-LCA
MapGroup/URoute
1/31/2006

Operational Concepts

URoute is an extendible application designed to provide optimal directions between two points. Simply, it is an application to provide driving directions around Seattle, with the ability to be expanded by altering its maps and routing preferences. To demonstrate this extensibility, the project will offer two plug-ins to start: a traffic add-on that adjusts driving directions based upon live traffic data, and a bicycle add-on that creates directions specific to bikers' needs.

Major Benefit:
URoute offers extensibility not seen in current routing systems (Mapquest, Mappoint, Yahoo Maps, etc.) that allows the user to customize their concept of an ideal route beyond the fastest in ideal conditions. For example, maps generated by for cars do not always select the best possible routes for bikers. Alternatively, some users prefer driving on some highways more than others (such as I-90 instead of SR-520). A plug-in to URoute can easily address these use cases. The plug-in engine allows users to modifying routing criteria beyond "fastest, or least transfers" into the realm of "best suited" for a user's needs.

What it can do:
The initial URoute system will offer driving directions around Seattle, with the option to extend the maps to non-Seattle areas (or more detailed Seattle maps than just roads), as well as changing preferences on road routes (for traffic, personal preferences, construction projects, etc.). To highlight this extensibility the project will offer two plug-ins, one for adding traffic logic to driving directions, and a second for creating biking directions around Seattle.

What it can't do
One key to success with URoute is having a simple, easy to use directions product. In order to achieve this, it is important to carefully refine the scope of the project. Version one will not include the following features: reverse geocoding (existing applications can be better adopted for these needs), historical traffic information, live traffic from intersection cameras (a computer vision problem to be solved for version two), adaptation of content to fit mobile displays, maps beyond the greater Seattle area, and integration with car GPS systems.

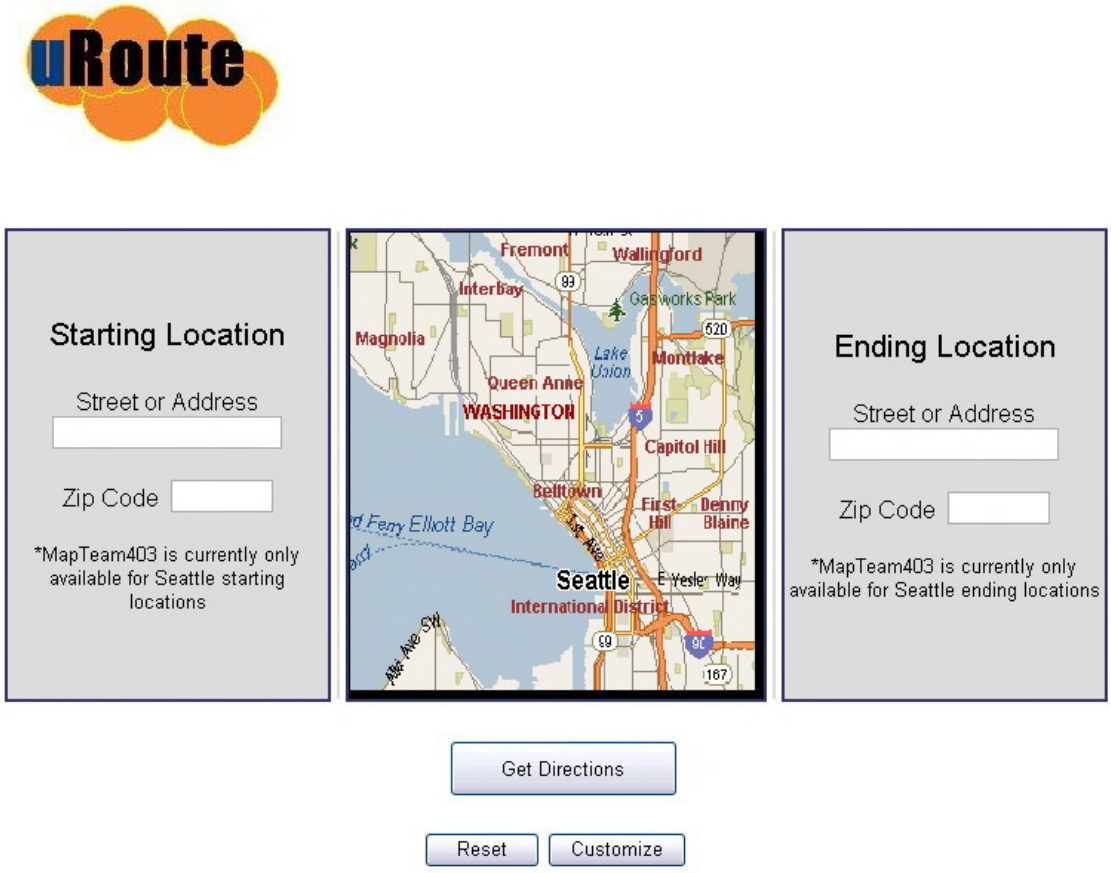
User Community
URoute is aims at all of the Internet public, with varying ranges of technological fluency from almost none to that of software developers. Anyone looking to drive to a new location in Seattle, or someone unfamiliar with Seattle will find this project very useful. Moreover, bikers in Seattle will find the bicycling plug-in a useful tool for planning bike trips and commutes. Finally, the Department of Transit is another key user group that will

hopefully feed data into the system and use its directions report to help consider alternative routes taken by drivers.

Program Environment
For easy of use, URoute will be available on the Internet (via Microsoft IIS server). This way it can easily be ported to mobile devices and converted to run as a web service in the future.

Functional Specifications

GUI
Most importantly, the GUI needs to be simple and easy to use. While the routing. The user interface is a website with an easy to remember URL so that it is quickly accessible to the general public. The main page should be as plain as possible to facilitate a quick and easy route search. It will have a simple white background with minimal extra text. The goal is to get the user their directions as quickly as possible so they don't have to click through page after page before getting their directions, only to realize they need to weave back through multiple pages to modify their search. The goal is to keep the interface as visually simple and easy to understand as possible.



There are two ways in which the user may specify the locations they want directions for. The first is that the user types in the address for both a starting location and an ending location. In real time, dots will appear on the map of Seattle corresponding to those locations and a route will be drawn between them in real time. The second is that they click on the map of Seattle and the addresses for the locations they clicked will be populated in the starting and ending location inputs. The user then clicks the “Get Directions” button that will display textual directions for traveling the route specified on the map.

If the customer is satisfied, they can print their directions that include both the map of the route and the written directions, or they can further specify their search by clicking the “Customize” button. This will take them to another screen that is still displaying their route. From the map on this page, they can draw boxes on the map of areas they don’t wish to drive through, or select from other criteria such as a “No Interstates” option that would draw routes that do not include and freeway edges. As on the first page, the route will appear in real time as the criteria changes. When they are satisfied with their route, they can click “Get Directions” just like they did with the original query on the main page.

uRoute

Starting Location

Street or Address
1010 18th AVE NE

Zip Code 98105

*MapTeam403 is currently only available for Seattle starting locations

Ending Location

Street or Address
Seattle

Zip Code

*MapTeam403 is currently only available for Seattle ending locations

Get Directions

Reset Customize

Depart **SOUTH** on 18th Ave NE

1. Turn **RIGHT** on NE 45th St
2. Turn **LEFT** onto 5th Ave NE
3. Turn **LEFT** onto ramp to I-5
4. At exit 165A, turn **RIGHT** onto 6th Ave towards James St
5. Turn **RIGHT** onto James St

Core Engine
Simply, the core engine provides an efficient routing for determining the shortest path between two points (via A* algorithm). It is fed graph data from the plug-in engine (which modifies graph edges as needed) and outputs its search results to the webpage UI.

Plug-In Engine
Plug-ins will have the ability to modify or add nodes to the graph structure as well as change how edges are weighted. Plug-ins connect through a plug-in handler which talks directly with the database. Each individual plug-in creates a database table containing the new edge waits relative to its scenario. Depending on the individual plug-in this table will be created once (bike plug-in), or updated at regular intervals (every 10 minutes for traffic).

Sample Plug-Ins (to be included with core engine)

Bicycle Routing
This plug-in improves basic routing by providing routes specific to bikers' needs in Seattle. Criteria such as average traffic, number of hills or average grade, and one-way streets will be considered when routing bikers on surface roads. Moreover, the plug-in will add bike paths to the underlying graph of Seattle to more intelligently route bikers.

The designed user group for this plug-in is bikers in Seattle, primarily those commuting to work every day and looking for the most optimal route. Secondly, bikers looking for a new way to explore the city or a weekend ride would find this URoute addition useful.

> **Sample Use Case 1**
> Mr. Speedy Wheels wants to bike to work every day, but is afraid to take the major roads he usually drives upon. Not knowing the city well enough to pick a safe and fast bike route, he logs onto URoute and finds a customized bike route, picking one-way streets so he doesn't have as much traffic to watch out for (no one turning in front of him), and avoiding unnecessary hills in the process.
>
> **Sample Use Case 2**
> Lazy Sally lives by University Village and wants to bike to the University of Washington for class, but can't stand going up the hill straight onto campus. She logs onto URoute and finds a customized bike route with a smaller minimum grade by heading farther north and cruising downhill towards the UW once she reaches 65^{th} and 17^{th}.

Traffic
The traffic plug-in will analyze live traffic data in order to alter edge weights on the underlying graph. This provides a feature not yet seen – driving directions that take into account traffic flow. Traffic data will come from three sources – incident reports submitted on URoute by traffic professionals, traffic incidents pulled off MapPoint, and flow information from the Department of Transit (DOT) highway loops.

The traffic plug-in will keep an updated table of edge weights, updated every 10 minutes with new traffic data.

Sample Use Case 1
It's 4:30pm and Lucky Joe just got out of work early in Downtown Seattle. He doesn't know the usual traffic at this hour and wants to get home to Bellevue the quickest. He checks his path on URoute and finds out that taking SR-520 is preferable to his typical I-90 route.

Sample Use Case 2
Suzie is heading to the Divisional Playoff game for the Seahawks and wants to know the quickest way there, which given the road closures and changes for the game, is not an easy thing to figure out. URoute however, provides an optimal solution around all road delays.

Desired User Experience
A few key reliability issues are important to ensure a positive user experience. The site needs to provide directions within a reasonable time-frame (5-10seconds max), and have close to 24/7 uptime. Most importantly, it needs to be very easy to use so it is accessible to people with varying levels of technological fluency.

Technical Specifications and System Architecture

Coding for URoute will be done in C#, due to the use of the Mappoint API. To store the graph data, URoute will interface with Microsoft's SQL Server 2005. MapPoint will be used as a web service for map functionality (drawing routes on maps, showing locations, picking start and stop points, zooming in and out, etc.) In more detail:

Core Routing Algorithm
There are several algorithms to consider when doing things like route finding which involve finding a shortest path along a graph. A* search will be the preferred search algorithm to find the shortest route path for this project based on its ability to handle large graphs and find the shortest path fast. Dijkstra's Algorithm would have also been a consideration but it will not run as fast with large complicated graphs due to the nature of the algorithm searching the nodes next to it before going deeper on the graph. A* will guarantee us a shortest path as long as the heuristic function h(n) is admissible; a simple admissible heuristic to start with is the distance from a given node n to the final destination. This could be simply implemented by latitude and longitude information extracted from the involved nodes. This is an admissible heuristic as it is an underestimate. The only case where this heuristic would be equal to the actual cost if it is the last road to the final destination and there are no other options. A more complicated heuristic can be determined at a later time if it is found to be necessary. It is important to note that there are thus 2 heuristics that will be used in the whole system, one will be

provided by the plug-in to determine the cost of the edges based on whatever factors the plug-in finds necessary and one will be used by the A* algorithm. The first mentioned heuristic is going to be the more important one, as it will be what is actually used for g(n) in the A* algorithm to determine what route is actually taken.

The question has arisen as to how the graph should be implemented when it is searching, if it should be stored in the database and should be queried at or if it should be stored in memory while doing the search. The latter is obviously the faster but is bounded by the size of the map. In the end, there is not that much data that is being stored for every node, even though it may seem like a lot, so quite a large map can be placed in memory. Looking at other map sites who offer limited features other than just driving directions like Mapquest have also addressed this problem. Mapquest allows one to avoid major highways when planning a route. According to their FAQ online:

> *Can I view directions that avoid highways? MapQuest uses a sophisticated routing algorithm to determine driving directions. But sometimes - for various reasons - MapQuest users may want to choose a different route. We now offer the option to generate a route that avoids major highways whenever possible. For practical reasons, this option is available for routes of less than 50 miles only. (http://www.mapquest.com/features/main.adp?page=help_faq)*

Plug-in Engine

Plug-ins for the map system work by modifying the data model underneath the core search algorithm. They must be able to view the data model, possibly in reasonably-sized chunks, for inspection. For security reasons however, they will not be able to change the underlying model. Plug-ins will be able to add one additional table, storing extra information on edge weights (plus auxiliary information if necessary), but this will not affect the underlying graph data.

Before the core engine receives a section of graph upon which to search, the plug-in engine will filter it by readjusting edge weights as necessary.

We will also use a system of regions to specify geographical areas of the model - this will allow small updates to take place without having to access the entire model. A region is defined as all nodes that are listed as belonging to that region; from this it follows that all edges connected to those nodes are part of that region also. This means that some edges will belong to more than one region.

The XML Schema that the plug-ins need to abide to when sending updates to the database through the Plug-in Interface is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<nodeCollection>
  <node ID="??" location="???" region="???" moreStuff="???">
        <edge ID="???" endNode="endNodeID" weight="???"
      moreWeights="???" type="pluginType"/>
```

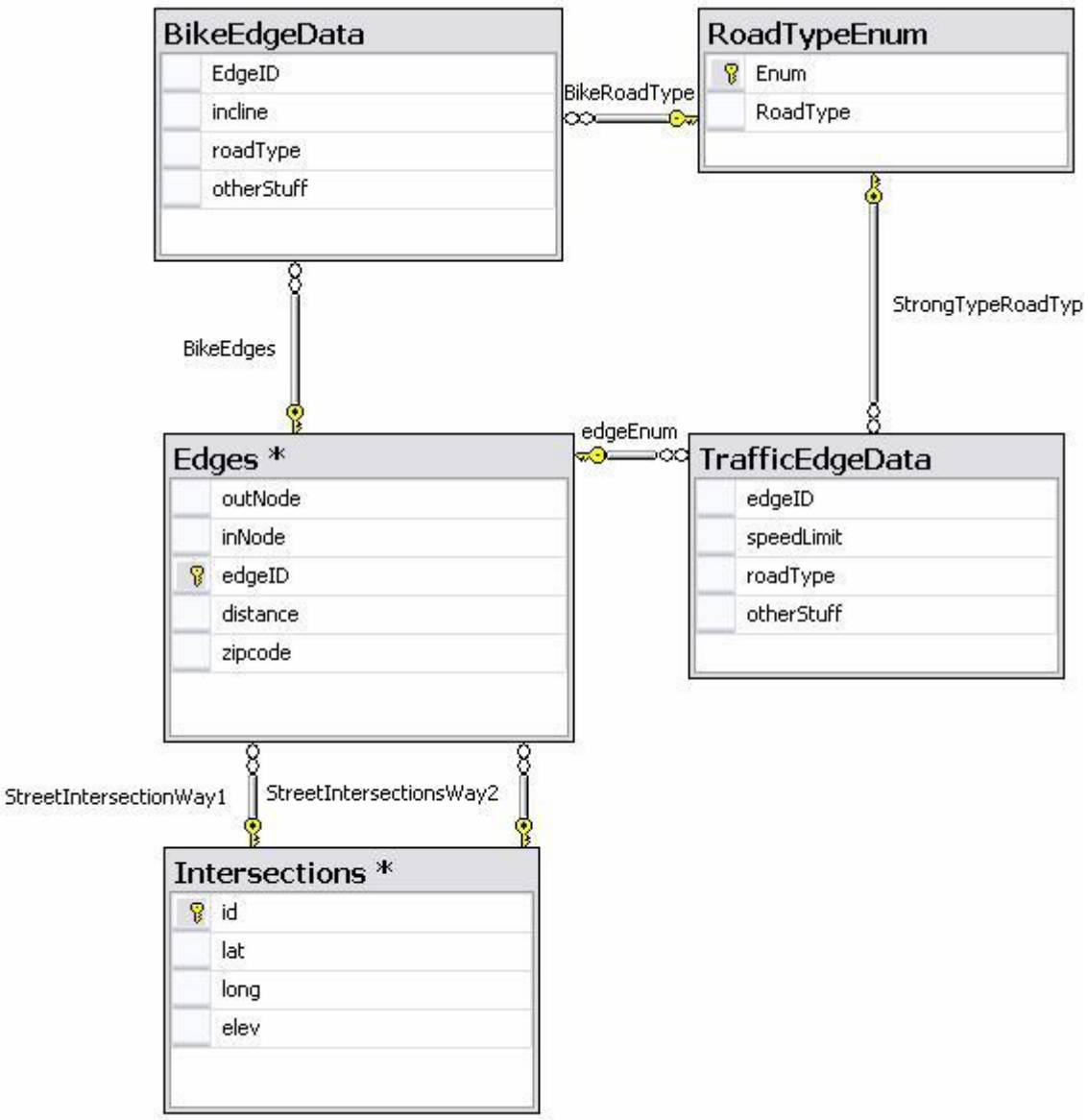
```
        <edge ID="???" endNode="endNodeID" weight="???"
       moreWeights="???" type="pluginType"/>
        <edge ID="???" endNode="endNodeID" weight="???"
       moreWeights="???" type="pluginType"/>
        <edge ID="???" endNode="endNodeID" weight="???"
       moreWeights="???" type="pluginType"/>
        <edge ID="???" endNode="endNodeID" weight="???"
       moreWeights="???" type="pluginType"/>
        ...
  </node>
</nodeCollection>
```

Below are the main functions' signatures:

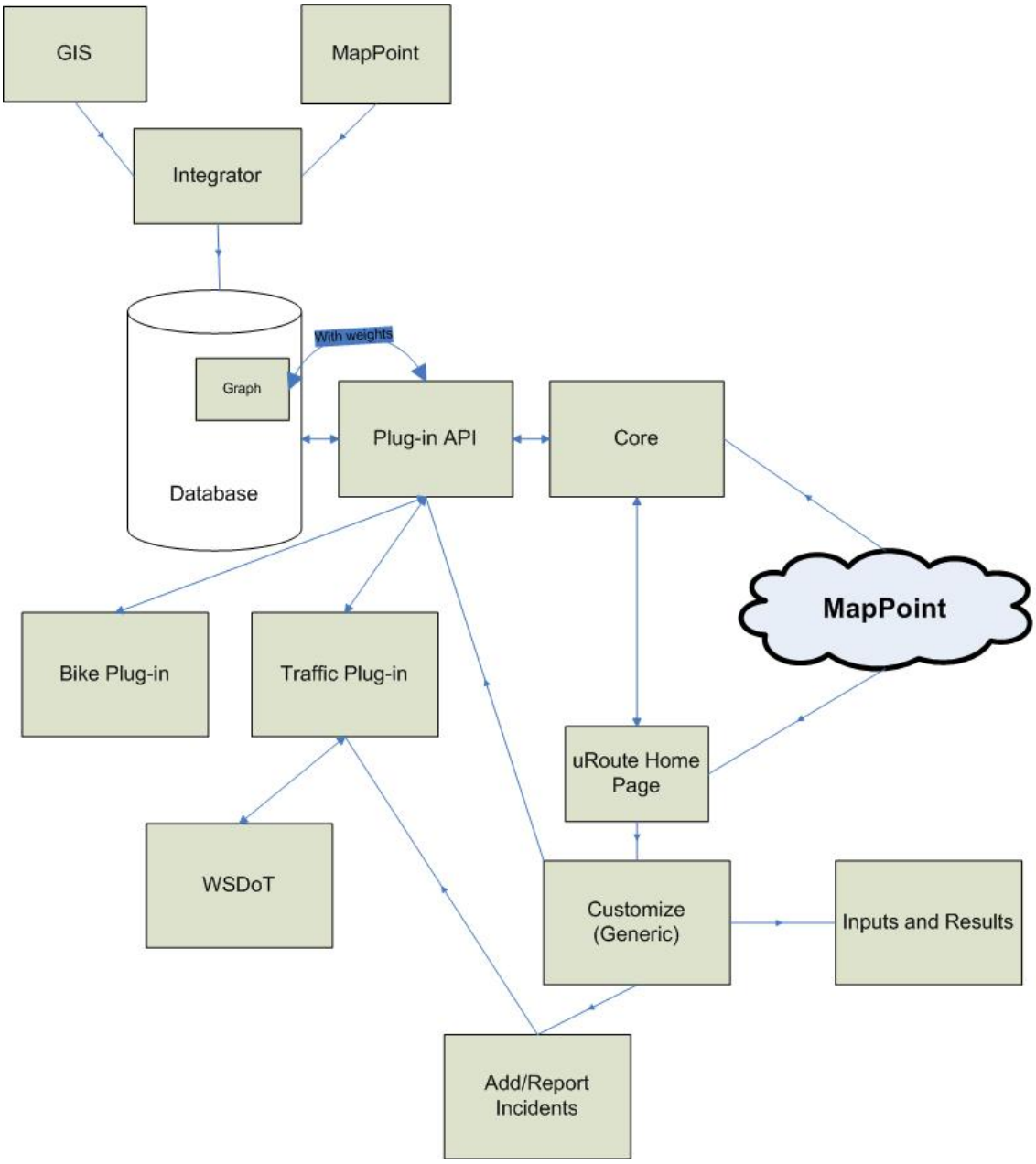
```
/**
 *
 * The regionName parameter will tell the method how much data to
return. The model will have each node associated with a hierarchical
set of regions, for example, a node in the U District would belong to
the U District region, which belongs to the North Seattle region, and
so on. The returned data is a (potentially very large) mapSection
object that represents all the nodes and edges for that region.
 */
 XmlMapSection GetData(string regionName);

/**
 * The regionName parameter tells the model which region the new
mapSection is replacing, or the name of a region that is receiving data
for the first time.  If the regionName parameter is not recognized by
the system, false will be returned. The newData parameter is a
mapSection object containing all the node and edge data for a region.
 */
 bool SubmitUpdate(string regionName, XmlMapSection newData);

/**
 * The regionName specifies the name of the new regions - if the name
is invalid, or is already taken, false will be returned. The
parentRegion parameter specifies which region this region is a part of.
The newData parameter is optional, and would contain the data for the
new region.
 */
 bool AddRegion(string regionName, string parentRegion);
 bool AddRegion(string regionName, string parentRegion, XmlMapSection
newData);
```

What is critically missing here is a precise description of the XmlMapSection data object. This will be derived directly from the database schema (see below).

Database Setup



Flow Diagram:



As seen above, the database is fed with data from GIS to create our own graph. This GIS data will be integrated with MapPoint data (primarily through GPS translation) to facilitate use of MapPoint display (drawing routes, etc). Once the graph is stored in the database, the core will access it via the plug-in engine to perform A* fastest path searches. The plug-in engine functions by grabbing the core graph from the database and running a filter on top based on which plug-in is enabled. The plug-in engine can reweigh the edges based upon its database tables, or on the fly (such as when the user selects path preferences), as necessary. After the search is done, results are passed to the MapPoint API and the results webpage, which display to the end user.

In the background, the plug-ins maintain tables of reweighed edges in the database, updating them as needed (every 10 minutes for the traffic plug-in, pulling from DOT and MapPoint)

The user fits into the diagram towards the bottom, just surfing between the Home Page and results pages. Alternatively s/he can add preferences to her/his search (selecting a plug-in and/or its options such as biking with lowest overall grade).

Lifecycle Plan

Objectives

No open-source routing system currently exists. All current providers of driving direction are proprietary and although sometimes feature-rich, non-extendable. Extensibility provides routing benefits to domains not traditionally served (bikers, commuters looking for a better route around traffic, hikers, etc.). Moreover, by providing an open-source platform for innovation URoute will most likely attract additional intriguing routing ideas from third-party developers.

In terms of development, we are taking the spiral model of analyzing risks, prototyping to resolve these risks, and then re-evaluating. The first spiral, which we are currently in, involves the definition of specifications, modules, and interfaces between them. The second spiral will be an implementation of the core system and one simple plug-in (bike) without live data updates. For the third and final spiral we will add full UI functionality and both plug-ins.

Rough Schedule

Week 1 (end Spiral 1): LCA Due Tuesday, present Wednesday. Finish specifications, server setup, database setup, and interface code by Sunday.

Week 2: Coding of Search Algorithm, Map data import, Traffic data retrieval, basic UI done, bike plug-in.

Week 3 (end of Spiral 2): Debug to Beta 1 from week 2

Week 4: Integration of traffic Plug-in (with live data), full UI done with MapPoint

Week 5: Beta 2 – Working as fully as possible

Week 6 (end of Spiral 3): Debug to Final

Responsibilities

Feature	Task	Owner	Orig Est	Cur Est	Elapsed	Remaining
Bike	Determination of algorithm	Michael	10	10		10
Bike	Integration with user input (preferred (	Michael	5	5		5
Bike	Data collection of new routes (Burke (	Michael	5	5		5
Core	Interface Define and Code structures	Jonathon, Uday	15	8	0	8
Core	Interface Documentation	Nick	10	10		10
Core	Search algorithm	Nick	6	6		6
Core	Debug	Jonathon, Uday	40	40		40
Core	Integration	Jonathon, Uday	10	10		10
DB	Creation of Database	Karl, Elizabeth, Yegor	5	5		5
DB	Map Data Import	Karl, Elizabeth, Yegor	15	15		15
DB	Import of Traffic Loop data	Karl, Elizabeth, Yegor	10	10		10
DB	Helper functions to provide easy DB a	Karl	5	5		5
DB	Debug	Karl	20	20		20
MapPoint	Click to pick start point for directions	Elizabeth, Nick	4	4	2	2
MapPoint	Draw route on map	Elizabeth, Nick	12	12		12
MapPoint	Debug	Elizabeth, Nick	20	20		20
Traffic	Interpret traffic flow data into reweight	Yegor, Pedro	2	2		2
Traffic	Integrate algorithm with live data	Yegor, Pedro	10	10		10
Traffic	Debug	Yegor, Pedro	20	20		20
UI	Home Page	Carolyn	5	5		5
UI	Traffic submit incident page	Carolyn	5	5		5
UI	Additional options/refine page	Carolyn	5	5		5
UI	Debug	Carolyn	20	20		20
MapPoint	Converting from DB Graph to MapPoi	Elizabeth	15	15		15
Core	Documentation and Specs	Michael	15	15		15
Core	Build engine (setting up nightly builds	Uday	10	10		10
Buffer	Slippage time	All	20	0		0
					Total	290

Approach

Following the schedule outlined above, the project will split into teams, each designing one of the core components (plug-in engine, traffic plug-in, bike plug-in, core algorithm, database, and UI). Priority will first be placed on developing plug-in interfaces so teams can work in parallel on the plug-ins themselves and other parts of the project. Secondly, the bike plug-in will be done first as a proof of concept since it will be technically easier to manage, not needing to update the database on regular intervals as the traffic plug-in needs to do.

The lifecycle development points (corresponding with spirals) will be A) specifications and interface done, B) Core and first plug-in done, specifications confirmed by their use, and C) second plug-in and full UI functionality added.

Resources and Support Necessary

Underlying map data should be manageable to hold on one database server. Extending beyond the greater Seattle area may require additional resources, but a simple map of Seattle should be well contained within a single database server. Webserver load will also not be an issue. Using A* to find fastest path will limit the resources used to route find (by intelligently limiting search area).

For the scope of this project we will not have massive amounts of users that would demand hefty web server resources. Moving forward we can envision a need for 24/7

operations support and expandable server resources as the user base grows. Additionally, the DOT will need to manually update construction news and traffic indicants to support the traffic plug-in.

The project is dependent upon accurate GIS data for the underlying map, MapPoint for map rendering, and the DOT for accurate and reliable traffic data.

Feasibility

In terms of stakeholder acceptance, the need for driving directions is already a proven market. If we can improve upon the existing models, users will be satisfied. However, there is a question of whether our traffic results will be substantially better to warrant using URoute. With many major traffic incidents, any reasonable path may be slow. It is important to note that the system offers more than just traffic however, and the bike plug-in offers functionality not yet seen. Moreover, further extension from online developers would hopefully draw in new users, but there is the risk that URoute wouldn't attract as many third-party developers as we hope.

On the simplest level, creating a graph search algorithm is not terribly complex and should be attainable. The challenge for our project lies predominantly in properly connecting the different components of our system (core, db, plug-ins, and UI), both technologically and in terms of team organization. Moreover, there lies a serious challenge in meshing our GIS data (which will create the underlying graph) and the MapPoint graph nodes (to be used for drawing directions). If we fail to find a workable algorithm to link between the two map sources, we can develop our own visualization for the final results page. It won't be as alluring as the sleek MapPoint UI, but for the purposes of version one, it will do the trick. Most users are more concerned about the actual text directions instead of the fancy map any way.

A further data integration challenge will be for the traffic plug-in, where we need to approximate the traffic incident reports into a location on the graph. This seems plausible since the highway sensors each have a GPS location, but some specific incidents (accidents, construction, etc.) may be hard to pinpoint.

Another potential problem is performance. Both retrieving large sets of data from the database and running the A* algorithm could take longer than expected. With the data sets confined to Seattle however, all the graph nodes should be able to fit in main memory, each being relatively small, and search time should be manageable.

Despite these challenges, we believe that the project is indeed feasible. It isn't a stretch to assume stakeholder acceptance if we ship the product, and having estimated the total hours of various tasks at 30+ hrs per team member over the next few weeks, the project seems within our capacities. Even if the man hours necessary were to double, it still would be feasible, albeit not quite as enjoyable for our schedules.

Future extensions

Beyond version 1 of URoute, we can envision many future upgrades. The most obvious first step is a move to mobile devices so that users can get live traffic data on their handheld and reroute themselves before getting too heavily stuck in traffic. It is not a stretch to imagine that in 10 years most cars will be equipped with some sort of GPS system that could contain a URoute engine inside for proving up-to-the-minute driving directions.

More simply, version 2 of URoute would extend beyond Seattle and look at automatic ways to glean traffic and bike information. Could we design a UI to attract bikers to help add data to the system to improve efficiency? Are there automatic ways to grab traffic incidents (police traffic for example)?

Beyond driving, an interesting expansion of URoute would be into the national parks as a route-finder for hikers. Park Rangers could track and control which were given out in order to balance load on popular trails as well as learn which trails to maintain most frequently.