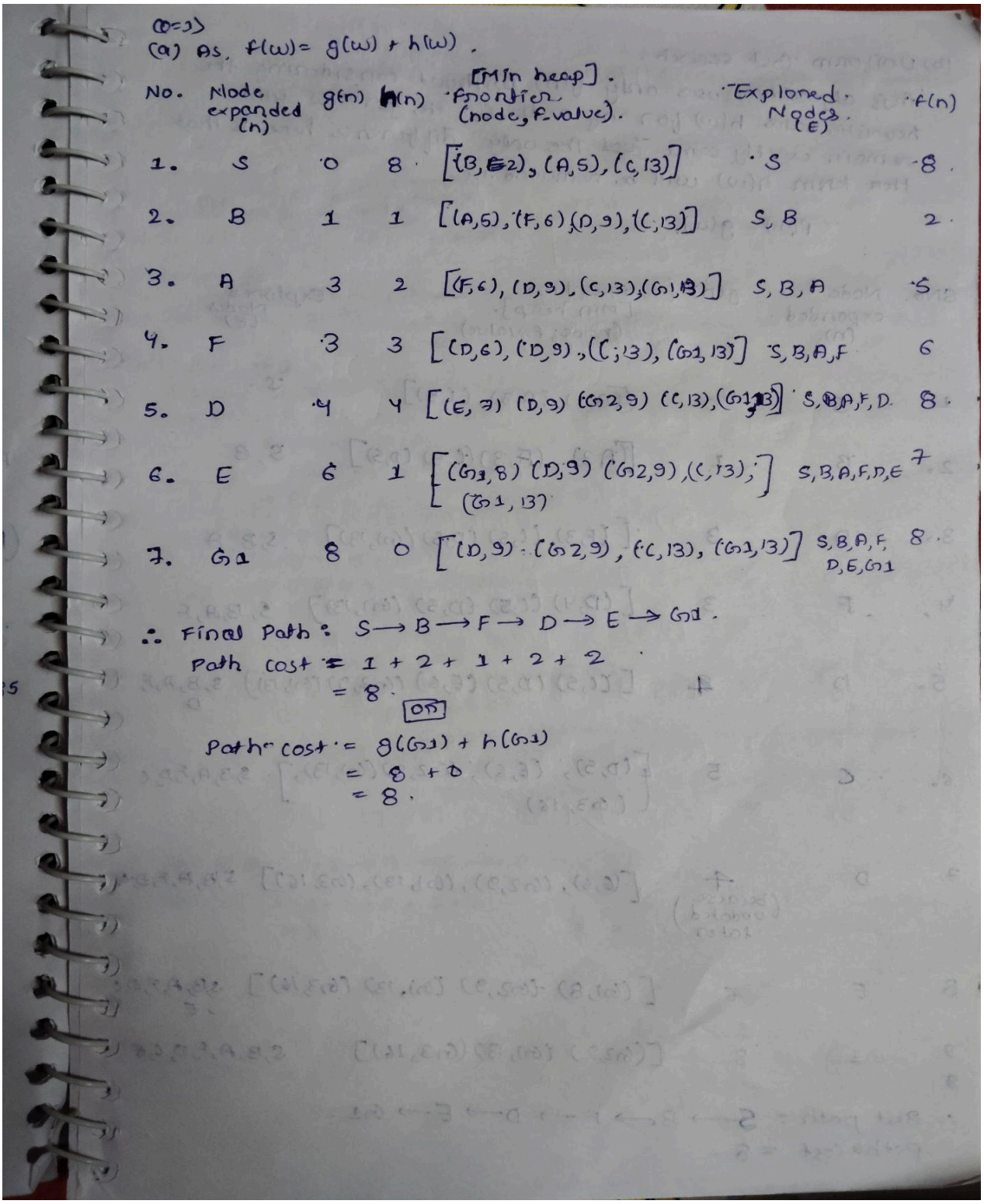
CSE643 Artificial Intelligence ASSIGNMENT-1
Search Algorithms

Subham Maurya, 2022510

Q1.



(Q=1)

(a) As, $f(w) = g(w) + h(w)$.

[Min heap].

No.	Node expanded (n)	g(n)	h(n)	Frontier (node, f-value)	Explored Nodes (E)	f(n)
1.	S	0	8	[(B,12), (A,5), (C,13)]	S	8
2.	B	1	1	[(A,5), (F,6), (D,9), (C,13)]	S, B	2
3.	A	3	2	[(F,6), (D,9), (C,13), (G1,13)]	S, B, A	5
4.	F	3	3	[(D,6), (D,9), (C,13), (G1,13)]	S, B, A, F	6
5.	D	4	4	[(E,7) (D,9) (G2,9) (C,13), (G1,13)]	S, B, A, F, D	8
6.	E	6	1	[(G1,8) (D,9) (G2,9), (C,13), (G1,13)]	S, B, A, F, D, E	7
7.	G1	8	0	[(D,9) (G2,9), (C,13), (G1,13)]	S, B, A, F, D, E, G1	8

∴ Final Path: $S \rightarrow B \rightarrow F \rightarrow D \rightarrow E \rightarrow G1$.

Path cost = 1 + 2 + 1 + 2 + 2

= 8.

OR

Path cost = g(G1) + h(G1)

= 8 + 0

= 8.

(b) Uniform Cost search.

This algorithm uses only g(n) without considering the heuristic value h(w) for any node. So, the process will remain exactly same ~~just~~. The only difference here is that the term h(w) will be removed. i.e.

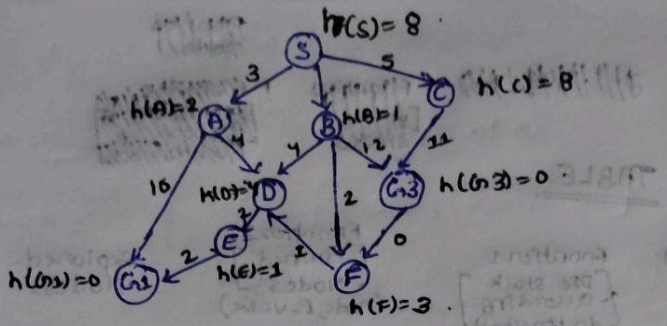
$$f(w) = g(w).$$

SNo.	Node expanded (n)	g(n)	Frontier [Min heap]. (node, f-value)	Explored Nodes (E)
1.	S	0	[(B,1), (A,3), (C,5)]	S.
2.	B	1	[(A,3), (F,3) (C,5) (D,5)]	S, B
3.	A	3	[(F,3) (C,5) (D,5) (G1,13)]	S, B, A
4.	F	3	[(D,4) (C,5) (D,5) (G1,13)]	S, B, A, F
5.	D	4	[(C,5) (D,5) (E,6) (G2,9) (G1,13)]	S, B, A, F, D
6.	C	5	[(D,5), (E,6), (G2,9) (G1,13), (G3,16)]	S, B, A, F, D, C
7.	D	4 (because updated later)	[(E,6), (G2,9), (G1,13), (G3,16)]	S, B, A, F, D, C
8	E	6	[(G1,8) (G2,9) (G1,13) (G3,16)]	S, B, A, F, D, C, E
9	G1	8	[(G2,9) (G1,13) (G3,16)]	S, B, A, F, D, C, E

∴ Best path : S → B → F → D → E → G1.

path cost = 8.

(c)



Iteration: 1

- Current Node : S
- Threshold : 8 [Because min. 8 threshold is required to move from S]

→ F(S) = 8 ≤ 8

Explore children A, B, C.

→ F(A) = 5 ≤ 8

Explore G1, D

→ F(G1) = 13 > 8 [Pruned G1]

→ F(D) = 11 > 8 [Pruned D].

→ F(B) = 2 ≤ 8

Explore D, F, G3.

→ F(D) = 9 > 8 [Pruned D].

→ F(F) = 6 ≤ 8

Explore D.

→ F(D) = 8 ≤ 8

Explore children E, G2.

→ F(E) = 7 ≤ 8

Explore children G1.

→ F(G1) = 8 ≤ 8.

Therefore, reached the goal Node.

∴ path ■ S → A → B → F → D → E → G1.

Path cost = 1 + 2 + 1 + 2 + 2

= 8.

TABLE

Iteration No.	SNo	Node expanded (n)	f(n)	Frontier 1 [DFS stack according to threshold]	Frontier 2 [Pruned Nodes (node, F-value)]	Explored Nodes
1. Threshold = (8)	1.	S	8	[ ]	—	S
	2.	A	5	[S]	—	S, A.
	3.	G1	13	[S, A]	[(G1, 13)]	S, A
	4.	D	9	[S, A]	[(D, 9), (G1, 13)]	S, A.
	5	B	2	[S]	[(D, 9), (G1, 13)]	S, A, B.
	6	D	9	[S, B]	[(D, 9), (D, 9), (G1, 13)]	S, A, B
	7	F	6	[S, B]	[(D, 9), (D, 9), (G1, 13)]	S, A, B, F
	8	D	8	[S, B, F]	[(D, 9), (D, 9), (G1, 13)]	S, A, B, F, D.
	9	E	7	[S, B, F, D]	[(D, 9), (D, 9), (G1, 13)]	S, A, B, F, D, E
	10	G1	8	[S, B, F, D, E]	[(D, 9); (D, 9); (G1, 13)]	S, A, B, F, D, E

∴ At Threshold = 8, we get

Path : S → B → F → D → E → G1

cost : 8.

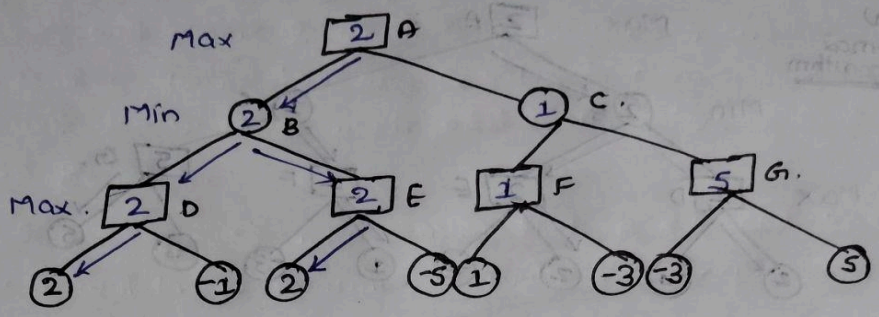
Note : In above table Frontier 2 can also be considered as a variable container which stores min threshold > current threshold . It is used as min heap just for conceptual understanding.

Q2.

(Q=2)

(a)

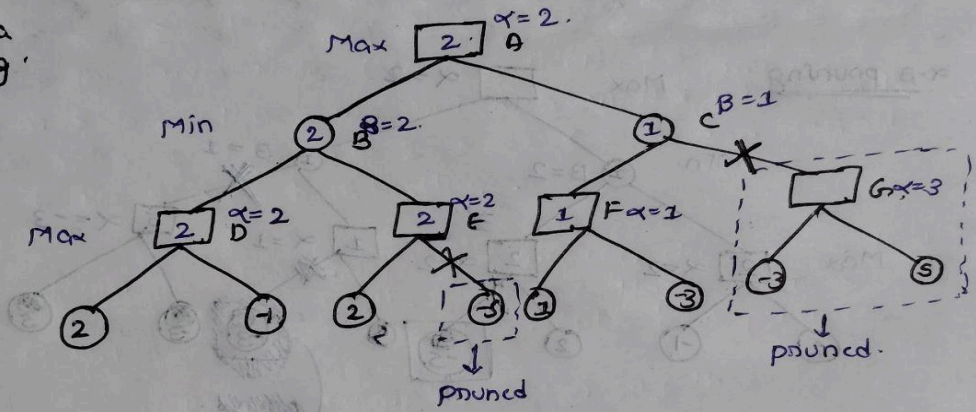
Minmax algorithm.



Best Moves (i) A → B → D → 2

(ii) A → B → E → 2.

Alpha beta pruning.



(b)

Best case: There is no need to rearrange the leaf nodes on internal nodes. because the given tree is already the best case for alpha beta pruning.

Reason: In the above tree.

Total no. of nodes = 15.

Total leaf nodes = 8.

Total non leaf nodes = 7

Depth of the tree = 3.

Now while doing alpha beta pruning i.e. assigning the best moves for maximizer and worst move for minimizer, we can see that:-

(i) At node E, the value of α (maximizer) is 2. While its parent node B (minimizer) has β = 2. which means that the value at node B must be less than or equal to 2. i.e.

value(B) ≤ 2.

Since, E is the maximizer and promises that value(E) ≥ 2 then there is no need to visit other leaf node.
Hence. -5, the children of E is pruned.

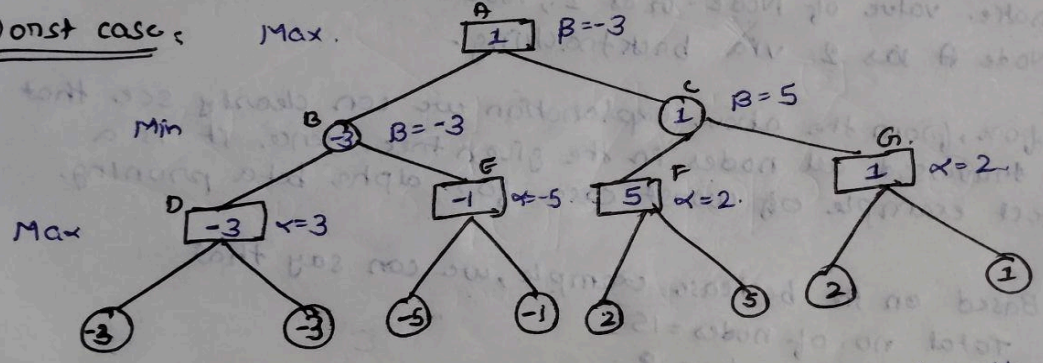
(ii) Similary. Node C is a minimizer with β = 1 while. Node A is the maximizer with α = 2 which means that

value(A) ≥ 2.
value(C) ≤ 1.

So, there is no need to expand Node C further. Hence the right child of Node C is pruned. i.e Node G, -3, -5 are pruned.

Therefore this makes that we have to expand less nodes as compare to every possible other cases which could be formed.

Worst case:



Reason: Starting with the Node D which is a maximizer. Now the value of α at Node D = 3 but we do not have any β value at Node B to compare with. Hence it will also visit -3 and the value of Node D will become -3 which is also forwarded as a value to β at Node B which is minimizer.

(iii) Now, moving to Node E where we will visit -5 and α_E becomes -5 due to which. there may be a chance that the value of Node B will become -5 or the value value of node E will become something ≥ -5. Hence we will also visit -1 which will make the value at Node E as -1 which further make value of Node B as -3 and $B_B = -3$.

(iii) Now, going to the right side of A to Node F where after visiting 2, α_F will become 2 which is a maximizer and since its parent Node C doesn't have any β_c value to compare with. Hence it will also visit 5 which will make value of Node F as 5 and β_c as 5.

(iv) As, $\beta_c = 5$ and $\gamma_A = -3$ which means that $\beta_c \leq 5$ and $\gamma_A \geq -3$. Hence Node C will try to minimize it further by exploring Node G.

(v) As, we reach Node G we will visit 2 which will make the value of $\alpha_G = 2$. As, $\beta_c = 5$ and $\gamma_G = 2$. ~~which exceed any [illegible] $\beta \leq 2$. Hence we also explore. This rejected two possibility that either β_c will become less than equal to 2 or $\beta \leq 2$ or~~ This gives Node C an opportunity to improve β from 5 to 2. Hence we will also visit 1 which will make value of Node G as 2, Node C as 2 and finally Node A as 2 via backtracking.

Therefore, from the above explanation we can clearly see that we traversed all nodes in the given tree. Hence it is a perfect example of worst case for alpha beta pruning.

(c) Based on the best case example, we can say that

Total no. of nodes = 15

Total leaf nodes = 8.

Total pruned nodes = 4.

depth of tree = 3

Without pruning, the no. of nodes to explore is $O(b^d)$ because algorithm evaluates every node but with alpa beta pruning; at each level half of the nodes are pruned. This reduces the number of nodes evaluated at each level by a factor of 2. Since in the best case we pruned {-5} and {6, -3, 5}. reducing the evaluation size at each level by approximately 2. Apart from that the branching factor remains same i.e 2.

Hence $O(b^{d/2})$ is the best case time complexity for above example

Q3.

(a)

```python
def get_ids_path(adj_matrix, start_node, goal_node):
    def dfs_limited(adj_matrix, node, goal, depth, path, visited):
        if node == goal:
            return path + [node]
        if depth == 0:
            return None
        visited.add(node)
        for neighbor, is_connected in enumerate(adj_matrix[node]):
            if is_connected and neighbor not in visited:
                result = dfs_limited(adj_matrix, neighbor, goal, depth - 1, path + [node], visited)
                if result:
                    return result
        # visited.remove(node)  #since it is a graph not a tree
        return None

    def ids(adj_matrix, start, goal, max_depth):
        for depth in range(max_depth):
            visited = set()
            path = dfs_limited(adj_matrix, start, goal, depth, [], visited)
            if path:
                return path
        return []

    max_depth = len(adj_matrix)
    return ids(adj_matrix, start_node, goal_node, max_depth)
```

```python
def get_bidirectional_search_path(adj_matrix, start_node, goal_node):
    # Edge case
    if start_node == goal_node:
        return [start_node]

    # BFS function for both start or goal traversal
    def bfs_directional(frontier, visited_from_this_side, visited_from_other_side, parents_from_this_side):
        level_size = len(frontier)
        for _ in range(level_size):
            current_node = frontier.popleft()

            for neighbor, is_connected in enumerate(adj_matrix[current_node]):
                if is_connected and neighbor not in visited_from_this_side:
                    visited_from_this_side.add(neighbor)
                    frontier.append(neighbor)
                    parents_from_this_side[neighbor] = current_node
                    if neighbor in visited_from_other_side:
                        return neighbor
        return None

    # Function to get the path using parent
    def reconstruct_path(parents_from_start, parents_from_goal, meeting_node):
        path = []
        curr = meeting_node
        while curr is not None:
            path.append(curr)
            curr = parents_from_start[curr]
        path.reverse()

        curr = parents_from_goal[meeting_node]
```

```
    curr = parents_from_goal[meeting_node]
    while curr is not None:
        path.append(curr)
        curr = parents_from_goal[curr]

    return path

# Main code begins
start_frontier = deque([start_node])
goal_frontier = deque([goal_node])

visited_from_start = {start_node}
visited_from_goal = {goal_node}

parents_from_start = {start_node: None}
parents_from_goal = {goal_node: None}

while start_frontier and goal_frontier:
    meeting_node = bfs_directional(start_frontier, visited_from_start, visited_from_goal, parents_from_start)
    if meeting_node is not None:
        return reconstruct_path(parents_from_start, parents_from_goal, meeting_node)

    meeting_node = bfs_directional(goal_frontier, visited_from_goal, visited_from_start, parents_from_goal)
    if meeting_node is not None:
        return reconstruct_path(parents_from_start, parents_from_goal, meeting_node)

return []
```

(b)

Public Test Case 1:

```
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]
```

Public Test Case 2:

```
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]
```

Public Test Case 3:

```
Enter the start node: 12
Enter the end node: 49
Iterative Deepening Search Path: []
Bidirectional Search Path: []
```

Public Test Case 4:

```
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]
```

We are getting the same answer/path for all the public test cases in both the algorithm.

No, it will not be identical for all the test cases for the given graph because of the following reasons:

1. **For Identical, the paths obtained by IDS and Bidirectional BFS can be identical in some cases. This is because both algorithms are searching for the shortest path from vertex u to vertex v, and in graphs without multiple equally short paths or cycles, the shortest path is unique.**
2. **For different paths in graphs, there can be multiple shortest paths of the same length (e.g., in a cyclic graph). Hence, IDS might explore paths in a depth-first manner, which can lead to it finding a different but equally short path compared to Bidirectional BFS. Similarly, Bidirectional BFS progresses level by level from both sides, and the first meeting point between the two searches could lead to a different shortest path than what IDS might find.**

Example of such case where both algorithm shows different is also shown below:

```
Enter the start node: 2
Enter the end node: 115
Iterative Deepening Search Path: [2, 6, 4, 34, 33, 11, 32, 31, 3, 5, 97, 98, 12, 57, 20, 26, 85, 118, 119, 115]
Bidirectional Search Path: [2, 9, 8, 5, 97, 98, 12, 57, 26, 85, 118, 119, 115]
```

(c)
The path between all the pair between nodes are listed below using both algorithm.

(1) Using IDS (Iterative Deepining Search):

```
Start Node:0 and Goal Node:0 are same, so no computation is required

Start Node:0 and Goal Node:1:
path:[]
cost:inf
Memory Used (in bytes):26352.058350997657
Execution Time (in sec):inf

Start Node:0 and Goal Node:2:
path:[]
cost:inf
Memory Used (in bytes):38475.86970379707
Execution Time (in sec):inf

Start Node:0 and Goal Node:3:
path:[]
cost:inf
Memory Used (in bytes):27317.48054631945
Execution Time (in sec):inf

Start Node:0 and Goal Node:4:
path:[]
cost:inf
Memory Used (in bytes):39552.27156134482
Execution Time (in sec):inf

Start Node:0 and Goal Node:5:
path:[]
cost:inf
Memory Used (in bytes):39287.76094825134
Execution Time (in sec):inf
```

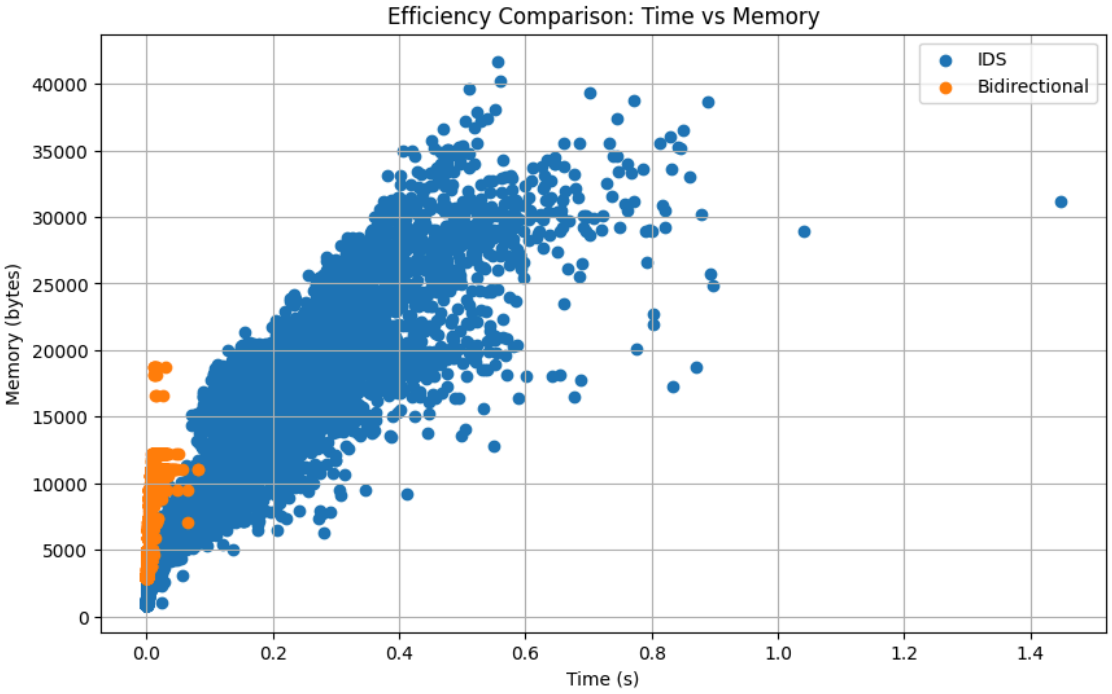
Use this link to see the full .txt file.

(2) Using Bidirectional BFS:

```
Start Node:0 and Goal Node:0 are same, so no computation is required

Start Node:0 and Goal Node:1:
path:[]
cost:inf
Memory Used (in bytes):3019
Execution Time (in sec):0.0003589999978430569

Start Node:0 and Goal Node:2:
path:[]
cost:inf
Memory Used (in bytes):3091
Execution Time (in sec):0.0005467000300996006

Start Node:0 and Goal Node:3:
path:[]
cost:inf
Memory Used (in bytes):3091
Execution Time (in sec):0.00035940000088885427

Start Node:0 and Goal Node:4:
path:[]
cost:inf
Memory Used (in bytes):3091
Execution Time (in sec):0.0002948000328615308

Start Node:0 and Goal Node:5:
path:[]
cost:inf
Memory Used (in bytes):3091
Execution Time (in sec):0.0003233999595977366
```

Use this link to view the full .txt file.

Now, we can see the comparison between these algorithm using the above result to plot a scatterplot of memory vs time which is given below:



(d)

```python
def get_astar_search_path(adj_matrix, node_attributes, start_node, goal_node):
    def euclidean_distance(a, b):
        x1, y1 = node_attributes[a]['x'], node_attributes[a]['y']
        x2, y2 = node_attributes[b]['x'], node_attributes[b]['y']
        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

    def heuristic(node):
        return euclidean_distance(start_node, node) + euclidean_distance(node, goal_node)

    def reconstruct_path(parent, curr):
        path = []
        while curr is not None:
            path.append(curr)
            curr = parent[curr]
        path.reverse()
        return path

    # Main code begins
    frontier = []
    g_scores = {start_node: 0}
    parent = {start_node: None}
    heapq.heappush(frontier, (g_scores[start_node] + heuristic(start_node), start_node))

    while frontier:
        _, current = heapq.heappop(frontier)

        if current == goal_node:
            return reconstruct_path(parent, goal_node)
```

```
    for neighbor, is_connected in enumerate(adj_matrix[current]):
        if is_connected:
            # tentative_g_score = g_scores[current] + euclidean_distance(current, neighbor)
            tentative_g_score = g_scores[current] + adj_matrix[current][neighbor]

            if neighbor not in g_scores or tentative_g_score < g_scores[neighbor]:
                g_scores[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor)
                heapq.heappush(frontier, (f_score, neighbor))
                parent[neighbor] = current

return []
```

```
def get_bidirectional_heuristic_search_path(adj_matrix, node_attributes, start_node, goal_node):

    # Function to calculate Euclidean distance b/w two points
    def euclidean_distance(a, b):
        x1, y1 = node_attributes[a]['x'], node_attributes[a]['y']
        x2, y2 = node_attributes[b]['x'], node_attributes[b]['y']
        return math.sqrt((x1 - x2)**2 + (y1 - y2)**2)

    # Function to calculate heuuristic h(w) = dist(u,w) + dist(v,w)
    def heuristic(node, start, goal):
        return euclidean_distance(start, node) + euclidean_distance(node, goal)

    # A star function to search both sides once at a time
    def a_star_directional(frontier, visited_this_side, visited_other_side, g_scores_this_side, new_start, new_goal, is_forward):
        _, current = heapq.heappop(frontier)

        if current in visited_other_side:
            return reconstruct_path(visited_this_side, visited_other_side, current, is_forward)

        for neighbor, is_connected in enumerate(adj_matrix[current]):
            if is_connected:
                # tentative_g_score = g_scores_this_side[current] + euclidean_distance(current, neighbor)
                tentative_g_score = g_scores_this_side[current] + adj_matrix[current][neighbor]

                if neighbor not in g_scores_this_side or tentative_g_score < g_scores_this_side[neighbor]:
                    g_scores_this_side[neighbor] = tentative_g_score
                    f_score = tentative_g_score + heuristic(neighbor, new_start, new_goal)
                    heapq.heappush(frontier, (f_score, neighbor))
                    visited_this_side[neighbor] = current

        return None
```

```python
def reconstruct_path(visited_from_start, visited_from_goal, meeting_node, is_forward):
    path_start = []
    current = meeting_node
    while current is not None:
        path_start.append(current)
        current = visited_from_start[current]

    path_goal = []
    current = visited_from_goal[meeting_node]
    while current is not None:
        path_goal.append(current)
        current = visited_from_goal[current]

    if is_forward:
        return path_start[::-1] + path_goal
    else:
        return path_goal[::-1] + path_start


# Main code begins
frontier_start = []
frontier_goal = []

g_score_start = {start_node: 0}
g_score_goal = {goal_node: 0}

heapq.heappush(frontier_start, (heuristic(start_node, start_node, goal_node), start_node))
heapq.heappush(frontier_goal, (heuristic(goal_node, goal_node, start_node), goal_node))
```

```python
visited_from_start = {start_node: None}
visited_from_goal = {goal_node: None}

while frontier_start and frontier_goal:
    path = a_star_directional(frontier_start, visited_from_start, visited_from_goal, g_score_start, start_node, goal_node, True)
    if path:
        return path

    path = a_star_directional(frontier_goal, visited_from_goal, visited_from_start, g_score_goal, goal_node, start_node, False)
    if path:
        return path

return []
```

(e)

Public Test Case 1:

```
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, 6, 2]
```

Public Test Case 2:

```
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, 98, 12]
```

Public Test Case 3:

```
Enter the start node: 12
Enter the end node: 49
Iterative Deepening Search Path: []
Bidirectional Search Path: []
A* Path: []
Bidirectional Heuristic Search Path: []
```

Public Test Case 4:

```
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, 34, 33, 11, 32, 31, 3, 5, 97, 28, 10, 12]
```

We are getting the different answer/path for all the public test cases except the third one where no path exists in both the algorithm.

No, it will not be identical for all the test cases for the given graph because of the following reasons:

1. **For Identical, If the graph has a unique shortest path from source u to destination v, both A* and Bidirectional A* will return the same path. Since both algorithms are optimal and guarantee the shortest path in the case of an admissible heuristic, they will find the same path if no alternative paths of the same length exist.**
2. **For different, If the graph has multiple shortest paths of the same length, the paths returned by A* and Bidirectional A* can be different. A* expands nodes in a unidirectional manner, which means it may find one of the multiple shortest paths earlier than Bidirectional A*. Bidirectional A*, on the other hand, searches from both directions and will find the meeting point where the searches overlap. This could result in it identifying a different shortest path than A*.**

Therefore, the path between all the pair between nodes are listed below using both algorithm.

(1) Using A* Search:

```
Start Node:0 and Goal Node:0 are same, so no computation is required

Start Node:0 and Goal Node:1:
path:[]
cost:inf
Memory Used (in bytes):1555
Execution Time (in sec):0.0003887999919243157

Start Node:0 and Goal Node:2:
path:[]
cost:inf
Memory Used (in bytes):1499
Execution Time (in sec):0.0005071000196039677

Start Node:0 and Goal Node:3:
path:[]
cost:inf
Memory Used (in bytes):1499
Execution Time (in sec):0.0002473000204190612

Start Node:0 and Goal Node:4:
path:[]
cost:inf
Memory Used (in bytes):1499
Execution Time (in sec):0.00023199996212497354

Start Node:0 and Goal Node:5:
path:[]
cost:inf
Memory Used (in bytes):1499
Execution Time (in sec):0.00023090001195669174
```

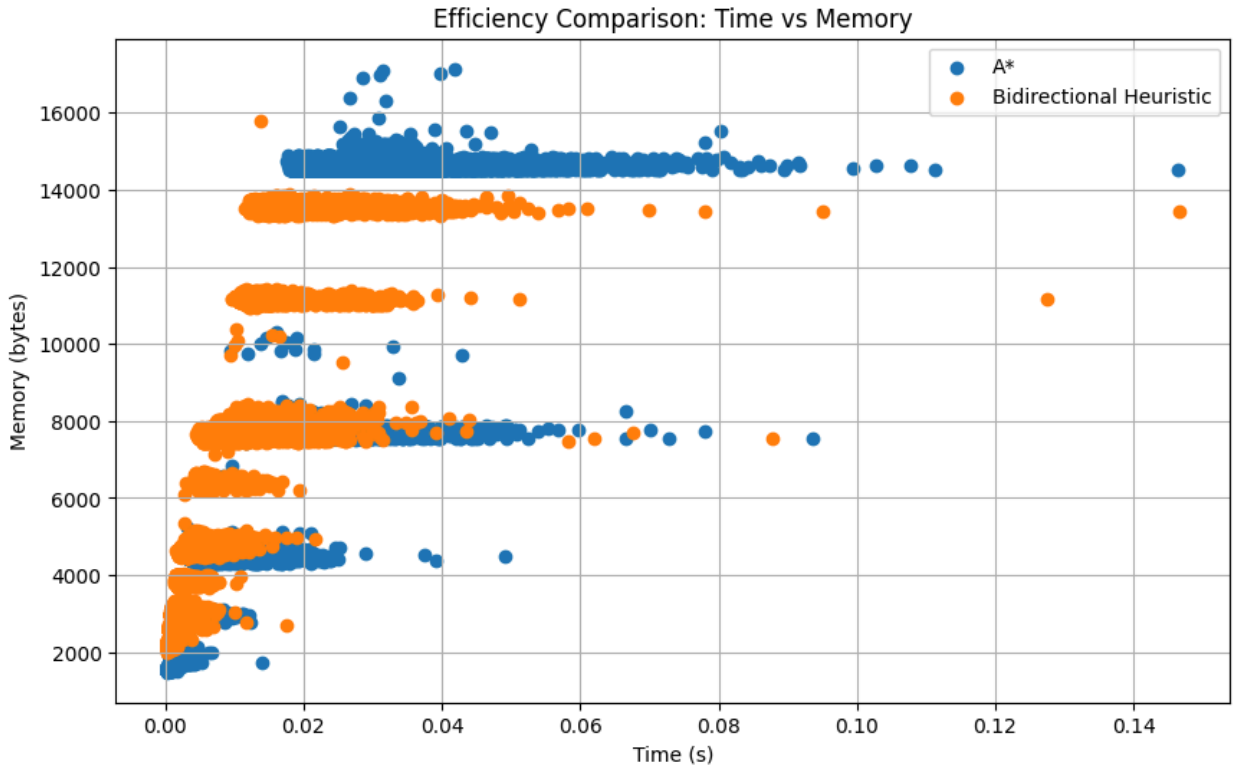
Use this link to view the full .txt file.

(2) Using Bidirectional A* Search:

```
Start Node:0 and Goal Node:0 are same, so no computation is required

Start Node:0 and Goal Node:1:
path:[]
cost:inf
Memory Used (in bytes):2195
Execution Time (in sec):0.0006271000020205975

Start Node:0 and Goal Node:2:
path:[]
cost:inf
Memory Used (in bytes):2195
Execution Time (in sec):0.00062959996284917

Start Node:0 and Goal Node:3:
path:[]
cost:inf
Memory Used (in bytes):2195
Execution Time (in sec):0.00036399997770786285

Start Node:0 and Goal Node:4:
path:[]
cost:inf
Memory Used (in bytes):2147
Execution Time (in sec):0.00033740000799298286

Start Node:0 and Goal Node:5:
path:[]
cost:inf
Memory Used (in bytes):2195
Execution Time (in sec):0.00034959998447448015
```

Use this link to view the full .txt file.

Now, we can see the comparison between these algorithm using the above result to plot a scatterplot of memory vs time which is given below:

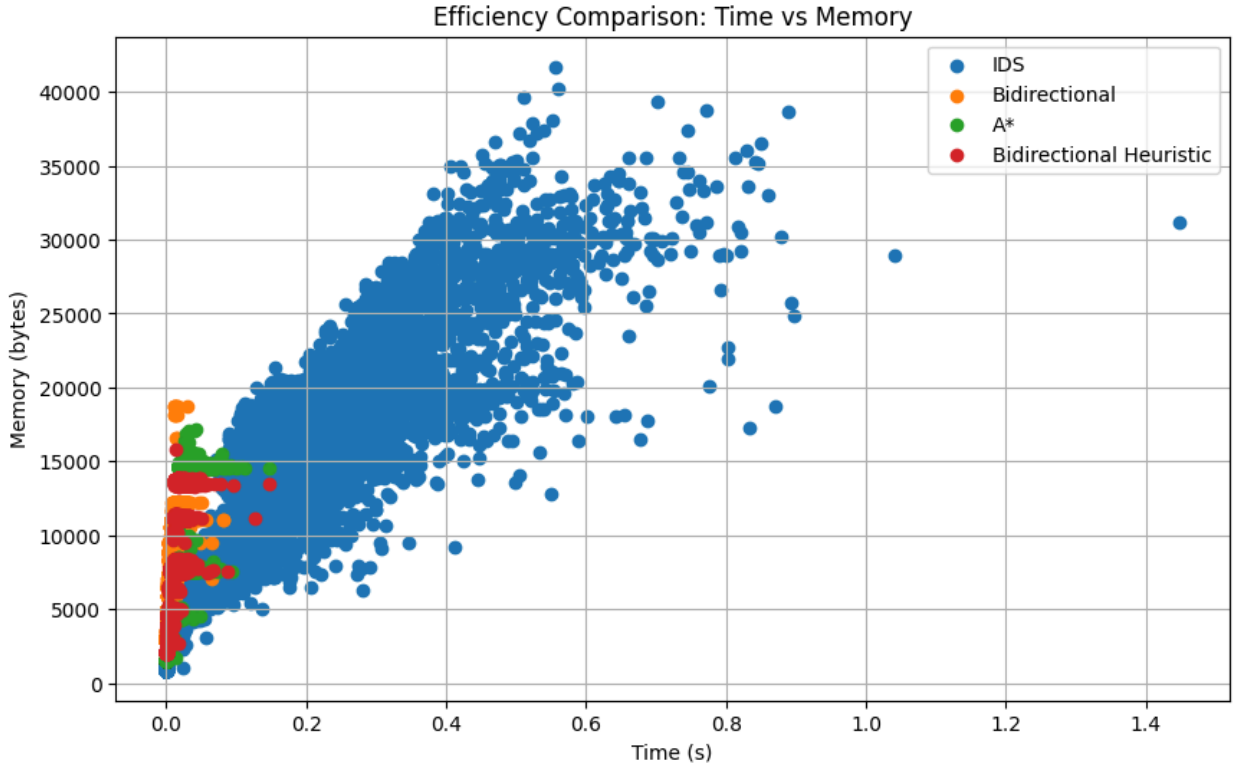


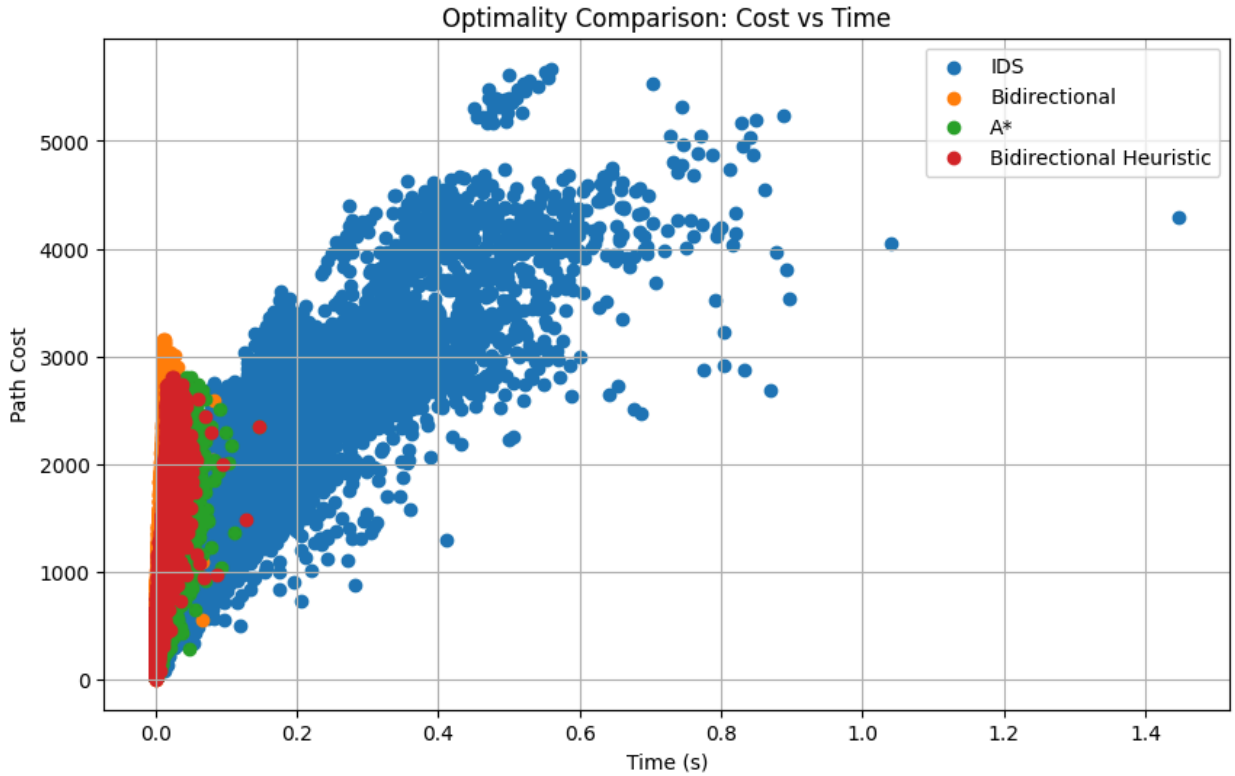
(f)

For analyzing the results obtained using informed and uninformed searches above, we plotted scatter plot of memory vs time for efficiency comparison and cost v/s time for optimaltiy comparison.

The data to generate the scatter plot was obtained by running the 4 algorithms to find their corresponding best path, execution time, memory usage, path cost for all pair of nodes and then that information is used to plot the above scatter plot. The is the link to see the same code.

Metrics Used for Scatter Plots:
- Time (X-Axis): The total computation time taken to find the path from the source to the goal in seconds.
- Memory (Y-Axis in Plot 1): The amount of memory used by the algorithm, measured in bytes.
- Path Cost (Y-Axis in Plot 2): The cost of the path found, which is the sum of edge weights between nodes from the source to the goal.


Efficiency Comparison: Time vs Memory
IDS
Bidirectional
A*
Bidirectional Heuristic
Memory (bytes)
Time (s)
0
5000
10000
15000
20000
25000
30000
35000
40000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4


Optimality Comparison: Cost vs Time
IDS
Bidirectional
A*
Bidirectional Heuristic
Path Cost
Time (s)
0
1000
2000
3000
4000
5000
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4

Plot 1: Efficiency Comparison (Time vs Memory):

- IDS is the least memory efficient, as seen by the spread of points higher on the Y-axis. It consumes more memory for the same time as compared to the informed algorithms.

- Bidirectional BFS search consumes less memory compared to IDS and tends to complete faster. It reduces the search space by searching from both directions (start and goal), which optimizes both time and memory.

- A* search is faster and uses less memory than IDS, but it still consumes more memory compared to Bidirectional search.

- Bidirectional Heuristic is the most efficient algorithm both in terms of memory and time. The points are clustered at the lower end of both memory and time scales, indicating that it is the fastest and most memory-efficient among all algorithms.

Conclusion: In terms of efficiency, Bidirectional Heuristic is the best followed by A*, Bidirectional BFS and IDS.

Plot 2: Optimality Comparison (Cost vs Time):

- IDS is spread across the plot, indicating that it takes longer to complete and finds paths with varying costs. IDS is uninformed and explores all possible paths, which leads to higher path costs and higher computation times.

- Bidirectional BFS Search generally finds lower-cost paths faster compared to IDS but is not as efficient as A* or Bidirectional Heuristic in finding optimal paths. It shows better performance with a tighter cluster of points in the lower cost region.

- A* is much more optimal in terms of path cost, as seen by the lower Y-axis values. It efficiently balances between path cost and time, making it one of the most effective algorithms for finding shorter paths in less time.

- Bidirectional Heuristic again performs the best, finding the optimal path (lowest cost) in the shortest time. It clusters near the lowest time and cost regions, indicating it consistently finds shorter paths quickly by utilizing the heuristic function from both directions.

Conclusion: In terms of optimality, Bidirectional Heuristic consistently finds the best (lowest-cost) path faster than the others, followed by A*, Bidirectional BFS and IDS.

Hence, Informed search algorithms like A* and Bidirectional Heuristic use heuristics to guide the search, resulting in faster execution and more optimal paths with lower

memory usage compared to uninformed searches like IDS. They efficiently focus on promising paths, reducing unnecessary exploration. Uninformed algorithms, though simple and heuristic-independent, tend to be slower and less efficient in finding optimal paths, as shown in the empirical analysis.

(g) Bonus

```python
def bonus_problem(adj_matrix):
    n = len(adj_matrix)

    tin = [-1]*n
    mn = [-1]*n
    parent = [-1]*n
    timer = [0]

    res = []
    def dfs(u):
        tin[u] = mn[u] = timer[0]
        timer[0] += 1

        for v in range(n):
            if adj_matrix[u][v]:
                if tin[v] == -1:
                    parent[v] = u
                    dfs(v)

                    mn[u] = min(mn[u], mn[v])
                    if mn[v] > tin[u]:
                        res.append((u, v))

                elif tin[v] != -1 and v != parent[u]:
                    mn[u] = min(mn[u], tin[v])

    for i in range(n):
        if tin[i] == -1:
            dfs(i)

    return res
```

Result of the Bonus part for given graph:

```
Bonus Problem: [(42, 29), (42, 30), (113, 42), (113, 43), (15, 46), (35, 15), (114, 84), (36, 114), (38, 36), (87, 88), (69, 124), (41, 70), (45, 17), (89, 90), (51, 50
), (39, 40), (73, 72), (19, 100), (106, 107), (108, 109), (108, 112), (111, 108), (111, 110), (106, 111), (75, 106), (55, 56), (12, 57), (53, 54), (95, 96), (53, 95), (
14, 53), (14, 99), (47, 48)]
```

Time Complexity: O(V+E)

References:

(1). Find brides in a graph

(2). AI playlist of GATE smashers for A*, IDS, Bidirectional BFS, Minmax pruning

(3) AI playlist by Mahesh Huddar for Iterative deepening A*, Bidirectional Heuristic

(4) Lecture slides posted on Google Classroom

(5) Course Book: RussellNorvig_AIMA_4thEdition