RBE550: Motion Planning

Project Template

Student 1: First Name, Last Name
Student 2: First Name, Last Name

1. **Q1**

   Difference 1 Approach to Leaving the Obstacle: In the Bug1 algorithm, the robot circumnavigates the obstacle until it finds the point on the obstacle's perimeter that is closest to the goal, known as the "leave point." This means the robot may traverse a significant portion of the obstacle's perimeter before resuming its path to the goal. In contrast, Bug2 employs the concept of an "M-Line," which is a straight line drawn from the starting point to the goal. The robot leaves the obstacle as soon as it intersects with the M-Line again, and it seems feasible to resume a direct path to the goal.

   Difference 2 Path Efficiency: The Bug1 algorithm finds the closest point to the goal on the obstacle's perimeter, which can lead to less efficient paths, especially in environments with complex or irregularly shaped obstacles. It can also result in the robot traversing the same obstacle multiple times if the closest point to the goal is not in a direct line of sight. In Bug2, the use of the "M-Line" often leads to more direct and potentially more efficient paths towards the goal. However, Bug2 can still encounter difficulties in environments with certain types of obstacles, such as concave shapes, where it might only get local optimal.

2. **Q2**

   (a)
   Admissible: We can use the Euclidean distance from a node to the goal, defined by the coordinates (x, y) for the current node and (g_x, g_y) for the goal. Then we have the $h_a(n) = \sqrt{(x - g_x)^2 + (y - g_y)^2}$. This heuristic is admissible because it never overestimates the true cost. The shortest path between two points in a plane is a straight line, and in a grid world where diagonal movements are allowed, the Euclidean distance will always be less than or equal to the true cost of reaching the goal.

   Non-Admissible: For a non-admissible heuristic h_b, any heuristic that overestimates the true cost could be considered. We introduce a scaling constant factor k, which $k > 1$. Then we have $h_b(n) = k \cdot \sqrt{(x - g_x)^2 + (y - g_y)^2}$, make it non-admissible.

   (b)
   Let h_1 and h_2 be consistent heuristics. We define a new heuristic h as: $h(n) = \max(h_1(n), h_2(n))$. To prove h is consistent, we must show for all nodes n and n': $h(n) \leq T(n, n') + h(n')$.
   Since h_1 and h_2 are consistent, for any nodes n and n':

$$h_1(n) \leq T(n, n') + h_1(n')$$
$$h_2(n) \leq T(n, n') + h_2(n')$$

   By the definition of h, for any node n, $h(n)$ is either $h_1(n)$ or $h_2(n)$, whichever is greater. Thus, we can say: $h(n) \leq T(n, n') + h(n')$ for both h_1 and h_2, and therefore:

$$h(n) = \max(h_1(n), h_2(n)) \leq T(n, n') + \max(h_1(n'), h_2(n'))$$

   Because the maximum of two values is less than or equal to the sum of the two values: $\max(h_1(n'), h_2(n')) \leq h_1(n') + h_2(n')$. So we can write: $h(n) \leq T(n, n') + h_1(n') + h_2(n')$. Which implies:

$$h(n) \leq T(n, n') + \max(h_1(n'), h_2(n'))$$

   Since $h(n')$ is $\max(h_1(n'), h_2(n'))$, we have:

$$h(n) \leq T(n, n') + h(n')$$

   Hence, the heuristic h satisfies the consistency condition for all n, n', proving that h is consistent.

3. **Q3**

(a)

The construction of a visibility graph is checking each pair of vertices to determine if they are visible to each other, that is if the line segment between them doesn't intersect any obstacle. For each pair of vertices, we need to check against all edges of the obstacles for a possible intersection. If there are n vertices, there are $\binom{n}{2}$ or $O(n^2)$ pairs of vertices to check. For each pair, we need to check for intersection against all edges. If there are n vertices, there could be up to $O(n)$ edges. Therefore, the upper bound for constructing the visibility graph is $O(n^3)$.

Algorithm 1 Construct Visibility Graph

```
1: function ConstructVisibilityGraph(vertices)
2:     graph ← EmptyGraph
3:     for each v1 in vertices do
4:         for each v2 in vertices do
5:             if v1 ≠ v2 and IsVisible(v1, v2, vertices) then
6:                 AddEdge(graph, v1, v2)
7:     return graph
8: function IsVisible(v1, v2, vertices)
9:     for each edge in Edges(vertices) do
10:        if Intersects(v1, v2, edge) then
11:            return false
12:    return true
```

(b)

The visibility graph can be used to plan a path from the start to the goal. Once the graph is constructed, any standard graph search algorithm like $A*$ and BFS for an unweighted graph can be used to find the shortest path from the start to the goal. Using $A*$, the time complexity would be $O(n(\log n) + m)$. The BFS is $O(n + m)$ since each vertex and edge is processed once in an unweighted graph.

Algorithm 2 Dijkstra's Algorithm

```
1: function Dijkstra(graph, start, goal)
2:     dist ← Map(vertex → ∞)
3:     prev ← Map(vertex → null)
4:     dist[start] ← 0
5:     priorityQueue ← NewPriorityQueue
6:     Add(priorityQueue, start, priority ← 0)
7:     while not IsEmpty(priorityQueue) do
8:         u ← Pop(priorityQueue)
9:         if u = goal then
10:            break
11:        for each v in Neighbors(graph, u) do
12:            alt ← dist[u]+ Distance(graph, u, v)
13:            if alt < dist[v] then
14:                dist[v] ← alt
15:                prev[v] ← u
16:                AddWithPriority(priorityQueue, v, alt)
17:    return ConstructPath(prev, goal)
18: function ConstructPath(prev, goal)
19:    path ← EmptyList
20:    u ← goal
21:    while prev[u] ≠ null do
22:        Prepend(path, u)
23:        u ← prev[u]
24:    return path
```

Algorithm 3 Breadth-First Search

```
1: function BFS(graph, start, goal)
2:     queue ← NewQueue
3:     visited ← NewSet
4:     Enqueue(queue, start)
5:     Add(visited, start)
6:     prev ← Map(vertex → null)
7:     while not IsEmpty(queue) do
8:         node ← Dequeue(queue)
9:         if node = goal then
10:            return ConstructPath(prev, goal)
11:        for each neighbor in Neighbors(graph, node) do
12:            if neighbor ∉ visited then
13:                Enqueue(queue, neighbor)
14:                Add(visited, neighbor)
15:                prev[neighbor] ← node
16:    return EmptyList
17: function ConstructPath(prev, goal)
18:    path ← EmptyList
19:    node ← goal
20:    while node ≠ null do
21:        Prepend(path, node)
22:        node ← prev[node]
23:    return path
```

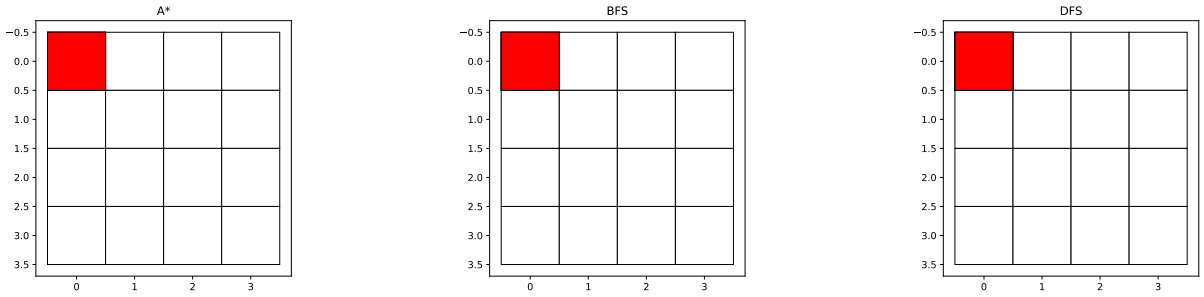
4. Q4

(a)

All three algorithms can be seen as graph traversal techniques that can be abstractly managed by a data structure to handle the open set: a queue for BFS, a stack for DFS, and a priority queue for $A*$.

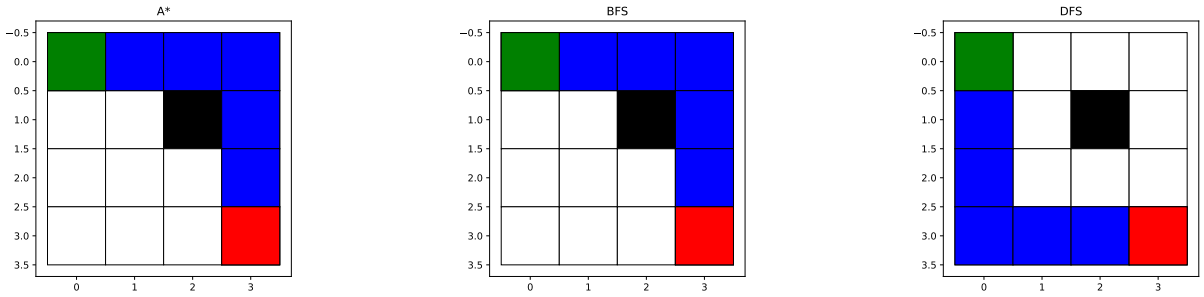
(c)

Boundary Case 1: start and goal at the same location



(a) (b) (c)

Boundary Case 2: start or goal at the edge or corner



(a) (b) (c)

Large map test