Rideshare:
A Web Application for Automated Carpool Matching and Optimization

Submitted To

CheolHee Park
Dr. Constantine Caramanis

Prepared By

Yoni Ben-Meshulam
Garrett Cooper
Derrick Huhn
Patrick Lowry

EE464 Senior Design Project
Electrical and Computer Engineering Department
University of Texas at Austin

Spring 2008

CONTENTS

CONTENTS (Continued)

FIGURES

EXECUTIVE SUMMARY

As gas prices continue to rise, citizens of the world are faced with a choice of how they can reduce their ecological footprint and contribute to the sustainability of our planet. The Rideshare project provides citizens with an option that brings people together while reducing pollution, cost, and energy dependence associated with transportation vehicles.

Rideshare is a web application that connects drivers and passengers who seek to participate in carpools and provides these citizens with efficient routes so that travel time and distance are minimized. The following paper gives an overview of our solution to the carpooling problem and details the design choices that were made to achieve this solution.

Matching riders to drivers is neither a trivial task nor can it be easily implemented. The Rideshare team explored several avenues for accomplishing this effectively. This paper gives insight into how and why our solution can be successful in practice. We also give an overview of the challenges we faced during the implementation stage and provide a detailed summary of the events that have transpired in this past semester.

1.0 INTRODUCTION

In the following paper, we discuss the Rideshare web application in terms of design and implementation and detail the challenges, constraints, and successes the Rideshare team faced during the development process. We first introduce the carpooling problem and describe challenges involved in implementing a solution. After researching related services and papers that attempted to solve the carpooling problem, we realized the problem had no existing solutions so we had to formulate a new design. We present our structure for a solution and describe the requirements and challenges for each module of the Rideshare application. We then present the results of our implementation and characterize and compare the different ways you can run it. From our characterizations, we show how our implementation solves the rideshare problem and exceeds the initial expectations we drafted in our proposal.

Through the course of the project, we made decisions that affected the outcome of the Rideshare project. We cover the choices we made during the implementation stages and analyze the overall impact these decisions had with respect to time and cost. Finally, we asses the safety aspects of Rideshare and address the ethical issues related to environmental concerns.

The final version of the Rideshare project contains a code base that is extensible for enhancements and can be easily adapted to other software projects. We believe making the source code available to other projects benefits the engineering community and encourages its continued development. The Rideshare Team developers are Derrick Huhn, Yoni Ben-Meshulam, Patrick Lowry, and Garrett Cooper. Dr. Constantine Caramanis supervised the algorithm design and implementation while CheolHee Park acted as our technical advisor.

2.0 THE RIDESHARE PROBLEM

Rising fuel prices are pressuring people to rethink their transportation alternatives. One of the best choices is cooperative transportation, or carpooling. The carpooling alternative is one which is widely recognized as a good solution for alleviating traffic congestion [1][2][3]. Currently, there is very little being done to assist people in organizing carpools. Websites that are dedicated to carpooling do little more than offer their users a bulletin-board for posting their ride offers and requests. Users are left to search for and organize carpools themselves [4][5].

The Rideshare application helps people coordinate carpool programs and eliminate the need to plan an ideal route and schedule. People who regularly share rides will reduce their ecological footprint and save on travel expenses. In addition, our solution will help people find rides that would otherwise not be possible without exhaustive effort. The problem Rideshare solves is that of taking user ride requests and offers, and producing matches based on compatibility.

The underlying optimization problem is a well known problem called the Vehicle Routing Problem. The following excerpt from the Wikipedia entry explains it well:

> *"The vehicle routing problem (VRP) is a combinatorial optimization and nonlinear programming problem seeking to service a number of customers with a fleet of vehicles. Proposed by Dantzig and Ramser in 1959, VRP is an important problem in the fields of transportation, distribution and logistics. Often the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. Implicit is the goal of minimizing the cost of distributing the goods. Many methods have been developed for searching for good solutions to the problem, but for all but the smallest problems, finding global minimum for the cost function is computationally complex."*
>
> *--Wikipedia entry for the vehicle routing problem [6]*

Our version of this is an optimization problem where one searches for the optimal matches between two sets of people: those requesting rides and those offering rides. In other words, we are matching riders to drivers while globally optimizing the routes to match the most number of people and minimizing the distance traveled by all the drivers. Formally known as the Capacitated Multi-Depot Vehicle Routing Problem with Time-Windows (CMDVRPTW), this problem is classified as NP-Hard, which means that there are no known algorithms that find the optimal solution in polynomial time.

Rideshare provides a service that produces sets of matches that may not be the most optimal solution to the problem, but are generated within reasonable time-limits such that the results could be used by the people being matched. In addition to simply matching people, we wanted our service to also consider the individual preferences of Rideshare users. These preferences include allowing a user to select age-ranges and the gender of their match and time windows for departure and arrival.

The problem is therefore one of creating a web application for users to enter their trips and preferences, and providing them with rideshares. The rideshares must represent an optimal solution to the CMDVRPTW.

3.0 OUR SOLUTION

We first attacked the Rideshare problem by researching algorithms that could possibly be employed. The simple brute-force algorithm, alternatively known as the exhaustive-search algorithm, has an exponential time-complexity relative to the number of riders and drivers being matched. Due to the time-complexity, we decided to implement the brute-force algorithm as a base-line for comparing the results of the other optimization algorithms while using small test cases. After researching different optimization methods, three candidate algorithms were proposed: the genetic algorithm, bipartite graph matching, and linear systems solving. Since no references could be found that had actually attempted to solve the problem using any of the aforementioned algorithms, we had to implement all chosen algorithms and compare their performance ourselves. Having a limited amount of time to work on the project, we decided not to implement linear systems solvers and instead focused our attention to the the brute-force, genetic, and bipartite graph matching algorithms. For the map interface on the front-end, we wanted to make use of existing map frameworks such as Google Maps, Yahoo! Maps, and MapQuest rather than developing a home-brewed solution [7][8][9]. Interfacing with these frameworks required us to develop a user interface (UI) that is predominately web-based.

Implementing these algorithms required the team to decide upon a structure for our test data and generated solutions. One possibility was to have a single application that allowed you to enter all your data at the beginning, work on the unit, then display results when it was done; however, we determined this wouldn't be representative of normal behavior. We imagined users would more than likely enter their data individually and continuously instead of inputting all their data at once. We decided to deal with this by separating the optimization part of the service from the web-based front-end, and have them communicate through some central data housing mechanism. Disjoint front and back-ends enabled us to continuously bring in data from the front-end to the database while the back-end queried from the database at configurable intervals. This structure also enabled us to implement the two components with completely different interfaces to the database, tailoring them to their programming language.

Having laid out rough requirements for the solution structure, we looked for a platform to house our diverse components. We considered many solutions for each category but eventually settled upon GNU/Linux as our host system, the Apache webserver and PHP/CSS web toolkit for the front-end, MySQL for the database solution, and a C++ framework for the back-end

[10][11][12][13][14]. All of these languages and tools used in enterprise environments, and GNU/Linux also provides a robust platform to develop C++ software in addition to excellent software-project management and build tools. Furthermore, all chosen solutions are open-source and free of cost.

3.1 Front-End

As previously mentioned, we decided to implement the front-end as a web-based PHP solution. We needed to provide a means of distinguishing users from one another and allow them to set preferences, and create and view routes and assigned matches through a map interface. The front-end needed to validate all data input by users - authenticated or not - and write this data the database which could later be processed by the optimizers.

We decided to implement an authentication-based structure, common among web-applications, where users authenticate themselves with a password they set upon creating their account. For security reasons, we decided to never store the users' passwords in plain-text; instead we saved a digest of the password. This policy addresses the possibility of identity theft if our server ever became compromised since an attacker would not be able to extract our users' passwords.

The map of web-pages can be seen in figure x.xx. [FIXME] The number of pages is kept to a minimum so users are less likely to get lost while navigating the site. As can be seen in the design layout screenshot of figure x.xy, [FIXME] the interface is designed to be minimalistic and intuitive so that the users can focus on what is important. We decided to go with Google Maps as opposed to Yahoo! Maps or MapQuest for aesthetic reasons and their simple yet effective API.

In addition to using traditional forms of notifications such as email to notify users of matches, we also integrated Really Simple Syndication (RSS) feeds into the user interface [15]. RSS feeds provide new ways of interaction by allowing users to subscribe to Rideshare content from within other websites or other applications such as iGoogle and Outlook, respectively. This allows Rideshare's users to view their results in real time since the RSS feed would be generated dynamically using PHP.

3.2 Database

The database contains all the data brought in from the front-end and represent it in such a way that both the front-end and back-end are able to understand it. As such, we opted to implement the database as what is known as a relational database. This type of database allows us to have separate tables containing different types of information that are linked to each other when appropriate. This linkage avoids data duplication across records and reduces the overall overhead involved in storing the data. The amount of data duplication or complexity of the data structures being represented can be thought of as being at odds with the complexity of the database structure. To reduce data duplication or be able to represent more complex data structures, one has to increase the complexity of the database structure, increasing the number of tables and links between them. This strategy is known as Database Normalization [16]. We chose to go with a database structure that was a balance between complexity and accessiblity, about the middle range of the hypothetical normalization scale.

3.3 Controller Back-end

The back-end of the Rideshare application - the Controller - was composed of four components that operated on the data entered by users: the Controller Library, optimizers, algorithm visualizer, and a notification mechanism. The optimizers communicated with the database through the Controller Library which served as the primary interface for back-end components to retrieve users' ride offers and requests. The Rideshare team faced many unique design and implementation decisions when crafting the Controller Library - appropriately named libController - and even included third party libraries such as the MySQL++ and MapQuest libraries.

3.3.1 Controller Library

The Controller Library is the mechanism by which optimizers communicate with the database. The library contains data structures necessary to represent rideshares, in addition to logic for retrieving and writing information from and to the database server. Instead of rewriting all the logic for querying the database server, we decided to leverage an existing library: MySQL++ for C++. Since all the Rideshare team members would eventually interact with the database during the development process, we wrapped the MySQL++ library inside of our database access API.

This level of abstraction gave each team member the comfort of interfacing with one ubiquitous API, rather than dealing with the intricacies of third party libraries. Wel also applied this approach to MapQuests' Advantage API, as it required only one member to learn how to use the library.

3.3.2 Optimizers

Our design for the optimizers involved a modular approach. We first created a framework containing all the functionality the optimizers need to execute but focused on making the functionality usable by all of the optimizers instead of specific to each.

Using this framework, we implemented three optimizers. We required optimizers to accept an initial solution, i.e. a set of rideshares and unmatched riders, and return an optimized solution. We designed the optimizers to accept rideshares that still had a vacancy in the car and potentially matched riders to the rideshare in later optimization sessions. The optimized solution returned by the optimizer represents the solution with the highest score it had found while optimizing, i.e. best possible match of riders to drivers.

The following details the functionality of each optimizer. Pseudo-code for each of the optimizers, as well as the scoring function, can be found in Appendix #.

The first optimizer used the brute force approach. This naive approach involves checking every possible solution in the search space and returning the highest scoring solution. The time complexity for this optimizer is too large for it to return an optimized solution in a tangible amount of time so we did not use it in practice. Figure # illustrates the brute force approach.

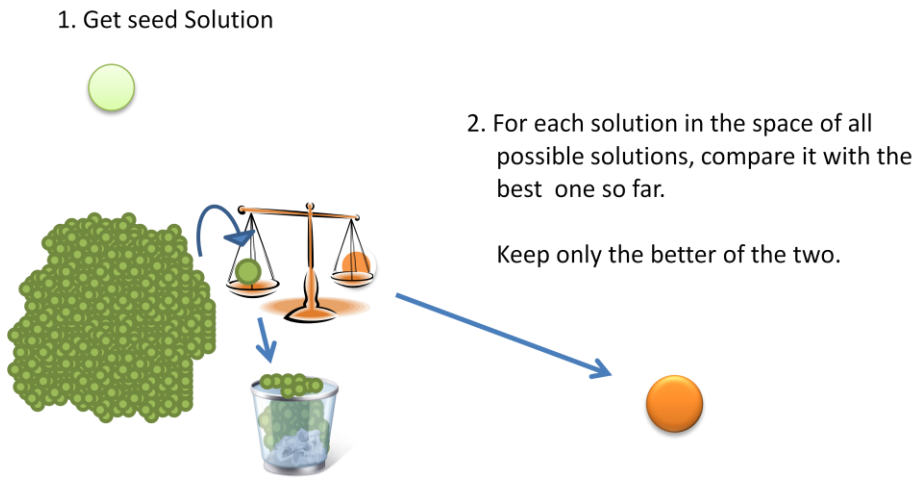


Figure 3.1 Brute Force Algorithm

The second optimizer used the genetic approach. This approach uses the theory of natural selection. It begins with an initial solution and spawns a population of other solutions based on mutations of the original. Taking the best solutions from the population, we again introduce mutations and generate a new population. This process is repeated until a predetermined amount of time has elapsed, or the solution scores have converged to an optimal score. Figure # illustrates the functionality of the genetic algorithm.

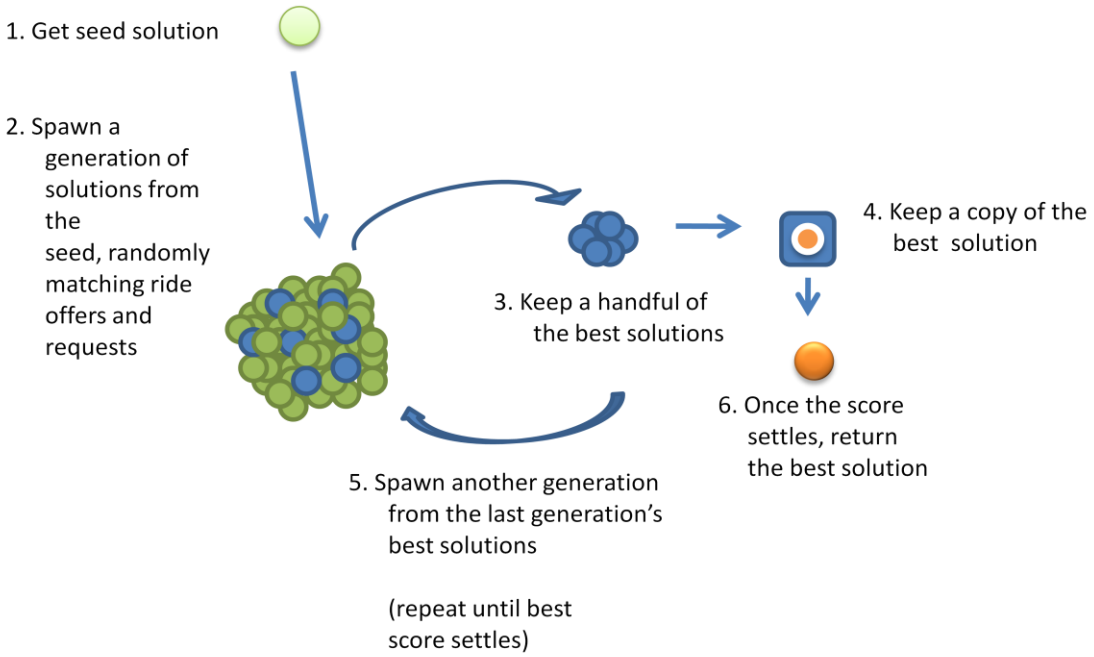


Figure 3.2 Genetic Algorithm

The third optimizer used an approach for maximal matching on bipartite graphs. This approach views the problem as that of matching between two disjoint sets of points: rideshares and unmatched riders. These two sets have a relationship of compatibility between them, namely, a rider's compatibility with the rideshare. The algorithm for maximally matching the sets utilizes some properties of paths between these sets. Figure # illustrates how it works.

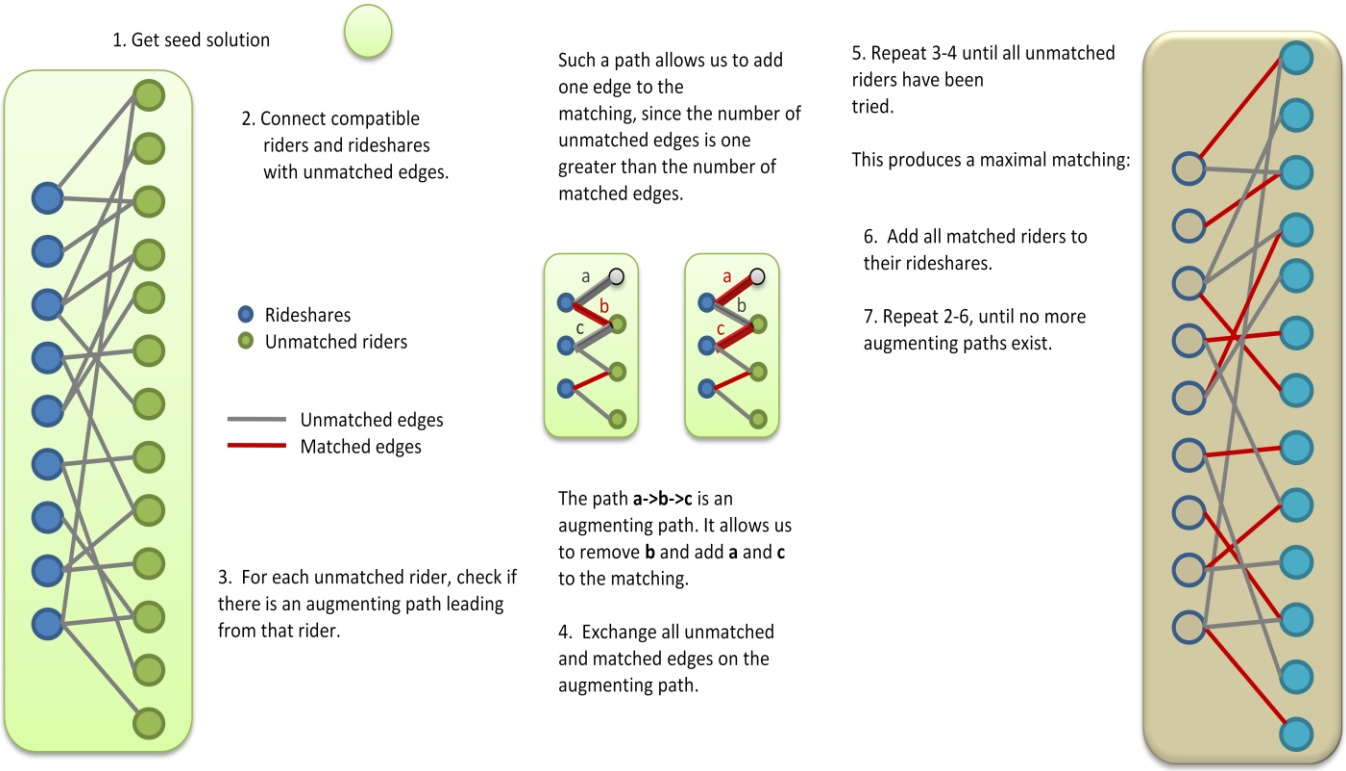


Figure 3.3 Bipartite Graph Algorithm

3.3.3 Algorithm Visualizer

Although we were able to verify the correctness of optimizers through unit tests and simulated test cases, the Rideshare team also developed a tool for visually analyzing the results produced by the different optimizers. Even though the algorithm visualizer was initially proposed for debugging purposes, it later became an aesthetic feature that exhibited the core functionality of the back-end of the application.

The algorithm visualizer worked in concert with the optimizers as it took 'snapshots' of the intermediate solutions generated during the different stages of optimization. As each optimizer performed calculations, a 'camera' embedded in fixed stages of the algorithms would capture the current state of the solution and save it for later use. After an optimizer completed working on a set of rider and driver submissions, all of the 'snapshots' become viewable through the algorithm visualizer. These images were created using the MapQuest Image API. When we designed the visualizer, we had to examine several design alternatives. The correct language, framework, and tools to use were not obvious from the requirements of the visualizer so we initially considered Java and Ruby as the language for implementation and Swing and GTK s the Graphical User Interface (GUI) tool kits. Eventually, the team decided to use XHTML and JavaScript since Patrick had already established a framework and style that could be easily adapted for the visualizer.

Generating the match information in a usable format for the visualizer was accomplished through meta-programming by generating JavaScript data using a C++ program. This C++ program translated the information in the solution 'snapshots' into JavaScript and also generated images of the solutions by invoking the image creation feature of the MapQuest library. Once the JavaScript and images were generated, the XHTML page would then execute the JavaScript and an animation of the different solutions would be displayed. A screen shot of the algorithm visualizer can be seen in Figure x.xx.

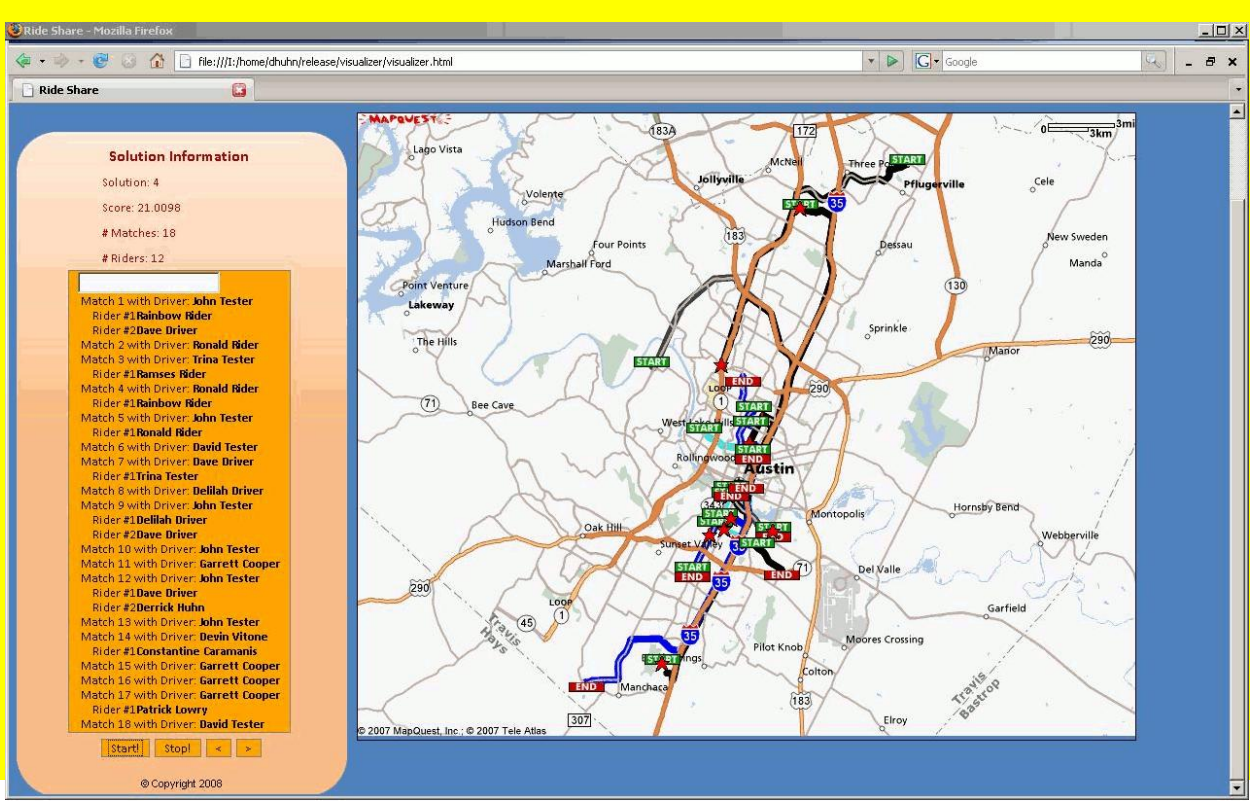


Figure 3.4 Algorithm Visualizer

3.3.4 Notifications

The Controller program also has the responsibility of notifying users that have been matched in a rideshare. After solutions have been produced by optimizers inside the Controller program, they are written to the database and can be viewed through the My Routes page from the Rideshare website. Following the database update, an email notification is sent by the Controller program and the user's RSS feed is also updated. These two forms of notifications were designed to keep Rideshare users updated on their current matches in real time so they can adjust their traveling schedules appropriately.

4.0 CHALLENGES AND IMPLEMENTATION

During the development process of both the front-end and back-end, the Rideshare team faced a series of problems related to programming languages, library limitations, and the design of the different components. For example, the front-end and back-end were implemented in part using the Asynchronous Javascript and XML (AJAX) and C++ programming languages, respectively,

and each of these languages had idiosyncrasies associated with them. In the following section, we describe the challenges we faced during the implementation stages and how overcoming these obstacles shaped our final design.

4.1 Front-End Challenges

Web development has recently seen a wave of new technologies such as XHTML, CSS, AJAX, and RSS feeds. We decided to adopt as many of these new technologies as we saw necessary. This proved to be both a blessing and a challenge, because although they provide new and wonderful functionality, documentation was often sparse and sometimes we had to drastically change our programming paradigms to accomodate.

4.1.1 XHTML

XHTML is a rewrite of the core language used to create webpages (HTML). It is much more rigorous in its syntax and now conforms to the XML standard. This strict conformance solves the long-standing problem that webpages were displayed differently in different browsers. Once this standard is followed, all browsers should render exactly the same. However, XHTML, though similar to HTML, is a different language, and many of the familiarities of HTML have been removed. As a result, we found ourselves having to refer directly to the language specification when attempting to implement webpages [17].

4.1.1 Asynchronous Javascript and XML (AJAX)

The Google Maps API is built upon a new technology called AJAX that allows webpages to make use of other websites' data by simply querying for it. The complication is in that it doesn't return immediately: it returns once the query is complete. Following their API, we implemented callback functions that Google would execute once our request was complete. This proved to be rather problematic, as the data for points in a route was often dynamic, so we had no fixed location to put the data once it arrived. Instead, we had to make our webpages adapt using complex Javascript. When the data arrived, the page was able to browse itself, and locate and place the data in the appropriate style or format such as a hyperlink.

4.1.2 Validating user input data

Our webpages were also responsible for making sure that the data input by users was valid. A substantial portion of the overhead in creating a new page went into simple validation of input

data. Data that had a regular format but could not be simply compared, like age preferences, had to be parsed using a technology known as regular expressions. Regular expressions are a programming construct included in many scripting languages, including PHP, and allow you to compare a structure to text to see if it matches, even if specifics about the text being tested aren't known. Learning and implementing them proved to be a feat; however, most of the bugs found in the webpages were related to mistakes in our implementation of regular expressions.

4.2 Controller Challenges

While implementing the Controller and its different components, we encountered difficulties with the C++ programming language and required the assistance of other team members. We attempted to reduce these types of language specific problems by creating template classes that new classes would be modeled after. Additionally, we setup a development infrastructure that allowed each Rideshare developer to easily contribute to the project and enhance or debug existing source files.

4.2.1 Remote Development

The Controller Library was the only module in which all Rideshare team members contributed to the code base. This required us to setup a system for all users to use so that everyone had a uniform and consistent development environment. As a result, Rideshare team members developed the Controller Library remotely and were required to use the same tools and integrated development environments (IDEs); however, not all team members had previous experience in these environments. Having a shared IDE amongst the Rideshare team eventually contributed to the robustness of the library and Controller program since we were able to focus on logic problems rather than dealing with environmental inconsistencies. The complicated design of the library also introduced problems not considered during the initial design phase.

4.2.2 Build System

After understanding the complexity and requirements of the libController code base, we first setup a build system to systematically compile our code and create executable files for release. This build system would need to be extensible and support both third party libraries and code developed in-house. After drafting multiple methods for accomplishing this task, we decided to use a GNU Tool called Make as it was very commonly used in a wide range of C++ projects and Derrick had experience writing Makefiles. The Make build system also played an important role

in encapsulating the features in libController so that the library could be both portable and reusable in projects outside of Rideshare. Although the build system was one of the early challenges we faced during development, it played a vital role in allowing the Rideshare team to focus on writing code rather than deal with build related problems.

4.2.3 Third Party Libraries

Another challenge we faced when developing libController was incorporating third party libraries into our code base. Since these libraries were not developed by the Rideshare team, we were unable modify or enhance their features. For example, MapQuest's Advantage API provided developers with a way of calculating route distances as a batch query to minimize the number of requests sent to the MapQuest server. We took advantage of this feature, the RouteMatrix class, but discovered there was a limitation on how many locations could be specified in the batch request. The free version of the MapQuest Advantage library limited developers to processing only 25 locations per batch request, however, the optimizers would be operating on data sets with hundreds to thousands of locations. This limitation required us to create a similar class to RouteMatrix that used a local agent for calculating distances instead of MapQuest's servers. Our RouteMatrixLocal class shared the same interface as MapQuest's RouteMatrix class so the change required minimal change to other parts of libController. We also faced other problems with using third party libraries with inconsistent documentation.

Although the MySQL++ library was very useful and effective at performing database related tasks, the documentation provided to developers was out of date and incorrect. Additionally, the documentation for the Boost DateTime library lacked sufficient examples so we were forced to figure out how they worked on our own. These unexpected inadequacies required us to devote more time to resolving these problems and we subsequently drifted behind schedule. In the end, however, we were able to successfully implement all the features in the back-end by working through spring break and also taking advantage of an extra week of development time given to us by our technical advisor.

4.3 Optimizer Challenges

4.3.1 Optimizer Framework

In order to implement the optimizers, we first needed a common framework that the optimizers would build upon. In this section we discuss two of the more important methods.

We needed a general scoring function for all optimizers, allowing us to evaluate and compare the different optimizers. Each optimizer returns a solution, which includes a set of rideshares (a rideshare is taken to mean a driver with zero or more riders) and a set of unmatched riders. We used two criteria for scoring a given solution: the primary criteria is the number of users matched and the secondary criteria is the inverse of the total route distance. The total route distance is the sum of the route lengths over all drivers, taking into account deviations due to carrying riders. This scoring function has interesting properties such as a tendency to score higher solutions that spread out riders across drivers, as drivers without riders are not counted as matched and total route length is effectively a tie-breaker of solutions when the same number of people matched.

Figure 1 [FIXME] details the equation the scoring function uses. The integer part of the score corresponds to the primary criteria, and the fraction corresponds to the secondary criteria for scoring.

$$Score = \text{Users Matched} + \frac{1}{\text{Distance}}$$

Figure 4.1 Scoring Function

Optimizers would also need to be able to check for compatibility between a rideshare and an unmatched rider. We took a long time developing the compatibility check and were forced to redesign several aspects of it throughout the semester. In order to assert that a rider is compatible with a rideshare, we check that all of the following criteria are met:

1. The rider's age, gender, and rating comply with the age, gender, and rating preferences of all members of a rideshare.
2. All members of a rideshare's ages, genders, and ratings comply with the rider's age and gender preferences.
3. The maximum capacity of the vehicle is not exceeded.
4. The rider's schedule is within the time windows of the driver's schedule.
5. The rider's beginning and ending locations are within a reasonable distance from the driver's beginning and ending locations, or along the way.

6. The route length of the resulting rideshare would not cause the driver to deviate more than a reasonable distance out of his or her way.
7. The schedules of all members of a rideshare, as well as the rider's schedule would all be met if the rider is added to the rideshare.

The first three criteria are simple comparisons and require nothing more than a simple series of "if...then" statements to implement. The fourth and fifth criteria are essentially common sense criteria, i.e. heuristics. The role of these criteria is to minimize the use of the sixth and seventh criteria which have much larger time costs than the rest of the criteria combined. The reason for this is that they both require us to first find the optimal order for going from the driver's origin, through all of the rideshare members' locations, and ending at the driver's destination.; This route should also include the prospective rider.

In order to compute the optimal route, a simple brute force method is used, which requires that we check all possibilities. The number of possibilities can be shown to be the factorial of the vehicle's capacity. Since the capacity of a vehicle for riders generally does not exceed four, the time required can be considered on the order of O(4!)=O(24)=O(1). In big-O terms, that is sufficient to consider it constant time for purposes of the optimizers.

Once the optimal route is computed, the order of points and time and distance required to reach each point is known. This allows criteria number six to be checked by comparing the total route length with the driver's initial route length. If it is within a factor of 1.3 of the driver's initial route, then that criteria is met.

Finally, making sure that everyone's schedule is met involves two stages. The first stage involves finding the narrowest time window for the driver to leave. The second stage is adding to this time window the time required to get to each of the members' locations in the order of the optimal route. At each location, we check that the time window is within the corresponding user's schedule.

Figure 4.2 illustrates the algorithm for asserting the schedule criteria is met. We narrow the driver’s leave window until we reach a worst case leave window for the driver. We then add to this leave window the time necessary to get from point to point, each time verifying that the users’ time windows do not conflict with the arrival window for that point.

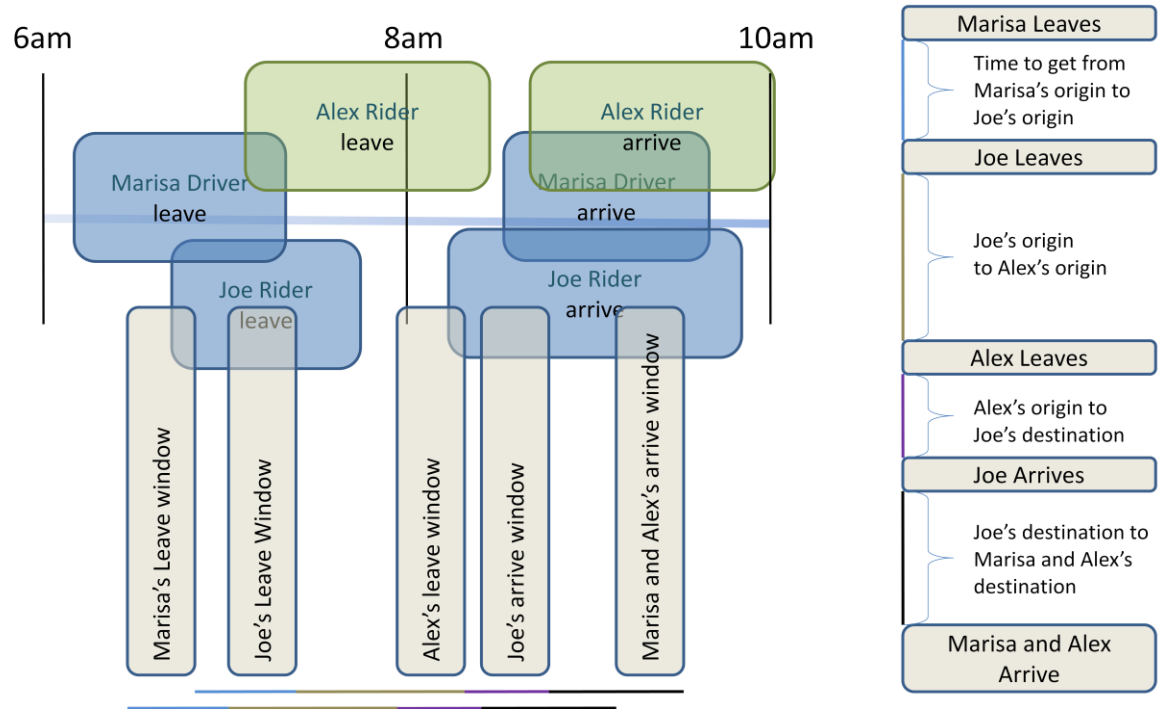


Figure 4.2 Schedule Criteria

4.3.2 Optimizers

Brute force and genetic optimizers proved to be very simple to implement, once we had the framework ready. The framework methods included most of the logic used by both optimizers, making implementation a straight-forward task.

The bipartite matching optimizer was a different story. We needed to use several graph algorithms, and planned on using one of several C++ graph libraries, including Boost, and LEDA [18][19]. When we tried using each of these libraries, we met with lots of problems. Eventually, we abandoned our original plan, and decided to augment our existing data structures to provide an interface for the bipartite matching optimizer. Using this interface, we implemented the necessary graph algorithms, which could then be used for bipartite matching. This alone probably set us back about two weeks on all of the optimizers.

5.0 VERIFICATION AND EVALUATION

Rideshare was primarily developed using the Test Driven Development paradigm. For new every feature we implemented, we also wrote a corresponding unit test to verify the feature

worked correctly. Subsequently, we executed all other unit tests to ensure that the new feature didn't break existing code. We believe this design pattern was very useful since we were able to quickly detect bugs and design problems. Each team member also owned specific modules of code so when a unit test failed, the owner was notified and the issue was addressed. We had over one hundred unique assertions that ranged from testing the functionality of copy constructors to verifying the behavior of the MapQuest libraries.

For design testing, we created several unique corner cases designed to test the optimizers and verify correct behavior. Each of these cases tested a different function: one simple case put three identical riders in the path of the same driver who only had the capacity to pick up two of them. This simple cast tested the logic in the canAddRider function pertaining to the capacity of drivers and was used to verify that a driver could not be assigned more riders than it had spaces available. Other tests verified the distance logic such as whether the driver would pick up the closer of two riders, or how far the driver would go out of his way to pick up a rider. As we tested each of these cases, we were able to appreciate having multiple optimizers since a few of the bugs we fixed in one optimizer were found quickly as a direct result of comparing results from another optimizer using the same inputs. For the bipartite matching optimizer, we even had a test case specifically designed to verify the algorithm was finding augmenting paths correctly.

In order to accurately simulate real Rideshare users, we acquired actual user data from a shipping website called uShip [20]. Though the data they gave us was all from Texas, it was all across the state. Initially, we put all of the routes in the database under Garrett's account, but we felt this did not adequately simulate real users. We wrote a program augment this data, listing the routes under different test users, each with individual preferences. We randomized whether the route would be a driver or a rider and what capacity the driver would have. Each route also had a stochastic probability of including specific preferences. Once we established the functionality of the optimizers using the corner cases, we used sixteen test cases from the enhanced uShip data to test the time and score performance of the algorithms. The data has several properties which are important to consider when analyzing the resulting data.

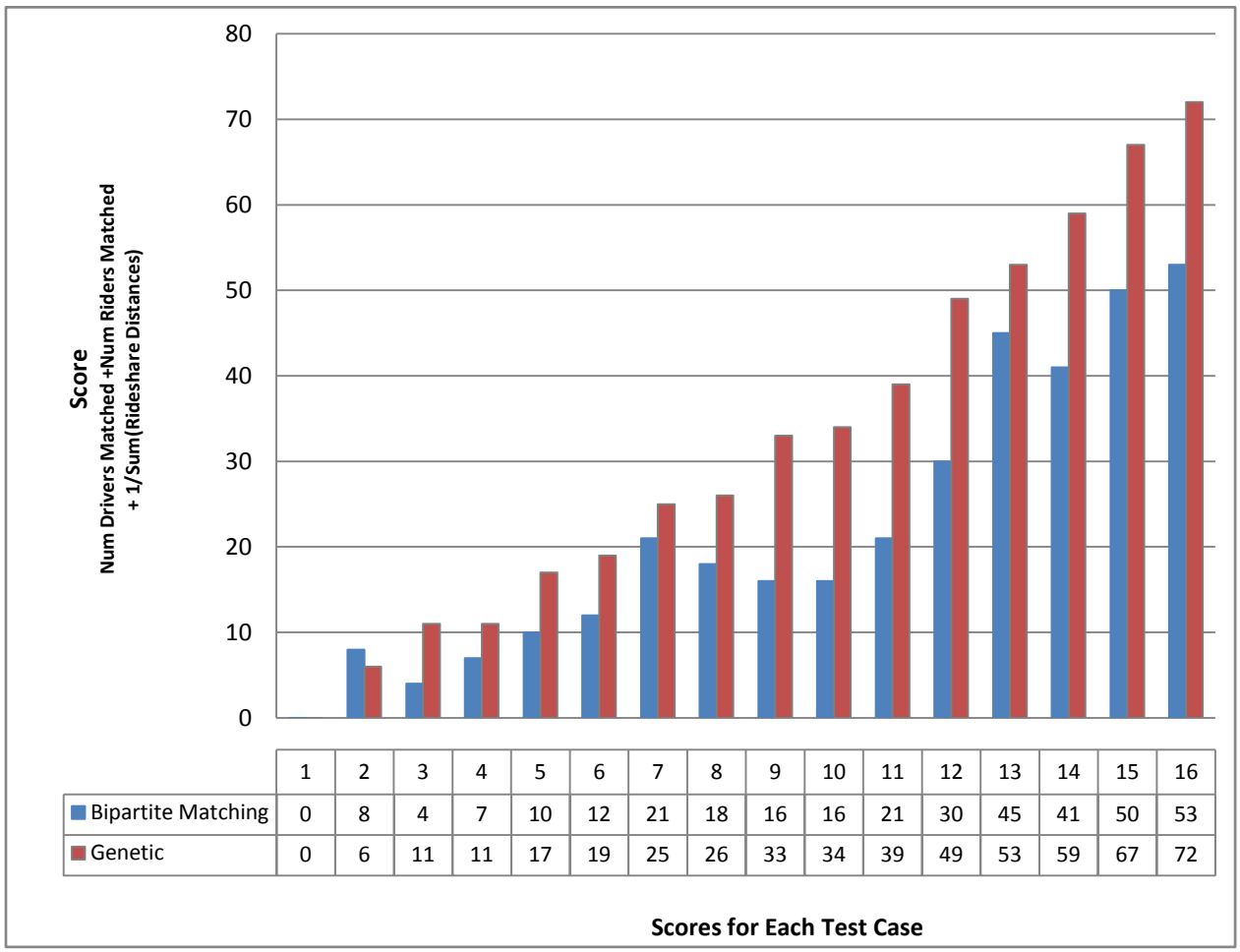
The first property observed in the results was that the data sets increase with the number of users. This property ensures that we can measure horizontally over the test cases with a standard measure of the number of users. In addition, all of the test cases contain the same distribution of

drivers and riders, which is roughly one driver for every two riders. Finally, it is important to note that each test case is independent of other test cases i.e. the users in one test do not necessarily include the users from another test.

With these facts in mind, we now take a look at the test results. We will not discuss the performance of the brute force optimizer in this section, because already at the third test case, it fails to complete execution within a reasonable time frame. If we were to graph the results with the other optimizers, brute force's would be off the charts.

In Figure 5.1, we compare the score results for the bipartite matching and genetic optimizers. The horizontal axis shows each of the test cases 1-16, and the vertical axis shows the score. For the genetic optimizer, we observed that the score increased linearly. On the other hand, the score of the bipartite optimizer has a more irregular pattern which changes for each test case. Last, we noticed that the ratio of the bipartite matching score to the genetic score increased, i.e. the bipartite algorithm might be "catching up" with the genetic algorithm with respect to score.

From these results, we concluded that the algorithms perform differently on the same data, and the advantage of using the genetic optimizer for larger test cases is clear. Furthermore, it would be of interest to use larger test cases and determine if these patterns continue or change with the size of the test case.

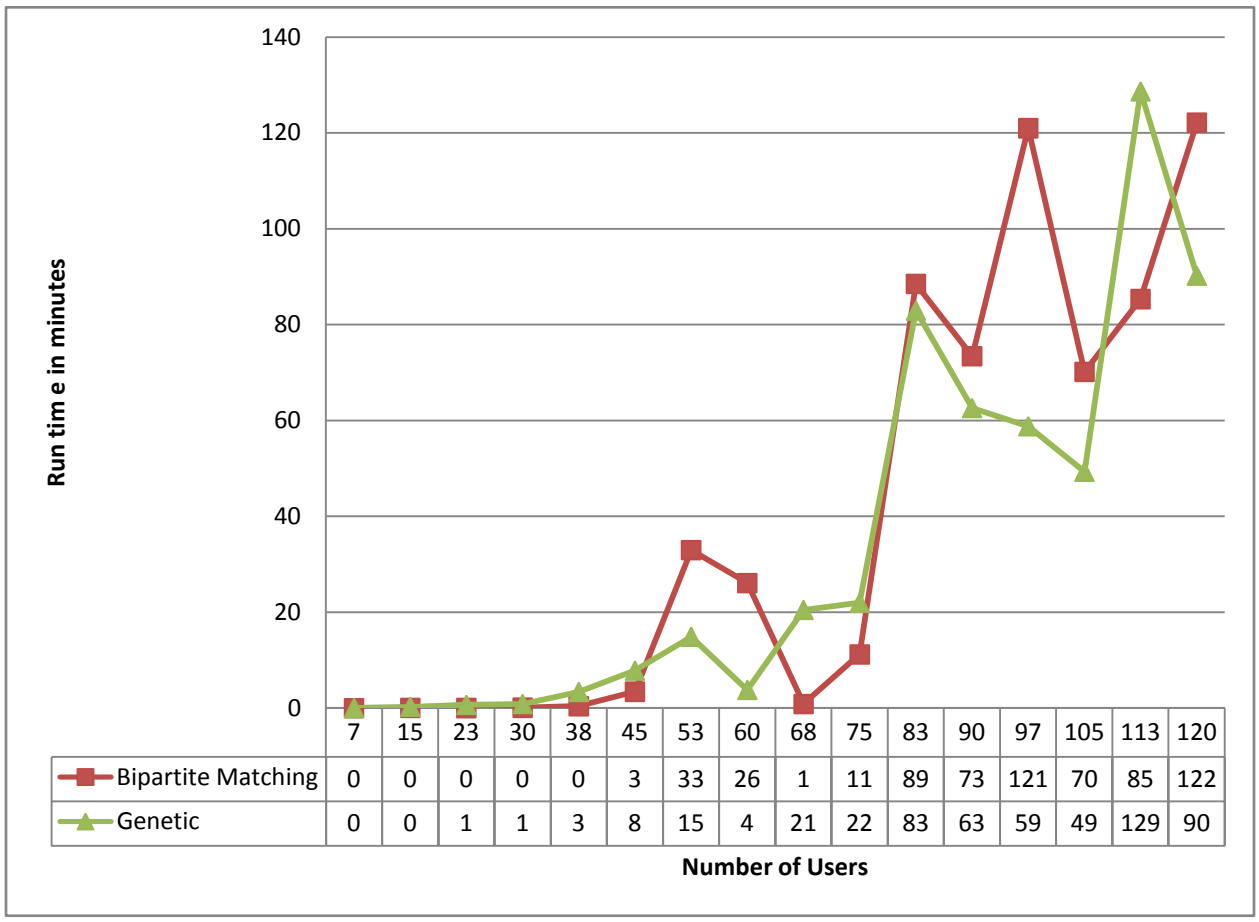


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Bipartite Matching	0	8	4	7	10	12	21	18	16	16	21	30	45	41	50	53
Genetic	0	6	11	11	17	19	25	26	33	34	39	49	53	59	67	72

Figure 5.1 Score Comparison: Bipartite Matching vs. Genetic

Figures 2 and 3 show the number of users verses time for the bipartite matching and genetic optimizers. Figure 2 shows the time in minutes, whereas Figure 3 shows time on a logarithmic scale.

In Figure 2, we see that the time complexities are somewhat similar for both optimizers. Each of them has several rises and falls in the time. Some of the points seem to show dips in both graphs but not all of the points are correlated. In addition, it looks like the run time increases rapidly over the first test cases, but slows down and even reverses over the larger test cases. We would need to test further with more data points and larger test cases in order to find a curve for the time complexity, but we can already tell it's not growing too rapidly.

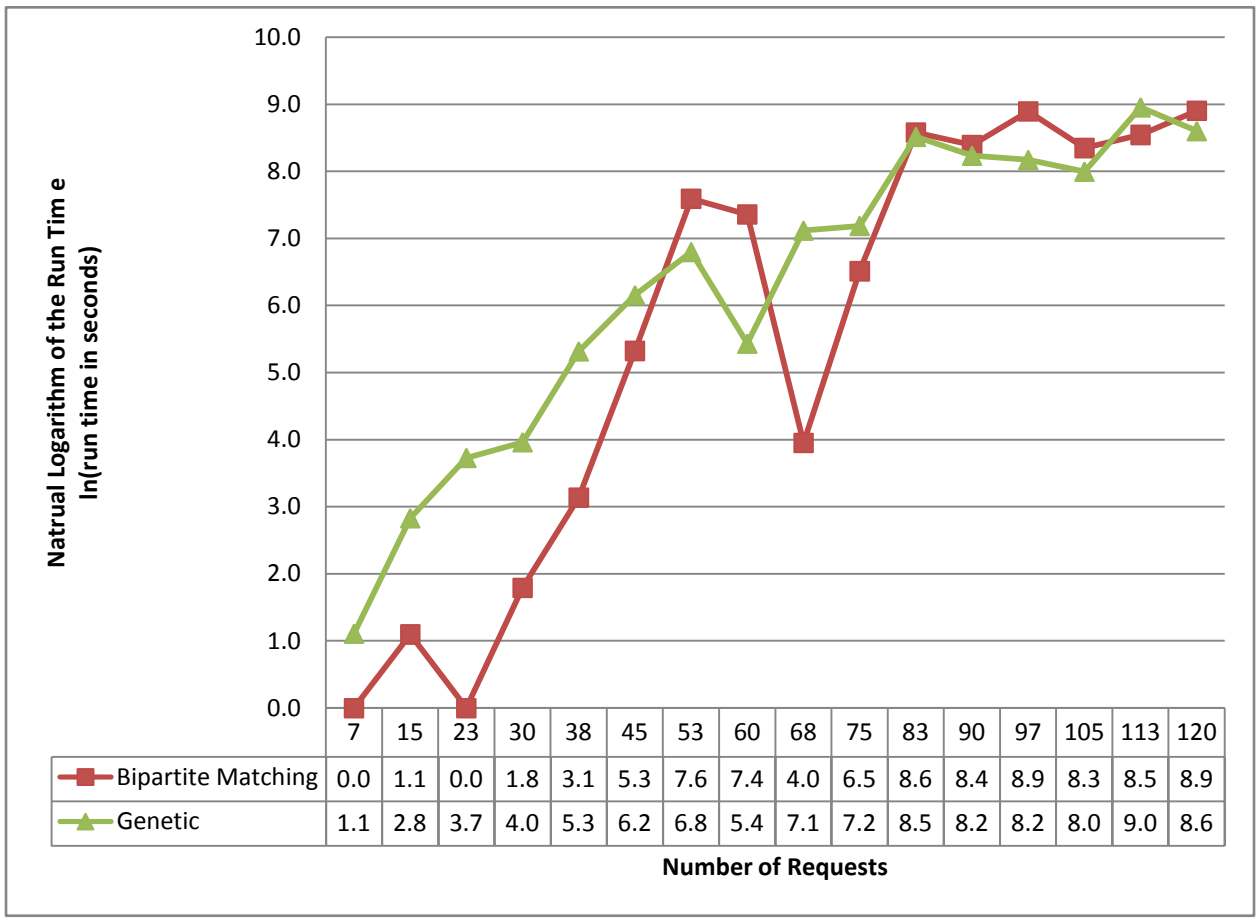


	7	15	23	30	38	45	53	60	68	75	83	90	97	105	113	120
Bipartite Matching	0	0	0	0	0	3	33	26	1	11	89	73	121	70	85	122
Genetic	0	0	1	1	3	8	15	4	21	22	83	63	59	49	129	90

Figure 5.2 Time Comparison: Bipartite Matching vs. Genetic

In Figure 3, we take a look at the same data from Figure 2, though we evaluate it on a logarithmic scale. The point of this evaluation is to compare the time complexity of the optimizers to that of an exponential curve. In a logarithmic graph, a linearly increasing graph represents exponential growth.

Sure enough, the first test cases cause what appears to be exponential growth in the amount of time taken, but the larger test cases seem to break this pattern, and follow a flat, fairly steady curve. This is good news, and confirms that our optimizers do run in polynomial time. Again, we would like to add larger test cases to confirm this pattern.



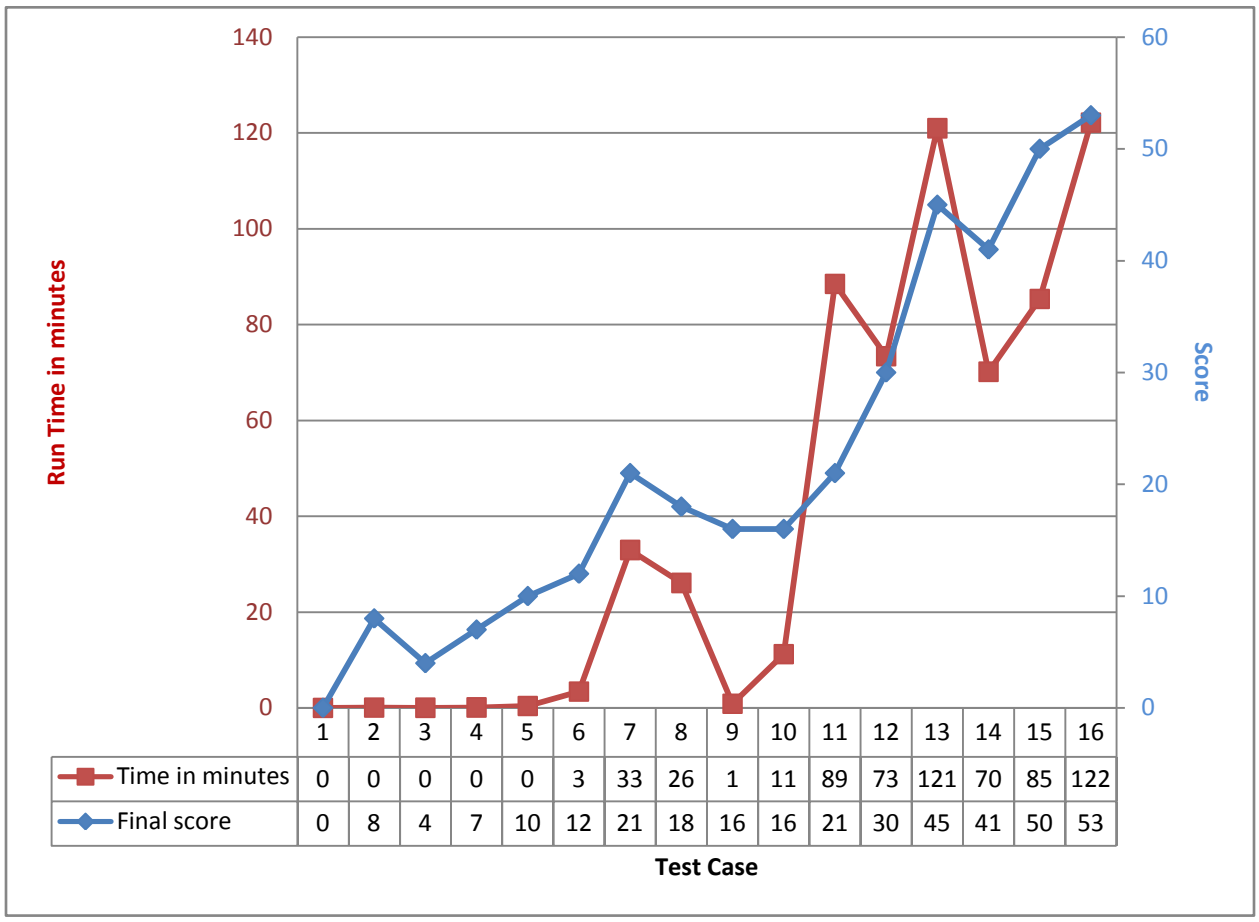
	7	15	23	30	38	45	53	60	68	75	83	90	97	105	113	120
Bipartite Matching	0.0	1.1	0.0	1.8	3.1	5.3	7.6	7.4	4.0	6.5	8.6	8.4	8.9	8.3	8.5	8.9
Genetic	1.1	2.8	3.7	4.0	5.3	6.2	6.8	5.4	7.1	7.2	8.5	8.2	8.2	8.0	9.0	8.6

Figure 5.3 Logarithmic Time Comparison: Bipartite Matching vs. Genetic

In Figures 4 and 5, we compare the final score with the run time for each algorithm. The motivation for this graph comes from the fact that the previous graphs suggested that the time is correlated to something other than the number of users. If each point had the absolute optimal solution, we would expect to see a linear curve for the score.

Bipartite matching seems to have dips in both scales around the same test cases. This implies that the time to complete bipartite matching is strongly correlated to the final score it produces. This result might be due to the fact that bipartite matching runs time consuming compatibility checks more frequently when it has compatible users.

When we take a look at the genetic optimizer, Figure 5, we see what looks like a near perfect linear increase in the score. Therefore, it seems, the genetic optimizer's time complexity is not correlated with the score it produces. Further, the genetic optimizer seems to be producing a curve which reflects our expectation of a linear increase of the score with the number of users.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time in minutes	0	0	0	0	0	3	33	26	1	11	89	73	121	70	85	122
Final score	0	8	4	7	10	12	21	18	16	16	21	30	45	41	50	53

Figure 5.4 Run Time vs. Score: Bipartite Matching

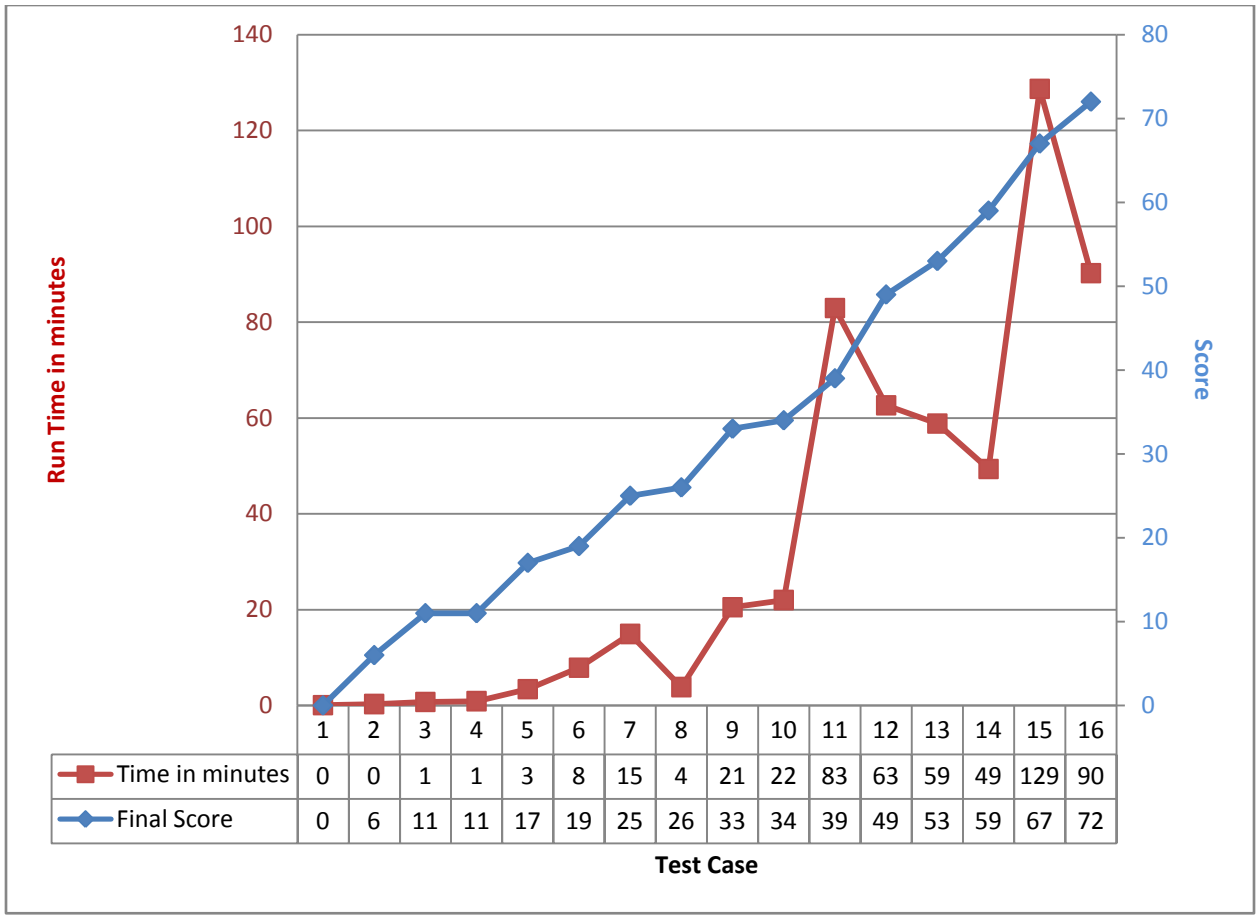


Figure 5.5 Run Time vs. Score: Genetic

The results from Figures 1-5 tell us a lot about the performance and functionality of our optimizers. We were especially surprised with the irregular increase in the time required by both algorithms, and the linear increase in the score for the genetic optimizer. In addition, we established that our optimizers operate in polynomial time.

The end result, though, is more questions than answers. Can we expect these trends to continue? Will the behavior of the optimizers behave differently with test cases in the thousands of users? If we had time, we could generate more results to test our conclusions.

At no time during the semester did any team members purchase any hardware or software for the Rideshare project. The systems used for the web server and optimization development box were generously loaned to us by the department; they also paid all operating costs.

Although the Rideshare team members spent most of their time developing effectively in C++ and PHP, there were times when some developers spent more time debugging language related

problems. Overall, we estimate that approximately 15% of our time was devoted to debugging C++ related problems; this does not include 20% of development time dedicated to debugging logic and design problems In the C++ code base, we also dedicated time to removing cyclical dependencies from our code i.e. removing the dependency between two files, as this prevented us from compiling our code. We addressed this issue by using a programming technique called forward declarations to tell the compiler that the referenced function existed without explicitly including it.

We spent about five percent of our time writing code we eventually did not use. This includes initial attempts at developing components we kept such as the Solution class as well as features such as the compatibility matrix that were not necessary to implement. This matrix would have been used by the bipartite optimizer to try to offload the compatibility checking to the start of the function, but as it turned out, we would have had to update the matrix every time we added or removed a rider from a match. Since the run-time cost of the compatibility matrix outweighed the benefit of using it, we decided not to include it in the final version of Rideshare.

The problems we faced with C++ related issues did affect the progress with respect to our proposed schedule, but no cost adjustments were necessary as these changes were made purely in software. Specifically, all three optimizers took longer to complete than we had anticipated. Each of them was built on top of the Controller Library which did not reach its schedule milestones due the number and complexity of its components. Every component of the Controller Library needed to be functional before the optimizers could even run on small test cases. For instance, a small problem in the RouteInfo class caused problems four abstraction layers up in the optimizers. Additionally, we could not characterize the optimizers without large, complete test cases. The test cases were completed late because we had not yet fully implemented the database access functions that it required, and around that time had less time for Rideshare because of other classes.

We feel that we have spent an above average time on our project. It consists of over ten thousand lines of code across two servers with custom-installed software. We had to spend many late nights working on it when we had other classes taking our time during the day. In the end, we implemented the core functionality to an ambitious project and we feel that our time was well spent.

7.0 SAFETY AND ETHICAL ASPECTS

Since Rideshare is a software project, many of the safety concerns shared by hardware products did not apply. Rideshare can not pose direct bodily harm to any of its users, nor will it pollute the environment; however, someone with questionable intentions could use the website to find a victim though this was not considered during the design stages. Within the scope of this course, we did not have time to implement advanced predator-catching features such as background checks or even simple measures to verify that users are actually who they say they are. If we asked for too much information, some people would shy away from losing their anonymity and privacy, and our user base would suffer as a result of our prying.

We have, however, included a warning message to the users on the About Rideshare page. This message encourages users to incorporate common sense when riding with people they have met through the internet. In addition to restricting Rideshare to users older than seventeen, we also gave users preferences to help match them with other users within their comfort zone. The gender and age preferences were designed specifically with safety in mind. A young woman could set her preferences such that she would only be matched with other young women. When she got to the car of her match, she would very likely be able to tell whether it was actually another woman to be matched with.

If we were to launch this project, we would pay special attention to trust and making it as safe as possible. We would most likely launch as part of an existing trusted and heavily moderated social networking site like Facebook [21]. These sites have an existing trust network, i.e, each user already has a list of friends that they presumably trust. If they were to restrict their matchings only to their friends, they would already trust who they are matched with.

With respect to ethical issues, Rideshare has the potential to positively affect the environment since a large number of people using Rideshare would reduce their carbon footprint since the number of drivers on the road would decrease. Rideshare could reduce the demand for gas and foreign oil, though it would take a very large number of users to make a large impact.; We can save people time and energy by helping them find rides, and we can promote cooperation and help users make friends. Within reason, we have done everything we could to protect our user's safety. If they use common sense and the tools we've given them, they will be safe.

8.0 CONCLUSION AND RECOMMENDATIONS

Throughout the past semester, we completed most of the tasks and features introduced in our project proposal. We created a web application for matching users who want to participate in carpools and solved a design problem that involved several complex challenges, including confronting an intractable optimization problem, designing an interactive, map-based user interface, and communicating results to users using email and RSS feeds. Moreover, we needed these components to work together harmoniously and utilize a common database. Confronting and overcoming these challenges provided us all with a major learning experience and a great feeling of accomplishment.

Our approach began with setting up a collaborative work environment which would enable us to discuss how to solve each of the challenges, utilize a ticket system, and keep a complete revision history of all of the code. We designed the software with modularity in mind, in order to allow for simultaneous development by all team members. This modularity ensured that we could easily add or replace features in the future. In addition, we used the Test Driven Development paradigm which increased the reliability of code during each stage of development.

There were four major components to our design for Rideshare: user interface, database, controller, and the optimizers. The idea behind this design was to decouple the user interaction from the optimization process, making the database an interface between these two processes. In addition, we developed a testing framework for unit-testing each individual component of the software, and also created a set of larger test cases for assessing the functional and time performance of the optimizers. Each of the major components had its challenges but in the end we completed all but two of the original features we proposed.

If we were to launch this product, we would implement some of the features we had initially wanted to use, like friends lists. We would try to launch from within existing trust networks like Facebook and would try to add additional safety features.

We have each learned a lot from this experience in terms of the theoretical knowledge, technological tools, development environment, team work, and communication skills. We feel that .

REFERENCES

[1] Washington Department of Transportation, "Washington State Freeway HOV System", Washington DPT, http://www.wsdot.wa.gov/hov/ (current 2008).

[2] North Carolina Department of Transportation, "HOV Planning Guidelines", NCDPT, http://www.ncdot.org/projects/hov/hPlaGuidelines.html (current 2008).

[3] Wikipedia Contributors, "High-occupancy Vehicle Lane," Wikipedia, http://en.wikipedia.org/w/index.php?title=High-occupancy_vehicle_lane&oldid=186485776 (current 24 Jan. 2008).

[4] eRideshare Inc., "E-Rideshare" eRideshare, http://www.erideshare.com/ (current 2005).

[5] TouchBase Consulting Inc., "CarpoolConnect" Carpool Connect, http://www.carpoolconnect.com/ (current 2007).

[6] Wikipedia Contributors, "Vehicle Routing Problem," Wikipedia, http://en.wikipedia.org/w/index.php?title=Vehicle_routing_problem&oldid=178307741 (Current 16 Dec. 2007).

[7] Google Inc., "Google Maps API," Google, http://code.google.com/apis/maps/index.html (current 2008).

[8] MapQuest Inc., "Link to MapQuest," MapQuest, http://www.mapquest.com/features/linktomapquest (current 2008).

[9] Yahoo! Inc., "The Yahoo! Maps Developer APIs," Yahoo, http://developer.yahoo.com/maps/ (current 2008).

[10] Canonical Ltd., "Ubuntu," Ubuntu, http://www.ubuntu.com/ (current 2007).

[11] Apache Software Foundation, "The Apache HTTP Server," Apache, http://httpd.apache.org/ (current 15 Jan. 2008).

[12] The PHP Group, "PHP: Hypertext Preprocessor," PHP, http://www.php.net/ (current 17 Mar. 2008).

[13] World Wide Web Consortium, "Cascading Style Sheets," W3C, http://www.w3.org/Style/CSS/ (current 29 Feb. 2008).

[14] MySQL AB., "MySQL," MySQL, http://www.mysql.com/ (current 2008).

[15] Wikipedia Contributors, "RSS," Wikipedia, http://en.wikipedia.org/w/index.php?title=RSS&oldid=209250278 (current 30 Apr. 2008).

[16] Wikipedia Contributors, "Database Normalization," Wikipedia, http://en.wikipedia.org/w/index.php?title=Database_normalization&oldid=209174566 (current 30 Apr. 2008).

[17] World Wide Web Consortium, "XHTML Abstract Modules," W3C, http://www.w3.org/TR/xhtml-modularization/abstract_modules.html (current 2008).

[18] R. Rivera, "Boost C++ Libaries," Boost, http://www.boost.org (current 23 Apr. 2008).

[19] Algorithmic Solutions Software, "LEDA 6.1," LEDA, http://www.algorithmic-solutions.com/leda/index.htm (current 2008).

[20] uShip Inc., "uShip.com," uShip, http://www.uShip.com/ (current 2008).

[21] M. Zuckerberg, "Facebook," Facebook, http://www.facebook.com/ (current 2008).

APPENDIX A – PSEUDO CODE FOR ALGORITHMS

APPENDIX A – PSEUDO CODE FOR ALGORITHMS

```
GeneticOptimize( Solution S )
1   Solutions  Population, Best
2   Population.GenerateNextPopulation(S, NULL)
3   time = amount of time to run
4   time.countDown()
3   while(time not expired)
4       for each solution in Population do
5               score(solution)
6       Best = getBest(Population)
7       GenerateNextPopulation(Population, Best)
8   end while

GenerateNextPopulation( Solutions Population, Best )
1   if( Best = NULL)
2       for each solution in Population
3               solution = S
4   else
5       for each solution in Population
6               randomly assign riders to rideshares in solution

Score( Solution S )
1   score = S.numRidersMatched() +  S.numDriversMatched()
2   original_route_lengths  = Sum( drivers' seed route lengths )
3   total_route_length = Sum( final route lengths )
4   deviation = original_route_lengths/total_route_lengths
5   score += deviation
6   return score
```