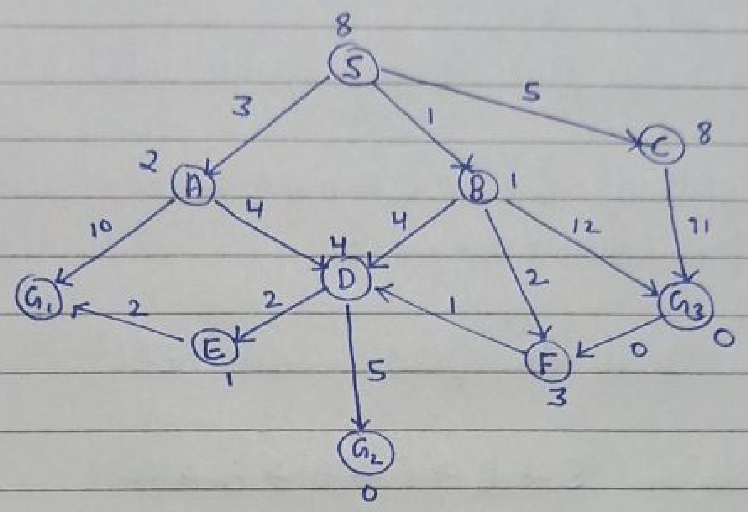
Question - 1

a) A*



Start node = S   Goal node = G_1

We are starting from S. S have three Children A, B and C

now we Calculate f(A), f(B) and f(C)

f(A) = 2 + 3 = 5

f(B) = 1 + 1 = 2   here f(B) is min. so we select node B

f(C) = 8 + 5 = 13   ∴ now path is S → B and path Cost = 1

Now we ~~can~~ reach B. we Can reach D, F, G_3 from B

f(D) = 4 + 4 = 8

f(F) = 3 + 2 = 5   here f(F) is minimum so we select

f(G_3) = 0 + 12 = 12   node F

∴ now path is S → B → F Path Cost = 1 + 2 = 3

Now we reach F. We Can reach D from F

f(D) = 4 + 1 = 5   here we have only one Value so this is minimum. We select D

∴ now path is S → B → F → D Path Cost = 3 + 1 = 4

Now we reach D. We Can reach E from D

f(E) = 1 + 2 = 3   Only one Value so we select E

∴ now path is S → B → F → D → E   path Cost = 4 + 2 = 6

Now we reach E. We can reach G.

$f(G) = 0 + 2 = 2$.

so min f is f(G). thats why we select G

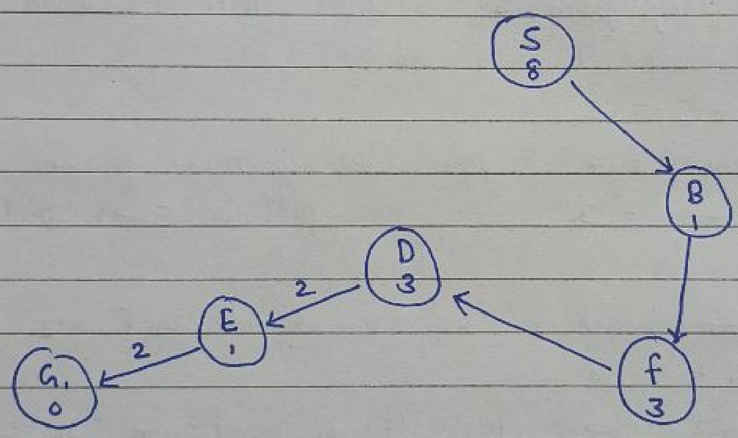
∴ now path is $S \to B \to f \to D \to E \to G$

and path cost = 6 + 2 = 8

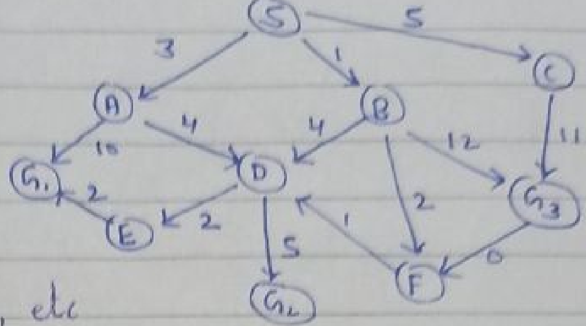
Now we reach the final (Goal node)

∴ path is $S \to B \to f \to D \to E \to G$

path cost = 8



b) Uniform Cost Search



We will be using
Priority Queue
We will be using S_0, A_{11} etc
where A is node and 11 is path cost

Priority Queue ← $\{S_0\}$ here S_0 is min. so we remove and add all the Children of S

$\{A_3, B_1, C_5\} \Rightarrow \{B_1, A_3, C_5\}$

Now we will remove B and add all the children of B

$\{D_5, F_3, (G_3)_{13}, A_3, C_5\} \Rightarrow \{A_3, F_3, C_5, D_5, (G_3)_{13}\}$

here A and F have same path cost so we will take out alphabetically. now remove A from Priority Queue. and add all the children of A

$\{(G_1)_{13}, D_7, F_3, C_5, D_5, (G_3)_{13}\} \Rightarrow \{F_3, C_5, D_5, D_7, (G_1)_{13}, (G_3)_{13}\}$

Now we pull out F_3 from Priority Queue. and add all the children of F in Priority Queue.

$\{D_4, C_5, D_5, D_7, (G_1)_{13}, (G_3)_{13}\}$

Now pull out D_4 from Priority Queue and add all the children of D in Priority Queue.

$\{E_6, C_5, D_5, D_7, (G_1)_{13}, (G_3)_{13}\} \Rightarrow \{C_5, D_5, E_6, D_7, (G_1)_{13}, (G_3)_{13}\}$

*

Now pull out C_5 from Priority Queue and add all the children of C in Priority Queue.

$\{(G_3)_{16}, D_5, E_6, D_7, (G_1)_{13}, (G_3)_{13}\} \Rightarrow \{D_5, E_6, D_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

Now pull out D_5 from Priority Queue and add all the children of D in Priority Queue.

Swap it

$\{E_7, E_6, D_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\} \Rightarrow \{E_6, E_7, D_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

Now pull out E_6 from Priority Queue and all the children of E in Priority Queue.

$\{(G_1)_8, D_7, E_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\} \Rightarrow \{D_7, E_7, (G_1)_8, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

Now pull out D_7 and add all the children of D

$\{E_9, (G_2)_{12}, E_7, (G_1)_8, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

$\Downarrow$

$\{E_7, (G_1)_8, E_9, (G_2)_{12}, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

Now pull out E_7 and add all the children of E in PQ.

$\{(G_1)_9, (G_1)_8, E_9, (G_2)_{12}, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

$\Downarrow$

$\{(G_1)_8,$ [illegible], $E_9, (G_1)_9, (G_1)_{12}, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

Now pull out G_1 which is the Goal state

$\therefore$ Path Cost = 8

and Path found = $S \to B \to F \to D \to E \to G_1$

$\{S_0\}$

$\downarrow$

$\{B_1, A_3, C_5\}$

$\downarrow$

$\{A_3, F_3, C_5, D_5, (G_3)_{13}\}$

$\downarrow$

$\{F_3, C_5, D_5, D_7, (G_1)_{13}, (G_3)_{13}\}$

$\downarrow$

$\{D_4, C_5, D_5, D_7, (G_1)_{13}, (G_3)_{13}\}$

$\downarrow$

$\{C_5, D_5, E_6, D_7, (G_1)_{13}, (G_3)_{13}\}$

$\downarrow$

$\{D_5, E_6, D_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

$\downarrow$

$\{E_6, D_7, E_7, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

$\downarrow$

$\{D_7, E_7, (G_1)_8, (G_1)_{13}\ (G_3)_{13}, (G_3)_{16}\}$

$\downarrow$

$\{E_7, (G_1)_8, E_9, (G_2)_{12}, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

$\downarrow$

$\{(G_1)_8, E_9, (G_1)_9, (G_1)_{12}, (G_1)_{13}, (G_3)_{13}, (G_3)_{16}\}$

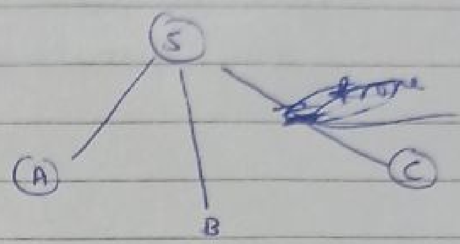
c) Iterating deepening A^*

RS = min. prone value = 8 for S

$f(C) = 13 >$ min. prone value

$f(A) = \cancel{8}\ 5$

$f(B) = 2$



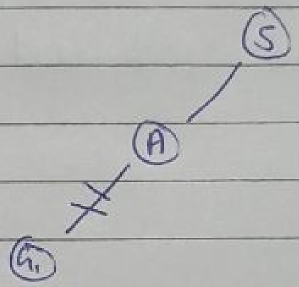
We have to Expand node in from left to right

$f(A) < 8$

$5 < 8$

$f(G_1) = 13 > 8$

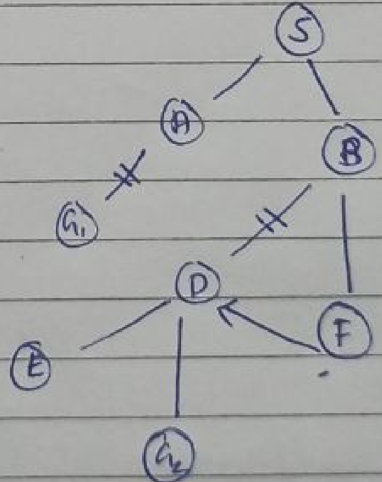
Path = $[S, A, G_1]$



Prone this branch

Now

$f(B) = 2$



$f(D) = 8$

∴ prone the branch

$f(F) = 5 < 8$

∴ expand

$f(D) = 5 < 8$

∴ expand

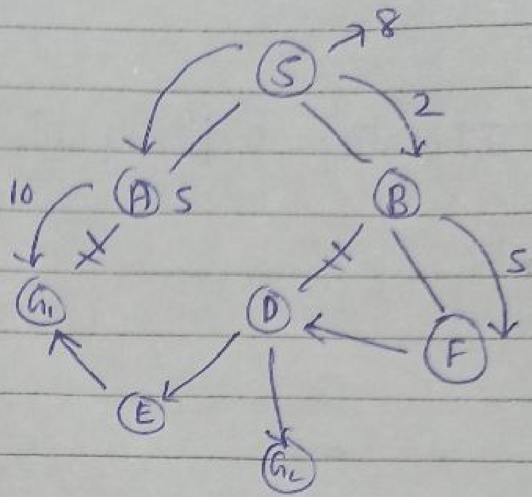
$f(E) = 3$ $\quad f(G_2) = 5$ $\quad$ E is left most

∴ expand E

$f(G_1) = 2$

$\therefore$ expand G_1

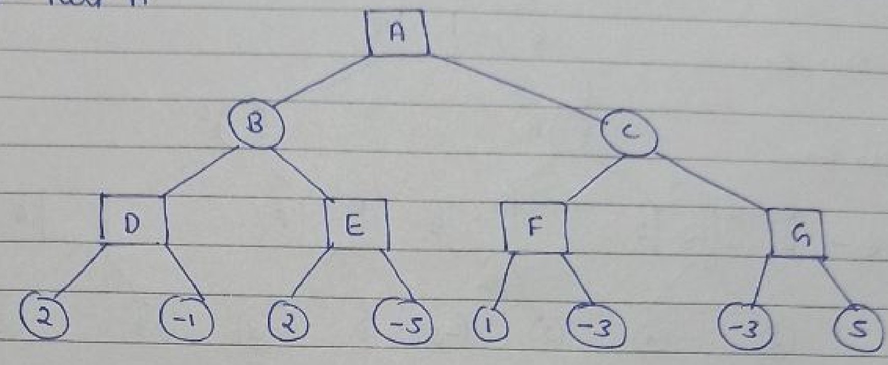
here we reach the
final Goal



$\therefore$ path is $S \rightarrow B \rightarrow F \rightarrow D \rightarrow E \rightarrow G_1$

and path cost = 8

Q.2 a) Part A



□ → max ○ → min

~~for D $D = \min(2,-1) = -1$~~

~~for E $E = \min(2,-5) = -5$~~

~~for F $F = \min(1,-3) = -3$~~

~~for G $G = \min(-3,5) = -3$~~

for D $D = \max(2,-1) = 2$

for E $E = \max(2,-5) = 2$

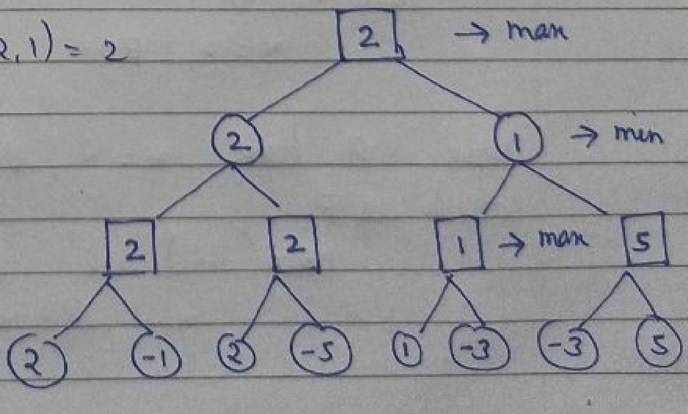
for F $F = \max(1,-3) = 1$

for G $G = \max(-3,5) = 5$

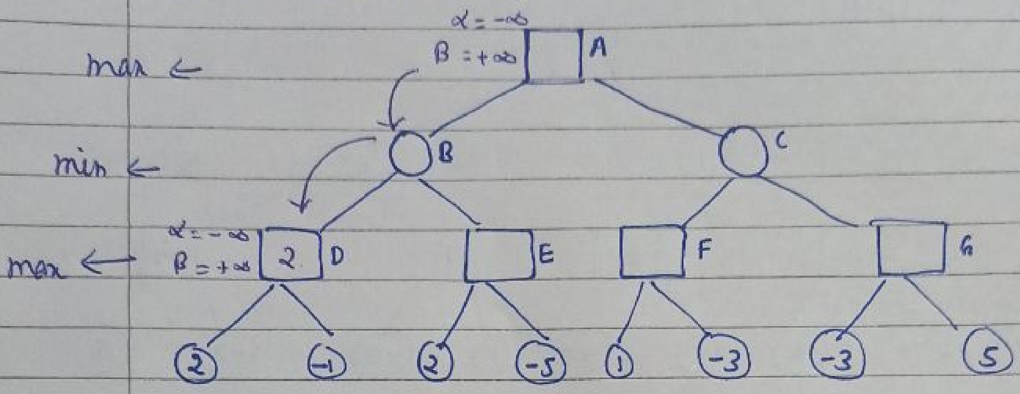
for B $B = \min(2,2) = 2$

for C $C = \min(1,5) = 1$

for A $A = \max(2,1) = 2$



- Alpha - beta pruning

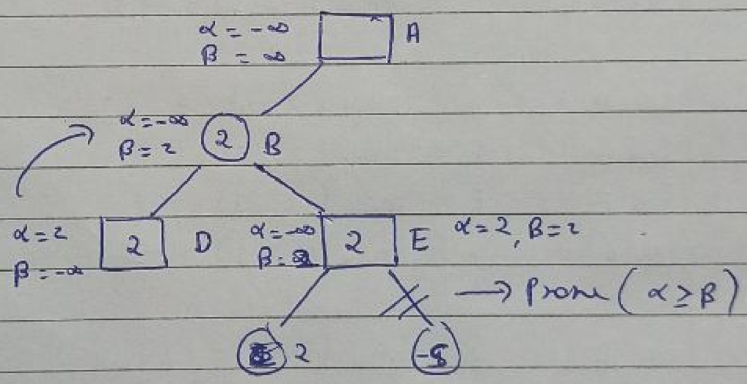


at D $\max(2,-1) = 2$

at D α will change $\alpha = 2$, $\beta = -\infty$

Now for B $\min(2, \infty) = 2$

$\therefore$ ~~$\alpha = 2$, β~~ $\alpha = -\infty$, $\beta = 2$



for E

$\max(2, -\infty) = 2$

$\therefore \alpha = 2, \beta = 2$

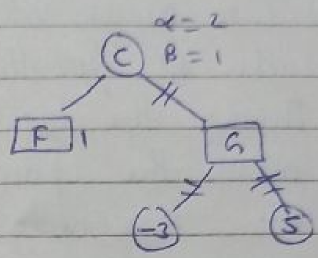
Now A have $\alpha = 2$, $\beta = \infty$ Now go left most node which is F

$\max(1,2) = 2$ no change

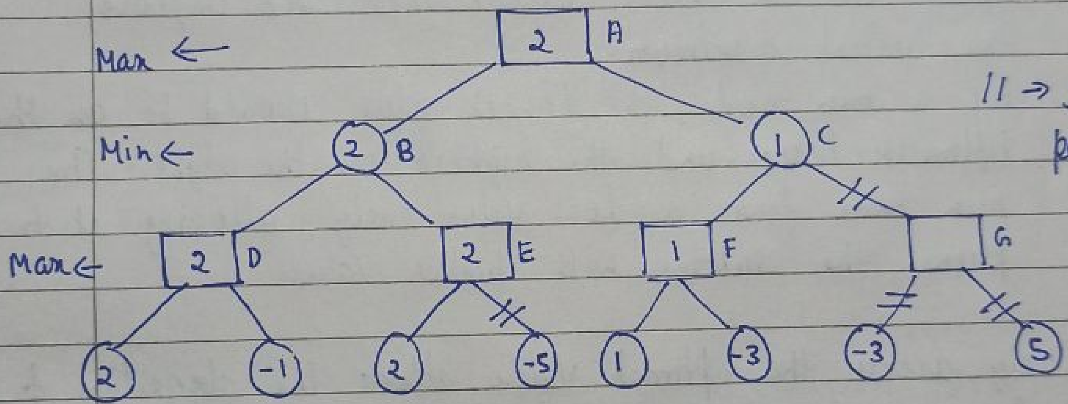
$\therefore \alpha = 2$, $\beta = \infty$

Now return to C having $\alpha = 2$, $\beta = 1$

Since $\alpha \geq \beta$ prune children of edge b/w C and G and all consecutive node



Final tree is



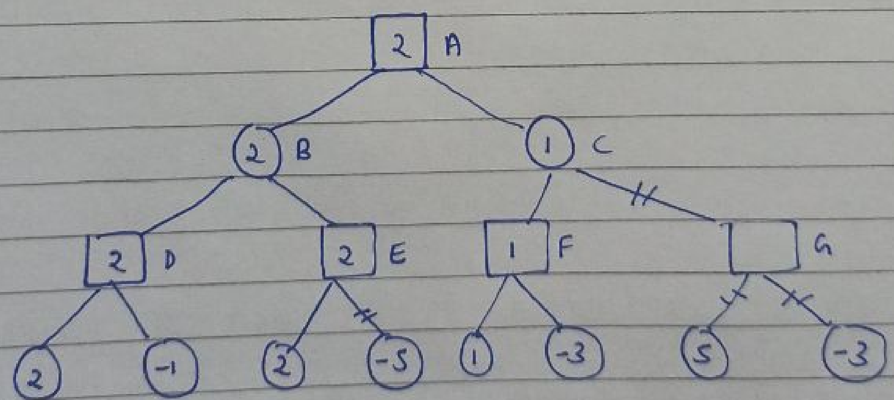
// ⇒ represent prune branch

b) Part B

* Best Case:
Best Case occur when the algorithm can prune as many branches as possible early in search.
  - Arrange the tree so that the first child nodes searched contain the optimal values.
  - For a max node, we want the highest value on the leftmost leaves. This way the max nodes find a high value early and when it moves to evaluate subsequent nodes, it can prune subtree that can't improve the current maximum.

  For a min node, the lowest value should be on the leftmost leaves and the highest on the right. The min node finds a low value early, allowing it to prune the subtree with higher values

Early access the optimal value allow the algorithm to perform maximum pruning reducing the no. of nodes it has to evaluate. This reduces the no. of comparisons, making the search more efficient.



* Worst Case

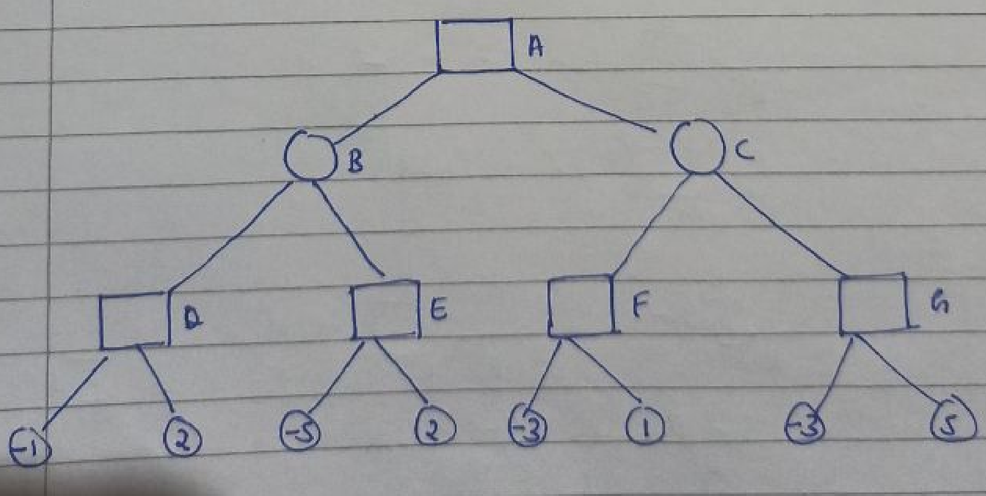
The worst case occur when alpha-beta pruning is unable to prune any branches or does so minimally.

Minimize Pruning:

1) Arrange the leaves in such a way that the max nodes have the lowest values on the leftmost leaves and the highest value on right. This forces the max node to evaluate all its children before it find a high value, preventing it from pruning any branches.

2) Similarly arrange the leaves so that the min nodes have the highest values on the left and the lowest values on the right. This forces the min node to evaluate all its children before finding a low value, again preventing early pruning.

In this arrangement, the algorithm cannot prune any subtree early since it always finds non-optimal value first. This forces it to explore more branches.



c) Part c

In the best case, alpha-beta pruning results in a complexity of $O(b^{d/2})$. This is because

In each step, half of the branches are pruned. therefore instead of evaluating b^d nodes only $b^{d/2}$ nodes need to be evaluated.

$\therefore$ Complexity is $O(b^{d/2})$ for best case

$O(b^d)$ for worst case

QUESTION 3

b)

For test case 1

Iterative Deepening Search Path: [1, 7, 6, 2]

Bidirectional Search Path: [1, 7, 6, 2]

For test case 2

Iterative Deepening Search Path: [5, 97, 98, 12]

Bidirectional Search Path: [5, 97, 98, 12]

For test case 3

Iterative Deepening Search Path: None

Bidirectional Search Path: None

For test case 4

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]

It give same path for all four test cases . Iterative Deepening Search (IDS) and Bidirectional Breadth-First Search (BFS) can potentially produce different paths between a pair of nodes in a given graph **G** for the following reasons:

- **IDS** performs a depth-first search up to a certain depth limit, iteratively increasing the depth until the goal is found. IDS may explore nodes in a depth-first order, so the exact path found will depend on the depth limit and the structure of the graph.
- **Bidirectional BFS** simultaneously searches from both the start and goal nodes, meeting in the middle. This strategy is generally faster because it explores fewer nodes, and it explores nodes level by level in a breadth-first manner.
- **IDS** may find a path by going deep along one branch before exploring others, potentially finding a suboptimal path (in terms of the number of nodes explored) earlier, especially in non-uniform graphs.
- **Bidirectional BFS** tends to find the shortest path (in terms of the number of edges) between two nodes, as it explores all possible paths of increasing length from both the start and the goal.
- **IDS** is not guaranteed to find the shortest path on its first successful iteration, as it focuses on depth-first exploration.
- **Bidirectional BFS** guarantees finding the shortest path (in terms of edge count) in an unweighted graph, as it explores nodes layer by layer.

The paths discovered by **IDS** and **Bidirectional BFS** will **not always be identical**. While they might produce the same path in certain cases (such as when there is only one unique path or the shortest path is found early in both searches), their exploration methods differ significantly, leading to potentially different paths for most graphs.

c)

test case 1

Finding path between 1 and 2:

IDS Path: [1, 7, 6, 2], Time: 0.0226s, Memory: 3.78KB

BDS Path: [1, 7, 6, 2], Time: 0.0081s, Memory: 3.27KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(1, 2, 0.022558212280273438, 3.7802734375)

Bidirectional BFS Results:

(1, 2, 0.008051156997680664, 3.2685546875)

Test case 2

Finding path between 5 and 12:

IDS Path: [5, 97, 98, 12], Time: 0.0223s, Memory: 3.23KB

BDS Path: [5, 97, 98, 12], Time: 0.0174s, Memory: 3.23KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(5, 12, 0.022267818450927734, 3.234375)

Bidirectional BFS Results:

(5, 12, 0.017354726791381836, 3.234375)

Test case 3

Finding path between 12 and 49:

No path found from 12 to 49 (Different components)

IDS Path: None, Time: 0.0116s, Memory: 10.73KB

No path found from 12 to 49 (Different components)

BDS Path: None, Time: 0.0159s, Memory: 10.73KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(12, 49, 0.01157379150390625, 10.734375)

Bidirectional BFS Results:

(12, 49, 0.015914201736450195, 10.734375)

Test case 4

Finding path between 4 and 12:

IDS Path: [4, 6, 2, 9, 8, 5, 97, 98, 12], Time: 0.0407s, Memory: 3.23KB

BDS Path: [4, 6, 2, 9, 8, 5, 97, 98, 12], Time: 0.0199s, Memory: 4.65KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(4, 12, 0.040726423263549805, 3.234375)

Bidirectional BFS Results:

(4, 12, 0.0198519229888916, 4.6484375)

```python
import time
import tracemalloc
import numpy as np


def compare_algorithms(adj_matrix, start_node, goal_node):
    ids_time_memory = []
    bds_time_memory = []

    print(f"Finding path between {start_node} and {goal_node}:")

    # Measure time and memory for IDS
    tracemalloc.start()
    start_time = time.time()
    ids_path = get_ids_path(adj_matrix, start_node, goal_node)
    ids_exec_time = time.time() - start_time
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    ids_memory_usage = peak / 1024  # Convert to KB

    ids_time_memory.append((start_node, goal_node, ids_exec_time, ids_memory_usage))
    print(f"IDS Path: {ids_path}, Time: {ids_exec_time:.4f}s, Memory: {ids_memory_usage:.2f}KB")

    # Measure time and memory for Bidirectional BFS
    tracemalloc.start()
    start_time = time.time()
    bds_path = get_bidirectional_search_path(adj_matrix, start_node, goal_node)
    bds_exec_time = time.time() - start_time
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    bds_memory_usage = peak / 1024  # Convert to KB

    bds_time_memory.append((start_node, goal_node, bds_exec_time, bds_memory_usage))
    print(f"BDS Path: {bds_path}, Time: {bds_exec_time:.4f}s, Memory: {bds_memory_usage:.2f}KB\n")

    return ids_time_memory, bds_time_memory



# Taking user input for start and end node
try:
    start_node = int(input("Enter the start node: "))
    goal_node = int(input("Enter the end node: "))

    if start_node == goal_node:
        print("Start node and goal node are the same. No path needed.")
    elif 0 <= start_node < len(adj_matrix) and 0 <= goal_node < len(adj_matrix):
        ids_time_memory, bds_time_memory = compare_algorithms(adj_matrix, start_node, goal_node)
    else:
        print("Invalid node numbers. Please enter values within the graph size.")
except ValueError:
    print("Invalid input. Please enter integer values for start and end nodes.")

# Summarize results
def summarize_results(ids_time_memory, bds_time_memory):
    print("\nSummary (start_node, goal_node, time (s), memory (KB))")
    print("IDS Results:")
    for result in ids_time_memory:
        print(f"{result}")

    print("\nBidirectional BFS Results:")
    for result in bds_time_memory:
        print(f"{result}")

# Only summarize if valid results were obtained
if 'ids_time_memory' in locals() and 'bds_time_memory' in locals():
    summarize_results(ids_time_memory, bds_time_memory)
```

e) b)   test case 1

A* Path: [1, 27, 9, 2]

Bidirectional Heuristic Search Path: [1, 27, 9, 2]

test case 2

A* Path: [5, 97, 28, 10, 12]

Bidirectional Heuristic Search Path: [5, 97, 28, 10, 12]

test case 3

A* Path: None

Bidirectional Heuristic Search Path: None

test case 4

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]

It gives different path for all four test cases.   It can be different path for both algorithms having same test cases . its because in bidirectional we use a* from both direction that's why we may have different node compare to simple a*.

e) c)

test case 1:

Finding path between 1 and 2:

IDS Path: [1, 7, 6, 2], Time: 0.0053s, Memory: 9732.96KB

BDS Path: [1, 7, 6, 2], Time: 0.0037s, Memory: 3.27KB

A* Path: [1, 27, 9, 2], Time: 0.0108s, Memory: 16.98KB

Bidirectional A* Path: [1, 27, 9, 2], Time: 0.0176s, Memory: 29.45KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(1, 2, 0.0052716732025146484, 9732.96484375)

BDS Results:

(1, 2, 0.003726959228515625, 3.2685546875)

A* Results:

(1, 2, 0.010778427124023438, 16.9765625)

Bidirectional A* Results:

(1, 2, 0.01763153076171875, 29.4453125)

 # A* and **Bidirectional A*** were slower but more memory efficient than IDS

Test case 2:

Finding path between 5 and 12:

IDS Path: [5, 97, 98, 12], Time: 0.0202s, Memory: 3.56KB

BDS Path: [5, 97, 98, 12], Time: 0.0274s, Memory: 3.26KB

A* Path: [5, 97, 28, 10, 12], Time: 0.0208s, Memory: 14.82KB

Bidirectional A* Path: [5, 97, 28, 10, 12], Time: 0.0545s, Memory: 29.72KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(5, 12, 0.020209550857543945, 3.5634765625)

BDS Results:

(5, 12, 0.027410030364990234, 3.2578125)

A* Results:

(5, 12, 0.0207827091217041, 14.8203125)

Bidirectional A* Results:

(5, 12, 0.054450035095214844, 29.71875)

A* was relatively fast and used moderate memory.

Bidirectional A* was the slowest and used the most memory.

Test case 3:

Finding path between 12 and 49:

No path found from 12 to 49 (Different components)

IDS Path: None, Time: 0.0342s, Memory: 11.15KB

No path found from 12 to 49 (Different components)

BDS Path: None, Time: 0.0222s, Memory: 10.73KB

No path found from 12 to 49 (Different components)

A* Path: None, Time: 0.0297s, Memory: 11.00KB

No path found from 12 to 49 (Different components)

Bidirectional A* Path: None, Time: 0.0105s, Memory: 11.27KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(12, 49, 0.034154653549194336, 11.154296875)

BDS Results:

(12, 49, 0.022238969802856445, 10.734375)

A* Results:

(12, 49, 0.0296933650970459, 11.0)

Bidirectional A* Results:

(12, 49, 0.01054072380065918, 11.265625)

Bidirectional A* was the fastest here

Test case 4:

Finding path between 4 and 12:

IDS Path: [4, 6, 2, 9, 8, 5, 97, 98, 12], Time: 0.0617s, Memory: 3.23KB

BDS Path: [4, 6, 2, 9, 8, 5, 97, 98, 12], Time: 0.0127s, Memory: 4.76KB

A* Path: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12], Time: 0.0275s, Memory: 15.42KB

Bidirectional A* Path: [4, 6, 27, 9, 8, 5, 97, 28, 10, 12], Time: 0.0601s, Memory: 29.42KB

Summary (start_node, goal_node, time (s), memory (KB))

IDS Results:

(4, 12, 0.06170058250427246, 3.234375)

BDS Results:

(4, 12, 0.012667179107666016, 4.7578125)

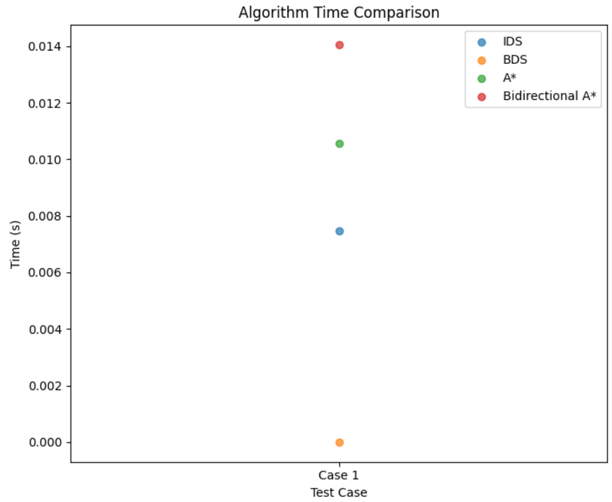
A* Results:

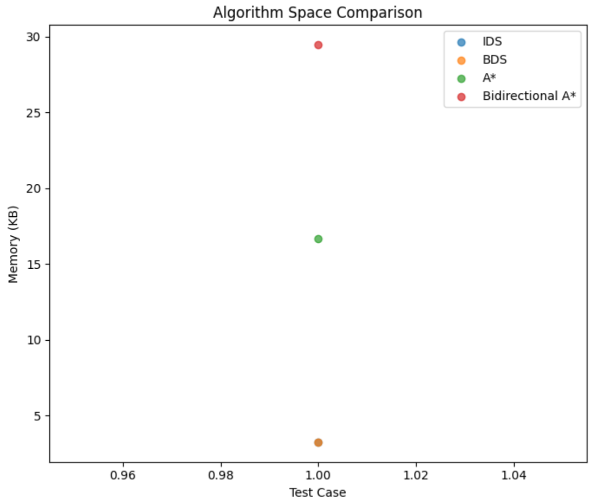
(4, 12, 0.027537107467651367, 15.421875)

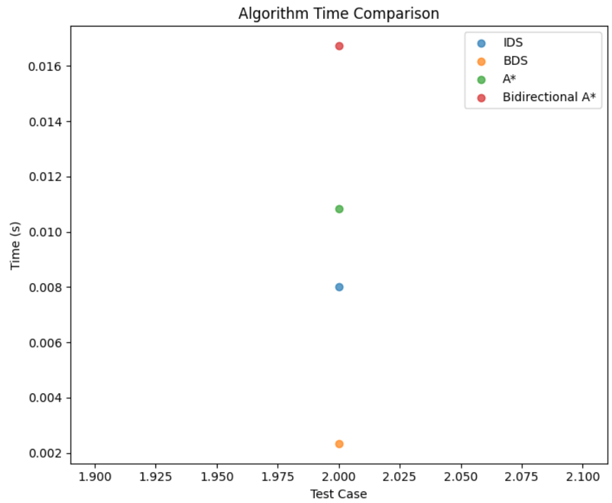
Bidirectional A* Results:

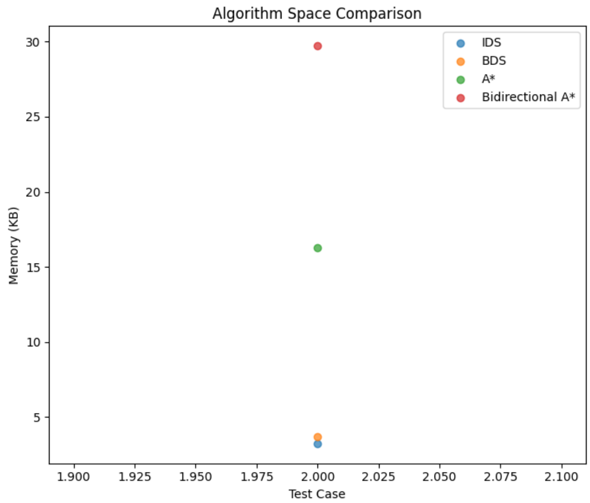
(4, 12, 0.06013774871826172, 29.421875)

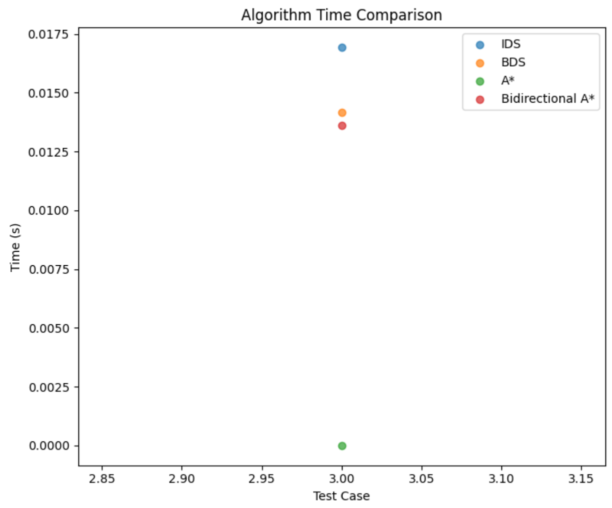
f)

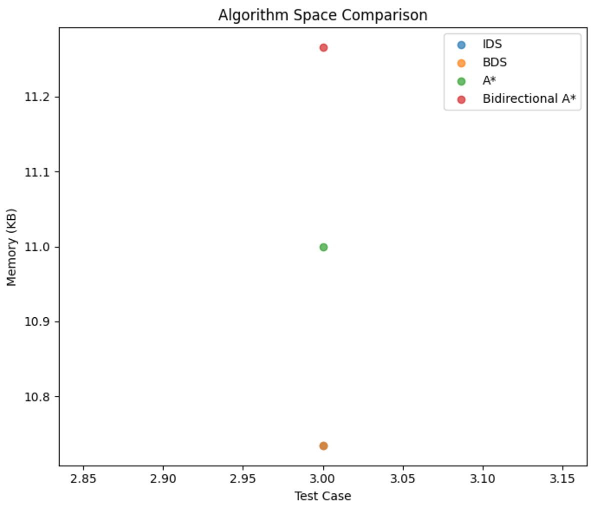

Algorithm Time Comparison
IDS
BDS
A*
Bidirectional A*
Time (s)
Case 1
Test Case

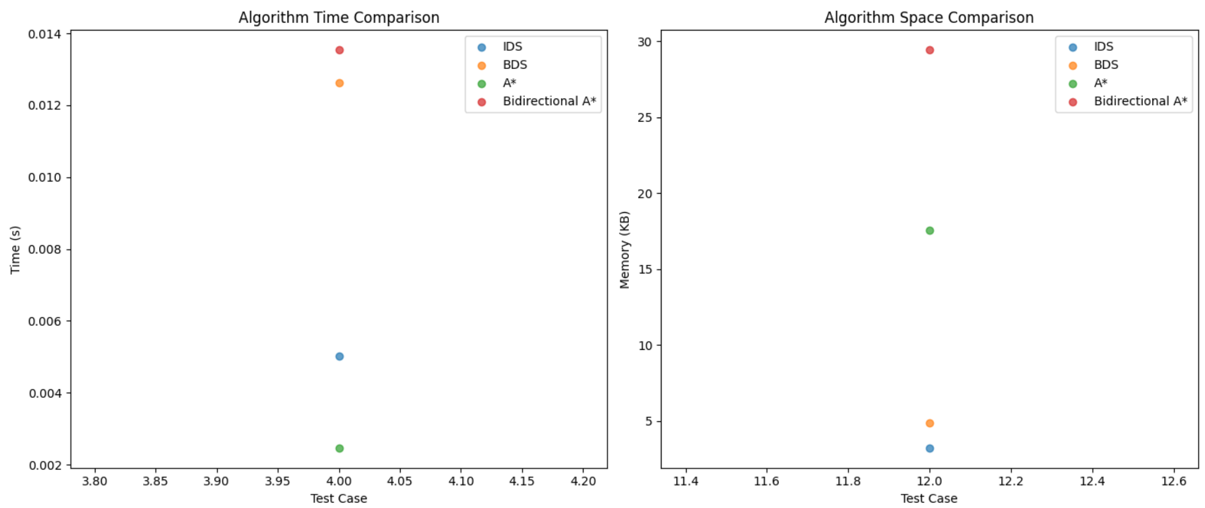

Algorithm Space Comparison
IDS
BDS
A*
Bidirectional A*
Memory (KB)
Test Case


Algorithm Time Comparison
IDS
BDS
A*
Bidirectional A*
Time (s)
Test Case


Algorithm Space Comparison
IDS
BDS
A*
Bidirectional A*
Memory (KB)
Test Case


Algorithm Time Comparison
IDS
BDS
A*
Bidirectional A*
Time (s)
Test Case


Algorithm Space Comparison
IDS
BDS
A*
Bidirectional A*
Memory (KB)
Test Case



Benefits and Drawbacks of Informed vs. Uninformed Algorithms

Informed Algorithms:

- **Benefits**:
  - **Faster Execution**: Heuristic functions guide the search, leading to faster solutions in terms of time.
  - **Path Quality**: Often find more optimal paths because they focus the search on promising directions.
- **Drawbacks**:
  - **Higher Space Usage**: May use more memory to store additional data structures.
  - **Heuristic Dependency**: Performance is dependent on the quality of the heuristic. Poor heuristics can degrade performance.

Uninformed Algorithms:

- **Benefits**:
  - **Simplicity**: Easier to implement and understand. No need for heuristic functions.
  - **Predictable Memory Usage**: Often have more predictable and potentially lower memory usage.
- **Drawbacks**:
  - **Slower Execution**: May explore many unnecessary paths, leading to longer execution times.
  - **Less Optimal Paths**: Might not always find the most optimal path, as they do not utilize domain knowledge.

Comparison

Efficiency:

Time:

Bidirectional A*and **Bidirectional BFS** are usually more time-efficient due to their dual-directional approach, reducing the number of nodes expanded.

A* is efficient with a good heuristic but can still have high time complexity.

IDS can be less efficient due to repeated depth-limited searches.

Space:

Bidirectional A* and **Bidirectional BFS** generally use less space compared to **A*** and **IDS** because they search from both ends.

A* has high space complexity due to storing all nodes in open and closed lists.

IDS uses space proportional to the depth and current search level, making it more space-efficient than A* but potentially less efficient than bidirectional methods.

Optimality:

Bidirectional A* and **A*** are optimal if the heuristic used is admissible and consistent.

Bidirectional BFS and **IDS** are optimal for uniform cost scenarios.

Benefits and Drawbacks:

Informed Searches (A*, Bidirectional A*):

Benefits: Generally faster with good heuristics, leading to optimal paths.

Drawbacks: Can be memory-intensive, especially A*.

Uninformed Searches (IDS, BDS):

Benefits: Simpler to implement and understand, less dependent on heuristics.

Drawbacks: Can be slower and use more space compared to informed searches.

```python
import matplotlib.pyplot as plt

def plot_results(results):
    algorithms = ["IDS", "BDS", "A*", "Bidirectional A*"]

    # Set up the figure and axes
    plt.figure(figsize=(14, 6))

    # Plot Time
    plt.subplot(1, 2, 1)
    for algo in algorithms:
        times = [result[2] for result in results[algo]]
        # test_cases = [f"Case {i+1}" for i in range(len(times))]
        test_cases=4
        plt.scatter(test_cases, times, label=algo, alpha=0.7)
    plt.xlabel('Test Case')
    plt.ylabel('Time (s)')
    plt.title('Algorithm Time Comparison')
    plt.legend()

    # Plot Space
    plt.subplot(1, 2, 2)
    for algo in algorithms:
        spaces = [result[3] for result in results[algo]]
        # test_cases = [f"Case {i+1}" for i in range(len(spaces))]
        test_cases=12
        plt.scatter(test_cases, spaces, label=algo, alpha=0.7)
    plt.xlabel('Test Case')
    plt.ylabel('Memory (KB)')
    plt.title('Algorithm Space Comparison')
    plt.legend()

    plt.tight_layout()
    plt.show()

# Call the function with results
plot_results(results)
```