Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 32 (2014) 339 – 347

Procedia Computer Science

5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014)

Theory and Practice in Large Carpooling Problems

Irith Ben-Arroyo Hartman[a,b,*], Daniel Keren[a], Abed Abu Dbai[a], Elad Cohen[a], Luk Knapen[c], Ansar-Ul-Haque Yasar[c], Davy Janssens[c]

[a]*Department of Computer Science, University of Haifa, Haifa 3498838, Israel*
[b]*Caesarea Rothschild Institute, University of Haifa, Haifa 3498838, Israel*
[c]*Transportation Research Institute(IMOB) Hasselt University, Wetenschapspark 5, B3950 Diepenbeek, Belgium*

Abstract

We address the carpooling problem as a graph-theoretic problem. If the set of drivers is known in advance, then for any car capacity, the problem is equivalent to the assignment problem in bipartite graphs. Otherwise, when we do not know in advance who will drive their vehicle and who will be a passenger, the problem is NP-hard. We devise and implement quick heuristics for both cases, based on graph algorithms, as well as parallel algorithms based on geometric/algebraic approach. We compare between the algorithms on random graphs, as well as on real, very large, data.

Keywords: Carpooling, Linear Programming, Maximum Weighted Matching, Star Partition Problem, Gradient Projection Algorithm, Scalability, Incremental Algorithms

1. Introduction

Car pooling is a scheme according to which several people share a common vehicle simultaneously, in order to reach common, or nearby destinations. The advantages of carpooling are multiple: it decreases the number of cars on the roads, thus reducing traffic congestion, vehicle emissions, and noise. It saves fuel, time, parking spaces, and decreases the number of accidents. In addition, it encourages sociability between people, and reduces stress in driving to work.

The car pooling problem has been approached from diverse points of view, see for example[2]. The main challenge is how to match between people to share a ride, and to decide who picks up whom with their vehicle, so that it will be worthwhile both for the driver and all the passengers. Even if we assume that every driver can pick up at most one other passenger, the problem is challenging.

In[7,8] an automatic service for carpooling is described. People register their *periodic trip executions* (PTE), these are periodic trips, where the base period is typically one week. Each PTE contains detailed information about the trip such as origin and destination of the trip, earliest and latest departure and arrival times, the maximal detour distance acceptable, and the capacity of the car, if it is available. In addition, each PTE also contains registration information of

[*] Corresponding author. Tel.: +972-4-828-8370 ; fax: +972-4-828-8181.
E-mail address: irith.hartman@gmail.com

the person such as age, gender, educational level, special interests (like music type preferences), job category, driver license availability, etc. The system suggests to individuals to share a car based on the details of their trips, as well as their registration information. The assumption is that continued successful cooperation between people requires some level of similarity. After the system makes its suggestions, together with appropriate financial incentives for the driver, the individuals evaluate the proposed carpool, negotiate it, and possibly agree to cooperate. Once the individuals have shared a trip, they can evaluate each other (e.g. was the driver/passenger punctual? Did the driver obey traffic rules? Was the vehicle comfortable?) and give feedback to the system, for consideration in future trips.

In Section 2 we formulate the car-pooling problem as a graph-theoretic problem which was shown to be NP-hard[5]. If the set of drivers (i.e. vehicles) are known and all we want to do is assign passengers to vehicles, the problem becomes tractable from the theoretical point of view. However, since we are dealing with a very large number (millions) of vertices, it is still challenging to find quick algorithms and heuristics which run efficiently. We describe some of these algorithms in Section 3. In Section 4 we propose some heuristics and their results for the general carpooling problem, and finally, in Section 5 we define some extensions to the carpooling problem defined so far.

2. Problem formulation

Let V denote the set of Periodic Trip Executions, or for short, trips. Each trip is associated with an individual who either drives her own vehicle, or travels as a passenger with some other driver. A loop (x, x) denotes the fact that x can be a driver of her own vehicle. A directed edge $(i, j) \in E$, where $i \neq j$, has weight w_{ij} representing the compatibility of person i to get a ride with driver j (in driver j's vehicle), i.e. it represents the probability that the negotiation between them succeeds and i can travel in $j's$ vehicle. It takes into account the information related to both PTE's i and j, as well as registration information and feedback. The weight of a loop is zero. Each driver j also has a maximum capacity of people that she can take in her vehicle (including herself), denoted by c_j. A linear programming formulation of the problem was mentioned in[7].

A graph theoretic formulation was mentioned in[5]. We repeat here the definitions for the sake of completeness. A *directed star* S_r is a graph $(K_{1,t})$ consisting of a center vertex r called the *root* and t vertices directed to it called *leaves*. A *star family* is a collection $\Gamma = \{S_r : r \in R\}$ of vertex disjoint directed stars. Given a directed graph $G = (V, E)$ and a function $c : V \to \mathbb{N}$. A star family $\Gamma = \{S_r : r \in R\}$ is *feasible* if each star with root r contains in G a loop (r, r) and its star-in-degree, including the loop, is at most $c(r)$. A *star partition* in G is a star family that covers $V(G)$, i.e. it is a collection of vertex disjoint directed stars that cover $V(G)$. Given a weight function $w : E \to \mathbb{R}$, the *weight* of a star, $w(S_r)$, is the sum of the weights of its edges, and $w(\Gamma) = \Sigma_{S_r \in \Gamma} w(S_r)$.

We remark that a directed star with root r may have in-degree one, in which case it is a single vertex $v = K_1$ with a loop, which we call the *trivial star*. Therefore, if every vertex in G has a loop, then G has a feasible star partition, the trivial one, consisting of $|V|$ trivial stars.

The carpooling problem can be formulated as follows:

Problem 2.1. *Let $G = (V, E)$ be a directed graph with edge weights $w : E \to \mathbb{R}$, and let $c : V \to \mathbb{N}$. Find a feasible star partition $\Gamma = \{S_r : r \in R\}$ of maximum weight.*

Solution to problem 2.1 guarantees maximum compatibility between people and their priorities for "taking rides." It does not guarantee, however, that the number of vehicles used is as small as possible. However, when $w_{ij} = 0$ or 1 these problems are equivalent, as was shown in[5].

Problem 2.2. *Let $G = (V, E)$ be a directed graph, and let $c : V \to \mathbb{N}$. Find a feasible star partition $\Gamma = \{S_r : r \in R\}$ containing a minimum number of stars.*

We have shown[5] that Problem 2.2 is NP-hard even for the special case where $c : V \to \mathbb{N}$ is restricted to $c(v) \leq 3$, i.e. each car can hold two passengers and the driver. Assume, now, that $c(v) \leq 2$ for all $v \in V$. Then whenever we have a cycle of length two, (i.e. edges (x, y) and (y, x) in the graph), we may, without loss of generality, remove the smaller weight edge (or arbitrarily one edge, if they are of equal weight), and remain with a directed graph with no 2-cycles. The problem is then reduced to finding a maximum weight matching (or maximum cardinality matching) in a general graph. These problems are well known and have solutions with algorithmic complexity $O(|E||V|\log|V|)$[4] and $O(\sqrt{|V|}|E|)$[12], respectively. For graphs with millions of vertices the computing time is too large, and these algorithms are not practical. We will therefore implement heuristic algorithms, and incremental algorithms which we describe in Section 3.

3. The bipartite graph case

Assume now that the set of drivers is known. In other words, we are given set R of roots, and we are looking for a feasible star partition $\Gamma = \{S_r : r \in R\}$ of maximum weight.

Problem 3.1 (Weighted Bipartite). *Let $G = (V, E)$ be a directed graph with edge weights $w : E \to \mathbb{R}$, let $c : V \to \mathbb{N}$, and let $R \subseteq V$ be a set of potential roots. Find a feasible star family $\Gamma = \{S_r : r \in R\}$, such that the total weight of the edges of the stars is maximized.*

In this problem all edges in the original graph connecting two passengers or two drivers, are irrelevant, and can be deleted. Hence the graph is reduced to a bipartite graph $G = (X \cup Y; E)$, where X denotes the set of passengers, or non-drivers, and Y denotes the set of drivers. We would like to match as many vertices in X to Y under the capacity constraints of the vertices in Y. When the graph is weighted we would like to maximize the total weight of the matched edges. We remark that when the capacities of the vertices in Y are greater than two, it is not a matching we are looking for, but a (g, f)- factor; in-other-words, we are looking for a maximum weight subgraph H that meets every vertex in X in at most one edge, and every vertex y in Y in at most $c(y) - 1$ edges ($c(y) - 1$ is the maximum number of passengers y can carry). This problem can be reduced to a matching problem by making $c(y) - 1$ copies of each vertex y in Y, say $y_1, y_2, ...,$ and connecting each y_i to all the neighbors of y. It is not difficult to verify that a matching in the new graph corresponds to a (g, f)- factor in the original graph of the same weight, and vice-versa. We assume, therefore, that we are faced with a maximum weight bipartite matching problem. The matching problem in bipartite graphs can be solved in $O(\sqrt{|V|}|E|)$[6]. In the weighted case, the problem is also known as the "assignment problem", and can be solved in $O(|V|^2 \log|V| + |V||E|)$[3]. These complexities do not allow us to run the algorithm on very large graphs, hence heuristics and incremental algorithms are desirable.

3.1. Greedy Solutions

The greedy heuristics for finding a maximum weight matching in a graph (bipartite, or general) is as follows:

Algorithm *Greedy Matching*

```
1.  Initialize: Sort the edges in E in decreasing order of weights, $E' \leftarrow E$, $M \leftarrow \emptyset$
2.  while ($E' \neq \emptyset$)
3.      do Let $e = (x, y)$ be the first edge in $E'$
4.          $M \leftarrow M \cup \{e\}$
5.          $E' \leftarrow E' \setminus \{e; e$ has an endpoint in $x$ or $y\}$
6.  return $M$
```

The complexity of the greedy matching algorithm is the same as the complexity of sorting the edge weights, i.e. $O(|E|\log|V|)$. If the graph is un-weighted (i.e. the edge weights are 0 or 1), then the greedy matching runs in time $O(|E|)$.

Performance of the greedy matching:

Theoretically, the weight (or cardinality) of the greedy matching algorithm is at least half the weight (or cardinality) of the optimal matching, where equality can occur. This can be easily proved, since in the worst case, every edge picked by the greedy algorithm destroys two edges of the optimal solution. The surprising fact is that when random graphs were generated, with random weights according to a normal distribution, the greedy heuristics performed extremely well! The ratio of the greedy algorithm to the optimal algorithm was on average better than 0.96.

Data Generation: 1000 random bipartite graphs were generated, each graph of size 500 by 500 vertices with 10% edge density. Similar results were found for 50% edge density. The results are given in Figure 1.

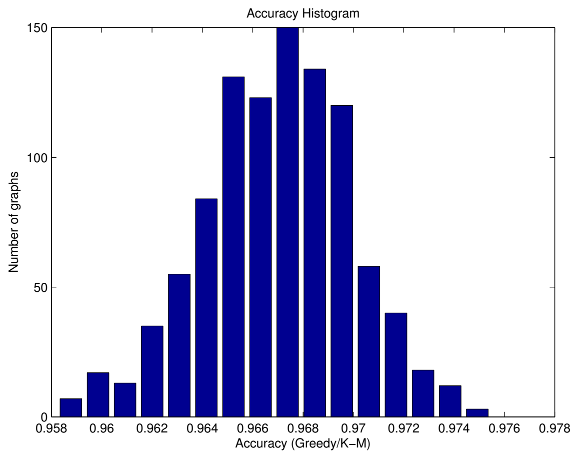

Fig. 1. Solution quality of the greedy algorithm for maximum weight matching on random bipartite graphs with 1000 vertices

3.2. Incremental Solutions

The weighted bipartite matching problem can be described by the following linear program:

$$\text{Maximize} \sum_{ij} w_{ij} x_{ij}$$

Subject to

$$\text{for all } i \ \sum_j x_{ij} = 1 \tag{1}$$

$$\text{for all } j \ \sum_i x_{ij} = 1 \tag{2}$$

$$\text{for all } i, j \ \ x_{ij} \geq 0 \tag{3}$$

The dual LP is given below:

$$\text{Minimize} \sum_i u_i + \sum_j v_j$$

Subject to

$$\text{for all } i, j \ \ u_i + v_j \geq w_{ij} \tag{4}$$

$$\text{for all } i \ \ u_i, v_i \geq 0 \tag{5}$$

The variables x_{ij} correspond to the edges (i, j) in the bipartite graph, and $x_{ij} = 1$ if and only if the edge (i, j) is in the matching. Otherwise, $x_{ij} = 0$. Without loss of generality we assume here that the bipartite graph is complete, by adding edges of weight zero to the original graph. The dual variables, u_i and v_j correspond to the vertices of the bipartite graph. We assume now that we have an optimal, or close to optimal solution to the matching (or weighted matching) problem in a given bipartite graph $G = (X \cup Y, E)$. Suppose a small fraction of the edges changes. They can appear, or disappear, or their edge weight may change. We call the new graph the *perturbed graph* - G'. We address the following problems: 1. Can we estimate how far the solution of the new, perturbed graph is from the solution of the original graph? 2. Can we quickly find an accurate solution, or an estimated solution to the perturbed graph, based on a solution to the original graph? (without starting the computation from “scratch”)

To address Problem 1, assume we are given optimal solutions x^0_{ij} and u^0_i, v^0_i to the LP and the Dual LP problems, respectively, for a given graph G with edge weights vector $\mathbf{w}$. Assume that a new weight vector $\mathbf{w}^1$is given by

$w^1_{ij} = w_{ij} + \epsilon_{ij}$, and let x^1_{ij}, and u^1_i, v^1_i be optimal solutions to the new weight vector $\mathbf{w}^1$. A straight forward upper bound to the solution of the new problem with perturbed weights can be obtained by defining: $u''_i = u^0_i + \max_j \epsilon_{ij}$ and $v''_i = v^0_i$.

It is not difficult to see that u''_i and v''_i is a feasible covering to the new problem, and hence we have the bound

$$\sum_{ij} w^1_{ij}x^1_{ij} \leq \sum_i u''_i + \sum_j v''_j = \sum_i u^0_i + \sum_i \max_j \epsilon_{ij} + \sum_j v^0_j = \sum_{ij} w_{ij}x^0_{ij} + \sum_i \max_j \epsilon_{ij} \tag{6}$$

Similarly, we can define $u''_i = u^0_i$ and $v''_j = v^0_j + \max_i \epsilon_{ij}$ to obtain another feasible covering to the new problem, and another upper bound to the weight of $\mathbf{x}^1$.

$$\sum_{ij} w^1_{ij}x^1_{ij} \leq \sum_i u''_i + \sum_j v''_j = \sum_i u^0_i + \sum_j v^0_j + \sum_j \max_i \epsilon_{ij} = \sum_{ij} w_{ij}x^0_{ij} + \sum_j \max_i \epsilon_{ij} \tag{7}$$

The bounds in the RHS of equations (6) and (7) are not necessarily the same and the smaller bound is clearly a better upper bound.

We can obtain even better bounds for the optimal matching problem with the perturbed edge weights by improving the coverings u''_i and v''_j defined above. One way to do that is by defining

$u''_i = u^0_i + \max_j\{\epsilon_{ij} - (u_i + v_j - w_{ij})\}$ and $v''_i = v^0_i$.

This bound is better than the previous bound since $u_i + v_j - w_{ij} \geq 0$ for all i, j.

The vectors u''_i and v''_j are still a feasible covering to the perturbed problem, and we get the improved bound

$$\sum_{ij} w^1_{ij}x^1_{ij} \leq \sum_i u''_i + \sum_j v''_j = \sum_i u^0_i + \sum_i \max_j\{\epsilon_{ij} - (u_i + v_j - w_{ij})\} + \sum_i v^0_i = \sum_{ij} w_{ij}x^0_{ij} + \sum_i \max_j\{\epsilon_{ij} - (u_i + v_j - w_{ij})\} \tag{8}$$

.

Similarly, we can change only the vector $\mathbf{v}$ and leave the vector $\mathbf{u}$ as it is, i.e. define:

$u''_i = u^0_i$ and $v''_j = v^0_j + \max_i\{\epsilon_{ij} - (u_i + v_j - w_{ij})\}$ to obtain another feasible covering to the new problem.

So far we have chosen an initial vertex covering u''_i and v''_j by updating either $\mathbf{u}$ or $\mathbf{v}$. Another approach would be to update both the vectors $\mathbf{u}$ and $\mathbf{v}$ by first finding a minimum vertex cover to the unweighted problem defined by the set of edges in G which are not covered, with the perturbed weights, by u^0_i and v^0_i. In other words, consider only the edges (i, j) for which $\epsilon_{ij} - (u_i + v_j - w_{ij}) > 0$. We run König's algorithm (also known as the "Hungarian Algorithm") for finding a maximum cardinality matching and minimum cardinality covering u^1, v^1 for the subgraph spanned by these edges. Now we define

$u''_i = u^0_i + \max_j\{\epsilon_{ij} - (u_i + v_j - w_{ij})\}$ and $v''_j = v^0_j + \max_i\{\epsilon_{ij} - (u_i + v_j - w_{ij})\}$ only for vertices i, j for which $u^1_i = 1$ and $v^1_j = 1$, respectively. For vertices i (or j) which are not in the covering, i.e. $u^1_i = 0$, (or $v^1_j = 0$) we define $u''_i = u^0_i$ (or $v''_j = v^0_j$).

To address Problem 2 we use the Kuhn-Munkres[9,11] (K-M) algorithm. Assume we are given a maximum weight matching and a minimum weight covering of G. This can be found using any graph-theoretic algorithm (like the K-M algorithm), or by solving the LP described above. We now update the covering so it will be feasible for the new perturbed graph G'. We use the upper bound estimates for a covering as described above. Once we have an updated covering of G', we update the "equality subgraph" (as in the K-M algorithm), and continue running the K-M algorithm until an optimal solution is found.

Practical results: A random bipartite graph with 400 by 400 vertices and 20% edge density was generated. The graph was perturbed 40 times with a 5% change of the edge weight, in each generation. The solution for each generation was calculated by both the standard K-M algorithm and and three different incremental K-M algorithms. The results, presented in Figure 2, showed a significant improvement in the running time of the incremental K-M compared to the running time of the standard K-M. Similar results were found for 50% edge density, and for the other incremental algorithms. When 20% of the edges were changed in each generation, there was no advantage to using the incremental K-M algorithm.

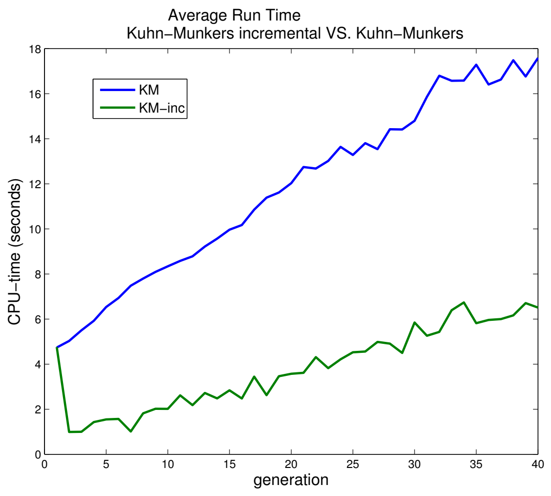

Fig. 2. Running time of the standard K-M algorithm vs. incremental K-M algorithm for maximum weight matching

3.3. Algebraic Approach

The Kuhn-Munkres algorithm is serial. For very large graphs, it is desirable to find an algorithm that can be run in parallel, thus allowing substantial speedup. We now describe an iterative algebraic algorithm for solving the LP described above, which is very easy to parallelize.

The *standard simplex* is defined by $\{\mathbf{y} \in R^n \mid \sum_{i=0}^{n-1} y_i = 1, y_i \geq 0\}$. We observe that all the constraints in (1) and (2) actually describe the standard simplex. This observation leads to a *gradient projection algorithm*. In each step, the algorithm increases the entries of $\mathbf{x}$ in the direction of the gradient of the relevant objective function, and then projects the rows and columns of $\mathbf{x}$ on the standard simplex (the gradient is trivial to compute, as we seek to optimize linear functions). There are many simple and efficient algorithms for the projection onto the standard simplex [1,10]. The main issue is choosing the order of the vectors which we project onto the standard simplex, since the projections are not all independent; the projections of the rows are independent of each other and so are the projections of the columns. However, projecting a column may violate some of the row constraints and projecting a row may violate some of the column constraints. Hence, we repeat this step until $\mathbf{x}$ converges to a doubly stochastic matrix, i.e., a matrix that satisfies all rows and columns constraints. In practice, this step is repeated only a few times (typically twice or three times) until the desired convergence is reached. In the final step, we round $\mathbf{x}$ to a binary doubly stochastic matrix, which represents the solution for the problem.

```
Algorithm Gradient projection to the standard simplex algorithm
Input: $\mathbf{x} \in R^{n\times n}$, $w : \mathbf{x} \to \mathbb{R}$, $\alpha \in \mathbb{R}$, $\delta \in \mathbb{R}$
1.  Initialize: $\mathbf{x}^0 = 0$, $k = 1$
2.  while $\alpha \geq \delta$
3.      do $\mathbf{x}^k = \mathbf{x}^{k-1} + \alpha w_{ij}$
4.          repeat
5.              $\forall i$ project row $i$ of $\mathbf{x}$ onto the standard simplex
6.              $\forall j$ project column $j$ of $\mathbf{x}$ onto the standard simplex
7.          until $\mathbf{x}$ converges to a doubly stochastic matrix
8.          decrease $\alpha$
9.          $k = k + 1$
10. round $\mathbf{x}^{k-1}$ to a binary doubly stochastic matrix
11. return $\mathbf{x}^{k-1}$
```

Practical results: A thousand random graphs, each with $2n$ vertices, were generated with edge probability $\frac{10}{n}$ and weight function according to a normal distribution. We ran the experiments for $n = 10, 50, 250, 500, 1000$ on a serial computer. The results are shown in Figure 3. In practice, the running time of the gradient projection algorithm is much shorter, since the projections, which consume around 98% of the running time, may be computed in parallel. The approach presented in this section is especially suitable to the incremental scenario (in which only a fraction of the weights are changed), as one can initiate the iterative process at the previous value.

4. Heuristics for the general carpooling problem

We have mentioned in Section 2 that the general carpooling problem is NP-hard. We suggest here two heuristic algorithms, a simplest greedy heuristic and a more complex one. We will compare between the algorithms in terms of running time, as well as their output.

The input of the algorithms is a directed graph $G = (V, E)$, weight function $w : E \to \mathbb{R}$, capacity function $c : V \to \mathbb{N}$ denoting the capacity of the vehicle associated with the vertex and a set of potential drivers who have vehicles, $D \subseteq V$. Vertices not in D do not have a vehicle and must be assigned as passengers in other vehicles. The output of the algorithm is a set of edges H which is a feasible star family, and a set U of vertices not covered by the edges in H. The vertices (PTE's) in U will either take their own vehicle (if they have one), or will remain unmatched by the algorithm. The following considerations are taken into account in the second heuristic algorithm: (i) It is preferable to match non-drivers before potential drivers, since potential drivers, if unmatched, can always drive their own vehicle, (ii) Since we are trying to minimize the number of vehicles, it is preferable to assign passengers to existing vehicles, than to 'new' vehicles, (iii) If a new vehicle is used, it is preferable to use a vehicle with larger capacity, than to use a small capacity vehicle. The complexity of the basic greedy algorithm is the same as the complexity of sorting the edge set - $O(|E| log|V|)$. The complexity of the second greedy algorithm is linear $O(|V| + |E|)$.

```
Algorithm Basic greedy carpooling algorithm
Input: $G = (V, E), w : E \to \mathbb{R}^+, c : V \to \mathbb{N}, D \subseteq V$
1.  $H \leftarrow \emptyset$
2.  sort $E$ by their weights in decreasing order
3.  for each $e = (u, v) \in E$
4.      do if $d^+_H(u) = d^-_H(u) = 0 \wedge v \in D \wedge d^-_H(v) < c(v) - 1$
5.          then $H \leftarrow H \cup \{e\}$
6.  return $H$
```

```
Algorithm Greedy carpooling algorithm
Input: $G = (V, E), w : E \to \mathbb{R}, c : V \to \mathbb{N}, D \subseteq V$
1.  Initialize: Perform 'bucket sort' on the edge weights. Sort the vertex set $V$ so that vertices not in $D$ precede
    vertices in $D$, $H \leftarrow \emptyset$, $U \leftarrow \emptyset$
2.  for all buckets in decreasing order
3.      for all $u \in V$
4.          do if $d^-_H(u) = d^+_H(u) = 0$ {i.e. $u$ is not assigned to any car}
5.              then choose $v \in N^+_G(u)$: higher priority for $v$ with $0 < d^-_H < c(v) - 1$, else prefer $v$ with
                    $d^-_H(v) = d^+_H(v) = 0$ and higher capacity
6.                  if such $v$ exists
7.                      then $H \leftarrow H \cup \{(u, v)\}$; update $d^+_H(u), d^-_H(v)$
8.                          $V \leftarrow V \setminus \{u\}$
9.                      else {no such v exists}
10.                         if the current bucket is the last one
11.                             then $U \leftarrow U \cup \{u\}$
12. return $H, U$
```

Practical results: The data was generated by FEATHERS, an activity based model simulation system, for the region of Flanders in Belgium[7]. We compared the Basic greedy algorithm to the Greedy algorithm using 2,4,8 and 16

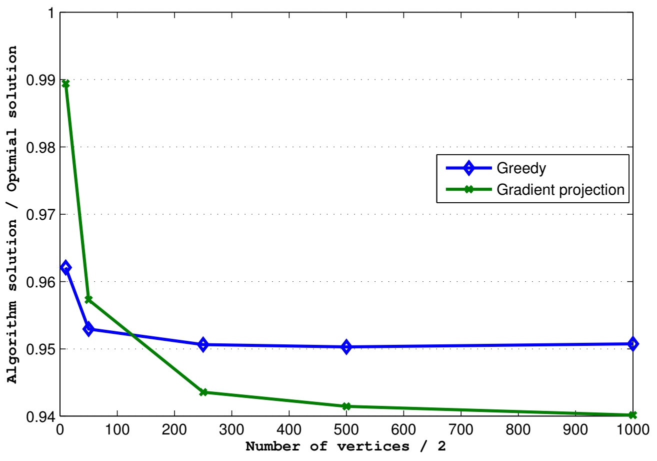

Fig. 3. Solutions quality of the gradient projection algorithm and greedy algorithm compared to the optimal solution for maximum weight matching in bipartite graphs

buckets. The results, presented in Table 1, show that the Greedy carpooling algorithm outperforms the Basic greedy carpooling algorithm in all three parameters of running time, solution weight, and the number of drivers. We also see that using 8 or 16 buckets is faster than using 2 or 4 buckets. This may happen due to the fact that choosing, in line 5, $v \in N_G^+(u)$ in the current bucket is faster when the bucket is smaller, i.e. when more buckets are used, and once v is chosen, vertex u is deleted together with all edges incident to it.

Algorithm	Running time	Solution weight	Number of drivers
Basic greedy	3.588	39627.3	177577
2-Buckets	2.215	40450.4	174011
4-Buckets	2.168	40450.4	174011
8-Buckets	1.467	40596.8	175383
16-Buckets	1.482	40550.1	175742

Table 1. The running time and solution quality of the Greedy and the Basic greedy algorithms

5. More carpooling problems

1. **Carpooling with aversion**: This is an extension of Problem 3.1, where in addition, we are given an *aversion function* $a : V \setminus R \times V \setminus R \rightarrow [0, 1]$ which signifies the amount of aversion passengers have to each other, in other words, passengers who do not wish to share a car ride. Can we find an assignment of passengers to a given set of vehicles by not allowing passengers with high aversion factor to share the same vehicle? We have shown that this extended problem is NP-hard. The reduction is from the minimum vertex coloring problem.
2. **Carpooling with attachment:** Same as the problem above, but here we have an *attachment function* which signifies the amount of attachment passengers have to each other, in other words, passengers who prefer to share a car ride, or insist on sharing a car ride (so they can work together, etc.) We have shown that this problem is NP-hard by reduction from the knapsack problem.
3. **Carpooling with VIP passengers:** Same as Problem 3.1 above, but in addition, some passengers insist on being alone in the car (except for the driver), or with no more than a given number of passengers. This problem is also NP-hard by reduction from the knapsack problem. We remark that this problem is equivalent to the carpooling with attachment problem since we can identify passengers who wish to be together as a unique passenger, who takes two seats.

6. Conclusion

We have defined the carpooling problem as a graph-theoretic, NP-hard problem. If the drivers of the cars are known in advance then the problem is tractable and is of complexity $O(|V|^3)$. We found that the greedy linear algorithm gives close to optimal results in this case. We have also found and implemented quick and efficient incremental solutions for a 'perturbed' graph, where some of its edges change. In addition, we found and implemented a fast heuristic algorithm, based on an algebraic approach, which can be parallelized for very large graphs. We suggested and implemented on real data several heuristics for the general intractable problem using linear and almost linear heuristics, and compared between them. Finally, we defined extensions of the carpooling problems where the drivers are known, by allowing the passengers to indicate their priorities, and showed that these extensions are NP-hard.

Acknowledgments
The research leading to these results has received funding from the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement no 270833.

References

1. Y. Chen, X. Ye, *Projection Onto A Simplex*, arXiv:1101.6081, January 2011.
2. E. Ferrari, R.Manzini, A. Pareschi, A. Persona, and A. Regattieri, *The Car Pooling Problem: Heuristic Algorithms Based on Savings Functions*, Journal of Advanced Transportation, 37(3), (2003), 243–272.
3. M.L. Fredman and R.E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM, 34(3), (1987), 596–615.
4. Z. Galil, S. Micali, and H. Gabow, *An $O(EV \log V)$ Algorithm for Finding a Maximal Weighted Matching in General Graphs*, SIAM journal of computing, 15(1),(1986), 120–130.
5. I. B-A. Hartman, *Note: Optimal assignment for carpooling* - submitted for publication.
6. J.E. Hopcroft and R.M. Karp, *An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs*, SIAM journal of computing, 2(4), (1973), 225–231.
7. L. Knapen, I. B-A. Hartman, D. Keren, A. Yasar, S. Cho, T. Bellemans, D. Janssens and G. Wets, *Estimating scalability issues while finding an optimal assignment for carpooling*, The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013).
8. L. Knapen, A. Yasar, S. Cho, D. Keren, A. Abu Dbai, T. Bellemans, D. Janssens, G. Wets, A. Schuster, I. Sharfman, and K. Bhaduri, *Exploring Graph-theoretic Tools for Matching in Carpooling Applications*, Journal of Ambient Intelligence and Humanized Computing (to appear).
9. Harold W. Kuhn, *Variants of the Hungarian method for assignment problems*, Naval Research Logistics Quarterly, 3, (1956), 253–258.
10. N. Maculan and G. Galdino, *A linear-time median-finding algorithm for projecting a vector on the simplex of $\mathbb{R}^n$*, Operations Research Letters, 8(4), (1989), 219–222.
11. J. Munkres, *Algorithms for the Assignment and Transportation Problems*, Journal of the Society for Industrial and Applied Mathematics, 5(1), (1957), 32–38.
12. S. Micali and V. Vazirani, *An $O(V^{1/2}E)$ algorithm for finding maximum matching in general graphs*, 21st Annual Symposium on Foundations of Computer Science,. IEEE Computer Society Press, New York. (1980), 17–27.