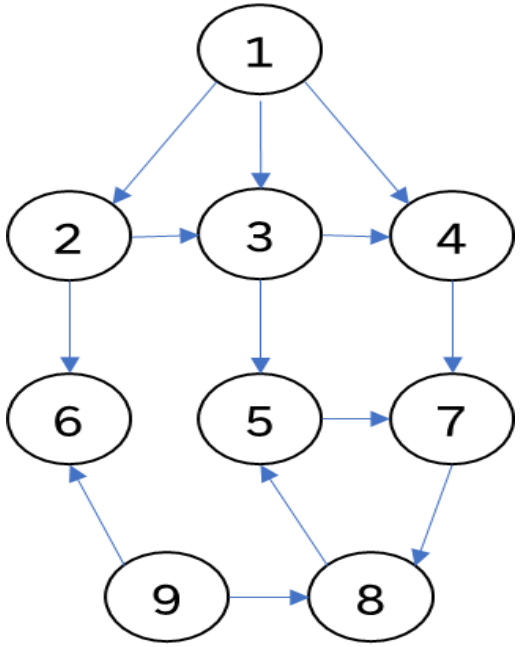
Queue-Representation

S

1				
2	3	4		
3	4	6		
4	6	5		

G

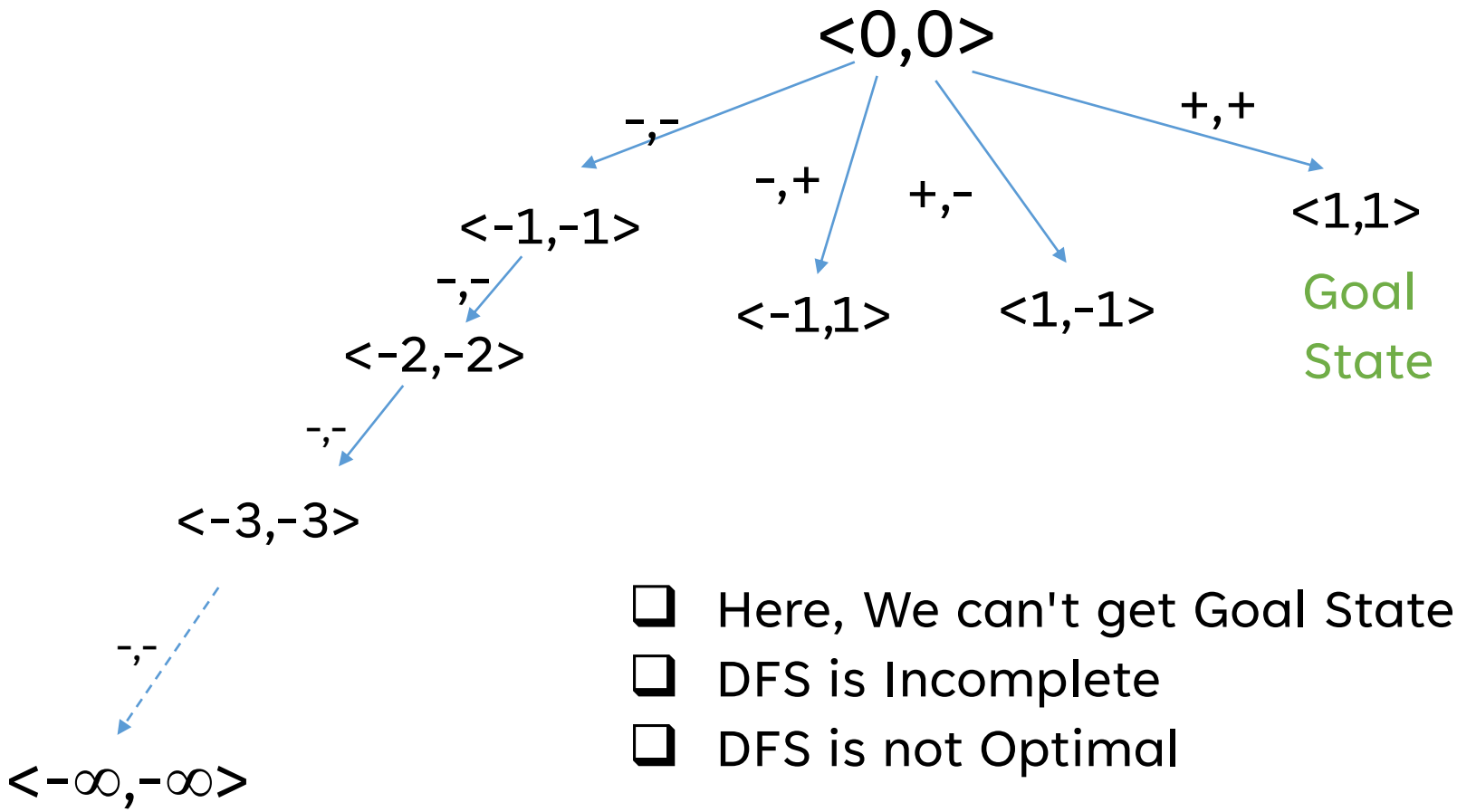
6	5	7		
5	7			
7				



Adjacency list:

1: {2,3,4}
2: {3,6}
3: {4,5}
4: {7}
5: {7}
6: { }
7: {8}
8: {5}
9: {6,8}

State Space may be Infinite



- Here, We can't get Goal State using DFS
- DFS is Incomplete
- DFS is not Optimal

COMPLETENESS

Completeness refers to the algorithm's ability to find a solution if one exists, ensuring that the algorithm does not overlook or miss any valid solutions within the problem space.

or in other words

Given a starting state and that a goal state exists, A Complete Algorithm will always reach the goal

DEPTH LIMITED DFS (DDFS)

Adding on to Depth-first search algorithm, We apply a limit to the maximum depth the search algorithm goes to avoid the failure of DFS in case of infinite deep state space tree.

- Uses Stack (LIFO) as the main Data Structure
- Not-Optimal Algorithm
- Time Complexity – $O(b^l)$
- Space Complexity – O(bl)

Here b is the maximum branching factor of the search tree

l is the chosen depth limit

- It is still an Incomplete Algorithm since for l < d, where d is the minimum depth of a goal state, it will not be able to find any goal state.

ITERATIVELY DEEPENING DFS (IDDFS)

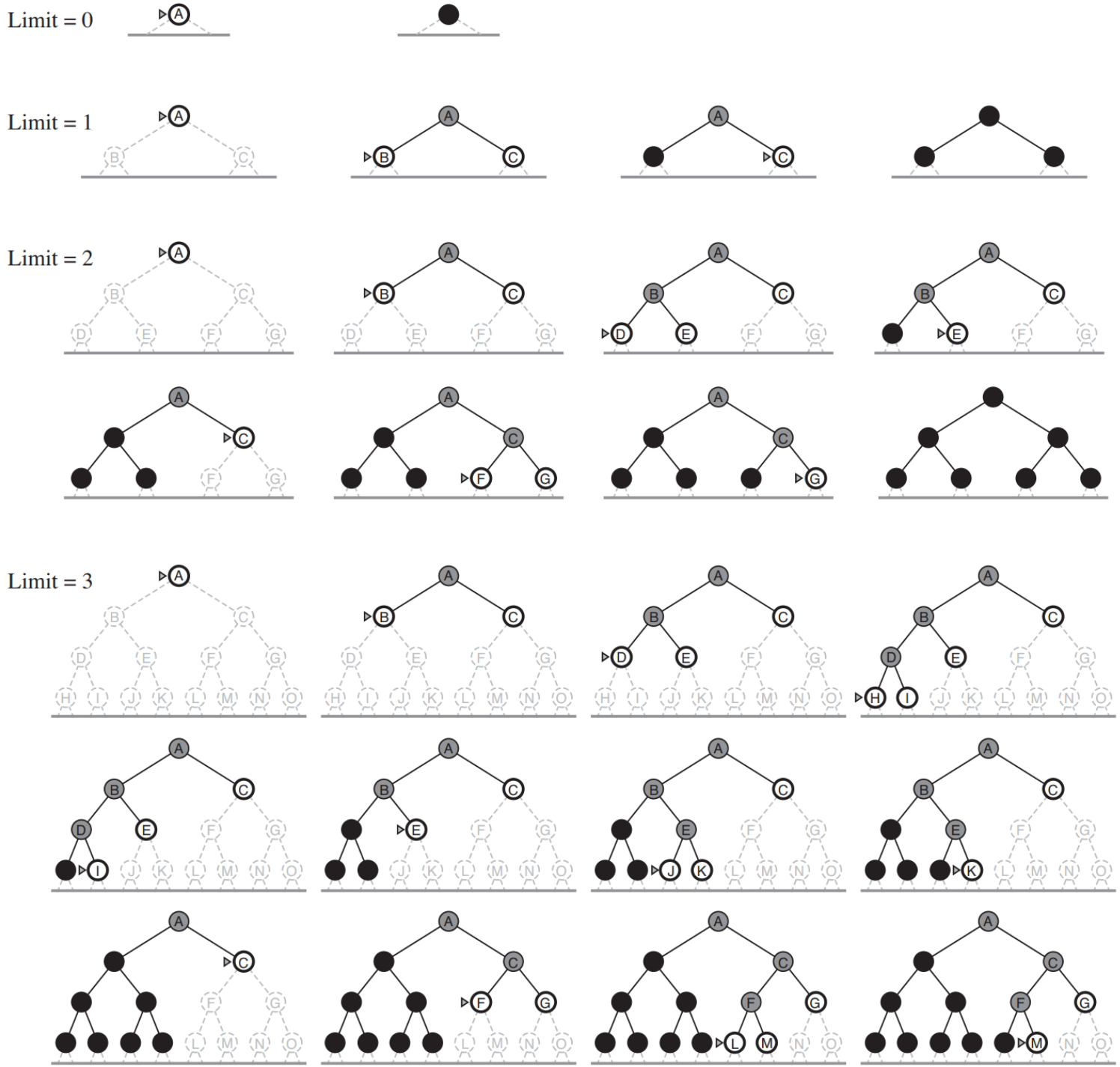
Modifying Depth Limited Search, we 'iteratively' increment the depth-limit to find a goal state. It is used to find the best depth-limit. This combines the benefits of both DFS and BFS.

- Like depth-first search, its Space Complexity is – O(bd)
- Like breadth-first search, it is:
  - **Complete** when the branching factor is finite
  - **Optimal** when the path cost is a nondecreasing function of the depth of the node.
- Time Complexity – $O(b^d)$

Here b is the maximum branching factor of the search tree

d is the depth of the shallowest goal state

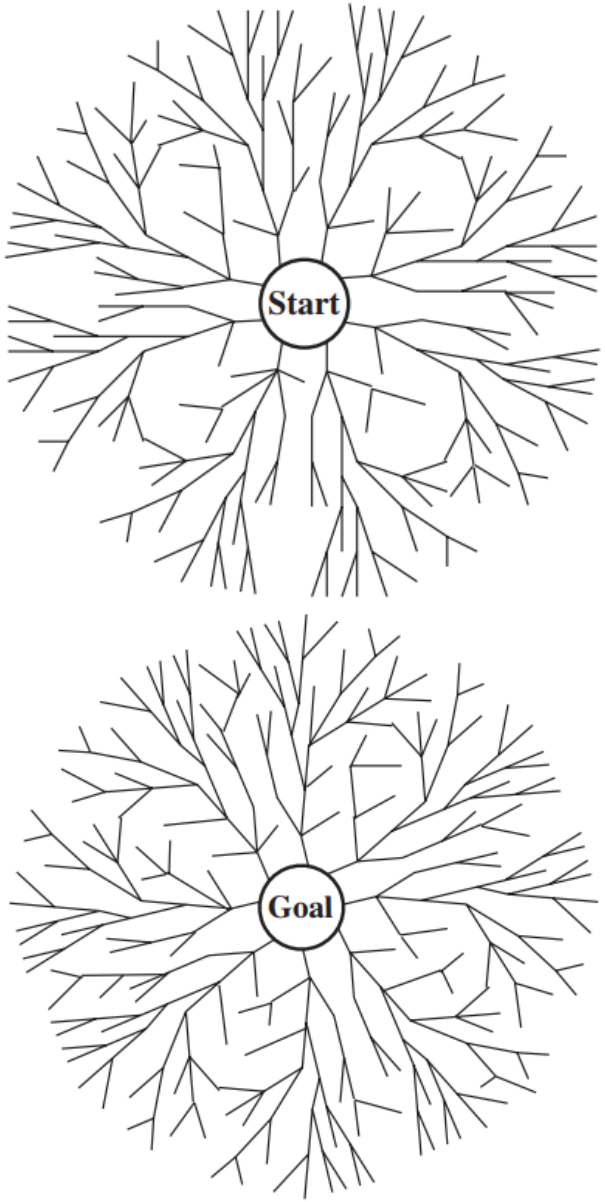
➢ Disadvantage :- IDDFS repeats all the work of the previous phase


Limit = 0
Limit = 1
Limit = 2
Limit = 3

BIDIRECTIONAL SEARCH

Bidirectional search explores a solution from both the start and end points at the same time to meet in the middle, making it potentially more efficient.

- Optimal and Complete Algorithm
- Time Complexity – $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$
- Space Complexity – depends on the base algorithm [BFS – $O(b^{d/2})$]
- The actual algorithm used can be any of the other discussed like BFS, DFS
- Bidirectional Search is only applicable only when the goal state is known, and the path is the unknown



COMPARISON

	Depth Limited DFS	Iterative Deepening DFS	Bidirectional BFS (If Possible)
Time Complexity	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity	$O(b^d)$	$O(bd)$	$O(b^{d/2})$
Optimality	No	Yes	Yes
Completeness	No	Yes	Yes

UNIFORM COST SEARCH

- It is a Variant of Dijkstra's Algorithm.
- Here, instead of inserting all vertices into a priority queue, we insert only the source, then one by one insert when needed.
- In every step, we check if the item is already in the priority queue (using the visited array). If yes, we perform the decrease key, else we insert it.
- It is useful for infinite graphs and that graph which are too large to store in memory.

- Uniform Cost Search explores the graph by prioritizing paths with lower cumulative costs, ensuring that the most cost-effective path is discovered. UCS is commonly used in algorithms for finding the shortest path in weighted graphs.

Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions

- Optimal and Complete Algorithm
- Time Complexity – $O(b^{1+C/e})$
- Space Complexity – $O(b^{1+C/e})$

Here C is the cost of the optimal solution

e is the least cost of any action

Steps to follow

This algorithm assumes that all operator have a cost

1. Initialize: Set OPEN={s}, CLOSED={}, C[] = 0
2. Fail: If OPEN={}, Terminate with Failure
3. Select: Select a minimum Cost Search, n,
4. Terminate: If n ∈ G, Terminate with success.

5. Expand: Generate the successor of n using o

   For each successor m:

   If m ∉ [open u closed]
   Set c(m) = c(n) + c(n, m) and

   If m ∈ [OPEN U CLOSED]
   Set C[m] = min{ C[m] ,C[n] + C(n, m)}

   If C[m] has decreased and
   m ∈ CLOSED , move it to OPEN

6. Loop: GOTO 2

BIBLIOGRAPHY

- GeeksForGeeks - https://www.geeksforgeeks.org/
- Artificial Intelligence – A Modern Approach by Stuart Russell and Peter Norvig
- Watson (latech.edu)

CONTRIBUTIONS

2201AI12 Hari Om Kumar	20%
2201AI13 Harpranav Singh Uppal	20%
2201AI14 Harsh Kumar	20%
2201AI15 Harshit Tomar	20%
2201AI16 Himani Yadav	20%