AI Assignment 1 :-Search Algorithms

Parth Sandeep Rastogi

September 19, 2024

1 Q1

1.1 A* Search

Initial Frontier: $(f(n), g(n), \text{node}) = [(8, 0, S)]$

- Explored Node: S, $g(S) = 0$, $f(S) = 8$
  - **Frontier:** $(2, 1, B), (5, 3, A), (13, 5, C)$
- Explored Node: B, $g(B) = 1$, $f(B) = 2$
  - **Frontier:** $(5, 3, A), (6, 3, F), (9, 5, D), (13, 5, C), (13, 13, G3)$
- Explored Node: A, $g(A) = 3$, $f(A) = 5$
  - **Frontier:** $(6, 3, F), (9, 5, D), (13, 13, G1), (13, 5, C), (13, 13, G3)$
- Explored Node: F, $g(F) = 3$, $f(F) = 6$
  - **Frontier:** $(8, 4, D), (9, 5, D), (13, 13, G1), (13, 5, C), (13, 13, G3)$
- Explored Node: D, $g(D) = 4$, $f(D) = 8$
  - **Frontier:** $(7, 6, E), (9, 5, D), (9, 9, G2), (13, 5, C), (13, 13, G1), (13, 13, G3)$
- Explored Node: E, $g(E) = 6$, $f(E) = 7$
  - **Frontier:** $(8, 8, G1), (9, 9, G2), (13, 5, C), (13, 13, G1), (13, 13, G3)$
- Explored Node: G1, $g(G1) = 8$, $f(G1) = 8$ (Goal Found)

Path: S$\Longrightarrow B \Longrightarrow F \Longrightarrow D \Longrightarrow E \Longrightarrow G1$

1.2 IDA* Search Algorithm

Starting with initial bound: 8 IE heuristic of S
Steps :-

- **DFS Frontier at node S:** {S} , f(n)= 8
- **DFS Frontier at node A:** {S, A} , f(n) = 5
- **DFS Frontier at node G1:** {S, A, G1} , f(n) = 13
  → **Cutoff:** f(G1) > limit (8)
- **DFS Frontier at node D:** {S, A, D} , f(n)= 11
  → **Cutoff:** f(D) > limit (8)
- **DFS Frontier at node B:** {S, B} , f(n)= 2
- **DFS Frontier at node D:** {S, B, D} , f(n) = 9
  → **Cutoff:** f(D) > limit (8)
- **DFS Frontier at node F:** {S, B, F} , f(n) = 6
- **DFS Frontier at node D:** {S, B, F, D} , f(n)= 8
- **DFS Frontier at node E:** {S, B, F, D, E} , f(n) = 7
- **DFS Frontier at node G1:** {S, B, F, D, E, G1} , f(n) = 8
- **Goal Found:** Path {S, B, F, D, E, G1}

References taken for IDA* Algorithm implimentation

- GeeksForGeeks
- Algorithm Insights

1.3 Uniform Cost Search (UCS)

Initial Frontier: $(g(n), \text{node}) = [(0, S)]$

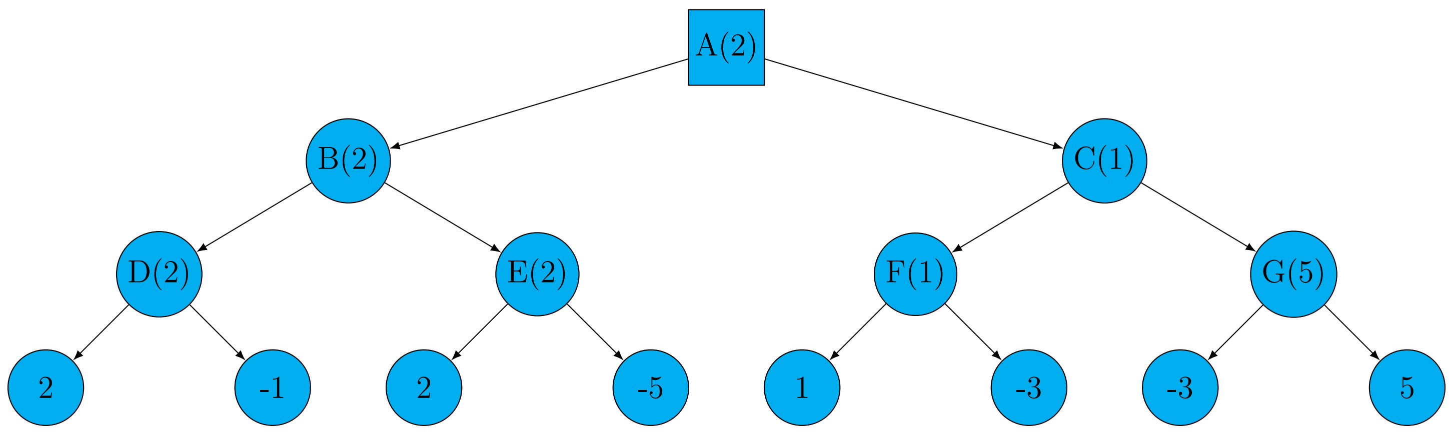
- Explored Node: S, $g(S) = 0$
  - **Frontier:** $(1, B), (3, A), (5, C)$
- Explored Node: B, $g(B) = 1$
  - **Frontier:** $(3, A), (3, F), (5, D), (5, C), (13, G3)$
- Explored Node: A, $g(A) = 3$
  - **Frontier:** $(3, F), (5, D), (13, G1), (5, C), (13, G3)$
- Explored Node: F, $g(F) = 3$
  - **Frontier:** $(4, D), (5, D), (13, G1), (5, C), (13, G3)$
- Explored Node: D, $g(D) = 4$
  - **Frontier:** $(5, C), (6, E), (9, G2), (13, G1), (13, G3)$
- Explored Node: E, $g(E) = 6$
  - **Frontier:** $(8, G1), (9, G2), (13, G1), (13, G3)$
- Explored Node: G1, $g(G1) = 8$ (Goal Found)

Path: S$\Longrightarrow B \Longrightarrow F \Longrightarrow D \Longrightarrow E \Longrightarrow G1$

2 Q2

2.1 Part A: Minimax Algorithm

The following game tree shows the Minimax algorithm determining the best play for both players, alternating between Max and Min nodes.



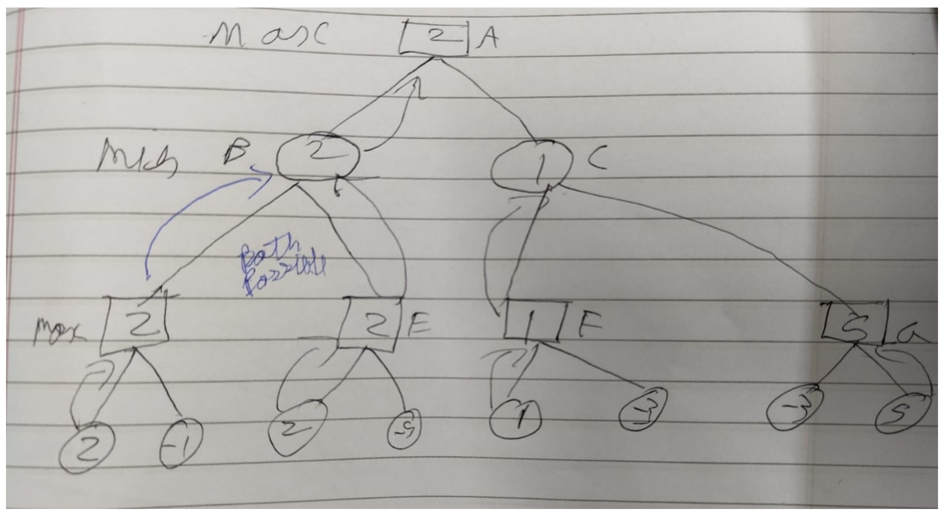
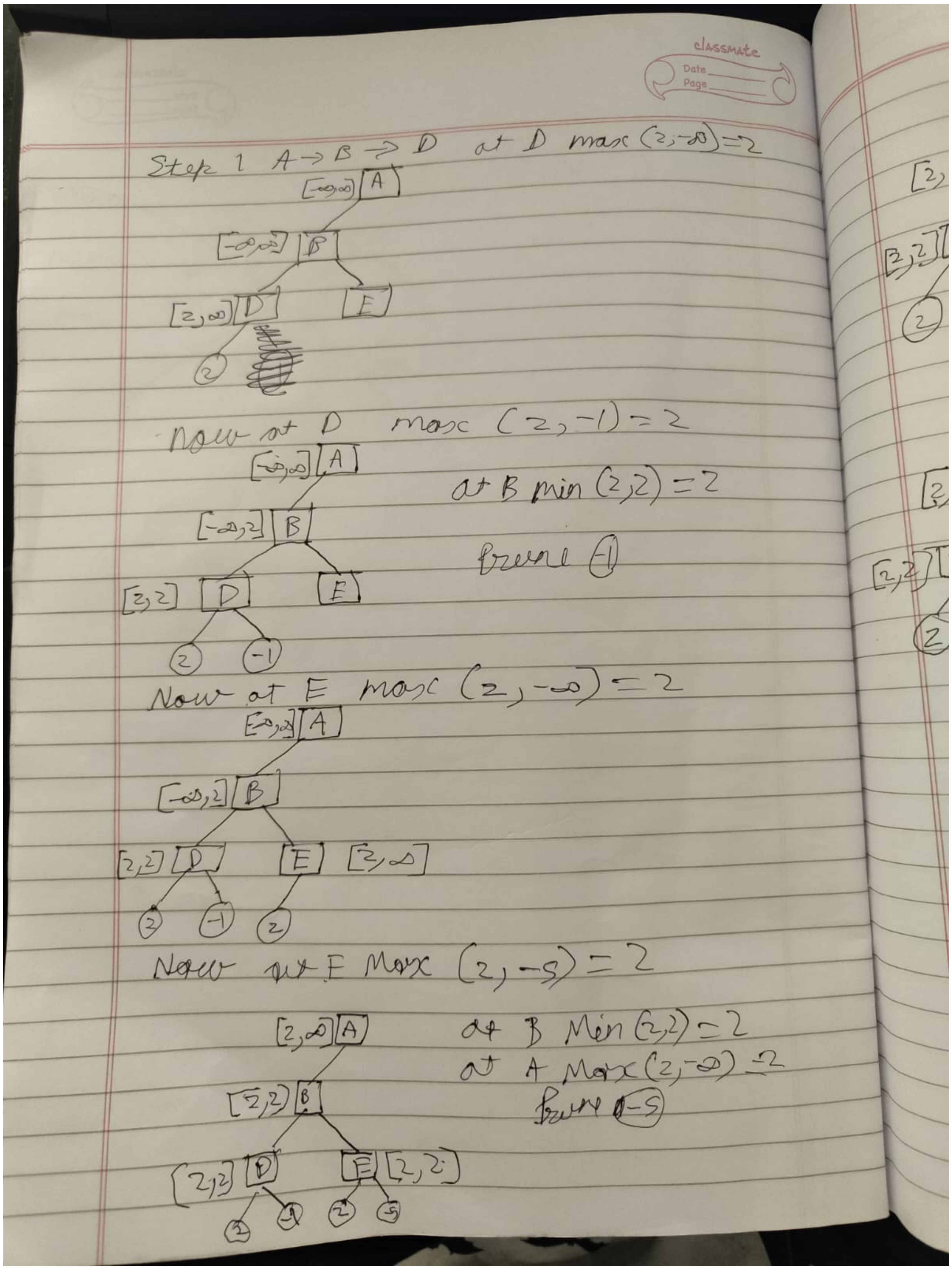


Figure 1: With arrows

From the tree:

- At the leaf nodes, the values are given.
- At the Max nodes, the higher value between the children is chosen:
  1. D=max(2,-1)=2
  2. E=max(2,-5)=2
  3. F=max(1,-3)=1
  4. G=max(-3,5)=5
- At the Min nodes, the lower value between the children is chosen:
  1. B=min(2,2)=2 So both D and E can be chosen
  2. C=min(1,5)=1
- Final result: Player A (Max) should choose node B, resulting in a score of 2.
- **Best Possible Paths:**
  - Path 1: A $\rightarrow$ B $\rightarrow$ D (resulting score = 2)
  - Path 2: A $\rightarrow$ B $\rightarrow$ E (resulting score = 2)

2.2 Part A: Alpha beta pruning



Step 1 $A \to B \to D$ at D $\max(2, -\infty) = 2$

Now at D $\max(2, -1) = 2$

at B $\min(2,2) = 2$

Prune -1

Now at E $\max(2, -\infty) = 2$

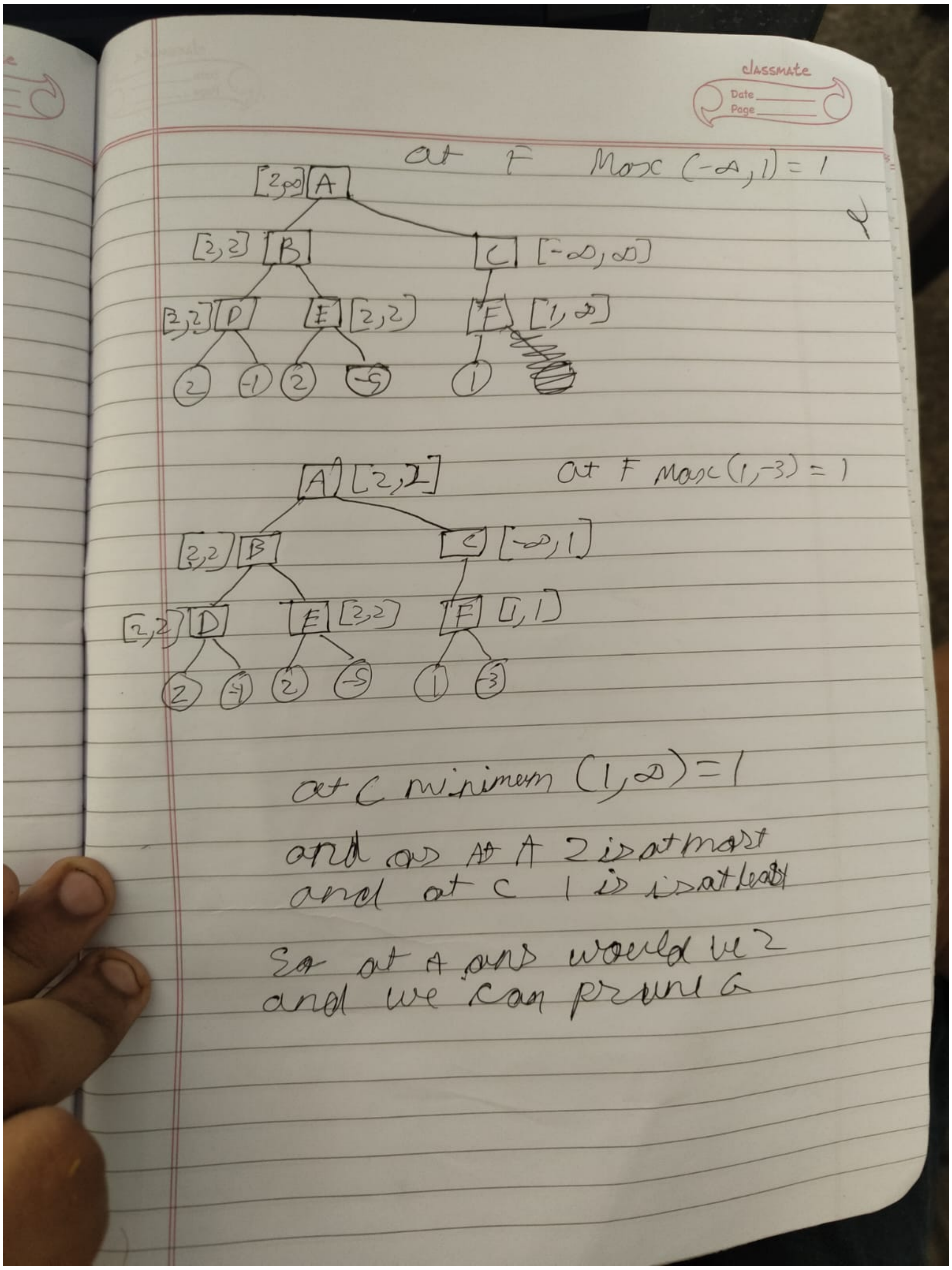
Now at E Max $(2, -5) = 2$

at B Min $(2,2) = 2$

at A Max $(2, -\infty) = 2$

Prune -5

Figure 2: Page1


at F Max$(-\infty, 1) = 1$

[2,∞] A
[2,2] B
C [-∞,∞]
[2,2] D
E [2,2]
F [1,∞]
2
-1
2
-5
1

at F Max$(1, -3) = 1$

A [2,2]
[2,2] B
C [-∞,1]
[2,2] D
E [2,2]
F [1,1]
2
-1
2
-5
1
-3

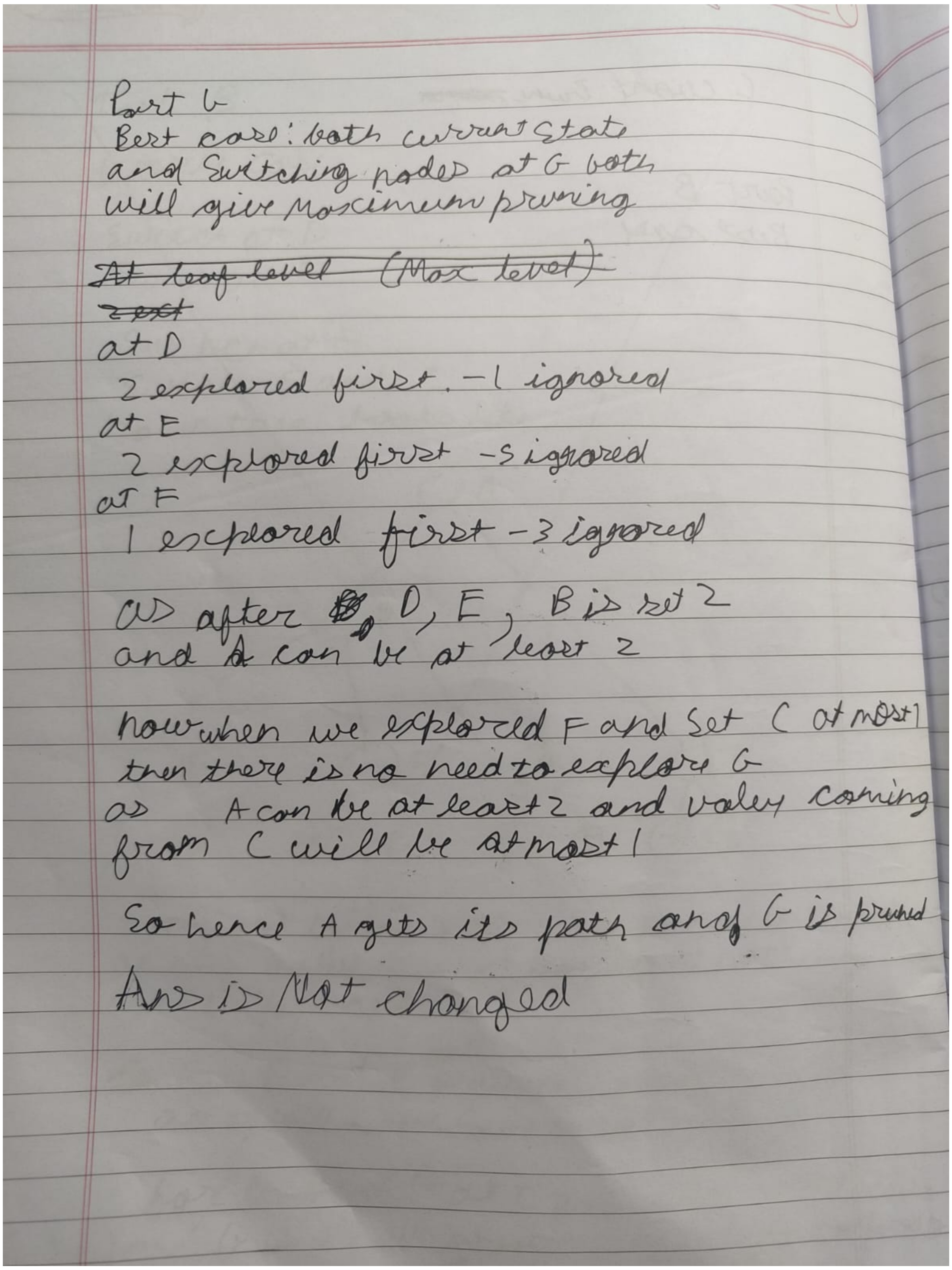
at C minimum $(1, \infty) = 1$

and as At A 2 is atmost
and at C 1 is is atleast

So at A ans would be 2
and we can prune G

Figure 3: Page2

2.3 Part B: MAX and MIN pruning



Part b
Best case: both current State
and Switching nodes at G both
will give Maximum pruning

~~At leaf level (Max level)~~
[illegible]
at D
2 explored first. -1 ignored
at E
2 explored first -5 ignored
at F
1 explored first -3 ignored

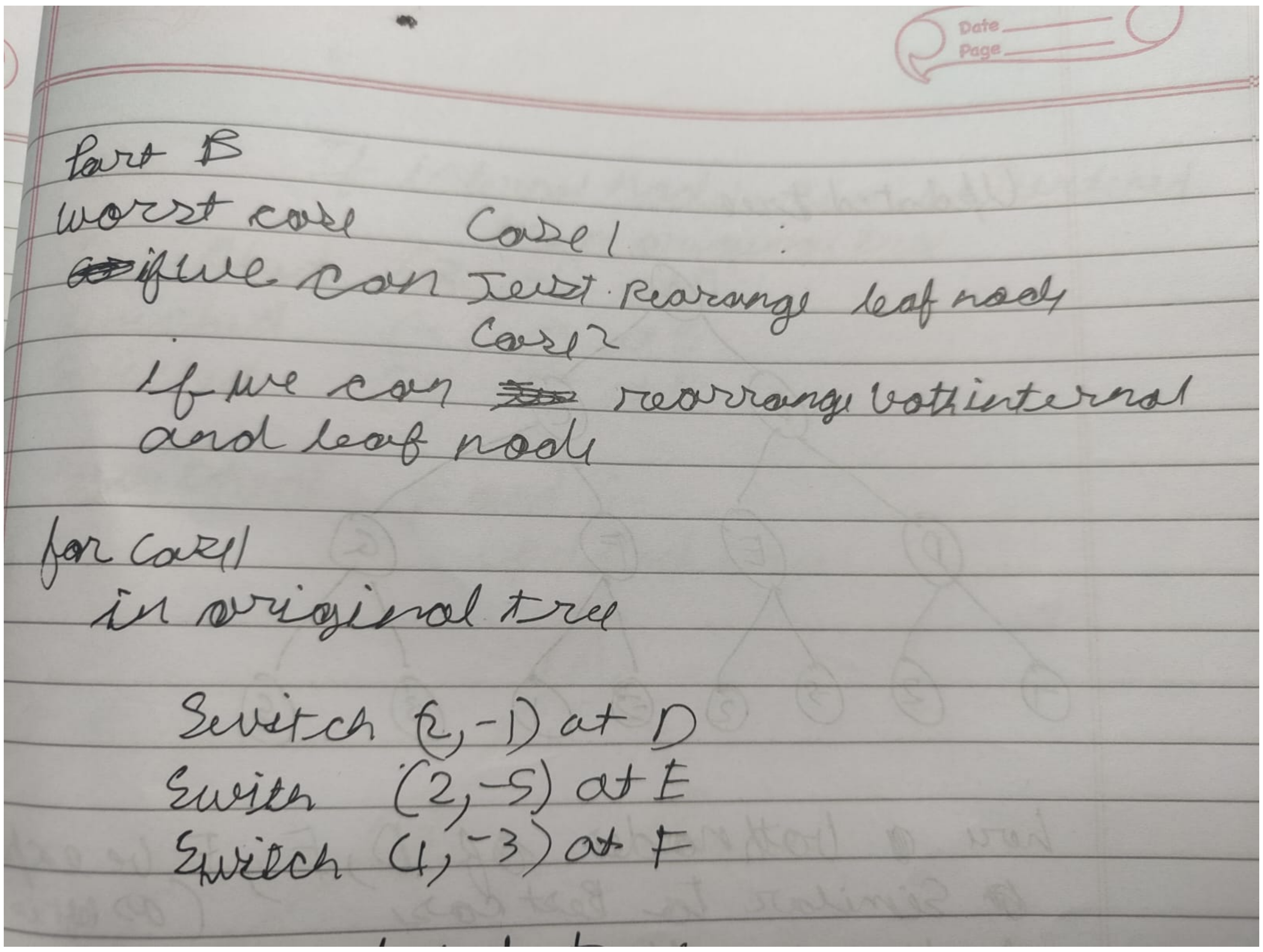
as after ~~B~~, D, E, B is set 2
and A can be at least 2

now when we explored F and Set C at most 1
then there is no need to explore G
as A can be at least 2 and valey coming
from C will be at most 1

So hence A gets its path and G is pruned

Ans is Not changed

Figure 4: Best Case



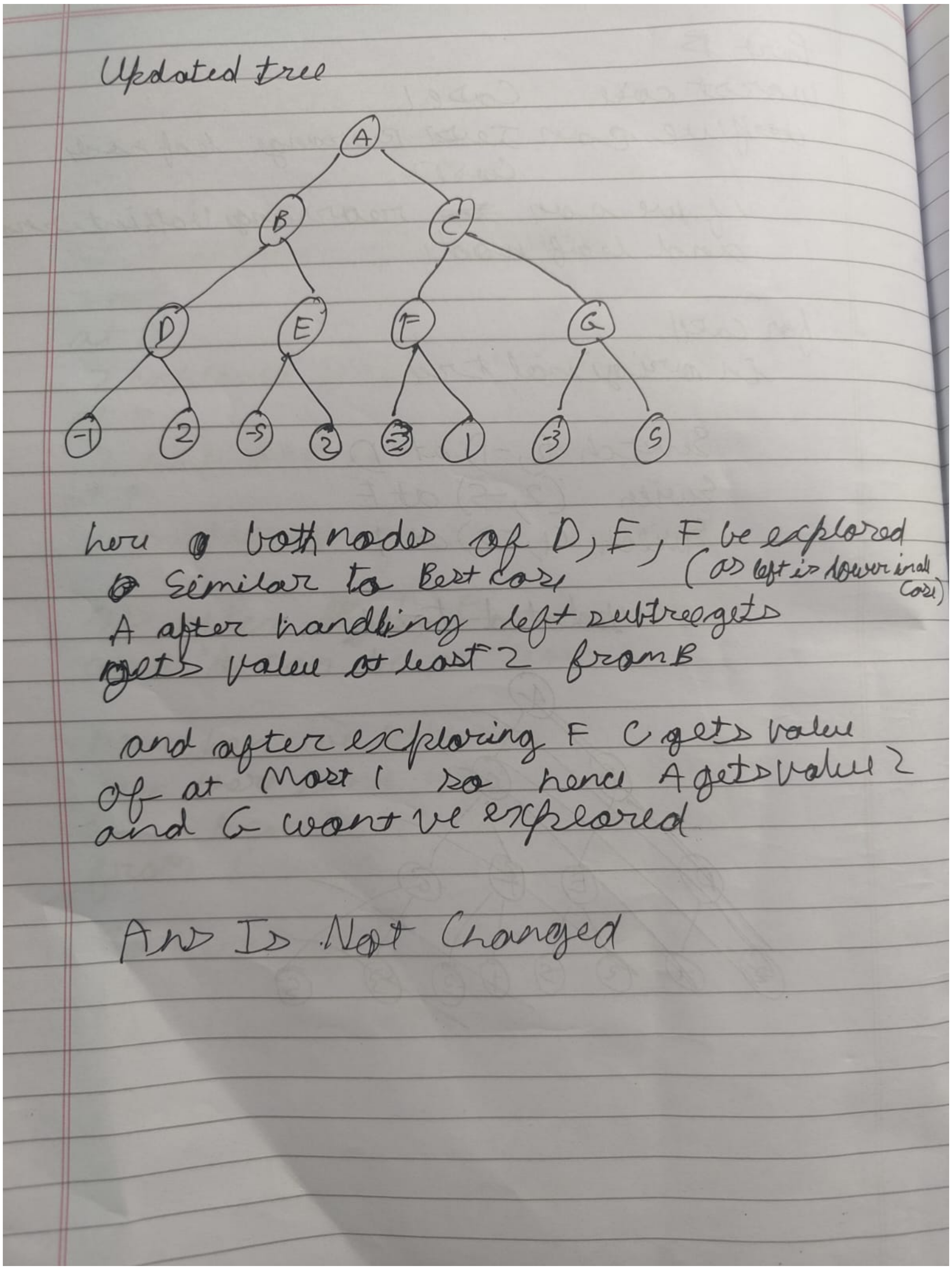
Part B
worst case Case 1
~~[illegible]~~ if we can Just Rearrange leaf node
Case 2
if we can ~~[illegible]~~ rearrange both internal and leaf node

for case 1
in original tree

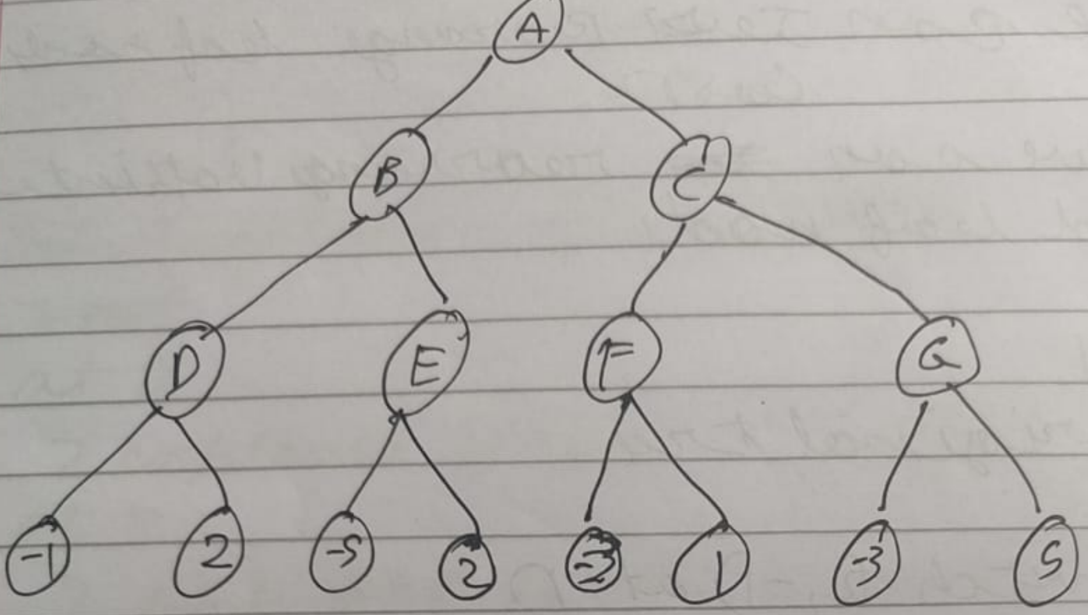
Switch (2, -1) at D
Switch (2, -5) at E
Switch (1, -3) at F

Figure 5: Worst Case 1

2.4 Part C



Updated tree

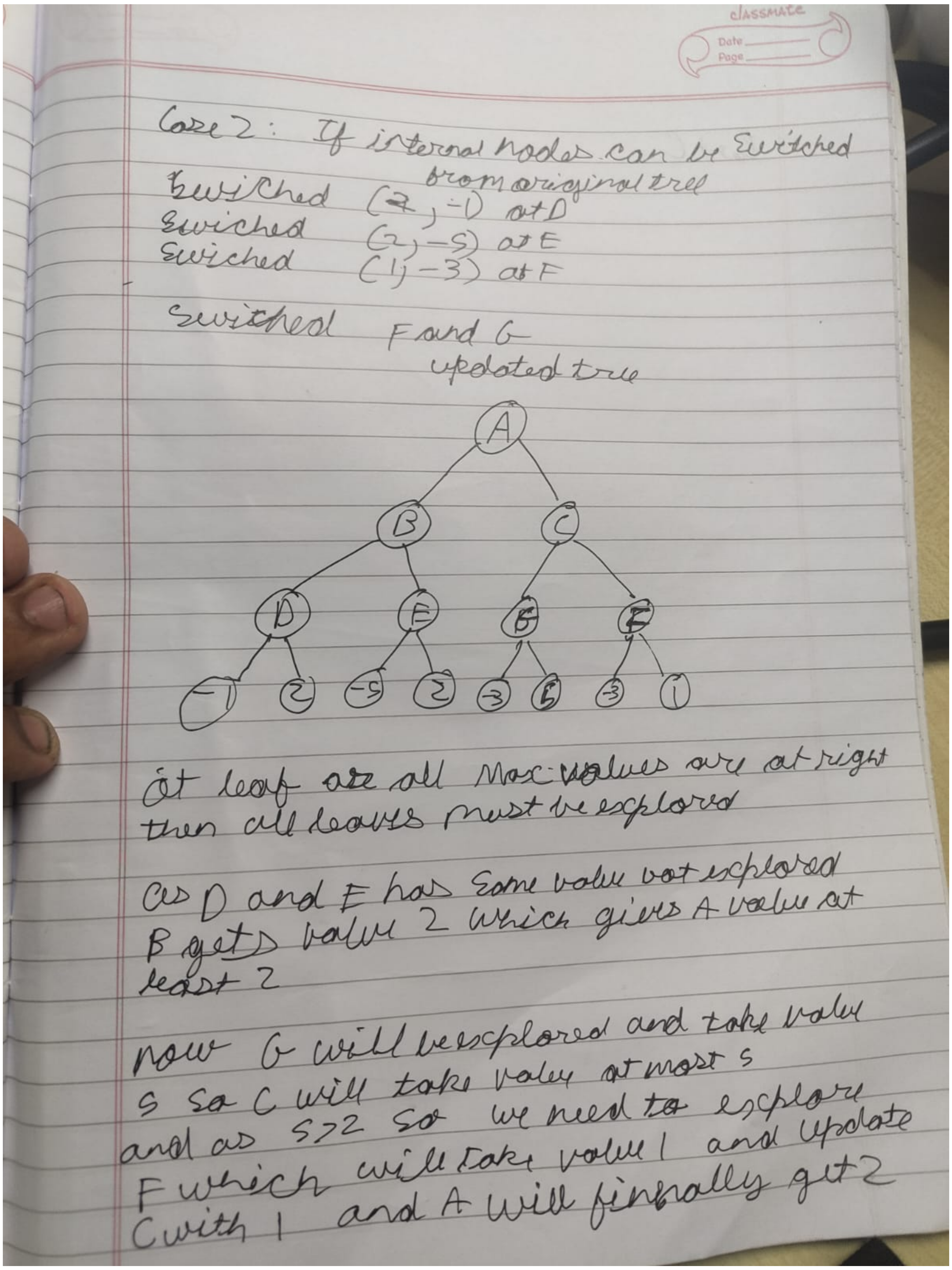


here both nodes of D, E, F be explored
Similar to Best case, (as left is lower in all case)
A after handling left subtree gets
gets value at least 2 from B

and after exploring F C gets value
of at most 1 so hence A gets value 2
and G wont be explored

Ans Is Not Changed

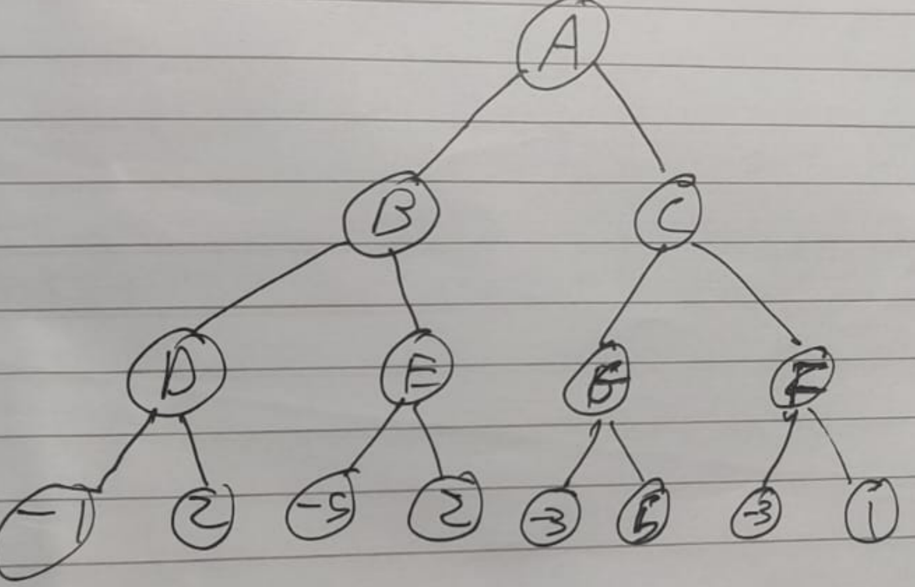
Figure 6: Worst Case 2



Case 2: If internal nodes can be Switched from original tree

Swiched (2, -1) at D
Swiched (2, -5) at E
Swiched (1, -3) at F

Switched F and G
updated tree



at leaf ~~are~~ all Max values are at right then all leaves must be explored

as D and E has same value not explored B gets value 2 which gives A value at least 2

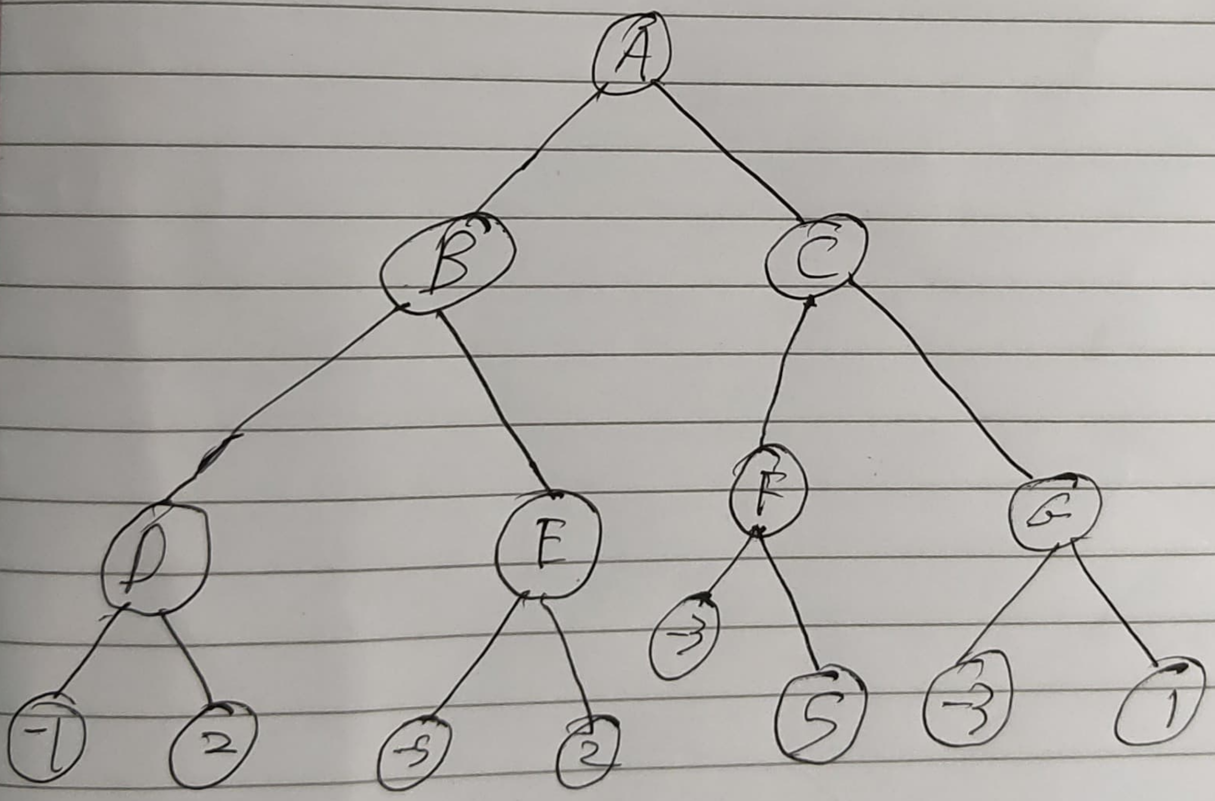
now G will be explored and take value 5 so C will take value at most 5 and as 5>2 so we need to explore and update F which will take value 1 and update C with 1 and A will finally get 2

Figure 7: Worst Case 3

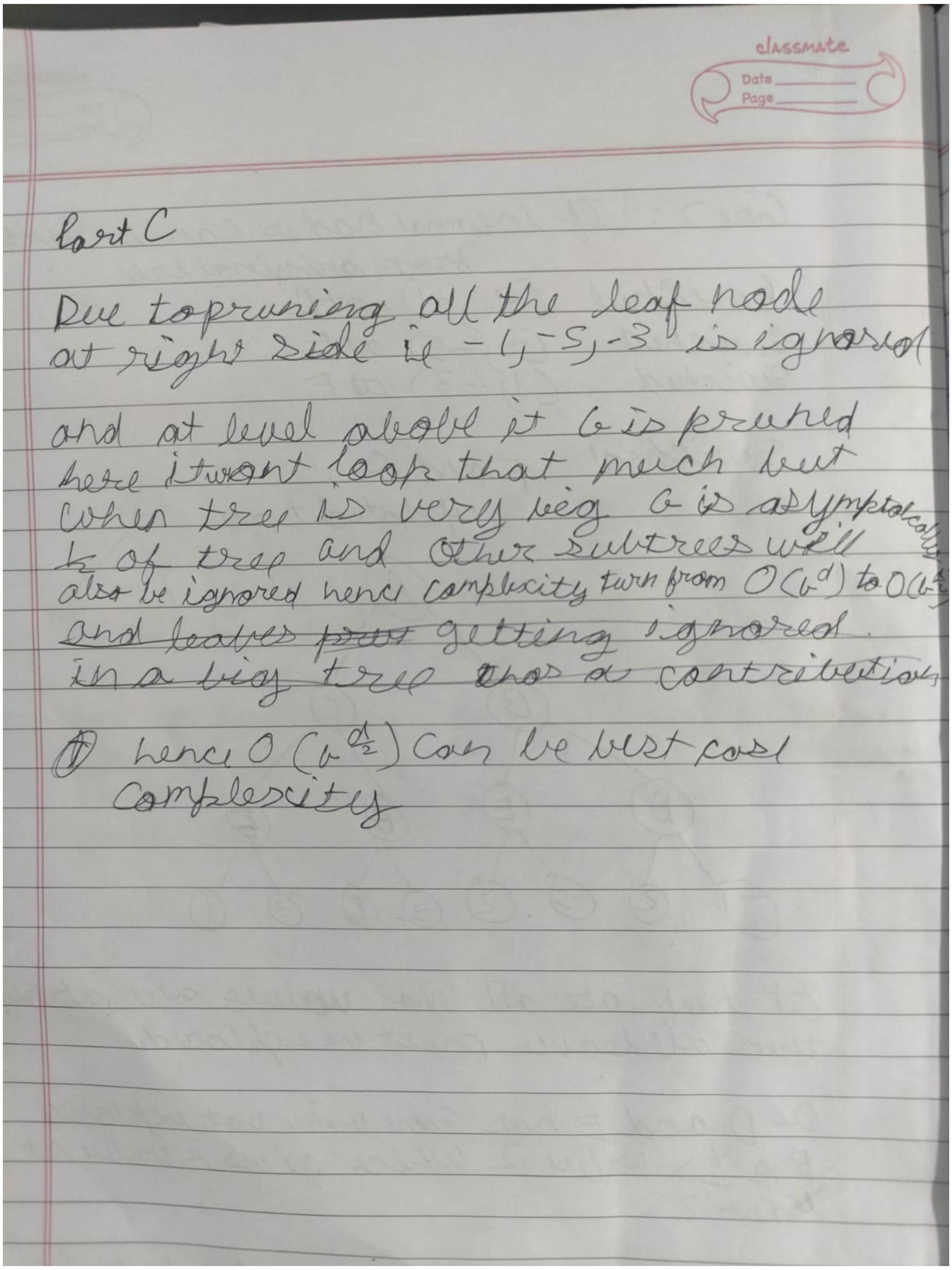
Case 3
where we can rearrange leaves in any order (even changing the parent)

Switch (2, -1) at d
Switch (2, -5) at E
switch 1 from children of F and ~~5~~ 5 from children of G and at end swich ~~[illegible]~~ 5 and 3 at children of F

final tree



at leaf level all nodes will be explored as D and E get Same value both explored and B gets value 2 which give value at least 2 to A
Now F will be explored and get value 5 C would be at max 5 which is More than min value at A ~~So~~ G will be explored and take value 1 which will give value 1 to C And A remain same 2


classmate
Date
Page

Part C

Due topruning all the leaf node at right side ie $-1, -5, -3$ is ignored

and at level above it G is pruned here it want look that much but when tree is very big G is asymptotically $\frac{1}{2}$ of tree and Other subtrees will also be ignored hence complexity turn from $O(b^d)$ to $O(b^{\frac{d}{2}})$ and leaves ~~pras~~ getting ignored. in a big tree thus a contribution

hence $O(b^{\frac{d}{2}})$ Can be best case complexity

Figure 9: Part C

3 Q3

3.1 Part A

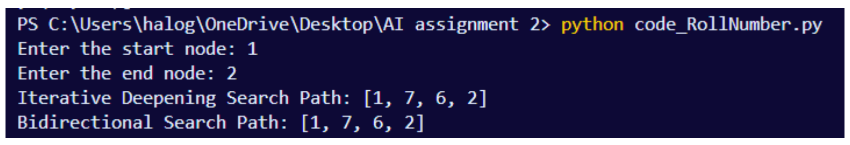
I have attached the code in the zip

3.2 Part B Analysis of Path Consistency Between Bidirectional BFS and Iterative Deepening Search

For each public test case, we analyzed the path obtained for traveling from the source node u to the destination node v using both the Bidirectional Breadth-First Search (Bidirectional BFS) and Iterative Deepening Search (IDS) algorithms. In all instances, the paths generated by both algorithms were found to be identical.

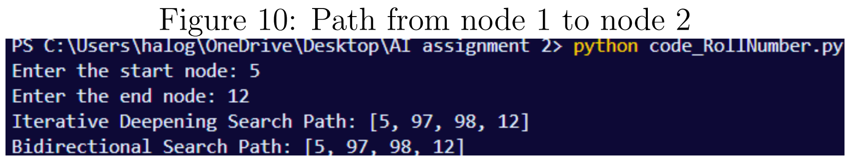
3.2.1 Test Cases Overview

The following images illustrate the paths obtained for several test cases, with both Bidirectional BFS and IDS yielding the same result for each case:



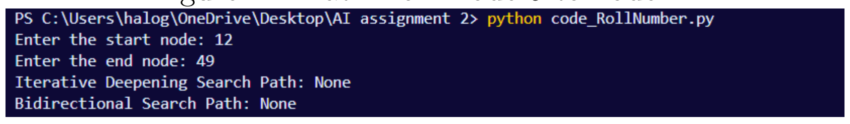
```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python code_RollNumber.py
Enter the start node: 1
Enter the end node: 2
Iterative Deepening Search Path: [1, 7, 6, 2]
Bidirectional Search Path: [1, 7, 6, 2]
```

Figure 10: Path from node 1 to node 2



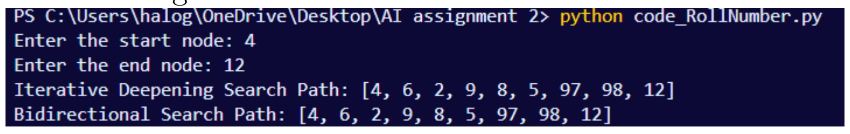
```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python code_RollNumber.py
Enter the start node: 5
Enter the end node: 12
Iterative Deepening Search Path: [5, 97, 98, 12]
Bidirectional Search Path: [5, 97, 98, 12]
```

Figure 11: Path from node 5 to node 12



```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python code_RollNumber.py
Enter the start node: 12
Enter the end node: 49
Iterative Deepening Search Path: None
Bidirectional Search Path: None
```

Figure 12: Path from node 12 to node 49



```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python code_RollNumber.py
Enter the start node: 4
Enter the end node: 12
Iterative Deepening Search Path: [4, 6, 2, 9, 8, 5, 97, 98, 12]
Bidirectional Search Path: [4, 6, 2, 9, 8, 5, 97, 98, 12]
```

Figure 13: Path from node 4 to node 12

Figure 14: Test cases demonstrating identical paths between Bidirectional BFS and IDS

3.2.2 Simultaneous Detection

In all test case, detected the same path. This means that both search strategies converged on the same path, further reinforcing the consistency in performance for these two algorithms when applied to the same graph.

3.2.3 Will the Paths Always Be Identical?

In the case of unweighted graphs, the path lengths obtained from Bidirectional BFS and IDS will always be identical, as both algorithms are designed to find the shortest path. However, the actual paths they return may differ in structure. This is because multiple shortest paths of the same length can exist between two nodes in a graph, and the traversal order of the algorithms can lead to different results:

- **Bidirectional BFS**: By simultaneously exploring from both the source and destination, Bidirectional BFS guarantees that a shortest path is found as soon as the two search fronts meet. The path returned may depend on how the meeting point between the two fronts is chosen and the structure of the graph.
- **IDS**: By progressively deepening the search limit, IDS ensures that the shortest path is found once the depth limit encompasses the distance between the source and destination nodes. The path returned may depend on the depth limit and the structure of the graph.

Hence, for any pair of nodes in an unweighted graph G, while the paths obtained by these two algorithms will always be of the same length, the exact paths may differ due to the different methods of traversal. In cases involving weighted graphs or graphs with other properties, the behavior of these algorithms may differ further, and additional analysis would be required to assess the paths.

3.3 Part C) Comparison of Memory and Time Performance

In this subsection, we compare the peak memory usage and execution time of different search algorithms, specifically Iterative Deepening Search (IDS) and Bidirectional Search. The results are summarized below:

Algorithm	**Total Time (s)**	**Peak Memory (bytes)**
Iterative Deepening Search (IDS)	832.889	7456
Bidirectional Search	81.775	70760

Table 1: Comparison of Total Time and Peak Memory Usage for IDS and Bidirectional Search

Memory Efficiency Iterative Deepening Search (IDS) is designed to be memory-efficient by exploring nodes up to a certain depth and discarding them afterward. Consequently, its peak memory usage is relatively low compared to Bidirectional Search. IDS uses 7456 bytes of memory, whereas Bidirectional Search requires significantly more, at 70760 bytes. This results in Bidirectional Search consuming approximately 9.5 times more memory than IDS.

Time Performance The execution time of each algorithm varies considerably. IDS takes 832.889 seconds to complete, while Bidirectional Search performs the same task in 81.775 seconds. Bidirectional Search is faster by a factor of approximately 10.2 times compared to IDS. This speed advantage is due to Bidirectional Search's approach of simultaneously exploring from both the start and goal nodes, reducing the effective search space more quickly.

Impact of Memory and Time Performance

- **IDS:** Although IDS is more memory-efficient, its execution time is longer. It is suitable for applications where memory constraints are critical but where longer search times are acceptable.
- **Bidirectional Search:** This algorithm provides faster search times but at the cost of higher memory usage. It is ideal for scenarios where execution speed is a priority and sufficient memory resources are available.

Trade-offs and Practical Considerations The choice between IDS and Bidirectional Search involves balancing memory usage and execution time:

- **IDS:** Best suited for environments with limited memory, where the efficiency of memory usage outweighs the need for faster execution times.
- **Bidirectional Search:** Preferred when the speed of finding a solution is more critical than memory usage. It is effective in systems where memory is not a limiting factor.

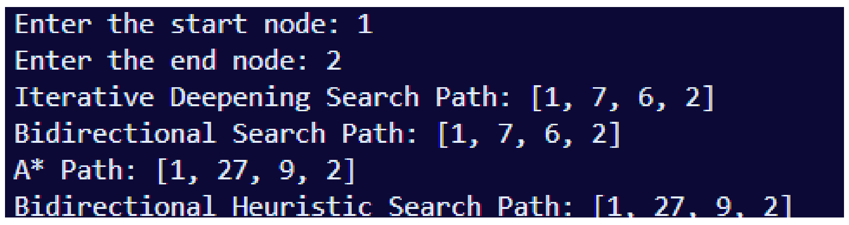
In summary, the comparison of both peak memory usage and execution time provides insights into the trade-offs between memory efficiency and performance for different search algorithms. Selecting the appropriate algorithm depends on the specific requirements of memory constraints and performance goals in a given application.

3.4 Part E Analysis of Path Consistency Between A* and Bidirectional A*

For each public test case, we analyzed the path obtained for traveling from the source node u to the destination node v using both the A* and Bidirectional A* algorithms. Similar to the previous analysis, we observed that in all instances, the paths generated by both algorithms were identical.

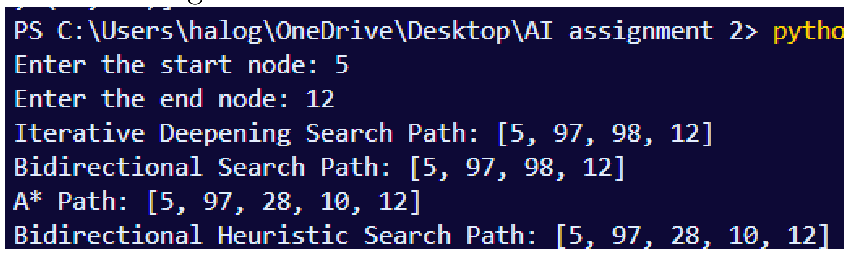
3.4.1 Test Cases Overview

The following images illustrate the paths obtained for several test cases, with both A* and Bidirectional A* yielding the same result for each case:



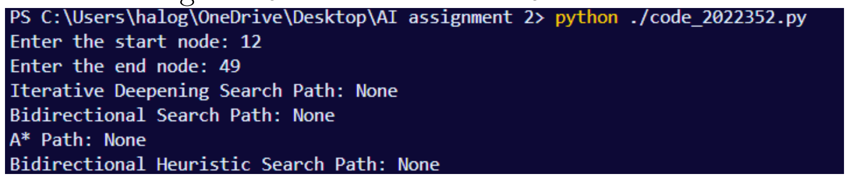
```
Enter the start node: 1
Enter the end node: 2
Iterative Deepening Search Path: [1, 7, 6, 2]
Bidirectional Search Path: [1, 7, 6, 2]
A* Path: [1, 27, 9, 2]
Bidirectional Heuristic Search Path: [1, 27, 9, 2]
```

Figure 15: Path from node 1 to node 2



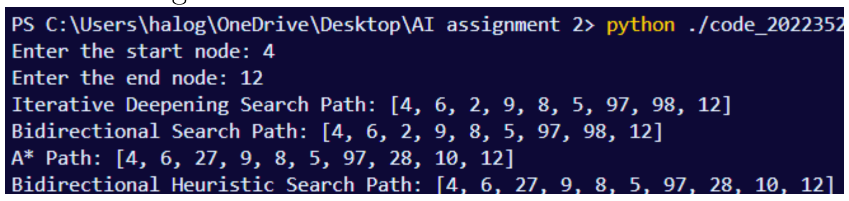
```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> pytho
Enter the start node: 5
Enter the end node: 12
Iterative Deepening Search Path: [5, 97, 98, 12]
Bidirectional Search Path: [5, 97, 98, 12]
A* Path: [5, 97, 28, 10, 12]
Bidirectional Heuristic Search Path: [5, 97, 28, 10, 12]
```

Figure 16: Path from node 5 to node 12



```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python ./code_2022352.py
Enter the start node: 12
Enter the end node: 49
Iterative Deepening Search Path: None
Bidirectional Search Path: None
A* Path: None
Bidirectional Heuristic Search Path: None
```

Figure 17: Path from node 12 to node 49



```
PS C:\Users\halog\OneDrive\Desktop\AI assignment 2> python ./code_2022352
Enter the start node: 4
Enter the end node: 12
Iterative Deepening Search Path: [4, 6, 2, 9, 8, 5, 97, 98, 12]
Bidirectional Search Path: [4, 6, 2, 9, 8, 5, 97, 98, 12]
A* Path: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12]
Bidirectional Heuristic Search Path: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12]
```

Figure 18: Path from node 4 to node 12

Figure 19: Test cases demonstrating identical paths between A* and Bidirectional A*

3.4.2 Simultaneous Detection

In all test cases, A* and Bidirectional A* detected the same path. This indicates that both algorithms converged on the same optimal solution, further demonstrating their consistency when applied to the same graph with similar conditions.

3.4.3 Will the Paths Always Be Identical?

In the case of the given test cases, the paths obtained from A* and Bidirectional A* were identical due to the heuristic function used being very small or close to zero. This resulted in the heuristic playing a minimal role, causing both algorithms to behave similarly to a uniform cost search and thus finding the same path But this might not always be the case as mostly

Specifically:

- **A***: The A* algorithm uses a heuristic to guide the search towards the goal, ensuring that the first path found is the shortest path when the heuristic is admissible (i.e., it never overestimates the true cost).
- **Bidirectional A***: By simultaneously expanding from both the source and destination using a similar heuristic, Bidirectional A* meets in the middle and guarantees the shortest path as soon as the two search fronts converge.

In this case, since the heuristic values were small, they did not significantly influence the search process, causing both algorithms to effectively behave like breadth-first search algorithms, thus leading to identical paths. However, in cases where more complex or diverse heuristic functions are used, or in weighted graphs, the paths may differ.

3.5 Part E)Comparison of Memory and Time Performance: Bidirectional Heuristic vs. A*

In this subsection, we compare the peak memory usage and execution time of the Bidirectional Heuristic and A* algorithms. The results are summarized in Table 2.

Algorithm	Total Time (s)	Peak Memory (bytes)
A*	131.079	11442
Bidirectional Heuristic	29.089	49440

Table 2: Comparison of Total Time and Peak Memory Usage for Bidirectional Heuristic and A*

Memory Efficiency Bidirectional Heuristic algorithm uses significantly more peak memory compared to A*. Specifically, Bidirectional Heuristic has a peak memory usage of 49440 bytes, whereas A* uses 11442 bytes. This indicates that Bidirectional Heuristic requires approximately 4.3 times more memory than A*.

Causes for Memory Differences The higher memory consumption in Bidirectional Heuristic is due to several factors:

- **Simultaneous Searches:** Bidirectional Heuristic performs searches from both the start and goal nodes simultaneously, necessitating the storage of nodes and heuristic information for both search directions.
- **Heuristic Storage:** It maintains separate data structures to store the heuristic values and node states for both directions, contributing to increased memory usage.

In contrast, A* performs a single search from the start node to the goal node, requiring less memory to store nodes and path information.

Time Performance In terms of execution time, Bidirectional Heuristic is considerably faster than A*. The Bidirectional Heuristic completes the search in 29.089 seconds, while A* takes 131.079 seconds. This means Bidirectional Heuristic is approximately 4.5 times faster than A*.

Causes for Time Differences The faster performance of Bidirectional Heuristic can be attributed to:

- **Reduced Search Space:** By searching from both the start and goal nodes, Bidirectional Heuristic reduces the effective search space more quickly, leading to faster convergence.
- **Efficient Meeting Point:** The simultaneous search approaches meet in the middle, reducing the amount of exploration needed compared to A*, which searches from one direction only.

On the other hand, A* explores the entire path space from start to goal, leading to longer search times, although it is designed to find the optimal path efficiently.

3.6 Part F) Analysis of Search Algorithms

3.6.1 Metrics for Efficiency and Optimality

The scatter plots were generated using the following metrics:

- **Total Time (s)**: The time taken by each algorithm to find the solution.
- **Peak Memory (bytes)**: The maximum memory consumed during execution.
- **Average Path Cost**: The cost associated with the path found by the algorithm, representing optimality.

Efficiency: The efficiency of each algorithm is compared using total time and peak memory. Informed search algorithms (A*, Bidirectional Heuristic) show better time efficiency but may consume more memory, while uninformed search algorithms (IDS, Bidirectional Search) are slower but sometimes more memory-efficient.

Optimality: Optimality is measured using the average path cost. The results show that informed search algorithms (A*, Bidirectional Heuristic) yield lower path costs, indicating better optimality. Uninformed algorithms (IDS, Bidirectional Search) typically have higher path costs due to their lack of heuristic guidance and depth-first focus.

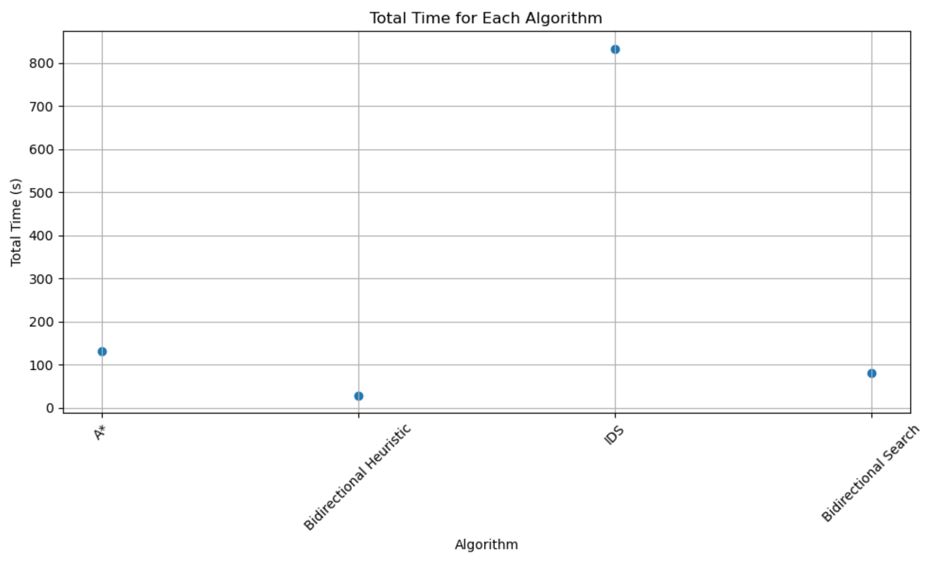


Figure 20: Time

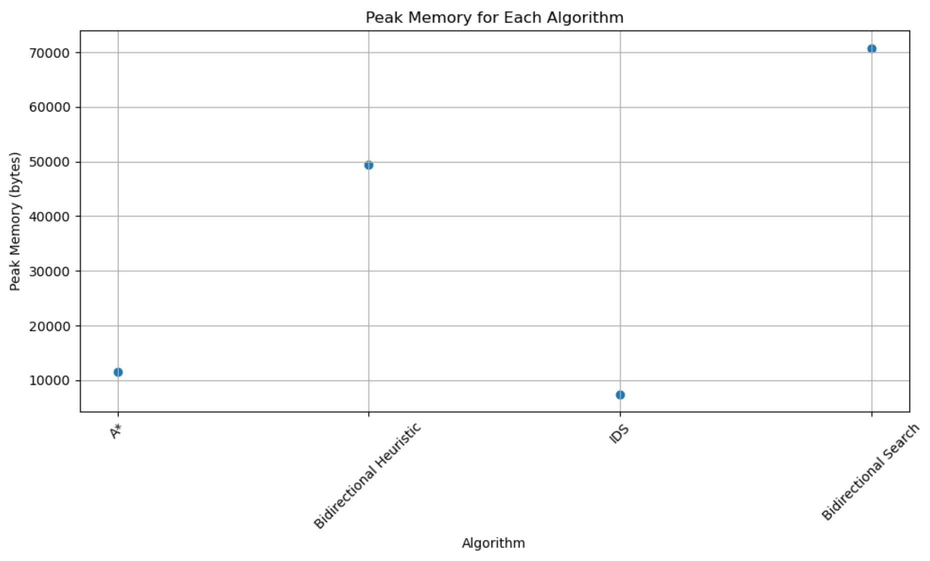


Figure 21: Space

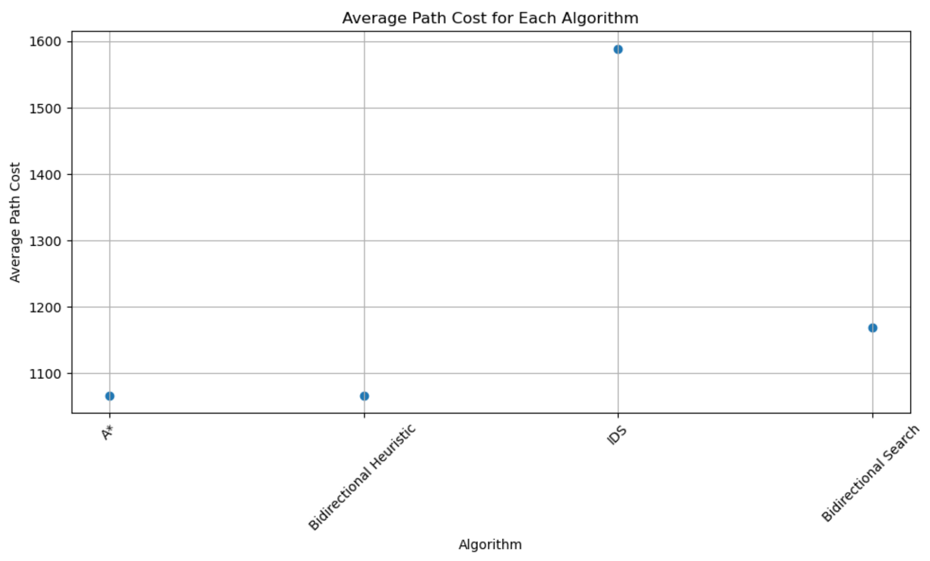


Figure 22: Average Path

3.6.2 Comparison of Informed and Uninformed Search

Informed Search Algorithms (A*, Bidirectional Heuristic) :

- **Benefits**: They achieve lower path costs, indicating optimal solutions. They are faster due to the use of heuristics (A*) or bidirectional search strategies (Bidirectional Heuristic).
- **Drawbacks**: They may consume more memory, especially Bidirectional Heuristic, which has the highest peak memory usage.

Uninformed Search Algorithms (IDS, Bidirectional Search) :

- **Benefits**: They are often more memory-efficient, particularly IDS. Simpler algorithms that do not rely on heuristics.
- **Drawbacks**: They result in higher path costs and are generally slower due to the exhaustive search nature (IDS) or depth-first approach (Bidirectional Search).

3.6.3 Conclusion

Informed search algorithms are generally more efficient and optimal, producing faster results with lower path costs, but they often require more memory. Uninformed search algorithms, while more memory-efficient, tend to have higher path costs and longer execution times. The choice between informed and uninformed search depends on whether memory or time efficiency is prioritized, as well as the importance of optimality in the solution.

3.7 Bonus :Identification of Vulnerable Roads Using Tarjan's Algorithm

To identify vulnerable roads in the network, I implemented Tarjan's algorithm. This algorithm is designed to find all the *bridges* (or *cut edges*) in a graph. A **bridge** is an edge whose removal increases the number of connected components in the graph, making it a critical road in this context.

The algorithm works by performing a depth-first search (DFS) on the graph, keeping track of the discovery time and the lowest point each vertex can reach. If an edge (u, v) satisfies the condition:

$$low[v] > disc[u]$$

then the edge is identified as a bridge.

3.7.1 Implementation Details

I applied Tarjan's algorithm to the road network graph G. The algorithm explores all vertices and edges in $O(V + E)$ time complexity, where V is the number of vertices and E is the number of edges. For each edge, I computed its impact on the graph's connectivity and identified the critical edges, which represent the vulnerable roads.

After running the algorithm, I found a total of **35 cut edges**. These edges are vulnerable roads, and their removal would increase the number of disconnected components in the network.