Les Cahiers du GERAD ISSN: 0711–2440

Computational comparison of several algorithms for the minimum cost perfect matching problem

S. Wøhlk
G. Laporte

G–2017–11

February 2017

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Computational comparison of several algorithms for the minimum cost perfect matching problem

Sanne Wøhlk [a]

Gilbert Laporte [b]

[a] *CORAL - Centre 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, Canada H3T 2A7*

sanw@econ.au.dk
gilbert.laporte@cirrelt.ca

February 2017

Les Cahiers du GERAD

G–2017–11

Abstract: The aim of this paper is to computationally compare several algorithms for the Minimum Cost Perfect Matching Problem on an undirected 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 computation of matchings of the odd-degree nodes of a graph. The algorithms used in the comparison include the CPLEX solution of an exact formulation, a recent implementation of the Blossom algorithm, as well as six constructive heuristics. The Blossom algorithm works well on graphs containing fewer than 4000 nodes and outperforms CPLEX in terms of computing time. For larger instances, we found that one of the constructive heuristics consistently exhibits the best behavior compared with the other five.

Keywords: Perfect Matching Problem, Blossom algorithm, Capacitated Arc Routing Problem, rural postman problem, garbage collection

Acknowledgments: 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.

1 Introduction

The purpose of this paper is to provide a computational comparison of several algorithms for the Minimum Cost Perfect Matching Problem, 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 complete 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 complete 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 [5] and [10]. By optimal partitioning of the RPP solution, an approximation algorithm with fixed ratio can be obtained for the CARP [12] 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 [4].

The Matching Problem can be solved exactly by CPLEX and by the Blossom algorithm [3]. The most resent publicly available implementations of the latter are [2, 8]. The matching problem has also been investigated from an approximation point of view and the most recent algorithm has a worst-case performance ratio of $c(M)/c(M^*) \leq \log^2(n)$ [11]. Either of the exact algorithms can be time consuming. In some 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 on the large instances, and with the exact algorithms when this was possible. 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.

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.

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

Algorithm 1 *Greedy*

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

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 $i = \arg\max\{H(i') | i' \in S\}$
  Select $j = \arg\min\{c_{ij'} | j' \in S, j' \neq i\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
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. This is illustrated in Algorithm 3. 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.

Algorithm 3 *Largest**

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

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 4 which has a time complexity of $\mathcal{O}(n^2)$.

*Sum**, which is outlined in Algorithm 5, 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 6 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. $m_1(i) = \arg\min\{c_{ij} | j \in S, j \neq i\}$ and $m_2(i) = \arg\min\{c_{ij} | j \in S,\ j \neq m_1(i), j \neq i\}$. 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 *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 $i = \arg\max\{\Delta(i') | i' \in S\}$
  Select $j = \arg\min\{c_{ij'} | j' \in S, j' \neq i\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
end while
return M

Algorithm 5 *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 $i = \arg\max\{\Delta(i') | i' \in S\}$
  Select $j = \arg\min\{c_{ij'} | j' \in S, j' \neq i\}$
  $M = M \cup \{(i, j)\}$
  $S = S \setminus \{i, j\}$
  Update $\Delta(k)$, $\forall k \in S$
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 posible 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 $\mathcal{O}(n^3)$.

Algorithm 6 *Regret*

Set $S = N$, $M = \emptyset$
Calculate $R(i)$, $\forall i \in N$
while $S \neq \emptyset$ **do**
  Select $i = \arg\max\{R(i') | i' \in S\}$
  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

3 Results and analysis

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 [6]. 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 set one. 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 64 GBs 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 [8, 7].

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.

The 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. It was therefore only run on graphs with fewer than 4000 nodes. On these graphs, it successfully obtained an optimal matching for 233 real graphs and for all 268 random graphs.

3.2 The optimal cost of matchings

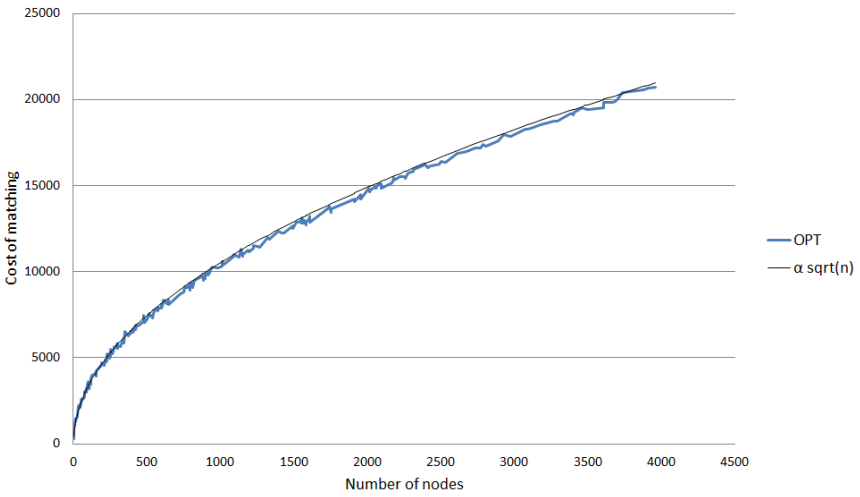

Figure 1: Optimal costs of matchings for random graphs.

We first consider the cost of optimal matchings for the randomly generated graphs. All 268 random graphs with $n < 4000$ were solved to optimality. Figure 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 $R^2 = 99.8\%$. This estimate is consistent with the result presented in [9, 2] and with the approximation formula of [1] for the Traveling Salesman Problem.

In Figure 2, we show the optimal matching costs of the 148 real graphs that could be solved to optimality within the time limit as a function of n. The figure indicates that no clear relationship exists for the optimal matching cost for real graphs.

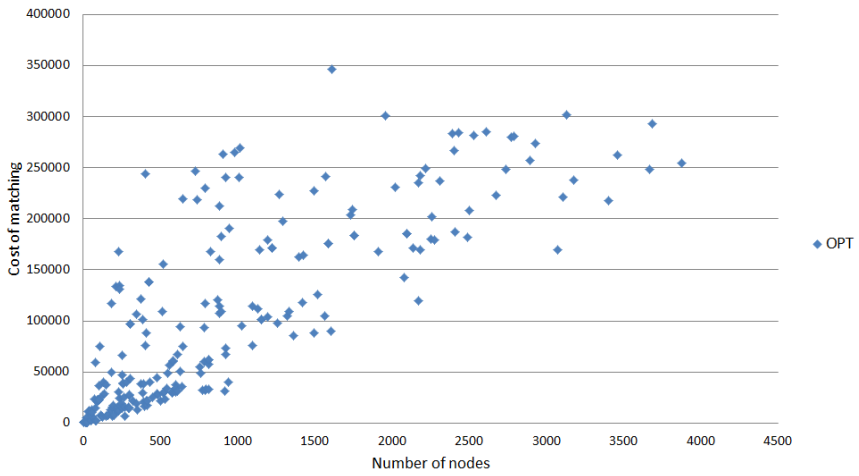

Figure 2: Optimal costs of matchings for real graphs.

3.3 Cost analysis of heuristics

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

Figure 3 shows the cost obtained by each of the heuristics for the 352 random graphs as well as the optimal cost for those with $n < 4000$. 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.

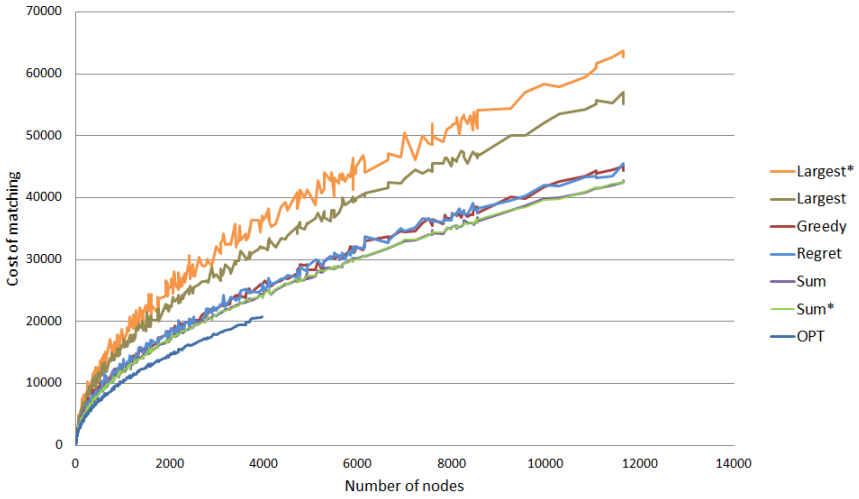

Figure 3: Optimal costs and costs obtained by the six heuristics for random graphs.

To further analyze the performance of the heuristics, we show in Figure 4 the relative cost obtained by each heuristic as a percentage above the best cost obtained. 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 2 for random graphs) and the standard deviation of the percentage (Table 3 for real graphs and 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.

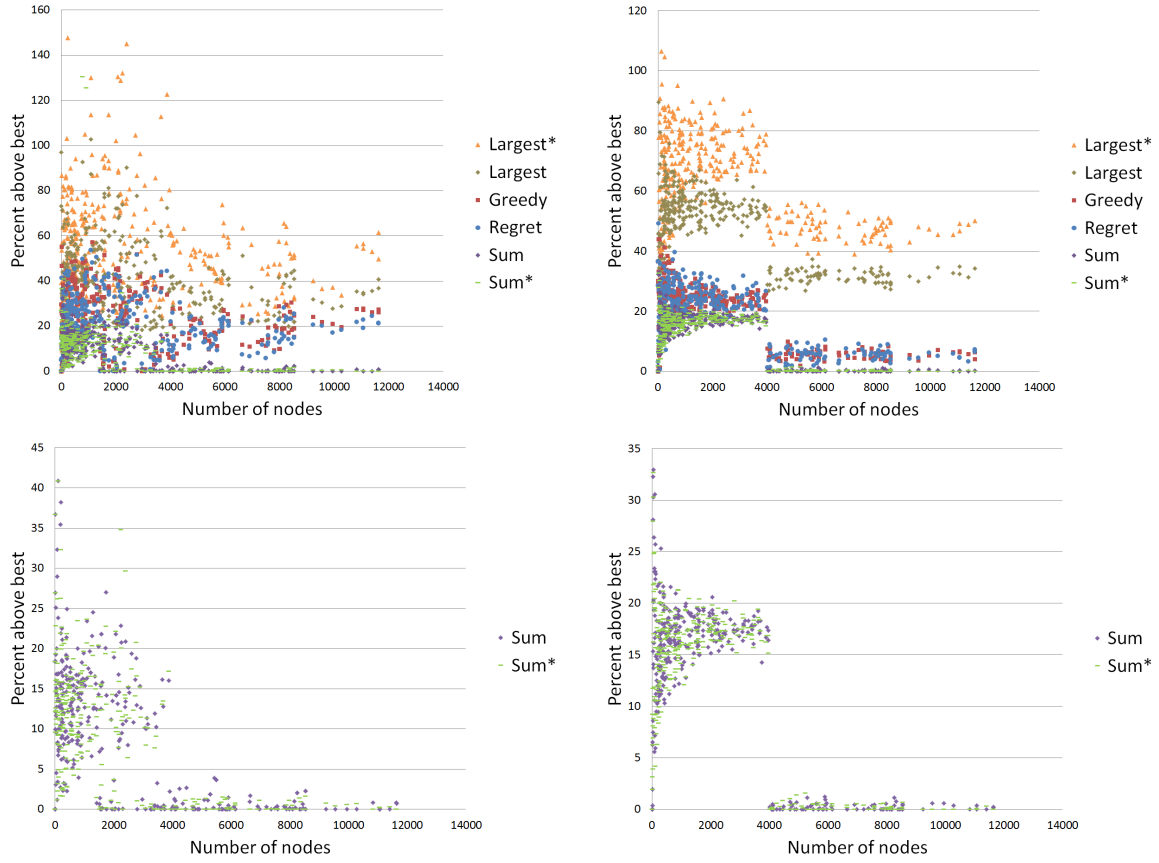

Figure 4: Cost obtained by the heuristics displayed as percent above the best. Left: Real graphs. Right: Random graphs. Top: All heuristics. Bottom: Only the best two heuristics.

Table 1: Mean percent above best known for real graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
All	352	55.0	40.3	23.1	20.9	9.3	9.3
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	65.1	48.3	26.1	24.0	11.7	10.9
2000-4000	60	64.7	43.4	23.7	22.1	8.8	8.8
4000-6000	41	48.5	33.1	17.3	16.5	0.7	0.4
> 6000	43	44.5	31.1	20.6	17.7	0.4	0.3

Note that when the optimal matching is known, the costs are relative to its value. However, when the optimal cost is not known, the costs are shown relative to the best heuristic result. This is particularly clear for random graphs where a shift occurs at $n = 4000$. A similar shift is also present for real graphs, but is less noticeable.

Because we observed a tight relationship between the optimal matching costs and the function $f(n) = l\sqrt{n}/3$ for random graphs in Section 3.2, we can use this function to estimate the optimal matching cost for graphs with $n > 4000$ and use it as a benchmark for evaluating the quality of the heuristics. This is done in the central part of Table 2. Consider, for instance, the graphs with $n > 6000$. For these graphs, the heuristic *Sum* is only 0.2% on average above the best result (because it often provides the best). However, when we compare with the estimated optimal cost, *Sum* only performs 19.8% on average above the optimal solution value. The shift listed in the last part of the table indicates the difference between these two numbers (adjusted for rounding) and corresponds to the size of the shift observed in Figure 4.

Table 2: Mean percent above best known for random graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
All	352	62.0	46.9	20.2	18.3	12.5	12.2
0-500	104	57.5	46.5	25.2	19.9	15.1	14.3
500-1000	55	70.7	55.5	25.3	23.6	16.9	16.2
1000-2000	49	72.4	55.4	24.6	24.8	17.2	17.1
2000-4000	60	73.6	54.3	23.9	23.5	17.4	17.4
4000-6000	41	48.6	31.7	5.7	5.4	0.3	0.3
> 6000	43	46.7	31.5	5.5	5.7	0.2	0.2
4000-6000	41	75.5	55.5	24.9	24.5	18.4	18.4
> 6000	43	75.5	57.3	26.2	26.4	19.8	19.8
The shift:							
4000-6000		26.9	23.8	19.1	19.1	18.2	18.2
> 6000		28.8	25.8	20.7	20.7	19.7	19.7

Consider the top part of Figure 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 22.8% for all real graphs compared to 17.8% for all random graphs. For both types of graphs, the 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 24.3% for $n < 500$, but only 3.3% for $n > 6000$. The corresponding values for real graphs are 23.8% and 12.2%, respectively. The last two lines of Table 4 provide the standard deviation of the shifted results, i.e. with performance relative to the estimated matching cost. Another way to look at the stable performance of the heuristics is to notice in Tables 1 and 2 how the mean of each heuristic only changes slightly when we consider different size groups. Alternatively, consider the shifts in Table 2, where the size of the shift for each heuristic is stable for the two size groups.

Table 3: Standard deviation of percent above best known for real graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
All	352	22.8	15.7	12.0	11.2	8.1	11.9
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	23.4	18.7	14.3	14.4	7.7	7.0
2000-4000	60	30.1	18.6	14.8	13.8	7.2	8.0
4000-6000	41	9.9	6.3	5.8	5.6	1.1	0.5
> 6000	43	12.2	8.8	6.1	5.7	0.7	0.5

Table 4: Standard deviation of percent above best known for random graphs.

	Number	*Largest**	*Largest*	*Greedy*	*Regret*	*Sum*	*Sum**
All	352	17.8	13.7	9.6	9.5	7.9	7.7
0-500	104	24.3	17.5	8.2	9.5	6.8	6.2
500-1000	55	10.6	5.1	4.1	5.5	2.1	2.1
1000-2000	49	8.5	5.4	2.5	3.9	1.8	1.7
2000-4000	60	6.4	3.4	2.0	3.2	1.1	1.1
4000-6000	41	4.2	2.4	1.5	2.3	0.3	0.4
> 6000	43	3.3	1.9	1.4	1.7	0.3	0.2
4000-6000	41	5.2	2.8	1.9	2.2	1.0	1.0
> 6000	43	4.1	2.2	1.7	1.9	0.9	0.8

We now compare the performance of the six heuristics. When we consider the top part of Figure 4, it is clear that the two heuristics *Largest** and *Largest* perform significantly worse than the other heuristics with a mean of 55.0% (62.0%) and 40.3% (46.9%), respectively, for all real (random) graphs. Both *Greedy* and *Regret* show an average performance with a mean of 23.1% (20.2%) and 20.9% (18.3%), respectively, for all real (random) graphs. The two heuristics that show the best performance are *Sum* and *Sum**, both having

a mean of 9.3% for real graphs and of 12.5% and 12.2%, respectively, for random graphs. The lower part of Figure 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 Figure 5 and further support the relative ranking. Even though this figure indicates that *Sum* is superior to *Sum** as it wins more often, Tables 1 and 2 show that it is a minor advantage when considering the average performance. Indeed, when *Sum* does not provide the best result, it is 3.1 (2.5) percent above the best cost for real (random) graphs on average. The corresponding number for *Sum** is 4.1 (2.1) percent, confirming that we cannot identify one of these as the preferred one.

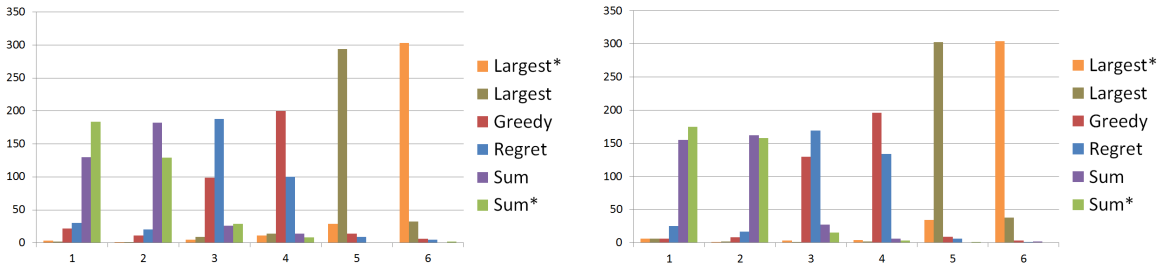

Figure 5: Relative ranks of the heuristics. Left: Real graphs. Right: Random graphs.

3.4 Run time analysis of heuristics

Finally, we consider the run time of the six heuristics as well as the Blossom V algorithm and CPLEX, both of 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 Figure 6, where run times in seconds are shown as a function of n.

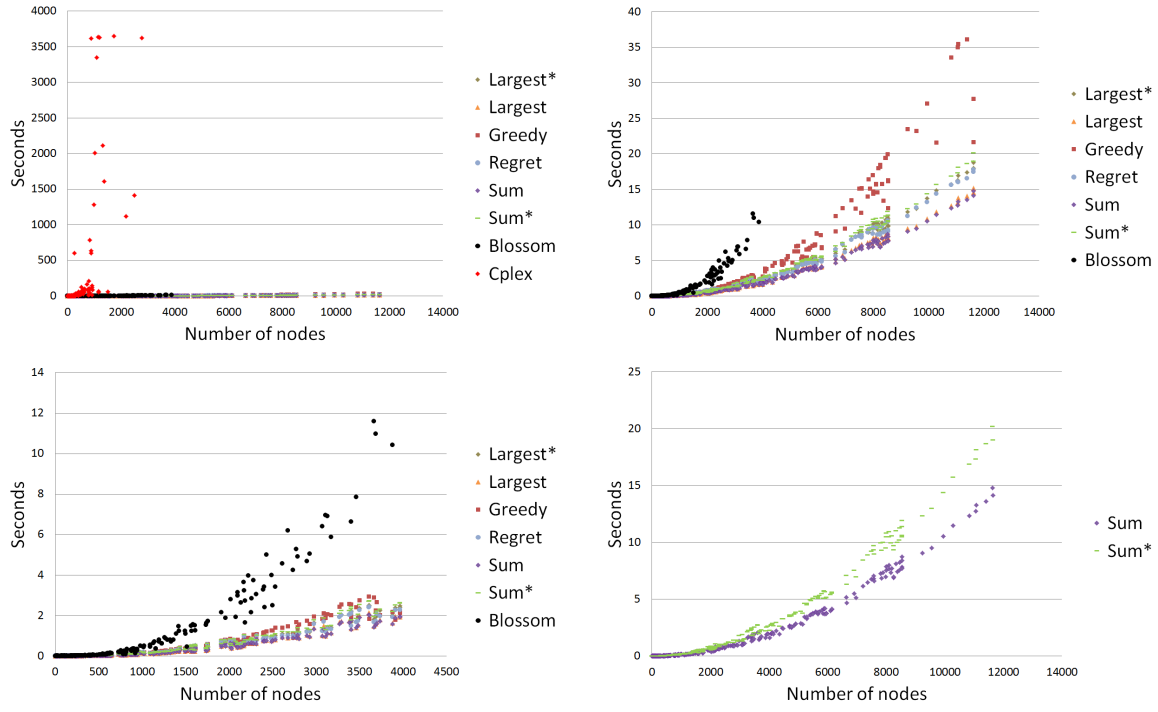

Figure 6: Run time for real graphs. Top: All graphs. Left: All algorithms, Right: All except CPLEX. Bottom left: Graphs with $n < 4000$. All algorithms except CPLEX. Bottom right: All graphs. Only *Sum* and *Sum.**

At the top left of the figure, we show all eight 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 top right part of the figure shows all graphs excluding CPLEX and can be used for comparison purposes. We note that the longest run time observed for Blossom is 12 seconds and for the most slow heuristic it is 36 seconds for larger graphs. In the lower left part of the figure, we only show 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 the run time is indeed very impressive and the Blossom algorithm is clearly preferred over CPLEX for solving matching problems with more than about 1000 nodes. 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 the plots at the top right and bottom left. These plots clearly show 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, the lower right of the figure shows the run time of the two heuristics that had the best performance, *Sum* and *Sum**. 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, and given that it is also one of the two preferred heuristics regarding quality, we will designate *Sum* as the best simple heuristic for solving the Matching Problem when it is not practical to solve the problem optimally.

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.

References

[1] Beardwood, J., Halton, J. H., and Hammersley, J. M. (1959). The shortest path through many points. Proceedings of the Cambridge Philosophical Society, 55:299–327.

[2] Cook, W. and Rohe, A. (1999). Computing minimum-weight perfect matchings. INFORMS Journal on Computing, 11:138–148.

[3] Edmonds, J. (1965). Paths, trees, and flowers. Canadian Journal of Mathematics, 17:449–467.

[4] Frederickson, G. N. (1979). Approximation algorithms for some postman problems. Journal of the Association for Computing Machinery, 26:538–554.

[5] Hertz, A., Laporte, G., and Mittaz, M. (2000). A tabu search heuristic for the capacited arc routing problem. Operations Research, 48:129–135.

[6] Kiilerich, L. and Wøhlk, S. (2017). New large-scale data instances for CARP and variations of CARP. Submitted.

[7] Kolmogorov, V. (2009) http://pub.ist.ac.at/~{}vnk/software.html#blossom5.

[8] Kolmogorov, V. (2009). Blossom V: A new implementation of a minimum cost perfect matching algorithm. Mathematical Programming Computation, 1:43–67.

[9] Papadimitriou, C. H. (1977). The probabilistic analysis of matching heuristics. Proceedings of the 15th Annual Allerton Conference on Communication, Control, and Computing, pages 368–378.

[10] Prins, C., Labadi, N., and Reghioui, M. (2009). Tour splitting algorithms for vehicle routing problems. International Journal of Production Research, 47:507–536.

[11] Wattenhofer, M. and Wattenhofer, R. (2004). Fast and simple algorithms for weighted perfect matching. Electronic Notes in Discrete Mathematics, 17:285–291.

[12] Wøhlk, S. (2008). An approximation algorithm for the capacitated arc routing problem. The Open Operational Research Journal, 2:8–12.