Computers and Operations Research 87 (2017) 107–113

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

Computational comparison of several greedy algorithms for the minimum cost perfect matching problem on large graphs

Sanne Wøhlk[a,*], Gilbert Laporte[b]

[a] CORAL - Cluster for Operations Research and Logistics, Department of Economics and Business Economics, Aarhus University, Fuglesangs allé 4, DK-8210 Aarhus V, Denmark

[b] GERAD and Canada Research Chair in Distribution Management, HEC Montréal, 3000, chemin de la Côte-Sainte-Catherine, Montréal, H3T 2A7, Canada

ARTICLE INFO

Article history:
Received 13 February 2017
Revised 30 May 2017
Accepted 5 June 2017
Available online 6 June 2017

Keywords:
Perfect matching problem
Blossom algorithm
Capacitated arc routing problem
Rural postman problem
Garbage collection

ABSTRACT

The aim of this paper is to computationally compare several algorithms for the Minimum Cost Perfect Matching Problem on an undirected complete graph. Our work is motivated by the need to solve large instances of the Capacitated Arc Routing Problem (CARP) arising in the optimization of garbage collection in Denmark. Common heuristics for the CARP involve the optimal matching of the odd-degree nodes of a graph. The algorithms used in the comparison include the CPLEX solution of an exact formulation, the LEDA matching algorithm, a recent implementation of the Blossom algorithm, as well as six constructive heuristics. Our results show that two of the constructive heuristics consistently exhibit the best behavior compared with the other four.

1. Introduction

The purpose of this paper is to provide a computational comparison of several algorithms for the Minimum Cost Perfect Matching Problem in complete graphs, simply referred to as the Matching Problem in what follows. The problem is defined on an undirected complete graph $G(N, E)$, where N is the set of n nodes (n is even) and E is the set of edges $e = (i, j)$, where $i, j \in N$ and $i < j$. For each node $i \in N$, we denote by $\delta(i)$ the set of edges incident to i. Let $c_e = c_{ij}$ be a non-negative weight associated with edge $e = (i, j)$. In practice this weight represents an edge length or a travel cost along it. Given a node i, all other nodes are called neighbors of i and we refer to the node closest to i as its nearest neighbor.

A perfect matching is defined as a set of edges $M \subset E$ for which $|M \cap \delta(i)| = 1$ for all $i \in N$. We denote by $c(M) = \sum_{e \in M} c_e$ the cost of matching M. The Matching Problem is to determine a minimum cost perfect matching M^* in G.

To formally describe the Matching Problem, we define binary variables x_e equal to 1 if and only if edge e belongs to the matching. The problem is then to

$$\text{minimize} \sum_{e \in E} c_e x_e$$

$$\text{subject to} \sum_{e \in \delta(i)} x_e = 1 \ \forall i \in N \tag{1}$$

$$x_e \in \{0, 1\} \ \forall e \in E. \tag{2}$$

Our work is motivated by the need to solve large-scale instances of the Capacitated Arc Routing Problem (CARP) arising in the optimization of garbage collection routes in Denmark. An effective heuristic for the CARP is to first solve a Rural Postman Problem (RPP) by ignoring the vehicle capacity constraints, and then cut the RPP solution into feasible routes as was done, for example in Hertz et al. (2000) and Prins et al. (2009). By partitioning the RPP solution optimally, an approximation algorithm with fixed ratio can be obtained for the CARP (Wøhlk, 2008). A good heuristic for the RPP consists of first identifying connected components of required edges (those with a positive demand), connecting these components by means of a minimum cost spanning tree, and then matching the odd-degree nodes of the graph induced by the connected components and the spanning tree (Frederickson, 1979).

The Matching Problem can be solved exactly by CPLEX, by LEDA's matching algorithm Algorithmic Solutions Software GmbH (2017), and by the Blossom algorithm (Edmonds, 1965). The most recent publicly available implementations of the latter are Cook and Rohe (1999) and Kolmogorov (2009). The matching problem has also been investigated from an approximation point of view

* Corresponding author.
E-mail addresses: sanw@econ.au.dk (S. Wøhlk), gilbert.laporte@cirrelt.ca (G. Laporte).

http://dx.doi.org/10.1016/j.cor.2017.06.006
0305-0548/

and the most recent algorithm has a worst-case performance ratio of $c(M)/c(M^*) \leq \log^2(n)$ (Wattenhofer and Wattenhofer, 2004). Any of the exact algorithms can be time consuming. In some large-scale applications, such as ours, it is desirable to compute high quality solutions within short computing times. We have therefore implemented several greedy heuristics that meet this requirement. In this spirit, we did not develop post-optimization procedures which would typically result in higher time complexity. We have compared the heuristics between themselves and with the exact algorithms. Our heuristics are described in Section 2, followed by comparative computational results in Section 3. Conclusions follow in Section 4.

2. Matching heuristics

We now outline the algorithms we have used in our analysis. All algorithms start with an empty matching M and iteratively add a single edge to M until all nodes are matched. Throughout this section, we use S to denote the set of unmatched nodes. When multiple nodes satisfy the selection criteria, ties are broken be always selecting the node or edge that occurs first in the list of nodes or edges.

In the *Greedy* heuristic, we add to the matching an edge of minimum cost that connects two unmatched nodes and repeat this until all nodes are matched. This is detailed in Algorithm 1. The time complexity of *Greedy* is $\mathcal{O}(n^3)$ in our implementation. It can be reduced to $\mathcal{O}(n^2 \log n)$ by using a priority queue.

Algorithm 1 Greedy.

Set $S = N$, $M = \emptyset$
while $S \neq \emptyset$ **do**
  Select $(i, j) \in \arg\min\{c_{i'j'} | i', j' \in S, i' \neq j'\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
end while
return M

The intuition behind the *Largest* heuristic is to first identify a match for the nodes that are most isolated in their location. To this aim, we first identify for every node i the distance $H(i)$ to the nearest neighbor, and we then consider the nodes in non-increasing order with respect to $H(i)$. If a node is not already matched, it will be matched to its nearest unmatched neighbor and the process is reiterated until all nodes are matched. *Largest* is outlined in Algorithm 2 and has a time complexity of $\mathcal{O}(n^2)$.

Algorithm 2 Largest.

Set $S = N$, $M = \emptyset$
Set $H(i) = \min\{c_{ij} | j \in N, j \neq i\}$, $\forall i \in N$
while $S \neq \emptyset$ **do**
  Select $i \in \arg\max\{H(i') | i' \in S\}$
  Select $j \in \arg\min\{c_{ij'} | j' \in S, j' \neq i\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
end while
return M

The heuristic *Largest** is a modified version of *Largest* in which the values of $H(i)$ are updated at each iteration, in such a way that the node that is furthest away from its nearest unmatched neighbor can be chosen as i at each iteration. Note that by storing additional information regarding the nodes resulting in the $H(i)$-values, this update needs only be performed for a subset of the nodes. However, in the worst case, the $H(i)$-value must be updated for all $i \in S$. This results in a worst-case time complexity of $\mathcal{O}(n^3)$ in our implementation.

In the *Sum* heuristic, the nodes i are considered in non-increasing order with respect to the sum $\Delta(i)$ of their distance to all other nodes. If a node is unmatched, it will be matched to the nearest unmatched neighbor. This is illustrated in Algorithm 3 which has a time complexity of $\mathcal{O}(n^2)$.

Algorithm 3 Sum.

Set $S = N$, $M = \emptyset$
Calculate $\Delta(i) = \sum_{e \in \delta(i)} c_e$, $\forall i \in N$
while $S \neq \emptyset$ **do**
  Select $i \in \arg\max\{\Delta(i') | i' \in S\}$
  Select $j \in \arg\min\{c_{ij'} | j' \in S, j' \neq i\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
end while
return M

*Sum** is a modified version of *Sum* in which the values of $\Delta(i)$ are repeatedly updated to reflect the sum of the distances to all the unmatched neighbors and the next node to be matched to its nearest unmatched neighbor is the one where this sum is highest. The update adds to the run time, but does not increase the worst-case time complexity which remains $\mathcal{O}(n^2)$.

The *Regret* heuristic outlined in Algorithm 4 requires some notation. For every node $i \in S$, let $m_1(i)$ denote the nearest neighbor and let $m_2(i)$ denote the second nearest, i.e. $m_1(i) \in \arg\min\{c_{ij} | j \in S, j \neq i\}$ and $m_2(i) \in \arg\min\{c_{ij} | j \in S,\ j \neq m_1(i), j \neq i\}$. The regret value of i is then defined as $R(i) = c_{im_{2(i)}} - c_{im_{1(i)}}$ and reflects the extra cost incurred if i is not matched to its nearest neighbor.

Algorithm 4 Regret.

Set $S = N$, $M = \emptyset$
Calculate $R(i)$, $\forall i \in N$
while $S \neq \emptyset$ **do**
  Select $i \in \arg\max\{R(i') | i' \in S\}$
  Set $j = m_1(i)$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
  Update $R(k)$, $\forall k \in S$ with $m_1(k) \in \{i, j\}$ or $m_2(k) \in \{i, j\}$
end while
return M

The intuition behind *Regret* is to give priority to those nodes that we would most regret not to match to their nearest neighbors and to try and match those nodes in the best possible way. Therefore, we repeatedly identify the unmatched node with highest regret value and match it to its nearest unmatched neighbor. During this process, the regret values are updated to only take unmatched nodes into account. As for *Largest**, the update can be time consuming in the worst case, but in practice we can decrease the time for this update significantly by storing $m_1(k)$ and $m_2(k)$ and only update $R(k)$ if one of these are one of the nodes just matched. The worst-case time complexity of *Regret* is $\mathcal{O}(n^3)$.

3. Results and analyses

We now describe our test instances and we provide extensive computational comparisons of our algorithms.

3.1. Test instances

We have used two sets of data in our analysis. The first set is based on the 88 real-life graphs presented in Kiilerich and Wøhlk

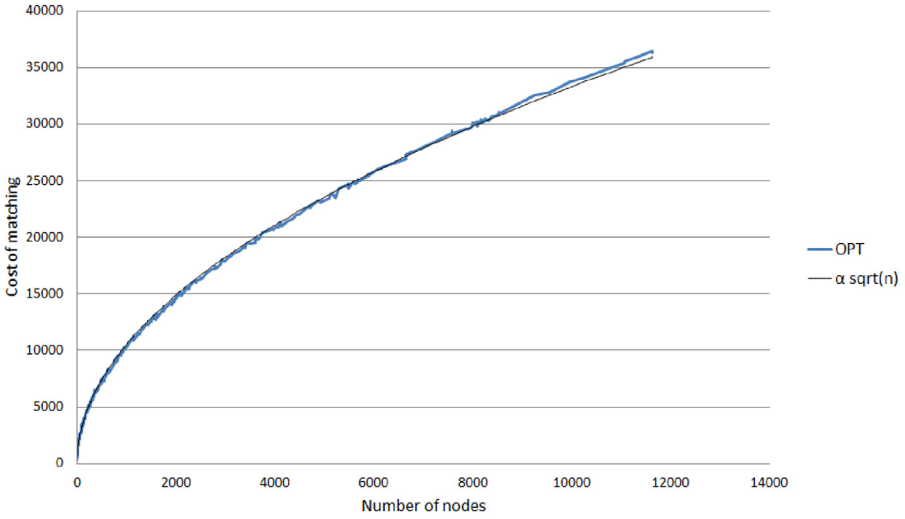

Fig. 1. Optimal costs of matchings for random graphs.

(2017). These graphs were originally designed for arc routing problems. For each graph, we have created four matching graphs by selecting a subset of the nodes using different criteria. A complete graph is then created for matching purposes by using shortest path distances between the selected nodes in the original graphs. In the first graph, all nodes are selected (if the number of nodes is odd, we leave out the one designated as "the depot" in the data). In the second graph, all odd-degree nodes are selected, whereas the odd-degree nodes with respect to the required edges are used in the third graph. Finally, in the fourth graph, only nodes of degree one are selected (in case of an odd number, the one listed first is left out). This procedure yields a total of 352 graphs that we will call real graphs. The number of nodes range from two to 11640.

The second set contains 352 randomly generated graphs with the same number of nodes as those in the first set. For each of the graphs, the nodes are randomly generated according to a discrete uniform distribution in the 1000 × 1000 square. Based on these nodes, a complete graph is generated with Euclidean distances between each pair of nodes, rounded up to the nearest integer. These graphs are referred to as random graphs.

The algorithms were implemented in C++ and executed on an Intel Xeon CPU with 12 cores running at 3.5 GHz and 64GBs RAM. We used CPLEX 12.6.1 with standard settings and allowed it to take advantage of the parallel processors. All other algorithms were executed sequentially. For the Blossom algorithm, we used the freely available Blossom V implementation (Kolmogorov, 2009), Kolmogorov. We also used LEDA 6.5 Algorithmic Solutions Software GmbH (2017).

We have only run CPLEX on the 268 graphs of each set having less than 4000 nodes, with a time limit of one hour. Within this time limit, CPLEX could find an optimal matching for 148 real graphs and for 236 random graphs.

Blossom V crashed for graphs with $n > 4000$. We contacted the author of this algorithm, but neither he nor we have been able to find the explanation for this unintended behavior, but we believe it regards the interaction between MS Visual Studio and the algorithm. It was therefore only run on graphs with fewer than 4000 nodes. On these graphs, Blossom V successfully obtained an optimal matching for 233 real graphs and for all 268 random graphs.

3.2. The optimal cost of matchings

We first consider the cost of optimal matchings for the randomly generated graphs. Fig. 1 depicts the cost of the optimal matchings as a function of n and indicates a clear relationship between n and matching cost $f(n)$. We have found the empirical relationship

$$f(n) = \frac{l\sqrt{n}}{3},$$

where $l = 1000$ is the size of the $l \times l$-square in which we have generated the nodes. The determination coefficient is $R^2 = 99.95\%$. This estimate is consistent with the result presented in Cook and Rohe (1999) and Papadimitriou (1977). It is also interesting to observe that the solution cost is proportional to $\sqrt{n}$, as in the Beardwood-Halton-Hammersley approximation formula for the Traveling Salesman Problem (Beardwood et al., 1959).

In Fig. 2, we show the optimal matching costs of the real graphs as a function of n. The figure indicates that no clear relationship exists for the optimal matching cost for real graphs.

3.3. Cost analysis of heuristics

We now investigate the six heuristics as regards their ability to obtain low cost matchings.

Fig. 3 shows the cost obtained by each of the heuristics for the 352 random graphs as well as the optimal cost. Note that the curve for *Sum* is partially hidden behind the curve for *Sum**. This figure suggests that the two heuristics *Largest* and *Largest** are outperformed by the other heuristics. We also note that *Sum* and *Sum** perform equally well and appear to perform better than *Greedy* and *Regret* on random graphs.

To further analyze the performance of the heuristics, we show in Fig. 4 the relative cost obtained by each heuristic as a percentage above the optimal cost. The left-hand side of the figure depicts the results for real graphs, whereas the right-hand side plots the results for random graphs.

To support the figure, Tables 1–4 provide the mean percent above the best (Table 1 for real graphs and Table 2 for random graphs) and the standard deviation of the percentage (Table 3 for real graphs and Table 4 for random graphs), both for the full set of graphs and for different size groups. The first column in each table states the range of n in each group and the second column gives the number of graphs in the group.

Consider the top part of Fig. 4 again. The first result to notice is how stable the performance of the heuristics is for random graphs compared to the real graphs. This is supported by Tables 3 and 4 in which the standard deviations are reported. As an example, the standard deviation for *Largest** is 22.2% for real graphs compared to 15.8% for random graphs. For both types of graphs, the

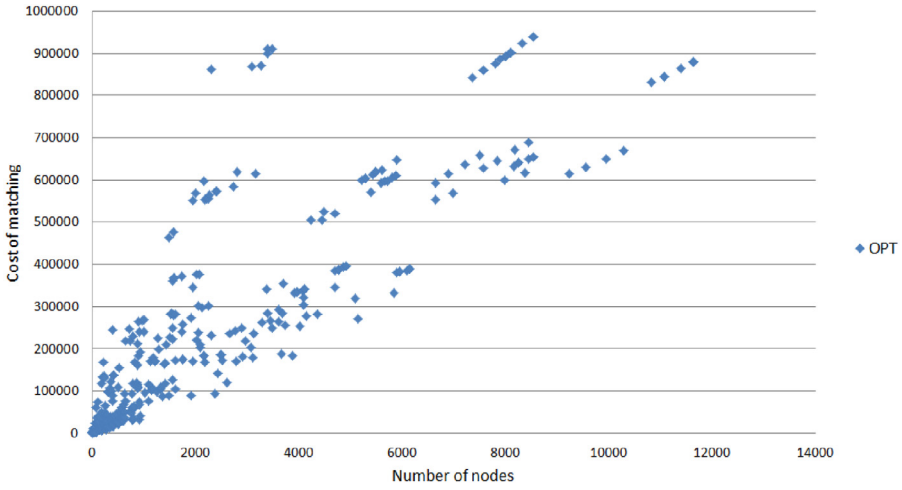

Fig. 2. Optimal costs of matchings for real graphs.

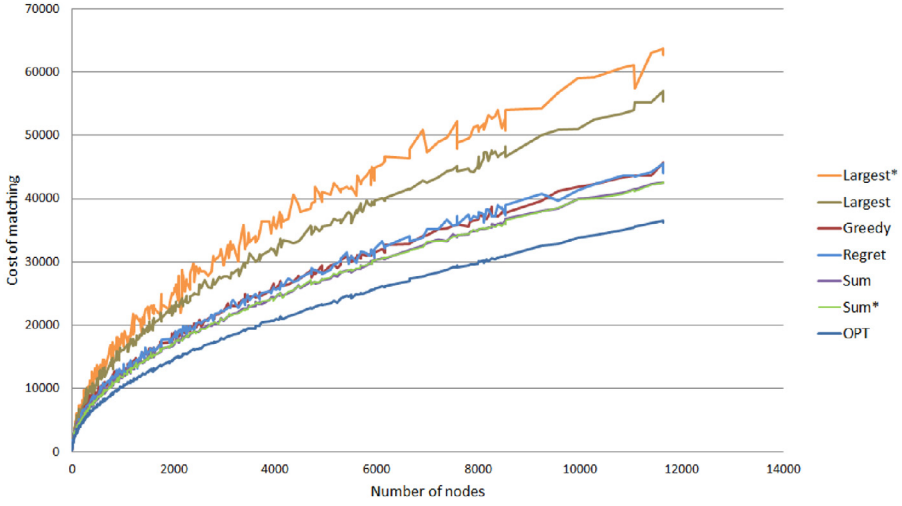

Fig. 3. Optimal costs and costs obtained by the six heuristics for random graphs.

results vary more for small graphs than they do for the larger ones. Again, this is more pronounced for random graphs, where the standard deviation of *Largest** is 23.3% for $n < 500$, but only 4.3% for $n > 6000$. The corresponding values for real graphs are 23.8% and 14.6%, respectively.

We now compare the performance of the six heuristics. When we consider the top part of Fig. 4, it is clear that the two heuristics *Largest** and *Largest* perform significantly worse than the other heuristics with a mean of 61.2% (67.8%) and 45.9% (51.8%), respectively, for all real (random) graphs. Both *Greedy* and *Regret* show an average performance with a mean of 28.0% (24.6%) and 25.7% (23.0%), respectively, for all real (random) graphs. The two heuristics that show the best performance are *Sum* and *Sum**, both having a mean of 13.6% for real graphs and of 16.5% and 16.1%, respectively, for random graphs. The lower part of Fig. 4 provides the same plot as the top part, but only shows *Sum* and *Sum**. The plot clearly indicates that the two heuristics do indeed perform equally well and no clear winner among the two can be selected solely based on matching cost. We counted the number of times a heuristic was the best (1), worst (6), and so on. These results are shown in Fig. 5 and further support the relative ranking. Even though this figure indicates that *Sum** is superior to *Sum* as it outperforms the other heuristics more often, Tables 1–4 show that it is only marginally better when considering the average performance and standard deviation. Indeed, Sum is 13.6 (16.5) percent above the optimal cost for real (random) graphs on average. The corresponding number for *Sum** is 13.6 (16.1) percent, confirming that we cannot identify one of these as the preferred one.

3.4. Run time analysis of the heuristics

Finally, we consider the run time of the six heuristics as well as the Blossom V algorithm, LEDA, and CPLEX, which solve the problem optimally. Since no significant differences in run time between real graphs and random graphs are observed, we only show the plots for real graphs. This is done in Figs. 6–9, where run times in seconds are shown as a function of n.

In Fig. 6, we show all nine algorithms run on all graphs. It is clear from this plot that the run time for CPLEX rapidly increases and in fact, CPLEX often fails to solve the problem within the time limit of one hour. CPLEX could solve 148 graphs with $n < 4000$ (out of 268) to optimality within the time limit. For random graphs, 236 graphs were solved by CPLEX. This means that for practical purposes CPLEX is not the best tool for solving the Matching Problem. The remaining plots exclude CPLEX.

In Fig. 7, we zoom in on the graphs with $n < 4000$. Here we clearly observe that even for relatively large graphs, the Blossom algorithm is quite fast and with this advanced implementation of a very complicated algorithm its run time is indeed very impressive and this algorithm is clearly preferred over CPLEX for solving

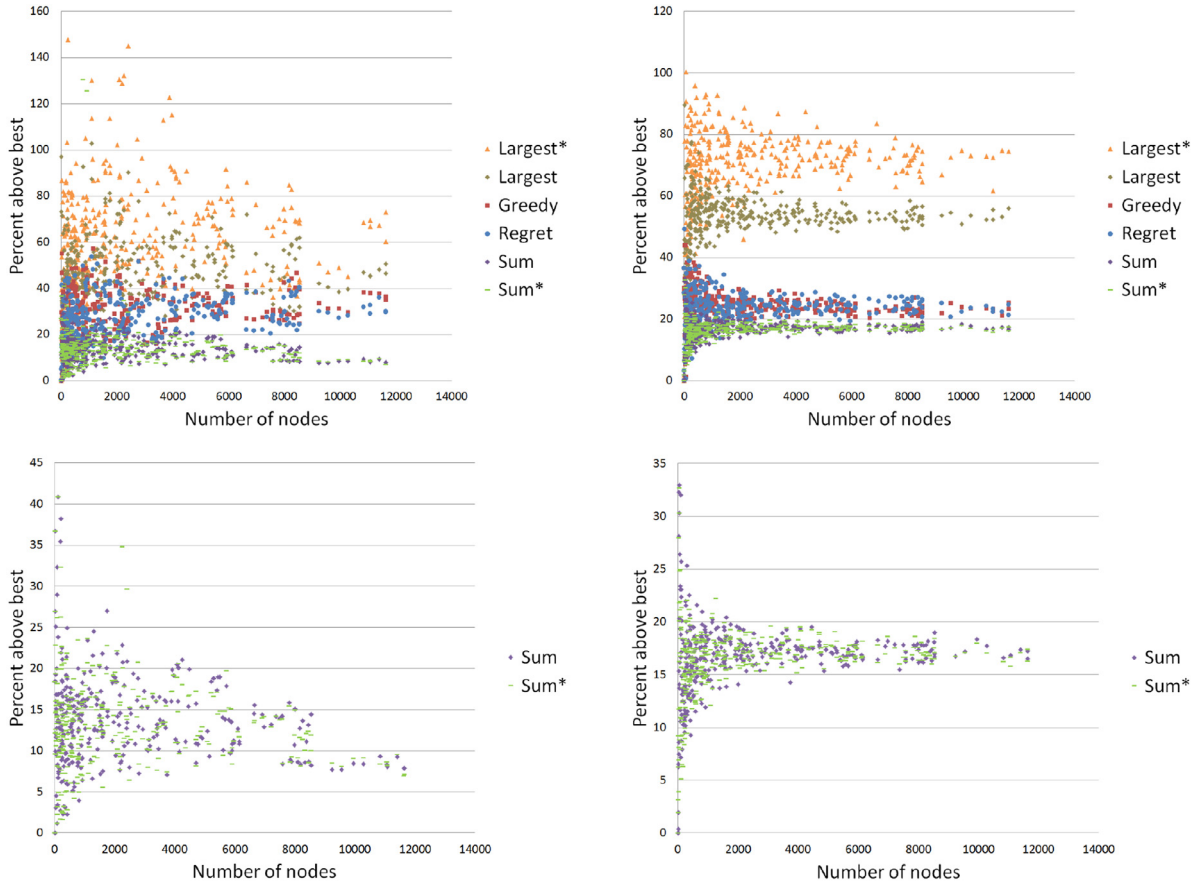

Fig. 4. Cost obtained by the heuristics displayed as percent above the optimum. Left: real graphs. Right: random graphs. Top: all heuristics. Bottom: only the best two heuristics.

Table 1
Mean percent above the optimal for real graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
0–500	104	51.7	39.8	22.0	19.6	13.7	12.4
500–1000	55	54.4	42.9	28.3	24.8	12.7	16.2
1000–2000	49	70.2	52.9	29.8	27.8	15.3	14.5
2000–4000	60	72.4	50.3	29.5	28.0	14.3	14.3
4000–6000	41	68.5	51.0	32.9	32.0	14.2	13.9
> 6000	43	60.0	45.2	33.4	30.2	11.2	11.0
All	352	61.2	45.9	28.0	25.7	13.6	13.6

Table 2
Mean percent above the optimal for random graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
0-500	104	57.1	46.1	25.1	20.5	15.0	14.3
500-1000	55	71.4	55.1	25.7	24.3	16.7	16.1
1000-2000	49	72.4	54.6	24.3	23.3	17.3	17.1
2000-4000	60	72.6	54.2	24.4	24.2	17.3	17.2
4000-6000	41	73.0	53.5	23.9	24.2	17.2	17.4
> 6000	43	71.9	53.2	23.0	24.2	17.2	17.0
All	352	67.8	51.8	24.6	23.0	16.5	16.1

Table 3
Standard deviation of percent above the optimal for real graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
0–500	104	23.8	15.5	12.0	10.7	8.0	7.4
500–1000	55	16.5	12.9	10.2	9.6	3.8	22.1
1000–2000	49	19.6	14.7	9.7	10.5	4.4	3.9
2000–4000	60	25.3	13.5	8.6	8.3	3.9	5.0
4000–6000	41	13.1	9.2	4.4	4.8	3.8	3.3
> 6000	43	14.6	10.7	6.0	6.0	2.8	2.4
All	352	22.2	14.4	10.4	10.1	5.5	10.2

Table 4
Standard deviation of percent above the optimal for random graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
0–500	104	23.3	17.5	8.1	9.8	6.6	6.2
500–1000	55	10.4	6.2	4.2	4.9	2.3	1.9
1000–2000	49	8.7	4.1	2.7	4.1	1.7	1.8
2000–4000	60	7.4	3.0	2.2	2.5	1.1	1.1
4000–6000	41	4.7	2.5	1.5	2.1	0.8	0.9
> 6000	43	4.3	1.9	1.3	1.9	0.8	0.7
All	352	15.8	10.7	5.0	6.3	3.9	3.8

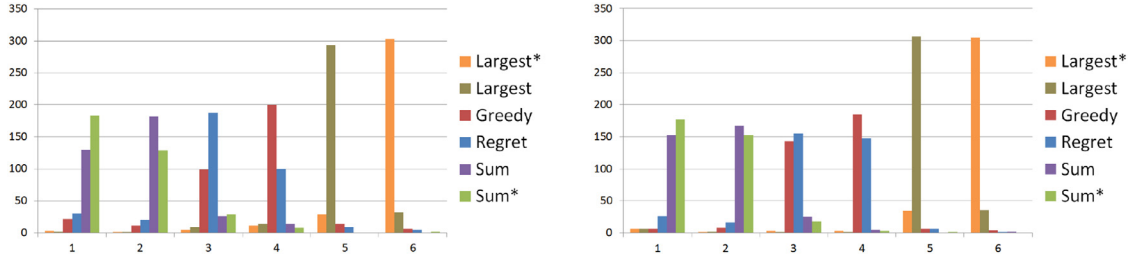

Fig. 5. Relative ranks of the heuristics. Left: Real graphs. Right: Random graphs.

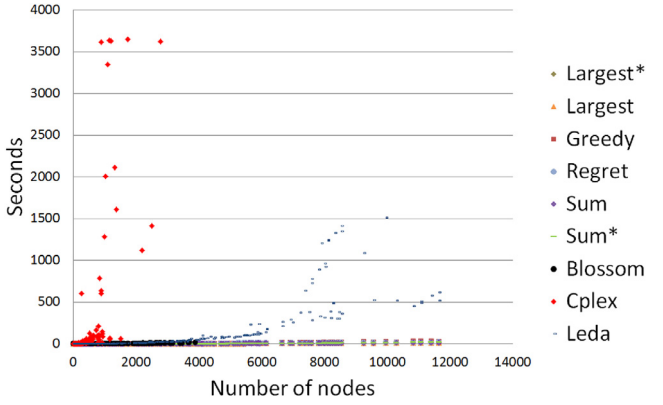

Fig. 6. Run time for real graphs. All graphs and all algorithms.

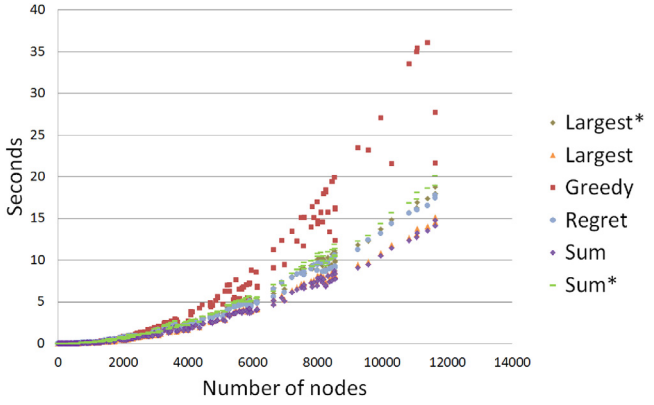

Fig. 8. Run time for real graphs. All graphs. The six heuristics.

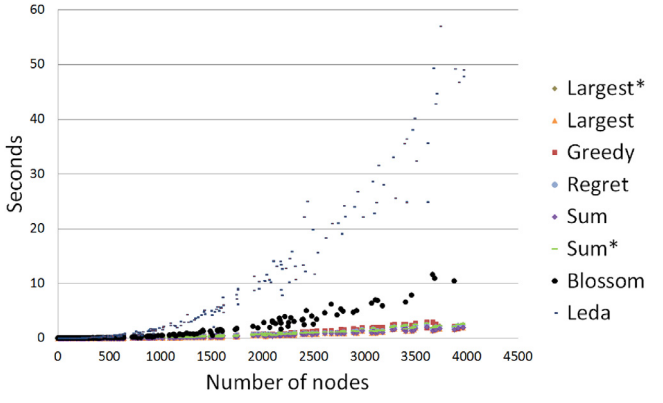

Fig. 7. Run time for real graphs. Zoom on the graphs with $n < 4000$. All except CPLEX.

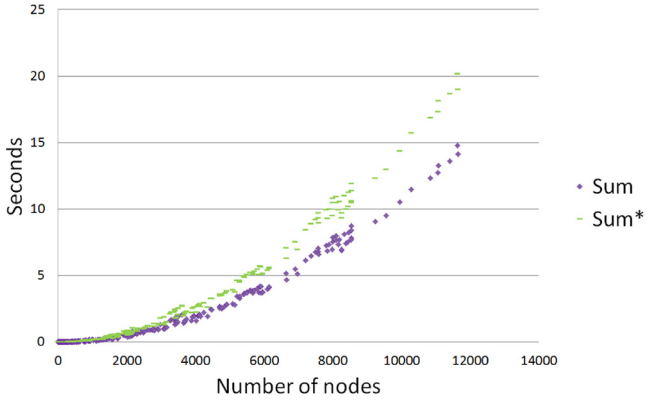

Fig. 9. Run time for real graphs. All graphs. Only *Sum* and *Sum**.

matching problems with more than about 1000 nodes. Blossom V if significantly faster than the LEDA algorithm. We note that the longest run time observed for Blossom is 12 seconds, whereas the longest time for LEDA on these graphs is 57 seconds. However, we also see that the run time of Blossom tends to grow faster than that of the heuristics, hinting that Blossom may not be a realistic choice for large graphs even if the code could be fixed. It is, however, the best option if optimal matchings are needed.

The relative run time of the heuristics is most easily observed in Fig. 8, which shows the runtime of the six heuristics for all graphs. This plot clearly shows that *Greedy* is slower than the other heuristics, particularly for large graphs. The two heuristics *Largest* and *Sum* show the best run time, while the remaining three are slightly slower. However, even the slowest of the five heuristics runs within 20 seconds on the largest graphs.

Finally, Fig. 9 shows the run time of *Sum* and *Sum**, the two heuristics that had the best performance in terms of solution quality. From this plot, we see that even though both show acceptable run times, *Sum* is indeed faster than the version with updates. In fact, *Sum* is the fastest of all the heuristics tested.

4. Conclusions

We have compared several algorithms for the solution of the Matching Problem arising, namely, in the solution of the Capacitated Arc Routing Problem. These include CPLEX applied to a standard integer linear programming model, the Kolmogorov implementation of the Blossom algorithm, and six new constructive heuristics. For graphs involving fewer than 4000 nodes, the Blossom algorithm outperforms CPLEX in terms of computing time. For larger instances, which cannot be solved optimally, the constructive heuristics called *Sum* and *Sum** consistently outperform the other four, but *Sum* is faster.

Acknowledgments

The authors thank Serge Bisaillon for his technical support. This project is funded by the Danish Council for Independent Research-Social Sciences. Project "Transportation issues related to waste management" [grant number 4182–00021] and by the Natural Sciences and Engineering Research Council of Canada [grant number 2015–06189]. This support is gratefully acknowledged. Thanks are due to the referees for their valuable comments.

References

Algorithmic, Solutions Software GmbH, 2017. http://www.algorithmic-solutions.com. Accessed 9 June 2017.
Beardwood, J., Halton, J.H., Hammersley, J.M., 1959. The shortest path through many points. Proc. Camb. Philos. Soc. 55, 299–327.
Cook, W., Rohe, A., 1999. Computing minimum-weight perfect matchings. INFORMS J. Comput. 11, 138–148.
Edmonds, J., 1965. Path, trees, and flowers. Can. J. Math. 17, 449–467.
Frederickson, G.N., 1979. Approximation algorithms for some postman problems. J. Assoc. Comput Mach 26, 538–554.
Hertz, A., Laporte, G., Mittaz, M., 2000. A tabu search heuristic for the capacited arc routing problem. Oper. Res. 48, 129–135.
Kiilerich, L., Wøhlk, S., 2017. New large-scale data instances for CARP and variations of CARP. INFOR: Inf. Syst. Oper. Res. Accepted http://dx.doi.org/10.1080/03155986.2017.1303960.
Kolmogorov, V., http://pub.ist.ac.at/~vnk/software.html#blossom5.
Kolmogorov, V., 2009. Blossom V: a new implementation of a minimum cost perfect matching algorithm. Math. Program. Comput. 1, 43–67.
Papadimitriou, C.H., 1977. The probabilistic analysis of matching heuristics. In: Proceedings of the 15th Annual Allerton Conference on Communication Control and Computing, pp. 368–378.
Prins, C., Labadi, N., Reghioui, M., 2009. Tour splitting algorithms for vehicle routing problems. Int. J. Prod. Res. 47, 507–536.
Wattenhofer, M., Wattenhofer, R., 2004. Fast and simple algorithms for weighted perfect matching. Electron. Notes Discrete Math 17, 285–291.
Wøhlk, S., 2008. An approximation algorithm for the capacitated arc routing problem. Open Oper. Res. J. 2, 8–12.