Traveling Salesman Problem: A Real World Scenario.

The world needs a better way to travel, in particular it should be easy to plan an optimal route through multiple destinations. Our main project goal is to apply a TSP algorithm to solve real world problems, and deliver a web based application for visualizing the TSP. In order to do so effectively, we must optimize our current TSP implementation to reduce computation time. In this paper, we present a greedy algorithm based approach for solving the TSP more efficiently and a web based front end that uses the Google Maps API for visualizing the problem instance and its solution. By leveraging the popularity of Google Maps, we are able to deliver a useful application to our end users.

One of the main goals of this project is to improve the performance of our existing TSP algorithm. Generally, the best approach for improving the performance of a branch and bound algorithm is to improve the lower bound. Our new strategy is to use a greedy method for edge selection, rather than trying all possible path permutations. The new algorithm works by selecting which edges in a given TSP instance will not be part of the solution [1]. Moreover, it uses a dense adjacency matrix to represent the distances between each of the points. The values in the rows and columns in the matrix are reduced by the smallest distance during each iteration. Therefore, the edge selection is determined by the edge that would allow us to have the largest possible reduction for a given iteration. This is the greedy step [2]. The program must examine all non visited edges to decide which one is the best, which takes \( O(n^2) \). However, for a 20 city problem instance, this should not be a major bottleneck. By intelligently selecting which edges to remove from the path, we converge to the optimal solution more efficiently, reducing the computation time significantly.

The greatest technical challenge associated with this part of the project is correctly implementing the new TSP algorithm and improving the performance of our previous implementation by a significant amount. In contrast with our original approach, where we would simply permute all possible solutions, the new algorithm is more selective in the paths it considers. Also, we no longer use a recursive function for calculating a particular path locally. The new implementation uses a stack data structure to ensure Depth First Search order.

We believe that the performance bottleneck in our homework 5 TSP implementation is the large amount of tasks generated for the 20 city instance. In the original design of homework 5, the threshold was set to a high value, in order to reduce the number of tasks. The threshold is used to determine when to end the recursion, computing the remainder of the problem locally. Increasing the threshold also increases the computation time for each task. For example, with 20 cities and a threshold of 17, by time the threshold would be reached, the server would have \((19) + (19 \times 18) + (19 \times 18 \times 17)\) tasks in its queue. Moreover, each task
will have an average runtime of approximately 6 seconds. In our new design, each task always
has exactly two sub tasks or \(2^{(n-1)}\) tasks for a given level \(n\). This allows us to significantly
reduce the amount of tasks generated, improving our TSP performance.

To collect some preliminary performance data for the new TSP algorithm, we used the
set of 20 cities from homework 5 and compared the results. We saw a drastic improvement in
performance. For this particular TSP instance, we were able to increase performance by as
much as 487X on a single computer (Figure 1). The TSP algorithm used in homework 5
would run for 20 minutes to solve 20 cities on a single computer. In our new implementation,
we take less then 8 seconds (Table 1). Our second test involved using our web based
application to generate 20 random coordinates on the map. After doing 15 runs, we observed
that the computation time did not exceed 10 seconds for any single run. Therefore, we were
also able to drastically improve average runtime performance for a random set of 20 TSP
coordinates. This is ideal for our web based application, taking the load off the server and
returning the results in a reasonable amount of time.

![TSP 20 City Performance](image)

**Figure 1.** TSP Performance speedup for 20 city instance.

<table>
<thead>
<tr>
<th>Computers</th>
<th>HWK 5 [ms]</th>
<th>Optimized [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1221372</td>
<td>2506</td>
</tr>
<tr>
<td>4</td>
<td>583091</td>
<td>5446</td>
</tr>
<tr>
<td>8</td>
<td>326589</td>
<td>5977</td>
</tr>
<tr>
<td>16</td>
<td>217189</td>
<td>6118</td>
</tr>
</tbody>
</table>

*Table 1.* Total Runtime for TSP for 20 city instance.
The remainder of this paper focuses on our experience in developing an application for visualizing the TSP. The main technical challenge of developing a web based front end for the TSP was the integration of Google Maps and RMI. To make this possible, a set of web 2.0 technologies including JavaScript, AJAX, Google Maps API and Servlets had to be used. Prior to this project, our experience with JavaScript, AJAX and Servlets was limited. This made coding simple programming tasks with JavaScript more difficult. Moreover, it was also a challenge to learn the appropriate Google Maps API calls to get the functionality we needed. Finally, debugging the application was also tricky, given that the bugs can now come from three places: RMI code, Servlet code and client side JavaScript code. In fact, we found bugs in our TSP code that we didn’t know existed because creating the problem instance itself was trivial with the GUI.

As stated above, coming up with a robust software architecture for our application was part of the challenge of this project. First, we had to determine the appropriate technologies for our application. The steps for doing so were as follows. The Google Maps API uses remote JavaScript, and so the majority of the code was written in JavaScript. Communication with the CSIL web server was done by using AJAX (Asynchronous JavaScript and XML). AJAX is an API that can be called from JavaScript to make requests to a server without reloading a page or submitting a form. On the server end, we decided that using Servlets was most appropriate, since it is Java based and we will need to make calls to RMI Client. The software architecture for our application is depicted in Figure 2. When the user visits our web page, his browser downloads the main page (index.html) from an Apache Tomcat web server running on CSIL on port 8080. When the browser loads the page, it makes a call to Google Maps API via a remote JavaScript function to draw the map. The user then determines a set of coordinates for each of the destinations he would like to visit. Once these are determined, another call is made to the Google Maps API to draw the points. In order to find the optimal path, the user clicks on the “Find Optimal Path” button (Figure 3). This invokes a server side Servlet running on CSIL using AJAX. We also have a Server and Computer running on CSIL, ready to solve a given TSP instance. The Servlet acts as a RMI Client and sends the coordinates to the RMI Computer, to solve the TSP. When a RMI Computer generates a result, it sends it back to the RMI client. The RMI client returns the optimal path to the user. The client side JavaScript code then makes a call to the Google API to draw the optimal route. The route is then displayed on the page, with the total distance traveled. As a result, this architecture makes it possible to use our existing RMI infrastructure and Google Maps to solve a real world TSP.
The most limiting aspect of our application is that we can only get the straight line distance between two points. As future work, we can incorporate a third party web service that provides the actual road distance between a set of coordinates. There is also rumors that Google plans to add this feature to its Maps API[3]. Another possible improvement would be to package our code as a browser plug-in, making it more accessible to the end users. With these improvements in mind, we feel that our application can already add value to all Google Maps users. After showing it off to some friends and family, they immediately came up with some creative ways for using it. One of my friends is going to Europe in the Summer, and he needs a way to find an optimal route through his destinations. My other friend is planning a fishing trip to Canada, and he would also like to optimize his driving route. In addition, my dad jogs around the neighborhood, and he would like to find out the total distance of his route. Virtually everyone that saw our project, came up with some novel application for it. Some of the use cases we didn’t even imagine, and that is when you know you have built something cool and useful.
Figure 3. Application screen shot, with the optimal path displayed.

References.

