Fall 2017

Artificial Intelligence
DD2380

Street View Planning

Aron Granberg, arong@kth.se
Jón Freysteinn Jónsson, jfjon@kth.se
Mårten Wiman, mwiman@kth.se
Vidar Palmér, vidarp@kth.se
October 13, 2017

Abstract

Planning paths for a small number of agents to cover all edges of a directed graph by going from node to node is a problem that has a high applicability in some real world scenarios. In our case we take inspiration from how Google Street View cars need to plan their paths in order to be able to photograph as many streets as possible in a city within a given time limit. We approach this using several different approaches, both approximate ones and optimal but slow algorithms. Finally we demonstrate our algorithms on several realistic test cases consisting of maps of cities such as Paris and New York.

Contents

1 Introduction

Several companies are currently making 3D renderings of the world so that they can later use it in various products. A common way to gather this data is to have cars drive around cities and highly populated areas and constantly photograph the surrounding world while driving. To speed up the process a good option is to use a fleet of many cars to observe different parts of the area simultaneously. This is a prime example of a multi agent planning problem.

More specifically the task handled in this project is to let multiple agents traverse as many edges (streets) of a city graph as possible in the least amount of time. This problem was given in the programming competition Google Hash Code 2014 [3] where the inspiration for this project came form. The goal of the project is to not only give 2 approximate algorithms that solve the problem but to also solve the problem optimally for simpler test cases, to construct some additional test cases and to make a visualizer to see the agents path through the graph.

2 Background

2.1 Eulerian cycle

An Eulerian cycle in a graph is a path that uses each edge exactly once and ends in the same vertex that it started in. The necessary condition for a directed graph to contain a Eulerian cycle is that each vertex in the graph has the same in-degree as out-degree and that the graph is strongly connected. A strongly connected graph is a graph where there is a path from every node to every other node. [1]

2.2 The Chinese postman problem

The Chinese postman problem is a problem where one agent seeks to traverse each edge in a connected, undirected graph at least once. If all nodes in the graph have an even degree there will exist an Eulerian cycle in the graph and this will be the optimal solution to the Chinese postman problem. [1]

If some nodes have odd degree some edges will need to be traversed several times. In this case traversing an edge an additional time can be viewed as duplicating that edge. To get a solution edges should be duplicated in the graph until all nodes have even degree. The best solution is then the solution when either the least amount of edges are duplicated or the lowest sum of the value of the duplicated edges is achieved depending on whether the edges has any values.

2.3 Mixed postman problem

A different version of the Chinese postman problem is the mixed postman problem. In this case the graph consists of both directed and undirected edges. It is known that the postman problem is solvable in polynomial time if the graph consists of either only directed edges or only undirected edges. However if both directed and undirected edges exist the problem is NP-complete and require heuristics to solve somewhat effectively. [5]

Raghavachari et. al. [6] found an approximate algorithm for the mixed postman problem which gives at most an answer that is 3/2 times as big as the optimal answer. This is accomplished by a combination of algorithms that modify the graph until an Eulerian cycle can be found.

They do this by using a modified version of an algorithm made by Fredrickson [2]. First they use an algorithm to modify the graph by matching the in and out degree of vertices by duplicating and directing edges. The result is then used in two ways. The first way to use it in an algorithm that duplicates undirected edges until every vertex has even degree. This gives an answer where an Eulerian cycle can be found and the number of extra distance traversed is the sum of the size of all duplicated edges.
In the other way the result is used in an algorithm that also duplicates edges to get a graph with even degree however in this case all directed edges are prioritized. Then the algorithm that was used to match the in and out degree of each vertex is used again followed by an algorithm that restores the even degree of all vertices but don't change the in or out degree of any of them. The result contains an Eulerian cycle where the extra cost is the sum of the size of all duplicated edges.
These two answers are then compared and the best of them is used.

2.4 Multiple agents

Oh et al. [4] used known heuristics for the knapsack problem to optimize the Chinese postman problem with multiple agents. The paper handled some related issues like turning radius which was accounted for with Dubins path planning.

3 Method

The first problem to be tackled was the visualizer which would make debugging and analyzing the other solutions simpler. Simultaneously several test-cases were constructed so that the algorithms could be examined. We constructed a small test case that we could solve optimally using a brute force solver. All test-cases are based on real cities or local areas. For example the smallest test case is based on a map of the KTH campus. A brute-force algorithm was implemented that could solve the problem completely for small graphs within reasonable time-frames. Solving larger problem instances with the brute force algorithm is however not reasonable as the algorithm has an exponential complexity. The brute force algorithm was optimized to use A* which did improve the performance significantly, however larger graphs are still to slow to solve.

To solve the problem approximately for larger graphs three approaches were tried.

The first approach determined the best path of every car by iterating through them and for every car determine its entire path before continuing with the next. In each step the car looks at the outgoing edges from the node it is currently positioned at, then it picks an arbitrary one of them that is not yet covered by any car and follows it to the next node. However if all edges from that node have already been covered then the algorithm will use Djikstra's algorithm to find the closest node with an edge that is not yet covered.

The second approach is based on the polynomial-time algorithm that solves the Chinese postman problem optimally when there is only a single actor and all edges are directed. The idea is to assign directions to the undirected edges in such a way that each node has approximately the same out-degree as in-degree, and then we can find the shortest cycle (with some edges repeated) passing through all edges in the new graph with only directed edges. The first car follows this cycle until it runs out of time, and then the second car repeats the process, finding a cycle of its own which it follows until it runs out of time, and so on.

3.1 Generating Test Data

The Google Hash Code data we retrieved from the 2014 competition included two test cases. One example test case that only included three nodes and only useful for understanding the data structure. The larger test case is a graph of Paris which includes around 11 thousand nodes and 18 thousand edges. The test cases are given as `.txt` files. The first line of the file includes five variables `N, M, T, C, S`. `N` denotes the number of nodes(junctions) in the city graph. `M` denotes the edges(streets) in the city graph. `T` denotes the time the cars have to finish their pass of the city. `C` represents the number of cars and `S` the starting junction of the cars. Then following `N` lines that include details for each node, then `M` lines for the edges. Each line for the nodes includes the latitude and longitude of the node. Each edge contains information `A, B, D, C, L`. `A` and `B` are the two nodes the edge is connecting together, `D` is whether or not the edge is directed, `C` is the time it takes the car to traverse and `L` is the length of the edge.

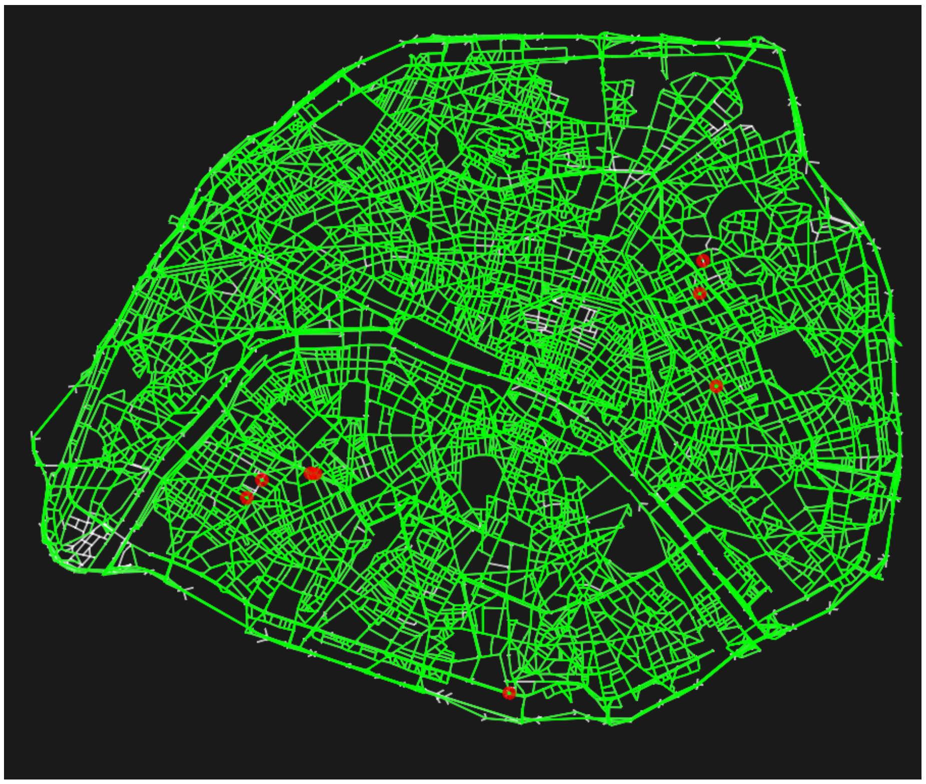

Figure 1: One solution to the Paris test case. The streets that have been covered are coloured in green and the edges that haven't been covered are coloured in gray. The final location of the cars are shown in red.

```
3 2 3000 2 0
48.8582 2.2945
50.0 3.09
51.424242 3.02
0 1 1 30 250
1 2 2 45 200
```

We decided to try and create some more test cases to test our solution on more than one city and provide some variety to the visualizations. We looked into a couple of different solutions that could provide us with the data required to generate the tests but ended up selecting a package based on Open Street Map called OSMnx. The python library allowed us to easily retrieve the needed street information to generate the tests. A simple API call fetches a graph of the desired City.

```
import osmnx
```

```
osmnx.graph_from_place('Stockholm', network_type='drive')
```

The function requests information from the web API and generates a graph object that contains way more functionality that we need for our project. We can retrieve a list of all the edges in the graph through the `edges` property on the graph object. Each edge contains information about the start node, the end node and basic information about the type of edge. Edges can be bridges, roads, highways, etc, and each contains information about direction and length. The start and end nodes can be used to retrieve the latitude and longitude from the graph object. With all this information we can generate our own test cases for any city we want.

3.2 Optimal solutions

3.2.1 A* search

Our first approach at solving the problem optimally was based on A^* search. In our solution, we process one car at a time and let a state consist of the following parameters:

- The edges that have been traversed at least once by some car.
- The location of the current car.
- The amount of remaining time for the current car.

The reason that we process one car at a time instead of choosing the actions of the cars in chronological order is that the latter option would require us to include the locations and times of all cars in the state, which would make the state space much larger. When we only process one car at a time, we don't need to include the locations and times of the earlier cars because they cannot contribute anymore.

In A^*, it is necessary to be able to compute an upper bound on the achievable score from each state. Our upper bound is computed as follows:

- Let T be the total amount of time remaining for all cars.
- For every edge e_i that has not yet been covered, let l_i be its length and let d_i be the time it takes to traverse it.
- Solve the knapsack problem where we want to find a set $E = \{e_{i_1}, e_{i_2}, \dots, e_{i_k}\}$ such that $\sum_{j=1}^{k} l_{i_j}$ is maximized subject to $\sum_{j=1}^{k} d_{i_j} \leq T$.
- The sum of the best possible value of $\sum_{j=1}^{k} l_{i_j}$ and the total length of the already covered edges is an upper bound for the achievable score from the current state.

3.2.2 PDDL-based solver

Our next attempt was based on translating the problem to PDDL. For simplicity, we restricted our attention to problem instances with only one car, and where we want to determine how fast we can cover all edges in the graph (instead of trying to cover as large a distance as possible in a given time). However, after translating the problem PDDL we found that the PDDL solver (Fast Downward) was unable to solve the easiest test case (Campus) in a reasonable amount of time, so we abandoned this approach.

4 Approximate solutions

For larger graphs, it is not feasible to solve the problem optimally. Therefore, we have implemented several approximate algorithms that run in polynomial time but may yield worse solutions.

4.1 Greedy solution

Our first approach is to choose the paths of the cars one car at a time, and the path of each car is chosen greedily so that as long as a car is able to traverse an edge that has not already been covered, it chooses one such edge at random. If a car is at a node without any uncovered outgoing edges, it finds the closest node that does have an uncovered outgoing edge and goes to that node instead.

Algorithm 1 Greedy solution

```
1:  for each car $c$ do
2:      $p \leftarrow$ starting location
3:      while $c$ hasn't run out of time do
4:          $E \leftarrow$ outgoing edges from $p$
5:          $E_u \leftarrow \{e \in E : e$ has not already been covered$\}$
6:          if $E_u = \emptyset$ then
7:              $Q \leftarrow \{q : q$ has an uncovered outgoing edge$\}$
8:              $q \leftarrow$ node in $Q$ such that distance from $p$ to $q$ is minimized
9:              Let $c$ traverse the shortest path from $p$ to $q$
10:             $p \leftarrow q$
11:         else
12:             Pick $(p, q) \in E_u$ uniformly at random
13:             Let $c$ traverse the edge $(p, q)$
14:             $p \leftarrow q$
```

A virtue of this solution is that it can be made very efficient. The bottleneck of the algorithm is line 8, where the shortest path from p to any node with an uncovered outgoing edge is found. We find the shortest path using Dijkstra's algorithm, and we optimize the algorithm by not constructing the set Q of nodes with uncovered outgoing edges explicitly, but instead checking on the fly for each node visited by Dijkstra's algorithm whether it has an uncovered outgoing edges. We also terminate Dijkstra's algorithm as soon as we find a node with an uncovered outgoing edge, which means that when there are many such nodes only a fraction of the graph will need to be searched.

Because this algorithm is fast and non-deterministic, it is a good idea to run it many times because each time a different solution is found. Then, we can simply return the best solution found in all the attempts.

4.2 Eulerian cycles

As mentioned earlier, the problem becomes solvable in polynomial time when there is only one agent, all edges are directed and we want to find the minimum time required to traverse all edges and return to the starting point instead of trying to cover as large a distance as possible in a given time.

The solution to the simplified problem is based on the observation that a connected graph contains a Eulerian cycle if and only if all nodes have the same in-degree as out-degree. We want to select some paths in the graph and add the edges in each path to the graph, and we want to do this in such a way that in the new graph each node has the same in-degree as out-degree. Therefore, if a node has in-degree a and out-degree b then we need to select $a - b$ paths that start at that node if $a > b$, and we need to select $b - a$ paths that end at that node if $a < b$. We can then choose the paths using weighted matching: For any node u which has a larger in-degree a than out-degree b, we add $a - b$ nodes to a set L, and for any node v which has a larger out-degree d than in-degree c we add $d - c$ nodes to a set R. Then we add edges from all nodes corresponding to u to all nodes corresponding to v, and let the cost of each such edge be equal to the distance from u to v in the original graph. Using the Hungarian algorithm, we can solve the weight matching problem in cubic time, and the solution determines which paths should be chosen to minimize the total time required to traverse the extra edges. After having added the extra edges, we can easily find a Eulerian cycle in the new graph using a depth-first search.

Because the problem solved above is a simplified version of the original problem, the approach above may not be applicable to the original problem. Let us therefore discuss the assumptions we made, one by one.

4.2.1 Only one agent

In the simplified problem we assumed that there were only one car but in the actual problem there could be several cars. We solve this by processing one car at a time: First we find a Eulerian cycle for the first car, which it follows until it runs out of time, then we mark all edges traversed by the first car as done and find a new cycle for the second car which attempts to cover all remaining edges in as short time as possible, and so on. This approach is generally not optimal, but it nevertheless gives reasonable results when the diameter of a graph is much smaller than the time limit of each car, because the worst possible scenario is that the second car will have to travel to the location where the first car stopped and continue from there.

4.2.2 Returning to the starting point

We also assumed that the cars had to return to the starting point, which was not a requirement in the original problem. The problem where the car is not required to return to the starting point (but there is only one car and all edges are directed) is in fact also solvable in polynomial time, and it can be solved by iterating over all nodes and computing the shortest possible time necessary if the car ends at that node. However, we don't think this problem is so big that it's worth the increase in computational cost, because the required time increases by at most the diameter of the graph (and generally much less than that).

4.2.3 Optimizing minimal time instead of maximal length

What we really want is to maximize the total length of the streets covered by the cars, but in the simplified problem we instead tried to minimize the time to cover all streets. In case we don't actually have time to cover all edges, we simply truncate the paths of the cars, which may or may not result in a good solution. The assumption that we get good results by truncating a solution in this way clearly holds for graphs where we're actually able to cover all streets in the given time, and it gives good results in the cases where we're very close to covering all streets. However, it doesn't yield very good results when the speed limits on different streets differ a lot and we're unable to cover all streets, because this solution will not prioritize the streets with high speed limits since it assumes that it will have to traverse all streets sooner or later anyway.

4.2.4 Directing the undirected edges

In the simplified problem, all edges were required to be directed. Therefore, we need to assign directions to the undirected edges somehow, but it's not obvious how to do this in a good way. How we do this often affects the quality of the solution very much, but since the mixed postman problem is NP-complete we cannot expect to do assign the directions optimally in reasonable time. Instead, we do the following:

- We greedily assign directions to the undirected edges in such a way that the in-degrees and out-degrees of each node are as close as possible.

- We greedily find paths between nodes so that the in-degree and out-degrees of each node in the new graph are equal. We do not use weighted matching at this step because the graphs are often too large to be able to apply an algorithm with cubic running time.
- We perform local optimizations to improve the greedy solution. We iterate through each node u and find all nodes and edges in a vicinity of u, and then we try all possible assignments of the directions of those edges and compute the minimal total time required for the paths between those nodes so that the nodes still have the same in-degree as out-degree. For the computation of the extra paths, we remove all the paths we already had between the nodes in a vicinity of u, but not paths for which one of the endpoints is not in a vicinity of u, and then the new paths are found using weighted matching which is computationally efficient as long as the chosen neighborhood of u is not too large.

4.3 Post-processing

In both the solutions discussed above, the cars are processed sequentially. This means that the path of the first car is determined, that car has no knowledge about the paths of the other cars, and therefore decisions made by the first car may no longer seem very good given the paths of the rest of the cars. For this reason, we implemented a post processing step which fixes some of the suboptimal decisions. More specifically, we loop over all cars and for each cars we consider all sub-paths consisting only of edges that are covered several times. Suppose that a car's path contains a sub-path π from u to v which only contains edges that are covered several times, then we use Dijkstra's algorithm to find the fastest path from u to v, and if there is a path from u to v that is faster than π then we replace π by the faster path. This gives the car more time that we can use to extend the path, so we try to extend the path using the same approach as in the greedy algorithm described in Section 4.1.

5 Results

We summarize the scores achieved using our different approaches in table 1. The optimal score for the campus test case is computed using the A^*-based algorithm described in Section 3.2.1.

Note that the results for some of the algorithms are unknown for some test cases. This is because they would have taken too much time to run.

It is worth noting that in the Hash Code competition, the winning team scored 1923588 points on the Paris test case.

Algorithm	Campus	Södermalm	Paris	New york
Greedy	2248	79671	1919293	10987200
Greedy + Post-processing	2248	**80102**	**1929817**	**11029265**
Eulerian	1776	71305	1846633	Unknown
Eulerian + Post-processing	1776	71658	1881851	Unknown
Optimal	**2261**	Unknown	Unknown	Unknown

Table 1: Maximum scores that the different algorithms managed to achieve on each of the test cases.

6 Discussion

The result show that our best scores were obtained by the greedy algorithm using post-processing. This did not follow our expectations which was that the Eulerian cycles approach would give us the best result. However, we made a few assumptions which must hold for the solution based on Eulerian cycles to work, and some of these assumptions may not actually be true for our test cases. In particular, the lower scores may be due to the fact that we had to assign directions to the undirected edges, which may have restricted the algorithm too much.

In the future it would be interesting to see if the Eulerian cycles approach could be improved. Another interesting way to extend this project would be to let all the agents act simultaneously rather than handling them sequentially. This might improve the cooperation of the cars when they are close to each other. However, it might make it harder to evaluate and analyze the decisions of each individual car.

The greedy algorithm with post processing achieved a higher score on the Paris map than the winning team of Google Hash Code 2014 which indicates that this approach was quite effective.

The fact that none of our heuristic approaches were able to obtain the optimal score even on the smallest test-case signifies that there is room for improvement.

References

[1] Jack Edmonds and Ellis L Johnson. "Matching, Euler tours and the Chinese postman". In: *Mathematical programming* 5.1 (1973), pp. 88–124.

[2] Greg N Frederickson. "Approximation algorithms for some postman problems". In: *Journal of the ACM (JACM)* 26.3 (1979), pp. 538–554.

[3] *Google Hash Code*. `https://hashcode.withgoogle.com/past_editions.html`. 2014.

[4] Hyondong Oh et al. “Coordinated road-network search route planning by a team of UAVs”. In: *International Journal of Systems Science* 45.5 (2014), pp. 825–840.

[5] Christos H Papadimitriou. “On the complexity of edge traversing”. In: *Journal of the ACM (JACM)* 23.3 (1976), pp. 544–554.

[6] Balaji Raghavachari and Jeyakesavan Veerasamy. “A 3/2-approximation algorithm for the mixed postman problem”. In: *SIAM Journal on Discrete Mathematics* 12.4 (1999), pp. 425–433.