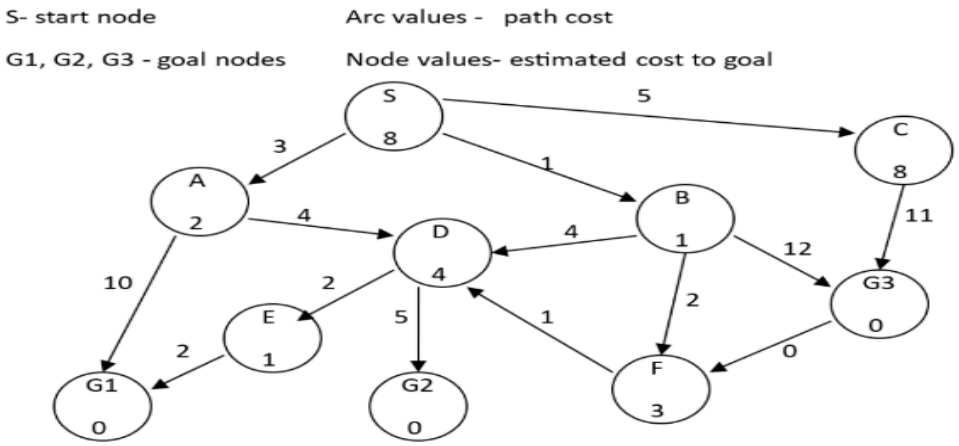
Q.1)Answer:



(a) A* Algorithm:

The A* algorithm combines both cost (g) and the heuristic estimate (h) to guide the search. The formula for the evaluation function is:

f(n)=g(n)+h(n)

Where:

- g(n) is the actual cost from the start node S to node n.
- h(n) is the estimated cost from node n to the goal node (the heuristic).

A* Table (Order of Expansions):

Step	Node Expanded	Frontier (with f-values)	Explored	Total Path Cost	Comments
1	S	A(5), B(2), C(9)	S	0	Start expanding from S
2	B	A(5), C(9), F(7), D(8)	S, B	1	B expanded (lowest f=2)
3	F	A(5), C(9), D(7), D(8)	S, B, F	3	F expanded (lowest f=7)
4	D	A(5), C(9), D(8), E(9), G2(12)	S, B, F, D	4	D expanded

5	E	A(5), C(9), G1(8), G2(12)	S, B, F, D, E	6	E expanded
6	G1	A(5), C(9), G2(12)	S, B, F, D, E, G1	8	Goal G1 reached

Path found: S→B→F→D→E→G1

- **Total path cost:** 8
- **Nodes expanded:** S, B, F, D, E, G1

(b) Uniform Cost Search (UCS):

Uniform cost search explores nodes in increasing order of path cost g(n). It ignores the heuristic values h(n) and relies solely on the actual cost incurred so far.

UCS Table (Order of Expansions):

Step	Node Expanded	Frontier (with g-values)	Explored	Total Path Cost	Comments
1	S	A(3), B(1), C(1)	S	0	Start expanding from S
2	B	A(3), C(1), F(3), D(5)	S, B	1	B expanded (lowest g=1)
3	F	A(3), C(1), D(5), D(4)	S, B, F	3	F expanded (lowest g=3)
4	D	A(3), C(1), D(5), E(6), G2(9)	S, B, F, D	4	D expanded
5	E	A(3), C(1), G1(8), G2(9)	S, B, F, D, E	6	E expanded
6	G1	A(3), C(1), G2(9)	S, B, F, D, E, G1	8	Goal G1 reached

Path found: S→B→F→D→E→G1

- **Total path cost:** 8
- **Nodes expanded:** S, B, F, D, E, G1

(c) Iterative Deepening A*:

Iterative Deepening A* combines the depth-first search's memory efficiency with A*'s optimality. It uses progressively larger thresholds on the fff-values, expanding nodes until a goal is found or the threshold is exceeded.

First Iteration (Threshold = 5):

Step	Node Expanded	Frontier (with f-values)	Explored	Total Path Cost	Comments
1	S	A(5), B(2), C(9)	S	0	Threshold is 5
2	B	A(5), C(9), F(7), D(8)	S, B	1	Threshold exceeded, restart

Second Iteration (Threshold = 7):

Step	Node Expanded	Frontier (with f-values)	Explored	Total Path Cost	Comments
1	S	A(5), B(2), C(9)	S	0	Threshold is 7
2	B	A(5), C(9), F(7), D(8)	S, B	1	Threshold exceeded, restart
3	F	A(5), C(9), D(7), D(8)	S, B, F	3	F expanded (lowest f=7)

Third Iteration (Threshold = 8):

Step	Node Expanded	Frontier (with f-values)	Explored	Total Path Cost	Comments
1	S	A(5), B(2), C(9)	S	0	Threshold is 8
2	B	A(5), C(9), F(7), D(8)	S, B	1	B expanded (lowest f=2)
3	F	A(5), C(9), D(7), D(8)	S, B, F	3	F expanded
4	D	A(5), C(9), E(9), G2(12)	S, B, F, D	4	D expanded
5	E	A(5), C(9), G1(8), G2(12)	S, B, F, D, E	6	E expanded

6	G1	A(5), C(9), G2(12)	S, B, F, D, E, G1	8	Goal G1 reached

Path found: S→B→F→D→E→G1

- **Total path cost:** 8
- **Nodes expanded:** S, B, F, D, E, G1

Summary of Nodes:

1. A*: S, B, F, D, E, G1
2. **Uniform Cost Search:** S, B, F, D, E, G1
3. Iterative Deepening A*: S, B, F, D, E, G1

Q.2)Answer:-

Part A: Min-Max Algorithm and Alpha-Beta Pruning

Min-Max Algorithm (without pruning):

1. **At Leaf Nodes:** The leaf values are:
   - Leftmost path: `2, -1`
   - Second path: `2, -5`
   - Third path: `1, -3`
   - Rightmost path: `-3, 5`
2. **At Level 1 (Min Nodes B and C):**
   - **B (Min Node):**
     Takes the minimum of its children:
     - D (2, -1): Min(-1)
     - E (2, -5): Min(-5)
     - So, B = Min(-1, -5) = **-5**
   - **C (Min Node):**
     - F (1, -3): Min(-3)
     - G (-3, 5): Min(-3)
     - So, C = Min(-3, -3) = **-3**
3. **At Root (Max Node A):**
   - **A (Max Node):**
     - B = -5
     - C = -3
     - So, A = Max(-5, -3) = **-3**

Best Play:

- The optimal move for Player 1 (Max) is to choose Node C since it provides a higher value (-3) than Node B (-5).
- Arrows can be placed from A to C, and from C to F (since Min at C prefers -3, and F leads to -3).

Alpha-Beta Pruning:

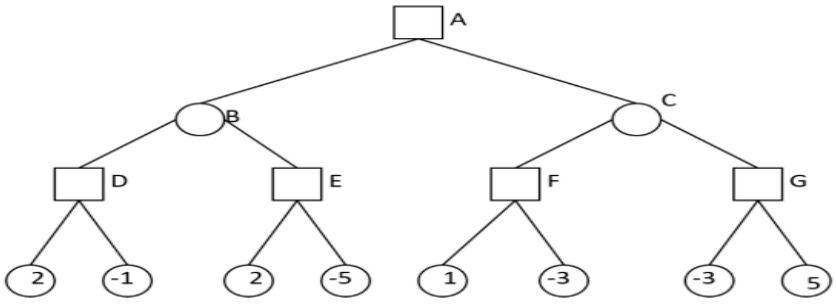
- Start with $\alpha = -\infty$ and $\beta = +\infty$
- Traverse from left to right:

1. **At Node B:**
    - For Node D: Check values (2, -1). Min is -1, set ($\beta = -1$).
    - For Node E: Check values (2, -5). Min is -5, now ($\beta = -5$) for B. Return -5 for Node B.
2. **At Node C:**
    - For Node F: Check values (1, -3). Min is -3, set ($\alpha = -3$).
    - For Node G: The first value is -3. Since ($\alpha = -3$), no need to check further (prune).

- Pruned branches:
    - The right sibling of 5 under G is pruned as it won't influence the outcome based on (α)-(β).

Visualization

Here is a visual representation of the game tree with arrows indicating the best moves and crossed-out branches where pruning occurs:



Best Moves:

A -> C and C -> F

Alpha-Beta Pruning: Prune right branch of G

Part B: Rearranging for Best and Worst Cases

Best Case:

To achieve maximum pruning, the tree should be rearranged to present the most extreme (minimum or maximum) values first, allowing earlier pruning:

1. **For Max Nodes (A):** The most promising child should be evaluated first. In the best case, prune as many nodes as possible. If the worst values for Player 2 (Min) appear first, Player 1 can prune the remaining branches.
   Rearrangement for Best Case Pruning:
   - Under Node B, switch -5 and 2, so -5 appears first.
   - Under Node C, switch -3 and 5, so -3 appears first.
   - This will allow maximum pruning by cutting branches early.

Worst Case:

To achieve the worst case, rearrange the tree such that pruning is delayed as much as possible by evaluating the least promising values first:

1. **For Max Nodes:** Present the best values last, so that every branch is evaluated fully before any pruning can occur.
   Rearrangement for Worst Case:
   - Under Node B, switch -1 and 2, so 2 appears first.
   - Under Node C, switch 5 and -3, so 5 appears first.

This will force the algorithm to evaluate all branches fully before concluding.

Part C: Best-Case Complexity

In the best case, pruning cuts the number of nodes evaluated roughly in half at each level, reducing the search space. The tree has a branching factor (b) and depth (d). Alpha-beta pruning can reduce the effective depth to half, leading to complexity (O(b^{d/2})).

This is because, in the best case, the algorithm skips nearly half of the nodes, cutting down the search significantly. Therefore, instead of evaluating (b^d) nodes, we evaluate closer to (b^{d/2}), which explains the reduced complexity.

Q.3)Answer:-

a) After running the public test cases the IDS and Bidirectional search give the outputs as below.

Test Case 1: Start node: 1, End node: 2
Output from IDS: [1, 7, 6, 2]
Output from Bidirectional Search: [1, 7, 6, 2]

Test Case 2: Start node: 5, End node: 12
Output from IDS: [5, 97, 98, 12]
Output from Bidirectional Search: [5, 97, 98, 12]

Test Case 3: Start node: 12, End node: 49
Output from IDS: None
Output from Bidirectional Search: None

Test Case 4: Start node: 4, End node: 12
Output from IDS: [4, 6, 2, 9, 8, 5, 97, 98, 12]
Output from Bidirectional Search: [4, 6, 2, 9, 8, 5, 97, 98, 12]

b) While IDS and Bidirectional Search are both graph search algorithms, they may produce different paths for the same input and output nodes. This discrepancy arises from their distinct search strategies. IDS conducts a depth-limited search, gradually increasing the depth limit, while Bidirectional Search explores from both the start and end nodes simultaneously. Due to its depth-first approach, IDS doesn't always guarantee the shortest path (in terms of edges). In contrast, Bidirectional Search ensures the shortest path by exploring nodes closest to both the start and end, converging in the middle.

c)

IDS

1. Start:1, End:2 ⇒ Time: 0.00231 sec, Memory: 2072 bytes
2. Start:5, End:12 ⇒ Time: 0.00195 sec, Memory: 2058 bytes
3. Start:4, End:12 ⇒ Time: 0.01927 sec, Memory: 3202 bytes

Bidirectional Search

1. Start:1, End:2 ⇒ Time: 0.00118 sec, Memory: 3875 bytes
2. Start:5, End:12 ⇒ Time: 0.00002 sec, Memory: 3989 bytes
3. Start:4, End:12 ⇒ Time: 0.00338 sec, Memory: 7941 bytes

Bidirectional search takes considerably less time than IDS. However, in terms of memory usage, IDS performs much better compared to Bidirectional search.

d) and e)

After running public test cases for A* and Bi-Directional A*. The output paths are different.

Test Case 1: Start node: 1, End node: 2

Output from A*: [1, 27, 9, 2]

Output from Bidirectional Heuristic: [1, 27, 6, 2]

Test Case 2: Start node: 5, End node: 12

Output from A*: [5, 97, 28, 10, 12]

Output from Bidirectional Heuristic: [5, 97, 98, 12]

Test Case 3: Start node: 4, End node: 12

Output from A*: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12]

Output from Bidirectional Heuristic: [4, 34, 33, 11, 32, 31, 3, 5, 97, 28, 10, 12]

Test Case 4: Start node: 12, End node: 49

Output from A*: None

Output from Bidirectional Heuristic: None

The difference in outputs between A* and Bidirectional A* is primarily due to their distinct search strategies and cost functions. A* explores nodes in a single direction toward the goal, guided by a combined cost function of g(n) + h(n). Bidirectional A*, on the other hand, explores from both the start and end nodes, potentially leading to different intermediate paths due to its bidirectional approach and potentially different heuristic

A*

1. Start:1, End:2 ⇒ Time: 0.00097 sec, Memory: 7102 bytes
2. Start:5, End:12 ⇒ Time: 0.00425 sec, Memory: 8013 bytes
3. Start:4, End:12 ⇒ Time: 0.00461 sec, Memory: 7795 bytes
4. Start:12, End:49 ⇒ Time: 0.02217 sec, Memory: 14723 bytes

Bidirectional A*

1. Start:1, End:2 ⇒ Time: 0.00221 sec, Memory: 14687 bytes
2. Start:5, End:12 ⇒ Time: 0.00812 sec, Memory: 15392 bytes
3. Start:4, End:12 ⇒ Time: 0.00534 sec, Memory: 15401 bytes
4. Start:12, End:49 ⇒ Time: 0.00208 sec, Memory: 14721 bytes

A* generally outperforms Bidirectional Heuristic Search in terms of both time and memory usage, except in cases where no valid path exists. In such scenarios, Bidirectional Heuristic Search may have an advantage. While A* typically requires less memory, the memory usage can become comparable when no path is found.