Tourist Path Optimization Problem

Brian Wheatman
Massachusetts Institute of Technology
wheatman@mit.edu

Sertac Karaman
Massachusetts Institute of Technology
sertac@mit.edu

Daniela Rus
Massachusetts Institute of Technology
rus@csail.mit.edu

ABSTRACT

We studied a variant of the Prize Collecting Traveling Salesmen Problem, which is a path finding problem with both constraints and rewards. What is unique about this variant is that the reward for going to a node is dependent on the amount of time spent at each node and thus the optimization must be done not only over which nodes to visit, but also how long to stay at each node. Our goal is to be able to more efficiently calculate good solutions. Because the problem is NP-Hard, we are trying to develop an online algorithm so new solutions can be found quickly after a single expensive computation is done. By good solution, we mean one that can get us within a small percentage of the optimal.

We describe an online algorithm which is able to take a completed solution to one problem and find the solution to related problems, which are generated by either adding a node to or removing a node from the problem graph. We show that this online algorithm is faster at generating results than the offline algorithm by up to a factor of 20.

1 INTRODUCTION

We are studying a motion planning problem with both constraints and rewards. The basic idea of the problem is: given a set of nodes, each with a reward function depending on the amount of time spent at that node and edges, each of which has a cost in time to travel, how do you plan a trip that maximizes the total reward given a total time and returns to the start? An example graph can be seen in Figure 1 where we see the start labeled $Start$, the distance of each edge labeled d_i, and each different time-dependent reward function labeled $f_i(t_i)$. Also of interest is the dual problem, which is minimizing the time required to achieve a certain level of reward. This problem is a variant of the Traveling Salesmen Problem, specifically its Prize Collecting variant, with the added complexity that the reward at each node is not received for only going to the node but instead dependent on the amount of time spent at that node.

The goal of this research was to develop an online algorithm so that once a solution is found for any specific graph, one can easily find the solution to similar problems. These similar problems are defined as any problem that can be created by either adding or removing a node from the original graph. That is to say, the goal of this research is to develop a way to take a problem, solve it once using the current technique, then use information gathered during that first run to find the solution for any nearby problem statistically faster. We show that not only can we re-solve statistically faster, but moreover, some of these problems can be solved a factor of 10 faster.

There are several uses for this problem. The first is that of a tourist visiting a city. The set of destinations that the tourist would like to visit are the nodes and the enjoyment they receive for staying at that location is the reward. The travel time is the amount of time it takes to travel between two locations. We want to determine the ideal set of locations they should go to in the city, and in what order, to get the most out of their stay. The name comes from this use of the problem[3]. For this problem, we can see the benefit of having an online algorithm. We know that the first solution will take a long time, so this could be precomputed. However, we can imagine that on the day of his tour a tourist might change his mind and decide to either remove a node from his path or add one to the set of possibilities.

Another use case is that of a sensing robot. Imagine it has a number of different locations from which it can collect data. The amount of information it collects at each location depends on how long it stays or how much power it uses. The problem becomes how to maximize the amount of information the robot can collect with a set amount of fuel. Now, consider the scenario that before the robot goes out we can spend time computing the best path that we want it to take. However, while the drone is collecting data it discovers something new, either a new point to go look for or that something that initially looked promising no longer does. With the new

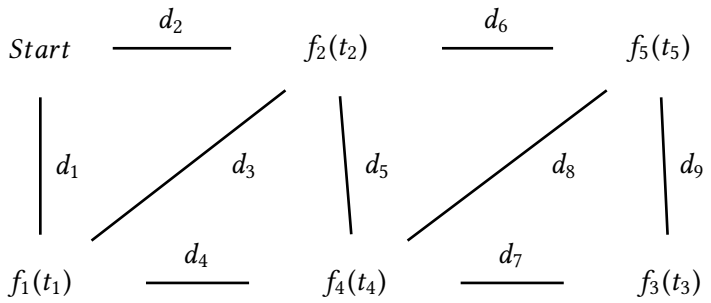

Figure 1: An example graph in which the Problem is given an amount of time T to maximize $\sum_{i=1}^{5} f_i(t_i)$ while keeping $\sum_{i=1}^{5} t_i + \sum_{i=1}^{9} x_i d_i \leq T$ [1], with only taking connecting paths and returning to the start.

algorithm, the drone can more easily recompute a new best path and still get provably good results.

2 RELATED WORKS

The problem as stated is NP-Hard, which we prove via reduction from the known NP-Complete problem, The Traveling Salesmen Problem, as follows: the reward at each node is 1 for 0 time and does not increase, and the minimum reward is equal to the number of nodes, which completes our reduction[1].

Since the problem as stated is intractable unless $P = NP$, current work has attempted to create an anytime algorithm that can solve small cases exactly and large cases approximately in reasonable amounts of time. An anytime algorithm is one that returns a feasible solution nearly immediately and iteratively makes it better over time, while remaining feasible. The algorithm will converge on the optimal solution if given enough time, but can be stopped early for an approximate solution.

This is done by setting up the problem as an Integer Linear Program (ILP) and solving it using the Branch and Bound technique. [2] With Branch and Bound, we immediately get a valid solution, which is either go to none of the nodes for the maximization problem, or all of the nodes for the minimization problem then make it better over time. We can choose to stop when the solution is optimal, after we have guaranteed to be close to the optimal either as a fraction of the optimal or within some ϵ of the optimal, or simply after some amount of time.

However, for reasonable sized problems with reasonable guarantees about how good the solution is, in the range of about 100 Points of Interest and within about 20% of optimal, the problem can take on the order of minutes to solve, even allowing for non-optimal results, which is unreasonable for any real-time application [3]. This leads to the main goal of this project, which is to create an online algorithm which will allow us to take the output of a run of the algorithm and more quickly find a solution to a nearby problem, which we have either added or removed a node from the graph. The idea is that we can amortize the expensive cost over many iterations or use it for the applications described above, where online solutions naturally apply.

The Traveling Salesmen Problem, of which this problem is a variant, is a highly studied problem and general approaches for an efficient solution fall into two categories. The first is to use heuristics: this can give fast and often good results in practice, but how close these results to optimal are cannot be proven. The second is to use a worse case exponential algorithm with provable bounds.[3][2]

For our problem, we considered two general approaches. The first was to follow the example given by Jaillet and

[1] $x_i = 1$ if you take the edge and 0 otherwise

[2] Branch and Bound first solves ILP by ignoring the integrality constraints. Then it chooses one of the variables, which has a non-integer optimal value, and creating two new subproblems, which are the same problem as the parent problem, but with the extra constraint that the chosen variable has to be either less than or equal to the floor, or greater than or equal to the ceiling of the current optimal value. This method of solving inherently gives rise to an anytime algorithm, because a feasible solution is found instantly, and iteratively better solutions are found from that. The solution of simply not going to any nodes gives a feasible solution, which is under the maximum distance and has no reward.

[3] Another approach is the simple algorithm of finding the minimum spanning tree and traversing every edge twice, which gives a 2-approximation and can be done in nearly linear time

Lu which creates an online algorithm for the Traveling Salesmen Problem with Service Flexibility, and then complete a reduction from our problem to this one[2]. However, they prove that a 2-approximation for a polynomial time online or offline algorithm is optimal[2]. We found that a 2-approximation is not good enough for many applications and so we decided to sacrifice theoretical polynomial time to achieve a better quality of solution.

For this reason, we use an ILP because while the Branch and Bound approach is exponential, in practice, it can find a good approximation in a reasonable time frame.

3 THE FORMAL PROBLEM STATEMENT

The problem is stated formally as follows:

Given:

- the set $V = \{v_1, v_2, ..., v_n\}$, each representing a node
- the set E, where $e_{i,j}$ represents the edge from v_i to v_j
- the set D, where $d_{i,j}$ is the length of edge $e_{i,j}$
- the set of functions $F = \{f_1(t), f_2(t), ..., f_n(t)\}$, where each $f_i(t)$ represents the time dependent reward received for staying at node n_i for time t. These functions are continuous and satisfy both $f_i'(t) \geq 0$ and $f_i'(0) > 0$.
- a maximum time T

Find: the path, the set of nodes, N', which makes a cycle, and the set of edges E' such that the edge points from one node in N' to the next, and the time t_i spent at each node achieves the maximum of the sum of each reward function.

$$\max \sum_{i \in N'} f_i(t_i)$$

$$\text{subject to} \sum_{i \in N'} t_i + \sum_{(i,j) \in E'} d_{i,j} \leq T$$

4 PROPOSED SOLUTION

The idea is that we use the current ILP to generate an initial solution for some problem graph. We can now use information gained during this run to add and remove a node from the problem graph and recompute a solution to the modified graph. This portion of recomputing the solution on the modified graph will take less time than running the original algorithm on the modified graph. We describe the method we use to generate these new solutions after each change, for both the maximization and minimization problems. For each subproblem we will construct a valid ILP and while in some cases we will not solve it completely for performance reasons this allows us to maintain correctness and the anytime property.

We can construct examples such that we gain no information in the first round and so at the worse case we will have to run the full algorithm as we did for the first solution. An easy example of this is the case where we had a solution that only visited one node in the graph, we we now remove that node we have no information about the rest of the graph. For this reason we cannot improve the worse case running time of the algorithm, only the average running time.

4.1 Finding the First Solution

To find the first solution, we use the algorithm Yu et al. presented in 2015[3]. This simply sets up the problem as a single ILP and gives it to an ILP solver. For the details of how the ILP is set up, see Yu et al.[4]

4.2 The Maximization Problem

For the maximization problem the goal is to find a set of nodes, an ordering, and an amount of time to spend at each node, such that we maximize our reward without exceeding a time bound.

4.2.1 Removing a Node. Removing a node from the problem can be represented in the ILP by adding a single constraint, that the max time spent at that node needs to be 0. We now need to solve this new ILP, which is the ILP which we used in Section 4.1 with this added constraint. However, we can use information from the first run to speed up the process.

The first thing we check is if the solution is already good enough. What is often the case, is that the original solution is closer to optimal than needed, so simply skipping the node and changing nothing else leaves a solution which is still provably good enough. This is helped by the fact that removing a node may also lower the optimal result. We then check if this solution is good enough. If it is, we are done; if not, we have to recompute the full ILP.

We do this as follows:

[4]Here we use Gurobi as our ILP solver.

- First, look up the amount of reward received from the node we are trying to remove.
- Add the constraint described above to the ILP and solve the LP relaxation of the problem.[5]
- Retrieve a new upper bound, by getting the solution to the LP relaxation. [6]
- Compute the new lower bound, which is the old solution less the reward from the removed node, since we can take the same tour and not stop at the removed node.
- Check if this is good enough. If so, we are done; else, we continue and have to solve the new full ILP in our ILP solver.

However, even when we need to recompute, we can start with these upper and lower bounds, which enable more pruning from the Branch and Bound tree and speed up the problem.

4.2.2 Adding a Node. Adding a node is more complex, but we follow the same general method. We first construct the new ILP as if we were to solve the problem from scratch. This new ILP has the new constraints for the new node as well as a different objective function which includes the new node. At this point, if we solve this ILP, it is equivalent to resolving from scratch and using the technique in Section 4.1.

Instead, we once again solve the LP relaxation to give us the upper bound on the modified problem. We get the lower bound by looking at the solution to the original problem, since we know that is a valid solution to the new problem as we do not have to go to all of the nodes.

If we find that the original solution is good enough as compared to the new upper bound, we are done and our path is the same as the path we found before. If the original solution is not good enough, we add the new constraints, the upper and lower bound, which help with pruning, and re-solve the problem in full using an ILP solver.

4.3 The Minimization Problem

For the minimization problem, the goal is to find a set of nodes, an ordering, and an amount of time to spend at each node, such that we achieve a specified level of reward while using as little time as possible.

4.3.1 Removing a Node. Removing a node can once again be represented by adding the same single constraint, but this time we need to worry about falling outside of the feasible region. That is to say, removing a node from the path may leave us with a solution that is not a valid solution to the problem since it does not achieve the required minimum reward. If we can remove the node and remain feasible, then our solution must be good enough. This is because it can only take less time, and as stated, we remain feasible, so by definition, we have the required reward. On the other hand, if we are no longer feasible, then we have to once again add the new upper and lower bound and re-solve the full ILP.

4.3.2 Adding a Node. Adding a node is nearly the same as described in Section 4.2.2: we again construct the new ILP and solve the relaxation. However, in this case, the LP-Relaxation gives us a minimum, since it is a minimization problem. The upper bound is found by looking at the old solution. We check if this solution is good enough, and if not, we re-solve the ILP with the extra bounds.

5 RESULTS

Due to the fact that the Tourist Path Problem is NP-Hard, we focus only on experimental running time and not on asymptotic running time, which is exponential in the number of nodes.

5.1 The Experiment Setup

The tests were run on a machine with an Intel Core i7-4700MQ, which has four cores and eight hardware threads, running at 2.4GHz base Frequency up to 3.4GHz with Turbo Boost. The machine has 16GB of memory, but memory was not a limiting factor. All tests were run on the same machine, so we focus only on the comparative running times.

5.2 The Grid Graph

The experiments are done on a grid graph. That is to say that the nodes were on the lattice points of an equally spaced grid and nodes have a direct path between them if the distance between them is less than the width of the graph. The graphs were grids of the following height and width respectively, which will be referred to as their

[5]The LP relaxation of an ILP is the same optimization problem, ignoring any integrality constraints.

[6]Solving the LP relaxation is both computationally faster and also gives us a new upper bound, since an LP only removes constraints and thus must have a solution at least as high as the original ILP that was relaxed.

total number of nodes (length*width) {(4,5), (5,6), (6,7), (7,8), (8,10)}.

5.3 The Tests

The tests were run as follows: for all four types of problems (removing a node in maximization, adding a node in maximization, removing a node in minimization, adding a node in minimization), we first run the program to generate an initial solution and time this, then either add or remove a node, re-solve, and time once again. For removing a node, we ensure that the node removed was part of the path. This is because removing a node outside the path is trivial. For adding a node it may or may not be part of the optimal solution, since somebody can use the program to see if any node is worth adding.

We then run these 4 types of problems on all five of the graphs described above for 5 different degrees of optimality. By this, we mean that problems in a category continue until they find a solution that satisfies that level of optimality. We say a problem satisfies a certain level of optimality if we can prove that the absolute difference between the upper and lower bounds over the upper bound is less than some value. Thus, an optimality of 0 means we achieved optimality and an optimality of .5 means our solution gives a reward, or uses time, within a factor of 2 of the optimal. We consider optimality in the set {.5,.4,.3,.2,.1}. This gives us 100 total problem types: minimization and maximization, adding and removing nodes, five different sized graphs, and five different optimalities. We then ran the program many times to decrease the variance due to the high amounts of randomness.[7].

5.4 The Data

Overall, we show that finding a solution to a modified problem after the original solution is found is faster by using the algorithm described in Section 4 than solving the problem from scratch.

Since we have four separate problems and 100 different types of runs for these problems, we do the analysis in three different ways. First, we look at all of the results at once, with each run normalized by the average time that problem type took. This will allow us to see the general pattern in the results. Next, we look at the four problem types. We sort all of the runs for these problem types, keeping in mind that we have the same number of each of the 25 sub-problems. We then look at the 10th, 25th, 50th, 75th, and 90th percentile and compare the time it took for the first run and the time it took to re-solve after adding or removing a node. Lastly, we look at every problem individually and evaluate whether we can statistically say that the online algorithm is faster.

[7]Some of the sources of randomness are the following: each node has a random reward function and the node we choose to remove or add is chosen at random.

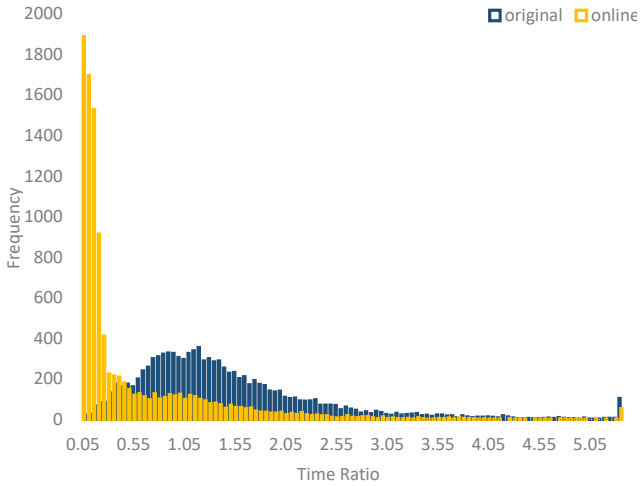

Figure 2: A Histogram showing the normalized times for all runs

5.4.1 Total Normalized Comparison. In Figure 2, we can see an overall comparison of the online algorithm to the original algorithm. This graph is a histogram of the times of all of the runs after they have each been normalized by dividing each data point by the average of all data points in the same subproblem. This allows us to compare all the different tests to one another. From this, we see two things. The first is that online algorithm is much more stacked to the left, which means that for the vast majority of runs, it takes less time. The second is that after the peak to the left, both sets seem to have similar trends. This is due to most of the performance coming from the fast paths and when it does not take one of the fast paths it takes a similar amount of time as the original.

5.4.2 The Four Subproblems. Here we look at each of the four subproblems to see the different performance characteristics of each one. For each subproblem, we take all runs, order them by time, and look at the

RT percentile	Original (s)	online (s)
10%	0.059	0.012
25%	0.138	0.041
50%	0.472	0.110
75%	1.859	0.318
90%	5.484	1.73

Table 1: Running time comparisons for removing a node in the maximization problem

RT percentile	Original (s)	online (s)
10%	0.066	0.009
25%	0.148	0.017
50%	0.508	0.067
75%	1.964	0.228
90%	5.804	0.528

Table 2: Running time comparisons for adding a node in the maximization problem

RT percentile	Original (s)	online (s)
10%	0.082	0.031
25%	0.163	0.060
50%	0.570	0.185
75%	1.752	0.827
90%	3.711	2.773

Table 3: Running time comparisons for removing a node in the minimization problem

RT percentile	Original (s)	online (s)
10%	0.068	0.000
25%	0.173	0.015
50%	0.618	0.021
75%	1.924	0.056
90%	4.063	0.114

Table 4: Running time comparisons for adding a node in the minimization problem

different percentiles to get an idea of the comparative times of the online algorithm.

Removing a node in Maximization: In Table 1, we see that when we remove a node from the maximization problem, we are faster by a factor of around four to five.

Adding a node in Maximization: In Table 2, we see that while the times for the original algorithm are similar as in Table 1, the times for the online portion drop such that there is almost an order of magnitude improvement over the original approach.[8]

Removing a node in Minimization: We see in Table 3 that while we are still faster for this subproblem, the improvement is not as strong as the others. This is because when removing a node in a minimization problem, the problem can easily become infeasible, and in this case, there are no options other than starting the algorithm over from scratch. We see this effect in the data that while the bottom 50% is almost 3 times faster than the original, the top 50% does not have a large performance improvement.

Adding a node in Minimization: This ends up being the easiest problem. We see in Table 4 that we have a performance gain of around a factor of 20. We know that adding a node is easier than removing one because when we remove a node we remove one of the nodes along the tour, as you would never want to remove a node not on the tour. However, when we add a node, we add any of the possible points from the grid, so it may be far removed from the tour, making it easier to ignore in many cases, leading to the large performance gains we see here.

5.4.3 Every Problem Type. Lastly, we evaluate if there are any specific subproblems that cause difficulty. We want to know which problems we could solve faster and which we could not. For each of the 100 problems, we ran a t-test to test if the modified problem was faster than the original. We see the p-values in Table 5 where the null hypothesis was that the online algorithm took the same amount of time. A low positive value shows that the online algorithm performed faster. A negative value means that for that problem the online was slower.

Removing a node in Maximization: We see that we are faster for all problems that require optimality less than .1 and do better with larger graphs. We do better with larger graphs because the original algorithm runs substantially slower and the fast path through our code is not as much effected by the size of the graph. When asked for high optimalities we almost always have to re-solve resulting in extra work.

Adding a node in Maximization: We see, again, that the online algorithm is always faster for optimality less

[8]The time of both of these original approaches should be the same because they are running the same code and should give the same distribution

than .1, but for this problem, the algorithm only fails for large graphs.

Removing a node in Minimization: We get the same results as before, where the online solution is faster for all but problems on large graphs that require a solution close to optimal.

Adding a node in Maximization: In this case, we are always faster, which we saw before in Section 5.4.2 that this is a particularly easy problem.

Overall, we found the following: for all problems with optimality worse than .1, the online algorithm was statistically better with p =.05.[9] For optimality .1, the modified problem is faster for adding a node in the minimization problem while the rest were statistically similar. The complete results can be found in Table 5.

6 CONCLUSION

We find that we can add and remove nodes from the problem graphs and solve them between three and twenty times faster as long as we do not need too high a degree of optimality.

We also describe ways that we can quickly and more easily compute whether we need to spend the effort in recomputing the entire problem. This could be useful in the drone example described earlier. The drone itself could do the easy computation, and if the more expensive computation needed to be done, then the problem could be sent back to a larger server to be solved.

There are several ways to extend this work. The first is to try it on different kinds of graphs, especially those which which are real world representations. The next is to try to extend it farther than adding and removing a single node. For example, we could determine the impact of iteratively adding or removing nodes in sequence, and if this iteration allows us to build up to bigger solutions than we could otherwise solve.

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[2] P. Jaillet and X. Lu. Online traveling salesman problems with service flexibility. *Networks*, 58(2):137–146, 2011.

[3] J. Yu, J. Aslam, S. Karaman, and D. Rus. Anytime planning of optimal schedules for a mobile sensing robot. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 5279–5286. IEEE, 2015.

[9] With 79 of the 80 tests having $p < .01$ and 78 of the 80 tests having $p < .0001$

Removing a node in Maximization

# Nodes in Graph	0.5 Optimal	0.4 Optimal	0.3 Optimal	0.2 Optimal	0.1 Optimal
20	6.13E-9	1.72E-9	3.18E-6	1.753E-2	-2.87E-3
30	1.91E-25	2.46E-16	4.20E-13	2.31E-6	-0.21
42	1.05E-19	1.24E-5	1.34E-6	6.58E-5	-6.40E-2
56	2.04E-27	3.68E-23	1.93E-21	2.81E-13	-0.82
80	5.32E-34	5.26E-34	2.73E-42	2.36E-24	8.15E-2

Adding a node in Maximization

# Nodes in Graph	0.5 Optimal	0.4 Optimal	0.3 Optimal	0.2 Optimal	0.1 Optimal
20	1.82E-30	2.40E-24	1.76E-20	1.270E-15	5.29E-8
30	3.08E-28	2.38E-30	1.66E-21	1.93E-16	3.78E-9
42	3.27E-17	5.70E-23	3.55E-25	1.94E-13	8.96E-2
56	2.46E-34	8.41E-27	1.18E-26	8.60E-11	0.42
80	9.70E-38	6.80E-40	9.70E-34	2.18E-14	0.11

Removing a node in Minimization

# Nodes in Graph	0.5 Optimal	0.4 Optimal	0.3 Optimal	0.2 Optimal	0.1 Optimal
20	2.02E-9	5.47E-9	3.62E-5	9.53E-6	6.89E-5
30	7.02E-10	3.23E-12	6.46E-9	3.11E-7	8.09E-2
42	6.98E-12	2.01E-12	4.56E-11	2.45E-7	0.28
56	3.40E-22	8.25E-19	1.87E-3	5.73E-9	0.14
80	6.62E-21	7.76E-22	5.20E-11	1.13E-3	0.27

Adding a node in Maximization

# Nodes in Graph	0.5 Optimal	0.4 Optimal	0.3 Optimal	0.2 Optimal	0.1 Optimal
20	4.98E-15	2.32E-20	2.97E-25	6.76E-15	3.43E-9
30	1.62E-65	1.54E-45	2.18E-26	1.22E-20	3.51E-18
42	4.92E-36	2.86E-53	1.20E-43	9.76E-39	2.16E-8
56	3.51E-49	8.72E-43	1.55E-43	1.43E-22	5.51E-12
80	6.47E-47	2.086E-37	5.73E-34	1.96E-15	3.64E-5

Table 5: The p-value for each of the 100 subproblems described. A negative p-value means that the modified problem ran slower. We have statistically significant results that simply modifying a problem is faster than resolving that problem in almost every case.