ALMOST-OPTIMUM PARALLEL SPEED-UPS OF ALGORITHMS FOR BIPARTITE MATCHING AND RELATED PROBLEMS

Harold N. Gabow
Robert E. Tarjan

CS-TR-223-89

January 1989

Almost-Optimum Parallel Speed-ups of Algorithms for Bipartite Matching and Related Problems*

Harold N. Gabow[1]
Department of Computer Science
University of Colorado
Boulder, CO 80309

Robert E. Tarjan[2]
Computer Science Department
Princeton University
Princeton, NJ 08544
and
AT&T Bell Laboratories
Murray Hill, NJ 07974

January 18, 1989

Abstract.

This paper focuses on algorithms for matching problems that run on an EREW PRAM with p processors. Given is a bipartite graph with n vertices, m edges, and integral edge costs at most N in magnitude. An algorithm is presented for the assignment problem (minimum cost perfect bipartite matching) that runs in $O(\sqrt{n}m\log(nN)(\log p)/p)$ time and $O(m)$ space, for $p \le m/(\sqrt{n}\log^2 n)$. This bound is within a factor of *log p* of optimum speed-up of the best known sequential algorithm, which in turn is within a factor of *log* (nN) of the best known bound for the problem without costs (maximum cardinality matching). Extensions of the algorithm are given, including an algorithm for maximum cardinality bipartite matching with slightly better processor bounds, and similar results for bipartite degree-constrained subgraph problems (with and without costs).

---

* This is an expanded version of a paper appearing in the *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, 1988.

[1] Research supported in part by NSF Grant No. DCR-851191 and AT&T Bell Laboratories.

[2] Research supported in part by NSF Grant No. DCR-8605962 and ONR Contract No. N00014-87-K-0467.

1. Introduction.

Problems such as cardinality matching, degree-constrained subgraphs and network flow have efficient sequential algorithms [L,T] but seem difficult to parallelize, in the sense of NC parallelism (e.g., [GSS]). This paper investigates parallel algorithms from the more practical viewpoint of speeding up the best known sequential time bounds. It achieves running times that are within a logarithmic factor of optimum speed-up, using significant numbers of processors.

To state the results we first give some definitions and notation; more definitions are at the end of this section. A *matching* on a graph is a set of vertex-disjoint edges. A *free* vertex is not on any matched edge. A *maximum cardinality matching* has the greatest number of edges possible; a *perfect matching* has no free vertices. When every edge has a numerical cost, the *cost* of a set of edges is the sum of its edge costs. A *minimum perfect matching* is a perfect matching of smallest possible cost. The *assignment problem* is to find a minimum perfect matching on a bipartite graph. This problem has many practical applications [Dan].

A *PRAM* (*Parallel Random Access Machine*) consists of p synchronized processors accessing a common memory. On an *EREW* (*Exclusive Read Exclusive Write*) *PRAM*, at most one processor can access a given memory cell in any instruction cycle. If a sequential algorithm runs in time t, an *optimum speed-up* of this algorithm on a PRAM with p processors runs in time $O(t/p)$.

We state resource bounds in terms of the following parameters: The given graph has n vertices, m edges, and integral edge costs at most N in magnitude. The model of computation is an EREW PRAM with p processors.

Let us first review the best known sequential algorithms for bipartite matching problems. For maximum cardinality matching the algorithm of Hopcroft and Karp runs in time $O(\sqrt{n}m)$ [HK]. For the assignment problem the best known strongly polynomial time bound is $O(n(m+n\log n))$, achieved by the Hungarian algorithm implemented with Fibonacci heaps [FT]. When the costs are integers of magnitude at most N and N is not huge, this can be improved: [GaT87] gives a cost scaling algorithm that runs in time $O(\sqrt{n}m\log(nN))$. This bound is within a logarithmic factor of Hopcroft and Karp's bound for the simpler problem of cardinality matching.

Our main result extends this bound to parallel computation:

Theorem 1.1. For integral costs of magnitude at most N, the assignment problem can be solved in time $O(\sqrt{n}m\log(nN)(\log p)/p)$ and space $O(m)$, for $p \le m/(\sqrt{n}\log^2 n)$.

For $p = 1$ this bound equals that of [GaT87], although the latter algorithm is simpler. For

$1 < p \leq m/(\sqrt{n} \log^2 n)$ our algorithm speeds up the sequential algorithm by the almost optimum factor of $p/\log p$. This gives a non-trivial speed-up even for very sparse graphs, e.g., graphs with $m = O(n)$. Using the greatest number of processors allowed, Theorem 1.1 gives a running time of $O(n \log^3 n \log(nN))$.

Now let us review previous parallel algorithms for matching and related problems. [KUW], [GP] and [MVV] give parallel algorithms for minimum cost matching that work even on general graphs. These algorithms are probabilistic, however, and use large numbers of processors. (The time is $O(\log^2 n)$ with $nmNM(n)$ processors for [MVV]; slightly fewer processors and slightly more time for [KUW] and [GP]. Here $M(n) \geq n^2$ is the sequential time needed to multiply two $n \times n$ matrices. The space for these algorithms is also large since it equals the processor count). Our work is related to the algorithm of Goldberg and Tarjan for the more general minimum cost flow problem [GoT]. A parallel version of their algorithm runs in $O(n^2(\log n)(\log nN))$ time using n processors and $O(n^2)$ space.

Our assignment algorithm can be used to solve the single-source shortest path problem on a directed graph with arbitrary edge lengths, in the bounds of Theorem 1.1. The assignment algorithm generalizes to the minimum cost degree-constrained subgraph problem. The result is similar to Theorem 1.1: all n's in the bounds of Theorem 1.1 change to U, the sum of the degree constraints. These results achieve close to optimum speed-up of sequential algorithms in [GaT87] for the same problems.

The basic problem addressed by our work is finding paths fast in parallel. Efficient transitive closure algorithms use too many processors for optimum speed-up. Our solution involves extending the notion of ϵ-optimality of [GoT] (and the equivalent notion of 1-optimality of [GaT87]) to r-optimality, a form more suited to parallel computation. We also we use the idea of a reduced graph and a path doubling technique to ensure that all paths explored are short on the average.

Our approach to the assignment problem simplifies when applied to the problem of finding a maximum cardinality matching on a bipartite graph, giving slightly better processor bounds:

Theorem 1.2. A maximum cardinality matching on a bipartite graph can be found in time $O((\sqrt{n}m \log p)/p)$ and space $O(m)$, for $p \leq m/(\sqrt{n} \log n)$.

This bound is within a factor of *log p* of an optimum speed-up of the best sequential algorithm [HK]. Using the greatest number of processors allowed, the running time is $O(n \log^2 n)$.

Shiloach and Vishkin [SV] give a parallel algorithm for cardinality matching. They achieve almost optimum speed-up but for fewer processors, $p \leq m/n$ (fastest running time $O(n^{1.5} \log n)$).

The algorithm of [KC] has a faster running time of $O(n \log n \log\log n)$, a factor of $\log n/\log\log n$ better than our fastest time. But transitive closure is used, whence the number of processors (and the space) is $M(n)$. The above-mentioned algorithms of [KUW], [GP] and [MVV] are more efficient for cardinality matching but still not close to optimum in their use of processors ($p = nM(n)$ for [KUW, GP], $p = nmM(n)$ for [MVV]).

Our cardinality matching algorithm generalizes to the maximum cardinality degree-constrained subgraph problem. For instance on a bipartite graph the time is $O((n^{2/3} m \log p)/p)$ for $p \le m/n^{5/6}$. This improves Shiloach and Vishkin's bound of $O((nm \log n)/p)$ time for $p \le m/n$ [SV].

The rest of this paper is organized as follows. Section 2 presents our algorithm for maximum cardinality bipartite matching, and along with some extensions. This illustrates our approach in a simple setting. Section 3 is devoted to the assignment problem. Section 4 discusses the extensions to the shortest path problem and the minimum cost degree-constrained subgraph problem. This section closes with definitions from graph theory.

The logarithm function *log n* is to the base two; *log^{i}n* denotes the i^{th} power of *log n*. It is convenient to take *log* 1 to be one. We use the following convention to sum the values of a function: If f is a real-valued function defined on elements and S is a set of elements, then $f(S) = \sum\{f(s) | s \in S\}$.

The given graph has vertex set V and edge set E. In general for a graph G, $V(G)$ and $E(G)$ denote its vertex set and edge set, respectively. We use the notation $v \in G$ ($vw \in G$) as a shorthand for $v \in V(G)$ ($vw \in E(G)$) when no confusion can arise. If H is a subgraph, an *H-edge* is an edge in H and a *non-H-edge* is not in H. If the given graph is bipartite we denote the bipartition as V_0, V_1. Hence any edge joins V_0 to V_1; if e is an edge, e_0 denotes its vertex in V_0 and similarly for e_1. For a subgraph H, $V_0(H)$ and $V_1(H)$ have the obvious meaning. In problems with edge costs, $c(e)$ denotes the *cost* of edge e. By our convention for functional notation, $c(S)$ denotes the total cost of a set of edges S.

An *st*-path is a path from vertex s to vertex t; two *st*-paths are *vertex disjoint* if their only common vertices are s and t. A directed graph is *layered* if $V(G)$ can be partitioned into sets W_i, $i = 0, \ldots, k$, such that any edge goes from some W_i to W_{i+1}.

In a graph with a matching M, an *alternating path (cycle)* is a simple path (cycle) whose edges are alternately matched and unmatched. An *augmenting path P* is an alternating path joining two distinct free vertices. *Augmenting along P* means enlarging the matching to $M \oplus P$, a matching with one more edge.

If M is a matching on a bipartite graph G, the *residual graph for M* (a term from network flow theory [T]) is a directed graph D that models the augmenting paths. The vertices of D are those in $V(G)$ plus two new vertices s and t; the edges of D are the unmatched edges of G, directed from V_0 to V_1; plus the matched edges of G, directed oppositely; plus edges from s to each free vertex of V_0; plus edges from each free vertex of V_1 to t. Augmenting paths for M correspond one-to-one with st-paths in D.

Consider a multigraph in which each vertex v has associated nonnegative integers $\ell(v)$ and $u(v)$. Let D be a subgraph, and let $d(v)$ denote the degree of vertex v in D (each copy of an edge incident to v contributes one to $d(v)$). A ***degree-constrained subgraph*** (***DCS***) is a subgraph in which each v has $\ell(v) \le d(v) \le u(v)$. In a *perfect DCS* each v has $d(v) = u(v)$. The number of edges in a perfect DCS is denoted by $U = u(V_0)$. A vertex is *free* in D if $d(v) < u(v)$. Other definitions for DCS— e.g., minimum perfect DCS, the residual graph, etc., follow by analogy with matching.

2. Maximum cardinality matching.

This section introduces our approach in the simple setting of cardinality problems. It discusses the problem of maximum cardinality bipartite matching, proving Theorem 1.2. This illustrates two main ingredients of our algorithms: efficient breadth-first search techniques and the reduced graph. The section begins by stating the cardinality matching algorithm of Hopcroft and Karp [HK]. Then it gives an efficient parallel implementation. The section ends with extensions of our algorithm to the maximum cardinality degree-constrained subgraph problem.

Let G be a bipartite graph. Consider an arbitrary matching M on G, with residual graph D (defined in Section 1). The *level* $\ell(v)$ of a vertex v is the length of a shortest sv-path in D ($\ell(v)$ is infinite if there is no sv-path). The *level graph* (*for* M) consists of all directed edges vw that have $\ell(w) = \ell(v) + 1$ (with both levels finite). The following algorithm of Hopcroft and Karp finds a maximum cardinality matching on G.

procedure *match.*
Initialize the matching M to $\emptyset$. Then repeat the following steps until the Search Step halts with the desired matching.

Search Step. Construct the level graph for M by doing a breadth-first search on the residual graph D, starting from vertex s. If $\ell(t)$ is infinite in D, halt with the desired matching M.

Augment Step. Find a maximal set $\mathcal{A}$ of vertex-disjoint st-paths in the level graph. Then for each path $P \in \mathcal{A}$, augment the matching along P. ▮

Two quantities are used to analyze the various matching algorithms in this paper:

$$I = \text{the number of iterations of the loop of } \textit{match};$$

$$A = \text{the total length of all augmenting paths found by } \textit{match}.$$

For the Hopcroft-Karp algorithm $I = O(\sqrt{n})$ [HK] and $A = O(n \log n)$ [ET]. Now we show that, on an EREW PRAM with p processors ($1 \leq p \leq m$), *match* can be implemented in time

$$O((Im/p + A)\log p). \tag{2.1}$$

We start with the data structure for graphs that is used throughout this paper. The given graph G is represented by sequentially-stored adjacency lists. More precisely, let $I(v)$ denote the set of edges incident to a vertex v. For each vertex v, the edges of $I(v)$ are stored in consecutive locations; v has pointers to the first and last edges of $I(v)$, and all lists $I(v)$ are stored in $O(m)$ consecutive locations. The two copies of any edge are linked to each other. Both copies of an edge have $O(1)$ fields for working storage (determined by the algorithm). In particular these fields contain the links for all lists of edges used by the algorithm. This data structure can be constructed by one processor in time $O(m + n)$, given any reasonable input representation of G; this suffices for our purposes.

This data structure facilitates scanning a set of edges in parallel. For example, the next p edges incident to a vertex v can be scanned concurrently, processor i scanning the edge i locations after the current position in $I(v)$. An algorithm for breadth-first search can be based on this principle. It finds the first L levels of a breadth-first search starting from any given set of vertices in time

$$O((m/p + L)\log p). \tag{2.2}$$

This fact may be viewed as an application of Brent's principle [B], since it is obvious that each level of a breadth-first search can be done in $O(1)$ time, assuming m processors and ignoring processor allocation problems. We will assume this breadth-first search routine for now. We sketch an implementation below.

The Search Step is implemented with the breadth-first search routine. Each search stops when vertex t is reached, or when there are no more vertices to scan. (The latter is true in the last iteration, since $\ell(t)$ is infinite.)

The Search Steps use total time (2.1). To see this note that there are I Search Steps, so the first terms of (2.2) for all breadth-first searches sum to $O((Im/p)\log p)$. In all searches except the last, the total number of levels L searched is less than A, by the stopping criterion of the Search Step. The last search explores $O(A)$ levels, since the search paths are alternating. So the second terms of (2.2) sum to $O(A\log p)$. This gives the desired time bound.

The Augment Step is implemented to avoid backtracking, using the following idea. Consider a directed acyclic graph D with distinguished vertices s and t. The *reduction* R of D is the maximal subgraph of D such that every vertex except t has positive outdegree (in R). Clearly any st-path of D is in R. Further, an st-path in R can be found by a greedy strategy: start at s and repeatedly traverse an edge xy directed from the most recently reached vertex x. The Augment Step uses the reduction graph, as follows.

For a subgraph H, the notation $V_{\_}(H)$ stands for $V(H)-\{s,t\}$.

The Augment Step finds st-paths one-by-one, adding successive paths to $\mathcal{A}$. Let L be the level graph. The Augment Step maintains the reduction R of the graph $L - V_{\_}(\mathcal{A})$. (It does this by marking the vertices that are in R.) It finds the next st-path P using the above greedy strategy: Starting with the above most recently reached vertex x, to find the next edge xy the processors repeatedly examine the next p edges directed from x, to find an edge directed to a vertex y of R. Using this approach the time to find all st-paths in the entire *match* algorithm is given by (2.1) (the term $O((Im/p)\log p)$ accounts for the time examining a group of p edges that do not lead to a vertex in R).

After an st-path P is found, the vertices $V_{\_}(P)$ are deleted from R. Then R is updated so that each vertex has positive outdegree. This is essentially a breadth-first search backwards from the level of t to the level of s. The search uses Cole's algorithm to sort p numbers in time $O(\log p)$ (to keep track of outdegrees when edges are deleted) [C]. A breadth-first search that deletes μ edges uses time $O((\mu/p+L)\log p)$. Since the preceding st-path has length L, the time for updating R over the entire algorithm is given by (2.1).

We finish the discussion of Theorem 1.2 by sketching the parallel breadth-first search routine. (This routine and its data structures are used throughout the paper.) First observe that it is easy to do a breadth-first search of G, from a given set of vertices S, in time $O(n\log p + m/p)$. The idea is that a parallel prefix computation broadcasts the next vertex v to scan; the processors scan the edges incident to v in parallel, each processor building up a "vertex list" of vertices on the next level.

A slightly more involved procedure achieves time (2.2). The search works in L stages (for L

the desired number of levels). For $\ell = 0, \ldots, L-1$, the ℓ^{th} stage scans all vertices on level ℓ and places all newly reached vertices on level $\ell + 1$. Define

$$m_\ell = \sum \{|I(v)| \mid \text{vertex } v \text{ is on level } \ell\}.$$

Stage ℓ uses time $O((1 + (m_\ell + m_{\ell+1})/p) \log p)$ time. (Clearly this gives bound (2.2)).

There are two data structures: each processor i has two linked lists, a *scan list* $\mathcal{S}(i)$ and a *vertex list* $\mathcal{V}(i)$. The scan lists contain the edges to be scanned in the ℓ^{th} stage and the vertex lists contain the vertices on level $\ell + 1$. More precisely, the scan lists partition the edges incident to vertices on level ℓ. Each scan list specifies at most $\lceil m_\ell / p \rceil$ such edges. An entry on a scan list is a triple (v, j, k), which corresponds to the j^{th} through k^{th} edges (inclusive) in the incident list $I(v)$. A vertex list is a list of at most $\lceil m_\ell / p \rceil$ vertices on level $\ell + 1$.

Stage ℓ works in two parts. The first part scans the edges incident to level ℓ. Processor i scans the edges on $\mathcal{S}(i)$. It adds newly reached vertices w to its vertex list $\mathcal{V}(i)$. Sorting and parallel prefix computations are used to coordinate the manipulations of vertices w. Cole's algorithm is used to sort p numbers in time $O(\log p)$ [C].

The second part of stage ℓ uses the vertex lists to construct scan lists for stage $\ell + 1$. This is done in two steps. Step 1 is a parallel prefix computation that calculates several quantities including $m_{\ell+1}$. In Step 2, each processor i constructs one or more triples for each vertex $v \in \mathcal{V}(i)$; the triples specify how $I(v)$ will be partitioned among scan lists. The construction uses the fact that scan list boundaries occur every $\lceil m_{\ell+1}/p \rceil$ edges. Since any processor examines at most $\lceil m_\ell / p \rceil$ vertices and $\lceil m_{\ell+1}/p \rceil$ boundaries, the time is as desired.

This completes the proof of Theorem 1.2. ∎

Now consider the problem of finding a maximum cardinality degree-constrained subgraph. Our implementation of the Hopcroft-Karp algorithm easily generalizes to this problem. (We omit the details here. Section 4.2 addresses the main issues, in the context of the weighted case.)

Corollary 2.1. Consider a bipartite multigraph, where all edge multiplicities are at most M ($M = 1$ for a graph). A maximum cardinality degree-constrained subgraph can be found in time $O(\min\{\sqrt{U}, n^{2/3} M^{1/3}\} m/p + \min\{U \log U, n\sqrt{MU}\}) \log p)$ and space $O(m)$.

Proof. The time is expression (2.1), using the bounds on I and A given in [ET,FM, GaT87]. ∎

3. The assignment problem.

This section presents our parallel algorithm for the problem of finding a minimum perfect matching in a bipartite graph, proving Theorem 1.1. For convenience we assume the given graph G has a perfect matching. (The algorithms of Section 4.2 handle other versions of the weighted matching problem.) Also for convenience, in this section n denotes $|V_0|$, which is half the number of vertices. We present the algorithm in a top-down fashion. Section 3.1 introduces r-optimal matchings and gives the basic routines of the algorithm. The two major subroutines are in Sections 3.2 and 3.3.

3.1. The basic algorithm: r-optimality.

Most algorithms for weighted matching, including ours, use the linear programming dual variables [Dan]. A *dual function* is a function $y : V \rightarrow \mathbf{Z}$ (for $\mathbf{Z}$ the set of integers); $y(v)$ is called the *dual variable* of vertex v. Our notational convention for functions (see Section 1) implies the following notation: For an edge e, $y(e) = y(e_0) + y(e_1)$ (since precisely speaking, $e = \{e_0, e_1\}$). Similarly if S is a set of edges, $y(S) = \sum\{y(e) | e \in S\}$. Observe that if M is a matching, $y(M) = \sum\{y(v)|$ vertex v is matched in $M\}$.

The Hungarian algorithm and other traditional approaches to weighted matching are based on the complementary slackness condition for minimum perfect matching [L]: A perfect matching M has minimum cost if and only if there is a dual function such that for any edge e, $y(e) \leq c(e)$, with equality holding for any $e \in M$. We call such a dual function an (*optimum*) *linear programming dual.*

Our approach uses a modification of linear programming duals. An *r-feasible matching* consists of a matching M, nonnegative integer r, and dual function y such that

$$y(e) \leq c(e) + (\textbf{if } e \in M \textbf{ then } 0 \textbf{ else } 1), \qquad e \in E; \tag{3.1a}$$

$$c(M) \leq y(M) + r. \tag{3.1b}$$

An *r-optimal* (*relaxed-optimal*) *matching* is a perfect matching that is r-feasible.

To motivate this definition, first observe that dropping the **if** term from (3.1a) and setting $r = 0$ gives the linear programming duals. Now put back the **if** term but keep $r = 0$. This notion is *1-optimality*, used in [GaT87] to design an efficient sequential algorithm for minimum perfect matching. The intuition is that the **if** term makes the cost of augmenting paths reflect their length (each unmatched edge contributes 1 to the path length, and an extra 1 to the path cost; in the context of scaling the extra 1 is significant, since costs are small). Because of this the algorithm tends to augment along paths of short length, as in the Hopcroft-Karp cardinality matching algorithm. 1-optimality is similar to the notion of ϵ-optimal flows in the minimum cost flow algorithm of [GoT]. The parallel algorithm presented here achieves the same asymptotic time as [GaT87] when it runs on one processor.

The notion of 1-optimality does not seem to lead to an efficient parallel algorithm. 1-optimality guarantees a low bound on the total length of all augmenting paths, but some augmenting paths can still be long. This implies that an algorithm must explore long candidate augmenting paths, which seems hard to do efficiently in parallel. We use r-optimality to overcome this difficulty: r-optimality guarantees a low bound on the total length of all augmenting paths (Lemma 3.5) and also gives the algorithm the flexibility to invalidate long candidate augmenting paths.

We now develop the properties of r-optimality, and at the same time state the basic algorithm. We start with the relation between r-optimal matchings and minimum perfect matchings; similar results are in [GoT], [GaT87].

Lemma 3.1. If some integer larger than $r+n$ divides each cost $c(e)$ evenly, then any r-optimal matching is a minimum perfect matching.

Proof. Consider a perfect matching P. It suffices to show that $c(M) \le c(P) + r + n$, since $c(M)$ and $c(P)$ are both multiples of the integer hypothesized in the lemma. From (3.1b), $c(M) \le y(V) + r$; from (3.1a), $y(V) \le c(P) + n$. Combining these gives the desired inequality. ∎

The algorithm is stated using three integer parameters,

$$r, \qquad b = 3r + 5n, \qquad g.$$

The value of these parameters is chosen in Section 3.2 (specifically we choose $r, b = \Theta(n)$, $g = \Theta(\log n)$; r is the parameter for r-optimality).

The *main routine* of the algorithm scales the costs. It first computes a new cost $\overline{c}(e)$ for each edge e, equal to $r+n+1$ times the given cost. Consider each $\overline{c}(e)$ to be a signed binary number $\pm b_1 b_2 \ldots b_k$ of $k = \lfloor \log(r+n+1)N \rfloor + 1$ bits. The routine maintains a variable $c(e)$ for each edge e, equal to its cost in the current scale. The routine initializes each $c(e)$ to 0 and each $y(v)$ to 0. Then it executes the following loop for index s going from 1 to k:

Double Step. For each edge e, $c(e) \leftarrow 2c(e) +$ (signed bit b_s of $\bar{c}(e)$). For each vertex v, $y(v) \leftarrow 2y(v) - 1$.

Match Step. Call the *scale_match routine* to find an r-optimal matching for costs $c(e)$. ▮

Lemma 3.1 shows that the *main routine* halts with a minimum perfect matching. Each iteration of the loop is called a *scale*. Clearly the total time is $O(\log((r+n)N))$ times the time for one scale. Note that the entire algorithm runs in the desired time bound if each scale runs in time

$$O((\sqrt{n}m/p + n\log^2 n)\log p). \tag{3.2}$$

This follows since as noted above we will choose $r = O(n)$. The time for the Double Step is $O(m/p)$.

The *scale_match routine* transforms costs so that they are small integers. (This is for conceptual convenience.) It changes the cost of each edge e to $c(e) - y(e)$; then it calls the *match* routine on these costs to find an r-optimal matching M with duals y'; then it constructs the new dual function $y + y'$, where y is the dual function before the call to *match*. The time for these transformations is $O(m/p + \log p)$ (a parallel prefix computation is used to broadcast dual values $y(v)$; each processor uses at most two duals that another processor uses).

Clearly when *scale_match* terminates, M with the new duals is an r-optimal matching for cost function c. Furthermore, the costs that *scale_match* inputs to *match* have these properties:

(a) The costs are integers -1 or larger.

(b) There is a perfect matching of cost at most $2r + 3n$.

Property (a) follows from the fact that the Double Step changes costs and duals so that each edge e has $y(e) - 1 \le c(e)$. Next we show that M, the r-optimal matching found in the previous scale, satisfies property (b) ((b) is obvious in the first scale). For any edge $e \in M$, let $\rho(e)$ be the value $c(e) - y(e)$ from the previous scale. After the Double Step, $2\rho(e) + 3 \ge c(e) - y(e)$. Hence e costs at most $2\rho(e) + 3$ in the costs for *match*. The conclusion for M follows.

In the *match* routine, an edge e is ***eligible*** if it is matched or constraint (3.1a) holds with equality. The *match* routine augments the matching along paths of eligible edges. (To motivate this, think of (3.1a) as placing a lower bound on $c(e)$. Then an unmatched eligible edge has smallest cost possible, and so using it in an augmenting path is desirable.) If there is no augmenting path of eligible edges, *match* adjusts the duals to create one. More precisely *match* works as follows.

procedure *match*.

Initialize all duals $y(v)$ to 0 and matching M to $\emptyset$. Then repeat the following steps until the Augment Step halts with the desired r-optimal matching.

Augment Step. **Find a maximal set $\mathcal{A}$ of vertex-disjoint augmenting paths of eligible edges. For each path $P \in \mathcal{A}$, augment the matching along P, and for each vertex $w \in V_1(P)$, decrease $y(w)$ by 1. If the new matching M is perfect, halt.**

Search Step. Do a Relaxed Hungarian Search (see below) to adjust the duals, maintaining r-feasibility, and create an augmenting path of eligible edges. ▌

To analyze *match*, we must first give some details of the Search and Augment Steps (the Search Step is described completely in Section 3.2; the Augment Step is in Section 3.3). The Relaxed Hungarian Search is a modification of the Hungarian search done in bipartite matching (the latter is essentially Dijkstra's shortest path algorithm [L,T]). The Relaxed Hungarian Search changes dual values in two ways: *dual adjustments*, which are also done in the ordinary Hungarian search, and *relax operations*, which are new. Each dual adjustment calculates a positive integer δ and increases or decreases various dual values by δ, so as to preserve r-feasibility and eventually create an augmenting path of eligible edges. A relax operation does not create any eligible edges.

At any point in *match* define

$f =$ the number of free vertices in V_0;

$\Delta =$ the sum of all dual adjustment quantities δ in all Hungarian searches so far.

(Δ is defined with respect to the current execution of *match*.) The duals are maintained so that any free vertex v has

$$y(v) = \textbf{if } v \in V_0 \textbf{ then } \Delta \textbf{ else } 0. \tag{3.3}$$

Now we analyze *match*. First observe that it is correct, specifically: (i) it maintains r-feasibility, and (ii) it halts with M an r-optimal matching. Property (i) holds after the initialization (by property (a) of the costs for *match*). It is part of the specification of the Relaxed Hungarian Search. Hence we need only consider an Augment Step. It decreases duals so that $y(e) = c(e)$ for every newly matched edge e. This implies that $(3.1a)$ holds. It also implies $(3.1b)$ (since every previously matched edge satisfied $y(e) \leq c(e)$). Now consider property (ii). If M is not perfect but G has a perfect matching, the Search Step creates an augmenting path of eligible edges. Hence (ii) eventually holds. (If G does not have a perfect matching, this is eventually detected in the Search Step.)

The efficiency analysis starts with a fact similar to the key result in the analysis of the Hopcroft-Karp algorithm.

Lemma 3.2. At any time during *match*, $f\Delta \le b$.

Proof. At any point in *match* let M be the current matching, and let M^* be a minimum perfect matching. Consider the expression

$$Y = y(M^*) - y(M).$$

$M^* \oplus M$ consists of alternating cycles plus exactly f augmenting paths. Hence $Y = y(\{v | v \text{ is free in } M\}) = f\Delta$, by (3.3). On the other hand (3.1) implies $Y \le c(M^*) + n - c(M) + r$. Properties (a)–(b) of the costs for *match* imply that the last expression is at most $3r + 5n = b$. ▮

Define the quantities I and A as in Section 2. In the definition of A, measure the length of an alternating path by the number of unmatched edges. For $1 \le i \le n$ define

$$\Delta_i = \text{ the value of } \Delta \text{ during the } i^{th} \text{ augmentation.}$$

(The i^{th} augmentation is when *match* augments along the i^{th} augmenting path.) These quantities are bounded as follows.

Lemma 3.3. $I < 2\sqrt{b} + 1$.

Proof. First we show that a Hungarian search $\mathcal{S}$ increases Δ by at least one. It suffices to show that $\mathcal{S}$ does a dual adjustment (since any dual adjustment quantity δ is a positive integer). Search $\mathcal{S}$ does a dual adjustment unless, when it starts, there is an augmenting path P of eligible edges. Clearly P intersects some augmenting path of $\mathcal{A}$ of the preceding Augment Step. It is easy to see that P contains an unmatched edge e, such that e_1 but not e_0 is in an augmenting path of $\mathcal{A}$. But e is ineligible after the Augment Step decreases $y(e_1)$. Thus P does not exist, and $\mathcal{S}$ does a dual adjustment.

This implies that at most $\sqrt{b}$ Search Steps end with $\Delta \le \sqrt{b}$. If a Search Step ends with $\Delta > \sqrt{b}$ then $f < \sqrt{b}$ by Lemma 3.2. There can be at most f more iterations, since each Augment Step enlarges the matching. ▮

Lemma 3.4. $\sum_{i=1}^{n} \Delta_i \le b + b\log n$.

Proof. Since $f = n-i+1$ right before the i^{th} augmentation, Lemma 3.2 implies $\Delta_i \le b/(n-i+1)$. ▮

Lemma 3.5. $A \le n + b + b\log n$.

Proof. Let P_i be the path used in the i^{th} augmentation. Let ℓ_i be its length, measured as the number of unmatched edges; let M_i be the matching after augmenting along P_i. (Note that $M_0 = \emptyset$ and $c(M_0) = 0$.) Using (3.3), the definition of "eligible", and (3.1a),

$$\Delta_i = y(P_i - M_{i-1}) - y(P_i \cap M_{i-1}) \ge \ell_i + c(M_i) - c(M_{i-1}).$$

Summing these inequalities implies $\sum_{i=1}^{n} \Delta_i \ge A - n$ (by property (a), $c(M_n) \ge -n$). Now Lemma 3.4 implies the desired bound. ▮

3.2. Relaxed Hungarian Search.

This section first describes ordinary Hungarian search, modified to accommodate the concepts of our paper — eligible edges and r-feasiblity. This version of Hungarian Search is what is needed in an efficient one-processor algorithm. The remainder of the section describes Relaxed Hungarian Search and presents its analysis.

Ordinary Hungarian search has two main components, the search forest $\mathcal{F}$ and the dual adjustment operation. Recall that the purpose of a Hungarian search is to create an augmenting path of eligible edges, by adjusting the duals in a way that preserves r-feasibility. The augmenting path is found by growing a forest $\mathcal{F}$. The roots of $\mathcal{F}$ are the free vertices of V_0; any path from a vertex to a root in $\mathcal{F}$ is an alternating path of eligible edges. Hence when $\mathcal{F}$ contains a free vertex of V_1 it contains the desired augmenting path.

If a maximal forest $\mathcal{F}$ does not contain an augmenting path, a *dual adjustment* can be done. Define the *dual adjustment quantity*

$$\delta = \min\{c(e) + 1 - y(e) | e_0 \in \mathcal{F},\ e_1 \notin \mathcal{F}\}.$$

Each $v \in \mathcal{F}$ has its dual $y(v)$ increased by $\delta \times$ (**if** $v \in V_0$ **then** 1 **else** -1). This adjustment preserves r-feasibility, since it does not change $y(e)$ when e has both vertices in $\mathcal{F}$. Furthermore, any edge e achieving the above minimum becomes eligible, and can be added to $\mathcal{F}$.

The Hungarian search alternates between growing $\mathcal{F}$ and doing dual adjustments. Specifically, $\mathcal{F}$ is grown until it is maximal: An eligible edge e with $e_0 \in \mathcal{F}$ and $e_1 \notin \mathcal{F}$ is added to $\mathcal{F}$ whenever possible; if e_1 is not free, its matched edge $e_1 e_1'$ is also added to $\mathcal{F}$. If the maximal $\mathcal{F}$ does not contain an augmenting path, a dual adjustment is done. Then the process repeats. Eventually $\mathcal{F}$ contains the desired augmenting path of eligible edges, at which point the ordinary Hungarian search halts.

The ordinary Hungarian search is adequate for $p = 1$ (it is used in the one-processor algorithm of [GaT87]). It is not efficient for our approach to parallel processing, however, for the following reason. As illustrated in Section 2, our approach charges search time to augmenting path length. But ordinary Hungarian search leads to two circumstances where search time can be much longer than augmenting path length: First, a search might grow a forest $\mathcal{F}$ with long paths, yet after dual adjustments, find a short augmenting path. Second, when the search halts there may be long alternating paths of eligible edges that the Augment Step must explore, yet these paths may not lead to any augmentations.

The Relaxed Hungarian Search remedies this using the relax operation. To *relax* a set of matched vertices $S \subseteq V_0$ means to decrease $y(v)$ by 1 for each $v \in S$. The relax operation makes every unmatched edge incident to S ineligible. Concerning r-feasibility, note that a relax operation preserves $(3.1a)$. It decreases $y(M)$ by $|S|$, so $(3.1b)$ places a limit on relax operations.

Relax operations can be used to overcome the above two difficulties, as follows. First the algorithm can limit the time to grow $\mathcal{F}$: If a parallel step adds just a small number of vertices to $\mathcal{F}$, the algorithm relaxes those vertices, preserving $(3.1b)$, yet cutting off the growth of $\mathcal{F}$. Second, after the parallel Hungarian search finds an augmenting path, there may still be eligible edges to add to $\mathcal{F}$. The algorithm continues to add vertices to $\mathcal{F}$ in parallel, until some parallel step adds just a small number of vertices. At that point the algorithm relaxes those vertices, cutting off further growth as desired. We shall see that these two remedies lead to an efficient algorithm.

Before presenting the algorithm in detail note that the following modification of the relax operation might be more efficient in practice: decrease $y(v)$ only if $v \in S$ is incident to an unmatched eligible edge. Our analysis applies without change to this modification. For definiteness, the rest of the paper assumes that the simpler relax operation given above is used.

Now we describe Relaxed Hungarian Search. The search initializes the search forest $\mathcal{F}$ to contain the free vertices of V_0. Then it repeats the following two steps until the Adjust Step halts with $\mathcal{F}$ as desired. (Recall that f denotes the number of free vertices in V_0 and g is a parameter of the algorithm.)

Adjust Step. Set $W_1 \leftarrow \{e_1 |$ some eligible edge e has $e_0 \in \mathcal{F}$, $e_1 \notin \mathcal{F}\}$, $W_0 \leftarrow \{e_0 | e_1 \in W_1,\ e \in M\}$. If $W_1 = \emptyset$ and $\mathcal{F}$ contains a free vertex of V_1, halt. If $W_1 = \emptyset$ and $\mathcal{F}$ does not contain a free vertex of V_1, do a dual adjustment and repeat this step.

Grow Step. For each vertex $w \in W_1$, add an eligible edge vw ($v \in \mathcal{F}$) to $\mathcal{F}$, and if w is not free, add the matched edge ww' to $\mathcal{F}$. If $|W_0| < f/g$ then relax W_0. ▮

Let us clarify the flow of control in this algorithm. First consider the Adjust Step. A dual adjustment in this step is well-defined, since it is done only when $\mathcal{F}$ does not contain a free vertex of V_1. A dual adjustment ensures that the next Adjust Step has $W_1 \neq \emptyset$; hence the Adjust Step repeats at most once.

Next consider the Grow Step. After it adds edges, the eligible edges L joining a vertex of $V_0(\mathcal{F})$ to a vertex not in $\mathcal{F}$ are all incident to W_0. If $|W_0| \geq f/g$, the next Adjust Step and Grow Step process these edges L (if $L \neq \emptyset$). If $|W_0| < f/g$ the relax operation makes the edges L ineligible; hence the next Adjust Step either halts or does a dual adjustment.

We define two more quantities for the analysis:

$$R = \text{the total decrease in duals caused by relax operations;}$$

$$H = \text{the total number of iterations in all Hungarian searches.}$$

Here an iteration is defined as an execution of an Adjust Step plus the following Grow Step (if it exists). Both quantities are defined with respect to the current execution of *match*. We shall choose the parameter r to be an upper bound on R.

The correctness of the Relaxed Hungarian Search amounts to these properties: (i) it preserves (3.3); (ii) it preserves r-feasibility; (iii) it eventually halts having created an augmenting path of eligible edges.

For (i), the dual of a free vertex v changes only in a dual adjustment. If $v \in V_0$ then every dual adjustment increases $y(v)$, so $y(v) = \Delta$. If $v \in V_1$ then no dual adjustment changes $y(v)$, so $y(v) = 0$.

Property (ii) was essentially verified in the above discussion. For (3.1b), we have observed that a dual adjustment does not change $y(M)$; an Augment Step does not increase $c(M) - y(M)$. Relax operations decrease $y(M)$. Hence our choice of r will guarantee r-feasibility.

For (*iii*), as already noted an Adjust Step repeats at most once. Then a Grow Step with $W_1 \neq \emptyset$ is executed. Hence every iteration of Adjust and Grow enlarges $\mathcal{F}$. Thus the routine eventually halts. When it halts, $\mathcal{F}$ contains a free vertex of V_1. Hence $\mathcal{F}$ contains an augmenting path of eligible edges. (Note that relax operations do not destroy the eligibility of edges in $\mathcal{F}$.)

We establish two other properties that are needed by the Augment Step (Section 3.3). The first is that $\mathcal{F}$ contains all vertices that are on an augmenting path of eligible edges. This follows since the search halts with $W_1 = \emptyset$.

For the second property, first recall that the Augment Step of the cardinality matching algorithm relies on the fact that the level graph is layered. In minimum cost matching the graph of eligible edges is not layered. This makes the Augment Step more difficult. The eligible edges have the following weaker property (similar to [GoT] for network flow).

Lemma 3.6. In *match* there is never an alternating cycle of eligible edges.

Proof. We proceed by contradiction. Suppose there is an alternating cycle of eligible edges C. C does not exist after the initialization of *match*, since there are no matched edges.

C is not created in a Relaxed Hungarian Search, for the following reasons: A relax operation does not create an eligible edge, so it does not create C. A dual adjustment does create eligible edges $e \notin M$, where $e_0 \in \mathcal{F}$, $e_1 \notin \mathcal{F}$. If C contains such an edge, it also contains an edge $f \notin M$ with $f_0 \notin \mathcal{F}$, $f_1 \in \mathcal{F}$. But f is ineligible after the dual adjustment.

Similar reasoning applies when the Augment Step creates new matched edges and changes duals. ∎

Now we analyze the efficiency of the Relaxed Hungarian Search.

Lemma 3.7. $H = O(b + gn \log n)$.

Proof. There are three possibilities for an iteration: (*i*) It adds at least f/g vertices to $V_0(\mathcal{F})$. (*ii*) It is the last or next-to-last iteration in its Hungarian search. (*iii*) The next iteration does a dual adjustment. These possibilities are exhaustive since if (*i*) does not hold and the Adjust Step does not halt, the Grow Step does a relax operation, making $W_1 = \emptyset$ in the next iteration.

Possibility (*ii*) occurs $O(\sqrt{b})$ times by Lemma 3.3. Possibility (*iii*) occurs at most b times, since Lemma 3.2 implies that the number of dual adjustments is at most b. Possibility (*i*) clearly occurs at most gn/f times in a given search. Each Hungarian search has a distinct value of f, since each Augment Step after the first does at least one augment. Thus (*i*) occurs less than $\sum_{f=1}^{n} gn/f = O(gn \log n)$ times, as desired. ▮

Lemma 3.8. $R \le (4b \log n)/g$.

Proof. Consider a Search Step that starts with f free vertices, whose dual adjustment quantities δ sum to some value d. The relax operations cause a total decrease in duals of at most $2fd/g$. To see this, observe that a relax operation that is not the last is followed by a dual adjustment. Hence there are at most $d+1$ relax operations, that decrease duals by at most $(d+1)f/g \le 2df/g$.

Thus $R \le \sum 2fd/g$, where the summation is over all Hungarian searches. Observe that $\sum fd$ (summation over all Hungarian searches) is precisely $\sum_{i=1}^{n} \Delta_i$. This follows since the duals of free vertices are changed only by dual adjustments. Now Lemma 3.4 implies the desired bound. ▮

The lemma and the definition of b imply that r can be chosen to be any value satisfying the inequality $r \ge 4(3r+5n)\log n/g$. Hence choose

$$r = 2n, \quad b = 11n, \quad g = 24\lceil \log n \rceil.$$

This implies that the number of scales is $O(\log(nN))$, and

$$I = O(\sqrt{n}), \quad A = O(n \log n), \quad H = O(n \log^2 n).$$

In the timing analysis we have assumed that all arithmetic operations use $O(1)$ time. To justify this we show that each dual $y(v)$ has magnitude $O(n^2 N)$. Since the input requires a word size of at least $\max\{\log N,\ \log n\}$ bits, the dual variables can be stored in at worst triple-word integers.

To show this first define Y_s as the largest magnitude of a dual value $y(v)$, $v \in V_0$, during the s^{th} scale. *match* increases $y(v)$ by at most $\Delta \le b$, and decreases it by at most $r < b$. Thus $Y_s \le 2Y_s + b - 1$, and $Y_0 = 0$. Hence $Y_s \le (2^s - 1)(b-1) = O(n^2 N)$. Hence the duals of V_0 satisfy the desired bound. When *match* changes the dual of a vertex $v \in V_1$, it preserves the relation $y(vv') = c(vv')$, for $vv' \in M$. So the duals of V_1 satisfy the desired bound.

It remains to describe the parallel implementation of the Relaxed Hungarian Search. It can be implemented so that the total time for all searches is

$$O((Im/p + b + H)\log p) \tag{3.4}$$

which is within the desired bound (3.2). Here we outline the ideas of the parallel implementation, leaving details to the reader.

First as in the Hungarian algorithm, it is most efficient to keep track of the dual variables by offsets. That is, the algorithm maintains the value of Δ. When a vertex v is added to $\mathcal{F}$, the current values of Δ and $y(v)$ are recorded as $\Delta^0(v)$ and $y^0(v)$, respectively. At any later time the value of $y(v)$ can be computed as $y^0(v) + (\Delta - \Delta^0(v)) \times (\textbf{if } v \in V_0 \textbf{ then } 1 \textbf{ else } -1)$. In a dual adjustment the value Δ gets changed, but no work need be done for the vertices in $\mathcal{F}$.

Second, the algorithm maintains lists of edges that may be used in future Grow Steps. The idea is similar to one proposed by Dial for Dijkstra's algorithm ([Dia]; see also [G85], [W]). For one processor the details are as follows. Each unmatched edge e with e_0 but not e_1 in $\mathcal{F}$ has a value $\Delta(e)$, such that if Δ reaches the value $\Delta(e)$ and e_1 is still not in $\mathcal{F}$, then edge e has become eligible and e_1 can be added to $\mathcal{F}$. The value $\Delta(e)$ is easily calculated when e_0 is added to $\mathcal{F}$, as $\Delta(e) = \Delta + c(e) + 1 - y(e)$. For each possible value Δ $(0 \le \Delta \le b)$ the algorithm maintains a list $L(\Delta)$ containing all edges e with $\Delta(e) = \Delta$. In the Adjust Step the set W_1 is found by examining the edges e on $L(\Delta)$ (for the current value of Δ). A dual adjustment is done by repeatedly increasing Δ by one until $L(\Delta)$ contains an edge e with $e_1 \notin V(\mathcal{F})$. In the Grow Step each edge e incident to W_0 is added to the appropriate list $L(\Delta(e))$. Note that $\Delta(e)$ is calculated after any relax operation, and there is no need to add e if $\Delta(e) > b$.

To implement this in parallel we allocate storage in "blocks" to allow edges on a list $L(\Delta)$ to be scanned in parallel. Specifically a *block* consists of $p+1$ consecutive words of memory. Define $s = \lceil \sqrt{b} \rceil$. The algorithm allocates space for $s + 1 + \lfloor m/p \rfloor$ blocks. The total space for this is $O(m)$, since $sp \le m$ by assumption on the number of processors. Blocks are used as needed. The unused blocks are consecutive in memory. The algorithm keeps a pointer to the first unused block so that the next block can be allocated when needed.

The algorithm uses a system of lists $L'(d)$, for $0 \le d \le s$. Each list $L'(d)$ consists of one or more blocks. Each block stores k edges in its first k words, for some $k \le p$; if $k = p$ then the $p+1^{st}$ word of the block points to the next block of $L'(d)$. If Δ is in the range $is \le \Delta < (i+1)s$ then for $0 \le d < s$, list $L'(d)$ corresponds to the above list $L(is + d)$; list $L'(s)$ corresponds to the union of all lists $L(is + d)$, $d \ge s$. Since at any time a given edge is in at most one block, the number of blocks used is as given above.

This data structure allows the edges of a list to be scanned in parallel. Specifically to scan the edges in a block, processor i accesses the edge stored in the i^{th} word of the block. Similarly edges are added to a list in parallel by placing them in consecutive locations at the end of the last block. If this last block does not have sufficient space a new block is allocated and added to the list.

Using sorting and parallel prefix computations as in Section 2, the time for all Adjust and

Grow Steps is given by (3.4).

The last detail concerns the processing when Δ takes on a value that is a multiple of s. Note from the definition of the data structure that at this point all lists $L'(d)$, $0 \le d \le s$, must be reinitialized, using edges currently on $L'(s)$. By Lemma 3.2, this reinitialization occurs at most $\lceil b/s \rceil \le \lceil \sqrt{b} \rceil$ times. Using sorting and parallel prefix computations the total time for this is $O(\sqrt{n}m(\log p)/p)$, as desired.

3.3. The Augment Step.

This section describes the Augment Step of *match*. Consider the Augment Step for some value Δ. Let A_Δ denote the total augmenting path length in this Augment Step. The algorithm of this section uses time

$$O((m/p + A_\Delta)\log p). \tag{3.5}$$

The bounds on I and A together with (3.5) imply that the total time for all Augment Steps is less than the desired bound (3.2).

The Augment Step works on the residual graph of the graph of eligible edges. This directed graph is acyclic, by Lemma 3.6. Thus it is easy to see that the Augment Step amounts to an algorithm for the following problem: Given a directed acyclic graph D with distinguished vertices s and t, find a maximal set $\mathcal{A}$ of vertex disjoint st-paths. This is the same problem as in Section 2, but now the graph is acyclic instead of layered. We present an algorithm for this problem.

For any graph D as in our problem, its ***reduction*** R is the subgraph induced by the vertices that are on st-paths. Equivalently R is the maximal subgraph of D such that every vertex except t has positive outdegree and every vertex except s has positive indegree. (Here indegree and outdegree refer to degrees in R. Section 2 uses a weaker notion of reduction.) Note that for any vertex v of R, a vt-path in R can be found by starting at v and repeatedly traversing an edge from the most recently reached vertex. An sv-path can be found similarly.

As in Section 2, for a subgraph H, $V_-(H)$ stands for $V(H) - \{s,t\}$.

The algorithm maintains the graph R as the reduction of $D - V_-(\mathcal{A})$. As in the Augment Step of Section 2, the algorithm repeatedly finds an st-path P, adds it to $\mathcal{A}$, and updates R by deleting $V_-(P)$ and all vertices whose indegree or outdegree drops to zero. The difficulty in this approach is that the vertex deletion time can be excessive. To see why, observe that the time to delete vertices for P is at least (a constant times) the length of any path Q of deleted vertices. In Section 2, R

is layered, so $|Q| \leq |P|$. This gives an acceptable bound on vertex deletion time. When R is not layered, $|Q|$ can be larger than $|P|$ — we know no bound on $|Q|$ except $n-1$. Thus the vertex deletion time might exceed the desired time bound.

The algorithm overcomes this difficulty with the following approach, based on doubling. The algorithm starts with a candidate path P. It determines the effect of adding P to $\mathcal{A}$ by tentatively deleting $V_{-}(P)$ and other vertices as appropriate. It checks if the time to do this is acceptable. If not, it uses tentatively deleted edges to construct a new st-path, over twice as long as P. It repeats the process for the new path. Eventually an acceptable path is found and added to $\mathcal{A}$.

This strategy is implemented in the algorithm *find_path* below. *find_path* is called on a graph R, the current reduction of $D - V_{-}(\mathcal{A})$. Its purpose is to add one path to $\mathcal{A}$ and update R. The In and Out Steps below estimate the deletion time by tentatively deleting vertices from R. These tentative deletions are either made permanent in the Double Step, or are ignored. Throughout this section, "tentatively deleting" a vertex or edge means tentatively deleting it from R.

procedure *find_path*.

Initialize P to be an arbitrary st-path. Then repeat the following steps until the Double Step adds the desired path to $\mathcal{A}$.

In Step. Tentatively delete all edges directed *from* $V_{-}(P)$. Then tentatively delete any vertex whose indegree has dropped to zero; repeat this until every vertex of $R - s$ has positive indegree. Let μ_i be the total number of edges tentatively deleted in this step. Let P_i' be a longest path of edges tentatively deleted in this step.

Out Step. Tentatively delete all edges directed *to* $V_{-}(P)$. Then tentatively delete any vertex whose outdegree has dropped to zero; repeat this until every vertex of $R - t$ has positive outdegree. Let μ_o be the total number of edges tentatively deleted in this step. Let P_o' be a longest path of edges tentatively deleted in this step.

Double Step. Set $\mu \leftarrow \mu_i + \mu_o$. Let P' be the longer path of P_i', P_o'. If $|P'| \leq 2(|P| + \mu/p)$ then make the deletions of the In and Out Steps permanent, delete $V_{-}(P)$, add P to $\mathcal{A}$ and halt. Otherwise ignore those tentative deletions; let S be a path from s to the first vertex of P'; let T be a path from the last vertex of P' to t; let P be the st-path formed by S, P' and T. ∎

The correctness of *find_path* amounts to the fact that if the routine is called with R a nonempty reduction graph, it eventually adds an st-path to $\mathcal{A}$ and halts. In the initialization, path P exists since R is a nonempty reduction graph. Similarly, in the Double Step paths S and T exist. Thus every iteration of *find_path* constructs a longer st-path. Hence *find_path* eventually halts as desired.

Before analyzing the efficiency of this routine, let us observe that it is not difficult to implement *find_path*. In particular in the In and Out Steps, paths P_i' and P_o' are readily available. To see why, consider for definiteness the Out Step and path P_o'. For a vertex v deleted in the Out Step, define $level(v)$ as follows. If $v \in V_{-}(P)$ then $level(v) = 0$; otherwise $level(v)$ is the smallest value such that for any edge vw, $level(w) \leq level(v) - 1$. Then the longest path of edges deleted in the Out Step starting at v has length exactly $level(v)$. Furthermore, such a longest path can start with any edge vw where $level(w) = level(v) - 1$. Thus P_o' can be found if for each vertex v, the algorithm records the edge that caused its outdegree to drop to zero.

Now we estimate the efficiency of *find_path*. Suppose *find_path* performs J iterations. For $1 \leq j \leq J$, let P_j be the candidate path P in the j^{th} iteration, and let μ_j be the number of edges tentatively deleted in the j^{th} iteration. (Path P is constructed immediately before the j^{th} iteration; μ_j is the value μ computed in the j^{th} iteration.) Thus P_J is the path that *find_path* adds to $\mathcal{A}$ and μ_J is the number of edges actually deleted from R. Let $P_{J+1} = P_J$. We shall see that *find_path* can be implemented so that the time for the j^{th} iteration is

$$O((\mu_j/p + |P_{j+1}|)\log p). \tag{3.6}$$

This implies the following bound.

Lemma 3.9. The time for one execution of *find_path* is $O((\mu_J/p + |P_J|)\log p)$.

Proof. From (3.6) it suffices to analyze the time for the first $J-1$ iterations. From (3.6) this time is $O(\log p)$ times $\sum_{j=1}^{J-1} \mu_j/p + \sum_{j=1}^{J-1} |P_{j+1}|$. The first summation is at most a constant times the second, since the Double Step implies that for $j < J$, $|P_{j+1}| > \mu_j/p$. The second summation is less than $2|P_J|$, since the Double Step implies that for $j < J$, $|P_{j+1}| > 2|P_j|$. ∎

The Augment Step works by repeatedly calling *find_path* until R becomes empty. Note that when the tentative deletions become permanent in the Double Step, R becomes the new reduction graph. Hence the entry condition for the next call to *find_path* is satisfied. A crucial part of the algorithm that is still unspecified is how R is initialized when the Augment Step begins (i.e., before the first call to *find_path*). Excluding that, it is clear that the Augment Step works correctly. The total time used is the sum of the bounds of Lemma 3.9, which equals (3.5).

Now we describe how R is initialized when the Augment Step begins. The first Augment Step of *match* is simple: There are no matched edges, whence R is the residual graph of the eligible edges. For an Augment Step after the first, the following routine is used. In addition to finding R it constructs a path P to be used as the first candidate path in *find_path*. Hence the routine ends by skipping the initialization of *find_path* and going directly to the In Step of *find_path*.

procedure *find_RP*.

R Step. Let $\mathcal{F}$ be the forest of the preceding Relaxed Hungarian Search. Do a breadth-first search of the eligible edges, as follows: Start the search from the free vertices of V_1 (not V_0). Stop the search upon reaching the first breadth-first level L that does not contain a vertex of $\mathcal{F}$. Set $V(R)$ to the vertices of $\mathcal{F}$ reached in the search. Construct $E(R)$ from the eligible edges that join vertices of R.

P Step. Let v be a vertex of $\mathcal{F}$ in the level preceding L. Let S be the alternating path of $\mathcal{F}$ from a free vertex of V_0 to v. Let T be the alternating path of the above breadth-first search, from v to a free vertex of V_1. Set P to be the st-path in R that corresponds to S followed by T. Go to the In Step of *find_path*. ▮

Now we show that *find_RP* is correct. Observe that the R Step constructs R correctly: When the Relaxed Hungarian Search halts, as noted in Section 3.2, $\mathcal{F}$ contains all vertices that are on an augmenting path of eligible edges. Hence $\mathcal{F}$ contains $V(R)$. Thus a vertex is in R if and only if it is joined to a free vertex of V_1 by an alternating path of eligible edges containing only vertices of $\mathcal{F}$. Such a vertex is reached by level L in the breadth-first search. This implies that R is initialized correctly.

In the P Step it is clear that the constructed path P exists and is in R. Hence *find_RP* is correct.

The time for *find_RP* is $O((m/p + |P|)\log p)$, since $|P| \geq |T| = |L| - 1$. The first augmenting path constructed by *find_path* will be at least as long as the path that it starts with, which is the P constructed by *find_RP*. Hence *find_RP* runs within the desired bound (3.5).

It remains to describe the parallel implementation of *find_path*. It can be implemented so the time for one execution is given by (3.6). As with the Relaxed Hungarian Search we leave most of the implementation details to the reader. The algorithm uses techniques similar to the Augment Step of Section 2. Tentative deletions are done using vertex and scan lists, and the paths S and T are found by the greedy strategy. In the data structure, each vertex v of G stores eight quantities:

its indegree and outdegree; a tentative indegree and tentative outdegree; a pointer to the undeleted eligible unmatched edge that is first in its adjacency list (if any); a pointer to the vertex to which it is matched (if any); if v is deleted in the In Step, a pointer to an edge directed to v in a longest path of deleted edges to v; a similar pointer to an edge directed from v.

This completes the derivation of Theorem 1.1. ▮

4. Extensions.

This section first presents an efficient algorithm for shortest paths in a directed graph with arbitrary integral edge lengths. Then it generalizes the assignment algorithm to the minimum cost degree-constrained subgraph problem.

4.1. Optimum duals and shortest paths.

Some applications of matching require the optimum linear programming duals. We begin by showing how such duals can be derived from r-optimal duals. Then, as an example, we show how this gives an efficient shortest path algorithm.

Let G^+ be G with an additional vertex $s \in V_0$ and an edge sv for each $v \in V_1$. Extend the given cost function c to G^+ by defining $c(sv)$ as an arbitrary integer; the cost function used by the *main routine* of our algorithm extends to G^+ by its definition, $\bar{c} = (r+n+1)c$. To specify a cost function on G^+ we write $G^+;c$ or $G^+;\bar{c}$. Let M be a minimum perfect matching on G; for vertex v let v' denote its mate, i.e., $vv' \in M$. For $v \in V_0$ let M_v be a minimum perfect matching on $G^+ - v; c$. (Such a matching exists, e.g., $M - vv' + sv'$.) Optimum linear programming duals are given by

$$y(v) = \textbf{if } v \in V_0 \textbf{ then } -c(M_v) \textbf{ else } c(vv') - y(v').$$

(This can be proved by an argument similar to the algorithm given below. Alternatively see [G87] for a proof from first principles.)

Recall the ordinary Hungarian search described in Section 3.2. Suppose such a search is done on $G^+;\bar{c}$, with matching M. It halts with a tree T of eligible edges, rooted at s. The construction of G^+ implies T is a spanning tree. For any $v \in V_0$, augmenting along the sv-path in T gives an r-optimal matching N_v on $G^+ - v;\bar{c}$. N_v is a minimum perfect matching on $G^+ - v; c$. This follows from Lemma 3.1, since $G^+ - v$ and G have the same number of vertices. Hence N_v qualifies as the above M_v.

This implies the following procedure to find optimum linear programming duals. Given is the output of our matching algorithm, i.e., an r-optimal matching on $G;\overline{c}$ with matching M and dual function y. Form $G^+;\overline{c}$, defining $c(sv) = \lceil y(v)/(r+n+1) \rceil$ for each $v \in V_1$. Set $y(s) \leftarrow 0$ to get r-feasible duals. Do an ordinary Hungarian search to construct a spanning tree T of eligible edges rooted at s. For a vertex $v \in V_0$, let P denote its path to the root in T. Compute v's linear programming dual as $c(M) + c(P \cap M) - c(P - M)$. Compute the dual of a vertex of V_1 using the above formula.

This algorithm can be implemented in time $O((m/p + n)\log p)$. The Hungarian search is implemented as in *match*. Note that the definition of $c(sv)$ ensures $\Delta \le n$. The duals are found by a depth-first traversal of T using one processor.

Corollary 4.1. Optimum linear programming duals on a bipartite graph can be found in the bound of Theorem 1.1. ▮

One application of these duals is to solve a shortest path problem.

Theorem 4.1. The single-source shortest path problem on a directed graph n vertices, m edges, and arbitrary integral edge lengths can be solved in the bound of Theorem 1.1.

Proof. This problem can be solved by finding optimum linear programming duals for a bipartite graph whose costs are the edge lengths, and then running Dijkstra's algorithm [G85]. The latter can be implemented in time $O((m/p + n)\log N \log p)$ using the scaling algorithm of [G85]. ▮

4.2. Degree-constrained subgraphs.

This section presents our algorithm for finding a minimum perfect DCS. The algorithm generalizes the matching algorithm of Section 3. Here we concentrate only on the aspects of the algorithm that are new. The analysis uses the same sequence of lemmas as Section 3. Most of the proofs here give only the facts needed to extend the argument of Section 3. The section also discusses several other DCS algorithms. Recall that the function u specifies the degree constraints; we denote a DCS by D, and the function d specifies the degree of a vertex in D.

Define a *relaxation function* to be a function $\rho : V \to \mathbf{N}$ (for $\mathbf{N}$ the nonnegative integers) that has $\rho(v) = 0$ for each $v \in V_1$. (The intent is that $\rho(v)$ equals the total amount that $y(v)$ has decreased in relax operations.) Note that in expressions such as $\rho(D)$ a vertex v contributes $\rho(v)d(v)$.

An *r-feasible DCS* consists of a degree-constrained subgraph D, a nonnegative integer r, a dual function y and a relaxation function ρ such that

$$y(e) \le c(e) + 1, \qquad e \notin D; \tag{4.1a}$$

$$y(e) \ge c(e) - \rho(e), \qquad e \in D; \tag{4.1b}$$

$$\rho(D) \le r. \tag{4.1c}$$

An *r-optimal DCS* is a perfect DCS that is r-feasible.

Lemma 4.1. If some integer larger than $r + n$ divides each cost $c(e)$ evenly, then any r-optimal DCS is a minimum perfect DCS.

Proof. Use the characterization that a perfect DCS D has minimum cost if and only if any alternating cycle C has $c(C \cap D) \le c(C - D)$. ∎

The algorithm is again stated using the integer parameters $r, b = 3r + 5U$ and g. We eventually choose $r, b = \Theta(U)$, $g = \Theta(\log U)$.

The *main routine* and the *scale_match routine* work exactly as in matching. The desired time bound for the algorithm follows if each scale runs in time

$$O((\sqrt{U}m/p \ + U\log^2 U)\log p). \tag{4.2}$$

Let D_- be the r-optimal matching of the previous scale. (For the first scale, D_- is any perfect DCS.) The costs input to *match* have these properties:

(a) Any edge not in D_- costs at least -1.

(b) Any subset of $E(D_-)$ costs at most $2r + 3U$.

The proof of (b) uses the nonnegativity of ρ.

In the *match* routine, edge e is *eligible* if equality holds in (4.1a) (for $e \notin D$) or (4.1b) (for $e \in D$). The *match* routine differs from the one in Section 3 in two respects: The first is initialization. Each relaxation amount $\rho(v)$ is set to 0. Further, the DCS D is initialized to $\{e | c(e) < -1\}$.

The second difference is the Augment Step. Define an *ap-set* to be a set of edge-disjoint augmenting paths of eligible edges, such that any vertex v is an end of at most $u(v) - d(v)$ paths. (In a multigraph, "edge-disjoint" means a given copy of an edge is in at most one path.) The Augment Step finds a maximal ap-set and augments the DCS along each path. Unlike Section 3, no duals are changed after an augment.

The properties of the Search and Augment Steps are similar to Section 3, with these changes: The definition of f is changed to the deficiency of the DCS, i.e.,

$$f = u(V_0) - d(V_0).$$

In addition to relation (3.3), any free vertex $v \in V_0$ has $\rho(v) = 0$ (it is never relaxed).

The correctness of *match* follows as in Section 3, using these observations to show r-feasibility: The initialization of D guarantees (4.1a); also the initial D is included in D_- by property (a), so it satisfies the degree constraints. Finally, the Augment Step does not increase $\rho(D)$, since a free vertex v has $\rho(v) = 0$.

The analysis of the efficiency of *match* follows Section 3:

Lemma 4.2. At any time during *match*, $f\Delta \le b$.

Proof. Consider the expression $Y = y(D_- - D) - y(D - D_-)$. ▌

Lemma 4.3. $I < 2\sqrt{b} + 1$.

Proof. In the Augment Step the edges on an augmenting path become ineligible, by the definition of "eligible" and the nonnegativity of ρ. This implies that any Hungarian search does a dual adjustment. ▌

Lemma 4.4. $\sum_{i=1}^{U} \Delta_i \le b + b \log U$. ▌

Lemma 4.5. $A \le U + b + b \log U$.

Proof. By the argument of Lemma 3.5 and the nonnegativity of ρ, $\sum_{i=1}^{U} \Delta_i \ge A + c(D_U) - c(D_0)$. The initialization of *match* shows that an edge in $D_U - D_0$ costs at least -1. So Lemma 4.4 implies the desired bound. ▌

Now we turn to the Relaxed Hungarian Search. It differs from Section 3 as a consequence of the fact that an edge of D need not be eligible. Hence the search forest $\mathcal{F}$ is grown edge-by-edge, rather than in pairs of unmatched and matched edges. Thus the dual adjustment quantity is defined as

$$\begin{aligned}\delta = \min\{c(e) + 1 - y(e) \mid e \notin D,\ e_0 \in \mathcal{F},\ e_1 \notin \mathcal{F}\}\\ \cup \{y(e) - c(e) + \rho(e) \mid e \in D,\ e_0 \notin \mathcal{F},\ e_1 \in \mathcal{F}\}.\end{aligned} \tag{4.3}$$

To *relax* a set of nonfree vertices $S \subseteq V_0$ means to decrease $y(v)$ by 1 and increase $\rho(v)$ by 1, for each $v \in S$. This operation makes any non-D-edge that is incident to S ineligible. It does not change the eligibility of any D-edge. Concerning r-feasibility, a relax operation preserves (4.1a)–(4.1b). Concerning (4.1c), it increases $\rho(D)$ by $u(S)$.

The Relaxed Hungarian Search works as follows. It initializes the search forest $\mathcal{F}$ to contain the free vertices of V_0. Then it repeats the following steps until the Adjust Step halts with $\mathcal{F}$ as desired.

Adjust Step. Set $W_1 \leftarrow \{e_1 \mid \text{some eligible edge } e \notin D \text{ has } e_0 \in \mathcal{F},\ e_1 \notin \mathcal{F}\}$, $W_0 \leftarrow \{e_0 \mid \text{some eligible edge } e \in D \text{ has } e_1 \in W_1,\ e_0 \notin \mathcal{F}\}$, $W_2 \leftarrow \{e_0 \mid \text{some eligible edge } e \in D \text{ has } e_1 \in \mathcal{F},\ e_0 \notin \mathcal{F}\}$. If $W_1 \cup W_2 = \emptyset$ and $\mathcal{F}$ contains a free vertex of V_1, halt. If $W_1 \cup W_2 = \emptyset$ and $\mathcal{F}$ does not contain a free vertex of V_1, do a dual adjustment and repeat this step.

Grow Step. For each vertex $w \in W_1 \cup W_2$, add an appropriate eligible edge vw ($v \in \mathcal{F}$) to $\mathcal{F}$; then do the same for each $w \in W_0$. If $u(W_0 \cup W_2) < f/g$ then relax $W_0 \cup W_2$. ∎

This routine works analogously to the one in Section 3. Note that after the Grow Step adds edges, an eligible edge with exactly one vertex in $\mathcal{F}$ is either a non-D-edge or is incident to $W_0 \cup W_2$. If a relax operation is done, it makes the non-D-edges incident to $W_0 \cup W_2$ ineligible. Hence the next Adjust Step halts or does a dual adjustment.

As in Section 3 we will choose parameter r as an upper bound to $\rho(D)$. The correctness of the Relaxed Hungarian Search is proved as in Section 3.

Now we establish the two properties needed by the Augment Step. The first is that $\mathcal{F}$ contains all vertices that are on an augmenting path of eligible edges. When the search halts an eligible edge e with exactly one vertex in $\mathcal{F}$ is either a D-edge with $e_0 \in \mathcal{F}$ or a non-D-edge with $e_1 \in \mathcal{F}$. Since an augmenting path starts at a vertex of $V_0(\mathcal{F})$ and is alternating, it cannot leave $\mathcal{F}$ on such an edge.

The second property is acyclicity:

Lemma 4.6. In *match* there is never an alternating cycle of eligible edges.

Proof. Consider an alternating cycle of eligible edges C. C does not exist after the initialization of *match*, since every D-edge is ineligible. C is not created by an Augment Step or relax operation, since neither creates an eligible edge. It remains only to show that C is not created by a dual adjustment.

A dual adjustment can create an eligible edge e that has exactly one of its vertices in $\mathcal{F}$. Suppose C contains such an e. Let f be the first edge after e in C with exactly one vertex in $\mathcal{F}$. The two possibilities for e are $e \notin D$ with $e_0 \in \mathcal{F}$, or $e \in D$ with $e_1 \in \mathcal{F}$. In either case since C is alternating, the two possibilities for f are $f \notin D$ with $f_1 \in \mathcal{F}$, or $f \in D$ with $f_0 \in \mathcal{F}$. But such an edge is ineligible after the dual adjustment. ▮

Now we analyze the efficiency of the Relaxed Hungarian Search.

Lemma 4.7. $H = O(b + gU \log U)$.

Proof. The only change in the argument is the definition of possibility (i) . It becomes: (i) The iteration increases $u(V_0(\mathcal{F}))$ by at least f/g. This occurs at most gU/f times in a given search. ▮

Lemma 4.8. At any time, $\rho(D) \leq (4b \log U)/g$.

Proof. A relax operation increases $\rho(D)$ by at most f/g. ▮

The rest of the development — choosing parameters, implementing of the Relaxed Hungarian Search, the Augment Step and its analysis — is entirely analogous to that in Section 3.

The following result is derived by proceeding as in Section 3.

Theorem 4.2. A minimum perfect degree-constrained subgraph on a bipartite multigraph can be found in time $O(\sqrt{U} m \log(nN)(\log p)/p)$ and space $O(m)$, for $p \leq m/(\sqrt{U} \log^2 n)$. ▮

Now consider the *general minimum cost degree-constrained subgraph problem* on a bipartite multigraph G. Each vertex v has given degree bounds $\ell(v)$ and $u(v)$. Also given are bounds on the cardinality of the solution, L and H. We seek a degree-constrained subgraph (i.e., for each vertex v, $\ell(v) \leq d(v) \leq u(v)$) that has minimum cost subject to the restriction that it contains between L and H edges (inclusive). Note that special cases of this problem include minimum cost matching ($\ell = 0, u = 1, L = 0, H = n$), minimum cost cardinality-k matching (change L and H to k), minimum cost maximum cardinality matching, and similar DCS problems.

It is straightforward to reduce this general problem on a multigraph G to a minimum perfect DCS problem on a multigraph G^*. For $i = 0, 1$, define $i' = 1 - i$ and add a vertex s_i to V_i. For each $v \in V_{i'} - s_{i'}$, add edge $s_i v$ with multiplicity $u(v) - \ell(v)$ and cost zero. Also add edge $s_0 s_1$ with multiplicity $H - L$ and cost zero. Set $u(s_i) = u(V_{i'} - s_{i'}) - L$. It is easy to see that a minimum perfect DCS on G^* is a solution to the general problem. This gives the following result. Define U as the sum of the upper bounds of the given multigraph G.

Corollary 4.2. The general minimum cost degree-constrained subgraph problem on a bipartite multigraph can be solved in time $O(\sqrt{U} m \log (nN)(\log p)/p)$ and space $O(m)$, for $p \leq m/(\sqrt{U} \log^2 n)$. ∎

Acknowledgments. We thank Andrew Goldberg and Serge Plotkin for helpful conversations.

References.

[B] R.P. Brent, "The parallel evaluation of general arithmetic espressions", *J. ACM 21*, 2, 1974, pp. 201-206.

[C] R. Cole, "Parallel merge sort", *SIAM J. Comput. 17*, 4, 1988, pp. 770–785.

[Dan] G.B. Dantzig, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, N.J., 1963.

[Dia] R.B. Dial, "Algorithm 360: Shortest path forest with topological ordering", *C. ACM 12*, 1969, pp. 632-3.

[Dij] E. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik 1*, pp. 269-271, 1959.

[Din] E.A. Dinic, "Algorithm for solution of a problem of maximum flow in a network with power estimation", *Sov. Math. Dokl. 11*, 5, 1970, pp. 1277-1280.

[ET] S. Even and R.E. Tarjan, "Network flow and testing graph connectivity", *SIAM J. Comput. 4*, 4, 1975, pp. 507-518.

[FM] D. Fernandez-Baca and C.U. Martel, "Theoretical efficiency of maximum flow algorithms on networks with small integer capacities", *Algorithmica,* to appear.

[FT] M.L. Fredman and R.E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms", *J. ACM* 34, 3, 1987, pp.596-615.

[G85] H.N. Gabow, "Scaling algorithms for network problems", *J. of Comp. and Sys. Sciences 31*, 2, 1985, pp. 148-168.

[G87] H.N. Gabow, "Duality and parallel algorithms for graph matching", manuscript.

[GaT87] H.N. Gabow and R.E. Tarjan, "Faster scaling algorithms for network problems", *SIAM J. Comp.*, to appear.

[GoT] A.V. Goldberg and R.E. Tarjan, "Solving minimum-cost flow problems by successive approximation", *Proc.* 19th *Annual ACM Symp. on Th. of Computing*, 1987, pp. 7-18.

[GP] Z. Galil and V. Pan, "Improved processor bounds for algebraic and combinatorial problems in RNC", *Proc.* 26th *Annual Symp. on Foundations of Comp. Sci.*, 1985, pp. 490-495.

[GSS] L.M. Goldschlager, R.A. Shaw and J. Staples, "The maximum flow problem is logspace complete for *P*", *Theoretical Comp. Sci. 21*, 1982, pp. 105-111.

[GW] H.N. Gabow and H.H. Westermann, "Forests, frames and games: algorithms for matroid sums and applications" *Proc. 20th Annual ACM Symp. on Th. of Computing*, 1988, pp. 407 – 421.

[HK] J. Hopcroft and R. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM J. Comp. 2*, 4, 1973, pp. 225-231.

[KUW] R.M. Karp, E. Upfal and A. Wigderson, "Constructing a perfect matching is in Random NC", *Combinatorica* 6,1, 1986, pp. 35-48.

[MVV] K. Mulmuley, U.V. Vazirani and V.V. Vazirani, "Matching is as easy as matrix inversion", *Combinatorica* 7,1, 1987, pp. 105-113.

[KC] T. Kim and K.-Y. Chwa, "An $O(n \log n \log \log n)$ parallel maximum matching algorithm for bipartite graphs", *Inf. Proc. Letters 24*, 1987, pp. 15-17.

[L] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[SV] Y. Shiloach and U. Vishkin, "An $O(n^2 \log n)$ parallel MAX-FLOW algorithm", *J. Algorithms 3*, 1982, pp. 128-146.

[T] R.E.Tarjan, *Data Structures and Network Algorithms*, SIAM Monograph, Philadelphia, Pa., 1983.

[W] R.A. Wagner, "A shortest path algorithm for edge-sparse graphs", *J. ACM 23*, 1, 1976, pp. 50-57.