**DATE:** February 14, 2008  
**TO:** CheolHee Park  
**FROM:** Yoni Ben-Meshulam, Garrett Cooper, Derrick Huhn, Patrick Lowry  
**SUBJECT:** Rideshare Web Application – System Design

**Introduction**

Rideshare is a web application that connects drivers and passengers who seek to participate in efficient carpools without the overhead of searching through listings and planning routes.  Rideshare must solve the problem of examining dynamic sets of ride offers and requests and finding efficient routing solutions.  Ultimately, Rideshare will maximize the number of shared rides, incorporate users' preferences, and present the results to participants in a timely and user friendly manner.

In our proposal, we described the problem of ride sharing and outlined our solution: a web application with a database backend, and optimization software that will be implemented using any of the four major algorithms we've created.  In this design review, we examine our design objectives for solving the ride sharing problem.  We describe the resources used to satisfy these objectives and provide a summary of related research and tools.  During the design process, our team made architectural decisions for the different modules of the application. We discuss the reasoning behind these decisions and explore alternative designs we considered, including our choices of software components, architecture, programming language, development environment, and project management tools.  For example, the Web 2.0 boom has birthed a plethora of free Map services such as GoogleMaps, Mapquest and Yahoo! Maps; the results of our tests and research shaped our decision to use GoogleMaps' and Mapquest's API for the front end and back end, respectively.

In addition to listing the services Rideshare will be using, we also analyze decisions related to the scalability and extensibility of the site.  Even though the Rideshare project may never entertain a large user base, we will design the back end of the system to be distributed and reliable.  The internals of Rideshare, namely the controller and optimizer modules, will be distributed across several servers.  Although these components are not visible to the user, the timeliness and accuracy of their results will have a significant impact on the user's experience, since distributing the data processing will lower the amount of time it takes to generate results.

**Design Objectives**

In order to match drivers and riders, we must understand the underlying optimization problem.  We need to take a set of drivers (D) and riders (R), and find routes between their starting locations and destinations such that we maximize the number of riders served.  Users can specify constraints for their route lengths and time-windows for departure.  They may also request to be only matched with users of a certain gender or age.  We must take all of these constraints into account and generate valid rideshares.

A similar problem, the Vehicle Routing Problem (VRP), can be summarized as finding minimal routes for a certain number of vehicles (k) to visit a certain number of customers (n), where the vehicles start at a central location called a 'depot.'  Although our problem is similar to a VRP, we allow for some users to not be matched in the case that their criteria is not met, or if there are no drivers routed near their starting location.  In addition, our problem has several constraints, which can be summarized as follows:

Constraints:

* Each route must originate at the driver’s origin and terminate at their destination.
* Each vehicle's schedule must adhere to the driver and riders' time window constraints.
* The total number of passengers in a vehicle at any given time cannot exceed that vehicle's capacity.
* User preferences, such as gender, ratings, and age, must be met.

Objectives:

* Maximize the number of riders matched.
* Maximize the total efficiency of the routes, in terms of time and distance, on a global level.

This problem is formally known as the Capacitated Multi Depot Vehicle Routing Problem with Time Windows (CMDVRPTW).  It is a variation on several well-known problems, which have been shown to be NP-Complete, meaning that the time it takes to solve them grows exponentially relative to the number of inputs.  Timeliness of results is important, since an effective result which doesn't complete in a timely manner is useless.  This last requirement necessitates the use of an algorithm which does not run in exponential-time, instead, it must complete in some bounded time-frame tolerable by humans, or at least produces an acceptable approximation to the solution [1][2][3][4].

In order to find an optimal solution to a given set of riders and drivers, we will employ several algorithms which have been shown to solve some of the variations on the VRP.  We will evaluate the algorithms based on their run time (i.e. the amount of time it takes them to finish), as well as the quality of their results.  A successful algorithm will generate good results within a bounded and configurable run time.

Each module of the site will contribute to the usability and scalability of the site, so our design must satisfy the needs of the user in addition to solving a unique engineering problem.  As the user base of Rideshare grows, the routing components of the back end will require more computation power so a centralized solution may not suffice.  Our solution distributes the required computations over a network so that disjoint problems can be solved simultaneously.  This will require generating test cases that represent both small and large user populations.  The results of these test cases will be evaluated by an internal scoring function.  We will also visualize the results using Google Map visualization tools, i.e. the routes will all be added to a navigable map.  As we develop different modules of the site, these test cases will help ensure the growth of the site does not affect the integrity of our results.  While scalability is an important factor in designing a web application, maintainability and extensibility will also influence the design of the modules; thus, our modules will be designed with code reuse in mind.

Designing a transparent User Interface (UI) is also imperative as it will determine how often users utilize the Rideshare service.  The UI design must balance both the visual and operational components in order for the system to be usable and adaptable to a user's changing needs [5].

We have included a data flow diagram which outlines the data flow through each component described in the following sections. See Appendix A: Data Flow Diagram.

**Alternative Designs and design choices**

As with any web-based application, many choices are available at every level of implementation.  From operating systems and programming languages to collaboration tools and content management systems, the availability of multiple alternatives required us to evaluate the pros and cons of early decisions.  The alternatives can be divded into two groups: framework and implementation.  The framework consists of the operating system and service programs, e.g. the web server, whereas implementation consists of choices such as programming languages and external packages, e.g. a map library, and structure.

In our implementation, we have a user interface which we call the front end, and a back end which consists of two units: a controller and a group of optimizing programs.  The controller spawns the optimizing programs and provides them with data to operate on.  This data represents a set of distinct driver/rider requests to be matched.  The optimizers return a solution, i.e. a set of matches of riders to drivers, which will then be confirmed or denied by users.

**Framework**

To provide a web-based interface to the end users, we must have a server that hosts the pages and manages user sessions.  Two very popular web servers to choose from are Apache, an open-source project freely available from the Apache Software Foundation, and Internet Information Services (IIS), a proprietary solution from Microsoft.  Both are widely used on the Internet for large scale and personal projects. Each platform restricts what tools are available at other levels of implementation; these differences are described in Table 1.

|  |  |  |
| --- | --- | --- |
| Supported Operating Systems | Windows and Linux | Windows Only |
| Web Servers | Apache | IIS |
| Web Toolkits | PHP, Python, Perl, Ruby on Rails | ASP/.NET |
| Databases | MySQL, Oracle, PostgreSQL, DB2 | MySQL |

**Table 1: Framework Compatibility**

As shown in the above table, some packages impose dependencies.  For instance, if we decided to use the ASP .NET framework, we would have to use IIS and therefore Windows; alternatively, PHP requires Apache.  It would also be possible to design a solution that does not use web toolkits, though this solution would not be capable of processing or displaying dynamic data.

Similarly, to house the data generated by users, we must use a data storage system: either a flat file implementation or specialized database program.  Flat files can not be changed easily, whereas specialized database programs provide the functionality and flexibility to easily modify table structures.  However, the web front end requires an interface to database programs.  MySQL, PostgreSQL, Oracle, Microsoft SQL Server (MSSQL), and IBM's DB2 are popular database programs that are easily interfaced by all the web toolkits.  The latter three are all commercial programs whereas the former two are free and open-source. As referenced in Table 1, only MSSQL has an operating system dependency.

We first evaluated ASP.NET, MSSQL, and IIS as possible platforms as they all require Windows as a host operating system.  ASP.NET has a generic database interface named Open DataBase Connection Model (ODBC) that allows it to connect to all the previously mentioned databases.  A .NET site could be implemented in any Microsoft .NET programming language, however, no one on the team was very familiar with .NET technologies.  MSSQL offered standard database functionality: table editing, querying, and database management, but lacked notable additional features[such as?]  Similarly, we found that Apache promised everything IIS offered.  Though IIS and MSSQL would have been adequate, free alternatives such as Apache and MySQL also offered free support from the open source community.

Python and Perl are scripting languages capable of being used for web development, though their original designs were for other purposes.  Both languages use the ODBC to allow them to connect to any of the database programs.  These languages, though extremely adaptable, aren't as well suited to web programming compared to those developed specifically for web development.  PHP, like ASP, was developed solely with the intent of it being used as a web toolkit.  PHP, though it does not directly use the ODBC, can connect easily to any of the database solutions mentioned and has been used in many popular websites.  Having found a suitable replacement for ASP in capability, we decided to use PHP as a web toolkit, and consequently had to use Apache as a web server.

MySQL, PostgreSQL, and MSSQL offer much of the same functionality and features.  Oracle and DB2, both proprietary database solutions, offer additional development tools and configurability, though at a much higher cost.  After designing a preliminary database structure, we decided that the advanced features and configuration were not needed by a project of this scope.  We subsequently chose MySQL because it is freely available and offers more resources for development compared to PostgreSQL and MSSQL.

With a web toolkit, web server, and database selected, we then had to choose whether to run on Linux or Windows.  We looked at cost again: Linux is free, and Windows is not; we decided to use Linux.  With the framework decided, we could now focus on real design.

**User Interface**

We designed a very minimal interface in an effort to keep it as easy to use as possible.  The color-scheme was decided to be a contrasting orange on blue so that information presented to the user is very noticable and easily read.  However, we wanted the design to be very flexible and extensible so it could be easily modified at a later time.  For this reason, we decided to extensively use the Cascaded Style Sheet (CSS) web standard while implementing, and actively try to reuse as much of the user interface design as we could.  This lets us define and share style preferences for all of the web pages in a single location which is easily changed later, updating all of the pages at once.

We decided to implement the concept of authenticated users so that we could establish a rating system between the users, similar to seller rating system on eBay and other online markets.  This concept necessitates a login page, a member profile page, and a de-authentication page, or a logout page.  These pages are detailed below.

**Figure 1:**  Login Page

The login screen is simple and intuitive.  Every user will be required to log in, entering the website through the secure session manager. If the user does not already have an account, they will be able to register through the link on the bottom right. Beyond this, the top of this page will have a logo and attractive heading for the website. The bottom left will have a link to an unsecured About page, where the user can read basic information about what the website does.

The authenticated user page presents the user with actions they can perform.  For example, users can create a new route, confirm adding a matched user to a route, change their preferences, and log out.  This page will also contain the status of pending matches e.g. driver or rider approval pending.  Each available action to the authenticated user would have to be a seperate page, except for the logout page, which would simply take them back to the login page.

**Figure 2:**  New Route Page

On this page, the user can input whether they are a driver or a rider, which will open the relevant options, as shown by figures 3 and 4 below. Each user must enter whether they are a rider or a driver so that the site can display relevant options. The optimizer needs to know how many passengers a driver can accommodate, and the number of passengers and amount of luggage in each ride request.  The user will then input their route, in terms of where their origin and destination in the form fields, or by using the map interface. They will also be required to enter a time window for their trip. The time window must be specified at least partially, but the user does not necessarily need to enter both. For example, if the user specifies only the departure time, then the application will be able to use the estimated drive time to specify the arrival time.

**Figure 3:** Driver’s New Route page

**Figure 4:** Rider’s New Route page

Figure 5 shows how a passenger confirmation page will look. This page allows a user to view a potential passenger, their details, and how the combined route would look. The passenger’s name (in this case “John Rider” will be a link to their profile, where the driver can view their details. The stars by the name represent the user’s rating. In this case the user has a rather good rating of 4 out of 5. The example page indicates that the passenger has not yet accepted the driver, which is indicative of the double-handshake process in which both the passenger and the driver have to approve each other before the rideshare can take place.

**Figure 5:** Passenger Confirmation Page

The profile page will allow the user to specify their preferences for rides (age range, gender, etc.) as well as change their information (age, gender, etc.) for matching purposes.  It will also allow users to change their authentication password if they so desire.

Each of these pages will have the home bar at the top, as shown in figure 2. Once logged in, every page the user sees will have this bar visible at the top. Through it, the user can navigate through the website and view the following pages:

* My Routes, which is the home page of users and will display the status of the rides the user has created or been added to.
* My Profile, where the user can change their details and set personal options.
* New Route, which would take the user to the new rideshare page, Figure 8.2.
* Logout, which de-authenticates the user and takes them back to the login page.

**Controller**

Transparent to the user are the controller and the optimizers.  The controller dictates the data fed to optimizers and the optimizers attempt to produce an optimal matching between groups of riders and drivers.  The controller is responsible for starting the optimizer programs and telling them what data to work on.  At first glance one might ask why we do not have a single optimizer being fed all the data; the reason: parallelization.  We can benefit by having multiple computers running separate optimizers on disjoint sets of data, producing solutions faster than a single one could.

Problem Reduction

We must first identify disjoint sets of data during a phase of execution termed "Problem Reduction."  In problem reduction, we take a set of drivers and riders, and reduce it to multiple disjoint sets, by recognizing that many matches are not possible.  These disjoint sets can then be fed to independent optimizers, since they represent separate optimization problems.

We do this by generating a compatibility matrix where on one axis we have all the route requests (riders), and on the other we have both requests and determined routes (riders and drivers).  The idea is that we can easily exclude matches that are clearly invalid based on preferences, or significantly different time windows or geographic locations.  Each entry of the matrix will hold a value of true if the rider's preferences do not conflict with the route, and false if there is a conflict.  The matrix will then be split into groups of riders and drivers that could possibly be matched such that no driver or rider in any one group is in any other group.  The controller will then assign each group to an optimizer.

Controller Library

In our design we have a controller dividing work amongst optimizer programs so we must have a common mechanism for querying the database and route information. The Controller Library will be used to access the database and retrieve route data from a map service, which is needed for optimizing routes.  An intelligent design of the controller library could significantly reduce code duplication and possibly contribute to the success of the distributed back end.  There are many approaches to implementing this code base.  The programming language we decide to use for implementation will dictate what existing libraries we will use as resources.  In the C language, there exists a mySQL client library for interfacing with a MySQL database and the CURL library can be used for making HTTP requests [6][7].

Alternatively, the libraries mysqlpp and libcurlpp are C++ wrappers for the C libraries mysqlclient and CURL, respectively.  This means they allow programmers to use C++ object-oriented programming to access the C libraries by wrapping them inside a C++ library.  We also have the option of making the Controller's code base as a flat-file or hierarchy structure.  Though a flat-file structure would be easier to implement, a nested file structure would allow for easier maintenance since different components would be separated into different directories in the file system.

Since the Controller Library will be the optimizer's gateway for accessing route information, the library must be able to retrieve this information from an online maps service and create a usable object for the optimizer.  Most of the notable map services provide only a web interface for geocoding i.e. they are not easily interfaced by anything outside of the web like a C or C++ library, so using these services in a C/C++ library did not seem easily feasible at first.

**Map Service**

From its conception, we envisioned Rideshare as a web application with a transparent map interface, such that users could be easily matched and view the routes of their assigned ride share.  We considered several different map APIs for this interface and weighed the pros and cons of each service.  Important factors we evaluated for each service included query limits, directions support, and the quality of geodata, i.e. Points of Interest (POI).  In table 2, Each service is shown below with pertinent data [8] [9].

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Service | API | Directions Capability | Querying Limits (per day) | Pricing (annual) |
| Google Maps | Javascript | Basic | 10k | Free  Enterprise Edition - $15k |
| Yahoo! | Javascript | None | 15k | Free Enterprise Edition – $10k |
| Mapquest | Javascript, C/C++, Java | Advanced | None | Free, Enterprise Edition - $5k |

**Table 2:** Map Services

There is a slight distinction between how the map service will be used on the front end compared to the back end.  The UI will not be using the geocoding capabilities provided by a map service; instead, the UI will use the map view and navigation features for display purposes only.  This differs from the back-end, since the controller library will use the map service strictly for obtaining route information between different locations.  In fact, the front end and back end may use two different map services independently if one map service does not provide all the features required by that end of the system.  We recognized multiple possibilities for each, described below.

Approach 1: Use Geogoogle Java Library

The Geogoogle Java library provides an interface to a subset of the GoogleMaps Application Programming Interface (API) and can be used by any C/C++ program through a Simplified Wrapper and Interface Generator (SWIG) [10].  Geogoogle allows conversions between postal addresses and latitude/longitude points but it does not provide access to GoogleMaps' Directions API.  Since the goal of the controller library is to provide other modules with a transparent interface to routing information, Geogoogle was not sufficient for our needs.

Approach 2:  Make HTTP Requests and Parse for Results

Google hosts a public PHP page that allows users to get directions by entering arbitrary postal addresses or latitude/longitude coordinates.  Since this page does not require any special key to access it, we can circumvent the query rate limits imposed on API key holders by making route requests to this public page.  This would require the controller library to make HTTP requests to the URL http://maps.google.com/maps and parse the HTML file for the route information.

For example, if the optimizer queries the controller library for the driving time from Austin, TX to San Antonio, TX, the library would first construct and issue an HTTP request to http://maps.google.com/maps?saddr=Austin,TX%20to:%20San%20Antonio,%20TX and then scrape, or parse, the page for the route's estimated travel time.  Although this approach may work for a couple of queries, Google will more than likely ban this type of behavior since it does not resemble typical human browsing habits [11].  After performing some tests, we verified this hypothesis after being blocked by GoogleMaps after about 350 automated requests in a short time period.

Approach 3:  Reverse Engineer a Map Service

Many of the popular map services provide only a web interface for accessing their API, so using these interfaces in a C/C++ library may prove to be problematic.  Modern browsers such as Firefox and Internet Explorer include JavaScript (JS) engines that execute JS code associated with a map service's API.  The JS then makes HTTP requests to a system hosted by the map service and subsequently creates JS objects that the client can then use.  Since the controller library does not contain a JS engine, it will be unable to directly use the APIs provided by map services.  Instead, we could reverse engineer the JS of the map service we've chosen and write a PHP or C/C++ interface for our library.  This solution is problematic however, as it could be rendered useless if the map service's API changes even slightly.  Although the GoogleMaps API has been reverse engineered by some end-users, we were unable to find a library that included routing capabilities [12].

Approach 4:  Use MapQuests' C++ Library

MapQuest's Advantage API provides developers with the largest tool set of all the map services we explored, allowing access to the MapQuest platform through Java, C++, .NET, and JavaScript libraries.  The route component of the controller library could wrap MapQuest's library into a usable interface for the problem reduction and optimizer components.  In addition, the MapQuest Advantage API does not have any hard querying limits imposed by their data server.

We chose the MapQuest platform for the geocoding queries since it had directions support and an ubiquitous API.  Furthermore, we decided to use a hierarchy file structure with a code base in C++.  We chose C++ for its object oriented features and its efficiency over interpreted languages, such as Java or PHP.

**Optimizer**

In order to generate valid and efficient matches for our users, i.e. solve our variation of the VRP, we must find algorithms proven to do so for similar problems.  During the past few months, we have evaluated several solutions by reading academic papers, referencing web pages that aggregate research on the VRP, and consulting with our advisor, Dr. Caramanis.  We found that a variety of approaches have been used for solving variations of the VRP.  Among these were exact, heuristics, and metaheuristics approaches.  Heuristics are a method of finding good solutions using educated guesses, intuitive judgments or rules of thumb [13].  Metaheuristics are higher level heuristic methods, such as evolutionary, linear programs, simulated annealing, tabu search, and ant colony algorithms [13].

Brute Force

The exact, or brute force, method has the advantage of producing *the* best possible solution, but is only applicable to very small problems, on the order of a few dozen drivers and riders.  Anything larger would be unsolvable because it would take too long to solve, since the time complexity grows exponentially with this method.  We plan on implementing the brute force method in order to compare with the other algorithms, and gain an understanding of their performance.

Evolutionary

Of the methods we researched, evolutionary approaches seemed to offer reduced development complexity, along with very good solutions.  In addition, this method constitutes a more adaptable approach to solving the problem, allowing us to easily migrate the methods of others to our variation of the VRP.  One form of evolutionary algorithm is the genetic algorithm.  The genetic algorithm attempts to implement the theory of natural selection. To elaborate simply, it starts with a set of random solutions (random assignments of riders to drivers), ranks them according to a scoring function (sometimes called a fitness function), then mutates the best solutions hoping to improve them (reassigns some riders to other drivers).  When solutions cease to improve as a result of mutation, they are either discarded, or spliced together with other solutions to hopefully produce a better solution.  A key point of this algorithm is that at any given time we have access to its best solution so far, or in other words, we can stop running the algorithm after any arbitrary duration and have an approximation of the best solution.  A side effect of this characteristic is that the approximation's accuracy is relative to the amount of time that we let it run: the longer an evolutionary algorithm runs, the more accurate the results.

Linear Programming

Another possible algorithm would be to represent the problem as a system of linear equations and use or write a solver for it.  Like the brute-force method, this method would produce the best possible solution.  However, also like the brute-force method, it is not guaranteed that it would finish in a reasonable amount of time, nor that we would be able to stop it and produce a "best-so-far" result like we can with the genetic algorithm. In addition, none of our team members currently possess the knowledge required to implement a linear program, so this method would require us to spend more development time researching.

Iterative Bipartite Matching

In addition to these methods, we have formulated our own approach with Dr. Caramanis's assistance by observing that if we treat our variation of the VRP as a bipartite graph matching problem, we can use simpler solutions for maximal bipartite graph matching.  Since bipartite matching algorithms match one driver per rider, we can iterate the process, adding a rider during each iteration.  We call this method Iterative Bipartite Matching.  Please see Appendix B for more information on bipartite graphs and maximal matching.

Tabu Search

We also plan on employing Tabu Search as a metaheuristic for all of our algorithms.  Tabu Search excludes the use of matches previously found to be invalid.  For example, if a driver-rider match doesn't  fit the criteria for route and time constraints, then we will never try to pair that driver with that rider again.  This method allows for a more efficient search of the solution space, since knowingly wrong solutions are excluded.  In addition, it can be implemented once, and used by all of the above mentioned algorithms.

As the only true test of algorithms is to run them and compare performance, we have decided to implement each of the algorithms mentioned above, except the linear program due to time constraints.  We plan on using the scoring function we develop for the genetic algorithm to grade the output of each algorithm.  To aide in analysis, we will also develop real-time displays of the algorithms so we can visualize their progress.  The implementation of this visualizer will be specific to each algorithm as it will depend on the internal data structures of each.  It will have to be developed in conjunction with each algorithm.

**INFORMATION SOURCES**

**Framework**

Information regarding the operating systems that each database solution and what web servers each web toolkit ran on were available on each of the products' websites.  We also had to establish whether each web toolkit could connect to the databases.  This information was found by searching each toolkit's website for information regarding each database in turn.  As the data is from the toolkit's website, it is both accurate and credible.  Because it turned out that all the toolkits had an interface to all the database solutions, it turns out that this information was not useful to us in making framework related decisions, however, it was still critical information regarding the project as that interface must be present.  The dependence chain of PHP directly affected our choice of web server platform, which influenced our choice on operating system.

Since some of us are working with unfamiliar programming languages, we may run into syntactical or conceptual problems and therefore might need to find information on them.  For PHP, we can reference their official website, which contains useful documentation and frequently asked questions [14]. The MySQL developer's page has support from the open source community, so documentation and help is freely available [15]. If we need help with the Google Maps API, we will use their documentation in addition to their developer forums [16]. MapQuest's Technical Resource Center should also help us understand their API [17].  In addition, we have access to the UT libraries, which offer several online resources, including programming manuals.  As each of these websites are run by the creators of their respective subjects, they should prove very reliable.  We have researched some of the technical abilities of the software choices from PHP, MySQL and Google websites, and this has helped us make informed design decisions, in that we know the project is feasible.

**Optimization**

To learn about the VRP, we have referenced several academic papers and websites. Google Scholar and IEEE Xplore are both search engines designed specifically for searching academic papers [18][19]. These have proved particularly useful for algorithm research, though we have found more specific papers by browsing through "The VRP Web"and "WebVRP", which are devoted to the vehicle routing problem [20].  These two resources have provided us with a breadth of reliable information, since they are the work of academic experts on the VRP, and are regularly updated with new approaches and results.  Academic articles are reliable by their nature, since they go through the process of peer review.  On the other hand, these resources can give us only an idea of the feasibility, since we will have to tailor the approaches to fit our own needs.  To a lesser extent, we have used Wikipedia to search for general information on any algorithms or tools we may use [21].  It has provided us with links to such resources as we have mentioned above, and helped us direct our research.

Dr. Constantine Caramanis helped us determine our algorithmic approaches.  We have been holding weekly meetings with topics ranging from high level choice of algorithms to low level pseudo-code.  Dr. Caramanis is very experienced with optimization algorithms so we feel he has been of great assistance.  He suggested we try several different optimization algorithms and evaluate the solutions as well as encouraged us to use a problem reduction algorithm to create several instantiations of the optimizer function to return fast results.  He has emphasized the importance of creating a high quality fitness function to determine which matches are best.

CheolHee Park is our technical TA. He is currently a student of the WNCG here at UT, working on his Ph.D. His research areas include wireless communications and networking. Also, he is skilled in algorithm analysis and C++.  CheolHee has helped us organize our project, define goals, and make sure our complexity level is appropriate.

**Project flow diagram**

The following project flow diagrams show the order and dependencies.  Each task has been well documented above, in the form of selected alternatives.  Note, however, that some tasks must be completed before others; we must design the module interfaces before we can set up the framework, and so on.  Within the optimizer, we must complete the problem reducer and the tabu search before we can work on the algorithms more than conceptually.  Additionally, when we are working on the framework, we may find that our module interfaces are impractical or overly complicated, at which point we can go back and improve upon them; the same is true with testing and setting up the framework.

Please see Appendix C: Project Flow Diagrams for module-level flow.

**Figure 6:** Overall Project Flow

**Conclusion**

In this report, we have presented our idea for a novel and promising web application which facilitates carpooling.  We believe that our product will be a friendly way of making carpooling much easier to coordinate, and encourage more people to use this mode of transportation.  We have introduced our plan for developing Rideshare, along with the resources and time table.

The underlying problem we are solving is categorized as NP-Complete.  This may be the biggest challenge of the project, and so we plan to implement several methods for performing the optimization, and assess the performance of each.  The user interface is also a challenging aspect of the project, since we are including an interactive map, as well as traditional forms.

In order to complete the application in two months, we have broken down the development tasks into four modules which are minimally interdependent, and assigned them to team members according to their skills and interests.  The actual tasks have been organized so that we complete the features which are common to several modules first.

With eight weeks remaining till the end of the development and testing process, we believe we now have the knowledge and tools necessary to implement the Rideshare web application.  The world wide web offers endless documentation covering the programming languages, APIs, and platforms we have decided to use.  In addition, we also have people resources, namely Dr. Caramanis and ChoeLee Park, that will provide guidance and feedback during the implementation stage of this project.  Project management has been carefully been delegated so that web site will be constructed in a timely and organized fashion.  We firmly believe the talent and ingenuity of all team members will play a role in the success of Rideshare and look forward to sharing and demoing the final result .

**References**

[1] Paul E. Black, "Vehicle Routing Problem", Dictionary of Algorithms and Data Structures, http://www.nist.gov/dads/HTML/vehicleRouting.html (Current 17 Dec. 2004), Cited 26 January 2008.

[2] Wikipedia contributors, "Vehicle Routing Problem," Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/w/index.php?title=Vehicle\_routing\_problem&oldid=178307741 (Current 16 Dec. 2007), Cited 27 Jan. 2008.

[3] Monaldo Mastrolilli, "Vehicle Routing Problems," http://www.idsia.ch/~monaldo/vrp.html (Current 8 June 2001), Cited 2008 Jan. 28.

[4] Bernabe Dorronsoro Diaz, "The VRP Web", http://osiris.tuwien.ac.at/~wgarn/VehicleRouting/vehicle\_routing.html , (Current March, 2007), Cited 13 Feb. 2008

[5] Wikipedia contributors, "User Interface Design," Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/wiki/User\_interface\_design, Cited 10 Feb. 2008.

[6] MySQL Development Team, "MySQL 5.0 Reference Manual," MySQL, http://dev.mysql.com/doc/refman/5.0/en/c.html, Cited 12 Feb. 2008

[7] cURL Development Team, "cURL - Frequently Asked Questions," cURL, http://curl.haxx.se/docs/faq.html, Cited 12 Feb.2008

[8] "Google Maps API," http://code.google.com/apis/maps/index.html (Current 2008), Cited 11 Feb. 2008.

[9] "Yahoo! Maps Web Services," http://developer.yahoo.com/maps/ (Current 2008), Cited 11 Feb. 2008.

[10] SWIG Development Team, "Simplified Wrapper and Interface Generator," SWIG, http://www.swig.org, Cited 12 Feb. 2008

[11] Wikipedia contributors, "Web Scraping," Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/wiki/Web\_scraping, Cited 12 Feb. 2008

[12] Wikipedia contributors, "Google Maps," Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/wiki/Google\_maps, Cited 12 Feb. 2008

[13] Wikipedia contributors, "Heuristic," *Wikipedia, The Free Encyclopedia,* http://en.wikipedia.org/w/index.php?title=Heuristic&oldid=188687945 (Current 13 Feb 2008), Cited 13 Feb. 2008.

[14] "PHP: Hypertext Preprocessor," http://www.php.net/ (Current 2008), Cited 29 Jan. 2008.

[15] "MySQL Developer Zone," http://dev.mysql.com/ (Current 2008), Cited 29 Jan. 2008.

[16] "Google Code," http://code.google.com/ (Current 2008), Cited 29 Jan. 2008.

[17] "MapQuest Technical Resource Center", https://trc.mapquest.com/ (Current 2008), Cited 11 Feb. 2008.

[18] "Google Scholar," http://scholar.google.com/ (Current 2008), Cited 2008 January 29.

[19] "IEEE Xplore," http://ieeexplore.ieee.org/Xplore/dynhome.jsp (Current 2007), Cited 2008 January 29.

[20] Wolfgang Garn, "WebVRP", http://neo.lcc.uma.es/radi-aeb/WebVRP/, (Current July 2007), Cited 13 Feb. 2008

[21] Wikipedia Contributors, "Wikipedia," Wikipedia, the Free Encyclopedia, http://en.wikipedia.org/wiki/Main\_Page (Current 29 Jan. 2008), Cited 2008 January 29.

**Appendix A:**

**Data flow diagram**

Invalid Data

User Authentication (Login) Page

Password Authentication

About Rideshare Page

Authentication Failure

New Account Page

My Routes Page

Data Validation

Data Submission

New Route Page

My Profile Page

Validation Email

Email Address Validated

Email Received

Database

Problem Reduction

Optimizer

Optimizer

Optimizer

Optimizer

Controller Library

**Appendix B:**

**Graph Theory**

Some understanding of elementary graph theory is necessary in order to solve our optimization problem. We include this section to assist the reader in understanding the optimization problem at hand, and thus gain an understanding of the optimizer module’s required functionalities. Mathematical definitions are marked with bullets.

* A *graph* is defined as a set of *vertices* V={v1,v2,…,vn} and a set of *edges* E={e1,e2,…,em}, for some positive integers m and n.
* The vertices can be thought of as nodes, and the edges as lines connecting those nodes, as shown in Figure A1

Our optimization problem can be viewed as a graph theory problem. In our case, the drivers (D) and riders (R) can be viewed as vertices, where two vertices are connected by an edge if they can be partners in a rideshare.

Figure A1: Bipartite graph of drivers and riders

* A *matching* is a set of vertices and edges. For a matching M, two vertices are said to be matched if M contains an edge which connects them.

Our optimization problem can be viewed as equivalent to the maximal matching problem. The maximal matching problem is the problem of finding a matching within a graph which includes the maximum number of vertices. Translated to rideshare, this means finding the most matches between drivers and riders, and thus providing rideshares for as many people as possible.

In particular, our optimization problem can be viewed as equivalent to a bipartite graph optimization problem.

A *bipartite graph* is a graph in which the vertices of the graph can be divided into two disjoint sets, where no two vertices in the same set have an edge between them.

In our case, we have two sets of vertices, D (drivers) and R (riders), which can never have rideshares within the sets, i.e. no two drivers will be in a rideshare together, and the same holds for riders. This relationship can be seen in figure 7.3.1, where no edges are visible within the sets D and R.

**Appendix C:**

**Project Flow diagrams**

**Figure C1:** Framework Flow

**Figure C2:** User Interface Flow

**Figure C3:** Optimizer Flow

**Figure C4:** Testing Flow