Task 1: Global Planning

1.1 A* Algorithm

Implementation of Algorithm:

The A* algorithm combines the advantages of Breadth-First Search (BFS) and Greedy Search, efficiently finding the optimal path by evaluating the cost function $f(n)$. The function is defined as $f(n) = h(n) + g(n)$, where $g(n)$ represents the cost of the traveled path, and $h(n)$ estimates the cost of the remaining path. The heuristic function $h(n)$ is calculated using either the Manhattan distance or the Euclidean distance. The cost of moving one step horizontally or vertically is set to 0.2, while the cost of moving diagonally is 0.282. The specific algorithm implementation can be found in the appendix.

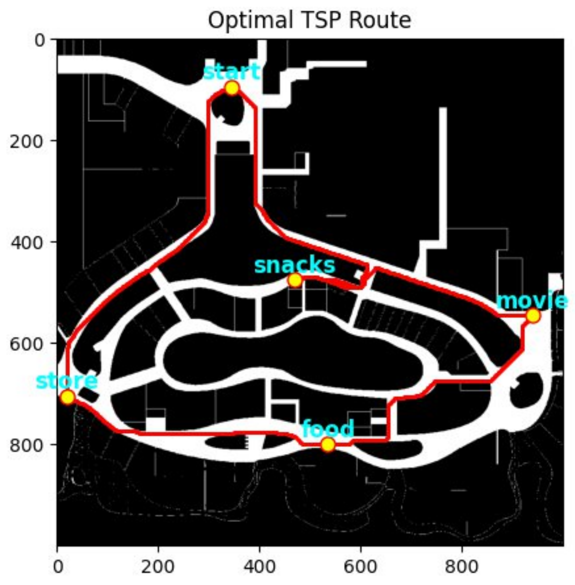
Since the human body has volume, it is necessary to inflate the original map. Given that the human body has a radius of 0.3m and each cell is 0.2m, a buffer zone of two cells must be added around all obstacle regions. This buffer zone is considered impassable to prevent collisions. The inflation algorithm is provided in the appendix.

Once the inflated map and the A* algorithm are obtained, path planning can be performed for every pair of locations in the location dictionary. This process results in a table containing the shortest paths between each pair of locations.

By traversing the table, the total distance of each path covering all required points is calculated. The path with the shortest total distance is identified as the most efficient route.

Algorithm Result:

The following results can be seen directly in the ipynb file.

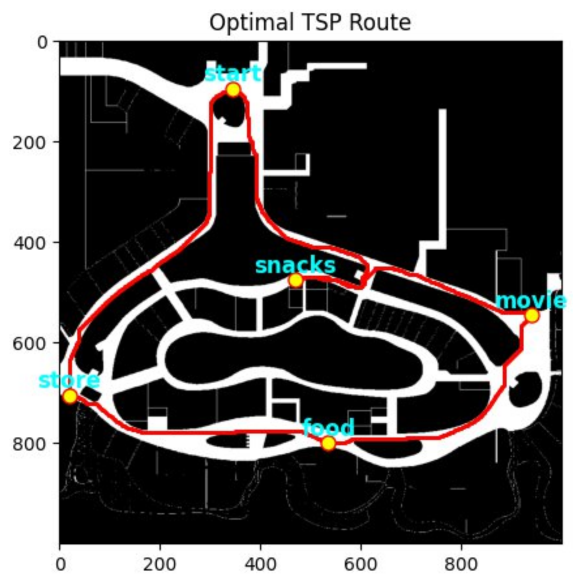


Manhattan Distance Version:

Total Distance (m): 633.48

The total run time (s): 0.64

Optimal Route: start -> store -> food -> movie -> snacks -> start



Euclidean Distance Version:

Total Distance (m): 632.66

Total runtime (s): 1.05

Optimal Route: start -> store -> food -> movie -> snacks -> start

1.2 Dijkstra's Algorithm

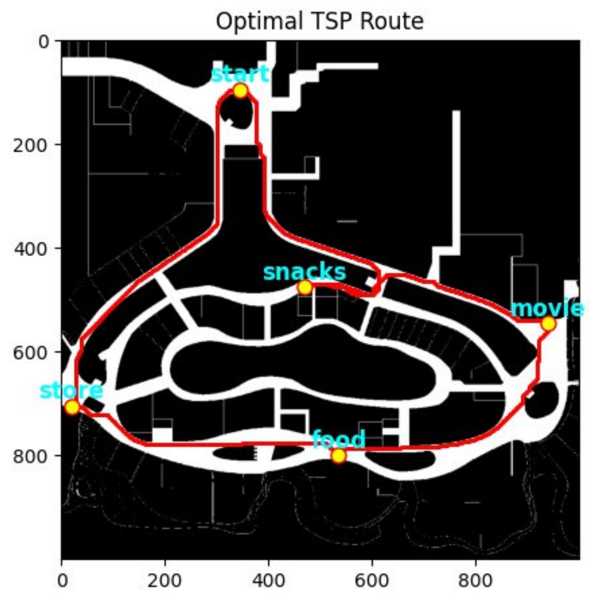
Implementation of Algorithm:

Dijkstra's Algorithm is a degenerate version of the A* algorithm, as it only considers the cost of the traversed path (i.e. $g(n)$) without using a heuristic function to save computational resources. As a result, it wastes time exploring unpromising directions. For the specific code, please refer to the appendix.

The generation of the expanded map and the method for traversing points are the same as described above.

Algorithm Result:

The following results can be seen directly in the ipynb file.



Total Distance (m): 633.84

The total run time (s): 3.94

Optimal Route: start -> snacks -> movie -> food -> store -> start

1.2 Comparation

In fact, we can observe that the planned routes obtained by the three methods are quite similar. However, A* is significantly faster than Dijkstra, which is expected.

If the Euclidean distance is used as the heuristic function, the planned path will be smoother. On the other hand, if the Manhattan distance is used, the algorithm tends to favor horizontal or vertical movements, which may result in a detour when moving from the "movie" node to the "food" node.

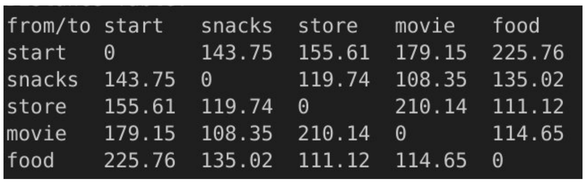
1.3 Difficulties and Solutions

Due to a lack of familiarity with the algorithm, I initially forgot to multiply the distance by the cost in the heuristic function, which resulted in incorrect path planning. It took me a long time to adjust and fix the issue.

1.4 Shortest Distances Between Each Pair of Locations

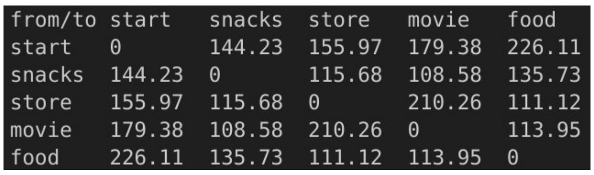
Use A* and Dijkstra's Algorithm to obtain the following distance table, selecting the minimum value at each position to fill in the table.

Distance Table produced by A* (Manhattan distance)



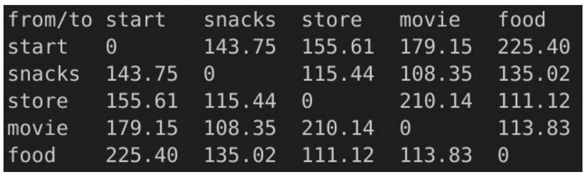
```
from/to start   snacks  store   movie   food
start   0       143.75  155.61  179.15  225.76
snacks  143.75  0       119.74  108.35  135.02
store   155.61  119.74  0       210.14  111.12
movie   179.15  108.35  210.14  0       114.65
food    225.76  135.02  111.12  114.65  0
```

Distance Table produced by Dijkstra



```
from/to start   snacks  store   movie   food
start   0       144.23  155.97  179.38  226.11
snacks  144.23  0       115.68  108.58  135.73
store   155.97  115.68  0       210.26  111.12
movie   179.38  108.58  210.26  0       113.95
food    226.11  135.73  111.12  113.95  0
```

Distance Table produced by A* (Euclidean distance)



```
from/to start   snacks  store   movie   food
start   0       143.75  155.61  179.15  225.40
snacks  143.75  0       115.44  108.35  135.02
store   155.61  115.44  0       210.14  111.12
movie   179.15  108.35  210.14  0       113.83
food    225.40  135.02  111.12  113.83  0
```

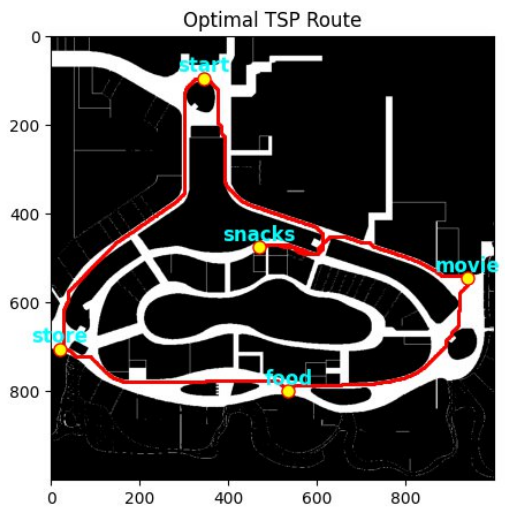
from / to	start	snacks	store	movie	food
start	0	143.75	155.61	179.15	225.76
snacks	143.75	0	115.44	108.35	135.02
store	155.61	115.44	0	210.14	111.12
movie	179.15	108.35	210.14	0	113.83
food	225.76	135.02	111.12	113.83	0

Task 2:

In Task 1, a preliminary solution to the "Travelling Shopper" problem was implemented using a traversal method. This approach exhaustively searches all node-to-node distances and brute-forces the shortest path. The corresponding code can be found in the appendix under the **Algorithm of Traversal Points** section.

Next, a greedy approach is used to solve the problem again, this method select the nearest unvisited point for the current point each time. The results are shown below, demonstrating that the greedy method computes faster while still obtaining the optimal path. This makes it more suitable for small-scale problems. Each algorithm was run 100,000 times to calculate the running time of the algorithm. The specific code for the greedy method can be found in the appendix.

Greedy method:	**Traversal method:**
Runtime (s): 1.85e-06	Runtime (s): 1.52e-05
Optimal Route: start -> snacks -> movie -> food -> store -> start	Optimal Route: start -> store -> food -> movie -> snacks -> start
Total Distance (m): 632.66	Total Distance (m): 632.66



Appendices:

Rough A* Algorithm:

```
def heuristic(a, b):
    # Manhattan distance
    return 0.2 *abs(a[0] - b[0]) + 0.2 *abs(a[1] - b[1])
    # Euclidean distance
    # dx = abs(a[0] - b[0])
    # dy = abs(a[1] - b[1])
    # return 0.2 * (dx + dy) + (0.282 - 2 * 0.2) * min(dx, dy)

frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next) # graph.cost is 0.282 when walk diagonal else 0.2
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

Rough Dijkstra's Algorithm:

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next) # graph.cost is 0.282 when walk diagonal else 0.2
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
      priority = new_cost
         frontier.put(next, priority)
         came_from[next] = current
```

Inflating Algorithm:

```
def expand_obstacles(grid, safety_radius = 0.3):
    expansion_radius = int(np.ceil(safety_radius / 0.2)) # compute how many cell need to be inflated (1.5 -> 2)
    struct = np.ones((2*expansion_radius+1, 2*expansion_radius+1)) # every obstacle inflate 2 cell outside (0.4m)
    expanded_grid = binary_dilation(grid == 0, structure=struct).astype(int) # if == 0, inflate
    return (255 - expanded_grid * 255)
```

Algorithm of Traversal Points:

```python
def solve_tsp(locations, distance_table):
    # point except start
    points = list(locations.keys())
    points.remove("start")
    best_route = None
    min_distance = float('inf')

    for perm in itertools.permutations(points):
        route = ["start"] + list(perm) + ["start"] # generate all possible path
        total = sum(distance_table[route[i]][route[i+1]] for i in range(len(route)-1)) # compute all possible length of path
        if total < min_distance:
            min_distance = total
            best_route = route
    return best_route, min_distance
```

Greedy Algorithm:

```python
def solve_tsp_greedy(locations, distance_table):
    unvisited = set(locations.keys())
    unvisited.remove("start")
    current = "start"
    route = ["start"]
    total_distance = 0

    while unvisited:
        # Select the nearest unvisited point for the current point each time
        next_point = min(unvisited, key=lambda p: distance_table[current][p])
        total_distance += distance_table[current][next_point]
        route.append(next_point)
        unvisited.remove(next_point)
        current = next_point

    # back start point
    total_distance += distance_table[current]["start"]
    route.append("start")

    return route, total_distance
```