Dijkstra Algorithm

Input:

grid (2D array), start (node), end (node)
Output:

PathResult containing shortest_path and step_info

```
DISTANCES = ∞ for all nodes, DISTANCES[start] = 0
PRIORITY_QUEUE = [(0, start)]
VISITED = empty set
PARENT = empty map
STEP_INFO = empty dictionary
CURRENT_LEVEL = 0

while PRIORITY_QUEUE is not empty do
   (distance_current, current) = POP(PRIORITY_QUEUE)

   if current in VISITED then
       continue
   VISITED = VISITED ∪ {current}

   if current = end then
       break

   for each neighbor in VALID_NEIGHBORS(grid, current) do
       if neighbor in VISITED then
          continue
       tentative_distance = DISTANCES[current] + 1

       if tentative_distance < DISTANCES[neighbor] then
          DISTANCES[neighbor] = tentative_distance
          PARENT[neighbor] = current
          PUSH(PRIORITY_QUEUE, (tentative_distance, neighbor))
          STEP_INFO[CURRENT_LEVEL][current] = VALID_NEIGHBORS

   CURRENT_LEVEL = CURRENT_LEVEL + 1

shortest_path = RECONSTRUCT_PATH(PARENT, end)
return PathResult(shortest_path, STEP_INFO)
```

Astar Algorithm

Input:

grid (2D array), start (node), end (node)

Output:

PathResult containing shortest_path and step_info

```
OPEN_SET = priority queue with (0, start)
G_SCORE[start] = 0
F_SCORE[start] = HEURISTIC(start, end)
CAME_FROM = empty map
STEP_INFO = empty dictionary
CURRENT_LEVEL = 0

while OPEN_SET is not empty do
    (f_current, current) = POP(OPEN_SET) // Node with smallest f_score

    if current = end then
        shortest_path = RECONSTRUCT_PATH(CAME_FROM, current)
        ADD_INTERMEDIATE_POINTS(shortest_path)
        return PathResult(shortest_path, STEP_INFO)

    for each neighbor in NEIGHBORS(grid, current) do
        if neighbor is out of bounds or grid[neighbor] = obstacle then
            continue
        movement_cost = COST(current, neighbor)
        tentative_g_score = G_SCORE[current] + movement_cost

        if tentative_g_score < G_SCORE[neighbor] then
            CAME_FROM[neighbor] = current
            G_SCORE[neighbor] = tentative_g_score
            F_SCORE[neighbor] = tentative_g_score + HEURISTIC(neighbor, end)
            PUSH(OPEN_SET, (F_SCORE[neighbor], neighbor))
            STEP_INFO[CURRENT_LEVEL][current] = NEIGHBORS

    CURRENT_LEVEL = CURRENT_LEVEL + 1

return PathResult([], STEP_INFO)
```

Jump Point Search Algorithm

Input:

grid (2D array), start (node), end (node)
Output:

PathResult containing shortest_path and step_info

```
OPEN_SET = [(0, start)]
G_SCORE[start] = 0
F_SCORE[start] = HEURISTIC(start, end)
CAME_FROM = empty map
CAME_FROM_PATH = empty map
CLOSED_SET = empty set
STEP_INFO = empty dictionary
CURRENT_LEVEL = 0

while OPEN_SET is not empty do
    (f_current, current) = POP(OPEN_SET)

    if current = end then
        path = RECONSTRUCT_PATH(CAME_FROM_PATH, start, end)
        expanded_path = EXPAND_DIAGONAL_MOVES(path)
        if expanded_path is valid then
            return PathResult(expanded_path, STEP_INFO)
        continue

    if current in CLOSED_SET then
        continue
    CLOSED_SET = CLOSED_SET ∪ {current}

    successors = GET_SUCCESSORS(grid, current, end, CAME_FROM)
    for each (neighbor, path_segment, move_cost) in successors do
        if neighbor in CLOSED_SET then
            continue
        tentative_g_score = G_SCORE[current] + move_cost

        if neighbor not in G_SCORE or tentative_g_score < G_SCORE[neighbor] then
            CAME_FROM[neighbor] = current
            CAME_FROM_PATH[neighbor] = path_segment
            G_SCORE[neighbor] = tentative_g_score
            F_SCORE[neighbor] = tentative_g_score + HEURISTIC(neighbor, end)
            PUSH(OPEN_SET, (F_SCORE[neighbor], neighbor))
            STEP_INFO[CURRENT_LEVEL][current] = [neighbor]

    CURRENT_LEVEL = CURRENT_LEVEL + 1

return PathResult([], STEP_INFO)
```