To appear in ORSA Journal on Computing.

Computational Experience with an Approximation Algorithm on Large-Scale Euclidean Matching Instances

David P. Williamson
IBM TJ Watson Research Center
Yorktown Heights, NY 10598
(914) 945-1743
`dpw@watson.ibm.com`

Michel X. Goemans
Department of Mathematics
MIT
Cambridge, MA 02139
(617) 253-2688
`goemans@math.mit.edu`

Abstract

We consider a 2-approximation algorithm for Euclidean minimum-cost perfect matching instances proposed by the authors in a previous paper. We present computational results for both random and real-world instances having between $1,000$ and $131,072$ vertices. The results indicate that our algorithm generates a matching within 2% of optimal in most cases. In over 1,400 experiments, the algorithm was never more than 4% from optimal. For the purposes of the study, we give a new implementation of the algorithm that uses linear space instead of quadratic space, and appears to run faster in practice.

Subject classification:

- Networks/graphs, matchings: experimental analysis of heuristics
- Analysis of algorithms: experimental analysis

The main criticism which is often formulated with regard to approximation algorithms is that, although they are backed up by a performance guarantee (often through very elegant arguments), they might not generate "nearly-optimal" solutions in practice. Indeed, a practitioner will seldom be satisfied with a solution guaranteed to be of cost less than, say, twice the optimum cost; by far, he will prefer a heuristic algorithm which typically generates a solution within, say, 3% of optimality, although this heuristic might from time to time generate a more costly solution. For example, in the context of the traveling salesman problem, computational studies show that the heuristic algorithm of Lin and Kernighan outperforms in practice the algorithm of Christofides, although the latter one has a performance guarantee of $\frac{3}{2}$ (i.e., the cost of Christofides' tour is guaranteed to be within a factor of $\frac{3}{2}$ of optimal) [8]. A similar comment can be made for running times both for exact or approximation algorithms. The fastest algorithm in practice might not be the one with best *worst-case* running time.

Motivated by these comments, we perform a computational study of an existing approximation algorithm for the Euclidean minimum-cost perfect matching problem. The minimum-cost perfect matching problem is the problem of finding a minimum-cost set of non-adjacent edges that cover all vertices. The algorithm we consider was proposed by Goemans and Williamson [11] for any cost function satisfying the triangle inequality, but we restrict our attention to Euclidean instances where the edge costs are the distances between points under the l_2 or l_∞ norms. The algorithm has a performance guarantee of 2, and the implementation described in [11] runs in $O(n^2 \log n)$ time, where n denotes the number of vertices. The algorithm has an important feature from a practical point-of-view: it is primal-dual. This means that it not only generates a perfect matching but also a lower bound on the optimum cost. The worst-case result proved in [11] says that the cost of the perfect matching generated is at most twice the value of the lower bound generated by the algorithm.

The main purpose of this computational study is to determine how far the perfect matching obtained by the algorithm is from optimal. Our results thus far indicate that in terms of quality of approximation, the algorithm behaves more like Lin-Kernighan's algorithm than Christofides': in over 1,400 experiments on random and structured instances ranging in size from 1,000 to 131,072 vertices, the algorithm was never more than 4% away from optimal and 7% away from the lower bound. On random instances drawn uniformly from the unit square, a result of Papadimitriou [16] shows that the cost of an optimal matching on n vertices almost surely converges to $\beta_M \sqrt{n}$, where β_M is constant. Assuming that our algorithm has the same behavior, we show experimentally that the matching constant for the algorithm is 1.6% away from β_M and 3.7% away from the constant of the lower bound when the Euclidean norm is used. When the l_∞ norm is used, these gaps become 2.0% and 4.4%. In the course of obtaining these results, we also obtain a new estimate of β_M. On several structured instances drawn from the Traveling Salesman Library, TSPLIB [18], the algorithm performed somewhat better than on random instances, generating matchings at most 2% away from optimal and usually within 1–1.5%.

For the purposes of this study, it was necessary to develop a different implementation of the algorithm than that given in Goemans and Williamson [11]. The main drawback of the implementation given there is that it requires $\Theta(n^2)$ space. This space bound severely limits the size of the instances that could be solved. Using some algorithmic ideas of Bentley and Friedman [6] and Bentley [5], we give an implementation that uses $O(n)$ space. Our implementation also seems to be faster in practice, although its worst-case running time, while still polynomial, is not as good as $O(n^2 \log n)$.

In addition to studying the quality of solutions produced by the algorithm, we report other various statistical properties of the solutions and of the algorithm's implementation.

Our study to this point does not yet answer the question of whether the approximation algorithm is a practical alternative to a very good implementation of an algorithm that can solve the matching problem exactly or to other heuristics for the matching problem. Edmonds [7] first showed that the matching problem is solvable in polynomial time, and since then several algorithms and implementations of these algorithms have been proposed; see Applegate and Cook [1] for a brief discussion. For our study, we used a very efficient implementation of Edmonds' algorithm written by Applegate and Cook [1] to find optimal matchings. Our implementation was only faster than Applegate and Cook's code on large random instances; it was slightly slower on small random and structured instances and usually significantly slower on large structured instances. It is likely that a speed-up of our implementation can be achieved if we first restrict our attention to a sparse subgraph as is done by Applegate and Cook. In addition, other matching heuristics described in the literature (such as that of Jünger and Pulleyblank [14] and those in the survey of Avis [2]) run much faster, although the quality of the solutions produced does not appear to be as good and the heuristics have no worst-case performance guarantee.

Our abstract is structured as follows. In Section 1, we review the approximation algorithm of Goemans and Williamson [11]. Section 2 discusses the new implementation, and Section 3 reviews known results about the behavior of Euclidean optimization problems on random instances that prompted some of our computational study. Section 4 gives the results of our study, and we conclude with some remarks in Section 5.

1 Review of the algorithm

In this section, we review the algorithm of Goemans and Williamson [11] and we also briefly indicate an implementation running in $O(n^2 \log n)$ time.

The algorithm is shown in Figure 1. The main portion of the algorithm (lines 1 through 12) finds a set F' of edges such that every vertex has odd degree and, thus, each connected component has even size. This part of the algorithm maintains a forest F of edges, which is initially empty. The algorithm loops, in every iteration selecting an edge (i, j) between two distinct connected components of F, then merging these two components by adding (i, j) to F. The loop terminates when all connected components C of F have even size. From these edges in F, we form the set F' of edges by keeping only necessary edges. Thus we remove any edge from F that joins two components of even size. A parity argument shows that every vertex in the resulting set must have odd degree.

The final step of the algorithm (line 13) turns the set of edges F' into a perfect matching. A matching of cost no greater than F' can be obtained as follows: duplicate all edges in F' so that each component is an Eulerian graph on an even number of vertices, shortcut each component to a cycle, then keep the best matching out of the two induced by the cycle. To obtain slightly better matchings, we find the optimal matching on the component by complete enumeration for all components of size 10 or smaller, and use the doubling-and-shortcutting method for larger components.

The good approximation properties of the algorithm follow from the way in which the edge is chosen in each iteration. The algorithm maintains variables $d(v)$ for each vertex v, and a lower bound LB; these variables are all initially zero. In each iteration, the algorithm chooses an edge $e = (i, j)$ spanning

```
**Input:** A complete graph $G = (V, E)$ and edge costs $c_e \geq 0$ satisfying the triangle inequality
**Output:** A perfect matching $M$ and a value $LB$
$F \leftarrow \emptyset$
$LB \leftarrow 0$
$\mathcal{C} \leftarrow \{\{v\} : v \in V\}$
For each $v \in V$
  $d(v) \leftarrow 0$
While $\exists C \in \mathcal{C} : |C|$ is odd
  Find edge $e = (i, j)$ with $i \in C_p \in \mathcal{C}$, $j \in C_q \in \mathcal{C}$, $C_p \neq C_q$ that minimizes $\epsilon = \frac{c_e - d(i) - d(j)}{p(C_p) + p(C_q)}$
  $F \leftarrow F \cup \{e\}$
  For all $v \in C_r \in \mathcal{C}$ do $d(v) \leftarrow d(v) + \epsilon \cdot p(C_r)$
  $LB \leftarrow LB + \epsilon \sum_{C \in \mathcal{C}} p(C)$
  $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_p \cup C_q\} - \{C_p\} - \{C_q\}$
$F' \leftarrow \{e \in F :$ There exists an odd connected component $N$ in $(V, F - \{e\})\}$
Convert $F'$ to matching $M$. For each component of $F'$, duplicate all the edges of $F'$, shortcut in order to obtain a collection of cycles and, for each such cycle, keep the best matching out of the two that it induces. Find optimal matchings on small components.
```

Figure 1: The main algorithm.

two distinct components (say C_p and C_q) that minimizes $\epsilon = \frac{c_e - d(i) - d(j)}{p(C_p) + p(C_q)}$, where $p(C)$ is the parity of component C (that is, 1 if C has odd size, and 0 otherwise). It can be shown that $\epsilon \geq 0$. Then $d(v)$ is increased by ϵ for all vertices v in odd components, and LB is increased by ϵ times the number of odd components. Goemans and Williamson [11] show that choosing the edges in this fashion implicitly constructs a feasible solution to the dual of a linear programming formulation of the matching problem such that the value of the dual solution is LB, and $\sum_{e \in F'} c_e \leq 2\,LB$. Because LB is a lower bound on the value of an optimal matching, this proves that the algorithm has a performance guarantee of 2.

The main problem involved in implementing the algorithm is deciding which edge to select in each iteration. The $O(n^2 \log n)$ time algorithm of [11] uses a priority queue of edges. Given a notion of *time* T in the algorithm (starting at zero and advancing by ϵ each iteration), the key value of each edge $e = (i, j)$ is $T + \frac{c_e - d(i) - d(j)}{p(C_p) + p(C_q)}$. We call this quantity the edge's *addition time*; it is the time at which the reduced cost $\frac{c_e - d(i) - d(j)}{p(C_p) + p(C_q)}$ of the edge will be zero, given that the parity of the two components containing its two endpoints remains the same. Since the parity of components change only when two components are merged, we only need to update the key values of edges adjacent to the two components that are merged. Also, for each pair of components we only need to keep the edge with the smallest addition time. We select the edge with the smallest addition time in each iteration. This analysis leads to $O(n)$ queue operations per iteration of the algorithm, with $O(n)$ iterations, for an overall running time of $O(n^2 \log n)$. Gabow, Goemans, and Williamson [10] have shown how this selection step can be implemented in $O(n^2 \sqrt{\log \log n})$ time by using many small priority queues.

2 Implementation Description

In this section, we describe another implementation of the edge selection step. The implementation described in the previous section has two main disadvantages for performing computational experiments. The first is that it uses $\Theta(n^2)$ space; this severely limits the size of the instances that can be solved.

The second is that the running time is in fact $\Theta(n^2 \log n)$: the algorithm must add at least $n/2$ edges to obtain a feasible solution, necessitating $\Theta(n^2)$ queue operations. The main theoretical advantage of our new implementation compared to the one given above is that it is much more space efficient, using only $O(n)$ space. Moreover, although its worst-case time complexity appears to be worse than the original implementation, it performs well enough on average to allow us to run relatively large instances. It will require $O(n^2)$ queue operations, and we will see experimentally that it requires $O(n^{1+\epsilon})$ queue operations on random instances, although other factors will now dominate the running time.

Before we go on to describe the main idea behind selecting edges in our implementation, we note that several small tricks are necessary to ensure that other parts of the algorithm do not force the running time to be $\Omega(n^2)$. For example, updating the $d(v)$ variables each iteration would take $\Theta(n^2)$ time. To avoid this, we augment a union-find structure used to keep track of the connected components of the current edge set. Since the $d(v)$ are increased by the same amount for all vertices v in the same component, we increase an offset in the root of the component, and define $d(v)$ to be the sum of the offsets along the path to the root. In addition, we let the increases for a component accumulate and only change the offset when we merge the component with another component, or when we need to calculate $d(i)$ for some vertex i in the component.

The basic idea of the implementation is that each component should maintain an estimate of its closest neighboring component under addition times. The corresponding edges are placed in a priority queue with the estimates as the key values. The estimates are maintained in such a way that the shortest edge (under addition times) between two components is always found in any iteration of the while loop; thus the algorithm can be successfully implemented. The main advantage of this implementation is its space efficiency: we need to keep track of the keys of only $|\mathcal{C}|$ edges, where $\mathcal{C}$ is the set of components. This approach is adapted from algorithms for the minimum-cost spanning tree and traveling salesman problems in Bentley and Friedman [6] and Bentley [5].

More formally, let $l(e)$ denote the current addition time of edge $e = (i, j)$. For two components C_p and C_q in $\mathcal{C}$, let $l(C_p, C_q)$ be equal to the smallest addition time of an edge with one endpoint in C_p and the other in C_q. The key of an edge e in the queue will be denoted by $k(e)$ and corresponds to the addition time of that edge when it was added to the queue. By abuse of notation, we let $k(C)$ denote the key of the edge in the queue which was selected by component C.

The implementation works as follows:

- Initially, every vertex calculates its nearest neighbor (under addition times) and puts the corresponding edge in the priority queue with a key value of the addition time.
- Whenever we pull an edge e off the queue, we check if its key value $k(e)$ is no less than its actual addition time $l(e)$. We maintain that whenever this is true, then the edge is the next edge that should be added; that is, it has the smallest addition time. Whenever two components get merged into one, we find its new nearest neighbor under addition times.
- When the key value of the edge e is less than the actual addition time, we then search for the component's real nearest neighbor, bounding the search by the correct addition time $l(e)$ of e, and insert the corresponding edge in the priority queue with its correct key value.

In order to prove that the implementation is correct, we first prove that it maintains an invariant.

Lemma 2.1 At any point in the algorithm, for all $C_p, C_q \in \mathcal{C}$, $\min(k(C_p), k(C_q)) \leq l(C_p, C_q)$.

Proof: Certainly the invariant is true initially. Suppose that we insert an edge e selected by component C to the queue. This insertion might be the result of either two components merging into C or the discovery that the edge in the queue corresponding to C has a key less than its addition time. In both cases, the invariant is maintained for any two components C_p and C_q different from C. Moreover, if $C_p = C$ then our choice of the edge to insert guarantees that $\min(k(C), k(C_q)) \leq k(C) = k(e) = l(e) \leq l(C, C_q)$, implying that the invariant continues to hold. ■

The invariant leads to a proof of correctness.

Theorem 2.2 The implementation selects an edge with the smallest addition time in every iteration.

Proof: In each iteration of the algorithm, we must find the edge with the smallest addition time. Let a denote the smallest addition time of this iteration, and let e be the edge at the top of the queue. We will show that whenever $k(e) \geq l(e)$ then $l(e) = a$ and thus the algorithm correctly selects edge e. Whenever $k(e) < l(e)$ we replace the queue element e with another edge e' such that $k(e') = l(e')$. Such a replacement does not affect the distances between components or the other key values in the queue, so we can replace at most $|\mathcal{C}|$ number of edges before we must reach the case that $k(e) \geq l(e)$ for the top element e of the queue.

Suppose $k(e) \geq l(e)$. By the invariant, for any $C_p, C_q \in \mathcal{C}$, we have $\min(k(C_p), k(C_q)) \leq l(C_p, C_q)$. But since e is the edge at the top of the queue, $k(e) \leq k(C)$ for all $C \in \mathcal{C}$ and, thus, $k(e) \leq l(C_p, C_q)$. The fact that $l(e) \leq k(e)$ now implies that $l(e) \leq l(C_p, C_q)$ for any $C_p, C_q \in \mathcal{C}$. In other words, e is an edge with smallest addition time. ■

We now evaluate the worst-case number of queue operations. The argument of the theorem shows that we perform at most $O(n)$ queue operations for each edge selection. This implies that the algorithm performs a total of $O(n^2)$ queue operations.

In order to complete the description of our implementation, we must describe how to find an edge e that represents the nearest neighbor of a component C under addition time. To do this, we use *k-d* trees of Friedman, Bentley, and Finkel [9] as described in Bentley [4]. A *k-d* tree is a binary tree that corresponds to a partitioning of a given set of points in d-dimensional space; here we use $d = 2$. The tree is constructed by determining whether the points are spread out most in the x or y direction, then finding the vertical or horizontal line that splits the points in half in that direction. The line determines an internal node of the tree, and the procedure is performed recursively on each half until the number of points remaining is below a certain threshold (usually 5 or 6). Hence each leaf in the tree corresponds to a bucket containing at most a certain number of points. The tree can be used to perform a search for the nearest neighbor of a given point under a number of norms (including L_1, l_2 and l_∞): at each internal node we first search the subtree containing the given point, then search the other tree if the nearest neighbor found so far is no closer than the distance from the given point to the line determining the internal node. Friedman et al. show experimentally that this search takes $O(\log n)$ expected time, and Bentley [4] gives a bottom-up variation that seems to take $O(1)$ expected time. Notice that this search method also lends itself to searching for the nearest neighbor within a certain radius.

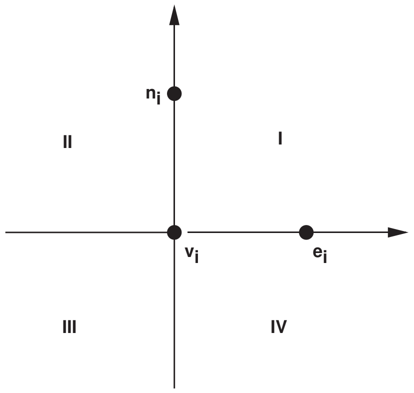


Figure 2: The four quadrants around v_i.

To find the nearest neighbor of a component C under addition time, we iterate through the vertices in C to find the smallest edge (under addition time) from the vertex to a vertex not in C. We use the smallest edge found so far to bound the search on the next vertex. Let v_i denote the point in the Euclidean plane corresponding to vertex i, and let $||v_i - v_j||$ denote the distance between vertices i and j, and hence the cost of edge (i, j). Suppose we are searching from vertex i in component C, the current time is T, and the smallest edge found so far has addition time a. Thus any potentially smaller edge $e = (i, j)$ must satisfy $T + \frac{c_e - d(i) - d(j)}{p(C)+p(C')} \leq a$, which implies that $c_e - d(i) - T \leq (a - T)(p(C) + 1)$ since $d(j) \leq T$ for all $j \in V$ and $p(C') \leq 1$. Therefore j must be within distance $(a - T)(p(C) + 1) + d(i) + T$.

As one might suspect, the time spent performing these searches dominates all other operations in our implementation. Therefore, we introduce a few tricks for speeding up these searches. The first trick is that whenever we notice that all the vertices in a subtree of the *k-d* tree belong to the same component, we label the subtree with the name of that component. Then whenever we search for the nearest neighbor of a vertex in C, we ignore all subtrees labelled C. This trick is useful as C becomes large and most of the neighboring vertices of a given vertex in C are also in C.

Another trick we use is that when searching for the nearest neighbor of a vertex i in C, we can sometimes infer when possible nearest neighbor candidates will be closer to another vertex j in C than i. We say that i is *shadowed* by j. Thus we can disregard these nearest neighbor candidates. To be more formal, consider the four quadrants of the plane formed by using v_i as the origin. Let (x_i, y_i) denote the coordinates of v_i. Let n_i be the Euclidean point $(x_i + d(i), y_i)$ and let e_i be the point $(x_i, y_i + d(i))$ (see Figure 2).

Theorem 2.3 Suppose there is some other vertex j in C such that j is not in quadrant III (i.e. j is such that $x_j \geq x_i$ or $y_j \geq y_i$), and

$$||n_i - v_j|| \leq d(j) \text{ and } ||e_i - v_j|| \leq d(j).$$

Then, for any vertex q in quadrant I ($x_q \geq x_i$ and $y_q \geq y_i$) that is in component $C' \neq C$, vertex j is at least as close to vertex q under addition times as is vertex i.

Proof: To show this, we will prove that $||v_q - v_j|| - d(j) \leq ||v_q - v_i|| - d(i)$. From this it will follow that $T + \frac{||v_q - v_j|| - d(j) - d(q)}{p(C)+p(C')} \leq T + \frac{||v_q - v_i|| - d(i) - d(q)}{p(C)+p(C')}$.

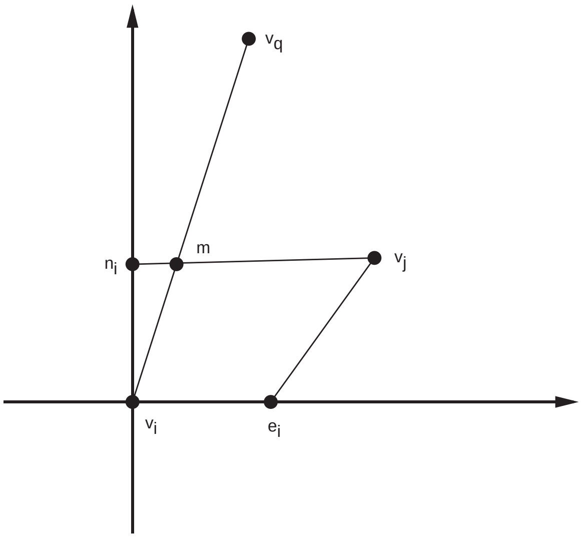


Figure 3: Illustration of the case in which v_q is in quadrant I.

We consider two cases. First suppose that v_j is in quadrant I. Because vertex q is not in component C, it cannot lie inside the triangle defined by v_i, n_i and e_i. Furthermore, v_q cannot lie inside the triangle defined by v_j, n_i and e_i since both n_i and e_i are within distance $d(j)$ of v_j. As a result, the line segment (v_q, v_i) must intersect either the line segment (n_i, v_j) or the line segment (v_j, e_i) (see Figure 3). Assume the former (the other case is similar), and let m be the intersection point. Then

$$\begin{aligned} &||v_i - v_q|| + ||v_j - n_i|| \\ &\quad = \quad ||v_i - m|| + ||m - n_i|| + ||v_q - m|| + ||m - v_j|| \\ &\quad \geq \quad ||v_i - n_i|| + ||v_q - v_j||, \end{aligned}$$

which implies that $||v_q - v_i|| + d(j) \geq ||v_q - v_j|| + d(i)$, as desired.

Now suppose that v_j is in quadrant IV (the case for II is similar). Since $||v_i - v_j|| \leq ||n_i - v_j|| \leq d(j)$ and v_q is not in the same component as v_i and v_j, v_q cannot be in the triangle defined by v_i, v_j, and n_i. Hence, the line segments (v_q, v_i) and (n_i, v_j) must intersect. Then the proof is the same as above. ■

Thus if vertex i is shadowed as in the statement of the theorem, we need not look for nearest neighbors of i in quadrant I. Obviously, shadowing is symmetric with respect to quadrants. Once the size of a component has increased by a certain amount, we sweep through the component, determining in which quadrants each vertex is shadowed. We store this information in four bits for each vertex and use it when we perform nearest neighbor searches to cut down the scope of the search for each vertex. If a vertex is shadowed in all four quadrants, then we do not perform a search on it at all.

3 Probabilistic Analysis of Euclidean Functionals

A great deal is known about the behavior of randomly distributed Euclidean instances of combinatorial optimization problems. In the basic version of the *Euclidean model*, the vertices of the problem instance are distributed independently and uniformly in the unit square $[0, 1]^2$ and the Euclidean metric plays the role of cost function. The *functionals* of interest are typically the values of combinatorial optimization problems (such as the traveling salesman, matching or minimum spanning tree problems) on these randomly distributed points. The behavior of these functionals is somewhat independent of the functional itself and, for these reasons, we briefly review some of these probabilistic results for the most studied problem, the traveling salesman problem. We also indicate results likely to hold, although they have never been proved. For a more detailed picture of the field, the reader is referred to [23, 15, 21, 27].

The asymptotic behavior of the value of the traveling salesman problem was first studied in the pioneering paper of Beardwood, Halton and Hammersley [3]. They prove the existence of a constant β_{TSP} such that

$$\lim_{n\to\infty} \frac{TSP_n}{\sqrt{n}} = \beta_{TSP} \qquad \text{almost surely} \tag{1}$$

where TSP_n denotes the value of the optimal tour on $X_1, X_2, \ldots, X_n$ with the X_i's being an infinite sequence of independently and uniformly distributed vertices from $[0,1]^2$. Steele has shown that $TSP_n/\sqrt{n}$ converges completely to β_{TSP} rather than almost surely [22]. We should point out that the exact value of β_{TSP} is not known. More recently, a careful analysis of the functional has led to the following results and/or conjectures. It is known that Var TSP_n is upper bounded by a constant (see Steele [22]): Var $TSP_n \leq \frac{16}{\pi} + O(\frac{1}{n})$. Quoting Steele [21],

> it seems inevitable that one has a genuine limit, $\lim_{n\to\infty} \mathrm{Var}\, TSP_n = \sigma^2 > 0$. It is less certain but (still very likely) that one has a central limit theorem $TSP_n - E[TSP_n] \sim N(0, \sigma^2)$.

Rhee and Talagrand [20] have proved a first step towards this central limit theorem by showing that the tails of TSP_n are Gaussian or subGaussian: there exists K such that, for all t, $Pr\left(|TSP_n - E[TSP_n]| > t\right) \leq Ke^{-t^2/K}$. From the knowledge of $E[TSP_n]$ for a finite value of n, one can derive a bound on the limiting constant β_{TSP}. Indeed (see Jaillet [13]), there exists a constant $\gamma_{TSP} < 9.5$ such that, for all n, $|E[TSP_n] - \beta_{TSP}\sqrt{n}| \leq \gamma_{TSP}$. Furthermore, it seems likely that there exists a limit α_{TSP} such that $\lim_{n\to\infty} |E[TSP_n] - \beta_{TSP}\sqrt{n}| = \alpha_{TSP}$.

Some of these results also hold for other functionals; see Steele [23], Yukich [27], and the references above. For the minimum-cost perfect matching problem and its associated functional M, the asymptotic behavior (1) is known to hold (Papadimitriou [16]), and has also been strengthened to complete convergence (Redmond and Yukich [17]). The other results and/or conjectures seem likely to be true as well.

From these results and/or conjectures, we shall implicitly assume for our experimental study that, for several functionals L, L_n is normally distributed with mean $\beta_L\sqrt{n} + \alpha_L$ and variance σ_L^2. These functionals are the values of the minimum-cost perfect matching, and also of F', the perfect matching and the dual solution returned by our approximation algorithm. We shall denote these additional functionals by F', P (for primal) and D (for dual).

Functionals of a more structural nature have also been studied. For example, Steele et al. [24] have shown that there exist constants ν_k such that the number of degree k vertices in a minimum spanning tree divided by n is almost surely equal to ν_k. It is known that $\nu_k = 0$ for $k \geq 6$.

Problem Name	Norm	LBGAP	OPTGAP	Time	AC Time
r1002	l_2	4.59	1.54	3.57	2.69
r2392	l_2	3.57	.98	7.38	11.51
pcb3038	l_2	2.98	.93	30.32	22.26
rl5934	l_2	2.37	.93	326.30	119.85
pla7396	l_2	1.72	.94	461.25	203.11
rl11848	l_2	2.87	1.18	944.60	229.82
d18512	l_2	3.57	1.64	3651.13	664.93
r20726	l_∞	4.40	1.88	718.95	4636.06
pla33810	l_2	2.14	1.69	24687.20	1704.09
pla85900	l_∞	1.52	1.34	107653.81	6202.22

Table 1: Experimental results on TSPLIB instances.

4 Results

We summarize our main experimental results in a sequence of tables. Table 1 contains our results on the TSPLIB examples. Tables 2 and 3 contain our results on random instances; the first table is for instances using the l_2 norm, and the second is for instances using the l_∞ norm. The columns LBGAP and OPTGAP give the percentage excess of the approximate matching over the cost of the dual lower bound and over the cost of an optimal matching respectively. For instance, the approximate matching found for the r1002 TSPLIB instance was 4.59% more than the cost of the dual lower bound, and 1.54% more than the cost of the optimal matching. The Time column specifies the running time of the approximation algorithm, while the AC Time column specifies the running time of the Applegate and Cook algorithm for finding optimal matchings. We used the variation of the Applegate and Cook code which begins with a fractional 10 nearest neighbor graph. All running times are in CPU seconds on a Silicon Graphics Challenge machine with eight 100Mhz MIPS R4400 processors. The Speedup column gives the ratio of the running time of Applegate and Cook's algorithm to the running time of the approximation algorithm. We summarize our estimates of the parameters β and α in Table 4. We include for comparison parameters given for other matching heuristics. Finally, some asymptotic estimates of structural properties of the solutions are given in Table 5. Each of these tables is discussed in the following paragraphs. Some sample runs of the algorithm are shown in Figures 4 to 7.

The structured examples in Table 1 were taken from the TSPLIB [18]. The number in the problem name indicates the number of vertices in the problem. We attempted to use the same procedure as given in Applegate and Cook [1]; namely, if the example contained an odd number of points, we sorted by x and y coordinates, then deleted the last point. We also attempted as much as possible to run the same suite of examples as given in [1].

The random examples in Tables 2 and 3 were generated on a 2^{20} by 2^{20} grid using the UNIX `random()` function. A single seed was used to generate all the instances of a given size. The first entry of Table 2 comes from a sequence of 1000 experiments run separately on a VAX 9000 (11 data points had to be omitted). We used these experiments to get an upper bound on the variance of the matching parameters for F', P, D, and M on the unit square. Using the parameter with the largest variance (F'), we obtained an upper bound of .05399. This information was used to decide the number of experiments to perform in order to obtain small confidence intervals on the parameters β. We did not use the 1000 experiments

Size	Trials	Ave LBGAP	Ave OPTGAP	Max LBGAP	Max OPTGAP	Ave Time	Ave Speedup
2^{10}	989	3.69	1.59	5.87	3.46	–	–
2^{10}	64	3.69	1.58	6.15	3.67	5.09	.92
2^{11}	64	3.69	1.60	4.86	2.45	21.60	.92
2^{12}	32	3.72	1.65	5.00	2.65	79.75	1.12
2^{13}	32	3.61	1.57	4.34	2.08	261.59	1.72
2^{14}	16	3.68	1.63	3.90	1.89	1330.69	2.17
2^{15}	16	3.71	1.65	4.00	1.89	7533.67	2.00
2^{16}	8	3.70	1.64	3.79	1.79	32942.70	2.00
2^{17}	4	3.74	1.63	3.83	1.70	200820.00	1.87

Table 2: Experimental results on random instances using the l_2 norm.

Size	Trials	Ave LBGAP	Ave OPTGAP	Max LBGAP	Max OPTGAP	Ave Time	Ave Speedup
2^{10}	64	4.40	1.90	6.44	3.82	4.04	.73
2^{11}	64	4.40	1.97	5.64	2.89	14.86	.87
2^{12}	32	4.40	2.01	5.26	2.65	53.15	1.27
2^{13}	32	4.40	1.97	5.14	2.54	246.27	1.73
2^{14}	16	4.33	1.95	4.56	2.22	885.21	2.79
2^{15}	16	4.35	1.96	4.52	2.13	4577.87	2.95
2^{16}	8	4.46	2.05	4.70	2.23	28017.50	2.16
2^{17}	4	4.45	1.97	4.51	2.03	150841.00	3.19

Table 3: Experimental results on random instances using the l_∞ norm.

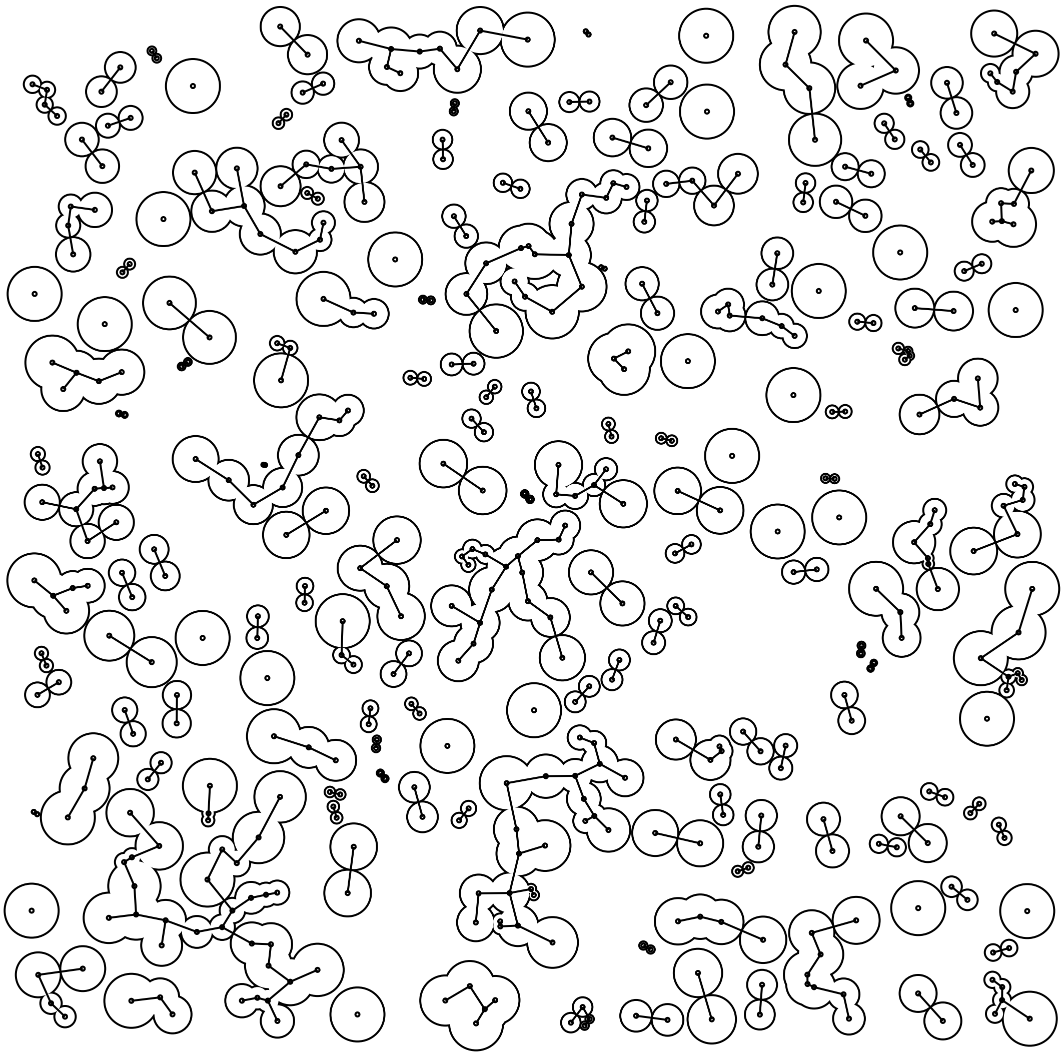
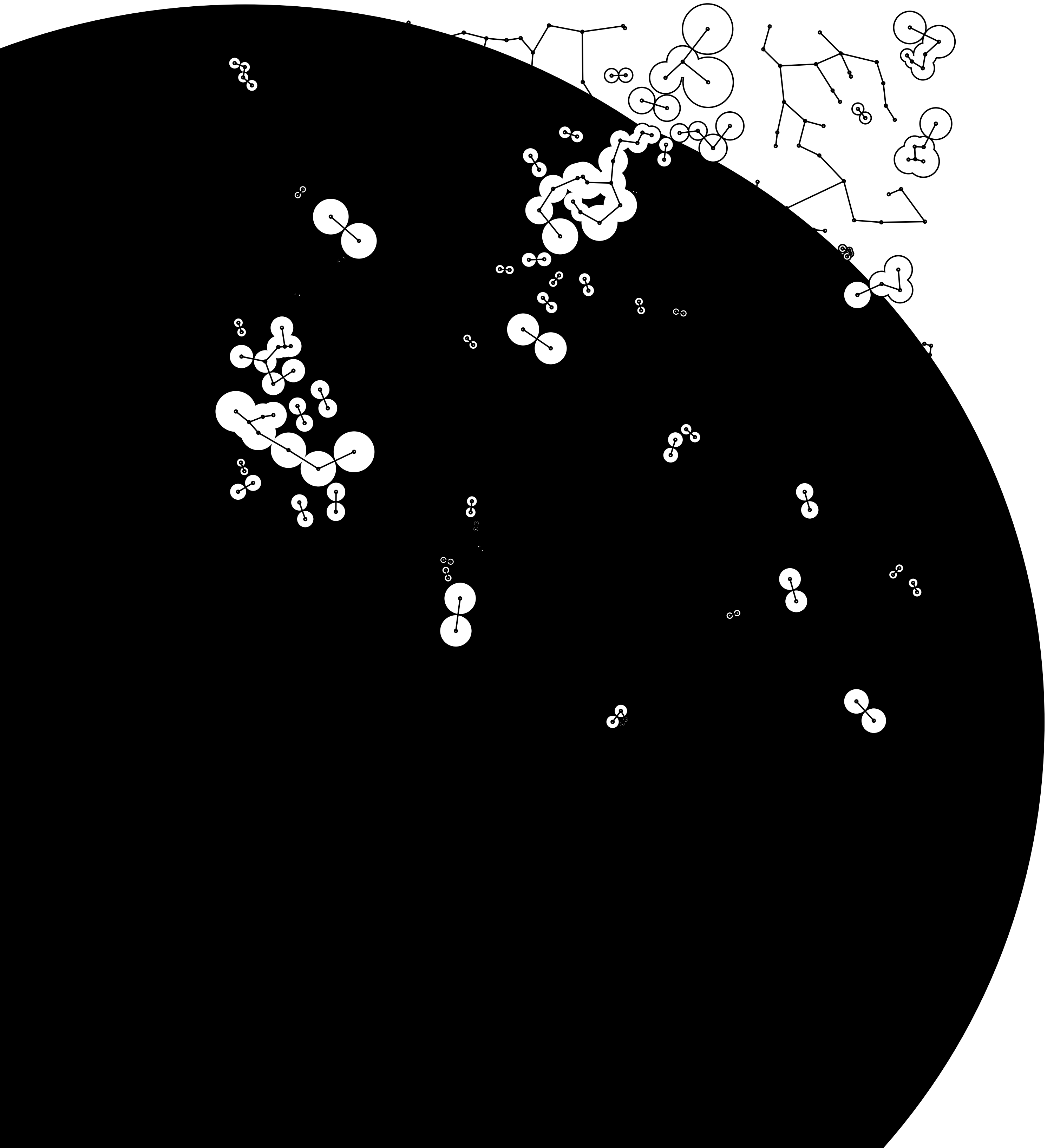


Figure 4: Snapshot of the algorithm working on a random instance of 500 vertices. The circle around each vertex i is of radius $d(i)$. Intuitively the algorithm expands these circles by ϵ in each iteration, causing two circles to touch. The edge (i, j) corresponding to these circles is then selected.



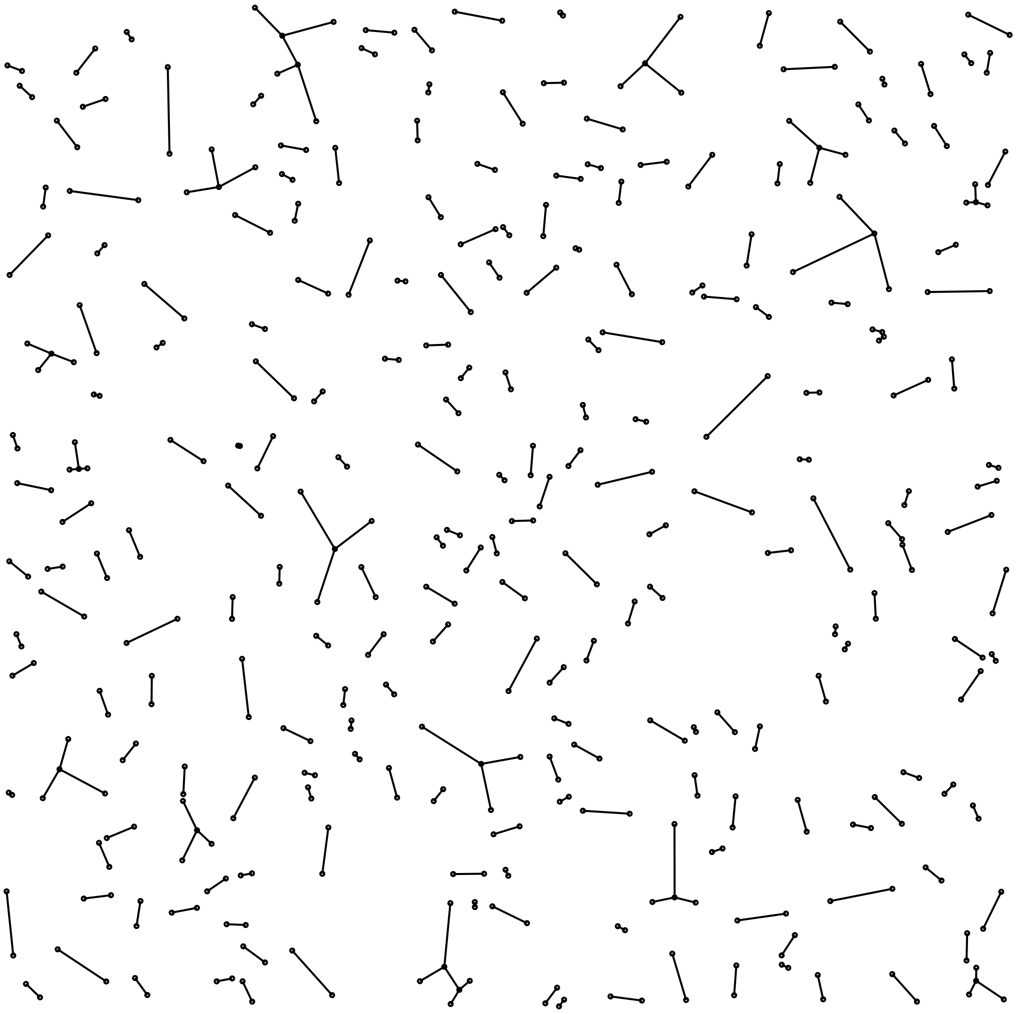


Figure 6: The forest F' produced by the algorithm for the 500 vertex random instance.

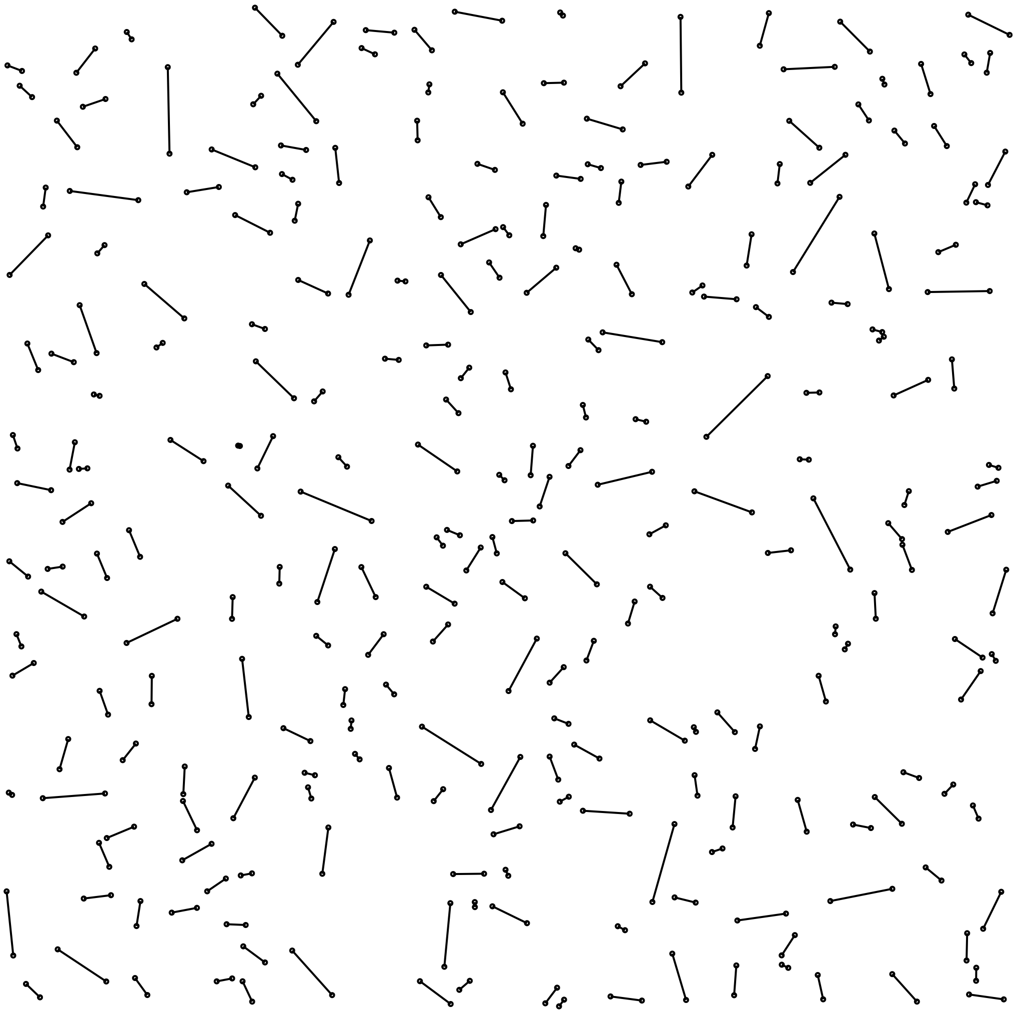


Figure 7: Final matching produced by the algorithm for the 500 vertex random instance.

	l_2 norm				l_∞ norm			
	$\hat{\beta}$	Std Err	$\hat{\alpha}$	Std Err	$\hat{\beta}$	Std Err	$\hat{\alpha}$	Std Err
Cost of F'	.3363	.0002	.2720	.0236	.2989	.0002	.2248	.0207
Approx. Matching (P)	.3154	.0001	.2327	.0150	.2807	.0001	.2038	.0149
Opt. Matching (M)	.3103	.0001	.2357	.0127	.2752	.0001	.2052	.0129
Lower bound (D)	.3041	.0001	.2298	.0121	.2688	.0001	.1978	.0123
Serpentine [12]	.585				.545			
Spiral-rack [12]	.495				.450			
Rectangular [25, 19]	.5164				.4288			
Strip [25]	.474				.436			
MST-H [25, 14]	.358				–			
DUST [14]	.338				–			
Greedy [5]	.385		.47		–			
Greedy + 2-opt [5]	.326		.24		–			

Table 4: Estimates of matching constants for our approximation algorithm (adjusted for the unit square) and other heuristics. The estimates on Rectangular and Strip are analytically determined.

in these parameter estimations.

We summarize our findings on the matching parameters in Table 4. Parameters were estimated using a least-squares fit. The "Std Err" column gives the standard error s_b, or the estimated standard deviation, of the estimated parameter. At 99% confidence, the actual parameter is within $\pm 2.576\, s_b$ of the estimated parameter; for example, with 99% confidence β_M is between .31 and .3106. To allow comparisons with our algorithm, we include the β coefficient of several other Euclidean matching heuristics from the literature, including the Serpentine and Spiral-rack heuristics of Iri, Murota, and Matsui [12], the Rectangle and Strip heuristics of Supowit, Plaisted, and Reingold [25], the MST heuristic of Papadimitriou as given in Supowit et al. (and tested in [14]), the DUST heuristic of Jünger and Pulleyblank [14], and two versions of a greedy algorithm as implemented by Bentley [5]. All of these heuristics run in $O(n \log n)$ time, except for the first two, which require $O(n)$ time, and the last two, which run in $O(n \log n)$ observed time. The table shows that for these random instances the approximation algorithm generates solutions that on average are significantly closer to optimal than the other heuristics. If the estimates of β are in fact correct, then we expect the approximation algorithm to deliver solutions of value no more than $(\beta_P - \beta_M)/\beta_M \approx 1.6\%$ away from optimal as n tends toward infinity. Similarly, the value of the dual constructed by the approximation algorithm is also expected to be a good bound on the value of the optimal solution.

The value of β_P is affected by the implementation of the final step of the algorithm (step 13), which changes even-sized components into a perfect matching. As we noted before, we find the perfect matching for the component on all components of at most 10 vertices. We will see below that most of these components fall in this range (see Section 4 for details). Some small-scale testing shows that this change improves the quality of the matching on random instances by 1%; that is, the value of the approximate matching is about 2.8% from optimal if the doubling-and-shortcutting method is used on all components.

We should mention that there have been other efforts to estimate the matching parameter β_M, where the cost of the matching was assumed to be $\beta_M \sqrt{n}$, rather than $\beta_M \sqrt{n} + \alpha_M$. Papadimitriou [16]

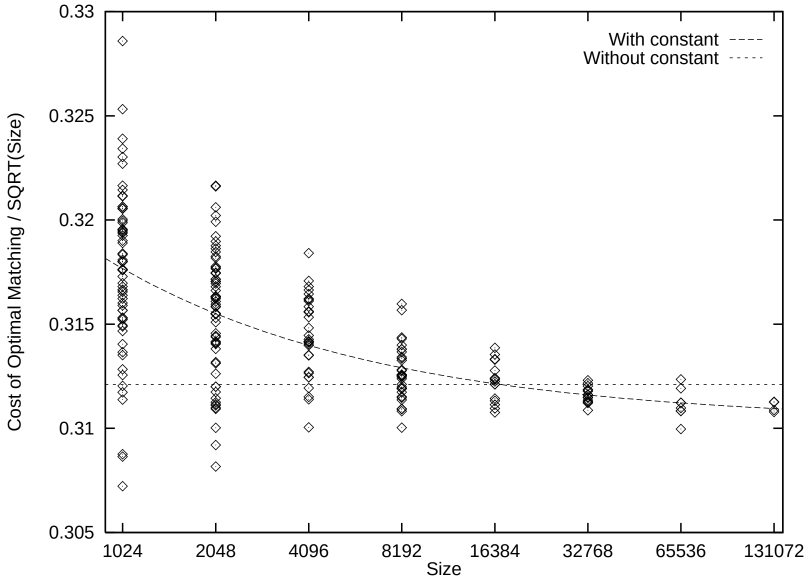


Figure 8: Scatterplot of n versus cost of optimal matching scaled to the unit square and divided by $\sqrt{n}$. Curves are regressions of the data with and without a constant term α.

conjectured that $\beta_M = .35$, based on some experiments that had at most 200 vertices. Iri et al. [12] noted that their experiments on sizes up to 256 vertices seemed to indicate that $.32 \leq \beta_M \leq .33$ (note that this agrees well with our predicted value for this value of n). Weber and Liebling [26] obtained an estimate $\beta_M \approx .3189$; the largest example used in their study was on 1,000 vertices. Because of the omission of α_M term, we believe these previous estimates are overestimates. (Note that the same may hold true for the estimated behavior of the various matching heuristics shown in Figure 4.) The influence of the α_M term is especially strong if the maximum number of vertices in the experiments is small. Figure 8 is a scatterplot of n versus the cost of the matching scaled to the unit square divided by $\sqrt{n}$. Two curves are shown: one of an estimated curve $\beta_M\sqrt{n}$ and one of an estimated curve $\beta_M\sqrt{n} + \alpha_M$. Clearly the curve with the constant gives the better fit.

We also include scatterplots of n versus the scaled cost of the approximate matching and the scaled cost of the lower bound in Figures 9 and 10 respectively. Comparing these figures to Figure 8 it seems reasonable to believe that our assumptions about the cost of the approximate matching and the lower bound are correct; namely, that they also are distributed around a mean of $\beta\sqrt{n} + \alpha$ with constant variance.

In Table 5, we list some asymptotic estimates of the structural properties of solutions. Data on the fraction of vertices of degree k in F and F' are shown in Figures 11 and 12. All structural properties listed in the table appear to be linear in n, and we modelled them either as γn or $\gamma n + \eta$ for some constants γ, η. The choice of which model to use was based on whether the residuals of the estimation were skewed for low values of n in the γn model: if so, the additive constant was included. For the most part, the variance of properties associated with the set of edges F tended to be quite high, growing with n^2, while the variance of properties associated with F' tended to be low, growing with $\sqrt{n}$. We judged the order of growth of the variance for each property by looking at its relative increase from instances of size 2^n to instances of size 2^{n+1}. We used this judgment to properly adjust the least-squares estimation

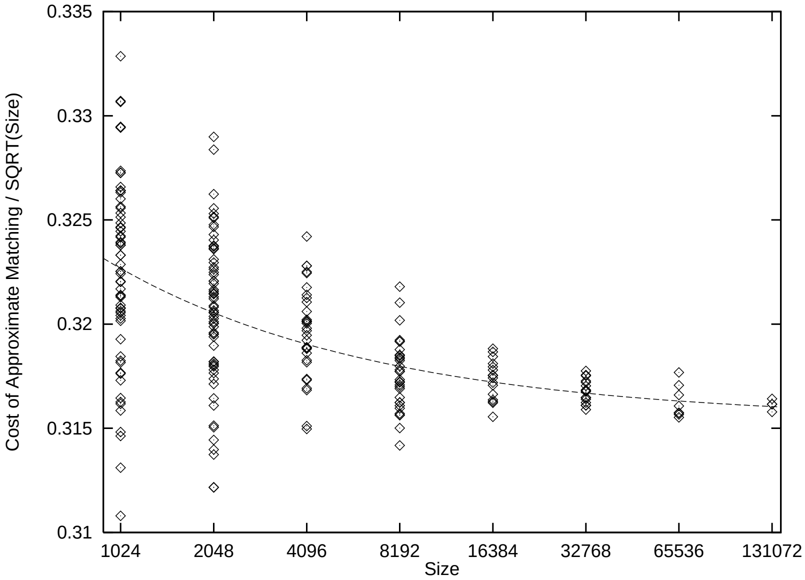


Figure 9: Scatterplot of n versus cost of approximate matching scaled to the unit square and divided by $\sqrt{n}$. The curve is the estimate $\hat{\beta}_P \sqrt{n} + \hat{\alpha}_P$.

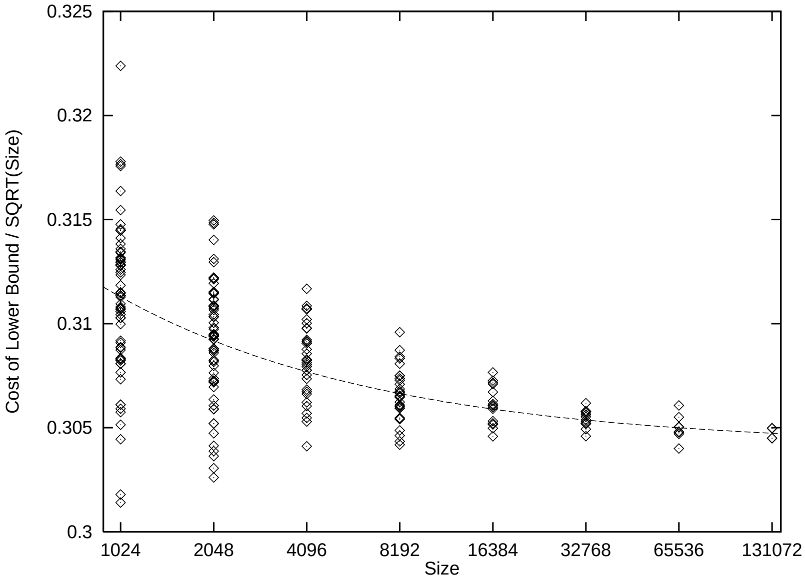


Figure 10: Scatterplot of n versus lower bound scaled to the unit square and divided by $\sqrt{n}$. The curve is the estimate $\hat{\beta}_D \sqrt{n} + \hat{\alpha}_D$.

	l_2 norm		l_∞ norm	
	F	F'	F	F'
Vertices of degree 1	$.311n + 36$	$.954n$	$.320n + 20$	$.950n$
Vertices of degree 2	$.508n - 21$	0	$.494n - 12$	0
Vertices of degree 3	$.166n - 12$	$.046n$	$.167n - 8$	$.050n$
Vertices of degree 4	$.014n - 2$	0	$.018n - 1$	0
Vertices of degree 5	$.001n$	≈ 0	$.001n$	≈ 0
Number of edges	$.931n$	$.546n$	$.936n$	$.550n$
Number of components	$.057n + 26$	$.454n$	$.057n + 15$	$.450n$
Components of size 2	–	$.416n$	–	$.409n$
Components of size 4	–	$.032n$	–	$.034n$
Components of size 6	–	$.005n$	–	$.006n$
Components of size 8	–	$.001n$	–	$.001n$
Components of size 10	–	≈ 0	–	≈ 0

Table 5: Estimates of asymptotic properties of solutions.

(least-squares requires constant variance in the observations). The data in the table indicates that the behavior of the algorithm shown in Figures 5 and 6 is typical: namely, in both F and F', almost half the components are matched vertices, and most components are not much larger, except that in F there is one giant component. Furthermore, most vertices are of low degree in both F and F'.

Looking at Figures 11 and 12, it seems plausible that as in the case of the minimum-cost spanning tree, the fraction of vertices of degree k in the forests F and F' is almost surely some constant ν_k. In looking at the data we observed that unlike the minimum-cost spanning tree it is possible to have vertices of degree 6 or 7 in F, but vertices of degree 7 are extremely rare, and we saw no vertices of degree 8. Similarly, vertices of degree 5 in F' are also extremely rare, and we saw no vertices of higher degree.

We conclude with some observations about the estimated behavior of our implementation. Modelling the number of queue operations as $\lambda n^\mu \epsilon$ (where ϵ is assumed to be a normally distributed error term), we obtained an estimate of $\hat{\mu} = 1.006$, with a standard error small enough to reject the hypothesis that the exponent is 1. The total number of calls to the routine to find the nearest neighbor of a vertex had an exponent $\hat{\mu} = 2.002$ for the l_2 norm instances and $\hat{\mu} = 1.901$ for the l_∞ instances. The running time of the implementation is highly correlated to the number of calls to this routine. We presume this fact leads to a running time of $\Theta(n^\mu \log n)$ for our implementation. Modelling the running time of Applegate and Cook's code as $\lambda n^\mu \epsilon$ gave an estimate of $\hat{\mu} = 2.29$ for the l_2 instances and $\hat{\mu} = 2.41$ for the l_∞ instances. We note that the variances of the running times for both algorithms increased significantly with n, making it difficult to say anything intelligible about asymptotic running times. As an illustration of this, Figure 13 shows the running times of the algorithms on the eight instances of size 65536.

5 Concluding Remarks

Empirically, our approximation algorithm seems to deliver perfect matchings which are very close to optimal. As with any computational study, one might argue that the observed behavior is dependent

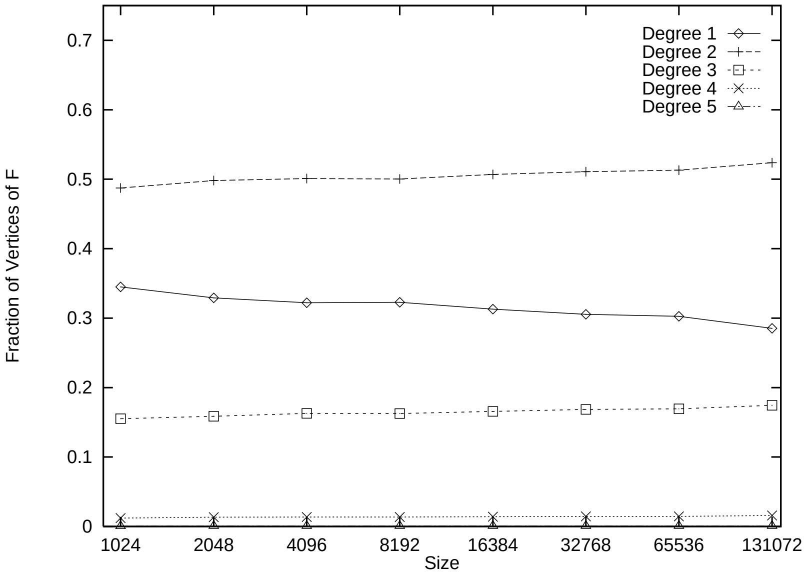


Figure 11: Plot of size of instance versus average fraction of vertices of degree k in F, $1 \leq k \leq 5$.

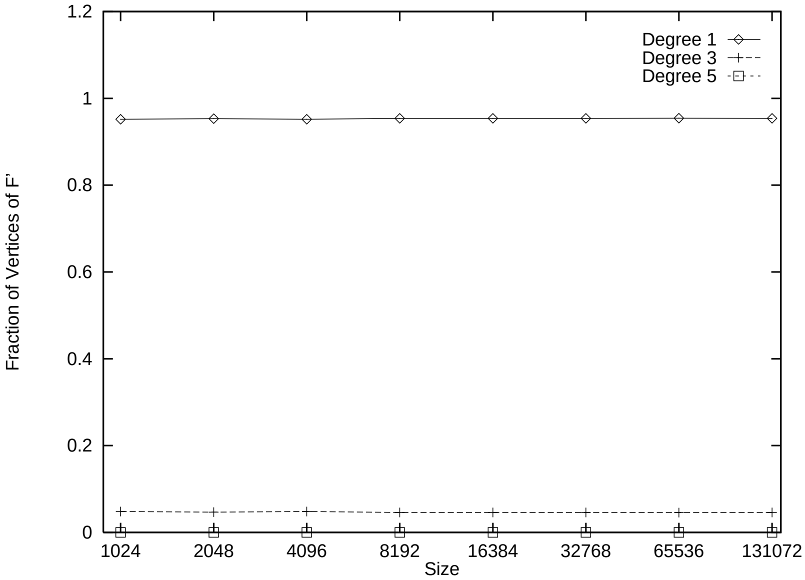


Figure 12: Plot of size of instance versus average fraction of vertices of degree k in F', $1 \leq k \leq 5$.

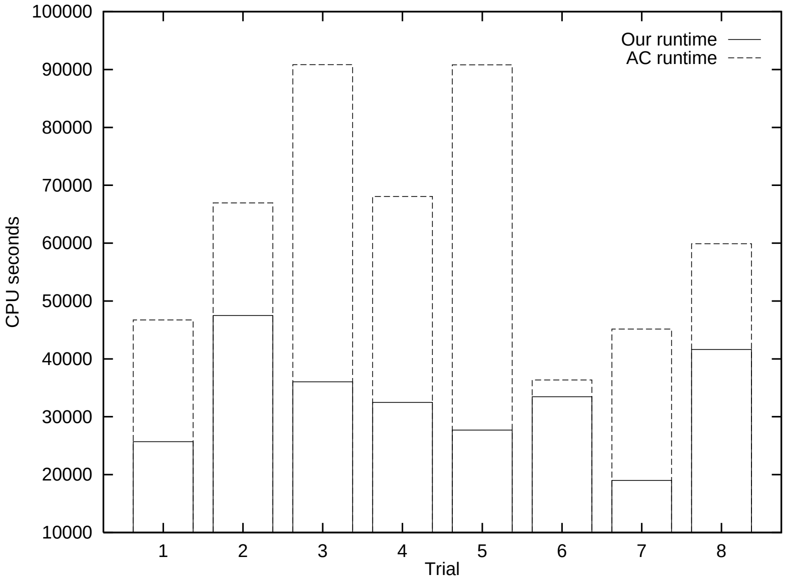


Figure 13: Comparison of the running time of our algorithm versus the Applegate-Cook algorithm on the eight instances of size 65536. Note the large variance in running times and the seeming lack of correlation between the times taken by the two algorithms.

on the classes of instances being considered. However, in this case, the knowledge of the worst-case performance gives us some peace of mind: our algorithm will never perform embarassingly poorly.

This study shows several directions for future research. We would like to be able to prove something about the algorithm's behavior on random instances on the unit square; for instance, it would be nice to prove that the solution converges almost surely to $\beta\sqrt{n}$, or that the fraction of vertices of degree k in F' converges almost surely to ν_k. This would seem to be a difficult task, however. With regard to practicality, we see that further enhancements in the implementation are necessary in order to be competitive with the best implementation of Edmonds' algorithm. These implementations, including the Applegate-Cook implementation, usually solve the instance on a sparse subgraph first, then correct the solution afterwards; it might be possible to do something similar with the approximation algorithm.

Acknowledgements

We are very grateful to David Johnson and David Applegate for their many helpful suggestions. In particular, David Johnson pointed us to the ideas of Bentley and Friedman [6] and Bentley [5]. We are also grateful to them for enabling these experiments: most of the experiments were run on a machine at Bell Laboratories. In addition, some early work on the algorithm implementation was performed while the first author was visiting Bell Laboratories. We are grateful to David Applegate and Bill Cook for allowing us to use their matching code. We thank Mark Hickman for some helpful discussions on statistics and on using statistical packages. David Wald provided the PostScript code for generating Figures 4 and 5.

The research in this paper was partially supported by DARPA contract N00014-92-J-1799. The first author was additionally supported by an NSF graduate fellowship, and the research was conducted while he was a graduate student at MIT. The second author was additionally supported by Air Force contract AFOSR-89-0271.

References

[1] D. Applegate and W. Cook. Solving large-scale matching problems. In *DIMACS implementation challenge workshop: Algorithms for network flow and matching.* DIMACS Technical Report 92-4, 1992.

[2] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13:475–493, 1983.

[3] J. Beardwood, J. Halton, and J. Hammersley. The shortest path through many points. *Proceedings of the Cambridge Philosophical Society*, 55:299–327, 1959.

[4] J. L. Bentley. K-d trees for semi-dynamic point sets. In *Proceedings of the 6th Annual ACM Symposium on Computational Geometry*, pages 187–197, 1990.

[5] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA Journal on Computing*, 4:387–411, 1992.

[6] J. L. Bentley and J. H. Friedman. Fast algorithms for constructing minimal spanning trees in coordinate spaces. *IEEE Transactions on Computers*, C-27:97–105, 1978.

[7] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards B*, 69B:125–130, 1965.

[8] M. Fredman, D. Johnson, L. McGeoch, and G. Ostheimer. Data structures for traveling salesmen. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 145–154, 1993.

[9] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

[10] H. N. Gabow, M. X. Goemans, and D. P. Williamson. An efficient approximation algorithm for the survivable network design problem. In *Proceedings of the Third MPS Conference on Integer Programming and Combinatorial Optimization*, pages 57–74, 1993.

[11] M. X. Goemans and D. P. Williamson. A general approximation technique for constrained forest problems. *SIAM Journal on Computing*, 24:296–317, 1995.

[12] M. Iri, K. Murota, and S. Matsui. Heuristics for planar minimum-weight perfect matchings. *Networks*, 13:67–92, 1983.

[13] P. Jaillet. Rates of convergence of quasi-additive smooth Euclidean functionals and application to combinatorial optimization problems. *Mathematics of Operations Research*, 17:964–980, 1992.

[14] M. Jünger and W. Pulleyblank. New primal and dual matching heuristics. Research Report 91.105, Universität zu Köln, 1991.

[15] R. M. Karp and J. Steele. Probabilistic analysis of heuristics. In E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys, editors, *The Traveling Salesman Problem*, chapter 6. John Wiley and Sons, Chichester, 1985.

[16] C. H. Papadimitriou. The probabilistic analysis of matching heuristics. In *Proceedings of the 15th Annual Allerton Conference on Communication, Control, and Computing*, pages 368–378, 1978.

[17] C. Redmond and J. Yukich. Limit theorems and rates of convergence for Euclidean functionals. To appear in the *Annals of Applied Probability*, 1994.

[18] G. Reinelt. TSPLIB – A traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.

[19] E. Reingold and K. Supowit. Probabilistic analysis of divide-and-conquer heuristics for minimum weighted Euclidean matching. *Networks*, 13:49–66, 1983.

[20] W. T. Rhee and M. Talagrand. A sharp deviation inequality for the stochastic traveling salesman problem. *Annals of Probability*, 17:1–8, 1989.

[21] J. Steele. Notes on probabilistic and worst case analyses of classical problems of combinatorial optimization in Euclidean space. Lectures given at Cornell University, 1987.

[22] J. M. Steele. Complete convergence of short paths and Karp's algorithm for the TSP. *Mathematics of Operations Research*, 6:374–378, 1981.

[23] J. M. Steele. Subadditive Euclidean functionals and nonlinear growth in geometric probability. *Annals of Probability*, 9:365–376, 1981.

[24] J. M. Steele, L. A. Shepp, and W. F. Eddy. On the number of leaves of a Euclidean minimal spanning tree. *Journal of Applied Probability*, 24:809–826, 1987.

[25] K. J. Supowit, D. A. Plaisted, and E. M. Reingold. Heuristics for weighted perfect matching. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 398–419, 1980.

[26] M. Weber and T. Liebling. Euclidean matchings problems and the metropolis algorithm. *Zeitschrift für Operations Research*, 30:A85–A110, 1986.

[27] J. Yukich. Quasi-additive Euclidean functionals. To appear in the Proceedings of the workshop *Probability and Algorithms*, I.M.A., Minneapolis, 1993.