QuickMatch: A Very Fast Algorithm for the Assignment Problem

March 1993 WP # 3547-93

James B. Orlin*
Yusin Lee*

* Sloan School of Management
Massachusetts Institute Technology
Cambridge, MA 02139

QuickMatch: A Very Fast Algorithm for the Assignment Problem

by

Yusin Lee
and James B. Orlin

Abstract

In this paper, we consider the linear assignment problem defined on a bipartite network $G = (U \cup V, A)$. The problem may be described as assigning each person in a set U to a set V of tasks so as to minimize the total cost of the assignment. The cost of assigning person i to task j is c_{ij} if $(i, j) \in A$ and is infinite otherwise. This paper describes a new algorithm called *QuickMatch* for solving the assignment problem. QuickMatch is based on the successive shortest path (SSP) algorithm for the assignment problem, which in turn is a modification of Kuhn's primal dual algorithm. However, we have added several natural heuristics that speed up the performance of the standard successive shortest path algorithm by several orders of magnitude. We present some theoretical justifications as to why the algorithm's performance is superior in practice to the usual SSP algorithm, and we provide computational support as well. In addition, the theoretical analysis and computational testing supports (but does not prove) the hypothesis that QuickMatch runs in linear time (i.e., in expected time $O(m)$) on randomly generated sparse assignment problems.

1 Introduction

We consider the assignment problem on a network $G = (U \cup V, A)$ in which both U and V contain n nodes, and the arc set A has m arcs. This paper presents *QuickMatch*, a very fast algorithm for solving the assignment problem. We present strong empirical evidence of the efficiency of Quickmatch in practice, including evidence that supports the hypothesis that QuickMatch runs in linear time on randomly randomly generated assignment problems. QuickMatch improves upon the *successive shortest path* (SSP) algorithm for the assignment problem, which solves the assignment problem as a sequence of n shortest path problems.

We present two variants of the successive shortest path algorithm. The first variant solves the assignment problem as a sequence of n shortest path problems and has a com-

parable worst case bound to that of the usual implementation of SSP, but an improved average case performance, as measured empirically. QuickMatch is a further improvement of the SSP algorithm that apparently reduces the average running time to $O(m)$ on randomly generated problems. This improvement in the average performance comes at an expense of degrading the worst case bound by a factor of $O(logn)$. If QuickMatch does solve randomly generated assignment problems in linear time, then the time to solve assignment problems is comparable (to within a constant) to the time to generate these problems.

We have explored the possibility that the growth in running time is asymptotically worse than linear by a slowly growing function in n. In order to justify the claim of a linear time algorithm, we have proved that the algorithm runs in linear time if a certain convergence property is satisfied, and then we have provided computational evidence that is strongly suggestive of the convergence property, although it is not in and of itself a rigorous proof.

1.1 The assignment problem

Consider a problem of matching n persons to n tasks where for each person i and for each task j there is an associated cost c_{ij} of assigning person i to task j. The assignment problem is the problem of matching n persons to n tasks so as to minimize the total cost. Although the assignment problem is traditionally phrased in terms of assigning persons to tasks, it also models applications in a wide range of different settings. For example, the assignment problem also has applications in vehicle routing and signal processing, and it is an important relaxation of the traveling salesman problem. For a survey of the applications, see Ahuja, Magnanti, and Orlin [1].

The assignment problem is arguably one of the three most important subproblems of the minimum cost flow problem. (The other two are the shortest path problem and the maximum flow problem.) Tens of papers have been written which discuss computational

aspects of the assignment problem. Some of the authors who have proposed solution techniques for the assignment problem include Akgul [2], Balinski [3], Bertsekas [5], Gabow and Tarjan [6], Hung [7], Hung and Rom [8], Karp [9], Kennington and Wang [10], Lotfi [11], and Megson and Evans [12].

In section 2 of this paper, we briefly review the successive shortest path algorithm, the concept of the residual network, and describe the QuickMatch algorithm. We also provide an average case running time analysis of the QuickMatch algorithm. In section 3, we provide computational results on randomly generated networks. We present most of our computational results in the form of combinatorial counts instead of CPU time so that the results are largely independent of the computing environment. Also we compare our run time with two other codes.

We use n for the number of node pairs in an assignment problem instance and m for the number of arcs in an instance throughout the paper. Also all log() functions are of base 2 unless otherwise stated.

We view the primary contributions of this paper as two-fold:

1. First, we have developed a significant improvement over the traditional SSP algorithm for the assignment problem. Its running time is superior to all other implementations that we have seen of the assignment problem with the exception of the reverse auction algorithm of Bertsekas and Castenon [4]. Their algorithm appears to be faster by a small constant factor (less than 3 for problems with $n = 20,000$), and we do not know if this improvement is a result of improved implementation details and/or compiling or whether it is intrinsic to the algorithm.

2. We have used computational testing in conjunction with theoretical analysis to help establish an average case running time.

2 The Successive Shortest Path Algorithm

Let $G = (V \cup U, A)$ be the network for the assignment problem. The node set V represent the set of persons, and the node set U represents the set of tasks. Whenever a person i can carry out task j, arc (i, j) is in A. We do not assume that each person can carry out each task, but we do assume that there is a feasible assignment. The assumption of feasibility is without loss of generality since we may always add artificial arcs (i, i) for $i = 1$ to n with very large cost and create an artificial assignment. None of these arcs would appear in an optimal assignment, assuming that one exists.

The standard integer programming version of the assignment problem is defined as follows:

let

$$x_{ij} = \begin{cases} 1 & \textit{if person } i \textit{ is assigned to task } j \\ 0 & \textit{otherwise.} \end{cases}$$

$$\textit{Minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1a}$$

$$\textit{Sujbect to} \sum_{i\in N} x_{ij} = 1 \qquad \forall j = 1, \ldots, n, \tag{1b}$$

$$\sum_{j\in N} x_{ij} = 1 \qquad \forall i = 1, \ldots, n, \tag{1c}$$

$$x_{ij} \geq 0, \ \textit{integral} \qquad \forall\, i, j. \tag{1d}$$

The assignment problem is a special case of the transportation problem, which in turn is a special case of the minimum cost flow problem. As is well known, all corner points of the feasible region are integral, and so the integrality constraints may be relaxed without loss of generality.

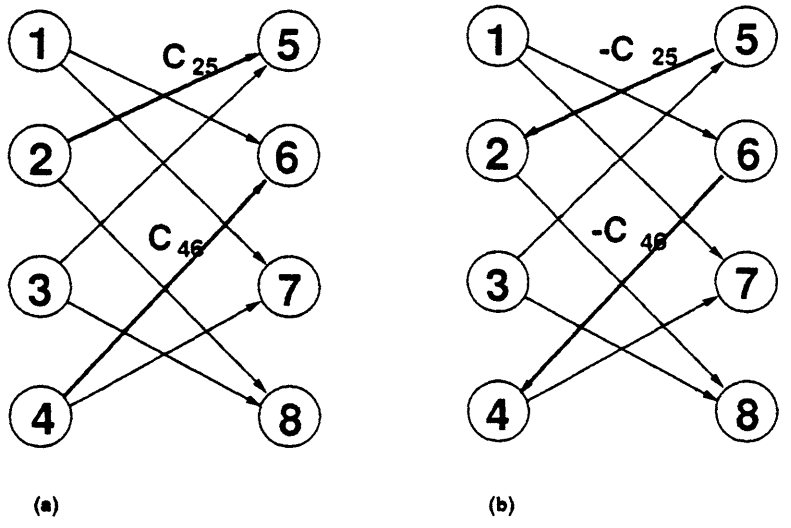


Figure 1: Illustration of residual graph of an assignment problem network.

2.1 The Residual Network and the Optimality Conditions

We refer to a solution x satisfying constraints (1b) and (1c) with inequality ($\leq$) rather than with equality as a partial assignment. If the constraints are satisfied with equality, then x is called a *complete assignment* or is called an *assignment.* For each partial assignment x, we define the residual network $G(x)$ as follows. For each arc $(i,j) \in A$, there is an associated arc (i,j) with cost c_{ij} in $G(x)$. For each matched arc (i,j), there is an additional associated arc (j,i) in $G(x)$ whose cost is $-c_{ij}$. We illustrate the residual network in Figure 1. The figure illustrates an assignment problem network of 8 nodes. Figure 1a shows a partial assignment matching two pairs of nodes, namely node 2 with node 5, node 4 with node 6. The cost of the matching arcs are c_{25} and c_{46}. Figure 1b shows the corresponding residual network. In the residual network there are arcs $(5,2)$ and $(6,4)$, with cost $-c_{25}$ and $-c_{46}$, respectively. The residual network is a standard tool for minimum cost flow problems. See for example Ahuja et al [1].

Associated with each node i is a node potential π_i, sometimes referred to as the dual variable for node i. The reduced cost c_{ij}^{π} of arc (i,j) is defined as $c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j$. The reader should note that we are defining reduced costs in a manner more commonly

associated with minimum cost flow problems. In particular, a $+\pi_j$ term is used instead of $-\pi_j$.

The following lemma is well known, easily proved, and is stated here without proof.

Lemma 1 *Let x^* be a feasible assignment and let π be any vector of node potentials. The solution x^* is optimal for the assignment problem with respect to costs c if and only if x^* is also optimal for the assignment problem obtained upon replacing c by c^π.* ∎

By selecting the node potentials carefully and using the results of Lemma 1 and Lemma 2 below, one can verify if an assignment x^* is optimal.

Lemma 2 (Optimality Conditions) *An assignment x^* is optimal if there is a set of node potentials π such that $c^\pi_{ij} \geq 0$ for all arcs (i,j) in $G(x^*)$. In particular, $c^\pi_{ij} = 0$ for all matched arcs (i,j).*

Proof: This result is well known. To see that the second statement of Lemma 2 follows from the first statement, note that $c^\pi_{ij} = c^\pi_{ji}$ in the residual network, and both of these costs are non-negative. ∎

2.2 Augmentations and the Successive Shortest Path Algorithm

An augmentation in $G(x)$ refers to a directed path P from an unmatched node in U to an unmatched node in V. We will refer to unmatched nodes in U as *origin nodes*, and we will refer to unmatched nodes in V as *destination nodes*. Note that in any augmenting path, the arcs directed from node set U to node set V are unmatched, and the arcs directed from V to U correspond to matched arcs in x. The cost of the augmenting path P is defined to be the sum of the costs of the arcs of the path in the residual network, and we denote this as $c(P)$. To augment along the path P is to replace x_{ij} by 1 for each arc $(i,j) \in P$ directed from U to V, and to replace each x_{ij} by 0 for each arc $(j,i) \in P$ directed from V to U. Suppose that x' is the matching obtained from x after augmenting along path P. Then the cost of x' is easily shown to be $cx' = cx + c(P)$.

The following algorithm is known as the successive shortest path algorithm for the assignment problem.

Algorithm SSP
begin
  $x = 0$;
  while some node is free **do**
  begin
    select an origin node i;
    in the residual network, find the minimum cost augmenting path P from i to some free destination node t;
    augment along the path P;
    update data structures appropriately;
  end
end

This description of the algorithm is at a very high level, and an efficient implementation of the algorithm relies on determining shortest augmenting paths quickly. We shall give some heuristics to speed up the algorithm.

As is common for the successive shortest path algorithms, we adopt a "primal-dual" approach. By this, we mean that at each iteration we will maintain a set of node potentials so that the optimality conditions are satisfied, i.e., each arc of the residual network has a nonnegative reduced cost. The modification of node potentials in the residual network does not effect which paths have minimum cost from origin to destination, as a consequence of Lemma 3 below. The lemma is well known and is stated without proof.

Lemma 3 *Let G be any directed network, and let i and j be any two nodes of G. Suppose that P is a minimum cost path in G from node i to node j with respect to cost vector c. Then for any vector of node potentials π, P is also a minimum cost path from i to j with respect to costs c^{π}.* ∎

In searching for a minimum cost augmenting path from an origin node to a destination node, we will modify the node potentials in such a way that all arcs on some path P from origin to destination have 0 cost. Such a path must be a minimum cost path in the residual network since all paths in the residual network have a nonnegative cost. We can then augment along P and create a larger matching.

In order to modify the node potentials, we rely on shortest path distances.

Lemma 4 *Let G be a network with arc costs c. Let π be a set of node potentials for which $c^\pi \geq 0$. For each node j, let $\mathcal{L}(j)$ denote the shortest path distance to j from an origin node s. Let K be any integer. Finally, for each j, let $\gamma(j) = \pi(j) + max(0, K - \mathcal{L}(j))$. Then $c^\gamma \geq 0$. Moreover, for any arc (i,j) with $\mathcal{L}(j) = \mathcal{L}(i) + c^\pi_{ij} \leq K$, it follows that $c^\pi_{ij} = 0$.*

Proof: Consider first the case that $\pi = 0$. The case for $\pi \neq 0$ is essentially the same. Let (i,j) be any arc of G. By the optimality conditions for the shortest path problem, $\mathcal{L}(j) \leq \mathcal{L}(i) + c_{ij}$, or equivalently $c_{ij} + \mathcal{L}(i) - \mathcal{L}(j) \geq 0$. Consider first the case that $\mathcal{L}(i) \leq K$ and $\mathcal{L}(j) \leq K$. In this case, $\gamma(i) = K - \mathcal{L}(i)$ and $\gamma(j) = K - \mathcal{L}(j)$, and thus $c^\gamma_{ij} = c_{ij} + \mathcal{L}(i) - \mathcal{L}(j) \geq 0$. (Note that if $\mathcal{L}(j) = \mathcal{L}(i) + c_{ij}$, then $c^\gamma_{ij} = 0$). We now consider the case that $\mathcal{L}(i) \geq K$ and $\mathcal{L}(j) \geq K$. In this case, $\gamma(i) = \gamma(j) = 0$, and so $c^\gamma_{ij} = c_{ij} \geq 0$. Now consider the case that $\mathcal{L}(i) \leq K$ and $\mathcal{L}(j) \geq K$. In this case, $\gamma(i) = K - \mathcal{L}(i)$ and $\gamma(j) \geq K - \mathcal{L}(j)$, and so $c^\gamma_{ij} \geq c_{ij} + \mathcal{L}(i) - \mathcal{L}(j) \geq 0$. Finally, we consider the case that $\mathcal{L}(i) \geq K$ and $\mathcal{L}(j) \leq K$. In this case, $\gamma(i) = 0$ and $\gamma(j) \geq 0$, and so $c^\gamma_{ij} \geq c_{ij} \geq 0$. This completes the proof of the lemma. ∎

There are various ways to employ the results of Lemma 4. Each of these ways has the nice property that it preserves the non-negativity of arc costs in the residual network and thus permits one to continue to run Dijkstra's algorithm. Consider, for example, selecting s to be an origin node of the residual network and selecting K to be an upper bound on the maximum distance from node s. This choice is equivalent to the successive shortest path algorithm as it is usually presented in the literature. After the modification of the node potentials, each arc in the shortest path tree directed from node s has a reduced cost of 0. (This is a consequence of Lemma 4 and the fact that each arc (i,j) on the shortest path tree satisfies $\mathcal{L}(j) = \mathcal{L}(i) + c_{ij}$.) One can then select any destination node t reachable from s, and augment along the unique path in the tree from s to t.

An alternative method for using Lemma 4 is to define K to be $\min(\mathcal{L}(j) : \mathcal{L}(j) > 0)$. Although this choice of K may seem too low to be efficient, the modification of the node potentials that it induces yields the original primal dual algorithm of Kuhn.

Our choice for the value of K from Lemma 4 is to define K to be $\min(\mathcal{L}(j) : j$ is a destination node). This choice ensures that there is a 0-cost path in the residual network from an origin node to a destination node. In addition, it is more economical in its implementation of Dijkstra's algorithm in that it terminates Dijkstra's algorithm as soon as an augmentation is discovered. This choice of K is exploiting the following well-known property of Dijkstra's algorithm.

Property 1 *Dijkstra's algorithm permanently labels nodes in non-decreasing order of their distance from the origin node.* ∎

In addition, the time to update the dual prices is not necessarily proportional to the number n of nodes. Rather it is proportional to the number of nodes made permanent, which in practice is far fewer than n. This modification is very efficient in practice; however, we note that this speed-up does not improve the worst case performance of the algorithm. We now summarize in pseudo-code the modification of the successive shortest path algorithm.

Algorithm Modified SSP
begin
  $x = 0$; $\pi = 0$;
  while some node is unmatched **do**
  begin
    select an origin node i;
    run Dijkstra's algorithm to grow a shortest path tree rooted at i until a destination node t is made permanent;
    let $\mathcal{L}(j)$ denote the shortest path length from i to j in the residual network using reduced costs c^{π}.
    replace $\pi(k)$ by $\pi(k) + \mathcal{L}(t) - \mathcal{L}(k)$ for each node k with $\mathcal{L}(k) < \mathcal{L}(t)$;
    augment along the shortest path from i to t;
  end
end

The modified SSP algorithm dramatically reduces the running time of Dijkstra's subroutine over running it to completion. Based on experimental results obtained on

randomly generated graphs, we believe that the average number of nodes labeled permanently by Dijkstra's algorithm is reduced from n to some number between $\log^2 n$ and $n^{1/2}$.

We now give a theorem that is suggestive as to why the modified SSP algorithm runs orders of magnitude faster than the original SSP algorithm .

Theorem 1 *Let k denote the number of matched arcs at some stage in the algorithm. Suppose that the probability of selecting a destination node next in Dijkstra's algorithm is at least $(n-k)/(nf(n))$ for some function $f()$ that depends only on n. Then the expected number of permanently labeled nodes in total for the modified SSP algorithm is $O(f(n)n\log n)$.*

Proof: Suppose first that we are running the successive shortest path algorithm starting with a partial assignment x in which k arcs are matched. Since the probability of selecting a destination node next is at most $O(n-k)/(nf(n))$, it follows that the expected number of permanently labeled nodes is 1 plus the number of consecutive non-destination nodes which is $O(nf(n)/(n-k))$. If we sum this amount for $k=1$ to $n-1$, the total number of permanently labeled nodes is $O(nf(n)\sum_k 1/(n-k)) = O(nf(n)\log n)$. In the case that $f(n)=1$, i.e., under the assumption that all nodes are equally likely to be selected next, the average number of permanently labeled nodes will be $O(n\log n)$. ∎

A plausible a priori assumption is that $f(n)=O(1)$, i.e., that the destination nodes are comparably likely to be selected next as non-destination nodes. On some limited experimental evidence, we rejected this hypothesis. Rather it seemed more likely that $f(n)$ was behaving like $O(\log n)$ on randomly generated problems. Even with $f(n) = O(\log n)$, the running time is orders of magnitude faster than the usual SSP algorithm.

We now improve the Modified SSP algorithm by adding two other heuristics. First of all, in running Dijkstra's algorithm, it is common to start at an origin node and look for a shortest augmenting path from the origin node to some destination node. However,

it is also possible to start the algorithm at the destination node, and search backwards finding shortest paths directed into the destination node until permanently labeling an origin node. This algorithm, called Reverse Dijkstra's Algorithm, is mathematically equivalent to starting Dijkstra's algorithm from the destination node and reversing all arcs of the network. We will refer to the usual method for implementing Dijkstra's algorithm as Forward Dijkstra's Algorithm to emphasize its difference from Reverse Dijkstra's Algorithm.

The other heuristic is to set a threshold T on the number of nodes labeled permanently by Dijkstra's algorithm. We refer to a T-phase as a sequence of searches for augmenting paths each of which labels at most T nodes permanently. Our rule in a T-phase is to permit each unmatched node to be a root node for Dijkstra's algorithm at most once. The search from an origin node terminates when either a destination node is reached or when T nodes are labeled permanently, whichever occurs first. Similarly, the search into a destination node terminates when either an origin node is permanently labeled or when T nodes are permanently labeled. Initially T is set to 2, and T is doubled at the end of each phase. We now describe the algorithm in more detail.

Algorithm QuickMatch
begin
  $x = 0$; $\pi = 0$;
  $T := 2$;
  for OuterCounter = 1 to $\lceil \log n \rceil$ **do**
  begin
    mark all unmatched nodes as unscanned;
    while there is an unscanned node **do**
    begin
      begin Forward Dijkstra's Algorithm
        select an unscanned origin node i
        mark i as scanned;
        destination node t is made permanent or until T nodes are labeled as permanent;
        let $\mathcal{L}^*$ denote the label of the most recently labeled permanent node;
        replace $\pi(k)$ by $\pi(k) + \mathcal{L}^* - \mathcal{L}(k)$ for each node k with $\mathcal{L}(k) < \mathcal{L}^*$;
        if a destination node t is made permanant, **then** augment along the shortest path from i to t and mark t as scanned;
      end
      if there is an unscanned node **then**
      begin Reverse Dijkstra's Algorithm

```
            select an unscanned destination node t;
            mark t as scanned;
            run Reverse-Dijkstra's algorithm to grow a shortest path tree rooted into node
               t until an origin node s is made permanent or until T nodes are labeled as
               permanent;
            let L* denote the label of the most recently labeled permanent node;
            replace π(k) by π(k) − L* + L(k) for each node k with L(k) < L*;
            if an origin node s is made permanant then augment along the minimum
               cost path from s to t and mark s as scanned;
         end
      end
      if all nodes are matched then quit;
      T := T × 2;
   end
end
```

The idea of alternating shortest path computations between origin nodes and destination nodes is also one of the primary aspects of the forward-reverse auction algorithm of Bertsekas and Castenon.

We now give a Lemma that suggests why alternating between Forward Dijkstra's Algorithm and Reverse Dijkstra's Algorithm in this manner may contribute to a reduced running time. More precisely, it is the combination of using a threshold value and the alternations that speeds up the algorithm.

We call a matched node i a *pseudo-origin* node if there is a directed path from an origin node to i consisting entirely of arcs whose reduced cost is 0. We denote the corresponding origin node as *Root*(i). Similarly, we call a matched node j a *pseudo-destination* node if there is a directed path from j to a destination node consisting entirely of arcs whose reduced cost is 0. We denote the corresponding destination node as *Root*(j). We assume for convenience that Root(i) is unique. If there are several 0-cost paths from origin nodes to i we can select any of these origins as Root(i). Similarly, we assume that Root(j) is unique. The advantage of pseudo-origin nodes and pseudo-destination nodes i is described in the next Lemma.

Lemma 5 *If Reverse-Dijkstra's Algorithm to destination j permanently labels a pseudo-origin node i, then subsequent to the modification of the node potentials there will be a 0-reduced cost path from Root(i) to node j. Similarly, if Forward-Dijkstra's algorithm*

from origin i permanently labels a pseudo-destination node j, then subsequent to the modification of the node potentials there will be a 0-reduced cost path from i to Root(j).

Proof: Let us consider only the case that we are running Dijkstra's algorithm from an origin node i. If node j is labeled permenantly, then subsequent to the change in node potentials, the path from node i to node j will have a 0 reduced cost. We now claim that $\mathcal{L}(Root(j)) = \mathcal{L}(j)$. To see this, note that $\mathcal{L}(Root(j) \geq \mathcal{L}(j)$ since otherwise $Root(j)$ would have been labeled prior to labeling node j, but then the shortest path algorithm would have terminated upon labeling $Root(j)$ as permanent, contrary to assumption. We also see that $\mathcal{L}(Root(j)) \leq \mathcal{L}(j)$ since there is a 0-cost path from j to $Root(j)$. It follows that $\mathcal{L}(Root(j)) = \mathcal{L}(j)$. We now claim that $\mathcal{L}(j)$ is the largest permanent label of a node at the termination of Dijkstra's algorithm. This claim follows from the fact that $Root(j)$ will be permanently labeled prior to labeling any node whose distance label exceeds $\mathcal{L}(j)$, and thus the algorithm will terminate prior to labeling any node whose distance label exceeds $\mathcal{L}(j)$. ∎

Lemma 5 suggests that once Forward Dijkstra's algorithm permanently labels a pseudo-destination node j, then the procedure will discover an augmenting path to Root(j). This detail depends on the implementation. We chose to terminate Forward-Dijkstra after labeling Threshold nodes even if this caused the algorithm to stop short of identifying a 0-cost augmenting path; however, it is generally true in practice that the augmenting path to Root(j) will be discovered shortly after permanently labeling j. Thus, permanently labeling a pseudo-destination node typically means that only a few more nodes will be labeled before an augmenting path is determined. We considered enforcing augmentations along 0-cost augmenting paths once a pseudo-destination node is labeled in Forward Dijkstra's algorithm, but preliminary testing indicated that this would not noticeably speed up our implementation. We chose not to implement this feature in our final algorithm. To summarize, it is almost as beneficial to permanently

label a pseudo-destination node as it is to label a real destination node.

The primary method for creating pseudo-destination nodes is through running reverse Dijkstra's algorithm in such a way that it terminated prior to an augmentation. This is a consequence of Lemma 5 which states that all arcs on the shortest path will have a 0-reduced cost. Similarly, the primary method for creating pseudo-origin nodes is through running forward Dijkstra's algorithm and terminating prior to an augmentation. In other words, if forward Dijkstra ends in an augmentation, then this is quite beneficial to the algorithm since it adds a matched arc (although it may have the negative effect of eliminating one or several pseudo-origin nodes and pseudo-destination nodes) . If forward Dijkstra does not end in an augmentation, then this is also beneficial to the algorithm since it typically (but not always) adds new pseudo-origins. However, one can take advantage of the new pseudo-origins only if one is running Reverse Dijkstra's algorithm. And one can only take advantage of pseudo-destination nodes if one is running forward Dijkstra's algorithm. To summarize, terminating early creates pseudo-origins and destinations which are potentially of value in subsequent shortest paths; however, to take advantage of the pseudo-origins and pseudo-destinations it is necessary to alternate at some frequency between forward Dijkstra and reverse Dijkstra.

The value of creating pseudo-origins and destinations is even greater since pseudo-origins and pseudo-destinations are preserved between successive augmentations, as stated in the next theorem.

Theorem 2 *Suppose that node i is a pseudo-origin or a pseudo-destination. Suppose further that the algorithm performs an augmentation whenever there is a 0-cost path from an origin to a destination. Then node i continues to be a pseudo-origin or a pseudo-destination until the time at which an augmentation takes place.*

Proof: Let us assume that node i is a pseudo-origin, and that P is a path from origin s to node i consisting of arcs whose reduced cost is 0. (The case that i is a pseudo-

destination can be proved in the same way.) Suppose that forward Dijkstra's algorithm relabels distances in such a way that the node potential changes for some $j \in P$. Let t be the last node made permanent in the shortest path algorithm run from node s. Then $d(j) < d(t)$ for otherwise the distance label of j would not change. But Dijkstra's algorithm would then have permanently labeled node i as well since $d(i) \leq d(j)$. It follows that subsequent to the shortest path subroutine from node s, i is again a pseudo-origin and its root node is s.

Next let us consider the case that reverse Dijkstra's algorithm into t labels a node of P. In this case, by Lemma 5, after the change in node potentials there is an augmenting path from Root(i) to t, and so the theorem is valid in this case too. ∎

2.3 The Average Case Running Time of the Algorithm

In this section, we describe the running time of the algorithm in terms of a convergence rate of the algorithm. If the convergence rate is at a sufficiently fast rate, which appears to be validated by computational experiments, then the running time of the algorithm is linear in the number of arcs of the network.

Let *Unmatched(k)* denote the number of node pairs that are unmatched at the beginning of the k-th phase of the algorithm, i.e., the phase at which the number of permanent nodes that are permitted to be labeled is 2^k. Let $Perm(k)$ denote the number of nodes made permanent by Dijkstra's algorithm during the k-th phase.

Convergence Rate Conjecture. There exists some constant $\alpha < 1/2$ such that $Unmatched(k) < n\alpha^{k-1}$.

Degree Bound Property. The number of arcs incident to each node is $O(d)$, where $d = m/n$.

We note that the Degree Bound Property will be true for almost all randomly generated networks. We do not know of a proof of the convergence rate conjecture for randomly generated graphs. However, in the next section, we provide computational

evidence that supports this conjecture.

Theorem 3 *Suppose that Quickmatch is applied to networks generated from some probability distribution X and that the Convergence Rate Conjecture and Degree Bound Property are both satisfied. Then Quickmatch runs in $O(m)$ (i.e., linear) time on these networks.*

Proof: The number of shortest path problems solved at the k-th stage is at most $O(n\alpha^{k-1})$ by the Convergence Rate Conjecture. During the k-th stage, each shortest path algorithm permanently labels at most 2^k nodes, and scans $O(d2^k)$ arcs. Thus the depth of the d-heap is $O(\log_d(d2^k)) = O(k/\log d)$. To delete the minimum element from such a d-heap will take time $O(dk/\log d)$. Thus, for each shortest path during the k-th phase, the time for all of the find-mins (including the deletion of the smallest element of the heap) is $O(dk2^k/\log d)$.

The number of arcs scanned at each shortest path during the k-th stage is at most $O(d2^k)$ by the Arc Bound Property. To insert any element into the d-heap will take time $O(k/\log d)$, and so the time for inserting all of the arcs from any shortest path problem is $O(dk2^k/\log d)$, which was also the time for find-mins. The total time spent during the k-th phase is at most $O(n\alpha^{k-1}dk2^k/\log d)) = O(mk\alpha^{-1}(2\alpha)^k/\log d)$. We know that $\alpha < 1/2$ and so we may choose $\alpha < \alpha' < 1/2$ and an integer K so that $k\alpha^{-1}(2\alpha)^k/\log d < (2\alpha')^k$ for all $k > K$. It follows that the running time for the k-th phase is $O(m(2\alpha')^k)$ for all k, and the running time time over all phases is $O(m)$. ∎

Note that if a binary heap were used instead of a d-heap, then the running time for the first phase alone would be $O(m \log d)$ since all arcs would be scanned, and inserting d arcs into a binary heap containing d arcs takes $O(d \log d)$ steps. Thus, we could not use binary heaps if we wanted the algorithm to run in linear time. In practice, we are mostly interested in very sparse graphs, and so $d = O(\log n)$. The increase in running time caused by using d-heaps is $O(\log \log n)$, which is a very slowly growing function of

n. For example, $\log\log 2^{32}/\log\log 256 < 2$, which implies that as networks increase from 256 nodes to 4 billion nodes, the asymptotic increase in running time accounted for by the log log term is less than 2. Nevertheless, we were pleased to include d-heaps in our algorithm because they led to a strict improvement in the asymptotic performance while they simultaneously led to a minor improvement in the observed running time.

3 Computational Results on Randomly Generated Networks

In this section, we present computational results on randomly generated networks with n nodes and m arcs. We generated the m arcs one at a time by generating an endpoint in U uniformly at random, and generating an endpoint in V uniformly at random, and then generating an integer cost uniformly at random from the interval [0,C]. In this section, we demonstrate the robustness of QuickMatch, and validate that the QuickMatch algorithm runs in linear time on these randomly generated networks. Our validation is via computational testing.

In running Forward Dijkstra's algorithm, one alternates between permanently labeling nodes in U and permanently labeling nodes in V. Once a node j in V is permanently labeled, the next permanently labeled node is automatically the node of U that is matched to j. Also, the only arcs scanned in each forward Dijkstra's algorithm turn out to be those arcs (i,k) such that $i \in U$ and i is made permanent. To summarize, each permanent labeling of a node j of V and its mate i of U takes time comparable to a single node being made permanent in the usual Dijkstra's algorithm. For this reason, we will only keep track of the number of nodes in U that are made permanent.

The first evidence that the QuickMatch algorithm runs in linear time comes from counting the average number of U nodes made permanent over all shortest path algorithms used to solve a single instance. In figure 2, we plot the histogram of the average number of U nodes made permanent per pair of nodes matched. Each of the four graphs

represent 100 different instances solved. The graphs show that for most of the instances, the average number of U nodes made permanent is less than 5 and does not vary with the problem size. In the 400 instances solved, the average number of U-nodes made permanent never exceeded 6. In fact, the average remains around 5 even for problems with 1,000,000 nodes. (This is 5 orders of magnitude better than the more direct approach of running each shortest path instance to completion. We solved a 1,000,000 node and 40,000,000 arc problem in 30 minutes on a CRAY Y-MP M92/21000, using one CPU and 370 million words of central memory. Had we solved the same instance while letting each shortest path algorithm go to completion, the running time on the same CRAY would have been over 5 years.)

3.1 The Convergence Rate

Let n_k be the number of unmatched U nodes at the beginning of the k-th phase of the algorithm. Thus, $n_0 = n$. We define the convergence rate of Quickmatch on an instance to be the smallest real number α such that $n_k \leq \alpha^k n$ for each k. We conjectured in Section 2.3 that $\alpha < 1/2$ for each instance, and we showed that QuickMatch runs in linear time if $\alpha < 1/2$. Figure 3 shows the observed α value for randomly generated instances. The data displayed in the figure are the *maximum* α values obtained in the following way: let f_k be the number of free node pairs at the beginning of phase k. For each k we calculate an α_k such that $\alpha_k = n\alpha^{k-1}$. Then we present the maximum of all α_k as the α for this network instance. Each of the four graphs represent the histogram of observed α for 100 different instances. For all but one of the 400 instances solved, the average of α was strictly less than $1/2$, and thus our conjecture appears to be correct, at least for the distributions that we considered. These networks ranged in size from networks with 2000 nodes and 20,000 arcs to networks with 16,000 nodes and 200,000 arcs. In most cases α is around 0.45. From the proof of theorem 3 we can see the number of nodes made permanent at stage k is at most $2n\alpha^{k-1}2^k$. If we plug in $\alpha = 0.45$, we

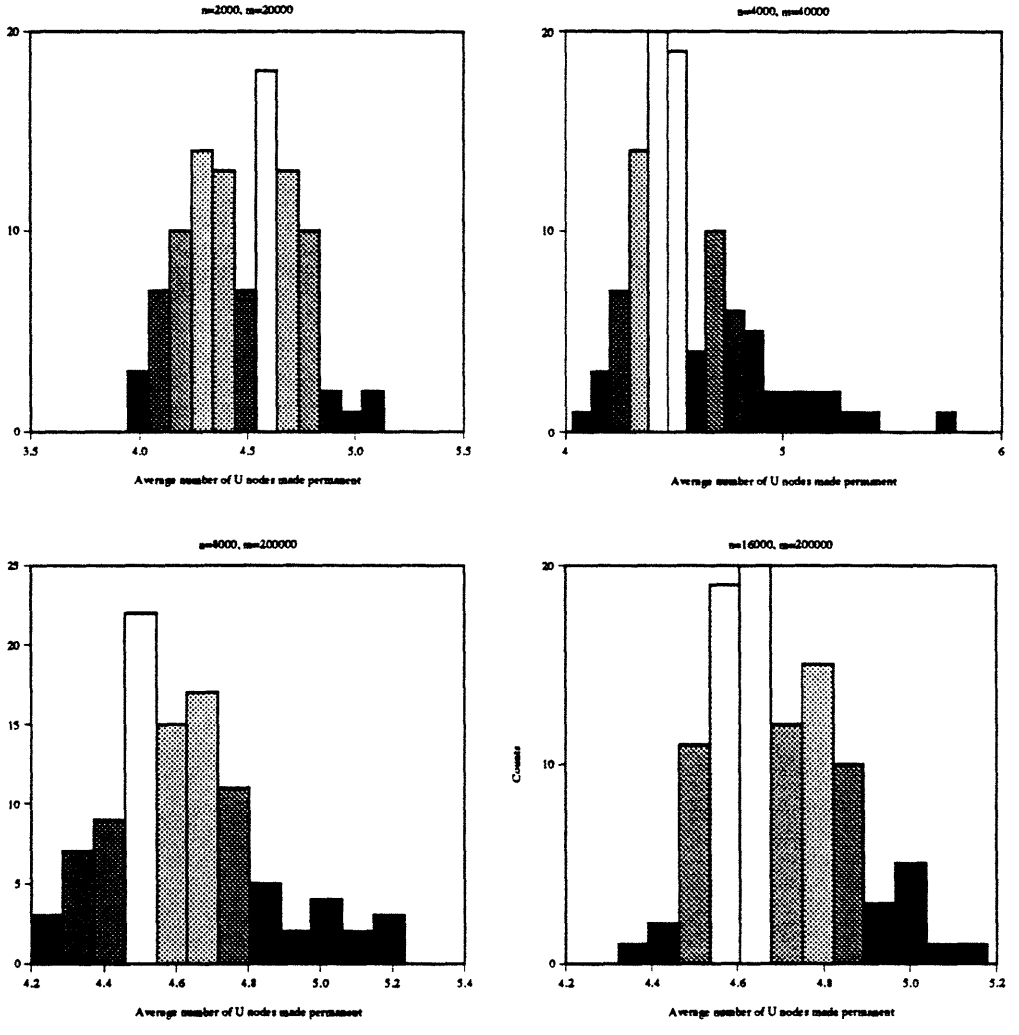


Figure 2: Average number of U nodes made permanent per pair of nodes matched.

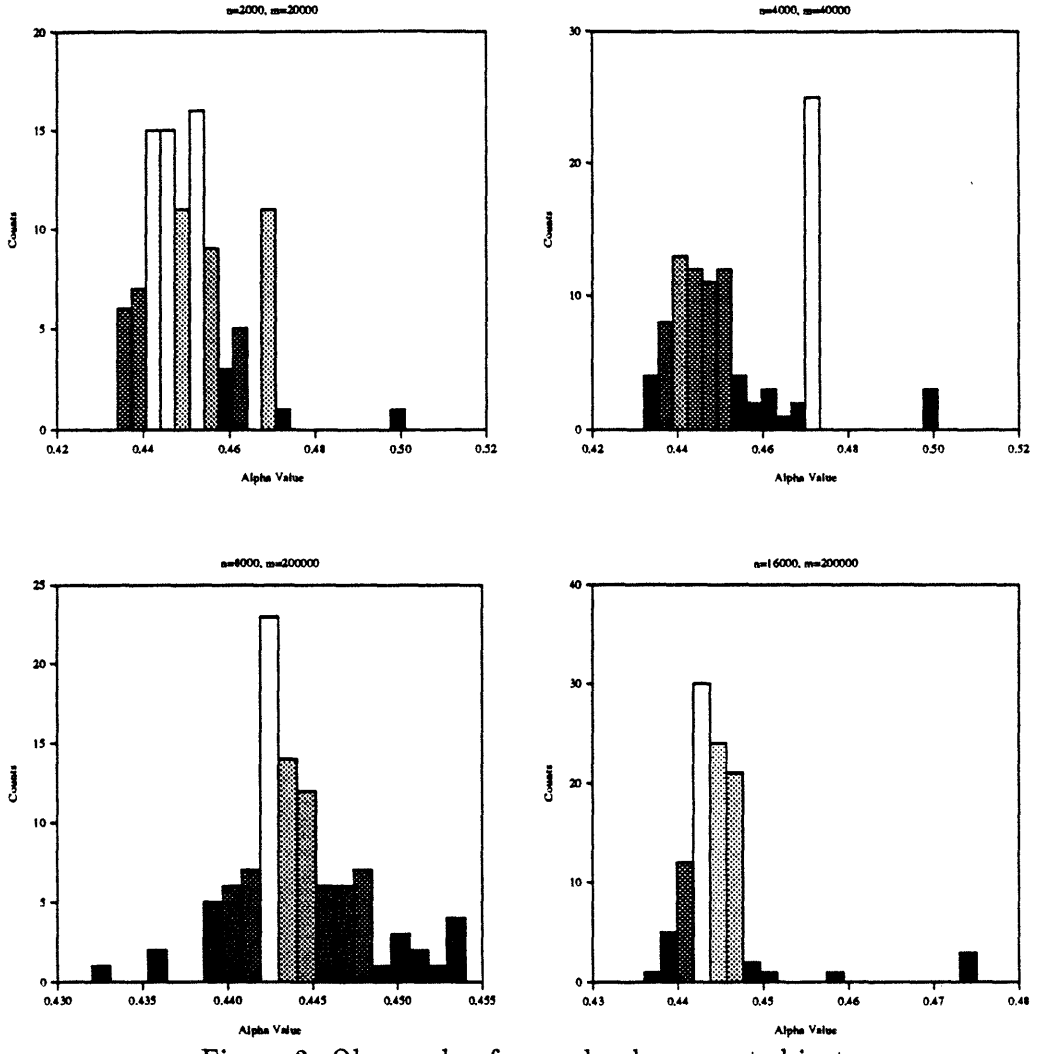


Figure 3: Observed α for randomly generated instances.

obtain that the expected number of permanent U nodes per shortest path is around 14, which indicates that the algorithm behaves better than the bound in practice.

We demonstrate that $\alpha < 1/2$ for each instance solved in Figure 4. The x-axis in Figure 4 corresponds to the stages. The y-axis corresponds to $\log n_k$. The inequality $n_k < n2^{-k}$ is equivalent to the inequality $\log n_k < \log n - k$. To demonstrate the inequality, we also graphed the line $\log n_k = \log n - k$ on each of the four graphs.

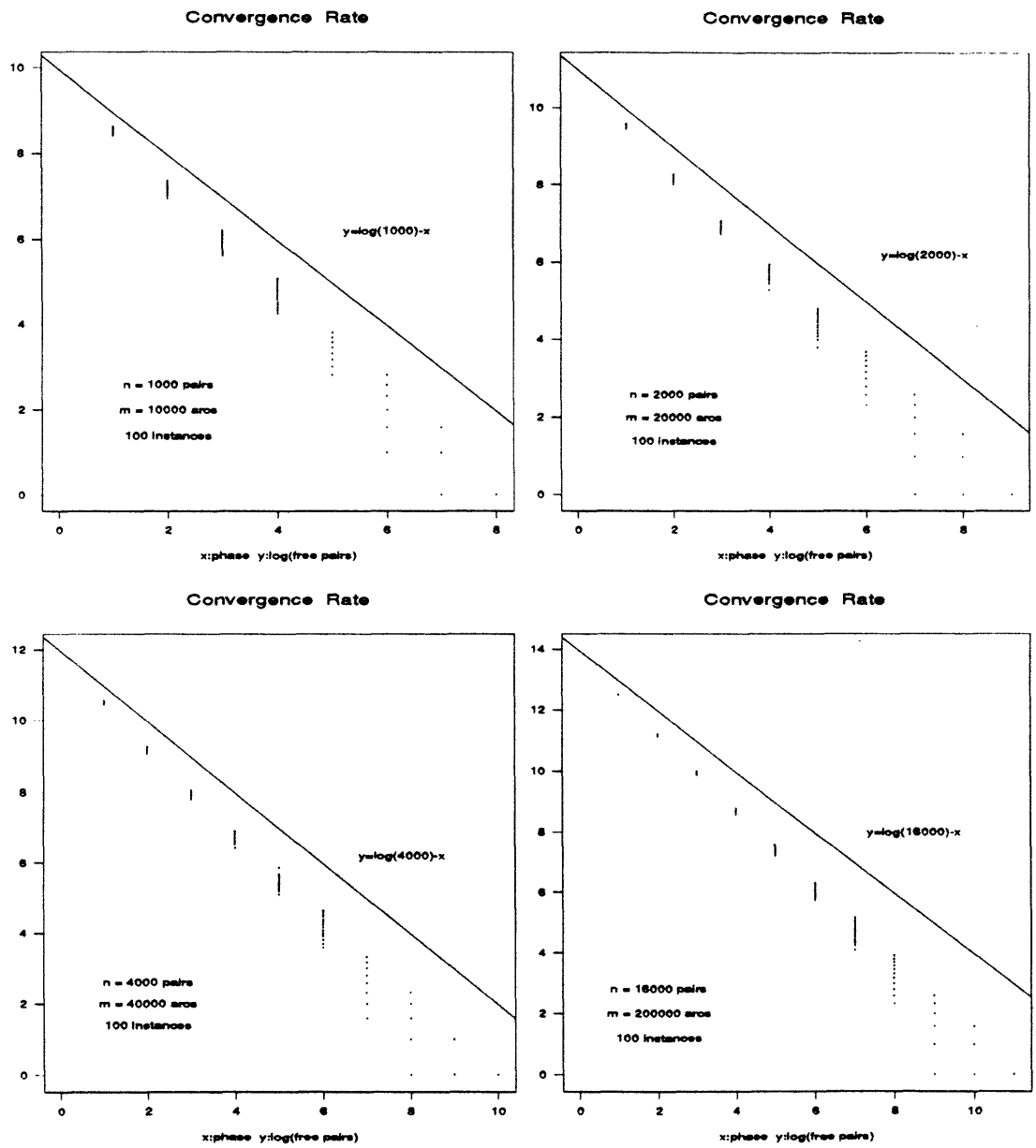


Figure 4: Number of free node pairs remaining at each stage, semi-log scale.

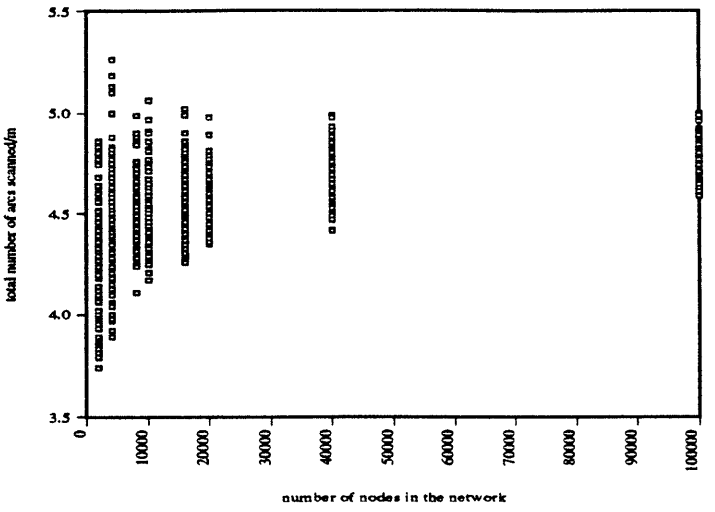


Figure 5: Total number of arcs scanned divided by m, plotted against n.

3.2 Number of Arcs Scanned

The bottleneck operation for the successive shortest path algorithm is scanning arcs emanating from permanently labeled nodes, and then inserting these arcs into a heap. Since arcs were generated randomly, one would anticipate that the number of arcs incident to a node would be approximately geometrically distributed. However, it is possible that the average number of arcs incident to a permanently labeled node would be different from $O(d) = O(m/n)$ for some subtle reason depending on the performance of the algorithm. This is important because the degree bound property is critical in the linear time proof of theorem 3.

To verify this, we examine the total number of arcs scanned to solve an instance divided by m, the total number of arcs in the instance. If the degree bound property holds, this ratio should remain constant regardless of the network size. Figure 5 shows the data plotted in semilog scale. It seems that the ratio (which corresponds to the number of times an arc is scanned on average) converges to some constant less than 5.

3.3 Relative Running Time of the QuickMatch Algorithm

In this section, we provide a measure of the running time of the QuickMatch algorithm that is relatively independent of the computational environment. (At the very least, it does not require the reader to appreciate the relative running times of different computers.) Here we use an idea also employed by Hao, Kai, and Kocur in the analysis of their matching algorithm.

The *Arc-Pricing Subroutine* refers to the algorithm that computes the reduced cost c_{ij}^{π} of every arc in the network. Clearly the Arc-Pricing Subroutine runs in time linear in the number of arcs of the network. We then computed the ratio of the running time of QuickMatch to the Arc-Pricing Subroutine. These plots are shown in Figure 6. Data for instances of $n = 2,000$, $m = 20,000$ are not shown because the run time for the Arc Pricing Subroutine are too short to be meaningful.

The plots suggest that the Arc-Pricing Subroutine is around 25 times faster than QuickMatch. This difference can be roughly accounted for as follows: the bottleneck operation on QuickMatch is the scanning of arcs emanating from permanent nodes, adjusting the temporary node labels and inserting the updated node labels into a heap. On average the depth of the heap was less than 5, and the number of arcs scanned was approximately $5m$. Taking into account that the time spent inserting an arc into a heap is roughly proportional to the depth of the heap, a difference of 30 in the running time of the Arc-Pricing subroutine to QuickMatch seems quite plausible.

3.4 Run time compared with other codes

We compared our QuickMatch code against the Semi code by Kennington and Wang, and the Auction code by Bertsekas and Castenon using randomly generated sparse networks. The result shows that QuickMatch is significantly faster than Semi but is slower than Auction for networks of 20,000 pairs of nodes.

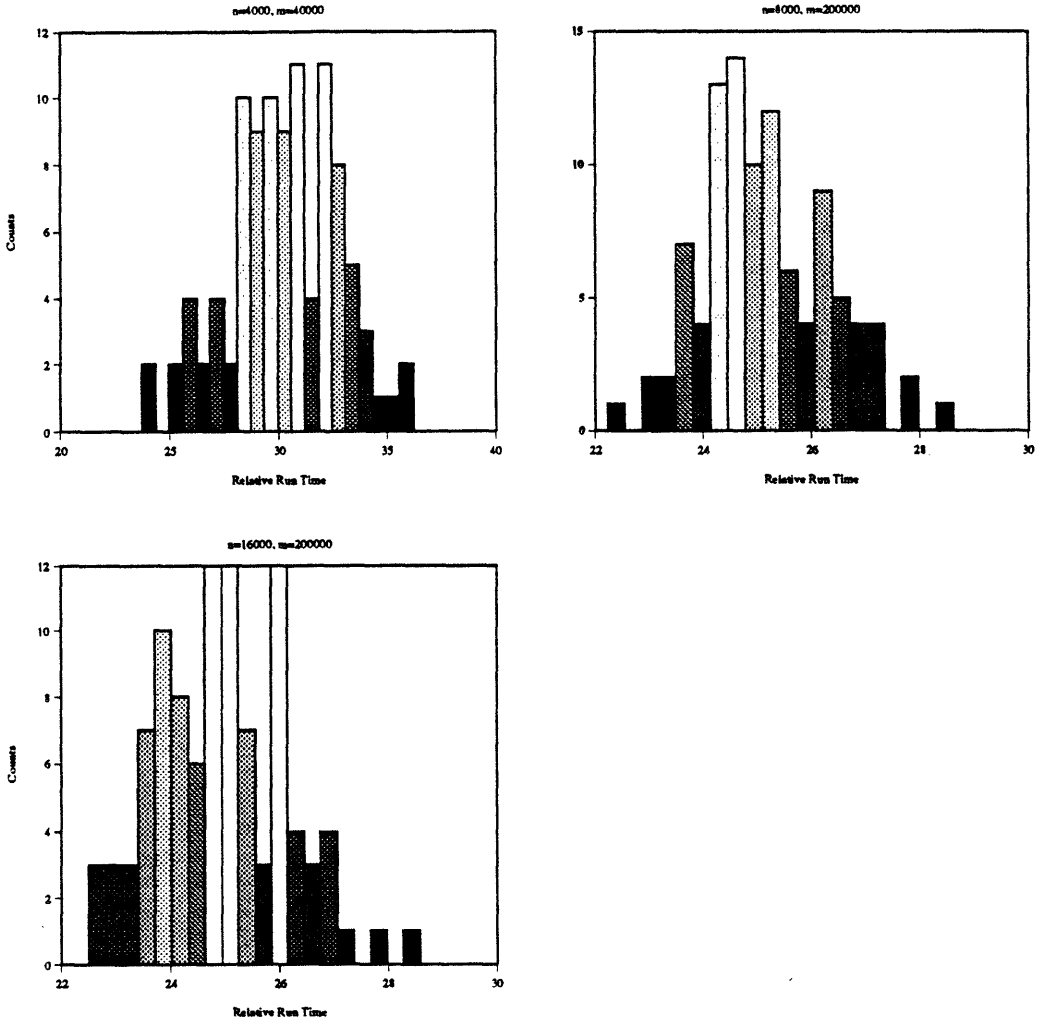


Figure 6: Relative Running Time of the QuickMatch Algorithm. The relative run time refers to the CPU time of the algorithm divided by the CPU time of the arc pricing subroutine.

We predicted in theorem 3 that the run time of the QuickMatch algorithm should be linear in m, the number of arcs. If this is the case, a linear model will be a good try for a regression. The simple least squares regression result is shown below:

$$\begin{array}{l} \text{QuickMatch time} = \underset{(-6.78)}{-1.25} + \underset{(122)}{1.34 \times 10^{-4} m} \qquad R^2 = 0.99 \\ \\ \text{Auction time} = \underset{(-13.59)}{-1.33} + \underset{(118)}{6.89 \times 10^{-5} m} \qquad R^2 = 0.99 \end{array}$$

The regression results shows that both QuickMatch and Auction runs in almost linear time, with a different constant. The difference in the constants is approximately 2. We are not sure if this difference is a result of improved implementation details and/or compiling or whether it is intrinsic to the algorithm. The run time of the three codes are shown in figure 7. Figure 7a shows the run time of all three codes plotted against n, the number of nodes; figure 7b shows the same data for QuickMatch and Auction on a different scale. Figure 7c shows the QuickMatch and Auction run times plotted against m as well as the fitted lines. The slightly upward trend in QuickMatch data is probably due to the phenomenon observed in secton 3.2.

References

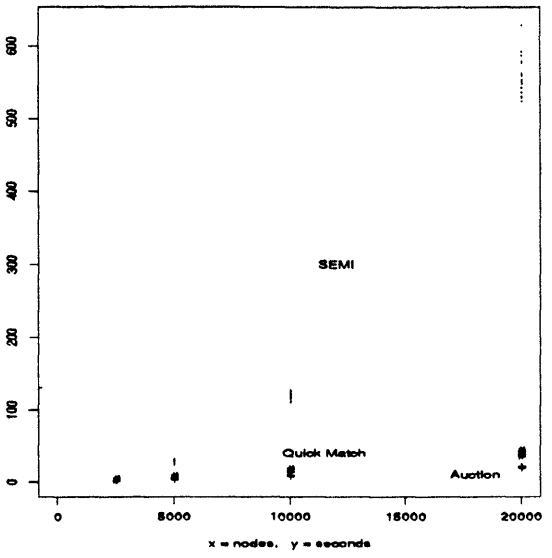
[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: Theory, Algorithms and Applications.* Prentice Hall, 1993.

[2] Mustafa Akgul. A forest primal-dual algorithm for the assignment problem. *Bilkent University, Ankara, Turkey, Research Report: IEOR-9014*, 0(0):1–2, Oct 1990.

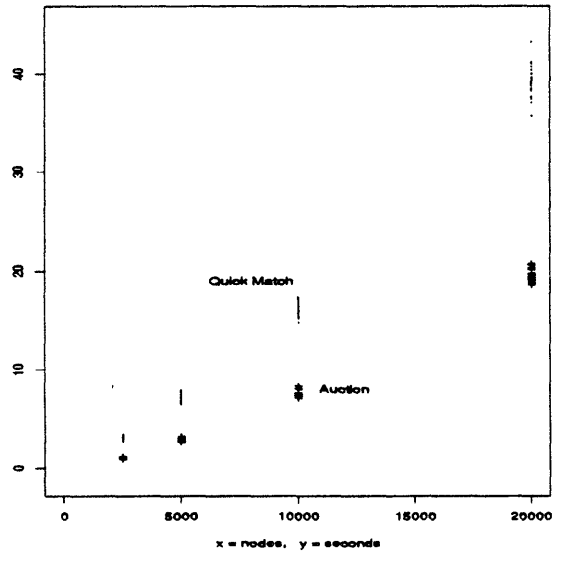
[3] M. L. Balinski. Signature methods for the assignment problem. *Operations Research*, 33(3):527–536, May–Jun 1985.

[4] Dimitri P. Bertsekas. A distributed asynchronous relaxation algorithm for the assignment problem. *Proceedings 24th IEEE Conference on Decision and Control*, 3:1703–1704, 1985.

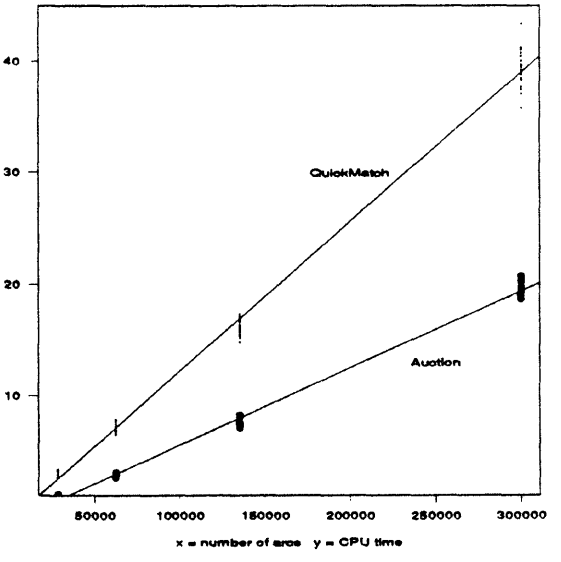
[5] Dimitri P. Bertsekas. The auction algorithm for assignment and other network flow problems: A tutorial. *Interfaces*, 20(4):133–149, Jul–Aug 1990.



(a) Run time of QuickMatch, Semi, and Auction codes.



(b) Run time of QuickMatch and Auction, plotted in a different scale.



(c) Run time of QuickMatch and Auction, plotted against m.

Figure 7: Run time of QuickMatch, SEMI, and Auction.

[6] Harold N. Gabow and Robert E. Tarjan. Faster scaling algorithms for network problems. *SIAM Journal on Computing*, 18(5):1013–1036, Oct 1989.

[7] Ming S. Hung. A polynomial simplex method for the assignment problem. *Operations Research*, 31(3):595–600, May-Jun 1983.

[8] Ming S. Hung and Walter O. Rom. Solving the assignment problem by relaxation. *Operations Research*, 28(4):969–982, Jul–Aug 1980.

[9] Richard M. Karp. An algorithm to solve the mxn assignment problem in expected time o(mn log n). *Networks*, 10:143–152, 1980.

[10] J. Kennington and Z. Wang. A shortest augmenting path algorithm for the semi-assignment problem. *Operations Research*, 40(1):178–187, Jan–Feb 1992.

[11] Vahid Lotfi. A labeling algorithm to solve the assignment problem. *Computers and Operations Research*, 16(5):397–408, 1989.

[12] G. M. Megson and D. J. Evans. A systolic array solution for the assignment problem. *The computer journal*, 33(6):562–569, 1990.