Cobalt Traveling Robot Challenge

Chenhao Yang

Carnegie Mellon University

1 Problem Statement

Given a collection of points randomly distributed, the robot need to visit each of them. However, the robot need to "go home to recharge" every so often. The challenge is to find in the shortest path possible.

Details of the problem:

- There are 5000 points distributed uniformly in [0, 1]
- The recharge station is located at (.5, .5)
- You cannot travel more than 3 units of distance before recharging
- You must start and end at the recharge station

2 Methods & Implementation

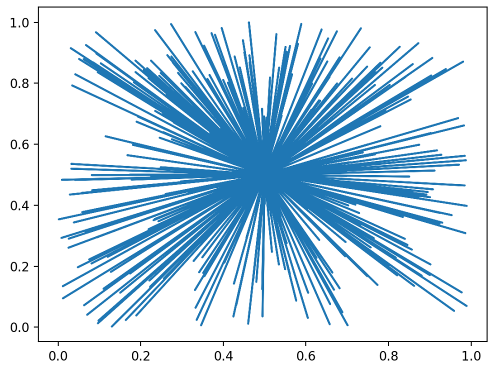
This problem resembles the classical NP-hard combinatorial optimization problem traveling salesman. We can find the exact optimal solution of traveling salesman problem by enumerating all possible combinations with $O(n!)$ time complexity or using dynamic programming to solve it with $O(n^2 2^n)$. Intuitively, we can inherit ideas from this problem to ours. In this sections, we present several methods for finding solutions for this problem and discuss trade-offs between different algorithms.

2.1 Naive Approach

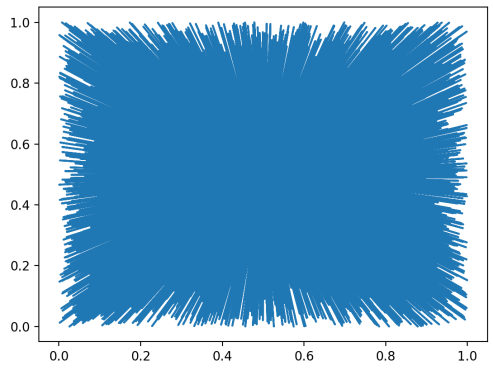
The naive approach is to visit one or several points once a time by random order (generated point order), this way the time complexity of generating such solution would just be linear since we are mainly copying the list. However, the downside is that we wasted a lot of path during 'reaching' and 'homing' process.

```python
def path_finder_naive(self):
  '''
  Naive approach to generate the order,
  smallest time complexity but not optimal for path at all
  '''
  self.order.append(0)
  for i in range(self.N):
    self.order.append(i+1)
    self.order.append(0)
```

Listing 1: Naive Approach



(a) 300 points



(b) 5000 points

Figure 1: Visualization of robot path using naive method

2.2 Sub-Traveling Salesman

To inherit the optimal solution of traveling salesman, we can divide the path we found to segments that can be reached by the robot. Therefore, this problem can be divided as:

1. Finding optimal solution of traveling salesman, using dynamic programming (memorization) method to achieve $O(n^2 2^n)$ time complexity;

2. let the robot following the optimal path, return for charging when needed;

To implement step 1, we can use python library `functools.lru_cache` for memorization and depth breadth first search (DFS) to recursively find all possible solutions.

```python
def path_finder_exact(self):
  '''
  Find exact solution of traveling salesman problem using memorization,
  and seperate path to segments that is robot-reachable/returnable
  '''
  # exact solution
  self.compute_distance_matrix()
  visited = frozenset([0])

  # step 1: Finding optimal solution of traveling salesman
  # DFS
  @lru_cache(maxsize=None)
  def DFS(curr, visited):
    if len(visited) == self.N+1:
      return (self.dist[curr][0], [curr])
    else:
      # enumerating all possible next point
      res = []
      for i in range(1, self.N+1):
        if i not in visited:
          # deep copy visited list and append next point
          ls_visited = list(visited)
          ls_visited.append(i)
          visited_copy = frozenset(ls_visited)

          d, order = DFS(i, visited_copy)

          # using heapq to always get the minimum distance path
          heapq.heappush(
            res,
            ( d + self.dist[curr][i], order)
          )

      distance, order = heapq.heappop(res)
      neworder = order.copy()
      neworder.append(curr)
      return (distance, neworder)

  opt_distance, opt_order = DFS(0, visited)

```

Listing 2: DFS with memorization

For implementing step 2, the only thing need to do is check if 'homing' needed and append 0s in the optimal path list.

```
# step 2: segment exact solution to sub paths
self.order = [0, opt_order[0]]
d = self.dist[0][opt_order[0]]
for i in range(self.N):
  cur_pt = opt_order[i]
  next_pt = opt_order[i+1]
  # check if homing needed
  if d + self.dist[cur_pt][next_pt]+ self.dist[next_pt][0] > self.max_charge:
    # return home
    self.order.append(0)
    d = self.dist[0][next_pt]
  else:
    d += self.dist[cur_pt][next_pt]
  self.order.append(next_pt)
```

Listing 3: segment path

However, our problem states that there are 5000 points randomly distributed in the workspace, the problem is still too complex to solve with reasonable time, not mentioning huge memory it takes. So just for checking my implementation, here I include the result of path using only 15 points.

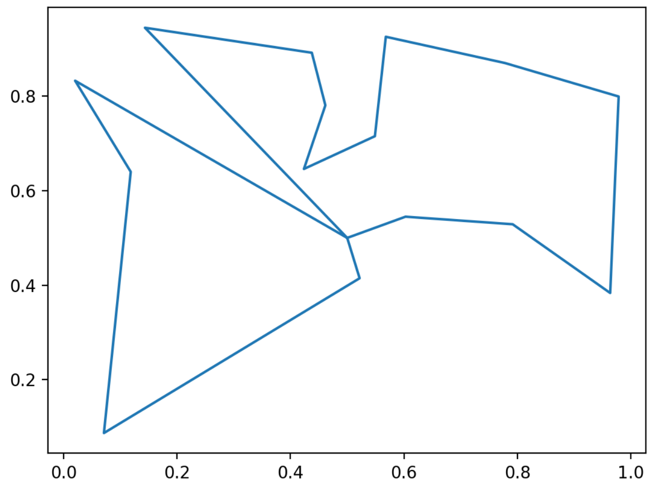


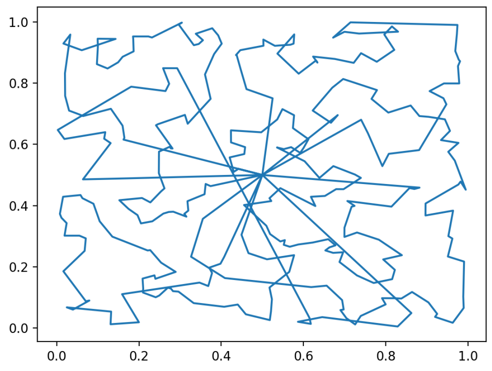
Figure 2: Exact solution with 15 points

2.3 Greedy Approach

Here, we are going to implement the 'working' solution that is both optimized and can be executed in reasonable time. The approach is to find nearest neighbour in each step and homing when needed. To minimize runtime, here I created a set for visited points. The total time complexity for this algorithm is $O(N^2)$

```python
def path_finder_NN(self):
  '''
  implementation of nearest neighbour search algorithm
  '''
  self.compute_distance_matrix()
  print('Distance map created')

  visited = set()
  curr = 0 # current node index
  d = 0
  self.order.append(0)

  while visited != set(range(self.N + 1)):
    # search for nearset neighbour
    min_dist = 2.0 # initialize, must larger distance between two points
    visited.add(curr)
    for i in range(self.N+1):
      if i not in visited:
        if self.dist[curr][i] < min_dist:
          min_idx = i
          min_dist = self.dist[curr][i]

    if d + min_dist + self.dist[min_idx][0] < self.max_charge:
      # go to the next node
      curr = min_idx
      self.order.append(min_idx)
      d += min_dist
    else:
      # need charging, next state is charged and continue from origin
      d = 0.0
      self.order.append(0)
      curr = 0

  self.order.append(0)
```

Listing 4: Nearest Neighbour Search


(a) 300 points

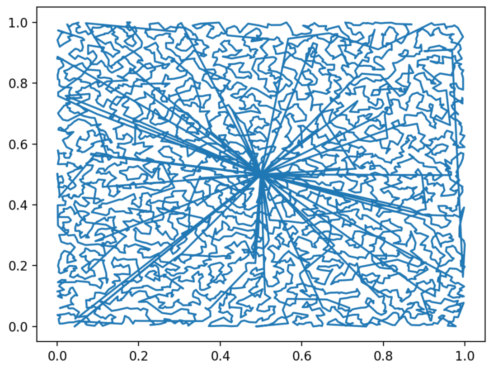

(b) 5000 points

Figure 3: Visualization of robot path using greedy method

3 Results & Discussions

The traveling robot challenge is an NP-hard optimization problem, in the scope of this report, we explored three methods: naive approach, exact solution and greedy approach, while each of these methods has their merits and drawbacks. We can find trade-offs between computation speed and path length generated. Therefore, I concluded a table for performance comparison. Notice that the 'Real Run Time' is the result that run on my local machine, may vary based on different PCs.

Table of Results (5000 points)			
Method	Time Complexity	Real Run Time[s]	Total Path Length
Naive Approach	$O(N)$	0.04324	3795.3284
Exact solution	$O(N^2 2^N)$	N/A	N/A
Greedy Approach	$O(N^2)$	4.7301	78.2843

As shown in table 3, the naive approach is the fastest to compute since we mainly just need to copy the entire point list, however, it comes with the huge cost of total path length for the robot. We also explored the exact solution to similar classical NP-hard problem traveling salesman, we used dynamic programming to reduce time complexity from $O(N!)$ to $O(N^2 2^N)$, however, it's still not efficient enough for computing 5000 points as our problem required. Therefore we are lack of valid results. At last, we tackled the problem using graph based greedy approach where we repeatedly find nearest neighbour as the next traveling target. This approach is easy to implement and computed in polynomial time, although it doesn't guarantee to be the optimal solution.

4 Future Work

By inspecting results of greedy approach in figure 3, one can find that some radial lines from origin to points far away. They are formed by robots returning home for charging, however, such paths are wastes to the system.
We can further optimize the solution by introducing sense of direction to the robot. For example, when remaining traveling distance is below certain percentage, instead of finding and examining the nearest neighbour as the next target blindly, we can compute 'homing' direction and select points that are 'closer' to the charging station as candidates. This way the robot can continue visiting new targets while on its way back to the charging station.