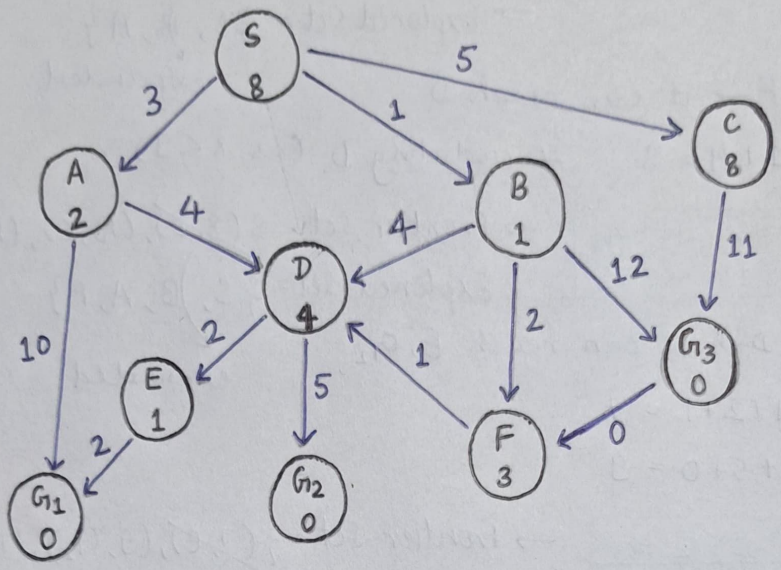
AI Assignment – Search Algorithm

SHOBHIT RAJ
2022482

Q1.



Start node – S
Goal node – G_1

a) A* Search – $f(n) = g(n) + h(n)$ → data structure → priority queue

1. Initially, starting with node S. → Frontier Set = {(~~8~~, S)}
   Explored set = ∅ → expanded

S	A	B	C	D	E	F	G_1	G_2	G_3
8	∞	∞	∞	∞	∞	∞	∞	∞	∞

⇒ f-values.

Node expansion of S → it can reach A, B, C

$F(A) = 3+2 = 5$
$F(B) = 1+1 = 2$
$F(C) = 5+8 = 13$

2. → Frontier Set = {(~~2~~, B), (5, A), (13, C)}
   Explored Set = {S} → expanded

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	∞	∞	∞	∞	∞	∞

Node expansion of B → it can reach D, F, G_3

$F(D) = 1+4+4 = 9$
$F(F) = 1+2+3 = 6$
$F(G_3) = 1+12+0 = 13$

3. → Frontier Set = {(~~5~~, A), (6, F), (9, D), (13, C), (13, G_3)}
   Explored Set = {S, B} → expanded

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	9	∞	6	∞	∞	13

Node expansion of A → it can reach G_1, D

$F(D) = 3+4+4 = 11$ // not updated as already D → 9.
$F(G_1) = 3+10+0 = 13$

P.T.O.

4.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	9	∞	6	13	∞	13

→ Frontier Set = {(6, F), (9, D), (13, C), (13, G_1), (13, G_3)}

Explored Set = {S, B, A}

expanded

Node expansion of F → it can reach D

F(D) = 1+2+1+4 = 8 # updating D (as 8 < 9)

5.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	8	∞	6	13	∞	13

→ Frontier Set = {(8, D), (13, C), (13, G_1), (13, G_3)}

Explored Set = {S, B, A, F}

expanded

Node expansion of D → it can reach E, G_2

F(E) = 1+2+1+2+1 = 7

$F(G_2)$ = 1+2+1+5+0 = 9

6.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	8	7	6	13	9	13

→ Frontier Set = {(7, E), (9, G_2), (13, C), (13, G_1), (13, G_3)}

Explored Set = {S, B, A, F, D}

expanded

Node expansion of E → it can reach G_1

$F(G_1)$ = 1+2+1+2+2+0 = 8 # updating G_1 (as 8 < 13)

7.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	13	8	7	6	8	9	13

→ Frontier Set = {(8, G_1), (9, G_2), (13, C), (13, G_3)}

Explored Set = {S, B, A, F, D, E}

expanded

Node expansion of G_1.

(As G_1 = Goal) → reached → stop search.

Optimal Path ⇒ S → B → F → D → E → G_1 ⇝ shortest path weight = 8.

b) Uniform Cost Search (DIJKSTRA) – $f(n) = g(n)$ → data structure → priority queue

1. Initially, starting with node S → Frontier Set = {(0, S)}
   Explored Set = ∅ → expanded

S	A	B	C	D	E	F	G_1	G_2	G_3
0	∞	∞	∞	∞	∞	∞	∞	∞	∞

⇒ f-values.

Node expansion of S → it can reach A, B, C

F(A) = 3
F(B) = 1
F(C) = 5

2.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	∞	∞	∞	∞	∞	∞

→ Frontier Set = {(1, B), (3, A), (5, C)}
Explored Set = {S} → expanded

Node expansion of B → it can reach D, F, G_3

F(D) = 1+4 = 5
F(F) = 1+2 = 3
F(G_3) = 1+12 = 13

3.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	5	∞	3	∞	∞	13

→ Frontier Set = {(3, A), (3, F), (5, C), (5, D), (13, G_3)}
Explored Set = {S, B} → expanded (A < F rule) [alphabetical]

Node expansion of A → it can reach D, G_1

F(D) = 3+4 = 7 // not updated as already D → 5
F(G_1) = 3+10 = 13

4.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	5	∞	3	13	∞	13

→ Frontier Set = {(3, F), (5, C), (5, D), (13, G_1), (13, G_3)]
Explored Set = {S, B, A} → expanded

Node expansion of F → it can reach D

F(D) = 1+2+1 = 4 // updating D (as 4 < 5)

5.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	4	∞	3	13	∞	13

→ Frontier Set = {(4, D), (5, C), (13, G_1), (13, G_3)}
Explored Set = {S, B, A, F} → expanded

Node expansion of D → it can reach E, G_2

F(E) = 1+2+1+2 = 6
F(G_2) = 1+2+1+5 = 9

6.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	4	6	3	13	9	13

→ Frontier Set = {(5, C), (6, E), (9, G_2), (13, G_1), (13, G_3)}
Explored Set = {S, B, A, F, D} → expanded

Node expansion of C → it can reach G_3

F(G_3) = 5+11 = 16 // not updated as already G_3 → 13

P.T.O.

7.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	4	6	3	13	9	13

→ Frontier Set = {~~(6, E)~~, (9, G_2), (13, G_1), (13, G_2)}

Explored Set = {S, ~~B~~, A, F, D, C}

expanded

Node expansion of E → it can reach G_1

$F(G_1) = 1+2+1+2+2 = 8$ // updating G_1 (as $8 < 13$)

8.

S	A	B	C	D	E	F	G_1	G_2	G_3
0	3	1	5	4	6	3	8	9	13

→ Frontier Set = {~~(8, G_1)~~, (9, G_2), (13, G_3)}

Explored Set = {S, B, A, F, D, C, E}

expanded

Node expansion of G_1.

(As G_1 = Goal) → reached → stop search.

Optimal path ⇒ S → B → F → D → E → G_1 ⇝ Shortest path weight = 8.

c) Iterative Deepening A* – $f(n) = g(n) + h(n)$ with threshold value

1. Iteration 1 – Initially, starting with node S (Root Node)

∴ Threshold = current root node's value

= F(S) = G(S) + H(S) = 8.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	∞	∞	∞	∞	∞	∞	∞	∞	∞

f-values.

→ Frontier list = {(8, S)}
Explored Set = ∅
Pruned set = ∅ → expanded

Node expansion of S → it can reach A, B, C

F(A) = 3 + 2 = 5

F(B) = 1 + 1 = 2

F(C) = 5 + 8 = 13 (∵ F-value > threshold) → C is pruned (added to pruned list for next iteration's threshold, if goal is not reached)

2.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	∞	∞	∞	∞	∞	∞

→ Frontier list = {(5, A), (2, B)}
Explored List = {S}
Pruned Set = {(13, C)} → expanded

Node expansion of A → explored first in S. (left to right)
it can reach D, G_1

F(D) = 3 + 4 + 4 = 11
F(G1) = 3 + 10 + 0 = 13
} as (F-value > threshold) → D & G_1 pruned.

3.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	∞	∞	∞	∞	∞	∞

→ Frontier Set = {(2, B)}
Explored Set = {S, A}
Pruned Set = {(13, C), (9, D) (13, G_1)} → expanded

Node expansion of B → it can reach D, F, G_3

F(D) = 1 + 4 + 4 = 9
F(F) = 1 + 2 + 3 = 6
F(G_3) = 1 + 12 + 0 = 13
→ as (f-value > threshold) → D & G_3 pruned

4.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	∞	∞	6	∞	∞	∞

→ Frontier Set = {(6, F)}
Explored Set = {S, A, B}
Pruned Set = {(13, C), (9, D), (13, G_1), (13, G_3)} → expanded

Node expansion of F → it can reach D

F(D) = 1 + 2 + 1 + 4 = 8 # can reach within threshold (update & delete from pruned)

P.T.O.

5.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	8	∞	6	∞	∞	∞

→ Frontier Set = {(8, D)}
Explored Set = {S, A, B, F}
Pruned Set = {(13, C), (13, G_1), (13, G_3)}

Node expansion of D → it can reach E, G_2 (expanded)

$F(E) = 1+2+1+2+1 = 7$

$F(G_2) = 1+2+1+5+0 = 9$ (f-value > threshold) → G_2 pruned

6.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	8	7	6	∞	∞	∞

→ Frontier Set = {(7, E)}
Explored Set = {S, A, B, F, D}
Pruned Set = {(9, G_2), (13, C), (13, G_1), (13, G_3)}

Node expansion of E → it can reach G_1 (expanded)

$F(G_1) = 1+2+1+2+2+0 = 8$ // can reach within threshold, hence updated & delete from pruned

7.

S	A	B	C	D	E	F	G_1	G_2	G_3
8	5	2	∞	8	7	6	8	∞	∞

→ Frontier Set = {(8, G_1)}
Explored Set = {S, A, B, F, D, E}
Pruned Set = {(9, G_2), (13, C), (13, G_3)}

Node expansion of G_1 (expanded)

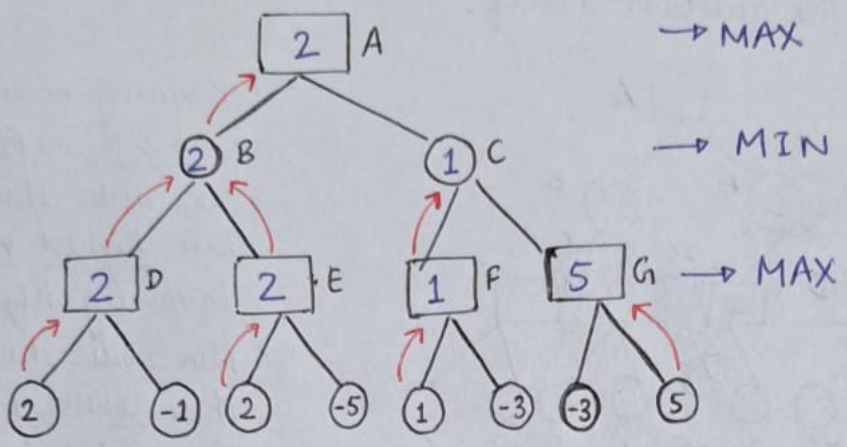
(As G_1 = goal) → reached → stop search.

(didn't need 2nd iteration) → first threshold was enough.

Optimal path ⇒ S → B → F → D → E → G_1 ⇝ shortest path weight = 8.

Q2.

a)

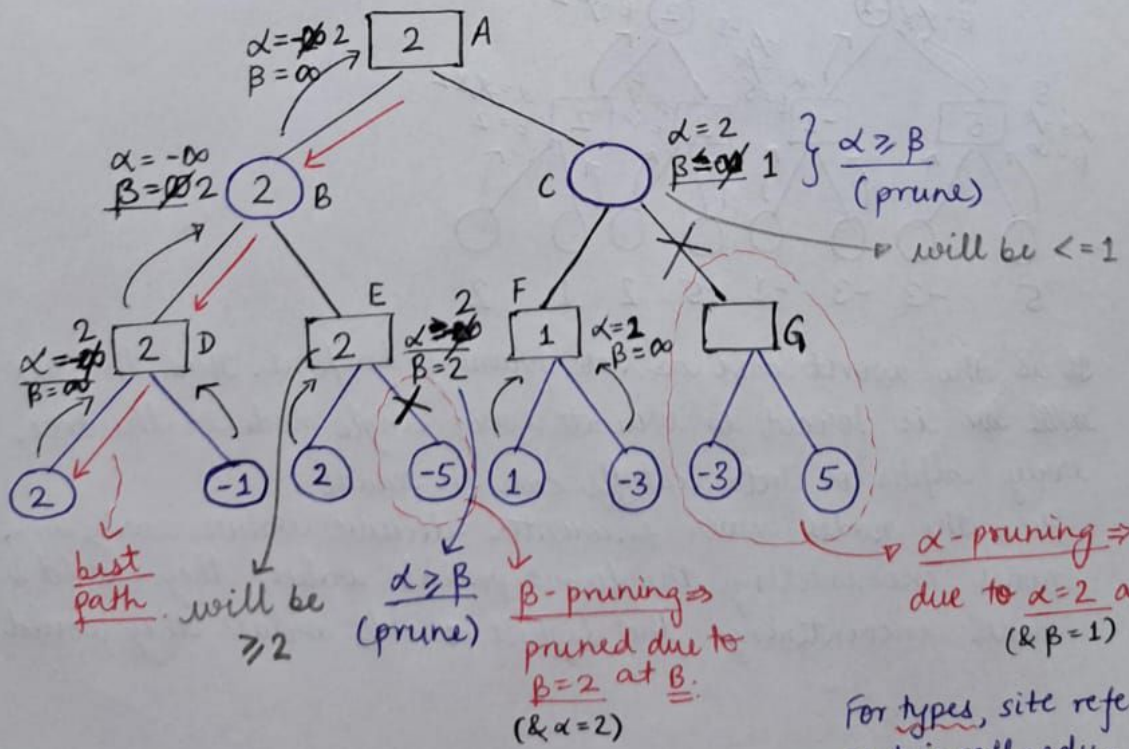


The arrows represent the best move for that player at that level.

i.e. A → moves towards B.
B → can take any decision, won't affect result
C → move towards F.
D → take decision which gives utility 2 (max)
E → take decision which gives utility 2 (max)
F → take decision which gives utility 1 (max)
G → take decision which gives utility 5 (max)

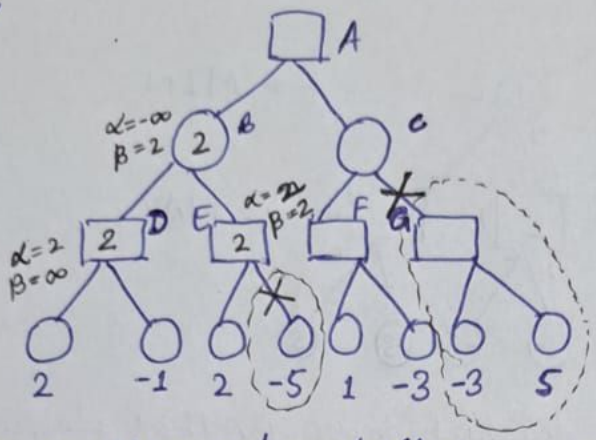
α-β Pruning ⇒

α = represents best value that maximiser can guarantee.
β = represents best value that minimiser can guarantee.



For types, site referred:
cs.trincoll.edu. (Trinity College Minimax Notes).

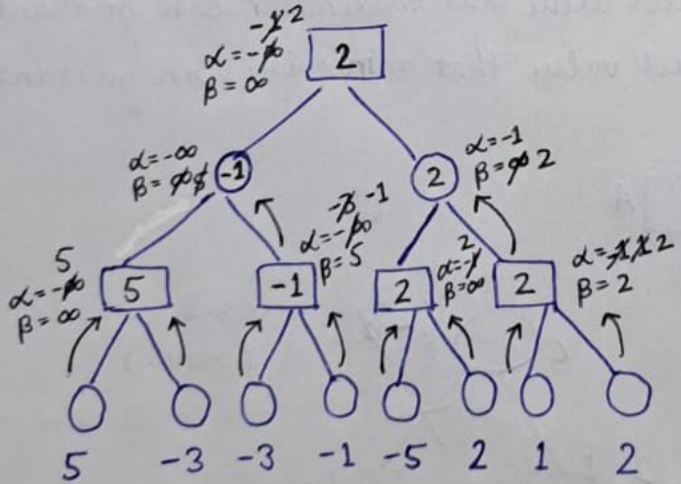
(b) Best Case: Maximum pruning is achieved in the game tree given in the question itself.



→ Pruning occurs when $\alpha \geq \beta$ ⇒ For this :-
→ Max nodes should explore their highest values first favouring higher α.
→ Min nodes should explore their lowest values first favouring lower β.

The nodes are arranged such that pruning eliminates maximum no. of branching. They are arranged in such a way that $\alpha \geq \beta$ condⁿ. occurs as early as possible. This achieves maximum pruning as D & E has to be evaluated early for determination of B. Here, the nodes are such that it sets $\beta = 2$ for E which prunes E→-5. As soon as F is evaluated & returns 1, as $\alpha = 2$ for C, this achieves $\alpha \geq \beta$ fastest leading to entire G subtree being pruned. The values ($2 \geq 1$) that lead to pruning are encountered as early and no other rearrangement can guarantee better pruning results.

Worst Case: Worst case will happen when no branches are pruned & all nodes are explored / branches are computed.

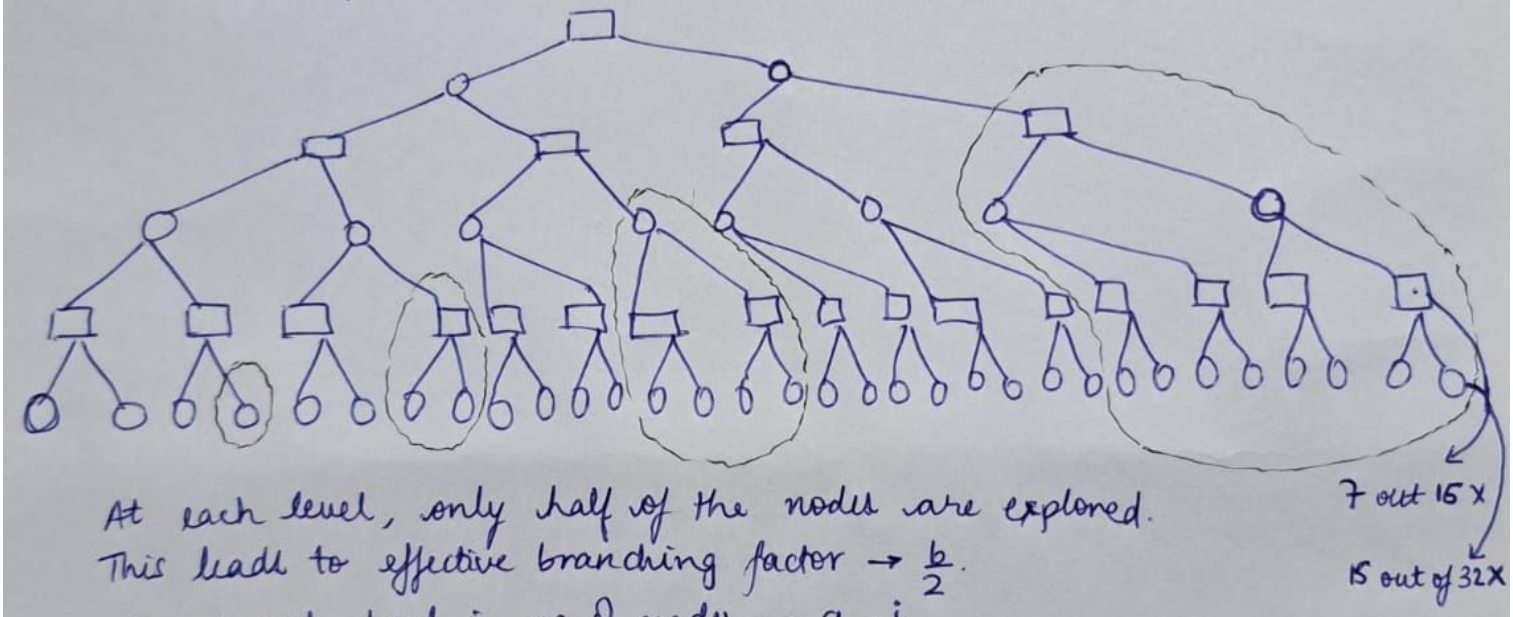


It is the worst case as no pruning happens. Here, the algorithm is forced to evaluate every single node in the tree, as no early alpha or beta cutoffs can be made.
Here, the nodes never encounter decisive values early on, the max nodes encountering the lowest possible values they could & min nodes encountering the highest possible values they could.

(c) In the naive Minimax algorithm without any alpha beta pruning, we have to explore every node in the tree. Since, each node at depth d has b children, so tree explored to depth d → b^d nodes.

∴ Time Complexity → $O(b^d)$.

Based on the best case for alpha beta pruning, as I mentioned that the algorithm tries to find the optimal values early on.
In this case, approximately half of the tree is pruned at each level as demonstrated in game tree of part (b). Even if there is more depth, best case pruning would look like :- (look ahead depth).



At each level, only half of the nodes are explored.
This leads to effective branching factor → $\frac{b}{2}$.
So, at each level i, no. of nodes → $(\frac{b}{2})^i$.
Instead of evaluating all depths, as exploring fewer nodes, the effective depth is reduced to roughly $d/2$. (following the symmetry of pruning)

∴ Time Complexity → $(\frac{b}{2})^{d/2} \Rightarrow O(b^{d/2})$.

Q3.

(a) For Iterative Deepening Search (IDS), I implemented a depth-limited DFS that recursively explores nodes up to a certain depth. The function increments the depth limit iteratively until a solution is found. Each time, it explores deeper levels of the graph from the start node, marking visited nodes and constructing the path. If no path is found at the current depth, it increases the depth and retries. This process continues until the goal node is reached or all depths are exhausted.

For Bidirectional Breadth-First Search (BFS), I implemented two simultaneous BFS processes, one starting from the source node and the other from the goal node. Each BFS explores adjacent nodes while tracking visited nodes and parent nodes. If the two searches meet or one of them encounters the other's explored nodes, the algorithm reconstructs the path by backtracking from both directions. This approach allows finding the shortest path more efficiently by reducing the search space.

(b) For each public test case, it can be observed that the paths obtained using both algorithms are the same. However, this is not true for every pair of nodes in the graph. For example,

start node = 1 and goal node = 17:

Iterative Deepening Search Path: [1, 27, 9, 8, 5, 97, 98, 12, 57, 26, 85, 86, 89, 116, 122, 123, 45, 17]

Bidirectional BFS Path: [1, 27, 9, 8, 5, 97, 98, 12, 57, 26, 85, 118, 18, 55, 52, 51, 45, 17]

In this example, the paths differ, but the total length of both paths remains the same.

The paths obtained by both algorithms will always be same if the shortest path between the source and the destination is unique i.e. where there is only one shortest path, both algorithms will find it. When there are multiple paths of the same length (i.e., multiple shortest paths), the paths produced by the two algorithms may differ. This is because:

- IDS explores nodes in a depth-first manner, which could cause it to find a different valid path compared to BFS.
- Bidirectional BFS works from both ends (source and destination), so the point where the searches meet could result in a different path being chosen.

Thus, while both algorithms are designed to find the shortest-length path, the specific path may differ when multiple shortest paths exist. However, **the length of the paths will always be the same** for any pair of nodes in the graph, as both algorithms aim to find the shortest-length path, not necessarily the optimal-cost path.

(c) --- Performance Report ---
Total time for IDS: 1840.3228 seconds
Total memory usage for IDS: 59.3984 MB

Total time for Bidirectional BFS: 112.5248 seconds
Total memory usage for Bidirectional BFS: 99.0129 MB

Time Complexity => Bidirectional BFS is significantly faster than IDS, as expected, given that it avoids redundant exploration by meeting in the middle. The inefficiency if IDS comes from redundant searches, as nodes near the start are explored multiple times.
Time complexity of IDS for a graph with branching factor b and depth d is approximately O(b^d). The time complexity of Bidirectional BFS is O(b^{d/2}) as it searches from both directions, effectively halving the depth.

Space Complexity => Bidirectional BFS uses more memory than IDS. It is because IDS only needs to store the current path and visited nodes, making it more memory efficient. Its space complexity is O(d), where d is the depth of the search tree, as it only stores information about the current search path and a few extra variables. While, Bidirectional BFS, explores from two fronts, maintains queues for both searches and requires storing all visited nodes which leads to increased memory consumption. The space complexity is O(b^{d/2}).

(d) For the A* Search Algorithm, I implemented a pathfinding method that uses a heuristic based on the Euclidean distance between nodes to estimate the cost of reaching the goal. The algorithm maintains g_cost (cost from the start node) and f_cost (total estimated cost), expanding the node with the smallest f_cost at each step. It iteratively selects the optimal path using a priority queue-like approach, updating the cost values as it explores neighboring nodes. Once the goal node is reached, the path is reconstructed by backtracking through the parent nodes.

In the Bidirectional A* Search Algorithm, I used two simultaneous A* searches, one starting from the source and the other from the goal. Both searches expand nodes using the same Euclidean distance heuristic. The algorithm tracks the forward and backward costs (f_cost and g_cost) separately, and the path is constructed when both searches meet at a common node. The algorithm ensures that the total path cost is minimized by comparing paths from the two directions at each step, making it more efficient than standard A* for larger graphs.

(e) For each public test case, it can be observed that the paths obtained using both algorithms are the same.

I observed that both algorithms consistently produced the same paths between any pair of nodes. This suggests that for the given graph, both algorithms follow the same optimal path, which is determined by the unique costs associated with each node traversal in this graph.

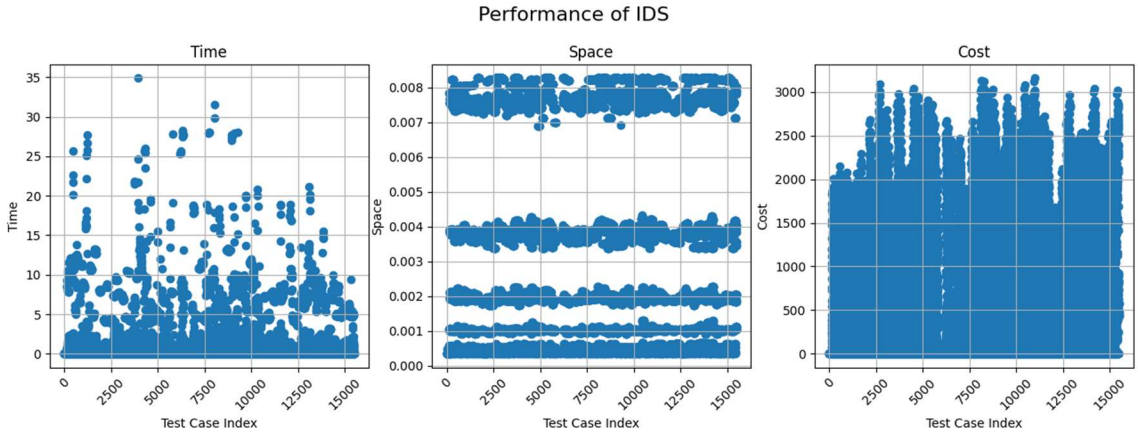
However, while the paths are identical in this graph, this may not always be the case for other graphs. When there is a **unique optimal cost** between nodes, both A* and Bidirectional A* will always yield the same path. But in cases where there are **multiple paths with equal optimal cost**, the paths generated by each algorithm may differ, depending on the order in which nodes are explored during execution. Despite the possible difference in paths, both algorithms will always ensure that the total cost of the path remains the same and minimal (if admissible heuristic). Therefore, for the current graph, both algorithms yield the same path for all node pairs, but this is not guaranteed for all graphs with multiple optimal-cost paths.

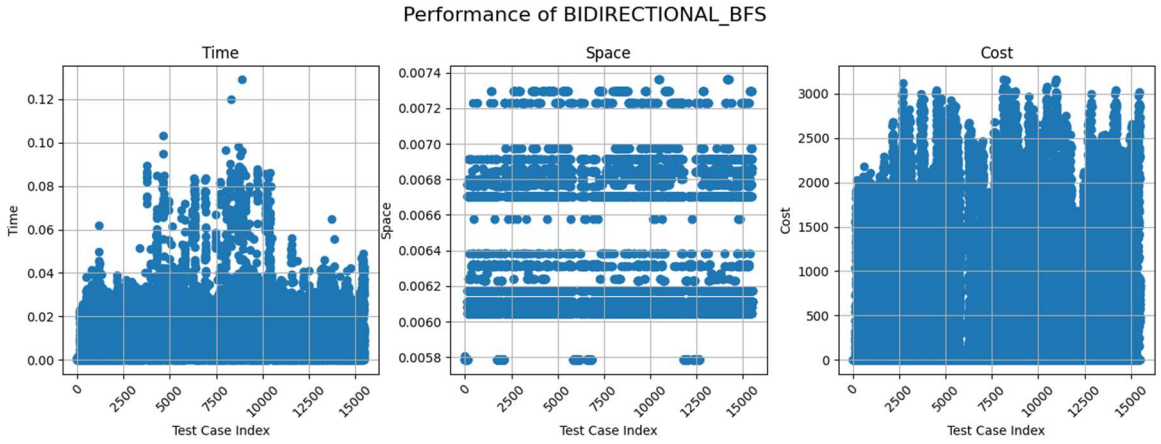
--- Performance Report ---
Total time for A* Search: 244.8788 seconds
Total memory usage for A* Search: 262.3402 MB
Total time for Bidirectional A* Search: 442.3882 seconds
Total memory usage for Bidirectional A* Search: 639.4812 MB

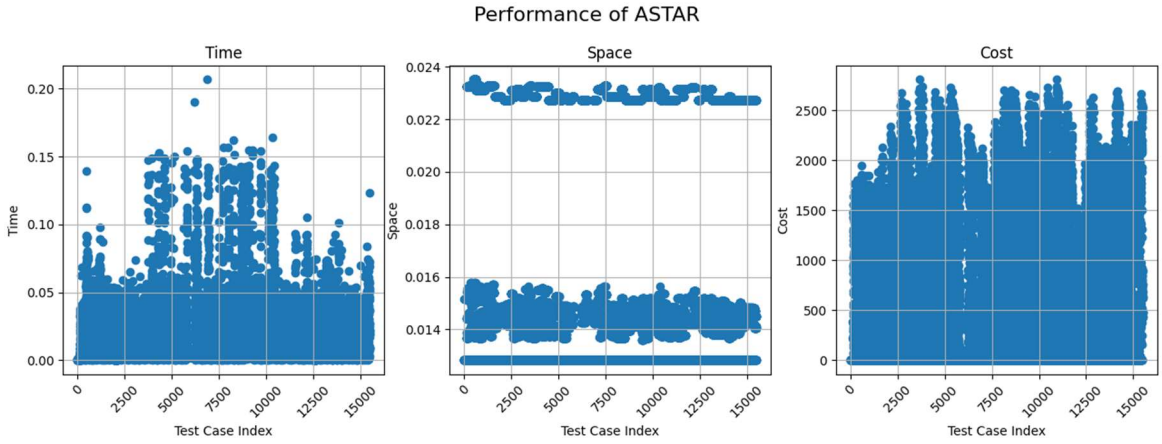
Time Complexity => The common perception is that bidirectional searches are faster, but in this case, the Bidirectional A* Search took significantly longer than the regular A* search. This shows that the bidirectional approach may not always reduce time complexity if the heuristic does not efficiently guide the search towards a common node quickly. Bidirectional A* is taking more time because it is forced to explore multiple paths around the middle of the graph to ensure that the path it finds is cost-optimal.

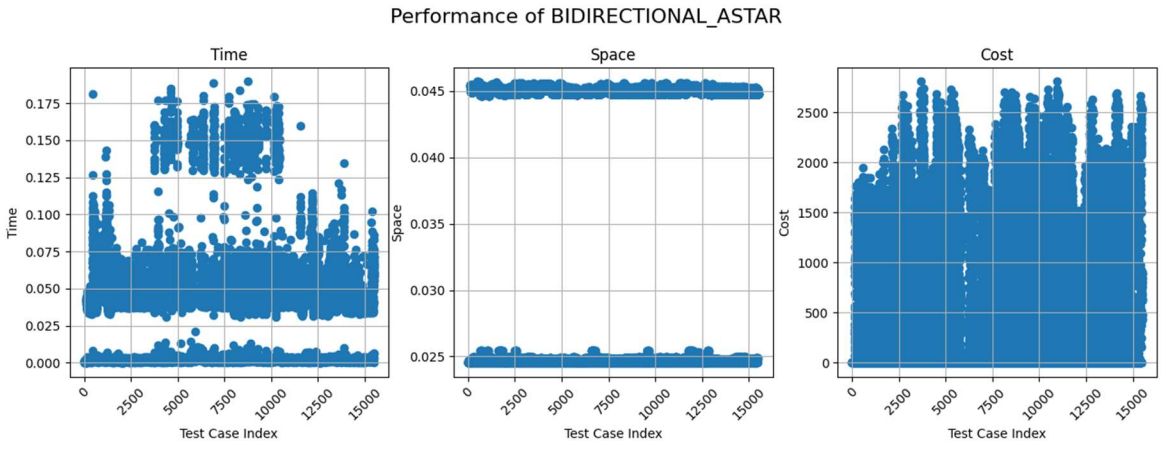
Space Complexity => Bidirectional A* Search required over twice as much memory as A* Search. This is expected, as bidirectional search maintains two sets of open and closed priority queue frontiers, one for each direction. As both searches progress, the memory usage grows more quickly due to storing additional node information for both the forward and backward searches.

(f)









For each test case, metrics were collected as the algorithm executed. This included measuring the time taken for computation using a timer, the memory consumption using tracemalloc, and the path cost obtained after solving each test case.

Time Analysis => Informed algorithms (A* and Bidirectional A*) generally have a more compact distribution of time values, indicating that they take less time in most cases. However, Bidirectional A* tends to take slightly longer than A* due to the overhead of managing two simultaneous searches. Uninformed search algorithms like IDS have much larger time values, with significant variance in the scatter plot.

Space Analysis => Informed algorithms (A* and Bidirectional A*) have a higher memory consumption compared to uninformed algorithms. This is because informed algorithms store more information (e.g., heuristic values) and often need to keep more nodes in memory. Bidirectional A*, in particular, uses more memory than A*, as it runs two search processes simultaneously. IDS, on the other hand, is memory efficient due to its depth-first nature and re-exploration in each iteration, which avoids storing large search frontiers. Bidirectional BFS has moderate space usage, more than IDS but less than A* and Bidirectional A*.

Cost Analysis => The cost is a metric to judge the **optimality** of the solution. All algorithms, both informed (A* and Bidirectional A*) and uninformed (IDS and Bidirectional BFS), seem to achieve optimal costs across most of the test cases, which is consistent with the fact that the informed algorithms give the most optimal path (if admissible heuristic) the uninformed algorithms give the shortest length path. The plots show that costs are clustered in a similar pattern across algorithms.

Benefits of Informed Algorithms =>
Faster Execution: Informed algorithms like A* and Bidirectional A* use heuristics to guide the search, which often results in faster convergence to the goal compared to uninformed algorithms. As seen in the scatter plots, A* completes most test cases in less time than uninformed searches like IDS.

Drawbacks of Informed Algorithms =>
Higher Space Complexity: Informed algorithms require storing additional information (e.g., f-scores, g-scores, heuristic estimates) and tend to have a higher memory footprint, as seen in the space scatter plots.
Heuristic Sensitivity: The performance of informed search algorithms depends heavily on the quality of the heuristic used.

Benefits of Uninformed Algorithms =>
Lower Memory Usage: As seen in the space plots, IDS is particularly memory efficient, which makes it suitable for scenarios where memory is the limiting factor.

Drawbacks of Uninformed Algorithms =>
Slower Execution: Without the guidance of a heuristic, uninformed algorithms like IDS and Bidirectional BFS must exhaustively search large portions of the graph, leading to significantly longer execution times in most cases, as shown in the time scatter plots.

Hence, Informed algorithms (A and Bidirectional A) are much faster and more efficient in terms of time but come with a higher memory cost. They are well-suited for problems where execution time is critical, and memory usage is not a limiting factor. Uninformed algorithms (IDS and Bidirectional BFS) are slower but more memory efficient, making them better suited for memory-constrained environments or when no heuristic is available.

(g) The code uses **Tarjan's algorithm** to find bridges in a graph. It performs a DFS to compute discovery and low times for each node, identifying edges as bridges when the condition low[neighbor] > discovery[node] is met. The adjacency matrix is first converted to an undirected graph to ensure symmetry, and duplicate bridges are removed since the graph is undirected.