AN EFFICIENT REDUCTION TECHNIQUE FOR DEGREE-CONSTRAINED SUBGRAPH AND BIDIRECTED NETWORK FLOW PROBLEMS

by

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

Abstract

Efficient algorithms are given for the bidirected network flow problem and the degree-constrained subgraph problem. Four versions of each are solved, depending on whether edge capacities/multiplicities are one or arbitrary, and whether maximum value/maximum cardinality or minimum cost/maximum weight is the objective. A version of the shortest path problem is also efficiently solved. The algorithms use a reduction technique that solves one problem instance by reducing to a number of problems.

1. Introduction

Bidirected network flow, introduced by Jack Edmonds [E67,L], models a broad class of integer linear programming problems, including ordinary network flow, graph matching, degree-constrained subgraphs, shortest paths and others. It is well-known that problems in this class can be solved in polynomial time by matching techniques [L, EJ70]. Two approaches have been used. The first is to apply the ideas of matching to the more general problem and work out the details of an efficient algorithm [e.g., EJ73, U, W]. This can be done "in principle" but made difficult by the complex structure of matching blossoms. In fact this conceptual complexity has apparently prevented researchers from developing good algorithms for some of these problems (see our list below). The second approach is to use a problem reduction, from the more general problem to a well-understood one. The drawback of this technique is expansion in problem size, which can give nonpolynomial algorithms [L] or can degrade the performance by one or more orders of magnitude [Berg, Gol, Sh].

This paper presents an efficient reduction technique for bidirected network flow problems. The major difference from previous work is that we do *not* attempt to reduce one problem instance to another. Instead a number of different reductions are used to solve one problem instance. Our results are for the following bidirected flow problems:

(1) *Maximum cardinality, unit capacity problems.* *(i) Degree-constrained subgraph (DCS).* Given a graph where each vertex i has integer bounds l_i and u_i. Find a subgraph H with the greatest possible number of edges, such that each vertex i has degree d_i (in H) with $l_i \le d_i \le u_i$. Our algorithm runs in $O(\sqrt{\sum_{i \in V} u_i}E)$ time. This generalizes the maximum cardinality matching algorithm of Micali and Vazirani [MV] (where all $l_i = 0$, $u_i = 1$ and the time is $O(\sqrt{V}E)$), and in fact our algorithm is a reduction to theirs. It improves the $O((\sum_{i \in V} u_i)V^3)$ algorithm of Urquhart [U].

(ii) *Bidirected network flow (biflow).* Given a bidirected network with unit edge capacities, find a maximum value flow. Our algorithm has run time $O(E^{\frac{3}{2}})$. This generalizes the result of Even and Tarjan [ET] which achieves the same time bound for the directed case.

(2) *Maximum cardinality, arbitrary capacity problems.* (i) Given a DCS problem as in (1) on a multigraph, where each edge e has an integral multiplicity μ_e. Find a degree-constrained subgraph with the greatest possible number of edges. (ii) Given a biflow problem as in (1), where each edge e has integral capacity c_e. Find a maximum value flow. For both problems our algorithms have run time $O(V E \log V)$. This generalizes the algorithm of Sleator and Tarjan [Sl, ST] for directed graphs (and our algorithm uses theirs. Note however that they allow *real*-valued capacities).

Recent work of Anstee [A] offers a competitive approach. We can implement his algorithm for the f-factor problem (DCS where $l_i = u_i$ for all vertices i) in $O(V E \log V)$ time, the same bound as ours. His method is based on solving one network flow problem and one problem of the matching type.

(3) *Maximum weight, unit capacity problems.* (i) Given a DCS problem as in (1), where in addition each edge has a real-valued weight. Find a degree-constrained subgraph of maximum weight. Our algorithm runs in $O((\sum_{i \in V} u_i)\min(E \log V, V^2))$ time. This generalizes the maximum weight matching algorithm of Galil, Micali, and Gabow [G, GMG] (and our algorithm uses theirs). It improves the algorithm of Urquhart [U] which is $O((\sum_{i \in V} u_i)V^3)$.

(ii) Given a biflow problem as above, where in addition each edge has a real-valued cost. Find a minimum cost flow of a prespecified value. Our algorithm runs in $O(E \min(E \log V, V^2))$ time.

(4) *Shortest paths in an undirected graph.* Given an undirected graph where each edge has a real-valued length; edges may have negative lengths but there are no negative cycles. Find the shortest path between a given pair of vertices, or more generally find the shortest path between all pairs of vertices. This problem cannot be solved by the standard algorithms for directed graphs [L]. It is an instance of a "natural" biflow problem. Our algorithm for this problem (single pair or all pairs version) runs in $O(V \min(E \log V, V^2))$ time (compared with

directed graphs, this matches the bound for the all-pairs problem, and compares to $O(VE)$ for the single source problem [T].) Bernstein [Bern] has recently claimed an $O(V^4)$ algorithm for this problem, based on Dijkstra's shortest path algorithm.

Other applications of the DCS problem include efficient algorithms for the matroid parity problem on matching matroids and their variants [GS].

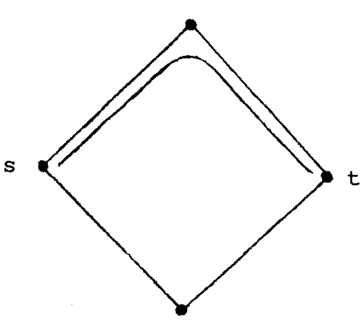
(5) *Maximum weight problems.* (i) The problem is the maximum weight DCS problem for multigraphs. (ii) The problem is the minimum cost biflow problem (with arbitrary integral capacities). Our algorithm runs in $O(E^2(log\ V)(log\ C))$ time where C is the largest capacity. It resembles the algorithm of Edmonds and Karp for the minimum cost network flow problem [EK]. (Actually it gives improved results for the special case of network flows, e.g., an $O(VE(log\ C))$ algorithm for maximum value network flow, and others).

The rest of this paper is organized as follows. Section 2 defines the above problems and also the upper degree-constrained subgraph problem (UDCS); we reduce all problems to UDCS. This section also sketches our reduction technique for augmenting paths. Sections 3-6 sketch the algorithms for problems (1) - (4). (Details that have been omitted due to space limitations can be found in [G 83]). (5) will be discussed elsewhere.

2. Basic Problems and Reductions

The three problems we investigate, stated in general form, are as follows.

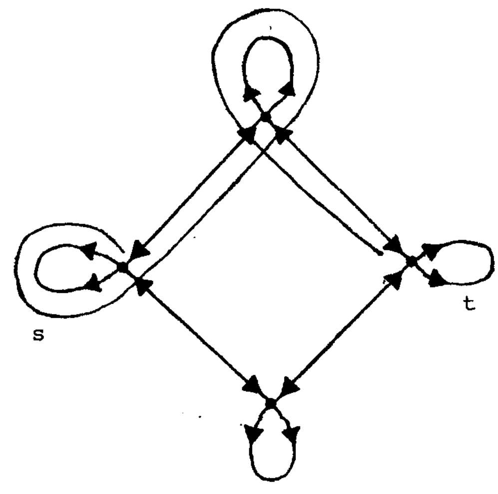
(i) *Bidirected network flow* [L, pp. 223-4]. In a directed graph an edge goes *from* one vertex *to* another. A *bidirected graph* allows this possibility and two others: an edge may be directed *from* both of its end vertices, or *to* both of them. (Additionally, the two vertices of an edge may coincide.)



(a)

Figure 2.1

(a) An undirected graph with a path.
(b) Corresponding bidirected graph and path.



(b)

Figure 2.1 illustrates the usefulness of this concept by giving an undirected graph with a path and the corresponding bidirected graph and path. (A *bidirected path* is a sequence of vertices and edges $v_0, e_1, v_1, \ldots, e_l, v_l$, such that if e_i is directed *to (from)* v_i then e_{i+1} is directed *from (to)* v_i. Paths in the undirected and bidirected versions correspond. This correspondence is achieved *without* duplicating edges, as is done in the correspondence between undirected and directed graphs. This allows one to solve the shortest path problem ((4) of Section 1).

Bidirected network flow (biflow) problems are defined from bidirected graphs, by analogy with ordinary directed flows, as illustrated by the above definition of path. Details are in [L, pp. 223-4].

(ii) The *degree-constrained subgraph (DCS) problem* is on an undirected graph, with lower and upper bounds l_i and u_i at each vertex i (see Section 1).

(iii) The *upper degree-constrained subgraph (UDCS) problem* is the special case of DCS where all lower bounds l_i are 0. Any feasible subgraph of G (i.e., one that satisfies the degree constraints u_i) is a *UDCS (upper degree-constrained subgraph)*.

We reduce biflow and DCS to UDCS. In this paper the reductions for biflow are omitted (see [L, pp. 224-225] for a related construction).

Now we sketch the reduction technique that forms the theme of this paper. It reduces a UDCS problem to matching problems. Consider a UDCS problem on a graph G. Fix a vertex i. Let $u = u_i$ be the given degree bound; let d be i's degree in G; define $\Delta = d-u$, the least possible number of unchosen edges. It is well-known [e.g., Berg] that a UDCS on G corresponds to a matching on G', where G' is constructed from G by replacing each vertex i by a *substitute* S that is the complete bipartite graph $K_{\Delta,d}$, as in Figure 2.2. The figure shows how a UDCS of G corresponds to a matching on G' (wavy edges are in the UDCS in Figure 2.2(a) and in the matching in Figure 2.2(b).) m denotes the number of edges incident to i in the UDCS. In this reduction a UDCS on G corresponds to a matching on G' that covers every internal vertex of every vertex substitute. *(Internal* and *external* vertices of a substitute are indicated in Figure 2.2.) Further, maximum cardinality and weight subgraphs correspond.

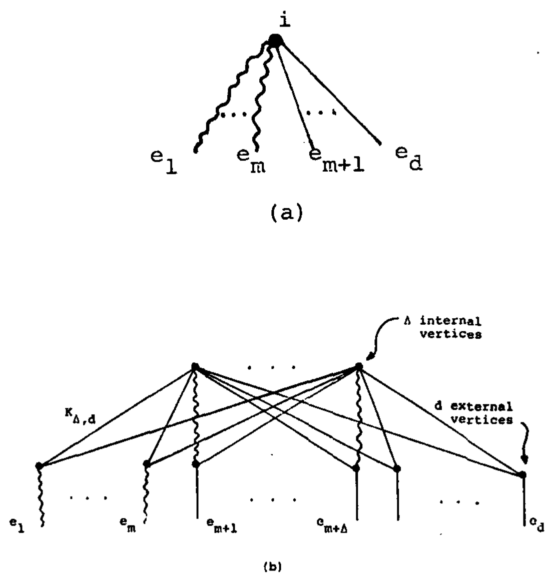


Figure 2.2

(a) Vertex i in UDCS on G.
(b) Vertex substitute S in matching on G'.

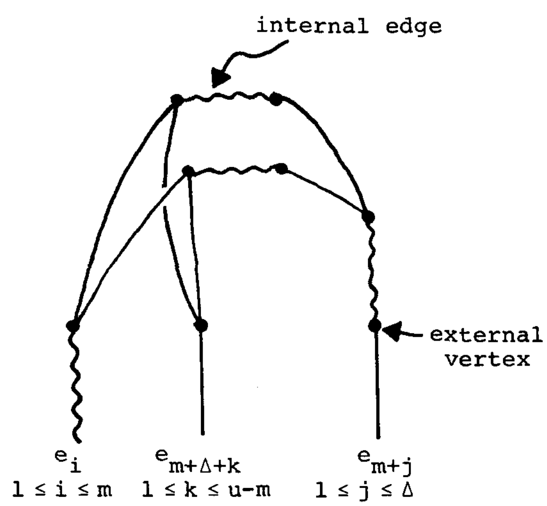


Figure 2.3

Sparse substitute for i.

This reduction is inefficient since it can increase the number of edges to $\Omega(VE)$. (For instance, if there are $\Omega(\frac{E}{V})$ vertices of degree $\Omega(V)$ and each such vertex i has $u = \frac{d}{2}$, the number of edges added is $\Omega(\frac{E}{V}V^2) = \Omega(VE)$). However we will show that an augmenting path need only pass through a given substitute S twice. Because of this the substitute S in Figure 2.2 can be replaced by the *sparse substitute* shown in Figure 2.3. Here the two (matched) *internal edges* correspond to the two uses of the substitute. Observe that a sparse substitute is defined *with respect to* a given matching; when the matched edges incident to i change, the substitute changes. Also note that sparse substitutes are efficient: The number of edges added for one sparse substitute is $2m + 2(u-m) + 3\Delta + 2 = O(d)$, so the total number of edges for all substitutes is $O(E)$. (The number of vertices added is $O(E)$ but this is not important.)

Our algorithm work by simulating the appropriate matching algorithm on G'. The simulation is done on a graph G_k, identical to G' except that sparse substitute are used. Each time a new matching is formed on G', a new graph G_{k+1} is formed by using sparse substitutes for the new matching. Since all graphs G_k have $O(E)$ edges, the matching algorithm runs fast and an efficient algorithm is maintained.

The technical difficulties in carrying out this approach are of two types. The cardinality matching algorithm finds a number of augmenting paths simultaneously. This causes difficulties in the simulation on G_k. The weighted matching algorithm maintains a structure from one augment to the next. This causes difficulties in the switch from G_k to G_{k+1}.

We close this section by briefly reviewing the notion of a matching blossom. Familiarity with the basic ideas of matching such as augmenting paths is assumed [see e.g., L].

A *blossom* is a subgraph B of a matched graph, defined as follows. (See Figure 2.4.) Let $k \ge 1$ be an integer. The vertices of B are partitioned into sets B_i, $1 \le i \le 2k+1$, where each B_i either consists of a single vertex or is itself a blossom. The edges of B are e_i, $1 \le i \le 2k+1$, where e_i is incident to a vertex in B_i and a vertex in B_{i+1} (when $i = 2k+1$, take $i+1$ to be 1). e_i is a matched edge iff i is even.

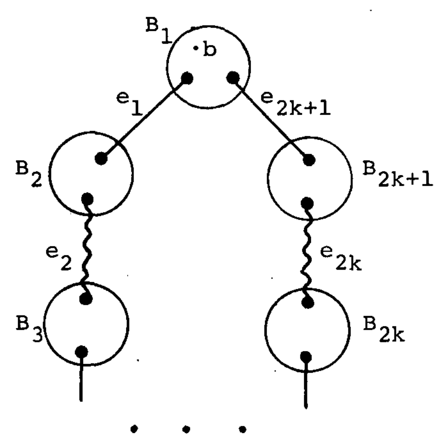


Figure 2.4

A blossom.

The shorthand $i \in B$ means that i is a vertex of blossom B. (Note that a blossom is not an induced subgraph, so we may have $i, j \in B$ without edge ij being in B.) A simple induction shows that except for one vertex $b \in B_1$, every vertex $i \in B$ has a matched edge ij with $j \in B$. The exceptional vertex b is the *base* vertex of B. Another induction shows that for each vertex $i \in B$, B and its subblossoms contain an alternating path that starts with a matched edge, from i to b. (If this path is $i, j, k, \ldots, b$, it is not usually true that $k, \ldots, b$ is k's path, e.g., in Figure 2.4 let $e_2 = ij$ and $j, k \in B_3$. This leads to pitfalls for the unwary - the open literature contains a number of blunders about blossoms!)

For cardinality matching blossoms are slightly simpler: Subblossoms such as B_2 and B_{2k+1} that are an odd distance from b are always vertices. We do not use this property here.

3. Maximum Cardinality, Unit Capacity Problems

This section presents algorithms for the maximum cardinality UDCS and DCS problems that use $O(\sqrt{\Sigma u_i}E)$ time and $O(E)$ space. We start with the UDCS problem.

Our approach is to simulate the cardinality matching algorithm on G', the graph with vertex substitutes. Recall how the cardinality matching algorithm works [HK]: An *sap* is a shortest length augmenting path. An *sap set* is a maximal set of vertex disjoint *sap*'s. The algorithm is organized into *phases*. Each phase finds an *sap* set, and then augments the matching along the paths of the set. The length of an *sap* increases every phase.

We can assume that any matching we construct on G' covers every internal vertex of every substitute. This allows us to estimate the number of phases of the matching algorithm on G'.

Lemma 3.1 At most $\frac{5}{2}\sqrt{\Sigma u_i}$ phases are needed to find a maximum matching on G'.

Proof. The argument is analogous to ones in [HK] and [ET]. ▪

The Lemma implies that to achieve our time bound it suffices to implement a phase of the matching algorithm on G' in $O(E)$ time. We do this by running each phase on the graph G_k. G_k is derived from G' and the current matching by using sparse substitutes. We must show that G_k is a correct model for G', i.e., an *sap* set of G_k gives an *sap* set of G'.

We begin with the basic principle behind the idea of sparse substitutes. Consider the graph G', with a matching that covers all internal vertices. Let S be a vertex substitute. Let P be an *sap* consisting of edges $e_1, \ldots, e_r$. Internal edges of S occur as pairs in P, say e_i, e_{i+1}. In each pair one edge is matched and the other is not. We say P traverses a pair in one of two *directions*, depending on whether the matched edge is first or second.

Cardinality Matching Reduction Principle. An *sap* traverses at most two pairs of edges from a given substitute, one in each direction.

Proof. Suppose P traverses two pairs in the same direction. So P has the form $\ldots, uv, vw, \ldots, xy, yz, \ldots$, where vertices v and y are in the same substitute, and edges uv and xy are matched. P can be shortened by replacing the subpath from v to z with the edge vz. This contradiction proves the result. ▪

Note that if G is bipartite only one direction, and hence one pair of edges, is possible for a substitute.

This principle implies that *sap*'s in G' and G_k correspond. We must show that *sap sets* on the two graphs correspond. In fact they do *not*: an *sap* set may pass through a substitute up to two times on *each* augmenting path. We analyze how an *sap* set uses a substitute, and show that G_k can still be used as a correct model.

It is convenient to work with a graph G'_k that is intermediate between G' and G_k. G'_k uses the same sparse substitutes as G_k; the only difference is that the substitute for a vertex i of G contains d_i internal edges (as opposed to two internal edges in G_k).

Lemma 3.2. An *sap* set in G'_k corresponds to an *sap* set in G' containing the same edges of G. ▪

The Lemma shows that we can take our goal to be finding an *sap* set on G'_k. We analyze the structure of such a set by using ideas from the cardinality matching algorithm of [MV]. Consider an arbitrary matched graph. For a vertex v, the *even level* of v, $e(v)$, is the length of a shortest even length alternating path from a free vertex to v; the *odd level* $o(v)$ is defined similarly. (We also refer to "paths defining $e(v)$or $o(v)$ ", with the obvious interpretation.) The *tenacity* of an edge e, $t(e)$, is the length of a shortest alternating path that contains e and ends at free vertices, but is *not necessarily* simple. (If it is simple it is an augmenting path.) So for $e = vw$, $t(e)$ is $o(v) + o(w) + 1$ if e is matched and $e(v) + e(w) + 1$ otherwise. A *blossom B of tenacity t* is defined, as in Section 2, from blossoms $B_1, \ldots, B_{2k+1}$ and edges $e_1, \ldots, e_{2k+1}$. The only difference is the added requirement that blossoms B_i have tenacity at most t, and edges e_i have tenacity t. [1]

We will refer to two simple but important properties of blossoms. In an arbitrary matched graph, let an *sap* have length $2s+1$.

Cardinality Blossom Properties

(i) Let vw be an edge of tenacity $t(vw) < 2s+1$. Then some blossom of tenacity at most $t(vw)$ contains both vertices v and w.

(ii) Let B be a maximal blossom of tenacity t, where $t < 2s+1$, and let b be its base. For any vertex $v \in B$, any path defining $e(v)$ or $o(v)$ passes through b.

These properties are obvious in [MV]. Alternatively they can be proved directly from the definitions. (It is convenient to prove them together, inducting on t. We leave this as an exercise.)

We show that *sap* sets on G'_k can be found using G_k. It is convenient to define a relation of "similarity" between vertices in G_k and G'_k: Let v and v' be vertices in either of the two graphs (perhaps the same graph). Then v and v' are *similar* if they are in substitutes for the same vertex of G, and either they are external vertices on the same edge of G, or they are matched to external vertices on the same edge of G, or they are vertices on the same (left or right) side of an internal edge.

[1] This definition can easily be proved equivalent to the one in [MV]: maximal blossoms of a given tenacity are identical in both definitions. We use our definition since it is the same as for weighted matching. It also appears to simplify the algorithm of [MV], eliminating the Double Depth First Search.

Lemma 3.3 (i) If v and v' are similar vertices then their levels are equal: $e(v) = e(v')$ and $o(v) = o(v')$.

(ii) If v and v' are similar vertices in the same graph, any maximal blossom of tenacity t contains both vertices or neither.

(iii) Let B be a maximal blossom of tenacity t in G_k or G'_k. Let B' consist of all vertices in the other graph that are similar to a vertex of B. Then B' is a maximal blossom of tenacity t.

Proof Sketch. (i) Uses the Cardinality Matching Reduction Principle. (ii) and (iii) use Cardinality Blossom Property (i).

Now we show how to find an *sap* set on G'_k using G_k. First we give a high-level description for a phase in the matching algorithm of [MV]: Let an *sap* have length $2s+1$.

Step 1. Calculate all levels $e(v)$ and $o(v)$ that are at most $s+1$. Construct all blossoms that have tenacity less than $2s+1$.

Step 2. Repeat the following steps until the graph does not contain an *sap* of length $2s+1$:

> *Step 2a.* Use level numbers and blossoms to find an *sap* P. Augment the matching along P.
>
> *Step 2b.* Delete vertices (along with their incident edges) that cannot be in an *sap:* First delete all vertices of P. Then repeatedly delete vertices (that are not in blossoms) whose "predecessor count" (see [MV]) decreases to 0. Furthermore, whenever the base of a blossom is deleted, delete all vertices in the blossom. Continue with Step 2.

When Step 2 ends, all vertices and edges of the graph are restored and the next phase is begun.

Note that Cardinality Blossom Property (ii) justifies the blossom deletion policy in Step 2b. It implies that any *sap* containing a vertex of a blossom B contains the base of B. Hence B can be deleted when its base occurs on an *sap* or becomes unreachable.

Consider how this algorithm works on G'_k. In a given substitute, at most two internal edges are deleted because they are in P. All other deletions in Step 2b remove *all* internal edges of the substitute. This is true because all internal edges are in the same maximal blossom, by Lemma 3.3(ii); also "predecessor counts" are based on level numbers and vertex adjacencies, which are the same for each internal edge by Lemma 3.3(i).

We can run this algorithm on G_k instead of G'_k and still find an *sap* set of G'_k. Step 1 is the same on both graphs, by Lemma 3.3 (i) and (iii). Step 2 on G_k will simulate Step 2 on G'_k if we make one modification: When P passes through a substitute whose internal edges are *not* in a blossom, these internal edges are *not* deleted (nor are they rematched in the augment). The reason is that in G'_k the substitute has d_i-2 other internal edges that can be used in other *sap's*. We keep the two internal edges in G_k to model these edges. On the other hand, all other deletions in Step 2 remove all internal edges of a substitute in either path, and so work the same in G_k and G'_k.

Thus we have shown that the matching algorithm (with the slight change given above) finds an *sap* set of G'_k. This gives the desired result.

Theorem 3.1. A maximum cardinality UDCS can be found in $O(\sqrt{\Sigma u_i}E)$ time and $O(E)$ space. ▪

We turn our attention to the DCS problem. Recall that in this problem each vertex i has both an upper bound u_i and a lower bound l_i on its degree. We will transform DCS so our UDCS algorithm applies. (This problem reduction approach differs from previous ones [U,S].)

Consider a DCS problem on a graph G. Figure 3.1 shows a corresponding UDCS problem on a graph G^*. G^* contains two copies of G. Both copies of a vertex i have upper bound u_i, the same upper bound as in G. In addition the two copies of i are joined by u_i-l_i paths of length three. Each of the $2(u_i-l_i)$ intermediate vertices on these paths has degree two in G^* and has upper bound one.

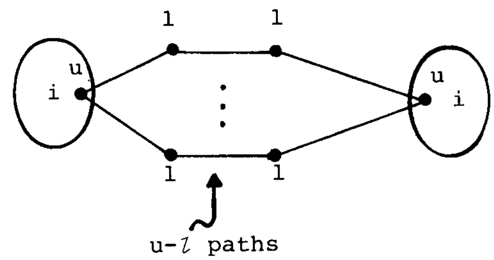


Figure 3.1

Graph G^*

It is easy to see that a DCS H on G has a corresponding complete [2] UDCS H^* on G^*. Conversely a complete UDCS H^* on G^* induces a DCS H on G. H need not have maximum cardinality, but it leads to the solution to our problem. Here is the complete algorithm for the DCS problem.

Step 1. Construct the UDCS problem G^* from G. Find a maximum cardinality UDCS H^*. Assume H^* is complete (else the DCS problem is infeasible). Let H be the DCS on G induced by H^*.

Step 2. Run the maximum cardinality UDCS algorithm on G, using H as the initial solution.

For Step 2, recall that the maximum cardinality matching algorithm of [MV] can be started with any initial matching. Hence the same is true of our UDCS algorithm, as required in Step 2. Next recall that the UDCS algorithm works by augmenting paths. Hence no degree of a vertex is ever decreased. So the algorithm halts with a subgraph that satisfies all upper and lower bounds u_i, l_i. It has maximum cardinality among all subgraphs that satisfy the upper bounds. Hence it is a maximum cardinality UDCS.

Theorem 3.2. A maximum cardinality DCS can be found in $O(\sqrt{\Sigma u_i}E)$ time and $O(E)$ space. ▪

4. Maximum Cardinality, Arbitrary Capacity Problems

This section presents algorithms for the maximum cardinality UDCS and DCS problems, when edges e have arbitrary integral multiplicities u_e. The algorithms run in $O(VE\log V)$ time and $O(E)$ space.

[2] In a *complete* UDCS, every upper degree bound u_i holds with equality.

We begin with UDCS. Define the graph G' as usual using the substitutes of Figure 2.2. An edge ij in G corresponds to u_{ij} distinct edges in G', each joining an external vertex in i's substitute to one in j's. As in Section 3, every internal vertex of G' is matched.

Our approach is to simulate the cardinality matching algorithm on G'. The Cardinality Matching Reduction Principle bounds the length of an *sap* and gives this estimate:

Lemma 4.1. At most $3V+1$ phases are needed to find a maximum matching on G'. ■

The Lemma implies that for our time bound it suffices to implement a phase on G' in $O(E \log V)$ time. To do this, as in Section 3 it is convenient to work with graphs G'_k and G_k. Both are derived from G' and the current matching by using sparse substitutes. In G'_k a substitute has d_i internal edges; in G_k, it has two internal edges. Furthermore, suppose ij is an edge of G, having m matched copies and u unmatched copies in the current matching on G'. (So $\mu_{ij} = m + u$.) Then G'_k contains m matched copies and u unmatched copies of ij; G_k contains $min(2,m)$ matched copies and $min(2,u)$ unmatched copies.

Lemma 3.2 still applies to graph G'_k. Hence it suffices to find an *sap* set on G'_k. We will show that this can be done on the smaller graph G_k. First it is convenient to extend the definition of "similar" vertices. In the current context we say that two external vertices in the same substitute of G_k or G'_k are *similar* if they are on copies of the same edge of G, *and* both copies are matched or both are unmatched. Substitute vertices that are matched to external vertices are handled analogously. The rest of the definition of similarity is unchanged.

It is easy to see that Lemma 3.3 remains valid for G'_k and G_k. In particular, the definition of G_k allows the Cardinality Matching Reduction Principle to apply in the proof of Lemma 3.3(i).

Lemma 3.3 allows us to use G_k to calculate levels and blossoms in G'_k. Now we must show how to actually find the *sap*'s. The algorithm is based on the following property of blossoms. Let an *sap* have length $2s+1$.

Lemma 4.2. In G'_k, let B be a maximal blossom of tenacity $t < 2s+1$, with base vertex b. Then no other vertex is similar to b.

Proof Sketch. Follows from Lemma 3.3 (ii). ■

The Lemma and (Cardinality Blossom Property(ii)) implies that at most one *sap* passes through any copy of any edge with a vertex in a blossom. So "most" *sap*'s do not pass through any blossoms. This allows us to use the fast techniques for network flows for most *sap*'s.

To carry out this approach we work with two graphs. The first is a multigraph M_k that is essentially G'_k. Let ij be an edge of G that has m matched copies in G'_k and u unmatched copies. Then M_k has a matched copy of ij with multiplicity m and an unmatched copy with multiplicity u. (In our data structure for multigraphs we store each edge and its an integral multiplicity. Thus M_k has size $O(E)$.) M_k is used for *sap*'s of multiplicity greater than one. It is processed with the dynamic tree data structure of Sleator and Tarjan [ST].

The second graph, U, consists of edges with unit multiplicity. It is used for *sap*'s that have multiplicity one. In particular it handles *sap*'s that involve blossoms.

Now we give the algorithm for a phase. It follows the outline of the algorithm of [MV] given in Section 3. Let an *sap* have length $2s+1$.

Step 1. Use the graph G_k to calculate all levels $e(v)$ and $o(v)$ that are at most $s+1$. Construct all blossoms that have tenacity $2s+1$.

Step 2. Construct the multigraph M_k. Initialize the graph U to be empty. Transfer from M_k to U all blossoms and all edges of multiplicity one. *(Comment:* M_k has no edges or blossoms of tenacity less than $2s+1$. It may have edges or blossoms of tenacity $2s+1$.)

Step 3. Repeat the following steps for every edge vw of tenacity $2s+1$. When no more edges vw remain, go to Step 4.

Step 3a. Use the method of dynamic trees [ST] to find a path P_v from v to a free vertex, and also a path P_w from w to a free vertex. P_v and P_w are paths defining $e(v)$ and $e(w)$ if vw is unmatched, or $o(v)$ and $o(w)$ otherwise. Let $P_v(P_w)$ end in the substitute for vertex $i(j)$ of G. Let $\delta_i(\delta_j)$ be the largest possible increase in the degree of $i(j)$ in the current UDCS.

Step 3b. If P_v and P_w are disjoint then let $\mu = \min\{\mu_e, \delta_i, \delta_j \mid e$ is an edge on P_v, P_w, or $vw\}$. Augment μ copies of the path P_v, vw, P_w. Go to Step 3d.

Step 3c. Otherwise P_v and P_w are not disjoint. By the method of dynamic trees they join at a vertex j, i.e., P_v consists of a path from v to j, P_{vj}, followed by a path from j to a free vertex, P_j. Similarly, P_w consists of P_{wj} and P_j. (Possibly P_j is a single vertex.)

Let

$\mu = \min\{\mu_e, \left\lfloor \frac{\mu_f}{2} \right\rfloor, \left\lfloor \frac{\delta_i}{2} \right\rfloor \mid e$ is an edge on P_{vj}, P_{wj}, or vw; f is an edge on $P_j\}$. *(Comment:* $\mu = 0$ if the paths form a blossom of tenacity $2s+1$.) Augment μ copies of P_{vj}, vw, P_{wj}; augment 2μ copies of P_j.

Step 3d. Delete all augmented edge from M_k. Transfer any new blossom (of tenacity $2s+1$) and any edge with new multiplicity one, to U. Continue with Step 3.

Step 4. Transfer the remaining edges of M_k to U, giving them multiplicity one. However make two copies of the internal edge of every substitute.

Step 5. Find an *sap* set on U, using the procedure of Section 3, and augment along the *saps*. Stop.

Theorem 4.1. A maximum cardinality UDCS on a multigraph on a multigraph can be found in $O(VE \log V)$ time and $O(E)$ space.

Proof Sketch. For the time bound, note that Step 3 is implemented with dynamic trees in essentially the same way as the algorithm for blocking flows [ST]. Also recall that the dynamic tree data structure finds deepest common ancestors as fast as its other primitive operations. This allows the join j of P_v and P_w to be found efficiently in Step 3c. ■

The DCS problem is solved in the same way as in Section 3.

Theorem 4.2. A maximum cardinality DCS on a multigraph can be found in $O(VE \log V)$ time and $O(E)$ space. ■

5. Maximum Weight, Unit Capacity Problems

This section presents algorithms for the maximum weight UDCS and DCS problems that use $O((\Sigma u_i)\min(E \log V, V^2))$ time and $O(E)$ space. We start with the UDCS problem.

Again our approach is to simulate the weighted matching algorithm on G', the graph with vertex substitutes. Recall how this algorithm works [E65, G, GMG]: A *map* is a maximum weight augmenting path. The algorithm repeatly finds a *map* and uses it to augment the matching. This implies that the algorithm finds a maximum weight k-matching,[3] for $k = 1, 2, \cdots$

Consider a vertex substitute S in G', as in Figure 2.2. For weighted problems all edges of a substitute are assigned weight W, the largest edge weight in G. We can assume that all internal vertices of S are matched, as in Figure 2.2. (Clearly this gives a maximum weight k-matching for some k).

Now we give the principle for sparse substitutes in weighted problems. Let P be a *map*. Recall from Section 3 that P traverses a pair of edges in S in one of two directions.

Weighted Matching Reduction Principle. There is a *map* that traverses at most two pairs of edges from a given substitute, one in each direction.

Proof. Suppose P traverses two pairs in the same direction. So P has the form $\ldots, uv, vw, \ldots, xy, yz, \ldots$, where vertices v and y are in the substitute and edges uv and xy are matched. Let P' be P with the subpath from v to z replaced by edge vz. We claim P' has weight at least that of P, i.e., $w(P)-w(P') \geq 0$. Observe that edges vw, vz, yw and yz all have the same weight. Hence $w(P)-w(P')$ is the weight of the alternating cycle formed by edge yw and the portion of P from w to y. This weight is nonpositive, since any alternating cycle in a maximum weight k-matching has nonpositive weight. We conclude P' is a *map*, as desired. ■

Suppose G' has a maximum weight k-matching. Define G_k by using a sparse substitute for each vertex, as in Figure 2.3. All edges in the sparse substitute have weight W. The Reduction Principle imples this result:

Lemma 5.1. A *map* in G_k corresponds to a *map* in G' containing the same edges of G. ■

The Lemma justifies our approach of using G_k to find a *map*. Unfortunately we cannot merely input G_k to the matching algorithm and find a *map*. The matching algorithm of [E65] and its efficient implementations [G, GMG] are primal-dual algorithms [D]: A set of dual variables is maintained throughout the algorithm. We must show how to construct dual variables on G_k from those of G_{k-1}.

It is convenient to describe the search routine of the matching algorithm in terms of its input and output. Both of these are in the form of a *search graph*. This is a graph with a maximum weight k-matching. The graph has a collection of disjoint blossoms (see Section 2). Each vertex i has a dual variable y_i and each blossom B has a dual variable $z_B \geq 0$. In addition these properties hold:

Search Graph Properties

(i) The free vertices have the smallest y_i-value, i.e., if i is free then $y_i = \min\{y_j \mid j \in V\}$.

(ii) For every edge ij,

$$y_i + y_j + \sum_{i,j \in B} z_B \geq w_{ij}.$$

Note the summation is over all blossoms B that contain both vertices i and j. Edge ij need not be in the *subgraph* B (see Section 2).

(iii) All edges that are matched or in a blossom subgraph are *tight*, i.e., the inequality of (ii) holds with equality.

We will use some simple properties of the search algorithm: Throughout its execution, the algorithm maintains a search graph structure on the given graph. It forms blossoms by combining existing blossoms B_i, $1 \leq i \leq 2k+1$, into a new blossom B. These properties hold:

Search Algorithm Properties

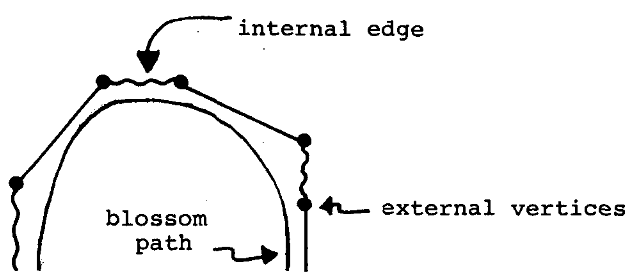
(i) When a blossom B is formed, $z_B = 0$. If vertex c is matched to the base b of B, then c is incident to an unmatched tight edge.

(ii) No dual variable is changed until every blossom of the search graph is maximal. $z_B > 0$ only if B is a blossom in a search graph immediately before a dual variable is changed.

(iii) Suppose the algorithm finds a *map* P that contains a vertex of a blossom B. The portion of P in B is an alternating path that starts with a matched edge of B, goes to the base of B, and contains only edges of B and its subblossoms.

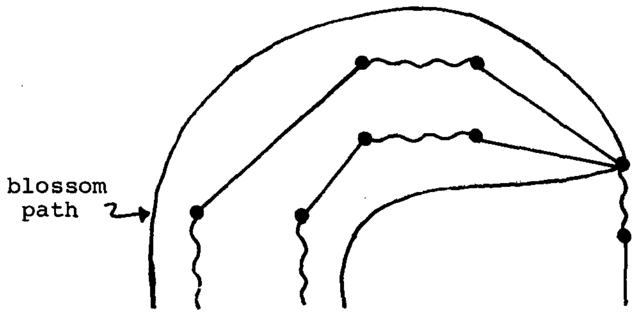
Now we examine graph G_k. A *primary blossom* for a substitute S is the smallest blossom containing an edge of S. The following result is the analog of Lemma 3.3(ii). It follows from Search Algorithm Properties (i)-(ii).

Lemma 5.2. Without loss of generality, a vertex of an internal edge of a substitute is not the base of a primary blossom. ■

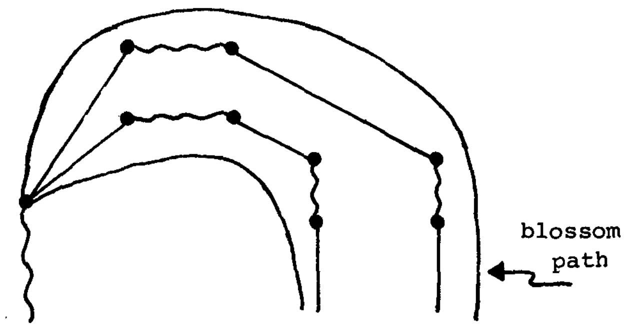


(a) 1,1 blossom

[3] A *k-matching* has exactly k edges.



(b) 2,0 blossom



(c) 0,2 blossom

Figure 5.1

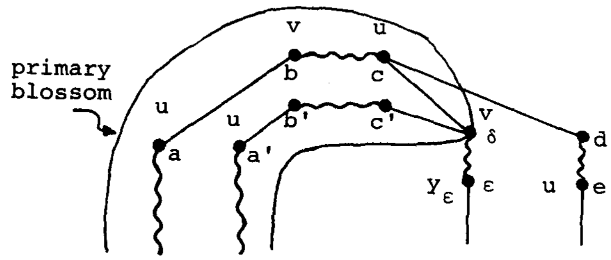
Topologies for primary blossoms.

Now it is easy to see that there are three types of primary blossoms, shown in Figure 5.1. Figure 5.1(a) shows the blossom when the base is not in the substitute, and Figure 5.1(b)-(c) show when it is. Observe that these are the only possibilities, since the Lemma shows the base is not on an internal edge. (Also, the base is not on an external vertex on the right of the substitute, since a base is on at least two unmatched edges. A minor variant of Figure 5.1(a) and 5.1(c) is when the external vertex on the left is free.)

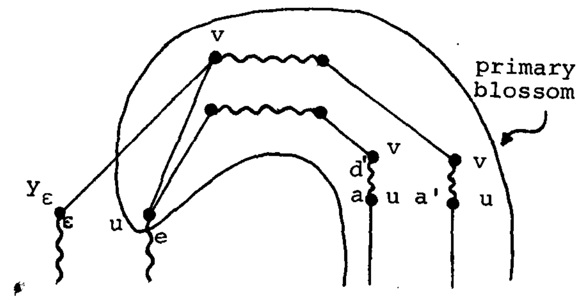
A primary blossom is an l, r *blossom* (with respect to a given substitute) if it contains l edges of G on the left of the substitute and r edges on the right. As shown in Figure 5.1 the three types of blossoms are 1,1; 2,0 and 0,2.

Now recall G_k is the substitute graph before the k^{th} augment, and G_{k+1} is the substitute graph constructed after this augment. We define a search graph structure on G_{k+1} by using essentially the same, structure as G_k. The blossoms are the same in both graphs, except that when the *map* goes through a substitute the new substitute edges of G_{k+1} replace the old ones of G_k. The dual variables y_i, z_B are the same with one exception: This is $y_{d'}$, the dual variable for a new vertex d' in the substitute. d' is created when an edge of G becomes unmatched, thereby moving from the left side of the substitute to the right, thus spawning a new matched edge and new vertex d'.

We will illustrate the details of our policy by considering one case: when the primary blossom B_0 is 2,0 and the *map* enters B_0 from a vertex d, as shown in Figure 5.2(a). In this figure B_0's base is δ, which is matched to external vertex ϵ. Vertex d is matched to external vertex e. Dual variables are easily calculated, and are indicated next to the vertices (e.g., $y_b = v$).



(a) 2,0 blossom in G_k.



(b) 0,2 blossom in G_{k+1}.

Figure 5.2

Augment of a 2,0 blossom.

Consider the *map* P. Search Algorithm Property (iii) shows P goes from c, through B_0, to δ. It is easy to see that P has the form $c, b, a, \ldots, a', b', c', \delta, \epsilon$. So G_{k+1} is as shown in Figure 5.2(b): a and a' are on unmatched external edges of B_0 and e is the new base; ϵ is on a matched external edge and is still joined to B_0. It is easy to see that all edges of Figure 5.2(b) are tight. So a blossom is defined in G_{k+1}.

Other cases in the analysis are similar. We conclude that the structure defined for G_{k+1} is a search graph.

Theorem 5.1. A maximum weight UDCS can be found in $O((\sum u_i)\min(E \log V, V^2))$ time and $O(E)$ space.

Proof Sketch. For the time bound, the search routine of [GMG] runs in $O(E \log V)$ time, and gives our first time bound. The search routine of [G] runs in $O(V^2)$ time. Minor modifications in the data structure make this $O(V^2)$ on our graph G_k, giving our second time bound. ∎

We solve the weighted DCS problem using the graph G^* of Figure 3.1.

Theorem 5.2. A maximum weight DCS can be found in $O((\sum u_i)\min(E \log V, V^2))$ time and $O(E)$ space. ∎

6. Shortest Paths

This section sketches an algorithm for the all-pairs shortest path problem on an undirected graph. The run time is $O(V \min (E \log V, V^2))$.

The algorithm is based on Lawler's reduction of the single-pair shortest path problem to DCS [L, pp. 220-222]. The reduction resembles the bidirected graph construction of Figure 2.1: Given an undirected graph with edge lengths, G. Let G^* be G where in addition, each vertex has a self loop of length 0. Consider a DCS problem on G^* where each vertex i has $l_i = u_i = 2$. This problem is closely related to any shortest path problem on G: If s is the source and t is the sink, the shortest $s-t$ path corresponds to the DCS where the bounds for s and t are changed to one. This mottivates the following algorithm:

Step 1. Given G, form G^* adn find a minimum weight DCS. *(Comment:* A minimum weight DCS consists of all the self-loops, since G has no negative cycles. Our algorithm finds this solution and more importantly, it constructs the corresponding dual variables and blossoms.)

Step 2. Repeat the following steps for each vertex.

Step 2a. Use G^* to form G', a copy of G^* with a new vertex S and edge Ss. Let M be the largest dual variable y_i in G^*. Let Ss have length $2M+1$ and let $y_S = M$. Let $l_S = u_S = 1$.

Step 2b. Use the DCS algorithm to search for an augmenting path from S. *(Comment:* The search halts unsuccessfully, since there is no DCS. However it computes dual variables y_i.)

Step 2c. Output each vertex t at distance $y_S + y_t - (2M+1)$ from s.

Theorem 6.1 The all-pairs shortest path problem on an undirected graph with no negative cycles can be solved in $O(V \min (E \log V, V^2))$ time and $O(E)$ space.

Proof Sketch. Correctness can be seen by noting that Step 2b simulates the DCS algorithm when G^* is modified to make t the sink. For the timing, Step 1 runs in the desired time. Each execution of Step 2b uses $O(min\ (E \log V, V^2))$ time. So Step 2 is also within the desired time bound. ∎

REFERENCES

[A] R.P. Anstee, "An algorithmic proof of Tutte's f-factor theorem", Res. Rept. CORR 81-28, Dept. of Combinatorics and Optimization, Univ. of Waterloo, Waterloo, Ontario, 1981.

[Berg] C. Berge, *Graphs and Hypergraphs*, North-Holland, Amsterdam, 1973.

[Bern] R. Bernstein, "Shortest paths in undirected graphs", M.S. Thesis, Stevens Inst. Technology, 1982.

[D] G.B. Dantzig, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, NJ, 1963.

[E65] Edmonds, J., "Maximum matching and a polyhedron with 0,1-vertices," *J. Res. Nat. Bur. Standards 69B* (1965), 125-130.

[E67] J. Edmonds, "An introduction to matching," Notes of Engineering Summer Conference, Univ. of Michigan, Ann Arbor, 1967.

[EJ70] J. Edmonds and E.L. Johnson, "Matching: A well-solved class of integer linear programs," *Proc. Calgray Int. Conf. on Combinatorial Structures and Their Applications*, Gordon and Breach, New York, 1970, pp. 89-92.

[EJ73] J. Edmonds and E.L. Johnson, "Matching, Euler tours and the Chinese postman," *Math. Programming* 5, 1973, pp. 88-124.

[EK] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *J. ACM 19*, 2, 1972, pp. 248-264.

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

[G83] H. Gabow, "An efficient reduction technique for degree-constrained subgraph and bidirectied network flow problems," Tech. Rept. CU-CS-243-83, University of Colorado, Boulder, Colorado, 1983.

[G] H. Gabow, "An efficient implementation of Edmonds' algorithm for maximum weight matching on graphs," Tech. Rept. CU-CS-075-75, University of Colorado, Boulder, Colorado, 1975.

[Gol] A.J. Goldman, "Optimal matchings and degree-constrained subgraphs," *J. Res. National Bureau of Standards 68B*, 1, 1964, pp. 27-29.

[GMG] Z. Galil, S. Micali, H. Gabow, "Priority queues with variable priority and an $O(E\ V \log V)$ algorithm for finding a maximal weighted matching in general graphs," *Proc. 23rd Annual Symp. on Foundation of Comp. Sci.*, 1982, pp. 225-261.

[GS] H.N. Gabow and M. Stallman, "Scheduling multitask jobs with deadlines on one processor," abstract.

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

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

[MV] S. Micali and V.V. Vazirani, "An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs," *Proc. 21st Annual Symp. on Found. of Comp. Sci.*, 1980, pp. 17-27.

[Sh] Y. Shiloach, "Another look at the degree constrained subgraph problem," *Inf. Proc. Letters 12*, 2, 1981, pp. 89-92.

[Sl] D.D.K. Sleator, "An $O(nm \log n)$ algorithm for maximum network flow," Ph.D. Diss., Tech. Rept. STAN-CS-80-831, Stanford Univ., Stanford, CA, 1980.

[ST] D. Sleator and R.E. Tarjan, "A data structure for dynamic trees," *Proc. 13th Annual ACM Symp. on Th. of Computing*, Milwaukee, Wisc., 1981, pp. 114-122.

[T] R.E. Tarjan, "Shortest paths," Ch. 17 in unpublished manuscript, 1982.

[U] R.J. Urquhart, "Degree-constrained subgraphs of linear graphs," Ph.D. Diss., University of Michigan, 1967.

[W] L.J. White, "A parametric study of matchings and covering in weighted graphs," Ph.D. Diss., Systems Eng. Lab., Dept. of Electrical Eng., University of Michigan, Ann Arbor, 1967.