Dijkstra Algorithm

Dijkstra's algorithm is a graph search algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. It uses a priority queue to keep track of the vertices with the shortest distance from the source vertex and updates the distances of its neighbors as it traverses the graph.

Here is a pseudocode for Dijkstra's algorithm:

```
function Dijkstra(Graph, source):
  create a set of unvisited vertices
  for each vertex in Graph:
    if vertex is not source:
      set distance to infinity
      add vertex to unvisited set
    else:
      set distance to 0
  create a priority queue to store the vertices based on their distance
  add the source vertex to the queue
  while the queue is not empty:
    u = vertex with the shortest distance in the queue
    remove u from the queue
    add u to the visited set
    for each neighbor v of u:
      if v is not in the visited set:
        temp_distance = distance of u + weight of (u, v)
        if temp_distance is less than the distance of v:
          update the distance of v to temp_distance
          add v to the queue
  return the distances
```

The algorithm starts by setting the distance of the source vertex to zero and all other vertices to infinity. The unvisited vertices are then added to a priority queue and sorted based on their distances from the source vertex. The algorithm then removes the vertex with the shortest distance from the queue and updates the distances of its neighbors if

a shorter path is found. The process is repeated until all vertices have been visited. The final result is an array of distances from the source vertex to all other vertices in the graph.

A* search

A* (A-Star) is a heuristic search algorithm that is used to find the shortest path between two nodes in a graph. It combines the use of a heuristic function, which provides a lower bound on the cost of reaching the goal node, and a best-first search strategy to find the optimal solution.

The algorithm works as follows:

1. Start by initializing an empty priority queue and adding the starting node to it.

While the priority queue is not empty:
a. Extract the node with the lowest f(n) value from the priority queue, where f(n) = g(n) + h(n), where g(n) is the cost of reaching the node from the starting node, and h(n) is the heuristic estimate of the cost of reaching the goal node from n.
b. If the extracted node is the goal node, return the path from the starting node to the goal node.
c. For each of the neighboring nodes, calculate their g(n) values and calculate their f(n) values.

2. d. If the neighboring node is not in the priority queue or if the new f(n) value is lower than the current f(n) value, update the f(n) value and add it to the priority queue.

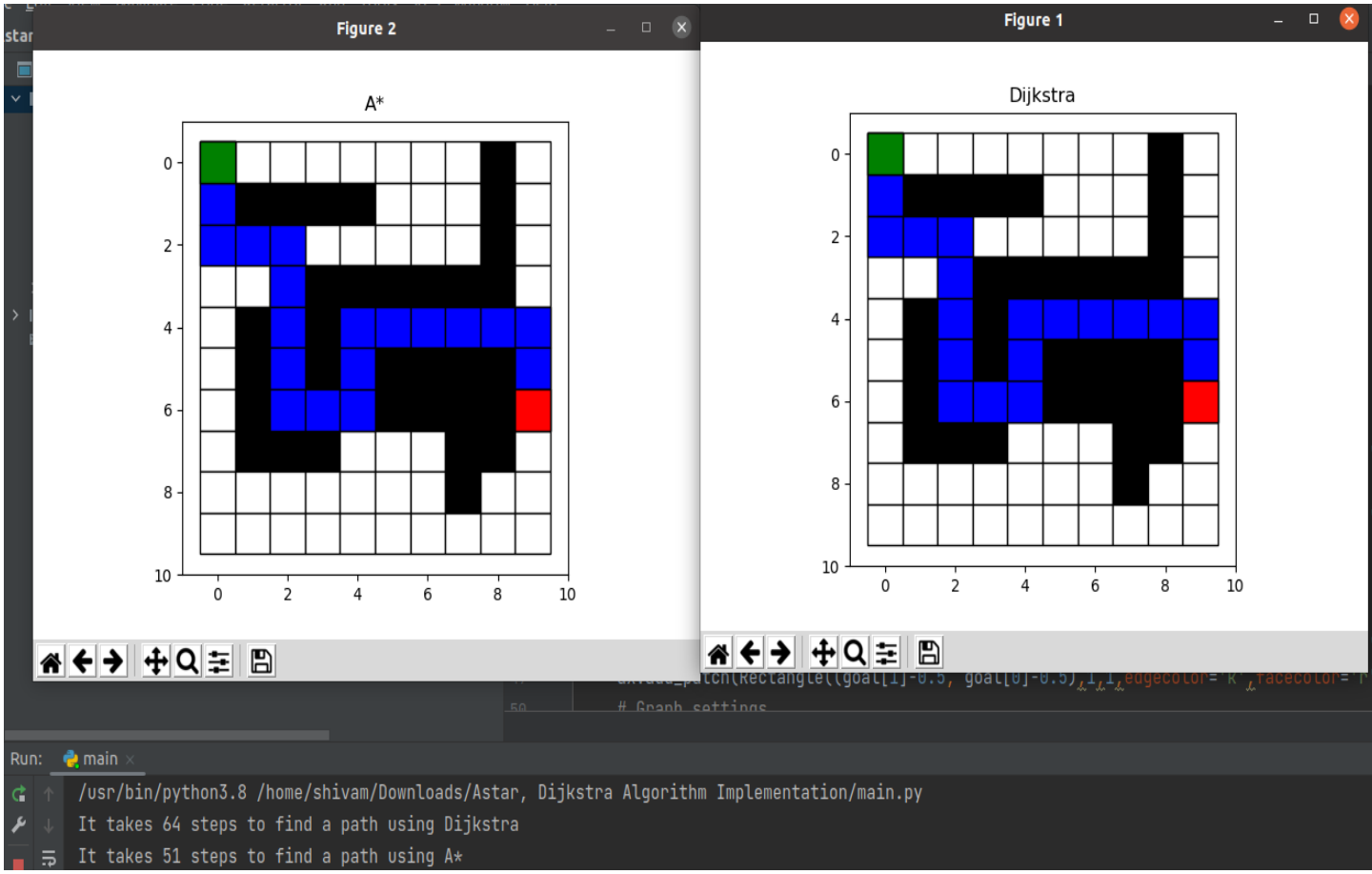
Pseudo-code for the A* algorithm:

```
function A*(start, goal)
  // Initialize an empty priority queue and add the start node to it
  priority_queue pq
  pq.add(start, 0)
  // Keep a map to store the g(n) value for each node
  g_values = { start: 0 }
  // Keep a map to store the parent of each node
  parent = { start: null }

  while pq is not empty
    // Extract the node with the lowest f(n) value from the priority queue
    current = pq.extract_min()
    // If the current node is the goal node, return the path
    if current == goal
      return construct_path(parent, goal)
    // For each of the neighboring nodes
    for each neighbor in current.neighbors
      // Calculate the g(n) value for the neighbor
      g_value = g_values[current] + cost(current, neighbor)
      // If the neighbor is not in the g_values map or if the new g_value is
      if neighbor not in g_values or g_value < g_values[neighbor]
        // Update the g(n) value and add the neighbor to the priority queue
        g_values[neighbor] = g_value
        h_value = heuristic(neighbor, goal)
        f_value = g_value + h_value
        pq.add(neighbor, f_value)
        parent[neighbor] = current

function construct_path(parent, node)
  if parent[node] == null
    return [node]
  else
    return construct_path(parent, parent[node]) + [node]
```

Map.csv output



1. Dijkstra always give shortest path

2. A star is guaranteed to find the optimal solution if and only if the heuristic function used is both admissible and consistent. Admissibility means that the heuristic must never overestimate the actual cost to reach the goal, while consistency means that the difference between the cost to reach a neighbor and the estimated cost to reach the goal through that neighbor must be less than or equal to the estimated cost of reaching the goal from the current node. If the heuristic function satisfies both of these properties, A star will always find the shortest path.
3. In our case, A star heuristic function is both admissible and consistent. Therefore, the path is the same as Dijkstra.
4. A* being faster and more efficient in finding the shortest path in many cases. Hence the less number of steps. It is because A star uses heuristic.

References

Chat GPT