A GPS-Based On-Demand Shuttle Bus System

Maria Vanina Martinez Gerardo I. Simari Carlos D. Castillo Nir J. Peer

Department of Computer Science
University of Maryland College Park
College Park, MD 20742
{mvm,gisimari,carlos,peer}@cs.umd.edu

Abstract

We propose the implementation of an on-demand system to serve requests to ride shuttles on a university campus. Assuming that each client and each shuttle bus has a GPS device, we devise two online algorithms: the *k-means based* algorithm, and the *general pool-based* algorithm. The former has two components: the first deals with the problem of assigning requests to shuttles, which it does through the k-means algorithm, and the second one solves the scheduling problem for each bus. The latter does not perform a prior assignments of requests to shuttles.

We evaluate the effectiveness of the two methods by means of a simulation of both the system that is currently in place at the University of Maryland College Park and the two proposed on-demand systems. An empirical comparison of all models is then carried out, focusing on the total waiting time to serve a request. A more fine-grained analysis can be done by breaking down the waiting time into waiting time at the stop, walking time (to and from the stop), and ride time. Finally, we also study the fuel consumption of all systems. These preliminary tests show that, in some cases, the on-demand systems are capable of delivering lower values in all of these performance parameters.

1 Introduction and Motivation

The proliferation of relatively low cost and precise GPS devices in the last few years has allowed the development of systems that take as input the position of one or more objects on a map to be resolved quite elegantly. While this type of systems used to rely on less effective positioning techniques such as visual or radio reports, the problem can now be solved seemlessly by the use of pocket-sized wireless devices.

In the case of shuttle bus systems, GPS delivers the flexibility needed to switch from fixed routes to dynamically scheduled ones. The ideal scenario that we envision is the following: a student needs to get from his office to his house, so he pulls out his GPS-enabled PDA and opens the campus transportation application which displays a map of the campus. After creating a new request for service, the system prompts him to indicate where he would like to go by tapping the location on the map. After a brief waiting time, the user receives a message from the system telling him where he should walk to, which bus he must board, and where he must get off at. Note that in this scenario there are no fixed stops. Internally, the system has evaluated which of the currently available buses the student must take, depending on the requests that they are currently serving or will be serving in the near future. The focus of the system is to try and minimize total waiting time; in particular, we would also like to minimize the waiting time at the stop (this is desirable for many reasons, including weather, safety, etc), and walking time (consisting of the sum of time spent walking from the current location to the pickup stop, and from the delivery stop to the destination). Lastly, we are also interested in minimizing the total fuel consumption of the shuttle buses in service.

In the rest of this section we will describe the system that is currently in place, and some assumptions that we will make in order to make the above scenario a reality. In Section 2 we present the design considerations of the proposed on demand system. Then, in Section 3 we describe the k-means based

algorithm, followed by the despcription of the general pool-based algorithm in Section 4. We then describe our empirical evaluation in Section 5. In Section 6 we discuss related work. In Section 7 we briefly describe the overall system design and implementation details; finally, in Section 8 we present our conclusions and recommendations for future work.

1.1 The Current System

The University of Maryland College Park currently has in place a traditional bus system consisting of a number of fixed-path services that cover most campus, as well as some off campus points of interest. In this paper we are concerned with the evening shuttle system, which has five routes (Gold, Orange, Blue, Purple, and CP Metro Station) covering most campus locations, the surrounding student residential areas, and the College Park metro station. All lines have two shuttles running simultaneously, except for the Metro Station line which has only one. Frequency varies, but on average a bus comes every 15 to 20 minutes. Even though some routes are longer than others, the average is 20 to 25 stops, and a total round trip of 30 to 35 minutes. These values are only guidelines, we will study more precise performance values in Section 5, when we compare both systems in our simulator.

1.2 Assumptions

In order to make the proposed on-demand system a reality, we will have to make a series of assumptions, which we describe next. First, we assume that each bus has a GPS device precise enough to report its position inside campus (within a few meters). We also assume that patrons will place requests from GPS-enabled cell phones or PDAs, with the capability of displaying color graphics such as maps. In case such a device is not available, the system could also provide a web-based form through which patrons can place requests. We also assume that each bus has the capability to communicate with a central system. In this manner, all handling of requests and scheduling is done centrally, and then communicated to the corresponding buses.

Regarding road conditions, we assume that there is no variation in time in the routes due to traffic; this isn't particularly significant since we're simulating the evening routes which usually start running around 6PM (after the afternoon rush hour). Finally, for the on demand system we assume that if a given shuttle has no requests to service, then it can wait at any of the nodes for incoming requests.

2 Towards an On-Demand System

To test our ideas regarding an on-demand system, we developed a simulator. The simulator has two modes, one for the current shuttle system and one for the on demand shuttle system we're proposing. We send the same stream of uniformly randomly generated requests to the simulator in both modes. We generate a certain number of requests per unit of time. A unit of time represents a minute in the real world.

Even though it seems simple, programming the current system was not straight-forward. It involved mapping all of the stops on campus (where each stop is assigned a name, a pair of GPS coordinates, and a set of lines that stop there), figuring out the sequence of stops that each shuttle line has, and also the frequency with which each shuttle line visits a stop. The latter also involved figuring out how many buses each line has in service at a given time. All of this data was gathered and inferred by perusing the campus maps, along with the schedules provided by the University of Maryland Department of Transportation [6].

Once all of the data was gathered and represented in adequate data structures, the simulation of the current system proceeds along the following lines. For each bus that is active in the system, its next hop

is looked up in a static table. When it arrives at a stop (calculated by considering each individual line's frequency) it first drops off at that stop all the people on the shuttle that need to get off there, and then proceeds to pick up all the people waiting for that bus. The order in which this is done is important to ensure that the bus never exceeds its capacity. Lastly, in case there are more people at the stop than the remaining capacity for the bus, only the people who can fit on the bus are allowed to board, according to a queue to ensure first come-first served service.

The on-demand systems have a completely different structure. First of all, the routes followed by each bus are not governed by a line as before, but are calculated dynamically in relation to the requests that are pending in the system. Our first approach was to assign newly placed requests to the bus that is currently the closest one to the patron. Then, once the patron is on the bus, we used a simple greedy strategy with respect to waiting times to solve the problem. This strategy led to high waiting time (although as a byproduct also tended to minimize the waiting time at the stop), but allowed us to gain insight into the problem. For the k-means based algorithm we divided the problem into two parts, *assigning a shuttle to the patron*, and *scheduling the next request to handle*. On the other hand, the general pool-based algorithm does not make a prior assignment of shuttles to patrons, but places them in a global pool accessible to all shuttles. We describe the key foundational parts of these methods in the next sections.

2.1 Problem Formulation

Before discussing our proposed solutions, we present a formulation of the scheduling problem we're tackling:

- We are given a directed graph $G = (V, E)$ representing the campus roads,
- there is a set of n shuttle buses, each with capacity c,
- requests are placed dynamically, each one consisting of pickup and delivery GPS coordinates (x_p, y_p, x_d, y_d), and a timestamp indicating when it was placed,
- we must find an assignment of shuttle buses to service the requests such that the average time to service a request is minimized.

There are many problems similar to this one, both dynamic and static, that have been studied in the literature. We will review some of these problems, and their proposed solutions, in Section 6.

2.2 Fairness Considerations

In this section we will briefly discuss ways in which malicious users can take advantage of certain characteristics of the system to introduce behavior that is unfavorable for the rest of the patrons. Because the system is based on dynamic scheduling, there are quite a few ways in which this can be done. Even though these issues are not closely related to course topics, we considered them interesting and deserving a brief treatment. We will then present some of these possibilities, along with ways in which they can be prevented or discouraged:

- *Coordinated solicitation of service:* In this attack, a group of patrons coordinate their requests in such a way as to make the shuttle drive through certain parts of campus, with the objective of delaying it or simply making it run when it could be on standby.

  This scenario is difficult to prevent because this type of request is hard to differentiate from legitimate ones. However, in the case in which the patrons only call the shuttle to a pickup location and then

do not wait at the agreed upon location, the system can penalize the user with points indicating that he did not honor the agreement.

- *False placement of requests:* This attack is a particular case of the previous one, in which the patron gets on the bus but gets off at a different stop (thus making the bus drive farther for no reason).

  This case is easier to discourage because each patron can be required to "log out" of the bus before getting off, and the system can therefore check if the current stop is the one agreed upon during the placement of the request. The same points system described above can thus be applied.

- *Impersonation:* Requests may be placed by a patron claiming to be a different person. This type of attack can be eliminated by requiring the user to show his university identification when boarding the bus (not only swiping it).

Many other types of attacks can be guarded against by simply logging all activity and periodically running detection algorithms on these logs. For instance, we may wish to detect activity such as an unusual number of requests by the same user over a short period of time, or suspicious activity such as requesting service as soon as getting off the bus.

3 The k-means Based Algorithm

One of the best methods we found for assigning requests to shuttles is the k-means algorithm, which we describe next.

3.1 The k-means Clustering Algorithm

The k-means algorithm is a general algorithm that clusters objects based on attributes into k partitions. For each object we define n atributes and a comparison function (usually called d, for distance) that describes how similar two objects are. It is a variant of the expectation-maximization algorithm, in which the goal is to determine the k means of data generated from gaussian distributions. It assumes that the object attributes form a vector space. The objective of the algorithm is to minimize total intra-cluster variance, or the squared error function

$$V = \sum_{i=1}^{k} \sum_{X_j \in S_i} d(X_j, \mu_i)$$

where there are k clusters $S_j = 1, 2 \ldots, k$ and μ_i is the centroid point of all the points $X_j \in S_i$. The optimal μ_i is calculated for a set of points nearest to it (according to d) according to the optimal policy for that loss function. For example, if d is quadratic (the euclidean distance), it can be shown that the optimal policy is to calculate the mean of all the X_k that are the closest to that centroid. For other loss functions the optimal policy will be different.

The algorithm starts by partitioning the input points into k initial sets, either at random or using some heuristic data. It then calculates the mean point, or *centroid*, of each set. It constructs a new partition by associating each point with the closest centroid. The centroids are then recalculated for the new clusters, and the algorithm is repeated by alternate application of these two steps until convergence, which is obtained when the points no longer switch clusters (or alternatively centroids are no longer changed).

The k-means algorithm is very popular because it converges extremely quickly in practice. In fact, many have observed that the number of iterations is typically much less than the number of points. Recently,

however, Arthur and Vassilvitskii [8] showed that there exist certain point sets on which k-means takes a superpolynomial time of $2^{\Omega(\sqrt{n})}$.

In terms of performance the algorithm is not guaranteed to return a global optimum. In fact, obtaining the global optimum is a well known NP-complete problem. The quality of the final solution depends largely on the initial set of clusters and may, in practice, be much poorer than the global optimum. Since the algorithm is extremely fast, a common method is to run the algorithm several times and return the best clustering found [9].

Another main drawback of the algorithm (although it is irrelevant to our particular application) is that it has to be told the number of clusters (the value of k) to find. If the data is not naturally clustered, the results may not make sense. Also, the algorithm works well only when spherical clusters are naturally available in data.
The basic structure of the algorithm is the following:

1. Arbitrarily choose k initial centers $C = \{c_1, ..., c_k\}$.
2. For each $i \in \{1, ..., k\}$, set the cluster C_i to be the set of points in χ that are closer to c_i than they are to c_j for all $j = i$.
3. For each $i \in \{1, ..., k\}$, set c_i to be the center of mass of all points in $C_i : c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$
4. Repeat Steps 2 and 3 until C no longer changes.

We will now describe a scenario in which this algorithm is usually used, and discuss its relationship with our own problem.

3.2 An Example Application: k-means for Facility Location

The general facility location problem is: a set of customers (along with their locations) who have requests then:

- Where should the facilities be located?
- Which customers should be served from which facilities so as to minimize the total cost of serving all the customers?

Typically here facilities are regarded as open (used to serve at least one customer) or closed and there is a fixed cost which is incurred if a facility is open. Which facilities to have open and which closed is our decision. Other factors often encountered are:

- Customers have an associated demand with capacities (limits) on the total customer demand that can be served from a facility
- Customers being served by more than one facility.

This problem is typically solved by grouping the locations of customers requests and opening facilities at the center of those groups. One of the definitive ways of solving this problem [10] is by using the k-means algorithm as described previously. Since they are irrelevant to our application, we will ignore these factors.

We define a patron request as a 4-tuple: (x_p, y_p, x_d, y_d), where x and y are coordinates of the origin (o) and destination (d) of the request. We will relate this problem to our setting in the following way: Customers will be the patron requests and facilities to locate will be the shuttles (in the space of patron

requests). Additionally, we want to do this dynamically in two ways. Dynamically because facilities will be located as the requests come, and because the facilities can be "relocated" if the requests aren't coming in for a given facility.

3.3 Assigning Shuttles to Patrons

In this section we will describe how the shuttles are assigned to patrons. The discussion will be done in a clustering algorithm neutral way, but we use k-means. Our assignment algorithm is as follows:

1. The last r requests are stored[1]. Each time a new request n comes in, an old request is removed.
2. These r requests are clustered into s shuttles.
3. Look up the cluster of r (one of the s there are) that is nearest to the incoming request n. Call this shuttle w.
4. Assign the incoming request n to w.

There are several other issues that we had to address in our method:

- *Distance function to compare patron requests:* The distance (or similarity) between two patron requests is computed using the Euclidean distance. We experimented with distances using dot products in higher dimensional spaces (using the kernel trick [13] and kernel k-means [15]) without any significant improvement. We also experimented with some one sided distance function without obtaining significant improvement.
- *Stopping criteria for k-means:* the stopping criteria for k-means was 20 iterations or clusters not moving. We also tried with 100 iterations or clusters not moving without obtaining any notable difference.
- *Initialization of k-means:* the 300 patron requests clustered via k-means are initialized to belong to cluster $i \mod k$.

Finally, our shuttle assignment algorithm is very simple but it is well founded. It uses the theory behind clustering algorithms and facility location problems.

3.4 The k-means Scheduling Algorithm

Based on the k-means clustering algorithm, we devised a solution to the problem formulation presented above. The basic structure of the algorithm is as follows. Assume there is a set S of shuttles, where $|S| = k$. Each shuttle $s_j \in S$ has a *current request* which it is servicing at any given moment. This request can be *null* if the bus is currently idle. The following is performed at each discrete time step in the simulation:

1. A new request r_i is placed in the system.
2. Calculate the closest pickup and delivery nodes, referred to as np_i and nd_i, respectively.

[1] In our simulations, we used 300 requests. There are several tradeoffs involved with this number. First of all, the more we store the slower it becomes. Moreover, the more we store the further back the algorithm will be considering, which is not necesarily good. If there are too few, the clustering algorithm will not have enough information about where the requests are coming from.

3. Using the k-means algorithm described above, assign r_i to shuttle s_i. Add r_i to s_i's set of pending requests if s_i's capacity has not been exceeded; if it has, add the request to s_i's set of overflow requests.

4. for each shuttle $s_j \in S$, if *currentRequest*(s_j) $\neq$ *null*:

   (a) if there is a request in the set of pending requests for s_j that has the current node as either pickup node or delivery node, then either pickup or drop off the corresponding patron.

   (b) if the current node corresponds to the target node according to the current request, then either pickup up or drop off the patron, depending on the nature of the current request; if the set of pending requests is not empty, decide which request in this set to serve next:

      i. assign a *score* to each pending request according to a scoring function. The scoring function used in this paper is $\alpha * \textit{waiting time} + \beta * \textit{distance to serve}$.

      ii. order the requests according to their scores and choose the one with the highest score.

   (c) calculate the node that minimizes the distance between the current node and the target node, and drive to that node.

5. Each time a patron is dropped off, check the set of overflow requests to see if a request can be moved to the set of pending requests.

In our implementation, we found through trial and error that $\alpha = 100$ and $\beta = 1$ was a good combination of weights. It must be noted here that this weighting scheme prevents starvation of requests, since at some point the oldest request in the system will be assigned the highest score by some bus and therefore be chosen for service.

In the next section we describe the other on-demand scheduling algorithm which is similar to the one just described, except that incoming requests are placed in a global pool instead of being immediately assigned to a bus in particular. As we will see in Section 5, this difference has large impacts on almost every aspect of the algorithms' performance.

4 The General Pool-based Scheduling Algorithm

The general pool-based algorithm assumes there are n buses running in the space, a common pool of requests GP and each bus has its own pool LP consisting of all requests which pick up has been already satisfied. Requests for a ride have a desired pickup point p and a desired delivery point d. For each request, p and d are mapped to a *pickup node* and *delivery node* which are the corresponding closest nodes in the map. All requests are added to a global pool GP and the bus(es) will try to visit nodes to pick up or drop off requests in the global pool. The following is done for each bus at each discrete time step in the simulation; bus moves are done in round robin fashion:

1. For every node S at which the bus arrives, the following changes to the pools are made:

   (a) All requests that are in LP that have S as their delivery node are removed from LP.

   (b) While the number of passengers does not excede the capacity of the bus, requests in GP which have S as their pick up node are removed from GP and added to the bus's LP. If the bus reaches its capacity, it no longer picks up patrons; the remaining patrons must then wait for the next bus to arrive.

2. Each bus evaluates the best next move to achieve overall minimization of the waiting time. The bus must select a future node to commit to, and an immediate node to visit in the path to the committed one.

3. The selection of the committed node is done by evaluating the benefit of visiting the corresponding node using the following formula:

$$B_{i,j} = \sum_k f(k)$$

where $B_{i,j}$ is the benefit of moving from node i, where the bus is in that moment, to node j, f is a function of the wait time of all requests k in GP that are going to be picked up in j and all requests that are already in the bus's LP that are going to be dropped off at node j. Function f could be a linear function $f = (t_{\text{current}} - pt_k)$, polynomial (for instance quadratic) $f = (t_{\text{current}} - pt_k)^2$, or exponential $f = 2^{(t_{\text{current}} - pt_k)}$, where t_{current} is the current system time, and pt_k is the time at which request k was placed.

4. The bus selects the node j that maximizes that function, and it becomes its "committed node", this means that once selected, the bus is committed to travel to it and that decision cannot change.

5. The bus must then select the future immediate node to visit in order to eventually reach the committed node j. To do this, it evaluates the highest benefit path from i to j assuming that there is no other bus running, and the next node to visit is the next node in that path.

6. Once the next node to visit is selected, the bus moves to it.

We found that the exponential function leads to better results in terms of total waiting time for the passengers.

There are additional enhancement that can be made to this algorithm in order to make it perform better. For instance, at all times the expected average wait for all passengers can be evaluated and, if it is over a certain threshold, a new bus can enter the system. We did not include this enhancement in our system because it would make it impossible to compare adequately with the other algorithms. Another possible enhancement is for the buses to take into account the existence of other buses in the system when evaluating the benefit function over the possible destinations.

5 An Empirical Evaluation

In this section we report the experiments we ran in our simulator for each system. The basic structure of each experiment is to perform a series of 20 runs, each consisting of 2,000 minutes of simulation time. The results we include in this section are the averages over the 20 runs. In each case, the same stream of random requests is fed to each system in order to ensure an adequate comparison.

We divided our empirical evaluation into two sections. In Section 5.1, we discuss how the systems behave when varying the number of requests per minute, while in Section 5.2 we vary the number of on-demand buses that are available in the system. For each parameter variation, include graphs that plot the total waiting time, the waiting time at the stop, trip time, and average distance covered per bus.

5.1 Varying Request Frequency

For this set of experiments, we maintained the number of on-demand shuttle buses fixed at nine, while varying the frequency of requests from three per minute to one every five minutes. The current system is

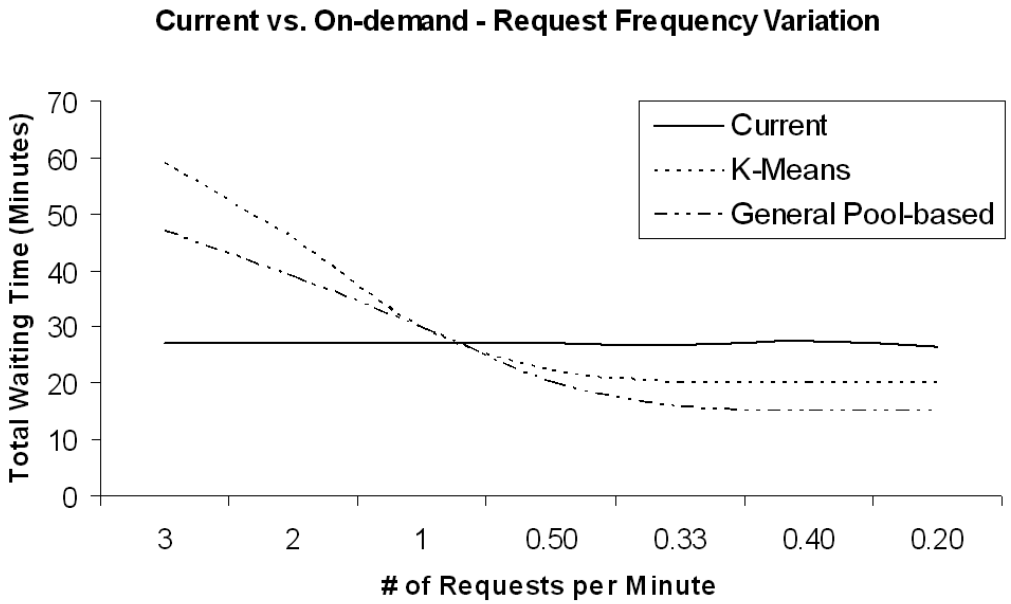

Figure 1: Total waiting time as a function of request frequency. We can see here that both on-demand systems start saturating around one request per minute, where they start performing worse than the current system.

fixed at a total of nine buses, distributed as they are in the actual system on campus (two for the Orange, Blue, Gold, and Purple lines, and one for the College Park Metro line).

The first observation we make when observing Figure 1 is that as the system becomes saturated with requests the waiting time becomes unacceptably high for both on-demand algorithms, while it remains constant for the current system. This is consistent with the intuition that a fixed route system will handle high loads better because it doesn't have to deal with particular cases. According to the curves in the figure, both on-demand algorithms start saturating around one request per minute. Between the two on-demand systems, the general pool-based method is consistently better with respect to total waiting time, though there is not a very large difference. This is remarkable given the excellent usage of resources (gas) of the k-means based method, as discussed below.

Figure 2 suggests that the k-means based algorithm makes better use of resources, since the distance covered is less in all cases. We believe this has to do with having a well founded criteria to assign the requests to buses early on in the life span of the request. Even though it's performance in this case is not as good, the general pool-based algorithm is consistently better than the current system in this department.

Regarding waiting time at node, the general pool-based algorithm performs significantly better than the k-means based algorithm as can be seen in Figure 3. This can be explained by the formulation of the general pool-based algorithm, which considers the sum of the waiting times for the set of requests assigned to a node (either for pickup or delivery) instead of each request individually; this tends to decrease the waiting time. However, this algorithm does pay the price for this improvement, as can be seen in the poor behavior regarding trip time in Figure 4.

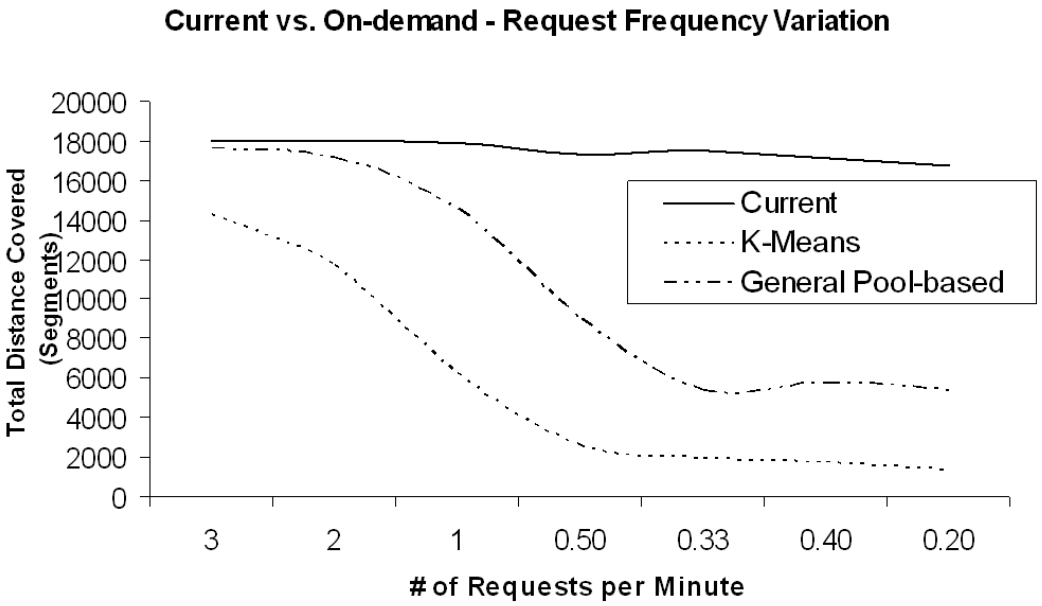

Figure 2: Total distance covered as a function of request frequency. The figure shows that the k-means based method is consistently better than the other two with respect to this measure.

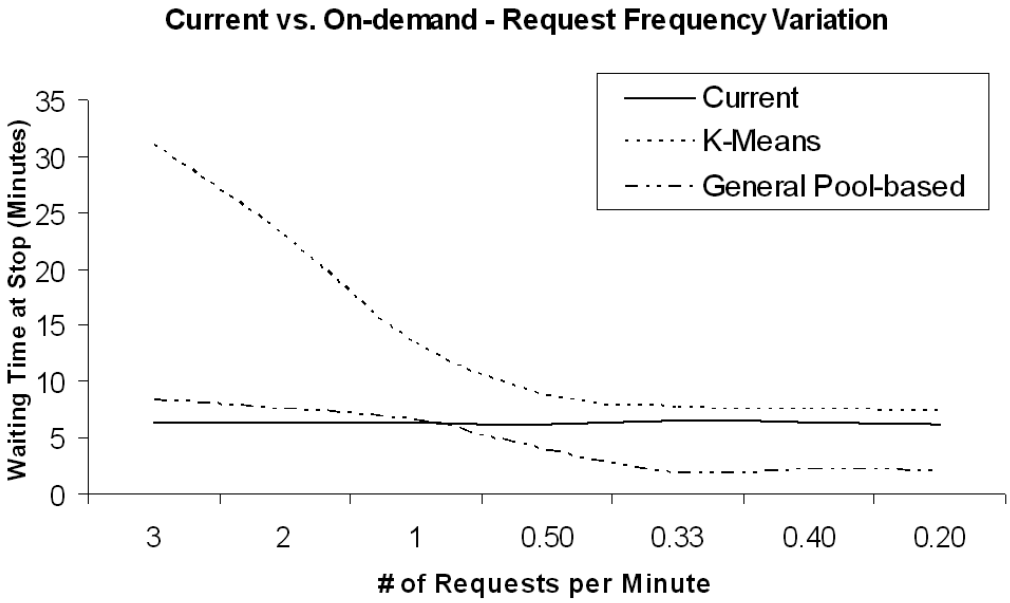

Figure 3: Waiting time at stop as a function of request frequency. The best algorithm in this department is the general pool-based method. Below one request per minute it is better than the current system. The k-means based algorithm shows a steep ascent as it becomes saturated before the other two.

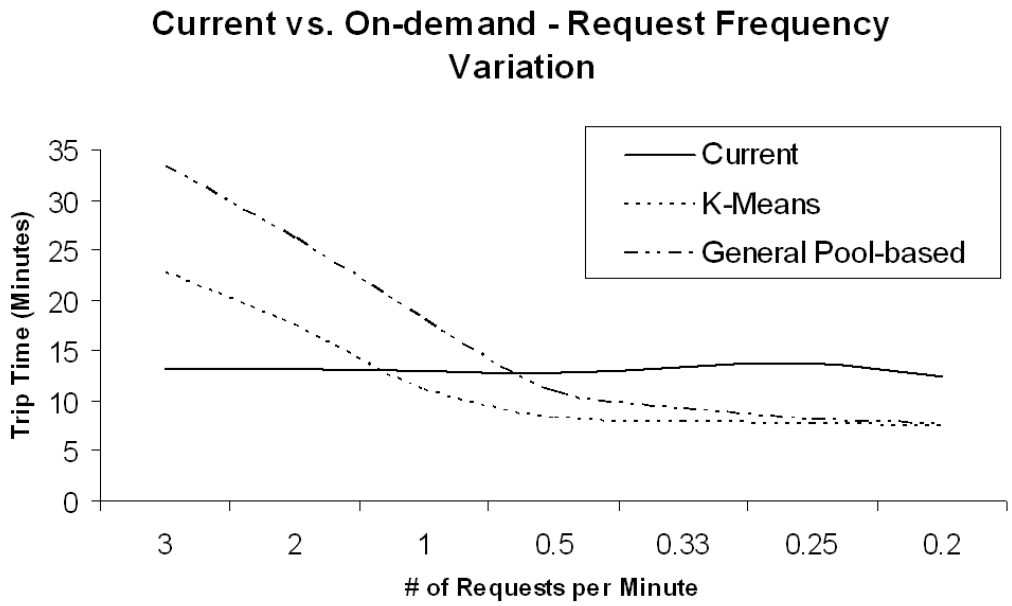

Figure 4: Trip time as a function of request frequency. Both on-demand systems show a similar behavior, though the k-means based algorithm is better than the general pool-based algorithm with respect to this measure. Both become better than the current system at two requests per minute or fewer.

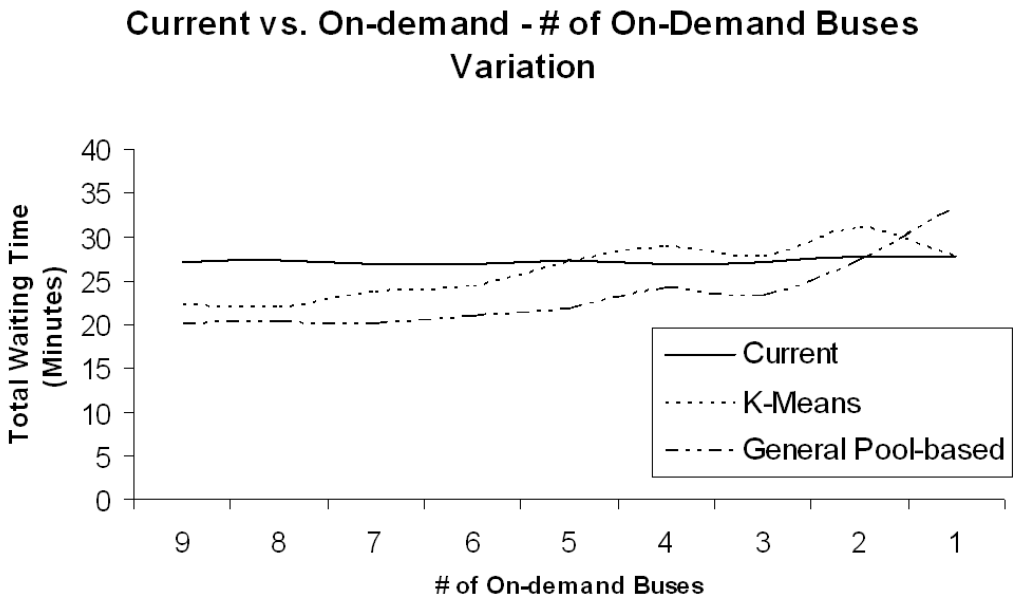

Figure 5: Total waiting time as a function of number of on-demand buses. Neither on-demand system shows significant change as the number of buses decreases, though the wait does have a tendency to increase.

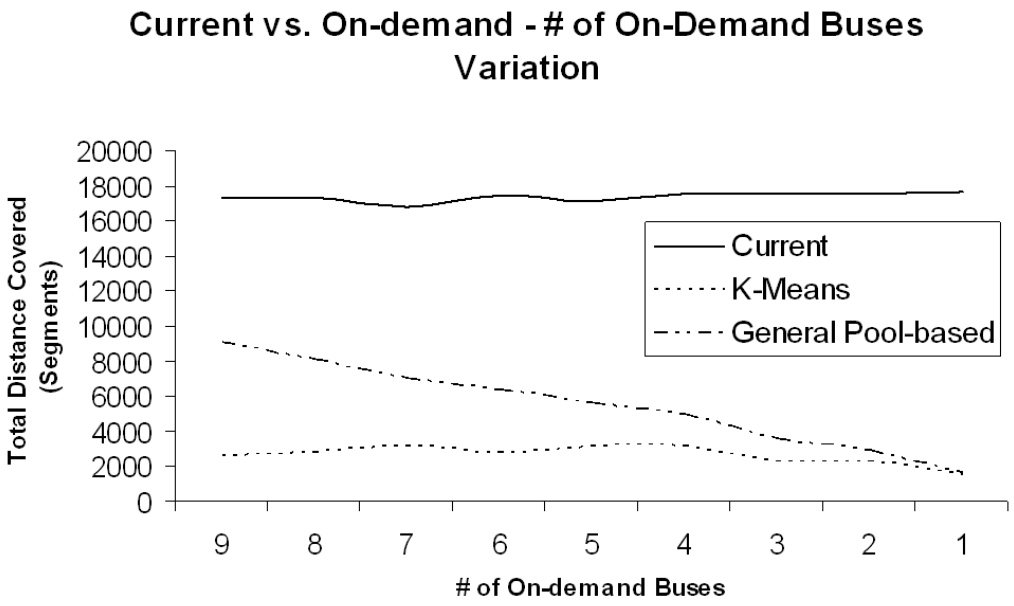

Figure 6: Total distance covered as a function of number of on-demand buses. Both on-demand algorithms are better than the current system with respect to this measure, and the k-means based method is the best of the two. The most affected by the variation is the general pool-based algorithm, which displays a gradual descent as the number of buses decreases.

5.2 Varying Number of On-Demand Buses

For this set of experiments, we maintained the frequency of requests fixed at one every two minutes, while varying the number of on-demand buses from nine to one. As before, the current system is fixed at a total of nine buses, distributed as they are in the actual system on campus.

Varying the number of on-demand buses available to both algorithms did not have the dramatic effect that we expected. In Figure 5, we can see that the general pool-based method is consistently better then the k-means based method, except for the case in which only one bus is available. In comparison with the current system, the k-means based algorithm performs better with more than 5 buses, while the general pool-based method is better with more than 2.

Figure 6 shows the total distance covered by each method. We can observe a difference between the two on-demand systems in that the k-means based algorithm doesn't seem to be greatly affected by the variation in number of buses, while the general pool-based algorithm shows a gradual descent as the number of buses decreases.

As we change the number of shuttles used there is a change of phase in the waiting time at node for the general pool-based algorithm, as shown in Figure 7. This has to do with its eager policy of letting people on when there are not many buses covering the area and therefore it is not very effective, and the wait time goes up.

Finally, Figure 8 shows how trip time varies as a function of the number of buses. There is a tendency in both on-demand systems to increase their trip times when the number of buses decreases, but they are not greatly affected by this change.

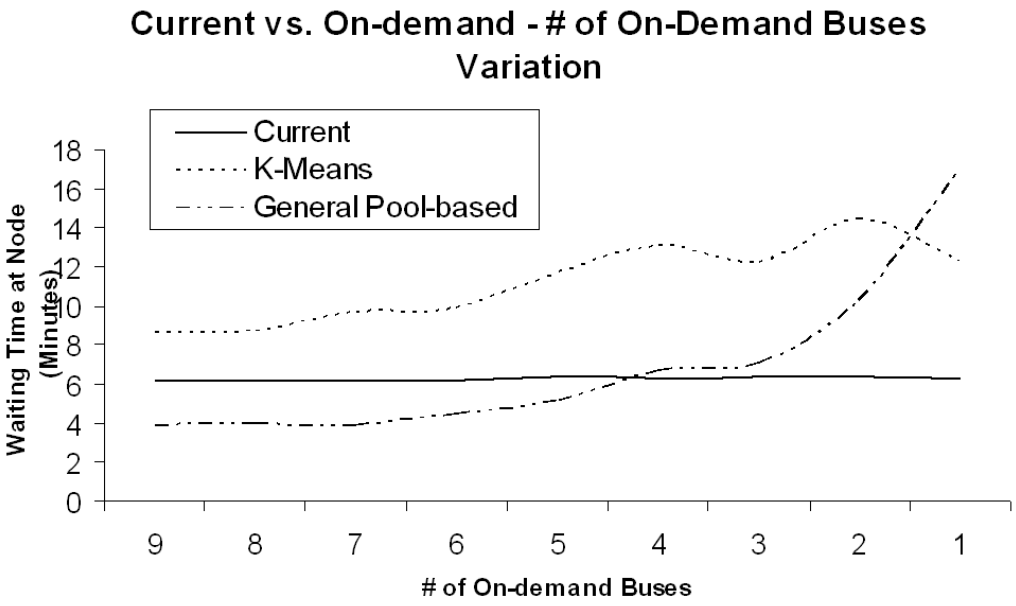

Figure 7: Waiting time at stop as a function of number of on-demand buses. This figure shows that the k-means based method always performs worse than the other two, except with only one bus where it beats the general pool-based method. The latter outperforms the current system with more than 4 buses, where it starts showing a growing ascent.

5.3 Summary of Results

One last measure that wasn't included in the results reported above was the average walking time to reach a stop from where the request was placed and from the dropoff point to where the patron is headed. Both on-demand systems yielded an average walking time of 5.20 seconds, while the current system yielded 7.50 minutes. Even though this difference may at first glance be explained by the fact that there are more nodes in the on-demand systems than stops in the current system (about 20% more), this is not enough to justify the increase. The fact that there are no fixed routes does, however, have a marked influence in this measure. For any given request placed in the current system, it is most likely that there will only be one or two lines that will serve it adequately, and therefore the stops will be more likely to be located farther away. On the other hand, in the on-demand system the patron is asked to walk to the closest node and wait for the bus to come, thus reducing the distance to be walked.

Summing up, the k-means algorithm is more restrictive because requests are assigned to particular buses, but it is very efficient delivering them to their destination. On the other hand, the general pool-based algorithm picks up more people quickly but has a hard time getting them to their destination. If the end-to-end trip time were the same, one could argue for one or the other, but in our experiment regarding total end-to-end time the general-pool-based algorithm has a slight edge. However, the distance covered by each bus (and therefore the amount of gas used) is significantly less in the case of the k-means based algorithm.

It is not possible to say which algorithm is the best overall, but our experiments show that both on-demand methods work well provided that the system does not get saturated with requests. If it does, the current static system is very difficult to beat.

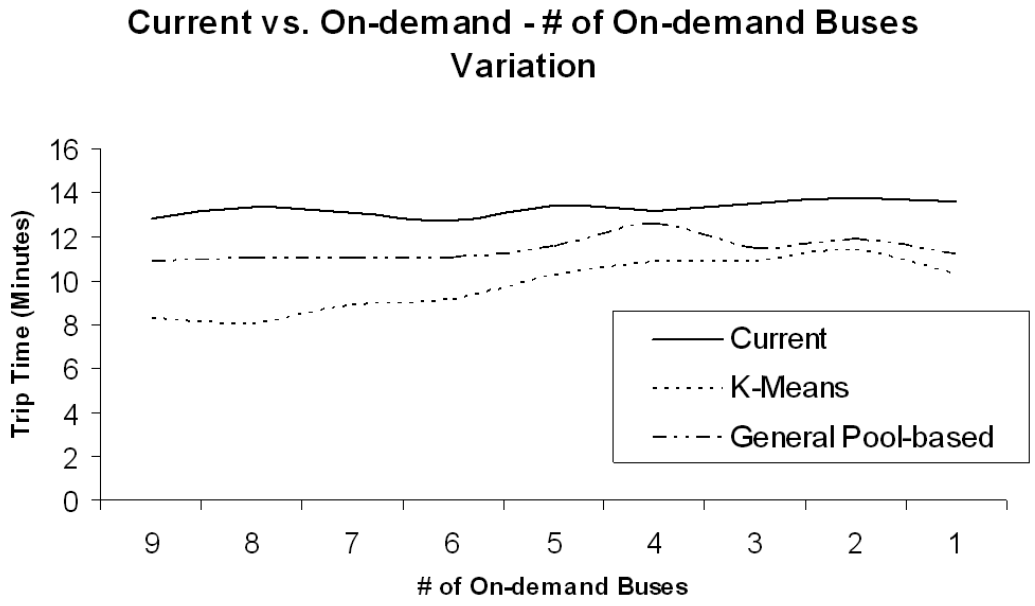

Figure 8: Trip time as a function of number of on-demand buses. Neither on-demand system seems to be greatly affected by this variation, though there is a tendency to increase with the decrease in number of buses. The k-means based algorithm outperforms the other two consistently.

6 Related Work

In this section we will briefly discuss some of the most relevant work done on problems that are similar to the one tackled in this project.

6.1 The Dial-a-Ride Problem

As it can be observed by a simple reduction, the Dial-a-Ride Problem is a generalization of the Travelling Salesman Problem, and therefore NP-hard. Much of the literature on this problem starts by making this observation and deciding that the focus should be on obtaining approximate solutions, which are usually focused on particular cases of the problem.

Paepe et al.[14] propose a general notation (based on similar notations used in scheduling and queuing problems) to be able to discuss, and classify according to complexity, all the possible variations of the *offline* DARP. This approach yielded a class of 7,930 problems, which the authors classified into complexity classes aided by a computer program, allowing them to discriminate among solvable in polynomial time, NP-hard, and open problems. The possible variations were classified into four categories:

- *Servers:* Servers can be of different type (unique, parallel, have different speeds, etc), and have varying capacities (unit, fixed, or unlimited capacities).
- *Rides:* Rides can have different types of locations for pickup and delivery. For instance, if the pickup and delivery locations always coincide, we have an instance of the Travelling Salesman Problem. This category also includes time-window constraints, as the possitbility of preemption and the application of precedence constraints.

- *Metric Space:* The metric space can be either a *line*, a *tree*, a *graph*, a *d-dimensional real space*, or a *general* metric space. In real spaces, Euclidean distance is applied, and on graphs the shortest path distance is used.
- *Objective Function:* There are three possible functions: minimize *makespan* (the maximum time to serve a request), minimize the sum of completion times, or minimize the sum of *weighted* completion times.

This gives rise to $15,360$ problems, but the authors reduced it to $7,930$ by applying equivalence lemmas. Their automated problem classifier had as input the current knowledge about these problems (vehicle routing, machine scheduling, and single-server problems on the line or on a tree). The program computed that: 2.27% (180) are easy, 96.76% (7,673) are hard, and 0.97% (77) are still open.

There is no direct mention of the offline version of our problem in this paper, but discussion of general characteristics suggest that it is hard.

6.2 The General Pickup and Delivery Problem

Another study, done by Savelsbergh et al. [24], discusses some variations of what they call the "general pickup and delivery problem" (GPDP). The GPDP consists of transporting loads from a set of origins to a set of destinations by a fleet of vehicles, where no transfer of loads is allowed. In the PDP (non-general case), each load has a single origin and destination. The DARP is a special case of the PDP, where the loads to be transported are persons (of size 1). Contrary to the one discussed above, this paper does consider the online case. The authors start by pointing out features of the problem that affect the way in which solutions can be found. Namely, these features are *requests*, *time constraints*, and *objective functions*.

In the GPDP, the way in which requests become available is a crucial aspect of the problem and its possible solutions. As pointed out by the authors, the system can be either static or dynamic, but well focus on dynamic systems because they are the most pertinent to our own work. Dynamic problems are often solved as a sequence of static problems. In *online* algorithms, the schedule is updated each time a new request is issued; however, it is usually beneficial to buffer incoming requests and update the schedule with a set of more than one request. Another aspect of dynamic systems is how to incorporate knowledge of the spatial or temporal distribution of requests, if any. For instance, if we know that at certain times of day it is more likely that requests become clustered around certain areas, how can we incorporate this knowledge into the scheduling algorithm?

Time constraints also usually play a crucial role in the GPDP. Time contraints can be divided into two categories: *explicit*, which are placed at issue time ("pick up no later than 15:00 hours"), and *implicit*, which arise as a byproduct of other considerations such as minimizing wait time at the pickup location, or keeping average waiting time below a certain threshold. The presence of time constraints makes the problem of finding a feasible plan NP-hard. However, as the authors point out, time constraints may make it easier for an optimization method to find a solution because they make the solution space smaller.

Objective functions in the GPDP can take a variety of forms, such as minimizing duration, completion time, travel time, route length, client inconvenience, number of vehicles, or maximizing profit (which usually will take all of the previous considerations into account). However, in dynamic systems, duration, completion, and travel time have no clear meaning. The authors conclude that in this case, the function should emphasize decisions that affect the near future over those that have a more distant effect.

Finally, the authors survey three solution proposals to variations of the GPDP. They point out that dynamic aspects of the Pickup and Delivery problem are not very well studied, which is true for most combinatorial optimization problems. The first work covered is a solution proposed for the *single vehicle*

problem. Even though this seems to be an extensive simplification, single vehicle DARP solutions are usually used as subroutines in large scale multi vehicle dial a ride environments, and are therefore interesting to study for our scenario.

In [21], Psaraftis proposed an extension to an algorithm for the static case to deal with dynamism. In his approach, request reception time defines a natural order among passengers; the actual ordering of the requests for delivery may defer from this ordering. The difference between the positions in each ordering defines the "delivery position shift", and constraints on this shift can be placed in order to avoid starvation. The problem is solved as a sequence of static problems; each time a new request arrives, a modified version is solved. All completed requests are discarded and current requests are updated to start at the bus's position. Later on, the same author developed an algorithm for a scenario where the vehicles are ships [23]. Even though it is a very similar problem, in this case the capacity of ports must be taken into account. The method is based on a rolling horizon principle, *i.e.*, it only considers requests that can be "seen". However, by not committing the whole set of requests, it allows some capacity to be saved for the future. The tentative assignment is solved in two phases: (1) calculate the utility of each (ship, load) pair, and (2) solve the assignment problem subject to all the capacity restrictions. This allows the application of heuristics that can approximate the optimal values while still dealing with the complexities of dynamism.

Finally, Frantzeskakis and Powell [16], among others, consider the problem of dynamic full-truck load PDP, which means that requests can be rejected. They represent the problem by means of a flow network, where arcs represent loaded moves known at time 0. In this representation, a maximum profit flow corresponds to an allocation of vehicles to requests. The network is extended with stochastic links in order to anticipate future uncertain trajectories. Because vehicles are only allocated to known loads, the network must be reoptimized periodically.

7 System Design

In this section we present the the implementation details of the system. In particular we focus on the aspects that have to do with geographic information systems, and the visualization that we built for our system.

7.1 Geocoding

While there is still some confusion within the geographic community regarding the formal definition of the term geocoding [17], for our purposes it suffices to regard it as the process of converting a postal address into a longitude-latitude pair. Accurate geocoding depends on the availability of high-quality addressing infrastructure such as the TIGER database [5] distributed by the US Census Bureau. Even if such database is available and contains up-to-date data, the problem of address format standardization remains; For example, *"615 Rhode Island Ave, Washington DC"*, can be interpreted as either *"615 Rhode Island Ave NE"* or *"615 Rhode Island Ave NW"*. Furthermore, sometimes a ZIP-code is needed to fully disambiguate an address such as in the case of *"100 Washington St, Boston MA"*, due to annexation of several cities without changing street names. Once the address is fully disambiguated, the geographical database may be consulted to find the corresponding geographical coordinates. Since not every single street address appears in the database, often some approximation is necessary. The easiest method is to interpolate using close-by addresses appearing in the database. However, due to numbering irregularities and occasional lack of correspondence between street address number and distance on the ground, the mapping may not be sufficiently accurate. Since the problem of accurate geocoding is an active research

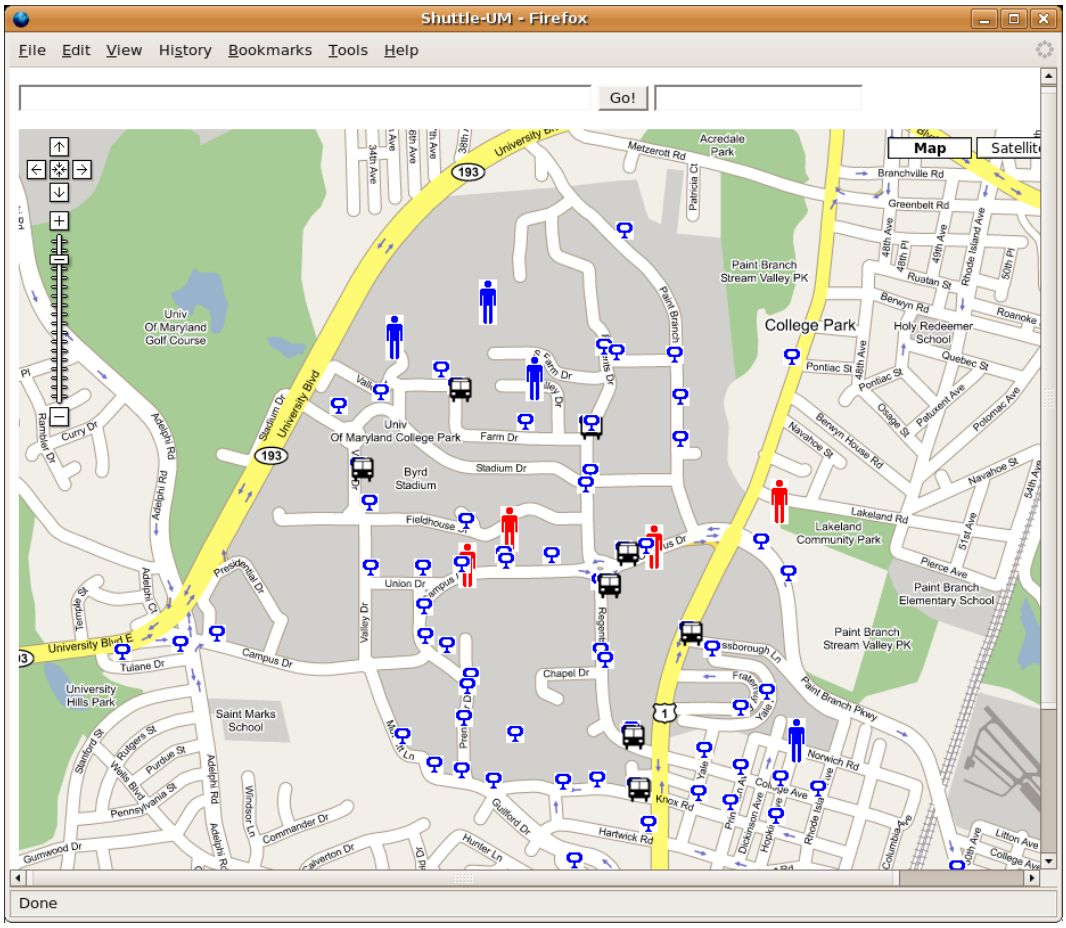

Figure 9: A screenshot of the system visualization. The system shows the current position of each bus, where requests were placed, and the bus stops.

area but not the focus of this work, we will use an existing geocoding service in our implementation.

7.2 User Interface

Figure 9 shows a screenshot of the user interface. Markers show the position of patron requests, the bus stops, and the current location of the buses. By hovering over an object with the mouse, a tooltip appears exposing useful information such as the bus number, its occupancy, its heading, etc. The exact type of information depends on the actual algorithm used in the specific simulation. Pickup locations are represented by blue person icons whereas drop off locations are represented by red person icons. The top-left text box allows inputting an address to be geocoded and presented on the map. Also, a draggable balloon-shaped marker can be used to find out the latitude-longitude coordinates of a given point. The pair will be shown on the top-right box. This is the user interface patrons would use to request bus pickups. Other means of inputting current location would be using a GPS receiver; In this case, location can be derived, for example, from the $GPRMC sentence of the NMEA 0183 protocol [4]. Our system simulates these patron requests, as if they came from a GPS equipped cell-phone or PDA. Therefore, sensor data collection does not have to be directly implemented in our system and is assumed to be performed by an existing module.

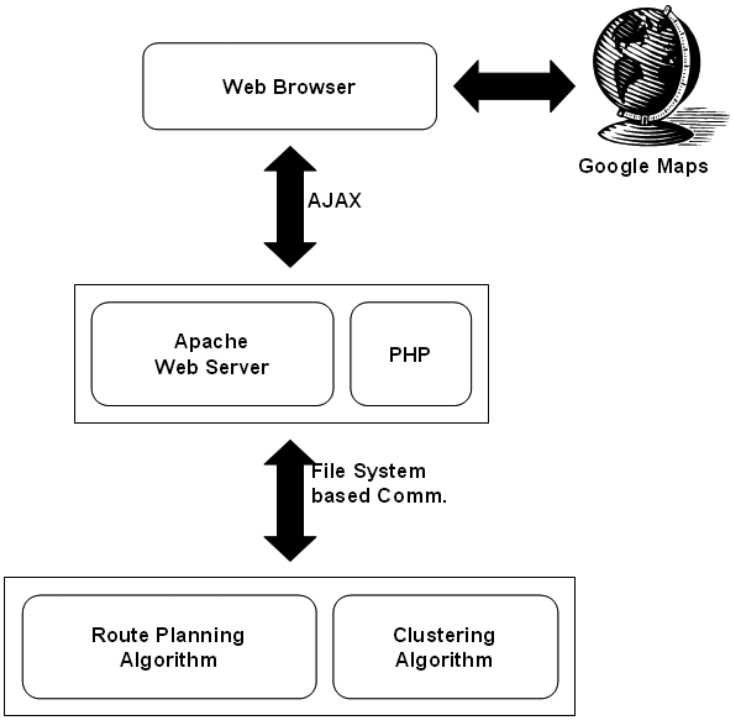

Figure 10: The system architecture.

7.3 System Architecture

There are several freely available mapping services such as Google Maps [1], MapQuest [2], Yahoo! Maps [7], and Microsoft Live Search Maps [3]. These services provide slightly different features, but all allow plotting a map of the US as a minimum. For our system, we have chosen to use Google Maps. This service provides a rich JavaScript API that is accessible using a unique API key generated at the signup for the service. The API key is usable only from within the domain name specified at signup, and is used to rate-limit the number of requests made to the Google servers. We use the API to place markers representing people, bus stations, and buses, to perform geocoding, and to control map parameters.

The system is composed of three main modules. The simulator core was written in Java (about 5100 LOC) and is charged with bus scheduling and route planning. In the case of the k-means method, it also uses the clustering algorithm in order to assign patrons to buses. In the next layer, we have the visualization server which uses several PHP scripts to dynamically generate map overlays that depict the current bus and patron locations. The communication between the core and the visualization server is file-system based. The core periodically generates comma-separated value files (CSV) and these are read by the server PHP scripts to produce dynamic content. To avoid potential race-conditions, we employ file-locking.

The CSV protocol is a general protocol for placing markers overlay on a Google Map and therefore can easily be extended to show any object as a marker without having to change code in the server or the client. To introduce new marker types, only the CSV generator has to be changed: in our case, the simulator core. The client layer uses a web browser to present the map and the dynamic overlays. The client periodically requests the server to produce new overlays giving a snapshot of the current status. In

order to provide seamless asynchronous display refreshes, we use a combination of JavaScript and XML (known as an AJAX architecture) to communicate with the server. Our choice of using open architectures leads us to believe that our system can be ported to many different platforms. We have successfully tested our system on versions of Linux and on Windows XP.

8 Conclusions and Future Work

We have presented an on-demand system to serve requests to ride shuttles on a university campus. In our experiments we have compared two on-demand methods (one k-means based and one general pool-based) and a static method which is a simulation of the system currently implemented on campus for the evening routes. Our algorithms are well founded and the results encouraging, showing that under many reasonable conditions the on-demand algorithm is more appropriate for a campus-wide system. In particular it is well suited for evening routes where the number of requests is not very high.

Our experiments show that the k-means algorithm is very selective letting people get on the bus, but it is very efficient delivering them to their destination, while the general pool-based algorithm picks up people quickly but has a hard time getting them to their destination. The general pool-based algorithm is slightly better in waiting time while the distance covered by each each bus (and therefore the amount of gas used) is significantly less in the case of the k-means based algorithm. Our experiments also show that as the number of requests increase, a static system (like the current one) is more convenient.

As future direction, a wide variety of strategies can be tested at each step in our formulation, since we tried simple (yet well founded) strategies. For example, for scheduling purposes a steepest-descent type of algorithm could be tried, where at each step the steepest descent is taken (*i.e.*, the direction that minimizes the sum of the scores). In the case of the assignment of patrons to shuttles, a weighted clustering algorithm can be used and the time waiting at the stop can be used as a natural weight of a request.

In future work, finding an exact solution to a small version of the problem (using some exponential algorithm) might be a good idea to get a better sense of how far off from the optimum the current generation on-demand methods are.

Acknowledgments

We would like to thank Dr. Nick Roussopoulos for his comments, and for proposing the *General Pool-based* scheduling algorithm.

References

[1] *Google Maps*. http://maps.google.com/.

[2] *MapQuest*. http://www.mapquest.com/.

[3] *Microsoft Live Search Maps*. http://maps.live.com/.

[4] *National Marine Electronics Association*. http://www.nmea.org/pub/0183/.

[5] *Topologically Integrated Geographic Encoding and Referencing (TIGER)*. http://tiger.census.gov.

[6] *University of Maryland Department of Transportation Services*. http://www.transportation.umd.edu/.

[7] *Yahoo! Maps*. http://maps.yahoo.com/.

[8] Arthur, D., and Vassilvitskii, S. How slow is the k-means method? In *SOCG* (2006).

[9] Arthur, D., and Vassilvitskii, S. k-means++: The advantages of careful seeding. In *SODA* (2007).

[10] Arya, V., Garg, N., Khandekar, R., Meyerson, A., Munagala, K., and Pandit, V. Local search heuristics for k-median and facility location problems. In *STOC* (2001).

[11] Ascheuer, N., Krumke, S. O., and Rambau, J. Online dial-a-ride problems: Minimizing the completion time. *Lecture Notes in Computer Science 1770* (2000), 639–650.

[12] Bonifaci, V., Lipmann, M., and Stougie, L. Online multi-server dial-a-ride problems. Technical Report 2006-04, Universit di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, Mar. 2006.

[13] Burges, C. J. C. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery 2*, 2 (1998), 121–167.

[14] de Paepe, W., Lenstra, J., Sgall, J., Sitters, R., and Stougie, L. Computer-aided complexity classification of dial-a-ride problems.

[15] Dhillon, I. S., Guan, Y., and Kulis, B. Kernel k-means: spectral clustering and normalized cuts. In *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM Press, pp. 551–556.

[16] Frantzeskakis, L. F., and Powell, W. B. A successive linear approximation procedure for stochastic dynamic vehicle allocation problems. *Transportation Science 24*, 1 (1990), 40–57.

[17] Goldberg, D. W. *Annotated Geocoding Reading List*. http://dwgold.com/geocoder/LiteratureSurvey.pdf.

[18] Jr., C. A. D., Fonseca, F. T., and Borges, K. A. V. A flexible addressing system for approximate geocoding. *Proceedings of V Brazilian Symposium on GeoInformatics* (2003), 223–248.

[19] Krumke, S. O., de Paepe, W. E., Poensgen, D., and Stougie, L. News from the online traveling repairman. vol. 2136 of *Lecture Notes in Computer Science*, pp. 487+.

[20] Potvin, J.-Y., Xu, Y., and Benyahia, I. Vehicle routing and scheduling with dynamic travel times. *Comput. Oper. Res. 33*, 4 (2006), 1129–1137.

[21] Psaraftis, H. N. A dynamic programming solution to the single vehicle many-to-many immediate request dial-a-ride problem. *Transportation Science 14* (1980), 130–154.

[22] Psaraftis, H. N. Analysis of an o(n2) heuristic for the single vehicle many-to-many euclidean dial-a-ride problem. *Transportation Research 17B* (1983), 133–145.

[23] Psaraftis, H. N. Dynamic vehicle routing problems. *Vehicle Routing: Methods and Studies* (1988), 223–248.

[24] Savelsbergh, M. W. P., and Sol, M. The general pickup and delivery problem. *Transportation Science 29* (1995), 17–29.