Smart Vacuum Robot

Harsh Aggarwal, Pranjal Aggarwal

1 Introduction

With advent of modern technology, once a thing of future, is now becoming a common reality. However AI agents employed in these systems have to face various challanges, as the cognitive abilities of AI is far less than that of humans. One such task is that of robot cleaning. The robot systems employed in this task have to sense the enviornment and take decisions without harming the humans or other objects. At the same time these systems should be efficient enough to complete their task in a limited time and also in limited amount of energy. In this report we study this problem of efficiency, and discuss various literature, which have tried to solve a problem similar to this. We use them and build upon a new heuristic based approach.

2 Problem Statement

Suppose we have an indoor building with n rooms, which need to be cleaned. Each room may have different sizes and floorplans, and different amount of dust in them. The task of the robot is to clean all the rooms in the building in a limited amount of time. This limited time can be due to both limited battery or storage capacity of robot or urgency to complete the task(such as when public centrs are closed for only small duration in the whole day).

At the same the robot has to maximise the amount of cumulative amount of dust cleaned in all rooms, while providing a minimum in each of the room. The problem can be modelled as a two stage problem:

- In the first stage, we find the best path for the robot to reach all the rooms. This is a Travelling Salesman Problem we propose Steiner TSP Algorithm for this.
- In the second stage, we allot a certain time to each room based on their size, and the robot has to find a path which maximizes the dust cleaned, while at the same time is within the alloted budget of time.

Both of these stages can be modelled as a path finding in graph problem, where in first stage the pathway connecting rooms is equivalent to edges and the rooms as vertices. For the second stage we divide a room into a grid of n*n cells, each either being occupied by an object, or having certain amount of dust. This grid can be considered as graph, with the unoccupied cells as the vertices and the amount of dust in them as their vertex value. Since it will take slightly more time for robot to clean cells with larger dust amount, the edge lengths will be dependent on the values of both the connected vertices:

$$C_{i,j} = k + f(V_i, V_j) \tag{1}$$

Where k is a constant value dependent on the speed of robot, and f is an arbitary function defined on the ordered values of two vertices.

3 Related Work

The problem at hand in stage 2 can be modelled as a variant of TSP known as Profit-TSP. In this problem, the Salesman has to maximise its profits however in a fixed finite amount of time. This has been shown to an NP-Hard Problem [1],[6]. In literature various algorithms have been proposed, ranging from genetic algorithms to simple heuristic based algorithms. [1],[2]. In this report we will discuss in detail these works and also propose modifications to suit our needs in solving the aforementioned problem statement.

4 Steiner TSP solution

The problem of starting from a point on the graph and returning to the same point after visiting a pre-determined fixed set of points is called steiner TSP. A steiner TSP is just the extension of the TSP problem and can be converted to the regular TSP problem very easily.

4.1 Approach

Consider a fully connected graph G with vertex set V and edge set E such that $E \subseteq V \times V$. Given a set of vertices $U \subseteq V$ such that the starting vertex $v \in U$ and all vertices in U must be visited exactly once with the final vertex being v.

We can construct a subgraph $H \subseteq G$ such that $V_H = U$ and $E_H = V_H \times V_H$ i.e. a complete subgraph comprising the vertex set U. We can define the edge length for an edge $e = (x, y) \in E_H,\ x, y \in V_H$ as the shortest path length from x to y in G. i.e. $e(x, y) = d(x, y)$ where $e(x, y)$ is the length of edge from x to y.

Executing TSP on the above subgraph H will yield the corresponding solution to the steiner TSP problem on G. The TSP is solved using a Dynamic Programming approach. Let $dp(src, visited)$ indicate the shortest path length starting from src to v, given that the set of already visited vertices is represented by the bit mask [1] $visited$.

To solve the DP we follow the steps below.

1. Mark src as visited by setting its bit in the $visited$ argument.
2. For all the unvisited neighbors of src calculate the sum of the edge length and the answer returned by the recursive call to $dp(neighbor, visited)$ which is the length of the shortest path from the neighbor to v with the updated $visited$ bit mask.
3. Calculate the minimum of all path lengths and store it as the optimal solution for this problem.

[1] If the number of vertices involved is N the $visited$ argument is an integer in the range $[1, 2^N]$. The i^{th} bit of the number $visited$ indicates whether the i^{th} vertex is visited or not. To mark a vertex as visited we simply set the i^{th} bit of the number. It is a very efficient approach for encoding information in small constrainted problems and is popularly known as *bit masking*.

Memoization of the results in an array of size $N \times 2^N$ ensures each subproblem is solved exactly once.

Algorithm 1: dp(src, visited)

```
visited ← visit(visited, src);
optimalLength ← INT_MAX;
forall neighbor of src do
    if !isVisited(visited, neighbor) then
        pathLength ← edgeLength(src, neighbor) + dp(neighbor, visited);
        optimalLength ← min(optimalLength, pathLength);
    end
end
return optimalLength;
```

4.2 Proof of correctness

The recursive *dp* procedure can be proved easily using the principle of mathematical induction by inducting on N (the number of vertices).

Base: The base case is for $n = 0$. The shortest path length is zero which matches with the algorithm output.

Induction Hypothesis: For a given $n = n_0$, the procedure $dp(src, visited)$ returns the shortest path length from *src* to v after visiting all the vertices exactly once and the initial visited vertices represented by the bitmask *visited*.

Induction step: For a given $n = n_0 + 1$ the algorithm marks the *src* vertex as visited and then recursively calls the *dp* procedure with the updated bitmask. Observe that setting a vertex as visited implies that no solution to the subproblems are allowes to visit the vertex. But this is equivalent to just removing the vertex from the graph for the subproblems. This leads to a graph with $n-1$ nodes for which the recursive calls are made. Now for $n - 1 = n_0$ vertices the Induction Hypothesis can be invoked to state that the shortest distance from a neighbor of *src* to v is specified by the recursive call $dp(neighbor, updated\_visited)$ where *updated_visited* is the updated bitmask after visiting *src*. If the cost of the edge is added then the total path length is determined and taking a minimum of all path lengths from the unvisited neighbors yields the shortes path from *src* to v. Thus for $n = n_0 + 1$ the Induction Hypothesis holds true.

Hence by **principle of mathematical induction** we have shown that the algorithm $dp(src, visited)$ returns the shortest path length from *src* to v with the current visited state represented by the bitmask *visited*.

4.3 Analysis

The total number of subproblems is $(N * 2^N)$. This is also the size of the memo that will be used during memoization for the DP solution. Hence the **space complexity** of solution is $\Theta(N*2^N)$. Now each subproblem requires a linear amount of time since it involves a search through all the neighbors of the source vertex. Since there are at most $\Theta(N * 2^N)$ subproblems the **time complexity** of the algorithm is $\Theta(N^2 * 2^N)$

4.4 Implementation

We have implemented the above solution in `C++` the results of which are in the next section. We have used the following data structures to implement the algorithm

- **Adjacency List:** We have converted the matrix into a graph with each cell being a vertex of the graph and edges connected to vertices represented by the adjacent cells of the matrix. The graph is stored in a adjacency list and can be constructed in $\Theta(n)$ time where n is the total number of cells in the graph.
- **Memo:** We have maintained a 2-D array that memoizes all the solutions of the DP. This prevents recomputation of the solution of a subproblem.
- **Queue:** To calculate the shotest path distances from one node to the other we have used BFS (since all edges are equally weighted). The resulting path is stored in a queue indicating the direction sequence of the robot.

4.5 Simulation

The robot starts from a certain point and is given a set of 8 points (or doors) to visit in any order. The objective of the simulation is to show that the robot takes the path that minimizes the total path length and hence is the most energy efficient path. The number of doors reached so far are displayed on the bottom left of the simulation and the progress of simulation is indicated by the progress bar. The path length covered so far is indicated to the right of the progress bar.

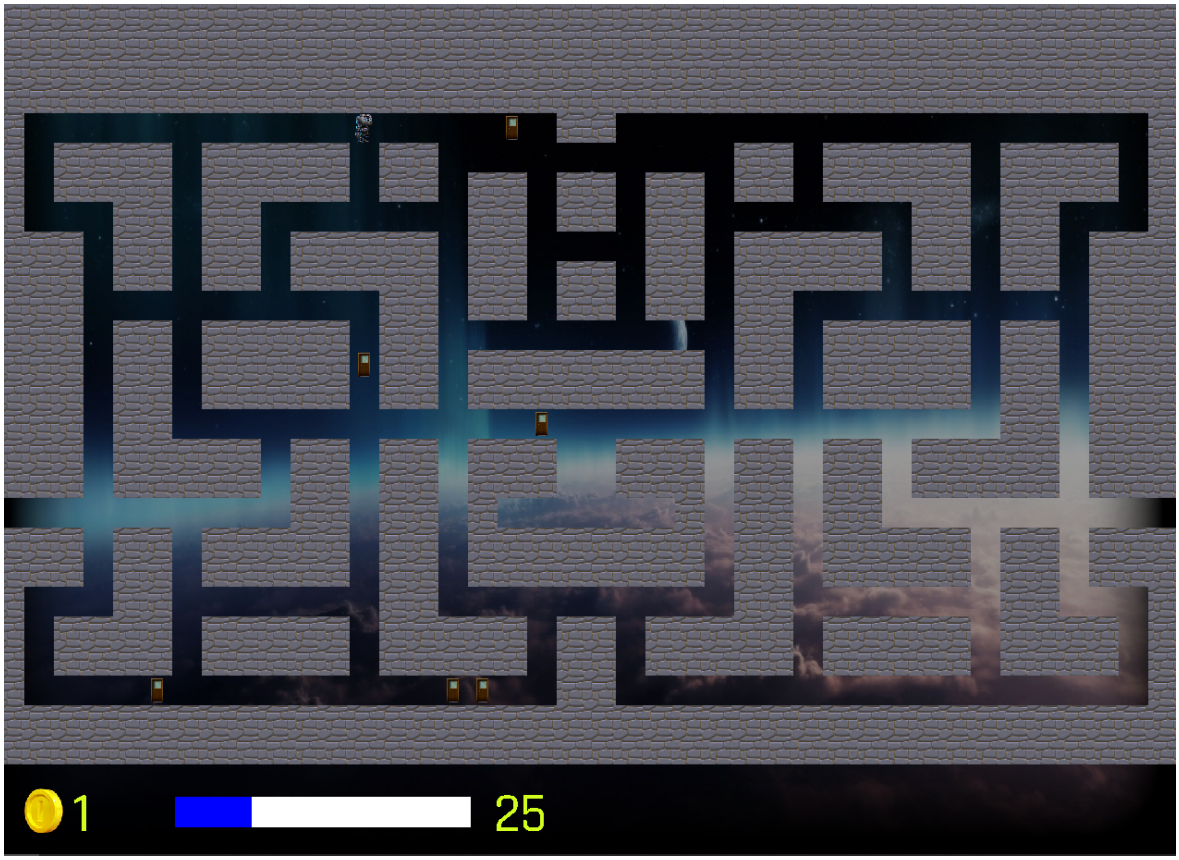

Figure 1: The robot is finding its path through the maze to reach the doors. It does so in such a manner that the total path length is minimized hence minimizing the energy consumption.

5 Solution: Time constrainted maximal coverage

In this section we will discuss two possible heuristic solutions to our problem. The initial two stages are the same for both the approaches but they differ in the last step.

We divide the solution to the problem in multiple stages each performing a different operation. All these stages are iterated over for multiple rounds, till convergence criteria is not met. We use 4 different hyperparameters whose values are determined emphericallly under some constraints.

5.1 Stage I: Insertion Stage

At the start of stage we assume that we have a path $\{L(i) : 0 < i \leq k\}$, where k is the path length, and L(i) denotes the i^{th} vertex in the path. Since we have to start and end at the same position, $L(1), L(2) = 1$. We define two sets Ψ and Ω , which contains the vertices in current path and and vertices not in current path respectively. Now we first decide, a vertex to insert, lets call it j. Now we need to find a point in the path where we can insert the given vertex, and which gives maximum increase in profit while minimising the cost. Then point of insertion can be found by solving the following Optimisation:

$$\Theta_j min_{i=1,\ldots,k}(C_{V(i),j} + C_{j,V(i+1)} - C_{V(i),V(i-1)}) \tag{2}$$

Where Θ_j is the calculated cost of given vertex for insertion. Also note that in 2 we only need to iterate over only neigbours of vertex j, which in our case can be a maximum of 8, and therefore step 1 can be done in O(1) time.

Now we iterate over all the sutiable vertices for insertions, i.e and calculate the corresponding Θ_j. Now the best vertex is the one which give the maximum profit while with minimum cost i.e we try to take that vertex which has the highest profit to cost ratio. Therefore we have to solve the following equation:

$$max_{j\in\Omega}\frac{P(j)}{C(j)}) \tag{3}$$

However this would give high weightage to high density vertices therefore we add an additional hyperparameter λ_1 in 3, and modify it as:

$$max_{j\in\Omega}\frac{P(j)^{\lambda_1}}{C(j)} \tag{4}$$

We hypothesize that $\lambda_1 < 1$. This is due the fact that a minimum time is always required for robot to cross a cell, and thus by adding this additional hyperparameter, we ensure that portions of small dust are not leftover.

We continue this process of insertion till the cost doesnt exceed $\lambda_{2,1} * \beta$, where $\lambda_{2,k}$ is another hyperparameter whose value varies in different rounds, and always increasing in subsequent rounds. If addition of any vertex increases the cost beyound alloted capacity, we omit it and move on to the next stage.

As we see that runtime complexity of this step is $O(n^2)$ in this case, wehere n is the final path length achieved after performin this stage. Note that its not $O(n^3)$ due to limited number of neighbours a vertex can have in the maze like structure we have used.

5.2 Stage 2: k-Opt

In this stage we apply the famous k-Opt Startegy [5] [4] increase the total profit while being in the budget. We start with applying 2-opt Optimisation, and if the final profit is more than $\{\lambda_3, \lambda_3 > 1\}$ times the previous score, we move on to higher order k-opts. However if increase in budget is observed without satisfying the previous equation, we go back to stage-1 since further insertions may have scope

for improvement. However if no increase in profit is improved, we move onto the next stage. The worst case complexity of this strategy on a complete graph is $O(n^2)$, however as in previous stage, since we have a maximum of 8 connexting edges from a vertex, the time complexity is just $O(n)$ for a single pass of this stage.

5.3 Stage 3: Final stage

There are two approaches discussed below for the last stage. The first approach deals with deleting vertices in the current path, whereas the second approach uses a centre of gravity approach [3] to modify the existing route.

5.3.1 Deletion

In this stage we try and delete vertices, that may not be useful for inclusion in path. Let S_{final} be the new profit after deleting a vertex v from path P. Then we define a metric:

$$\xi(j) = \frac{S_{initial} - S_{final}}{P(j)} \tag{5}$$

where j is the vertex removed from the path

Then we find the vertex $j \in P$ such that value of ξ is minimised. What this means is that deletion of j gives the lowest per unit price loss. After this step we check for the convergence criteria which terminates when the new overall profit P_{ov} is atleast λ_4 times the overall profit in last round $P_{ov,last}$. If the convergence criteria is not met we move back to stage 1 for further insertions and repeat the steps. Again the time complexity of this stage is $O(n)$ since we can delete only n points in the path, and deleteion of each point takes only $O(1)$ time

5.3.2 Centre of Gravity

Suppose now that node i has coordinates $(x(i), y(i))$. In this step, we calculate the center of gravity of L as $g = (\bar{x}, \bar{y})$, where

$$\bar{x} = \frac{\sum_{i \in L} P(i)x(i)}{\sum_{i \in L} P(i)} \tag{6}$$

$$\bar{y} = \frac{\sum_{i \in L} P(i)y(i)}{\sum_{i \in L} P(i)} \tag{7}$$

Let $a(i) = t(i, g)$ for $i = 1, 2, \ldots, n$ i.e. the time taken to reach from i to the centre of gravity g. Next a route including nodes 1 and n is formed as follows:

1. Calculate the ratio $P(i)/a(i)$ $\forall i$
2. Add nodes to the route in decreasing order of this ratio using cheapest insertion, until no additional nodes can be added without exceeding the time constraint T_{max}.
3. Use the route improvement step to make adjustments to the resulting route.

We now have a route L_1. This route's center of gravity gives rise to a repetition of (1)-(3). The resulting route is denoted by L_2. This process is repeated until a cycle develops, that is, route L_p and L_q are identical for some $q > p$. Finally, we select the route that has the highest score among the routes $L, L_1, L_2, \ldots, L_q$.

This stage calculates the centre of gravity and the updated ratios at each iteration. This requires $\Theta(n)$ time in each iteration. Since there are $\Theta(n)$ iterations, the overall time complexity of this stage is $\Theta(n^2)$. However we also repeat this stage q times which would account for a higher complexity of $\Theta(qn^2)$. Typically, q is kept to be a static constant and is taken such that $q \leq 10$. This ensures that the asymptotic analysis would yield a runninng time given by $\Theta(n^2)$.

5.4 Implementation

We have not implemented the heuristic algorithm and plan to do it in the future. However, the relevant data structures that are to be used for the implementation are outlined below.

- **Path List:** A doubly linked list of the path nodes in the current path. This can be used to insert a vertex between two vertices in constant time.
- **Hash table:** An unordered set of the vertices already included in the path to allow for constant order existence determination of a node in the current path.
- **Priority Queue:** Since insertion occurs only for the neighbors the priorities of insertion of each neighbor can be stored in a priority queue. At the time of insertion we wil extract the maximum element from the priority queue and add the priorities of the extracted node's neighbors to the queue. This will ensure logarithmic order insertion to the path.

5.5 Conclusion

As we have shown the overall complexity of each stage is not more than $O(n^2)$ and therefore each round of algorithm is fast enough. Also results in literature show that similar algorithms are able to achieve fast enough convergence, that they can be used in real time embedded systems, with minimum energy costs. [1] Simulation data suggests that the robot was able to find a very good path in very less time, thus showing the effectiveness of this algorithm.

5.6 Final Algorithm

Algorithm 2: Optimisation Algorithm

while *not converged* **do**
  1. Stage 1: Given a graph $G = (V, E)$ perform insertions based on 4 till the cost doesnt exceed the specified value;
  2. Stage 2: Apply k-opt strategy. **if** *improvements are good enough* **then**
    Go to Stage 1 if ;
  else
    Move to Stage 3 ;
  end
  3. Stage 3: Perform Deletions of vertex in path in accordance with 5;
end

6 Future Directions

The algorithms discussed in this report were mostly tested on random mazes. While the results on them look promising, in future we would like to simulate the problem in real floorplans, and data. Also this problem can be further extended by considering some more realistic settings, such as different speeds of robot in different terrain, getting more information about the dust, such as their weight, density etc, to create an more accurate timing model.

Most importantly we would like to implement the heuristic algorithm and test it using our simulation. We aim to extensively test the heuristics on several maze formats and carefully placed door locations. Moreover, it would be interesting to extend the problem to an overall time constraint where the robot has a total time constraint rather than a time constraint for the room. This would mean we would have to further analyze the modifications to be done in the steiner TSP solution.

7 Conclusion

Overall we see that with the right heuristics the given NP-Hard problem can be solved with good accuracy, in a real-time. While we focussed on a single task, similar algorithms can be used in domains not discussed in this report. [3],[6] Also with advent of modern technologies with capabilities of replacing human beings at non-trivial taks, such algorithms will be the need of the hour as evident from our application to cleaning robots. As robotics grows in the technological domain, more path routing and graph based algorithms would need to be implemented, the solutions of most of which are NP Hard. We have demonstrated that the appropriate heuristics can get us very close to the optimal solution as given by deterministic algorithms with exponential time complexities.

References

[1] An efficient four-phase heuristic for the generalized orienteering problem. *Computers & Operations Research*, 18(2), 1991.

[2] The time constrained maximal covering salesman problem. *Applied Mathematical Modelling*, 38(15), 2014.

[3] Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics (NRL)*, 34(3), 1987.

[4] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2), 1973.

[5] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10), 1965.

[6] T. Tsiligirides. Heuristic methods applied to orienteering. *Journal of the Operational Research Society*, 35(9), 1984.