Reading By Visualisation Derivation

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 can be due to both limited battery capacity of robot or urgency to complete the tasl]

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 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].

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.

Additionally we can memoize the results of the DP in an array of size $N \times 2^N$ so that 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;
```

[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*.

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 $\Theta(N^2 * 2^N)$

5 Solution

In this section we will discuss two possible heuristic solutions to our problem.

5.1 Centre of Gravity Based Algorithm

5.2 Multi-Stage Algorithm

While the previous algorithm requires an Euclidean cost function, in this algorithm we will model a genric enough cost function and solve the problem.

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:-

5.2.1 Stage I: Insertion Stage

At the start of stage we assume that we have a path $\{P(i) : 0 < i \leq k\}$, where k is the path length, and P(i) denotes the i^{th} vertex in the path. Since we have to start and end at the same position,

$P(1), P(2) = 1$. 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:

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

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. 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.

5.2.2 Final Algorithm

Algorithm 2: Optimisation Algorithm

while *not converged* **do**
1. Stage 1 ;
2. Stage 2 ;
3. Stage 3 ;

end

6 Computation and Results

7 Conclusion

References

[1] PlaceHolder. Placeholder, PlaceHolder.