Experimental Study: Approximation Algorithms for Bipartite Matching with Metric and Geometric Costs

Mohanad Almiski

Abstract

The bipartite graph matching problem appears in many different contexts and is useful in a variety of algorithmic problems and situations. An exact solution for bipartite matching runs in time O(n^3). Following a paper by Agarwal and Sharathkumar, we implement two approximation algorithms for bipartite graph matching that run in time O($n^{2+\delta} log(n) log^2(1/\delta)$) and in time O($\varepsilon^{-2} n^{1+\delta} \tau(n, \varepsilon) log^2(n/\varepsilon) log(1/\delta)$) and generate a O($1/\delta^{\alpha}$)-approximation cost matching ($\alpha = log_3(2) = .63$), and show that the algorithms described work well for large graphs and have very good approximation costs for a limited test dataset. We also discuss implementation challenges, and potential work that might be done in the future to build on and improve these algorithms.

I. Introduction

A bipartite graph is a graph (V, E) with edges only from points in one set of distinct vertices to points in another set of distinct vertices. A matching of a bipartite graph is a set of edges whose vertices are distinct. A perfect matching of a bipartite graph is a matching that matches all vertices in each of the of sets. A minimum-weight perfect match is a perfect match whose selected edges are of minimum weight, the problem of finding a minimum-weight perfect matching is called the assignment problem.

The assignment problem is the motivating problem for min-weight perfect matchings, and involves assigning jobs to workers given the payment that each worker will take for doing that job. The goal is to minimize the cost of completing all the jobs, by assigning the cheapest worker for a job given the other jobs that have to be completed, and the payment that other workers will take for each job [2]. This problem can easily be formulated as a weighted bipartite graph which is what motivated the matching algorithmic approaches.

Bipartite graphs often appear in a variety of algorithmic problems and contexts, often with the goal of matching one set of items to another set. One example use of weighted bipartite matching problem is in computer vision, when tracking objects or features moving from one frame of a video to the next, a bipartite graph between the objects in the two frames and a distance function that assigns each edge a weight proportional to the difference between the two objects, allows for a simple way of keeping track of an object. More applications exist in computer graphics and in analysis of geometry [1]

This paper will implement two approximation algorithms that solve the assignment problem. The algorithms are based on the paper by Agarwal et al. [1], and run in time O($n^{2+\delta} log(n) log^2(1/\delta)$) and in time O($\varepsilon^{-2} n^{1+\delta} \tau(n, \varepsilon) log^2(n/\varepsilon) log(1/\delta)$) and generate a O($1/\delta^{\alpha}$)-approximation cost matching ($\alpha = log_3(2) = .63$).

A. Related Work

The Hungarian algorithm was the first polynomial time algorithm for solving the assignment problem running in O(n^4) time, and was developed in 1955, by Dénes Kőnig and Jenő Egerváry [2]. It was then further refined to run in O(n^3) time by Edmonds and Karp [8]. The Hungarian algorithm is a deterministic algorithm that works on weighted bipartite graphs, and returns a guaranteed min-cost assignment using an early idea of primal-dual methods to determine the adjustments to make to each matching. Gabow and Tarjan were able to prove for integer edge costs that the matching could occur in time O($m\sqrt{(}n) log^{3/2}(n)$) [2]. Varadarajan and Agarwal presented a deterministic algorithm for bipartite matching in low dimensoins that runs in time O($n^3/2(log(n)/\varepsilon)^{O(d)}$)

The Hungarian algorithm was followed by a multitude of approximation algorithms that attempted to provide an approximate min-cost assignment that ran in faster time. Sharathkumar and Agarwal [9] were able to provide a near-linear time Monte-Carlo algorithm that runs in near-linear time.

A related problem is maximum-approximate bipartite matching which is just the maximization of edges weights when retrieving a perfect matching of a graph. In general, while maximum and minimum approximate matching are similar, minimum matching is harder to approximate, with greedy algorithms for maximum weight matching giving a 0.5-approximate cost match while the same type of algorithm giving a O($n^{\cdot 58}$) cost match [1].

II. Theory

The main idea that this paper presents is a modification of some of the conditions in the Gabow-Tarjan and the Hungarian algorithm it is based off of. The two algorithms presented here all rely on bipartite graphs whose edge weights are from a metric d($\cdot, \cdot$), i.e that d(a,b) + d(b,c) $\leq$ d(a,c), d(a,a) = 0, and d(x,y)=d(y,x). Therefore, these algorithms are limited to these specific types of bipartite graphs, and the modifications that the paper uses rely on this property to get their approximation.
The first modification is the consideration of different types of edges in the augmentation step. Instead of only considering edges that are already matched, the paper expands the types of edges that can be used to generate an augmenting path by including eligible edges.
The second modification is the addition of edge scaling to reduce the edge costs of the edge weights to ensure the runtime of the algorithm does not depend on the edge weights and to generate an approximation in a reasonable amount of time.
Finally, the last modification is that the paper only approximately matches a fraction of the original bipartite graph and uses the Hungarian algorithm to perfectly match the rest of the graph, increasing the accuracy of the algorithm.

The second algorithm in the paper, then takes these ideas further by making use of an Approximate Nearest Neighbor data structure to reduce the query times necessary for determining points likely to be the best match for that point. The authors of the paper then further increase the edges that the algorithm considers in the augmentation step by allowing so-called strongly/weakly eligible edges in a ε-relaxed matching, a superset of the eligible edges from the previous algorithm.
For notation, the set A and B refer to the two sets of vertices in the bipartite graph. The set of eligible edges or set of ε-edges is a subset of the set of edges $A \times B$. $\tau(n, \varepsilon)$ refers to the update time of the ANN data structure. (a, b) refers to the edge weight function between a and b and is rounded up and scaled depending on the step in the algorithm.

A. Eligible Edges

[1] introduce the concept of eligible edges in their paper. An edge (a,b) is eligible if it satisfies one of the following:
(a,b) $\in$ Matches and y(a) + y(b) = d(a,b)
(a,b) $\notin$ Matches and y(a) + y(b) = d(a,b) + 1
During the search for a match, we consider all eligible edges as the graph and make a set of vertex disjoint paths in this induced graph that start at the free vertices of B. These are used in the augmentation step to generate a matching that are 1-feasible, meaning the sum of y(a) + y(b) $\leq$ d(a,b) + 1 for all edges.

B. Alternating Paths

As introduced in the Hungarian algorithm, an alternating path, is simply a path starting at a free vertex of B, and ending at a free vertex of A, passing through potentially already matched vertices, with each vertex alternating between A and B in the path. The alternating path is useful in the augmenting step, where the previous matches are augmented with each augmenting path to add new matches.

C. Overview of Algorithm 1

Since this algorithm is heavily based on the Gabow-Tarjan algorithm [1], it will largely be very similar to their algorithm except for a couple of minor adjustments.
The algorithm works by keeping dual-weights associated to each vertex of the bipartite-graph. These dual weights

are called potentials and assign a value to each vertex. A pair is matched in the algorithm when the sum of the dual weights of the pair add to their edge weight.
The algorithm runs a method called the Phase-Match until it has matched 1/3 of the vertices in the graph. Then it uses the Hungarian algorithm to perfectly match the graph generated by the remaining $n^{2/3}$ vertices.
The algorithm uses eligible edges to expand the number of alternating paths that it can add in the Phase-DFS method to the augmentation step to increase the matching each step of the way. The Phase-DFS procedure performs a simple depth first search to generate the vertex disjoint paths, and is detailed more in [2] and [3], additionally the Phase-DFS procedure is described as a step in Algorithm 1.

To match the first 1/3 vertices, it calls the Phase-Match procedure. This procedure iteratively updates weights of free vertices until they have an eligible edge. It maintains dual weights to determine the eligible edges and generates a directed graph G from the original bipartite graph to update the weights. For any match collected so far, (a,b), we direct the edge from b to a, for any other edge we direct from a to b.
Part of the procedure requires a parameter ω such that $w(M*)/2 \leq \omega \leq w(M*)$, where $M*$ is the optimal matching. This can be achieved by using an n-approximation algorithm to get ω and dividing ω by 2^i, i from 1 to $\log_2$(n) and choosing the least cost matching from the $\log_2$(n) runs. For this paper however, to reduce the running time and the size of graph we had to consider, we used the previous runs of the hungarian algorithm to give each algorithm a 3/4w(M*) as a value for ω.

D. ε-relaxed matchings and strongly/weakly eligible edges

In the second algorithm, the concept of 1-feasible matchings is extended by an arbitrary free parameter $\varepsilon \in (0, 1)$ to consider ε-relaxed matchings.
A distinction is made between strongly eligible edges and weakly eligible.
Those edges between a $\in$ A, b$\in$ B such that the dual weights satisfy either of the following two conditions are called strongly eligible:
(a,b) $\in$ Matches and y(a) + y(b) == d(a,b) or
(a,b) $\notin$ Matches and y(a) + y(b) == d(a,b) + $\lceil \varepsilon d(a, b) \rceil$
While edges that satisfy the following are called weakly eligible:
(a,b) $\notin$ Matches and d(a,b) $<$ y(a) + y(b) $<$ d(a,b) + $\lceil \varepsilon d(a, b) \rceil$
Using this expanded version of the 1-feasible matching definition allows us to match more of the edges with a better approximation, and allows us to only use the Hungarian algorithm for the last 1/3 of the vertices.

E. Well Separated Path Decomposition

The ANN based data structure that the paper defines uses a Well Separated Path Decomposition of the interval [0:N] with $\rho = \epsilon/10$ to store an ANN of the set K at each pair of intervals (X,Y), where K = $\{a \mid \|y(a)\| \in X\}$.
The Well Separated Path Decomposition and the algorithm to generate it is detailed in Chapter 3 of [7]. Instead of a quadtree for the tree we use a binary tree decomposition of the interval to help generate the WSPD, since the interval is 1 dimensional.
The paper uses the WSPD to help separate points out and find strongly eligible edges faster, utilizing the ANN attached to each interval.

F. Overview of Algorithm 2

Following this expanded definition and through the use of any data structure that allows approximate nearest neighbor queries and deletions to be made in time $\tau(n, \varepsilon)$, the first algorithm is modified to the second algorithm. The changes from the previous algorithm are:
1) The number of stages that need to be taken in each step and the scale that the distance function is scaled by changes from the order of 3^i to the order of 2^i.
2) We now generate the vertex disjoint set of augmenting paths using the ANN based data structure and the dual

weights to find edges that can be traversed from each free vertex of B, if and only if the vertex b has a strongly eligible edge adjacent to it.

3) Finally, the last change is to maintain the ε-relaxed matching M by augmenting the dual weights for new matches by setting the dual weight of the matching vertex so that y(a) + y(b) = d(a,b).

Except for these differences, the algorithm is the same as before. However the resulting changes significantly affect the theoretical run time of the algorithm.

Algorithm 1: Approximate bipartite matching using 1-feasible matchings

Result: A $O(1/\delta^{\alpha})$-approximate matching of a bipartite graph

Function `Phase-Match` (*G = (A* $\bigcup$ *B, E), d*$(\cdot,\cdot)$, *y*$(\cdot)$, *k*) **:**

- M = $\emptyset$
- **for** *k steps* **do**
  - Generate P, set of vertex disjoint paths on eligible edges (determined using $d(\cdot,\cdot)$ and y$(\cdot)$) in G, starting from all free vertices of B
  - **for** $p \in P$ **do**
    - M = M $\bigoplus$ p, $\bigoplus$ is symmetric difference
  - **end**
  - **for** $b \in$ *free vertices of B* **do**
    - Create directed graph $\overrightarrow{G}$, (a $\rightarrow$ b) if (a,b) $\in$ M, otherwise (b $\rightarrow$ a)
    - **for** $a \in A$ *that is reachable from* b **do**
      - y(a) -= 1
    - **end**
    - **for** $b \in B$ *that is reachable from* b **do**
      - y(b) += 1
    - **end**
  - **end**
  - **if** $|M| = |A|$ **then**
    - **return** M
  - **end**
- **end**
- **return** M

Function `Approximate-Bipartite-Match` (*G = (A* $\bigcup$ *B, E), d*$(\cdot,\cdot)$, δ, ω) **:**

- n = $\|A\|$
- A_i = A
- B_i = B
- M = $\emptyset$
- $\epsilon = \frac{1}{2\log_3(\frac{1}{\delta})}$
- $d_i(\cdot,\cdot) = \lceil \frac{2 \cdot n \cdot d(\cdot,\cdot)}{\epsilon\omega} \rceil$
- **while** $\|M\| \leq n$ **do**
  - $k = \frac{30}{\epsilon} n^{3^i \delta}$
  - **if** $\|A_i\| \leq n^{\frac{2}{3}}$ **then**
    - M = M $\bigcup$ Hungarian-Algorithm($G = (A_i \bigcup B_i, E)$, $d_i(\cdot,\cdot)$, $y(\cdot)$, k)
    - **return** M
  - **end**
  - **else**
    - M = M $\bigcup$ Phase-Match(G, $d_i(\cdot,\cdot)$, $y(\cdot)$, k)
    - A_{i+1} = remaining free vertices of A_i
    - B_{i+1} = remaining free vertices of B_i
    - $d_{i+1}(\cdot,\cdot) = \lceil \frac{d_i(\cdot,\cdot)}{2(1+\epsilon)^2 n^{3^{i-1}\delta}} \rceil$
  - **end**
- **end**
- **return** M

return

Algorithm 2: Approximate bipartite matching using strongly/weakly edges in $\varepsilon-$relaxed matchings

Result: A O($1/\delta^{\alpha}$)-approximate matching of a bipartite graph

Function `Phase-Match-With-ANN-DS` (*G = (A $\bigcup$ B, E), $d(\cdot,\cdot)$, y(·), k, W, ε*) **:**

M = $\emptyset$

for *k steps* **do**

For every pair (X, Y) in W, construct, D, a ANN on the vertices of all K$\subseteq$ A, K = $\{a \mid \|y(a)\| \in X\}$ using d$(\cdot,\cdot)$ with accuracy ε

Generate P, set of vertex disjoint paths on strongly/weakly eligible edges (created by using D and y$(\cdot)$ to determine ε-relaxed edges, whenever we visit a $\in$ A, we delete it from all the D in every pair) in G, starting from every free vertex in B

for $p \in P$ **do**

M = M $\bigoplus$ p, $\bigoplus$ is symmetric difference

For every new edge (a,b) added to M, set y(b) = d(a,b) - y(a)

end

for $b \in$ *free vertices of B* **do**

Create directed graph $\overrightarrow{G}$, (a $\rightarrow$ b) if (a,b) $\in$ M, otherwise (b $\rightarrow$ a)

for $a \in A$ *that is reachable from* b **do**

y(a) -= 1

end

for $b \in B$ *that is reachable from* b **do**

y(b) += 1

end

end

if $|M| = |A|$ **then**

return M

end

end

return M

Function `Approximate-Bipartite-Match` (*G = (A $\bigcup$ B, E), $d(\cdot,\cdot)$, δ, ω, ε*) **:**

n = $\|A\|$

A_i = A

B_i = B

M = $\emptyset$

$\epsilon = \frac{1}{2\log_3(\frac{1}{\delta})}$

$d_i(\cdot,\cdot) = \lceil \frac{2 \cdot n \cdot d(\cdot,\cdot)}{\epsilon\omega} \rceil$

Construct a Well Separated Path Decomposition, W, of the integer interval [0:N], N = n/ε, with parameter $\rho = \varepsilon/10$

while $\|M\| \leq n$ **do**

$k = \frac{30}{\epsilon} n^{2^i \delta}$

if $\|A_i\| \leq n^{\frac{1}{3}}$ **then**

M = M $\bigcup$ Hungarian-Algorithm($G = (A_i \bigcup B_i, E)$, $d_i(\cdot,\cdot)$, $y(\cdot)$, k)

return M

end

else

M = M $\bigcup$ Phase-Match-With-ANN-DS(G, $d_i(\cdot,\cdot)$, $y(\cdot)$, k, W, ε)

A_{i+1} = remaining free vertices of A_i

B_{i+1} = remaining free vertices of B_i

$d_{i+1}(\cdot,\cdot) = \lceil \frac{d_i(\cdot,\cdot)}{2(1+\epsilon)^2 n^{2^{i-1}\delta}} \rceil$

end

end

return M

G. Comparing algorithms

To compare the algorithms to each other, we will be running the algorithms on a variety of bipartite graph data generated randomly and comparing the performances of the two algorithms against the Hungarian algorithm in addition to comparing the approximate costs of each algorithm compared to the true cost generated by the hungarian algorithm. Because each algorithm has hyperparameters that affect the runtime as well as the cost of the matchings, we will run the algorithms with several different configurations and comparing them altogether. The performance metrics that we will use will be the runtime of the algorithms in seconds and the percent of true matching cost for the approximate cost generated by the algorithms. This will simply be the approximate cost divided by the true cost.

III. Data and Implementation

The algorithms were implemented in C++-17, using the standard library, as well as a variety of libraries for more obscure data structures and algorithms.
The hungarian algorithm was imported and created by [5] and was used in both of the algorithms as stated in the paper. The algorithm was heavily optimized and uses the matrix formulation of the algorithm instead of the graph formulation, which might make it run faster.
Although the paper mentions using a dynamic Approximate nearest neighbor algorithm to accompany each interval in the WSPD, we could not find a data structure implemented in any language that had dynamic updates and performed approximate nearest neighbor search. Therefore we decided to use a static ANN, libANN [6], and make up for the lack of dynamic updates by recreating the ANN every time a point was deleted from it. For this reason the results from algorithm 2 may not be completely true to the spirit of the paper without remembering the delete time of the ANN was not the same as the query time as requested in the paper [1].
Additionally, as mentioned previously, WSPD algorithm was implemented using a binary tree where each interval is represented by a node in the tree, with the left and right children representing the two halves of the interval. This replaces the quadtree used in the algorithm detailed in Chapter 3 of [7].

The data was generated randomly using a python script to generate uniform random points from two circles in 2-dimensional space. Each set of the 10 bipartite graphs was generated from random points in 2-dimensional space and then ran through each of the algorithms.
Since the two algorithms here have multiple parameters, for each of the approximation algorithms we will consider different configuratins. We will consider 3 values for δ in Algorithm 1 and 2 values for each parameter (δ, ε) of Algorithm 2. This will help us explore the relation between the parameters, the approximate cost, and the run time of each algorithm. The data will cover 10 graphs going from size 50 all the way to size 3000, which is the limit of a reasonable run of all the algorithms on the machine that the testing was done on.

A. Implementation Challenges

There were some challenges in the implementation of the algorithms, in particular with the second algorithm. One large challenge was due to vagueness in the description of the black box ANN data structure that is used in the second algorithm. In particular it was not very clear when the ANN is generated for a set of vertices in the algorithm and how it is paired with the WSPD that is generated. For this reason, Algorithm 2 may not actually be representative of the original algorithm described in the paper. However, despite these challenges, good results were taken out of the implementation of Algorithm 2.

IV. Results

The runtimes for each algorithm and configuration for the 10 sample datasets are presented in Fig. 1. and Fig 2, the approximate costs as a percent of the true costs are represented in Fig 3. and Fig 4.. We split the tables according to the relative size of the set of vertices for each data set.
We can see from the data that the Hungarian algorithm is much faster on the smaller datasets, with none of the other algorithms beating its runtime until the 1000 vertex data set in the larger data set. Thereafter the hungarian algorithm continues to grow in computation time relative to the other algorithms and eventually takes the longest

time to compute the matchings, in the largest data set of 3000 vertices.
It is apparent that the second algorithm does a faster job matching than the hungarian algorithm and the other algorithm. Algorithm 1, however is not that far behind, matching faster than the hungarian algorithm for all the test data sets after the 750 vertices data set.
However, not only is the second algorithm faster, it is also as accurate as Algorithm 1 in comparison to the percent of the true cost of the matchings that the algorithm computes.
Algorithm 2 with the the configuration of $\delta = .15, \varepsilon = .25$ computed more accurate approximate matches than Algorithm 1 did with $\delta = .08$. All the algorithms actually had very good approximation ratios, however this may just be an artifact of the test data sets being too close to each other in the metric space they were generated in.
Somewhat surprisingly, smaller values of δ resulted in better approximate costs than larger values ($\delta \geq .15$). This may not hold in general as we couldn't test on larger and more varied graphs due to computational reasons. Further investigation of the effect of choosing δ values may be needed to understand why the costs were smaller for lower δ values.
Additionally, for algorithm 2, the algorithm ran much faster on the largest graph than the second largest graph. This may be because the largest graph contained points closer together than the graph before it resulting in the ANN data structure being able to utilize the closer distances between points to speed up the computation.

V. Conclusions and Future Work

Although the algorithms are useful for matching bipartite graphs approximately, they are still beaten rather easily for smaller bipartite graphs by the O(n^3) Hungarian algorithm using the matrix formulation. Due to the fact that they run rather slow for smaller graphs, and because they are still approximate, the Hungarian algorithm seems to be the best solution for smaller graphs.
For larger graphs, however, although both algorithms still have approximate costs, they ran slightly faster than the Hungarian Algorithm on the larger graph data sets, and much faster in the largest (vertex count = 3000) graph. Therefore, the algorithms may be much more successful on much larger graphs, on the order of hundreds of thousands of vertices, where they may be more widely used, such as in a large data-center or in a physics simulation.

A. Future Work

Unfortunately, we couldn't test beyond a graph of size at most 3000 vertices due to computational limitations. In the future, work can be done to further investigate how well the algorithms can do on graphs of much larger size. In addition, more work can be done to fine tune the algorithms and remove unnecessary computations to lower the overall runtime.
Finally, one area that we left rather untouched was exploring the parameter space of each of the algorithms. Since each algorithm relies on the use of free parameters that affect the runtime, more extensive exploration of the parameters could uncover optimal settings for larger graphs.
Since these algorithms are better suited for larger graphs, they could be applied to large scale planning where the O(n^3) time frame is too large, and where approximate costs would be acceptable. A good candidate could be in a computer graphics physics simulation or in a computer vision algorithm that requires many objects to be tracked but not that each object is exactly tracked.
For further information in regard to the algorithms and their theoretical foundations, the original paper by Agarwal and Sharathkumar [1] provides more theoretical justification to the algorithms.

References

[1] Pankaj K. Agarwal and R. Sharathkumar. 2014. Approximation algorithms for bipartite matching with metric and geometric costs. In Proceedings of the forty-sixth annual ACM symposium on Theory of computing (STOC '14). Association for Computing Machinery, New York, NY, USA, 555–564. DOI:https://doi.org/10.1145/2591796.2591844
[2] H. N. Gabow and R. Tarjan, Faster scaling algorithms for network problems, SIAM J. Comput., 18 (1989), 1013–1036.
[3] R. Sharathkumar and P. K. Agarwal, Algorithms for transportation problem in geometric settings, Proc. 23rd Annual ACM-SIAM Sympos. Discrete Algo., 2012, pp. 306–317.

Fig. 1. Small vertex data set: run times (seconds)

alg.	50 verts.	100 verts.	200 verts.	300 verts.	400 verts.	500 verts.
hungarian	**0.003**	**0.02**	**0.14**	**0.50**	**1.66**	**2.32**
algorithm 1 ($\delta = .08$)	0.08	0.36	1.29	2.45	6.94	9.10
algorithm 1 ($\delta = .15$)	0.05	0.17	0.85	2.00	5.62	5.45
algorithm 1 ($\delta = .3$)	0.03	0.12	0.63	1.48	4.22	4.40
algorithm 2 ($\delta = .15, \varepsilon = .25$)	0.71	1.38	7.00	7.02	25.58	23.45
algorithm 2 ($\delta = .15, \varepsilon = .75$)	0.04	0.15	1.04	1.40	4.89	3.68
algorithm 2 ($\delta = .3, \varepsilon = .25$)	0.55	1.73	7.16	14.86	20.35	45.56
algorithm 2 ($\delta = .3, \varepsilon = .75$)	0.06	0.16	1.82	2.80	4.68	6.33

Fig. 2. Large vertex data set: run times (seconds)

alg.	750 verts.	1000 verts.	1500 verts.	3000 verts.
hungarian	**7.31**	19.02	66.16	595.27
algorithm 1 ($\delta = .08$)	19.29	33.42	61.75	113.63
algorithm 1 ($\delta = .15$)	16.36	28.98	48.34	106.21
algorithm 1 ($\delta = .3$)	11.35	18.69	39.92	112.54
algorithm 2 ($\delta = .15, \varepsilon = .25$)	71.73	72.56	165.23	75.66
algorithm 2 ($\delta = .15, \varepsilon = .75$)	9.02	**15.47**	34.51	62.79
algorithm 2 ($\delta = .3, \varepsilon = .25$)	56.32	193.22	503.11	72.14
algorithm 2 ($\delta = .3, \varepsilon = .75$)	8.11	26.89	**22.97**	**60.55**

Fig. 3. Small vertex data set: approximate cost (percent of true)

alg.	50 verts.	100 verts.	200 verts.	300 verts.	400 verts.	500 verts.
true cost	***13444***	***27176***	***58066***	***84106***	***126291***	***143796***
algorithm 1 ($\delta = .08$)	1.013	1.012	1.008	**1.007**	1.007	**1.007**
algorithm 1 ($\delta = .15$)	1.012	1.015	1.009	1.009	1.008	1.009
algorithm 1 ($\delta = .3$)	1.021	1.013	1.013	1.012	1.010	1.010
algorithm 2 ($\delta = .15, \varepsilon = .25$)	**1.010**	**1.009**	**1.002**	1.009	**1.006**	1.009
algorithm 2 ($\delta = .3, \varepsilon = .25$)	1.026	1.025	1.006	1.022	1.018	1.020
algorithm 2 ($\delta = .15, \varepsilon = .75$)	1.020	1.012	1.010	1.008	1.009	1.010
algorithm 2 ($\delta = .3, \varepsilon = .75$)	1.035	1.027	1.020	1.022	1.017	1.021

Fig. 4. Large vertex data set: run times (percent of true)

alg.	750 verts.	1000 verts.	1500 verts.	3000 verts.
hungarian	***213221***	***279913***	***427506***	***851943***
algorithm 1 ($\delta = .08$)	**1.006**	**1.006**	1.008	1.033
algorithm 1 ($\delta = .15$)	1.007	1.008	1.009	1.033
algorithm 1 ($\delta = .3$)	1.010	1.011	1.014	1.033
algorithm 2 ($\delta = .15, \varepsilon = .25$)	1.007	1.008	**1.007**	1.023
algorithm 2 ($\delta = .3, \varepsilon = .25$)	1.018	1.018	1.015	1.023
algorithm 2 ($\delta = .15, \varepsilon = .75$)	1.007	1.008	1.008	**1.007**
algorithm 2 ($\delta = .3, \varepsilon = .75$)	1.019	1.017	1.018	1.023

[4] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, J. ACM, 34 (1987), 596–615.
[5] https://github.com/mcximing/hungarian-algorithm-cpp
[6] http://www.cs.umd.edu/ mount/ANN/
[7] S. Har-peled, Geometric Approximation Algorithms,
[8] Edmonds, Jack; Karp, Richard M. (1 April 1972). "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". Journal of the ACM. 19 (2): 248–264. doi:10.1145/321694.321699.
[9] R. Sharathkumar, A sub-quadratic algorithm for bipartite matching of planar points with bounded integer coordinates, Proc. 29th Annual Sympos. Comput. Geom., 2013, pp. 9–16.
American Mathematical Society, 2011.