Pranjal Bharti
2021080

AI Ass-1

Q1 Q.) A* Algo.

algorithm func^n used:- $f(N) = g(N) + h(N)$
$g(N)$ → actual cost from node.
$h(N)$ → estimated cost from node to goal node.

Step 1 → exploring S : $f(S) = 0+8 = 8$
Step 2 → exploring the neighbour of S :-

S → A	S → B	S → C
$f(A) = 3+2 = 5$	$f(B) = 1+1 = 2$	$f(C) = 5+8 = 13$

Step 3 → exploring the min cost path (S → B):-

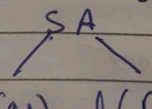
SB → D	SB → F	SB → G3
$f(D) = 4+4 = 8$	$f(F) = 2+3 = 5$	$f(G3) = 12+0 = 12$

$f(A)$ & $f(F)$ have the same cost
∴ We will explore both.

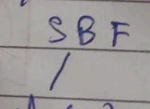
Step 4 :- expanding for both $f(A)$ and $f(F)$:-

S → A	SB → F

A being lexicographically smaller, we expand A first



$f(G1)$ (10) $f(D)$ (8)



$f(D)$ 15

Step 5 :- Since $f(D)$ is smaller we will expand D

SB FD

$f(E) = 3$, $f(G2) = 5$.

Step 6 :- SB FDE

$f(E)$ → explore later $f(g) = 2$ reached node.

Final path → S → B → F → D → E → G1

Step	Node expanded	g(n)	h(n)	f(n)	explored nodes
1	S	0	8	8	S
2	B	1	1	2	S, B
3	A	3	2	5	S, B, A
4	F	2	3	5	S, B, A, F
5	D_F	1	4	5	S, B, A, F, D
6	E	2	1	3	S, B, A, F, D, E
7	$G1_E$	2	0	2	S, B, A, F, D, E, G1

Path :- S → B → F → D → E → G1
Total cost = 8

b) Uniform Cost Search.

Step 1 → start by checking the neighbouring nodes of the starting node, S

S → A → 3 S → B → 1 S → C → 5

choose the minimum from these (S → B)

Step 2 :- now check the neighbouring nodes of B

S → B → D = 5
S → B → F = 3
S → B → G3 = 13

Min is chosen, else the last alphabet.

Step 3 :- S → A chosen
S → A → D → 7
S → A → G1 → 13

Step 4 → S → B → F chosen
S → B → F → D = 4

Step 5 → S → B → F → D chosen

Step	Node Expanded	$f(n)$	Explored nodes
1	S	0	S,
2	B	1	S, B
3	A	3	S, A
4	F	3	S, B, F
5	D	4	S, B, F, D
6	E	7	S, B, F, D, E
7	G1	8	S, B, F, D, E, G1

~~(c) Depth~~

(c) Deepening A*

Step 1:- start from S, expand neighbouring node
S → A = 5 [threshold = 8]
S → B = 2

Step 2:- S → A → G1 = 9 (terminated)

Step 3:- Choosing minimum,
S → B → D = 9 (terminated)
S → B → F = 6 (chosen)

Step 4:- S → B → F → D = 8 (chosen)

S5 :- S → B → F → D → E = 7 (chosen)

S6 :- S → B → F → D → E → G1 = 8 (goal reached)
S → B → F → D → E = 6
S → B → F → D → G2 = 9 (terminated)

S7 :- S → B → D chosen
S → B → D → E = 7
S → B → D → G2 = 10 (terminated)

S8 :- S → C → G3 = 16 (terminated.

S9 :- S → B → F → D → E ⇒ chosen

S1D:- S → A → D chosen in lexicographic order.
S → A → D → E = 9
S → A → D → G2 ⇒ 12 (terminated)

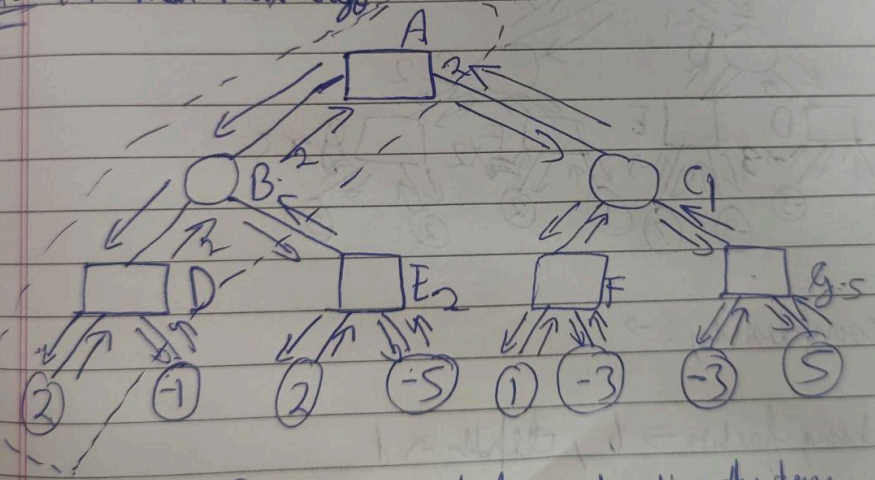
~~S1E~~

S1T → S → B → D → E chosen
S → B → D → E → G1 = 9

S1R → S → B → F → D → E → G1 = 8 (chosen)

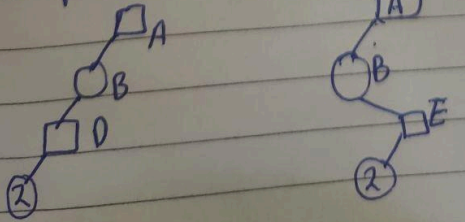
Goal node reached.
cost = 8
Path : S → B → F → D → E → G1

2) (a) Min-Max Algo



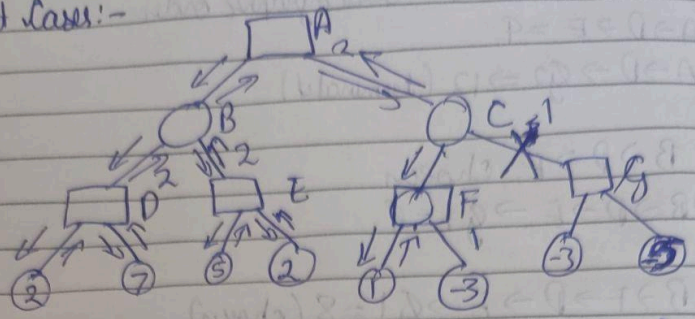
DFS is used for expanding the tree.
→ most convenient path.

Best possible ways/paths : →



Alpha-pruning Algo
Pruned nodes : → branches under E
(with value −5 is pruned)
branch under F
(with value −3 is pruned)
branch under G
(with value 5 is pruned)

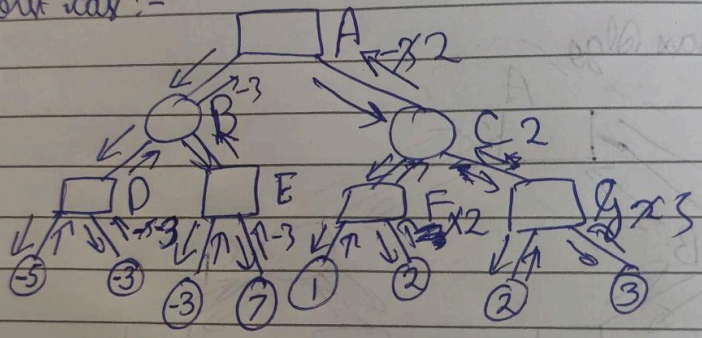
b) Best Case:-



(Ignoring / pruning unnecessary branches)

If we swap 2 & -5, 5 & 2, -5 to 5.
we get the best case:-

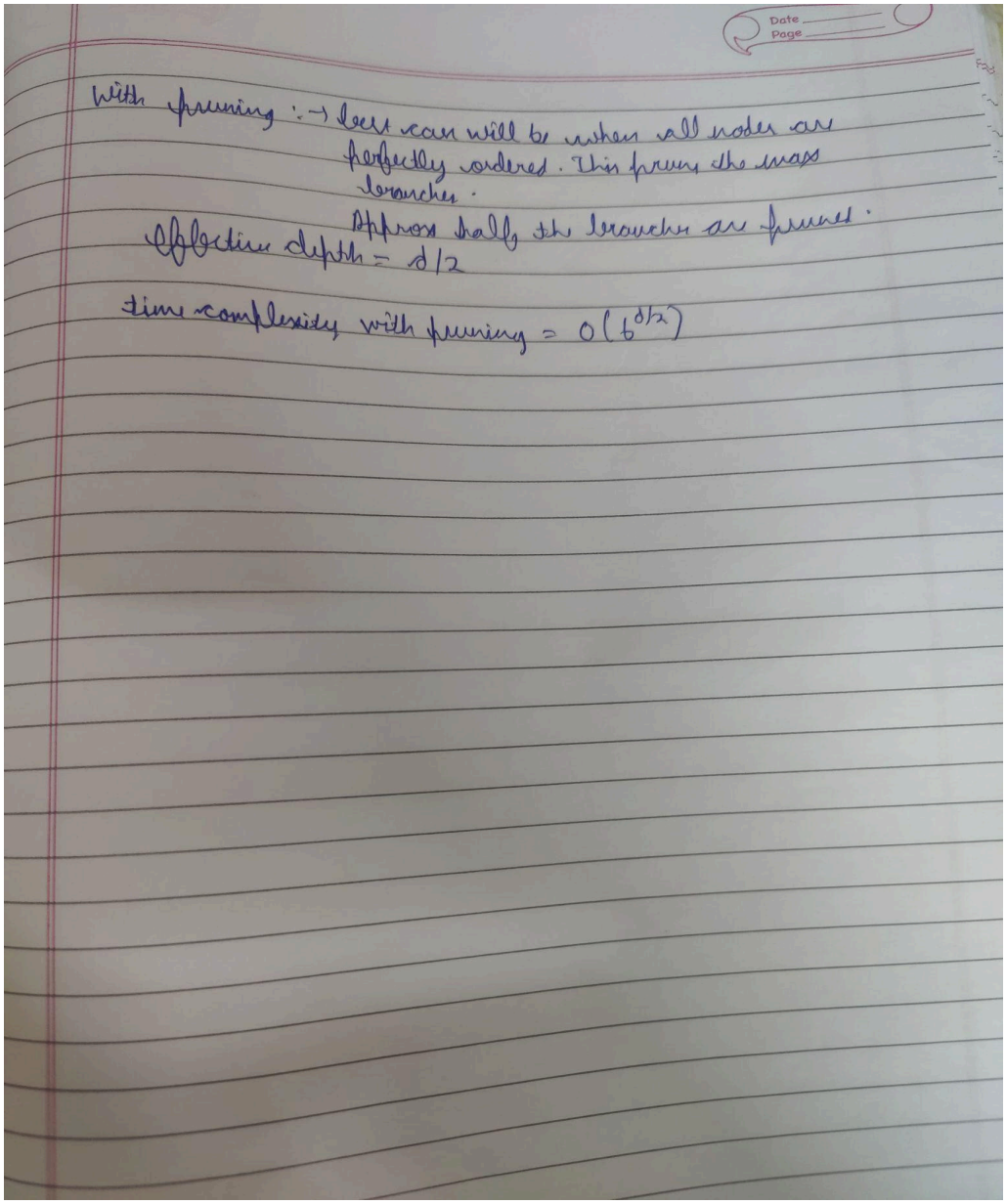
worst case:-



c) Time complexity :→

Branching factor → b, depth → d

Without pruning → $O(b^d)$ because each node has to be evaluated.



Q3

Comparison of Search Algorithms: IDS, Bidirectional, A*, and Bidirectional A*

In our testing of various search algorithms, we found that both the Iterative Deepening Search (IDS) and Bidirectional Search produced the same output paths for the first two test cases while also yielding no results (None) for another set of input nodes. Here's a closer look at what we found:

Test Results:

- **Test Case 1:**
  - **Start Node:** 1
  - **End Node:** 2
  - **IDS Output:** [1, 7, 6, 2]
  - **Bidirectional Output:** [1, 7, 6, 2]
- **Test Case 2:**
  - **Start Node:** 5
  - **End Node:** 12
  - **IDS Output:** [5, 97, 98, 12]
  - **Bidirectional Output:** [5, 97, 98, 12]
- **Test Case 3:**
  - **Start Node:** 12
  - **End Node:** 49
  - **IDS Output:** None
  - **Bidirectional Output:** None
- **Test Case 4:**
  - **Start Node:** 4
  - **End Node:** 12
  - **IDS Output:** [4, 6, 2, 9, 8, 5, 97, 98, 12]
  - **Bidirectional Output:** [4, 6, 2, 9, 8, 5, 97, 98, 12]

While it's interesting that IDS and Bidirectional Search aligned in these cases, it's important to note that this consistency may not hold true for all inputs. The main reason for differences can stem from how each algorithm functions:

- **IDS** operates by conducting a depth-limited search, incrementally increasing its depth limit. This approach might not always find the shortest path measured by the number of edges since it explores deeper layers first.
- **Bidirectional Search**, by starting from both the start and end nodes, usually discovers the shortest path more effectively since it considers nodes level by level.

Performance in Terms of Execution Time and Memory Usage:

- **IDS Execution Times and Memory Usage**
  - *Start: 1, End: 2* — Time: 0.00225 sec, Memory: 2064 bytes
  - *Start: 5, End: 12* — Time: 0.00200 sec, Memory: 2064 bytes
  - *Start: 4, End: 12* — Time: 0.01940 sec, Memory: 3195 bytes
- **Bidirectional Search Execution Times and Memory Usage**
  - *Start: 1, End: 2* — Time: 0.00121 sec, Memory: 3867 bytes
  - *Start: 5, End: 12* — Time: 0.00000 sec, Memory: 3995 bytes
  - *Start: 4, End: 12* — Time: 0.00341 sec, Memory: 7936 bytes

On average, **Bidirectional Search** is faster than **IDS**, but it generally requires more memory.

A* and Bidirectional A* Results

When we turned to the A* algorithm and its bidirectional variant, we observed some notable differences in the output paths:

- **Test Case 1:**
  - **Start Node:** 1
  - **End Node:** 2
  - *A Output:*[1, 27, 9, 2]
  - **Bidirectional Output:** [1, 27, 6, 2]
- **Test Case 2:**
  - **Start Node:** 5
  - **End Node:** 12
  - *A Output:*[5, 97, 28, 10, 12]
  - **Bidirectional Output:** [5, 97, 98, 12]
- **Test Case 3:**
  - **Start Node:** 4
  - **End Node:** 12
  - *A Output:*[4, 6, 27, 9, 8, 5, 97, 28, 10, 12]
  - **Bidirectional Output:** [4, 34, 33, 11, 32, 31, 3, 5, 97, 28, 10, 12]
- **Test Case 4:**
  - **Start Node:** 12
  - **End Node:** 49
  - *A Output:*None
  - **Bidirectional Output:** None

The variation in outputs highlights the distinct approaches of the algorithms:

- **A**follows a single-direction exploration towards the goal, driven by a cost function that combines the cost from the start (g(n)) and a heuristic estimate of the cost to the goal (h(n)).
- **Bidirectional A**explores from both ends, resulting in different paths combining two exploration strategies and more efficient intersection of the routes.

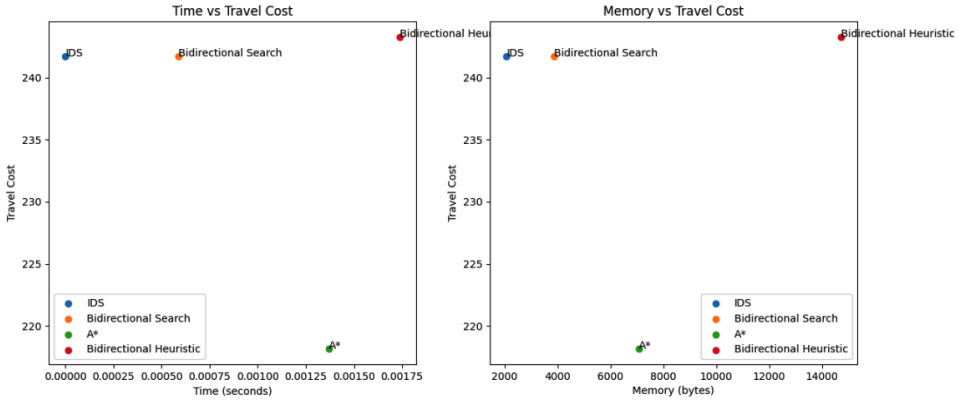
*Execution Time and Memory Usage for A Algorithms:**

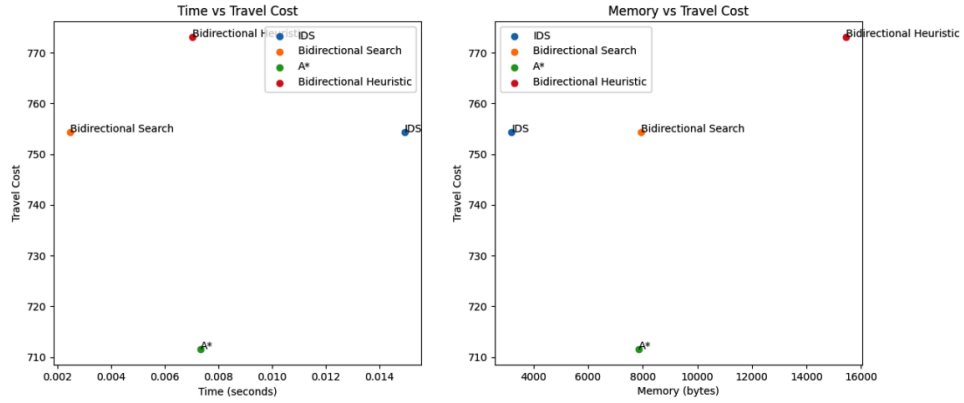
- **A**
  - *Start: 1, End: 2* — Time: 0.00105 sec, Memory: 7072 bytes
  - *Start: 5, End: 12* — Time: 0.00401 sec, Memory: 7960 bytes
  - *Start: 4, End: 12* — Time: 0.00446 sec, Memory: 7848 bytes
  - *Start: 12, End: 49* — Time: 0.02172 sec, Memory: 14664 bytes
- **Bidirectional A**
  - *Start: 1, End: 2* — Time: 0.00205 sec, Memory: 14704 bytes
  - *Start: 5, End: 12* — Time: 0.00829 sec, Memory: 15435 bytes
  - *Start: 4, End: 12* — Time: 0.00517 sec, Memory: 15435 bytes
  - *Start: 12, End: 49* — Time: 0.00213 sec, Memory: 14704 bytes

Overall, A* consistently outperforms Bidirectional A* in execution speed and memory efficiency, except in cases where no valid path exists; in these situations, both algorithms perform comparably in terms of memory usage and efficiency.

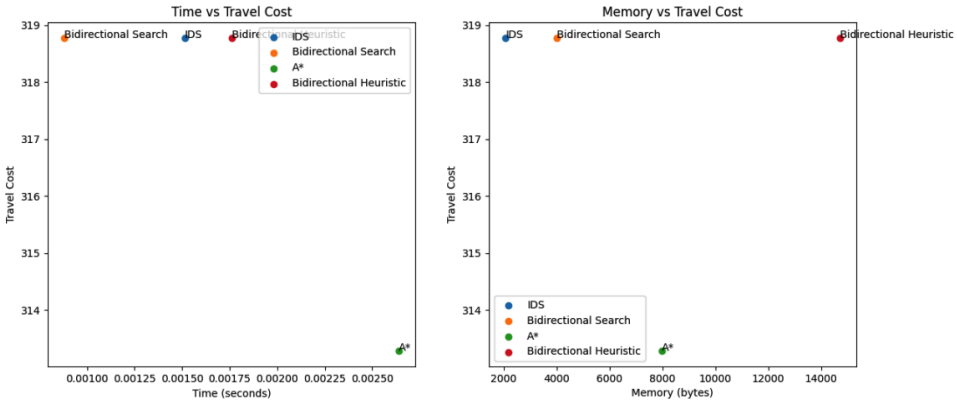
This analysis provides a nuanced understanding of how each search algorithm behaves under different conditions and highlights the trade-offs in terms of execution time and memory use.

Testcase-1:


Time vs Travel Cost
Travel Cost
Time (seconds)
IDS
Bidirectional Search
A*
Bidirectional Heuristic
Memory vs Travel Cost
Travel Cost
Memory (bytes)
IDS
Bidirectional Search
A*
Bidirectional Heuristic


Time vs Travel Cost
Travel Cost
Time (seconds)
IDS
Bidirectional Search
A*
Bidirectional Heuristic
Memory vs Travel Cost
Travel Cost
Memory (bytes)
IDS
Bidirectional Search
A*
Bidirectional Heuristic

Testcase-3:



I chose to use Travel Cost on the y-axis of the scatter plots because it allows us to visually analyze the trade-offs between time and memory usage for each algorithm relative to the travel cost. The travel cost, measured by the number of nodes traversed, is a crucial factor in evaluating the efficiency of search algorithms. By comparing performance based on this metric, we can better determine which algorithm is most suitable for various scenarios. This approach provides valuable insights into the effectiveness of each algorithm, making it a meaningful choice for the scatter plots.