CAB–SHARING: AN EFFECTIVE, DOOR–TO–DOOR, ON–DEMAND TRANSPORTATION SERVICE

Győző Gidófalvi[1], **Torben Bach Pedersen**[2]

[1]Geomatic ApS, Nybrogade 20, 1203 Copenhagen K, Denmark, T:(+45) 7020 5046, gyg@geomatic.dk

[2]Aalborg University, Fr. Bajers Vej 7E, 9220 Aalborg , Denmark, T:(+45) 9635 9975, tbp@cs.aau.dk

ABSTRACT

City transportation is an increasing problem. Public transportation is costeffective, but do not provide doortodoor transportation; This makes the far more expensive cabs attractive and scarce. This paper proposes a location-based Cab-Sharing Service (CSS), which reduces cab fare costs and effectively utilizes available cabs. The CSS accepts cab requests from mobile devices in the form of origin-destination pairs. Then it automatically groups closeby requests to minimize the cost, utilize cab space, and service cab requests in a timely manner. Simulation-based experiments show that the CSS can group cab requests in a way that effectively utilizes resources and achieves significant savings, making cab-sharing a new, promising mode of transportation.

KEYWORDS

Location-Based Services, LBS, cab–sharing, ride–sharing, optimization.

INTRODUCTION

Transportation in larger cities, including parking, is an ever increasing problem that affects the environment, the economy, and last but not least our lives. Traffic jams and the hustle of parking take up a significant portion of our daily lives and cause major headaches. Solving the problem by extending the road network is a costly and non–scalable solution. A more feasible solution to the problem is to reduce the number of cars on the existing road network. To achieve this, collective / public transportation tries to satisfy the general transportation needs of larger groups in a cost–effective manner. While being cost–effective, the services offered by public transportation often (1) do not meet the exact, door–to–door transportation needs of individuals, (2) require multiple transfers and detours that significantly lengthen travel times, and (3) are limited in off–peak hours. For these reasons, the far more expensive service offered by cabs / taxis, which meet the exact transportation needs of individuals and also eliminates the problem of parking, are in great demand. To better satisfy this demand, this paper presents an LBS that makes use of simple technologies and tools to offer a new cost– and resource–effective, door–to–door transportation means, namely *cab–sharing*.

Collective transportation is not a new concept. It is encouraged and subsidized by transportation authorities all over the world. The optimization of collective transportation has also been

considered. For an extensive list of research papers in this area, the reader is referred to [7]. Two papers, however, are worth highlighting. First, in [3], where the idea of the present research originates from, in an off–line fashion, long, shareable patterns are sought in historical trajectories of moving objects to aid an intelligent ride–sharing application. Second, in [1] an almost "personalized" transportation system is proposed that alters the fixed–line buss service to include variable itineraries and timetables. In comparison, the herein proposed CSS treats the optimization of collective transportation as a truly online process, and alters the inherently routeless transportation service offered by cabs.

PROBLEM STATEMENT

Let $\mathbb{R}^2$ denote the 2-dimensional Euclidean space, and let $\mathbb{T} \equiv \mathbb{N}^+$ denote the totally ordered time domain. Let $R = \{r_1, \ldots, r_n\}$ be a set of *cab requests*, such that $r_i = \langle t_r, l_o, l_d, t_e \rangle$, where $t_r \in \mathbb{T}$ is the *request time* of the cab request, $l_o \in \mathbb{R}^2$ and $l_d \in \mathbb{R}^2$ are the *origin* and *destination locations* of the cab request, and $t_e \geq t_r \in \mathbb{T}$ is the *expiration time* of the cab request, i.e., the latest time by which the cab request must be serviced. A cab request $r_i =< t_r, l_o, l_d, t_e >$ is *valid* at time t if $t_r \leq t \leq t_e$. Let $\Delta t = t_e - t_r$ be called the *wait time* of the cab request. Let a *cab–share* $s \subseteq R$ be a subset of the cab requests. A cab–share is valid at time t if all cab requests in s are valid at time t. Let $|s|$ denote the number of cab request in the cab–share. Let $d(l_1, l_2)$ be a distance measure between two locations l_1 and l_2. Let $m(s, d(.,.))$ be a method that constructs a valid and optimal pick-up and drop-off sequence of requests for a cab–share s and assigns a unique distance to this sequence based on $d(.,.)$. Let the *savings* p for a cab request $r_i \in s$ be $p(r_i, s) = 1 - \frac{m(s,d(.,.))/|s|}{m(\{r_i\},d(.,.))}$. Then, the *cab-sharing problem* is to find a disjoint partitioning $S = \{s_1 \uplus s_2 \uplus \ldots\}$ of R, such that $\forall s_j \in S$, s_j is *valid*, $|s_j| \leq K$, and the expression $\sum_{s_j \in S} \sum_{r_i \in s_j} p(r_i, s_j)$ is maximized.

CAB–SHARING SERVICE

The Cab–Sharing Service (CSS) is depicted in Figure 1. Before using the CSS, a user signs up with the CSS and creates a prepaid user account to pay for the service. A registered user sends a cab request in the form of an origin–destination pair and an optional validity period of the request. In case no validity period is specified, the cab request is assumed to be valid from the time it is received until some default time limit has been reached. The requests are submitted via a mobile device by sending two address lines to a premium SMS service, called Premium Cab–Sharing Service. A more user–friendly specification of requests could include GPS–based localization of origins and/or the ability of users selecting origins and/or destinations from a list of frequent addresses, or even a voice–recognition–enabled, automatic call center. In either case, once received, the Premium Cab–

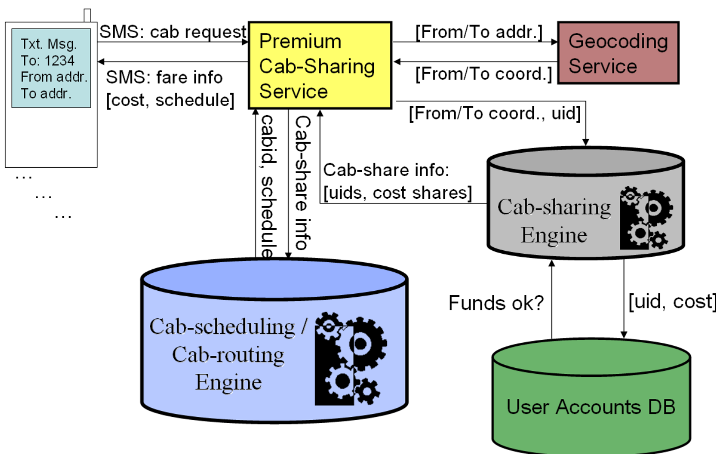


Figure 1 – Cab–sharing service components/process.

Sharing Service sends the two address lines to a Geocoding Service, which validates them and returns the exact coordinates for them. Then these origin and destination coordinates, along with a user identifier are sent to a Cab–Sharing Engine. The Cab–Sharing Engine then, in an online fashion, automatically groups "closeby" requests into a cab–share to minimize the total transportation cost, thereby providing significant savings to the users of the service. Once a cab–share is constructed, the cost of the share is deducted from the account of every participant of the cab–share. Then, after receiving the information about the cab–share, the CSS forwards the information to the Cab–Scheduling / Cab–Routing Engine, which assigns cabs to cab–shares so that cab space is utilized and cab requests are serviced in a timely manner. Finally, the CSS sends an SMS to the user about the cab fare, such as cost and schedule of the fare. A web–demo of the CSS is available at: `www.cs.aau.dk/~gyg/CSS/`.

GROUPING OF CAB REQUESTS

Cab requests can be viewed as data points in spatio–temporal space. Partitioning n data points into k groups based on pairwise similarity of the data points according to a distance measure is referred to as the clustering problem, an extensively researched problem of computer science. Clustering is known to be NP-hard[2]. However, there are a number of efficient bottom-up and top-down methods that approximate the optimal solution within a constant factor in terms of a clustering criterion, which is expressed in terms of the distance measure.

Hence it is only natural to approach the cab-sharing problem as a clustering problem and adopt efficient approximations to the task at hand. For these approximation algorithms to converge to a local optima, an appropriate distance metric $d(.,.)$ between two cab requests and/or cab–shares needs to be devised. For $d(.,.)$ to be a metric for any three cab requests or cab–shares i, j, k is has to satisfy the following four conditions: $d(.,.) \geq 0$ (non-negativity), $d(i,i) = 0$ (identity), $d(i,j) = d(j,i)$ (symmetry), and $d(i,j) + d(j,k) \geq d(i,k)$ (triangular inequality).

While a clustering approach may find a near-optimal cab-sharing solution, it has several drawbacks. Since a cab request is only valid during a specific time interval, the set of valid cab request that can be considered for clustering is changing over time. While a hard time-constraint can be incorporated into a distance measure, the measure will not satisfy the triangular inequality requirement. An alternative approach could at every time step t retrieve the set of valid cab requests, and perform a partitioning-based clustering on the set according to some distance metric. However, since at any time instance t the number of valid cab requests n_t and the number of possible K-sized valid cab–shares are comparable, an iterative partitioning-based clustering approach would entail $O(n_t^2)$ distance calculations per iteration at every time instance t, making it infeasible in practice.

Since a cab request r_i has a request time t_r and an expiration time t_e it is natural to view it as a part of a data stream. When finding cab–shares in such a stream, two opposite approaches are obvious. In the first approach, referred to as the *lazy* approach, the grouping of requests is delayed as long as possible to find cab–shares with higher savings. In the second approach, referred to as the *eager* approach, request are grouped into cab–shares as early as possible to deliver a timely service. Next, the lazy version of a greedy, bottom-up grouping of cab requests is described. For ease of presentation, the described grouping method instead of maximizing savings, solves the equivalent problem of minimizing total travel cost; it is shown in Figure 2.

```
(0)  cabShare (R, K, min_saving)
(1)   S ← {}
(2)   for (t = 1 .. T)
(3)     {R_x, R_q} ← getValidRequests(R, t)
(4)     E ← calculateE(R_x, {R_x ∪ R_q})
(6)     while (|R_x| > 0)
(7)       ê_min ← min_i min_{k≤K} (Σ_1^k E(i,k))/k
(8)       {i, k} ← argmin_i argmin_{k≤K} (Σ_1^k E(i,k))/k
(9)       s ← {r_i^1, r_i^2, ... r_i^k}
(10)      if p̂(r_i, s) < min_saving then break
(11)      S ← {S ∪ s}
(12)      E ← removeSfromE(E, s)
(13)    endwhile
(14)    S ← {S ∪ addRestAsSingles(R_x) }
(15)  endfor
```

Figure 2 – Lazy version of a greedy, bottom-up grouping of cab requests

At any time t, valid cab requests can be divided into two sets: R_x, the set of valid requests that expire at time step t, and R_q, the rest of the valid requests that expire some time after t (line 3). Given two cab requests r_i and r_j, let

$$e(r_i, r_j) = \frac{m(\{r_i, r_j\}, d(.,.)) - m(\{r_i\}, d(.,.))}{m(\{r_i\}, d(.,.))}$$

be the *fractional extra cost* of including r_j in r_i's cab fare. Using the pairwise fractional extra costs, the fractional extra cost of a cab share s w.r.t. r_i is estimated as $\hat{e}(s) = \sum_{r_j \in s} e(r_i, r_j)$, and the average savings for a cab request $r_j \in s$ is estimated as $\hat{p}(r_j, s) = 1 - \frac{1+\hat{e}(s)}{|s|}$. Furthermore, let r_i^k be the cab request that has the k-th lowest fractional extra cost w.r.t. r_i. In line 4, these fractional extra costs are calculated between cab requests in R_x and $\{R_x \cup R_q\}$ and for all $r_i \in R_x$ these fractional extra costs are stored in a 2–dimensional array E, such that $E[i,k] = e(r_i, r_i^k)$, i.e., E is is sorted increasingly in row major order. Then, using only the lowest K entries for each cab request in E, in an iterative fashion the currently best (lowest amortized cost / highest savings) cab–share s is found (lines 7–9) and request in it are removed from consideration (line 12) by setting $E[r_j, .]$ and $E[., r_j]$ to some large value for all $r_j \in s$. This process continues until the currently best savings p_{max} is less than `min_saving`, at which point all the remaining cab request in R_x are assigned to their own "cab–share" resulting in no savings for them (line 14).

A SIMPLE SQL IMPLEMENTATION

The grouping method for parameters `max_k` and `min_saving` can be easily implemented in a few SQL statements as described bellow. First, after geocoding, valid requests are stored in a table `R_q` = ⟨`rid,tr,te,xo,yo,xd,yd`⟩, where `rid` is a unique identifier for the request, `tr` and `te` are request and expiration times, and (`xo,yo`) and (`xd,yd`) are origin and destination coordinates. Then, using a temporal predicate, expiring requests are selected from `R_q` and stored in a table `R_x` with the same schema. Finally, a distance function `d(x1,y1,x2,y2)` is defined between two 2D coordinates. Then, the fractional extra cost function `e` for the origin and destination coordinates of the requests `ri` and `rj` can be defined in SQL–99 [5] as follows.

```
CREATE FUNCTION e(rixo FLOAT, riyo FLOAT, rixd FLOAT, riyd FLOAT,
                  rjxo FLOAT, rjyo FLOAT, rjxd FLOAT, rjyd FLOAT)
RETURNS FLOAT
BEGIN
  DECLARE ri_dist, ed FLOAT
  SET ri_dist = d(rixo, riyo, rixd, riyd)
  SET ed = d(rjxo, rjyo, rixo, riyo) + d(rjxd, rjyd, rixd, riyd)
  RETURN (ed / ri_dist)
END
```

Step 1: Calculating Fractional Extra Costs

After creating a table $\texttt{E} = \langle \texttt{ri,rj,e} \rangle$ to store the fractional extra costs, the fractional extra costs between the requests `ri` in `R_x` and `rj` in `R_q` can be calculated in SQL–99 as follows.

```
INSERT INTO E (ri, rj, e)
SELECT x.rid ri, q.rid rj,
       e(x.xo,x.yo,x.xd,x.yd,q.xo,q.yo,q.xd,q.yd)
FROM R_x x, R_q q
WHERE x.rid <> q.rid
  AND e(x.xo,x.yo,x.xd,x.yd,q.xo,q.yo,q.xd,q.yd) <= 1
```

The first condition in the `WHERE` clause excludes the fractional extra cost of `ri` with itself, which is 0 by definition. The reason for doing so is to avoid falsely identifying `ri` on its own as the currently best (lowest amortized cost = 0 / highest savings = 1) "cab–share" in the processing steps to follow. The second condition in the `WHERE` clause is a pruning heuristic that excludes $(\texttt{ri},\texttt{rj})$ request combinations from `E` where the fractional extra cost exceeds 1, in which case neither `ri` nor any cab–share containing `ri` can benefit from including `rj`.

Step 2: Calculating Amortized Costs

Relational Database Management Systems (RDBMSs) are *set* oriented and the inherently declarative SQL language does not provide adequate support to implement operations on *sequences*, e.g., cumulative sum. Procedural language constructs that allow iteration over the elements of a sequence do exist in SQL, but are implemented less efficiently. Hence, programmers normally revert to other procedural languages to perform such operations. Nevertheless, the calculation of cumulative sum can be implemented in SQL in a declarative fashion using a self–join. Hence, after creating a table $\texttt{AE} = \langle \texttt{ri,rj,ae,k} \rangle$, the summations on line 7 and 8 of the grouping method in Figure 2, i.e. the amortized costs can be calculated in a single SQL–99 statement as follows.

```
INSERT INTO AE (ri, rj, ae, k)
SELECT a.ri, a.rj, (SUM(b.e)+1)/(COUNT(*)+1) ae, COUNT(*)+1 k
FROM E a, E b
WHERE a.ri = b.ri AND a.e >= b.e
GROUP BY a.ri, a.rj
HAVING COUNT(*)+1 <= max_k
```

The `WHERE` clause for every $(\texttt{ri},\texttt{rj})$ combination from the table `a` assigns a *set* of $(\texttt{ri},\texttt{rj})$ combinations from the table `b`, such that `ri`'s match in the two tables and the fractional extra costs values (`e`) in table `b` are less than or equal to the values in table `a`. The latter condition in a sense imposes an *order* on the *set*. The aggregation for each such $(\texttt{ri},\texttt{rj})$ combination (set) is achieved through the `GROUP BY` clause. The corresponding aggregates `ae` and `k` are calculated by the two expressions in the `SELECT` statement, where `ae` is the amortized cost of the best cab–share of size `k` that contains requests `ri` and `rj`. Finally, the `HAVING` clause excludes cab–shares larger than size `max_k` from further consideration. Note that, while the calculations of sequence–oriented cumulative aggregates, for example amortized cost (`ae`) are simple to express in SQL, the computation performed is not optimal. While the computational complexity of sequence–oriented cumulative aggregates is $O(n)$, for a sequence of length n, the complexity of the above method based on self-joins is $O(n^2)$. Nevertheless, the self–join based simple SQL implementation can process in real–time up to 100,000 requests per day.

Step 3: Selecting the Best Cab–share

After creating a table CS = ⟨sid,rid⟩ to store the cab–shares, one can select the savings, b_savings, the size, b_k, and, conditioned on the min_savings parameter, store the requests of the currently best cab–share (with ID = s) in two SQL–99 statements as follows.

```
SELECT ri, (1-ae), k INTO b_rid, b_savings, b_k
FROM AE ORDER BY ae LIMIT 1
```

```
INSERT INTO CS (sid, rid)
SELECT s sid, b_rid rid FROM AE
UNION
SELECT s sid, rj rid FROM AE WHERE k <= b_k AND ri = b_rid
```

Step 4: Pruning the Search Space

Since a cab request can only be part of a single cab–share, if the current best cab–share meets the minimum saving requirement, and is added to CS, the requests in it have to be discarded from further considerations for finding cab–shares in the future. This can be achieved by deleting tuples from the E table that refer to the requests in question. The SQL–99 statement for this is as follows.

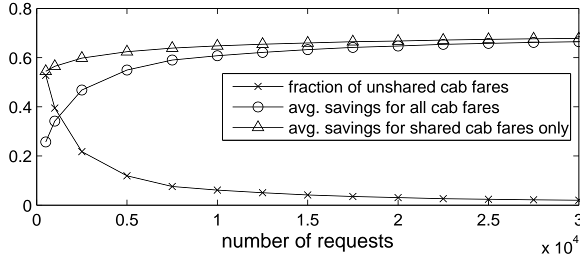
```
DELETE FROM E
WHERE ri IN (SELECT rid FROM CS WHERE sid = s)
   OR rj IN (SELECT rid FROM CS WHERE sid = s)
```

Periodic, Iterative Scheduling of Cab Requests

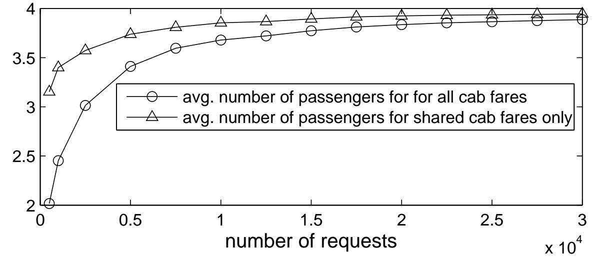
All cab requests in R_x are grouped in an iterative fashion by executing steps 2 through 4 until (1) there are no more cab–shares that meet the minimum savings requirement, or (2) all requests in R_x has been assigned to some cab–share. The loop iterating through these steps is placed in a stored procedure. Using the automatic task scheduling facilities of the operating system, cron in Linux or Task Scheduler in Windows, this stored procedure is executed periodically. Keeping the period of the executions of the stored procedure short (frequency of executions high) has several advantages. First, the shorter the period, the longer requests can be delayed until they *have to be* grouped into cab-shares, giving the requests more opportunities to end up in a good cab–share. In effect, the set of expiring requests is composed of requests that will expire before the next scheduled execution of the stored procedure. Second, smaller sets of expiring requests means smaller E and AE tables, which are cheaper to maintain during the iterations of a single execution of the stored procedure.

EXPERIMENTS

To test the proposed methods, cab request data was simulated using ST-ACTS, a spatio–temporal activity simulator [4]. Based on a number of real world data sources, ST–ACTS simulates realistic trips of approximately 600,000 individuals in the city of Copenhagen, Denmark. For the course of a workday, out of the approximately 1.55 million generated trips, approximately 251,000 trips of at least 3–kilometer length were selected and considered as potential cab requests. Experiments were performed for various maximum cab–share sizes $K \in [2, 8]$, wait times $\Delta t \in [1, 20]$, and cab request densities, i.e., various–sized, random subsets of the



(a) Average savings and fraction of unshared cab fares



(b) Cab space utilization

Figure 3 – Performance evaluations for varying number of cab requests.

set of potential cab requests. Figures [3(a), 3(b)], in which the units on the x scale is 10,000 cab requests, show some of the results for parameter settings $K = 4$, $\texttt{min_saving} = 0.3$, and $\Delta t = 15$ minutes (common for all cab requests).

Figure 3(a) shows (1) the fraction of unshared cab requests, and (2) the average savings for *all fares*, and for the *shared fares only*. As the density of cab requests increases, and hence the likelihood of two individuals wanting to travel around the same time from approximately the same origin location to approximately the same destination location increases, the number of cab–shares, meeting the required minimum savings also increases. Consequently, the fraction of unshared requests decreases to a point where only about 2% of the cab requests cannot be combined into cab–shares that meet the required minimum savings. Similarly, as the density of the cab requests increases, the average savings for fares also increases up to a point where the average savings per fares is 0.66 ± 0.11 considering all the fares, and is 0.68 ± 0.06 considering shared fares only. In other words, the CSS is able to group cab requests in a way such that the cost of 97.5% of the cab fares can be reduced by two thirds on average. Figure 3(b) shows how well cab space is utilized. As the density of cab requests increases, the average number of passengers per cab also increases up to a point where the average number of passengers per cab is 3.89 ± 0.49 considering all the fares, and is 3.94 ± 0.27 considering shared fares only.

Due to space limitations, the detailed results of the experiments showing the effects of parameters Δt and K are omitted, but they can be summarized as follows. The Δt experiments confirm that due to the linear relationship between Δt and the resulting spatio–temporal density of *valid* cab requests, there exists a correspondence between the above results and the omitted results, i.e., since the spatio–temporal density of valid cab requests for 15,000 requests with $\Delta t = 15$ minutes are about the same as for 30,000 requests with $\Delta t = 7.5$ minutes, the average savings and cab utilization are approximately the same in both cases. The K experiments confirm that under a fixed cab request density, both the savings and cab utilizations saturate. In the case of 30,000 requests for $K \in [2, 8]$, the average savings for shares gradually increases from 0.47 to 0.79 and the average number of passengers for shares gradually increases from 2 to 6.1.

The savings come at the expense of some delay in the CSS when meeting the end–to–end transportation needs of its users. There are three sources for this delay. First, the *grouping time*, i.e., the time that a user has to wait until his/her requests is grouped into a cab–share, which is upper bounded by the *wait time* parameter of the requests. Second, the *pickup time*, i.e., the extra time a user has to wait because some of the other members of the cab–share need to be picked up before him/her. Finally, the *additional travel time*, i.e., the extra time the cab–fare takes due to the increased length of the shared part of the cab–fare. Because no realistic simulation of the transportation phase of the CSS was performed, the delay incurred due to the latter two sources has been evaluated in terms of extra distances relative to the length of the original requests. Due to the close to constant results for various cab request densities,

the measurements on the delay due to the above three sources can be (independently from the cab request density) summarized as follows. The average grouping time is 11.7 minutes with a standard deviation of 5.9 minutes. The average pickup time is equivalent to $10.7 \pm 13.5\%$ of the length of the original request. Given the average length of requests of 4.95 kilometers, and assuming an average transportation speed of 40 km/h in the city, the average pickup time is approximately 0.8 ± 1.1 minutes. The average additional travel time is equivalent to $7.9 \pm 10.1\%$ of the length of the original request, or is approximately 0.6 ± 0.9 minutes. Hence in total, the approximate additional service delay an average CSS user experiences compared to using a conventional cab service is approximately 12.1 ± 7.9 minutes, arguably a small price to pay for the savings.

CONCLUSION AND FUTURE WORK

Motivated by the need for a novel transportation alternative that is convenient, yet affordable, this paper proposes a new LBS, namely a Cab–Sharing Service (CSS). To achieve the desired reduction in transportation cost, the paper proposes a greedy grouping algorithm, along with a simple but effective SQL implementation, that optimally groups "close by" requests into cab–shares. Experiments on simulated, but realistic cab request data show that in exchange of a short (5–15 minute) wait time, the CSS can group together requests in a way that effectively utilizes resources and provides significant savings to the user.

Future work is planned along several directions. First, since it is natural to view the incoming requests as a data stream, the CSS is being implemented using an in–memory Data Stream Management System (DSMS) [6]. Second, the cab–sharing problem is a hard optimization problem, hence investigating new heuristics for it is planned. Third, while the proposed greedy method is computationally efficient, a number of improvements to it are possible, for example to use spatial indices to prune the search space of possible cab–share candidates. Finally, while not considered here, the optimization of the Cab–Scheduling / Routing Engine through spatio–temporal cab request demand prediction is planned.

ACKNOWLEDGEMENTS

This work was supported in part by the Danish Ministry of Science, Technology, and Innovation under grant number 61480.

REFERENCES

[1] T. G. Crainic, F. Malucelli, and M. Nonato. Flexible many-to-few + few-to-many = an almost personalized transit system. In *Proc. of TRISTAN*, pp. 435–440, 2001.

[2] J. Han and M. Kamber. "Data Mining: Concepts and Techniques". Morgan Kaufmann, 2005.

[3] G. Gidófalvi and T. B. Pedersen. Mining Long, Sharable Patterns in Trajectories of Moving Objects. In *Proc. of STDBM*, pp. 49–58, 2006.

[4] G. Gidófalvi and T. B. Pedersen. ST–ACTS: A Spatio-Temporal Activity Simulator. In *Proc. of ACM-GIS*, pp. 155–162, 2006.

[5] Standard SQL 1999, ISO/IEC 9075:1999.

[6] E. Zeitler and T. Risch. Using stream queries to measure communication performance of a parallel computing environment. To appear in *Proc. of DEPSA*, 2006.

[7] Transportation Problems: `www.di.unipi.it/optimize/transpo.html`