Fundamenta Informaticae 154 (2017) 109–130
DOI 10.3233/FI-2017-1555
IOS Press

The Weighted Matching Approach to Maximum Cardinality Matching

Harold N. Gabow*
Department of Computer Science, University of Colorado at Boulder
Boulder, Colorado 80309-0430, USA
hal@cs.colorado.edu

Abstract. Several papers have achieved time $O(\sqrt{n}m)$ for cardinality matching, starting from first principles. This results in a long derivation. We simplify the task by employing well-known concepts for maximum weight matching. We use Edmonds' algorithm to derive the structure of shortest augmenting paths. We extend this to a complete algorithm for maximum cardinality matching in time $O(\sqrt{n}m)$.

1. Introduction

The most efficient known algorithms for cardinality matching on nondense graphs achieve time $O(\sqrt{n}m)$. The best known of these algorithms are not readily accessible: Micali and Vazirani were the first to present such an algorithm [1] but proving it correct has met difficulties [2, 3]. Gabow and Tarjan present a complete development but only at the end of a long paper with a different goal, a scaling algorithm for weighted matching [4]. Similarly Goldberg and Karzanov develop a new framework for flow and matching problems ("skew symmetric matchings") and again the cardinality matching algorithm requires mastery of this framework. Each of these papers tackles a difficult subject from first principles.

This paper presents an accessible matching algorithm with time bound $O(\sqrt{n}m)$. We simplify the task by taking advantage of the well-established theory for maximum weight matching. We include

*Address for correspondence: Department of Computer Science, University of Colorado at Boulder, Boulder, Colorado 80309-0430, USA

a review of Edmonds' weighted matching algorithm [5] but still it is helpful to be familiar with the algorithm. Complete treatments are in various texts e.g., [6, 7, 8, 9].

At first glance maximum weight matching seems to offer little insight to the problem. However the dual variables for weighted matching reveal structure that can either be used directly or must be rederived in a presentation from first principles. More importantly we show that a judicious choice of edge weights maps a large piece of the puzzle into simple properties of weighted matching.

The use of weighted matching for cardinality matching was introduced by Gabow and Tarjan [4]. Their cardinality matching algorithm uses a relaxation of the linear program duals that is helpful for scaling. Our algorithm is based on the standard LP duals. In that respect it differs from [4] at the structural level.

To make our presentation complete, at the data structure level we use the depth-first search procedure of [4], especially because of its simplicity. We also give a more detailed analysis of the procedure's correctness than [4], in an appendix. An alternative for this part of our algorithm would be to use the double depth-first search procedure of Micali-Vazirani [1]. To further demonstrate the power of the weighted matching approach, we use Edmonds' algorithm to deduce the key structure for double depth-first search, in another appendix. Establishing this structure is the central issue in proving the entire algorithm of [1] is correct.

The paper presents our algorithm in a top-down fashion. Section 2 gives the overall approach, using two Phases. Section 3 uses Edmonds' algorithm to implement Phase 1. Section 4 restates the Gabow-Tarjan algorithm for Phase 2 [4], with a complete correctness proof in Appendix A. Section 5 gives details of the data structure that achieve our desired time bound. Appendix B uses Edmonds' algorithm to prove existence of the starting edge for the double depth-first search of the Micali-Vazirani algorithm [1].

Terminology The given graph is always denoted as $G = (V, E)$. An edge in a minor of G is denoted by its preimage, i.e., xy for $x, y \in V$. For a set of vertices $S \subseteq V$, $\gamma(S)$ denotes the set of edges with both ends in S. Furthermore for $F \subseteq E$, $\gamma(S, F)$ denotes $\gamma(S) \cap F$. We often write $\gamma(S, P)$ where $S \subseteq V$ and P is a path; this notation treats path P as a set of edges.

A *matching* M on a graph is a set of vertex-disjoint edges. M has *maximum cardinality* if it has the greatest possible number of edges. Let each edge e have a real-valued *weight* $w(e)$. For a set of edges S define $w(S) = \sum_{e \in S} w(e)$. M has *maximum weight* if $w(M)$ is maximum. (Such M needn't have maximum cardinality.)

For an edge $xy \in M$ we say x and y are *mates*. A vertex is *free* if it is not on any matched edge. An *alternating path* is a vertex-simple path whose edges are alternately matched and unmatched. (Paths of 0 or 1 edges are considered alternating.) An *augmenting path P* is an alternating path joining two distinct free vertices. To *augment the matching along* P means to enlarge the matching M to $M \oplus P = (M \cup P) - (M \cap P)$. This gives a matching with one more edge.

2. The high-level algorithm

A *shortest augmenting path*, or *sap*, is an augmenting path of shortest length possible. Hopcroft and Karp [10] and independently Karzanov [11] presented an efficient approach to finding a maximum

cardinality matching: Repeatedly find a maximal collection of vertex-disjoint *saps* and augment the matching with them. This algorithm repeats only $O(\sqrt{n})$ times [10, 11]. Fig.1 gives our implementation of this approach.

```
M ← ∅
loop
    /* Phase 1 */
    for every edge e do w(e) ← if e ∈ M then 2 else 0
    for every vertex v do y(v) ← 1
    execute a search of Edmonds' algorithm to find a maximum weight augmenting path
    if no augmenting path is found then halt /* M has maximum cardinality */
    form the graph H of permissible edges

    /* Phase 2 */
    P ← a maximal set of vertex-disjoint augmenting paths in H
    for every path P ∈ P do
        Q ← the preimage of P in G
        augment M by Q
```

Figure 1. The high-level cardinality matching algorithm.

For Phase 1 we will show that Edmonds' algorithm halts having found an *sap* (unless there is no augmenting path). Furthermore Edmonds' algorithm provides the information needed to construct the graph H. H is a minor of G whose augmenting paths correspond (essentially 1-to-1) to the *saps* of G (see Corollary 3.3).

In Phase 2 we complete the algorithm using one of several known procedures [4, 1] to find the maximal set of augmenting paths $\mathcal{P}$ in H. Each path of $\mathcal{P}$ has a preimage in G specified by Edmonds' algorithm, and these preimages form the maximal set of *saps* required by the Hopcroft-Karp/Karzanov approach.

In every iteration of our algorithm Phases 1 and 2 each use $O(m)$ time. So the entire algorithm will use $O(\sqrt{n}m)$ time (see Theorem 5.1).

Note that our algorithm has the same structure as the $O(\sqrt{n}m)$-time matching algorithms of Hopcroft and Karp [10] and Karzanov [11] for the special case of bipartite graphs. In this case H is a subgraph of G, and a straightforward breadth-first search can be used to construct it. For general graphs G we allow H to be a minor of G. It may not even be clear why the desired minor H exists. But Edmonds' algorithm will provide the proof.

3. Phase 1 via Edmonds' algorithm

Section 3.1 states the simplified version of Edmonds' algorithm that we use. It also gives a brief review of blossoms. Section 3.2 characterizes the augmenting paths that are found. Section 3.3 gives the remaining details for Phase 1 and proves the graph H has the key property mentioned above.

```
make every free vertex the (outer) root of an S̄-tree
/* general algorithm also makes free blossoms into roots */
loop
    if ∃ tight edge e = xy with x outer, B_x ≠ B_y then
        if y ∉ V(S) then /* grow step */
            add xy, yy' to S, for yy' ∈ M
            /* general algorithm adds blossom B_y & its matched blossom */
        else if y is outer then
            if x and y are in the same search tree then /* blossom step */
                form a new blossom by merging all blossoms in the fundamental cycle of e in S̄
            else /* xy plus the S̄-paths to x and y form an augmenting path */
                return /* general algorithm proceeds to augment the matching */
    /* general algorithm may expand an inner blossom */
    else /* dual adjustment step */
        δ = min{y(e) − w(e) : e = uv with u outer, v ∉ V(S)} ∪
                {(y(e) − w(e))/2 : e = uv with u, v in distinct outer blossoms}
        /* general algorithm includes set for duals of inner blossoms */
        if δ = ∞ then return /* M has maximum cardinality */
        for every vertex v ∈ V(S) do
            if v is inner then y(v) ← y(v) + δ else y(v) ← y(v) − δ
        /* general algorithm changes duals of blossoms. in particular
            z(B) ← z(B) + 2δ for every maximal outer blossom B. */
```

Figure 2. Simplified search of Edmonds' algorithm.

3.1. Edmonds' weighted matching algorithm

Fig.2 gives pseudocode for a search of Edmonds' weighted matching algorithm [5]. The figure omits code that is never executed in the special case of Edmonds' algorithm that we use. Omissions are indicated by comments. (Basically the omissions result from the fact that there are no blossoms at the start of a search.) We will explain the code of the figure and then give a precise statement of the assumptions that justify our simplification of [5] (see (A) below).

Edmonds' algorithm can be used to find various "optimum" weighted matchings, e.g., a maximum weight matching, a maximum weight matching of maximum cardinality, etc. We use the variant of Edmonds' algorithm for the latter, i.e., it finds a matching that has the greatest possible weight of all maximum cardinality matchings.

In Fig.2 M denotes the matching. B_x is the maximal blossom currently containing vertex x, see below. (In particular note that a blossom can be a singleton vertex, i.e., we may have $B_x = \{x\}$.)

The algorithm is illustrated in Fig.3. Free vertices are square and matched edges are heavy. The dashed edges of Fig.3(f) form an augmenting path. The numbers in Fig.3 are defined below.

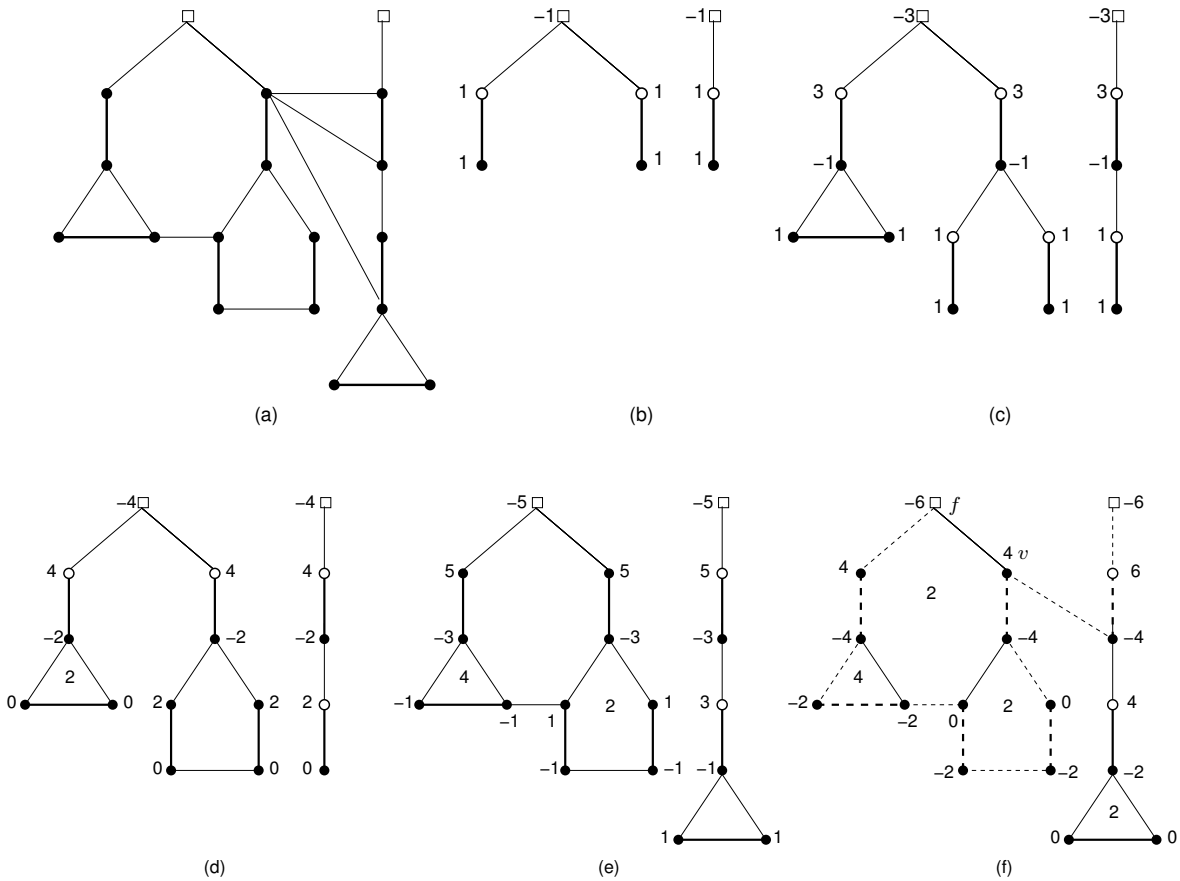

Figure 3. (a) Example graph for finding an *sap*. As in Fig.1 matched edges weigh 2, unmatched edges weigh 0, and every $y(v)$ is initially 1. (b)–(f) Search of Edmonds' algorithm. (b)–(e) show $\mathcal{S}$ with inner vertices drawn hollow, and the y and z dual variables. (f) shows the augmenting path as dashed edges.

The algorithm builds a "search structure" $\mathcal{S}$, a subgraph of G. (Fig.3(b)–(e) show the various $\mathcal{S}$ structures.) It also maintains $\overline{\mathcal{S}}$, the subgraph $\mathcal{S}$ with various sets (called "blossoms", defined below) contracted. $\overline{\mathcal{S}}$ is a forest. Its roots are the free vertices and contracted sets that each contain a free vertex. $\overline{\mathcal{S}}$ is an "alternating forest" [5]: Any path from a node to the root of its tree is alternating. A node of $\overline{\mathcal{S}}$ is *inner* (respectively *outer*) if its path starts with an unmatched (matched) edge. An inner node is always a vertex of G. An outer node can be a contracted set. Any vertex of V in such a set is also called *outer*.

The edges of $\mathcal{S}$ are all "tight", defined below with equation (1) when dual variables are discussed.

The algorithm adds new vertices and edges to $\mathcal{S}$ in a *grow step*, which adds an edge from an outer vertex to a new inner vertex y. It also adds the matched edge from y to its mate, which becomes a new outer vertex. (The mate exists, since otherwise y would be a free vertex already in $\mathcal{S}$.) Fig.3(b) shows the result of three grow steps.

Suppose the algorithm discovers an edge e joining two outer vertices. If e joins distinct trees of $\overline{\mathcal{S}}$ it completes an augmenting path. An *augment step* enlarges the matching. (In Phase 1 we delay the augment to Phase 2.) Fig.3(f) shows the algorithm's augmenting path as dashed edges.

If e joins nodes in the same tree of $\overline{\mathcal{S}}$ a *blossom step* is done: e is added to $\mathcal{S}$, and the fundamental cycle C of e in forest $\overline{\mathcal{S}}$ is contracted. The vertices of V belonging to contracted nodes on C form a *blossom*. This blossom is a new outer node. A blossom step is executed in each of Fig.3(c), (d), and (e).

To describe the last step, recall that the algorithm is based on Edmonds' formulation of weighted matching as a linear program [5]. Each vertex $v \in V$ has a dual value $y(v)$. Each nonsingleton blossom $B \subseteq V$ has a nonnegative dual value $z(B)$. The duals *dominate* an edge $uv \in E$ if

$$y(u) + y(v) + \sum_{u,v \in B} z(B) \geq w(uv), \tag{1}$$

and uv is *tight* if equality holds. The algorithm of [5] maintains the invariant that every edge is always dominated and every edge of $\mathcal{S} \cup M$ is always tight. (Our simplified version does not record z duals. But the last 2 lines of Fig.2 show how [5] maintains z duals, and z duals are shown in Fig.3.) We use the common convention that for $e = uv \in E$, $y(e)$ denotes $y(u) + y(v)$. Note that the test for tightness in Fig.2 has $B_x \neq B_y$ and so it does not require z values.

When no grow, blossom, or augment step can be done, the algorithm makes progress by executing a *dual adjustment step*. It modifies dual variables to make one or more unmatched edges tight. This enables one or more of the other steps to be performed. If $\delta = \infty$ no edge can be made tight, and the current matching has maximum cardinality.

In Fig.3 each matched (respectively unmatched) edge weighs 2 (0). Each vertex is labelled with its y-value. Each blossom is labelled with its z-value. The label is in the interior of the blossom, and only included when z is nonzero. For example the dual adjustment at the end of Fig.3(c) increases the z-value of the triangular blossom from 0 to 2. This dual increases to 4 after part (d), and does not change in the dual adjustment after part (e).

We conclude this section by stating the assumptions that allow our simplifications to the general weighted matching algorithm:

(A)
- The algorithm begins with a matching M and no nonsingleton blossoms.
- The algorithm begins with dual variables $y(v), v \in V$ that dominate every edge and are tight on every matched edge (there are no z variables).
- The algorithm does not need to track z values of blossoms (since they will not be used in a subsequent search).

Blossoms Recall that a blossom is a set of vertices that is either a singleton or a set formed in by merging blossoms in Fig.2.

Any blossom B has a *base vertex*: The base vertex of a singleton blossom $B = \{v\}$ is v. The base vertex of a blossom constructed in the blossom step of Fig.2 is defined as follows: The fundamental cycle of $e = xy$ in $\overline{\mathcal{S}}$ contains a unique node of minimum depth – the nearest common ancestor a of

x and y. The base vertex of the new blossom is the base vertex of a. (In Fig.3(b)–(f) the base of any blossom is the vertex closest to the root. For instance in part (e) the left subgraph is a blossom whose base vertex is the root.)

Note that the base vertex of an arbitrary blossom B is the unique vertex of B that is not matched to another vertex of B. It may be free (e.g. Fig.3(e)) or matched to a vertex not in B (e.g. Fig.3(c)).

Any blossom B has a natural representation as an ordered tree R_B. The root is a node corresponding to B. The leaves correspond to the vertices of V in B. Any interior node corresponds to a blossom B' formed in the blossom step of Fig.2. Let the fundamental cycle C of that step consist of blossoms C_i, $i = 0, \ldots, k$. Here C_0 contains the base vertex of B', and the indexing corresponds to the order of the blossoms in a traversal of cycle C (in either direction). The children of B' correspond to C_i, $i = 0, \ldots, k$, in that order. In addition R_B records the edge $c_i c_{i+1}$ that joins each child C_i to the next child C_{i+1} (taking $k + 1$ to be 0). These edges are alternately unmatched and matched.

For a matched edge, each end is the base vertex of its blossom. (Also, as clear from Fig.2, one of its ends is a singleton blossom, $C_i = \{c_i\}$. But we do not use this property.) The base vertex of B is also recorded in R_B, since it is not determined by a matched edge of C.

Note that the edges $c_i c_{i+1}$ of R_B all belong to $E(\mathcal{S})$ and are tight. Also R_B has $O(|B|)$ nodes. This follows since R_B has $|B|$ leaves, and every interior node has ≥ 3 children. (So there are $\leq (|B|-1)/2$ interior nodes.)

Any vertex $v \in B$ has an even-length alternating path $P(v, b) \subseteq E(\mathcal{S})$ that starts with the matched edge at v and ends at the base b. (The exception is $P(b, b)$, which has no edges.) $P(v, b)$ is specified recursively using R_B, as follows. Let $C_i, i = 0, \ldots, k$ be the children of the root B. Let v belong to child C_j. The path $P(v, b)$ passes through C_h for $h = j, j + 1, \ldots, k, 0$ if $c_j c_{j+1}$ is matched, else $h = j, j - 1, \ldots, 1, 0$ if $c_{j-1} c_j$ is matched. $P(v, b)$ consists of

(a) the edges $c_i c_{i+1}$ that join these children C_h

(b) a subpath in each C_h, specified recursively.

(Note that $P(v, b)$ traverses some recursive subpaths $P(v', b')$ in reverse order, from b' to v'. But the algorithm does not require this order.)

As an example, the augmenting path of Fig.3(f) traverses the blossom of Fig.3(e) on the path $P(v, f)$ of length 10.

The paths $P(v, b)$ are used (in the complete version of Edmonds' algorithm, as well as our algorithm in Phase 2) to augment the matching. Note that the order of edges in $P(v, b)$ is irrelevant for this operation, since we are simply changing matched edges to unmatched and vice versa. Also note that every edge of $P(v, b)$ is tight, since $P(v, b) \subseteq E(\mathcal{S})$. So augmenting keeps every matched edge tight.

3.2. Properties of Edmonds' algorithm

Implicit in Edmonds' algorithm is that it always finds a maximum weight augmenting path. This section proves this and characterizes the structure of all maximum weight augmenting paths. We make this assumption on the initialization:

(A′) The initial y function is constant, and no unmatched edge is tight.

A constant y is the usual initialization of Edmonds' algorithm, e.g., [7, 8, 9]. Having no unmatched tight edge means there is at least one dual adjustment – this assumption simplifies our notation. Observe that throughout the algorithm every free vertex f has the same dual value $y(f)$ (by (A′) and the dual adjustment step of Fig.2).

As usual in matched graphs we define the (incremental) weight of an alternating path P to be

$$w(P) = w(P - M) - w(P \cap M).$$

For an augmenting path, $w(P)$ is the change in the matching's weight when P is augmented. This change may be of arbitrary sign, even for a maximum weight augmenting path.

Lemma 3.1. At any point in Edmonds' algorithm, any augmenting path P and any free vertex f have $w(P) \leq 2y(f)$.

Proof:
We start by showing

$$w(P) \leq \sum_{uv \in P-M} \Big(y(u) + y(v) + \sum_{u,v \in B} z(B) \Big) - \sum_{uv \in P \cap M} \Big(y(u) + y(v) + \sum_{u,v \in B} z(B) \Big). \quad (2)$$

To get this upper bound start with the definition of $w(P)$ and replace each weight in the definition, $w(uv), uv \in P$, by the left-hand side of (1), i.e., $y(u) + y(v) + \sum_{u,v \in B} z(B)$. These replacements give the desired inequality since Edmonds' algorithm keeps every umatched edge dominated and every matched edge tight, as defined by (1).

Any interior vertex of P, say r, has $y(r)$ appearing in both sums of (2). Hence the y terms of (2) sum to exactly $2y(f)$. Thus it suffices to show the z terms have nonpositive sum.

Let B be any blossom and b its base. Every vertex of B except b has its mate contained in B. P is not contained in $\gamma(B)$ since P contains two free vertices. Consider a maximal length subpath S of $\gamma(B, P)$.[1] S is alternating, and by assumption contains at least one edge. There are two cases:

Case *S has b at one end*: Following edges starting from b shows S ends at a matched edge of $\gamma(B)$. So S has even length, and its edges make no net contribution of $z(B)$ terms in (2).

Case *Neither end of S is b*: The first and last edges of S are matched. Thus S makes a net contribution of $-z(B) \leq 0$ in (2).

We conclude the total contribution of z terms to the upper bound is ≤ 0, as desired. □

Call a blossom B *positive* if $z(B) > 0$. B is positive iff it was formed before the last dual adjustment.

The following corollary refers to the end of Edmonds' search – tightness refers to the final duals, and blossoms are those defined over the entire algorithm.

[1] The notation $\gamma(B, P) = \gamma(B) \cap P$ was defined in Section 1.

Corollary 3.2. An augmenting path P has maximum weight iff all its edges are tight and for every positive blossom B, $\gamma(B, P)$ is an even-length alternating path.

Remark: When $\gamma(B, P) \neq \emptyset$ even-length implies P contains the base vertex of B, since every other vertex of B has its mate in B.

Proof:
The if direction follows from the proof of the lemma. Specifically the proof implies that P achieves the upper bound of the lemma, i.e., $w(P) = 2y(f)$, if every edge of P is tight and P traverses positive blossoms as specified in the lemma.

Observe that the augmenting path A found in Fig.2 satisfies these sufficient conditions. (In fact A traverses *every* blossom as in the corollary.) Thus A is a maximum weight augmenting path.

This implies an augmenting path has maximum weight iff its weight is $w(A) = 2y(f)$. So again the proof of the lemma implies the conditions of the corollary must hold for the final dual variables and the positive blossoms of the algorithm. □

3.3. Phase 1

As in Fig.1 an edge weighs 2 if it is matched, else 0. So initializing all y values to 1 makes y constant, every matched edge tight, and every unmatched edge dominated but not tight. In every iteration of Fig.1 Edmonds' algorithm starts afresh with no blossoms. So the assumptions of (A) and (A′) hold.

Assuming the matching is not maximum cardinality, Edmonds' algorithm halts with a maximum weight augmenting path. Any augmenting path P of length $|P|$ has

$$|P| = -w(P) + 1. \tag{3}$$

So an augmenting path has maximum weight iff it is an *sap*.

The last step of Phase 1 constructs H, the graph whose augmenting paths correspond to the *saps* of G. H is formed from G by contracting every maximal positive blossom, and keeping only the tight edges that join distinct vertices.[2] As usual each edge of H records its preimage in E, allowing an augmenting path in H to be converted to its preimage in G. It is an easy matter to show that H has the key property claimed in Section 2:

Corollary 3.3. A set of edges P forms an augmenting path in H iff it is the image of an *sap* Q in G.

Proof:
For the if direction let Q be an *sap* in G. Consider any maximal positive blossom B with $\gamma(B, Q) \neq \emptyset$. Corollary 3.2 implies $\gamma(B, Q)$ is a path that starts with a matched edge and ends with an unmatched edge incident to the base vertex b of B. So Q either contains exactly 1 edge incident to B (if b is free), or 2 edges incident to B, one being the matched edge incident to b. In both cases Q's image P in H is an alternating path. This makes P an augmenting path in H.

For the only if direction let P be augmenting in H. Consider any contracted maximal blossom B on P. P contains an unmatched edge incident to some vertex $v \in B$. Letting b be the base vertex of

[2]Edmonds' search may end before exploring some tight edges. This does not affect the definition of H.

B, either b is free or P contains the matched edge incident to b. In both cases we add the $P(v,b)$ path through B. Doing this for every B on P yields an augmenting path Q in G. The definition of $P(v,b)$ paths shows all conditions of Corollary 3.2 are satisfied, so Q is an *sap*. □

Again it is an easy matter to show the overall correctness of the algorithm of Fig.1: Let $\mathcal{Q}$ be the family of preimages Q used in Fig.1 to augment the matching. Each $Q \in \mathcal{Q}$ is formed using the paths $P(v,b)$ as in the above proof. So Q is an *sap*. $\mathcal{Q}$ is a collection of vertex-disjoint paths, since $\mathcal{P}$ is. $\mathcal{Q}$ is a maximal collection, since $\mathcal{P}$ is. (An *sap* Q' vertex-disjoint from $\mathcal{Q}$ would have an image in H that is vertex-disjoint from $\mathcal{P}$, contradiction.)

4. Phase 2

We can find a maximal set of augmenting paths in H using the double depth-first search of Micali and Vazirani [1], or the algorithm of Goldberg and Karzanov [12], or the depth-first search of Gabow and Tarjan [4]. (The Micali-Vazirani search [1] requires an additional property of H, which is proved in Appendix B.) To make this paper complete Fig.4 restates the Gabow-Tarjan algorithm [4]. The *find_ap* algorithm is illustrated in Fig.5. This section discusses the idea of the algorithm and gives the high-level analysis, for both correctness and the linear time bound $O(m)$. Appendix A completes the analysis. The development is similar to [4] but includes more details.

Phase 2 finds $\mathcal{P}$ by executing procedure *find_ap_set* of Fig.4 on the graph H of Phase 1. So it is convenient use H to denote the input graph to *find_ap_set*. *find_ap_set* treats its input H as an arbitrary graph (i.e., the contractions of Phase 1 are irrelevant). *find_ap_set* will form its own search graph $\mathcal{S}$, its own blossoms, etc. – the analogous structures formed in Phase 1 are irrelevant.

Procedure *find_ap* (Fig. 4) implements a search of Edmonds' cardinality matching algorithm. The cardinality algorithm amounts to the search algorithm of Fig.2 with no dual variables; instead every edge is considered tight. *find_ap* executes the search in a depth-first fashion – this requires careful scheduling of blossom steps.

Now we review Fig. 4. The algorithm uses several types of paths. For $y, x \in V$, $\overline{\mathcal{S}}(B_y, B_x)$ denotes the $\overline{\mathcal{S}}$-path from B_y to B_x (assuming it exists). $\overline{\mathcal{S}}(B_y, B_x)$ is used in line 4 to find the u_i vertices. For instance in the blossom step of Fig.5(b) the algorithm has $x = 3, y = 5, \overline{\mathcal{S}}(B_y, B_x) = (5, 8, 4, 7, 3)$.

Line 2 enlarges $\mathcal{P}$ using a path $P(x)$. $P(x)$ exists for any outer vertex x. It is a naturally defined even-length alternating path in H from x to the search tree root f. Specifically $P(x)$ is formed by starting with $\overline{\mathcal{S}}(B_x, B_f)$, and traversing every blossom (of *find_ap*) using the appropriate $P(v,b)$ path. (Recall the $P(v,b)$ paths defined in Section 3.1.) For instance in Fig.5(c), $P(7)$ is formed starting with $\overline{\mathcal{S}}(B_7, B_1) = (3, 11, 1)$, and adding $P(7,3) = (7, 4, 8, 5, 3)$. In line 2 $yP(x)$ denotes the path formed by starting at y and proceeding along $P(x)$ (using edge yx).

Finally we discuss the test for blossom steps (line 3). We start by introducing a variant of previous notation, to be used in the rest of the paper: $\mathcal{S}^-$ denotes the subgraph of $\mathcal{S}$ consisting of the edges added in grow steps. Clearly $\mathcal{S}^-$ consists of trees that span $\mathcal{S}$. Also $\overline{\mathcal{S}}$ is a contraction of $\mathcal{S}^-$. We use $\mathcal{S}^-$ to state ancestry relationships, as in line 3. These relations essentially hold in $\overline{\mathcal{S}}$ but $\mathcal{S}^-$ has the

```
   procedure find_ap_set
   initialize S to an empty graph and P to an empty set
   for each vertex v ∈ V do b(v) ← v /* b(v) maintains the base vertex of B_v */
   for each free vertex f do
       if f ∉ V(P) then
           add f to S as the root of a new search tree
1          find_ap(f)
   return P

   procedure find_ap(x) /* x is an outer vertex */
   for each edge xy ∉ M do /* scan xy from x */
       if y ∉ V(S) then
           if y is free then /* y completes an augmenting path */
2              add xy to S and add path yP(x) to P
               terminate every currently executing recursive call to find_ap
           else /* grow step */
               add xy, yy' to S, where yy' ∈ M
               find_ap(y')
3      else if b(y) is an outer, proper descendant of b(x) in S⁻ then /* blossom step */
           /* equivalent test: b(y) became outer strictly after b(x) */
4          let u_i, i = 1, ..., k be the inner vertices in S̄(B_y, B_x), ordered so u_i precedes u_{i-1}
           for i ← 1 to k do
5              for every vertex v with b(v) ∈ {u_i, u_i'}, where u_i u_i' ∈ M do b(v) ← b(x)
               /* this adds B_{u_i} ∪ B_{u_i'} to the new outer blossom */
6          for i ← 1 to k do find_ap(u_i) /* process u_i in order of increasing depth */
7      return
```

Figure 4. Path-preserving depth-first search.

advantage of being more stable over time. Note that $\mathcal{S}^-$ is not alternating. (For instance in Fig.5(a) add a matched edge aa' with an unmatched edge from 11 to a.)

To understand line 3 note that base $b(v)$ is always an $\mathcal{S}^-$-ancestor of v. Thus in line 3 edge xy joins blossoms B_x and B_y, and a blossom step can be done for xy.

The idea for *find_ap* is to search for an augmenting path depth-first, making sure that all vertices currently being explored remain on the current search path. This is an obvious property of ordinary depth-first search. It is also desirable for finding disjoint augmenting paths: When we find an augmenting path and delete it from further consideration, no problems will be created by partially explored vertices remaining in the graph – these vertices all get deleted.

To achieve this property blossom steps are delayed. For instance in Fig.5 $mate(2)$ is the 10th vertex to become outer, not the third, delayed by the test of line 3. Also the blossom step explores the new outer vertices u_i in order of decreasing $P(u_i)$ length. (Vertex 7 is explored before vertex 8.)

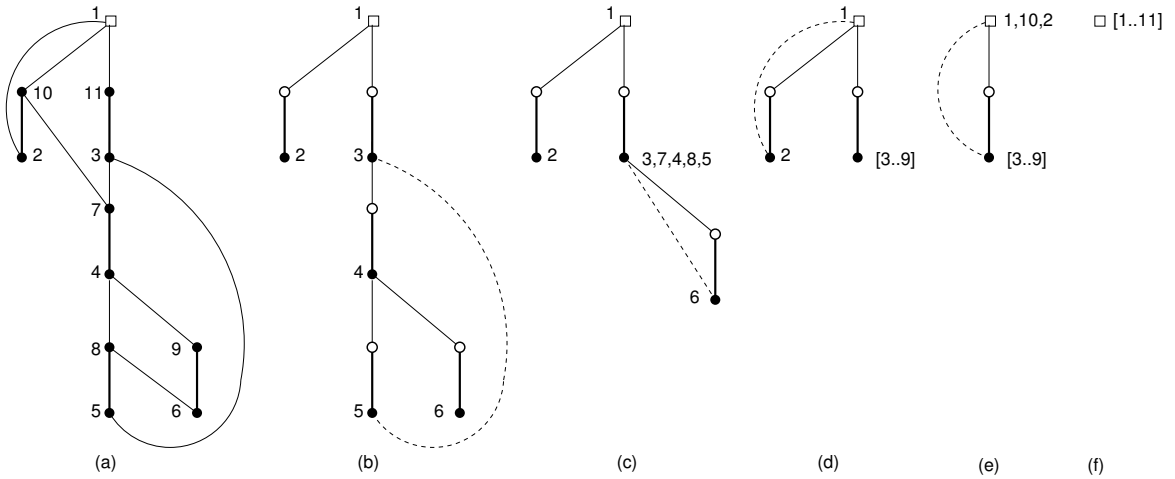

Figure 5. (a) Example graph for *find_ap*. Vertices are numbered in the order they become outer. (b)–(e) $\overline{\mathcal{S}}$ right before the dashed edge triggers a blossom step. Only outer vertices are numbered. $[i..j]$ denotes the set of consecutive integers $\{i, \ldots, j\}$. The blossom step of (b) implicitly defines $P(8) = (8, 5, 3, 11, 1), P(7) = (7, 4, P(8)) = (7, 4, 8, 5, 3, 11, 1)$. (f) Final blossom. $P(11) = (11, 3, 5, 8, 4, 7, 10, 2, 1)$.

It is easy to see that *find_ap* correctly implements the grow and blossom steps of Edmonds' cardinality matching algorithm. (Ignore the "equivalent test" after line 3 for now.) Specifically $\mathcal{S}, \overline{\mathcal{S}}, b(x)$, and $P(x)$ correspond to their definitions. However it is not immediately clear that *find_ap* does every possible blossom step. This is proved in Appendix A (Lemma A.2).

We close this section with an overview of proof that *find_ap_set* is correct. Details are in Appendix A. Correctness means that *find_ap_set* halts with $\mathcal{P}$ a maximal set of vertex-disjoint augmenting paths. The above remarks imply that each path of $\mathcal{P}$ is augmenting. So the issue is to prove maximality – no augmenting path of H is vertex-disjoint from $\mathcal{P}$. The proof is in three steps.

We first show the depth-first search path contains all the vertices that need to be further explored. We use the following definition to make this precise.

At any point in the execution of *find_ap_set* call an outer vertex x *completely scanned* if *find_ap*(x) has returned in line 7. The other possibility for an outer x – x is not completely scanned – holds when either *find_ap*(x) has not been called, or it is currently executing, or it was terminated (after line 2) because an augmenting path was discovered.

The first step of the proof establishes (P1):

(P1) When *find_ap_set* halts, every outer vertex that has not been completely scanned is on a path of $\mathcal{P}$.

(P1) can be established by proving an invariant, that the outer vertices not in $\mathcal{P}$ and not completely scanned all belong to the current search path $P(x)$ (in *find_ap*(x)). Line 6 is important for this. For instance in Fig. 5(c) when *find_ap*(8) is being executed, vertex 7 is not in $P(8)$, but 7 has been completely scanned. The detailed proof of this invariant and (P1) is in Appendix A.

The second step of the proof is more involved. It gives the key property for showing *find_ap* does every every possible blossom step. This property is Lemma A.2 proved in Appendix A and restated here:

(P2) At any point in *find_ap_set*, let rs be an edge that has been scanned from both its ends, with $r, s \notin \mathcal{P}$. Then $b(r) = b(s)$.

Only outer vertices scan edges, so r and s are outer in (P2). Edge $(7, 10)$ in Fig.5 illustrates (P2), after it is scanned from 10 in graph (e). (Fig.6 of Appendix A gives a more involved example.)

Using (P1) and (P2) we will now complete the proof of correctness. The following properties hold when *find_ap_set* halts:

(i) Any free vertex $f \notin V(\mathcal{P})$ is outer.
(ii) Any edge uv with u outer and $u, v \notin V(\mathcal{P})$ has v inner or $b(u) = b(v)$.

(i) holds since *find_ap_set* calls *find_ap*(f). For (ii) suppose u is outer and not in $V(\mathcal{P})$. (P1) shows u is completely scanned. So v is inner or outer, and we can assume $v \notin V(\mathcal{P})$. If v is inner (ii) holds. (Note v need not be in the same tree of $\mathcal{S}^-$ as u.) If v is outer (P1) shows it is completely scanned. Now (P2) applied to edge uv shows $b(u) = b(v)$. (Note $b(u) = b(v)$ implies u and v are outer and $B_u = B_v$.)

To complete the proof consider an alternating path P that is disjoint from $V(\mathcal{P})$ and starts at a free vertex f. We claim that every vertex of P is inner or outer. Furthermore for any outer blossom B with $B \cap V(P) \neq \emptyset$, the first vertex of P in B is the base b of B, and either $b = f$ or P enters B along a matched edge, i.e., the edge from $mate(b)$ to b. (Note B may be a singleton blossom, $B = \{b\}$.)

We prove this claim by induction on the length of P. The claim holds if P is just the starting vertex f, by (i). Inductively assume P has reached an outer vertex x, having already reached $b(x)$. Let y be the next vertex of P. Apply (ii) to xy. If $b(y) = b(x)$ the claim holds. The other possibility is that y is inner.[3] Its mate y' is outer and $b(y') = y'$. This implies y' has not been reached. (Any blossom base in $P - f$ was preceded by its mate.) Thus P proceeds from y to y'. The claim holds for y', since P enters $B_{y'}$ along the matched edge yy'. The induction is complete.

Finally note the claim shows P cannot be an augmenting path: Every vertex in $P - f$ is on a matched edge that either enters a blossom through its base, or stays within a blossom.

We close this section with two high-level properties of *find_ap_set* that lead to its linear time bound.

First, any edge is scanned at most twice, once from each end. In proof observe the call *find_ap*(x) occurs when x becomes a new outer vertex. Thus *find_ap* is called at most once for any given vertex $x \in V$.

Second, the test of line 3 is convenient to prove correctness of the algorithm, but is not straightforward to implement. The test of the comment is easily implemented. Appendix A shows the two tests are equivalent.

[3] y need not be in the search tree rooted at f.

5. The data structure

This section gives the data structures that show Phases 1 and 2 both use time $O(m)$.

The matching is represented using an array *mate*: A vertex v on matched edge vv' has $mate(v) = v'$. The forest $\mathcal{S}^-$ is represented using a pointer $\ell(v)$ for every outer vertex $v \in V$. Specifically a grow step adds y and y' to $\mathcal{S}^-$ by setting $\ell(y') = x$.

We now discuss each step of our algorithm in turn, presenting the data structure for it and verifying that linear time is achieved. We begin with Phase 1. Grow steps are handled as above.

Blossom steps We find the fundamental cycle C of edge xy in time $O(|C|)$ by climbing the paths from B_x and B_y to the root in parallel. This is accomplished using *mate* and ℓ pointers, and a data structure to find the base of a blossom B_x given an arbitrary vertex x.

The data structure is the incremental-tree set-merging algorithm of Gabow and Tarjan [13]. The operation $find(x)$ returns the base vertex of B_x. $union$ operations are used to contract C. The incremental-tree version of [13] works on a tree that grows by addition of leaves. So it works correctly on $\mathcal{S}^-$. The time for $O(m)$ $finds$ and $O(n)$ $unions$ is $O(m+n)$ and the space is $O(n)$.

Dual adjustment steps It is well-known how to implement dual adjustment steps efficiently using (a) the parameter Δ, defined as the current sum of all dual adjustment quantities δ so far; and (b) an appropriate priority queue. Our choice for (b) depends on the fact that all numerical quantities in Edmonds' algorithm are integral. To start we show the values $y(v)$, $v \in V(\mathcal{S})$, are always integers of the same parity.

We argue by induction. Initially every $y(v)$ is 1, giving the base case. In the definition of δ in Fig.2, every edge e is unmatched, so $w(e) = 0$. Thus $y(e) - w(e) = y(e)$ is an even integer. So the set defining δ consists of integers, and δ is integral. The adjustment of y values in Fig.2 keeps all values $y(v), v \in V(\mathcal{S})$ integers of the same parity. This completes the inductive step.

Next observe that throughout the algorithm every free vertex f has $y(f) = 1 - \Delta$. So Δ is always integral.

Now we show Δ is at most $n/2$. At any point in Edmonds' algorithm let P be any augmenting path. Lemma 3.1 shows $w(P) \le 2y(f)$. Using (3) gives

$$n - 1 \ge |P| = -w(P) + 1 \ge -2y(f) + 1.$$

Rearranging gives $y(f) \ge 1 - n/2$. With $y(f) = 1 - \Delta$ we get $\Delta \le n/2$ as claimed.

We can now sketch the algorithm and data structure. The priority queue consists of a collection of lists $L(d)$, each containing the edges that become tight when Δ has increased to d. To get the next edge for the search lists $L(\Delta')$ are examined, for $\Delta' = \Delta, \Delta + 1, \ldots$, until an edge triggering a grow, blossom, or augment step is found. This also gives the next value of Δ, as well as $\delta = \Delta' - \Delta$.

To populate the lists suppose a grow or blossom step makes vertex $u \in V$ outer. Every unmatched edge $e = uv$ is scanned. e is added to list $L(\Delta + d)$ where d is $y(e)$ if $v \notin V(\mathcal{S})$ or $y(e)/2$ if v is outer. The first case is for a future grow step: Here every dual adjustment decreases $y(e)$ by δ. So $y(e)$ becomes 0, i.e., e becomes tight, when Δ has increased by $y(e)$. The second case, for a future blossom or augment step, is similar: u and v are outer, so every dual adjustment decreases $y(e)$ by 2δ.

The total time and space for this data structure is $O(m+n)$, as desired.

The next two steps are implemented using the representation R_B for every maximal blossom B. In the data structure for R_B, the children of any node B' form a doubly linked ring. Each link also records the edge $xy \in E$ that joins the two subblossoms.

Constructing graph H To construct $V(H)$ first add every vertex except those in nonsingleton blossoms. The remaining vertices of $V(H)$ are contractions of the maximal positive blossoms. These are the blossoms that are maximal immediately before the last dual adjustment. During Edmonds' search each blossom is marked with the value of Δ when it is formed. We traverse each representation R_B top-down, discarding blossoms that have the final value of Δ. The remaining roots of R_B representations are the maximal positive blossoms (or possibly vertices of V) that get added to $V(H)$.

To construct $E(H)$, first continue the top-down traversals and label every leaf, i.e., vertex of V, with the $V(H)$ vertex containing it. Then scan every edge $e \in E$ and add e to $E(H)$ if it is tight and joins distinct $V(H)$ vertices, at least one being outer. (z values are not needed to detect tightness, since e joins distinct maximal blossoms.) H is represented as an adjacency structure, and the vertex labels are used to add e to the two appropriate adjacency lists. e also records its preimage in G, so augments can be performed in G.

Clearly the time to construct H is $O(m+n)$.

Turning to Phase 2, the blossom base function b is maintained as in Phase 1 using incremental-tree set-merging.

Computing $P(x)$ in *find_ap* First observe that an R_B data structure allows a path $P(v,b)$ to be computed in time $O(|B|)$ by following the recursive procedure sketched in Section 3.1. This suffices for our applications, since an augmenting path passes through a maximal blossom at most once. (In Phase 2 if an augmenting path passes through a blossom B, the vertices of B never get re-explored.) A more careful approach computes $P(v,b)$ in time linear in its length. The idea is not to start at the root of R_B (as in Section 3.1) but rather start at the node whose child contains the matched edge incident to v.

find_ap builds R_B representations for the blossoms it creates. They are used to compute the augmenting paths $yP(x)$ (line 2 of Fig.4). These paths are then converted to paths in G (the paths Q of Fig.1) by adding in $P(v,b)$ paths that traverse blossom-vertices of H. This is done using the R_B representations from Phase 1.

Again the total time is $O(m+n)$.

Combining our correctness proof and the above data structures giving linear time, our goal is achieved:

Theorem 5.1. A maximum cardinality matching can be found in time $O(\sqrt{n}m)$ and space $O(m)$.

A. Analysis of *find_ap_set*

This appendix completes the correctness proof for *find_ap_set* by proving properties (P1) and (P2) of Section 4. It concludes by validating the equivalent test for blossoms (line 3 of Fig.4) which is needed for efficiency of the algorithm.

Proof of (P1) We start by proving an invariant (I). Suppose *find_ap*(x) is currently executing but it has no recursive call that is executing. (x is deepest in the recursion stack.)

(I) $P(x) \cup \mathcal{P}$ contains every outer vertex that has not been completely scanned.

We allow this invariant to lapse for brief periods of time when the next call to *find_ap* is about to be made. This occurs in grow and blossom steps, see below. We also allow (I) to cover moments when no x exists, i.e., between *find_ap* invocations in line 1. At those moments $\mathcal{P}$ contains all the outer vertices not completely scanned.

We prove (I) by induction. When a new search is started in line 1, x is f, $P(x) \cup \mathcal{P} = \{f\} \cup \mathcal{P}$, and (I) holds by induction.

Assume (I) holds at a given instant in *find_ap*(x). If *find_ap*(x) is terminated because of an augmenting path (after line 2) all the vertices of $P(x)$, especially those not completely scanned like x, are added to $\mathcal{P}$. All invocations of *find_ap* are terminated. So (I) holds with no existant x.

If *find_ap*(x) executes a grow step, $P(y')$ contains $P(x)$. So (I) will hold when *find_ap*(y') is entered.

Suppose *find_ap*(x) executes a blossom step. After line 5 $P(x)$ does *not* contain the new outer vertices u_i – the invariant has lapsed. When *find_ap*(u_1) is entered, $P(u_1)$ contains every $u_j, 1 \le j \le k$, as well as x. So (I) is restored. When *find_ap*(u_1) returns, u_1 is completely scanned, so induction (applied to u_1) shows $P(u_2)$ contains every new outer vertex not completely scanned. (I) will hold when *find_ap*(u_2) is entered. This pattern continues for $u_3, \ldots, u_k$. When *find_ap*(u_k) returns, (I) holds for x.

The last case is when *find_ap*(x) returns in line 7. x is completely scanned. So (I) holds after the call *find_ap*(x), whether that call be from a grow step, a blossom step, or line 1. The induction is complete.

Examining (I) when the last call *find_ap*(f) in line 1 returns, we get (P1).

Proof of (P2), Lemma A.2 We shall use some simple facts:

If an outer vertex x has been completely scanned, any adjacent vertex is in $\mathcal{S}$ or $\mathcal{P}$.

The paths $P(x)$ are defined in Section 4 to contain the path $\overline{\mathcal{S}}(B_x, B_f)$. This containment holds even as blossoms are contracted and $\overline{\mathcal{S}}$ changes.

Let x be an outer vertex at any point in *find_ap_set*. Let b be the base vertex of a currently maximal blossom. $b \in P(x)$ iff b is an ancestor of x in $\mathcal{S}^-$. ($b \in P(x)$ iff B_b is an ancestor of B_x in $\overline{\mathcal{S}}$, by the previous fact. B_b is an ancestor of B_x in $\overline{\mathcal{S}}$ iff b is an ancestor of x in $\mathcal{S}^-$, since $\overline{\mathcal{S}}$ is a contraction of $\mathcal{S}^-$.)

Lemma A.1. At any point in *find_ap_set* consider an edge rs where r and s are outer and not in $V(\mathcal{P})$. $b(r)$ and $b(s)$ are related in $\mathcal{S}^-$.

Example: Consider edge $(8,6)$ immediately after the blossom step of Fig.5(b). $b(8) = 3$ is related to $b(6) = 6$ in $\mathcal{S}^-$. But 8 itself is not related to 6. Neither is 5, the mate of 8.

Proof:
We show each grow and blossom step preserves the lemma. Note that $b(r)$ and $b(s)$ may change over time but once they are related they remain related.

A grow step from x adds new vertices y, y' with y' outer, $b(y') = y'$. We can assume $y' = r$. Assume for the purpose of contradiction that $b(s)$ is not related to $b(r) = y'$. If $s \in P(x)$ then $b(s) \in P(b(x))$. This makes $b(s)$ an ancestor of $b(x)$, contradiction. Thus $s \notin P(x)$, and s is completely scanned by (I). Now edge rs implies yy' was added to $\mathcal{S}$ before the grow step, a contradiction.

Consider a blossom step. Assume r is a vertex that enters B_x, i.e., $b(r) = b(x)$. Again assume that $b(r)$ is not related to $b(s)$. Thus $b(s)$ is not a descendant or ancestor of $b(x)$. The first implies $b(s)$ became outer before *find_ap*$(b(x))$ was entered. The second implies $b(s) \notin P(b(x))$. Thus $b(s)$ was completely scanned before *find_ap*$(b(x))$ was entered. Every vertex in the blossom of $b(s)$ was also completely scanned, in particular s was completely scanned. So edge rs implies r was added to $\mathcal{S}$ before *find_ap*$(b(x))$ was entered. But r descends from $b(x)$. □

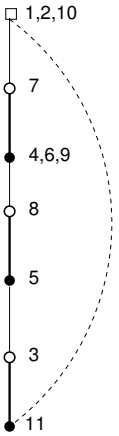

Figure 6. In the graph Fig.5(a) with $(4,6)$ added, vertex 1 scans $(1,2)$ before $(1,11)$. $\overline{\mathcal{S}}$ is shown after two blossom steps, with vertex 1 about to scan $(1,11)$. This blossom step makes $b(6) = b(8)$ and $b(7) = b(10)$, yet it is not triggered by $(6,8)$ or $(7,10)$.

Lemma A.2. At any point in *find_ap_set*, let rs be an edge that has been scanned from both its ends, with $r, s \notin \mathcal{P}$. Then $b(r) = b(s)$.

Proof:
The first time r and s are both outer $b(r)$ and $b(s)$ are related in $\mathcal{S}^-$, by Lemma A.1. Wlog assume at this time

(i) $b(r)$ is an ancestor of $b(s)$.

Clearly (i) holds even as $b(r)$ and $b(s)$ change.

Consider three possibilities for s at the moment when r scans edge rs.

Case *s is outer*: (i) shows the test of line 3 is passed and the blossom step makes $b(r) = b(s)$.

The next two cases are illustrated by Fig.6: 10 scans edge $(10, 7)$ with $7 \notin V(\mathcal{S})$; 6 scans $(6, 8)$ with 8 inner.

Case $s \notin V(\mathcal{S})$: A grow step makes s an inner child of r. Eventually s becomes outer in a blossom step. The new blossom has an outer base vertex, so the blossom includes r, i.e., $b(r) = b(s)$.

Case *s is inner*: Let s become outer in a blossom step triggered by edge xy, where $b(x)$ is an outer ancestor of s. We have assumed $b(\cdot)$ refers to the time immediately after the blossom step. So

$$b(s) = b(x).$$

Consider the moment r scans rs. s is inner so $b(x)$ is already outer. The blossom step has not occurred so $b(x)$ is not completely scanned. Now (I) shows $b(x) \in P(r)$. Thus

(ii) $b(x) = b(s)$ is an ancestor of r.

Recall $\overline{\mathcal{S}}$ is a contraction of $\mathcal{S}^-$ (Section 4). So every vertex on the $\mathcal{S}^-$-path from r to $b(r)$ is in B_r. (i) and (ii) show $b(s)$ is on this path. So $b(s) \in B_r$. B_r contains a unique vertex that is the base of a maximal blossom. Thus $b(s) = b(r)$. □

The equivalent test To implement the algorithm efficiently we change the test for a blossom step, line 3, to the test of the comment. We will show the two tests are equivalent. Specifically assume edge xy has both ends outer.

$b(y)$ is an outer proper descendant of $b(x)$ in $\mathcal{S}^-$ iff $b(y)$ became outer strictly after $b(x)$.

In proof first observe that as outer blossom bases, $b(x)$ and $b(y)$ were both added to $\mathcal{S}$ in a grow step making them outer.

The only if direction is trivial: Any vertex is added to $\mathcal{S}^-$ after its ancestors.

To prove the if direction, assume $b(y)$ became outer strictly after $b(x)$, i.e., $b(y)$ was added to $\mathcal{S}$ after $b(x)$. Edge xy ensures $b(x)$ and $b(y)$ are related in $\mathcal{S}^-$ (Lemma A.1). So $b(y)$ descends from $b(x)$.

B. Searching from the middle

The algorithm of Micali and Vazirani [1] is based on a "double depth-first search" (DDFS): This search begins at an edge $e = uv$. It attempts to complete an augmenting path using vertex-disjoint paths from each of u and v to a free vertex. This is done with two coordinated depth-first searches, one starting at u, the other at v.

The key fact justifying this approach is a characterization of the starting edge e. We will begin by describing the conditions satisfied by e, using our terminology. Then we prove that any *sap* contains such an e. We conclude by discussing implications of this structure – how DDFS can be used for our Phase 2, and how the analysis of this section could be extended to a complete proof of correctness of the Micali-Vazirani algorithm.

We need one preparatory remark. Recalling the definition of domination (1), say edge uv has *slack* equal to $\big(y(u)+y(v)+\sum_{u,v\in B} z(B)\big)-w(uv)$. Let P be an augmenting path, not necessarily maximum weight. Let σ be the total slack in all unmatched edges of P. Then

$$w(P) = 2y(f) - \sigma. \tag{B.1}$$

This is simply the upper bound (2), made tight by subtracting the total slack.

We start the characterization with terminology based on the state of the search immediately before the last dual adjustment. Let T' be the set of edges of G that are tight at that time. Let $D_1 \cup D_2$ be the set of edges that become tight in the last dual adjustment, where D_1 refers to a grow step and D_2 is for a blossom step. So $e \in D_1$ has slack $y'(e) = \delta$ with one end of e outer and the other not in $\mathcal{S}$. $e \in D_2$ has slack $y'(e) = 2\delta$ with both ends of e outer. (Recall $w(e) = 0$.) Here y' is the dual function right before the last dual adjustment, and "outer", $\mathcal{S}$, and δ also refer to that time.

Lemma B.1. Any maximum weight augmenting path of G can be written as

$$P_1, Q, P_2$$

where

each P_i is an even-length alternating path from a free vertex to an outer vertex, $P_i \subseteq T'$,

Q has the form (e) with $e \in D_2$, or (g_1, e, g_2) with $g_1, g_2 \in D_1$.

Remarks: Strictly speaking the augmenting path ends with the reverse of P_2, but we relax the notation for simplicity. Clearly e is unmatched in the first form and matched in the second. Neither end of e is in $\mathcal{S}$ in the matched form.

Proof:
Let y be the final dual function. The dual adjustment step shows that any free vertex f has $y(f) = y'(f) - \delta$. As mentioned in the proof of Corollary 3.2 an augmenting path P has maximum weight iff $w(P) = 2y(f)$. Thus

$$w(P) = 2y'(f) - 2\delta.$$

Comparing with (B.1) shows P contains edges that have total slack 2δ *wrt* y' but are tight *wrt* y. Clearly these are the edges of $(D_1 \cup D_2) \cap P$.

Suppose P contains an edge $e \in D_2$. Since $y'(e) = 2\delta$, P contains exactly 1 such edge. The properties of the lemma for both P_i and Q follow easily.

The other possibility is that P contains exactly two edges $g_1, g_2 \in D_1$. Each g_i is unmatched and has an end $v_i \notin \mathcal{S}$. P must contain a v_1v_2-subpath of edges in T'. It must consist of just one edge $v_1v_2 \in M$, since unmatched edges with no end in $\mathcal{S}$ are not tight. The properties of the lemma for both P_i and Q follow. □

It seems surprising that *find_ap* succeeds while ignoring this structure. So we take a closer look. We need a simple fact:

Proposition B.2. An edge $uv \in T'$ with u inner has v outer.

Proof:
A grow step that makes u inner has $y(u) = 1$. Every subsequent dual adjustment increases $y(u)$. So right before the last dual adjustment $y'(u) \geq 1$. If v is inner or not in $\mathcal{S}$ then $y'(v) \geq 1$ and $uv \notin M$. Thus $y'(u) + y'(v) \geq 2 > w(uv) = 0$, i.e., $uv \notin T'$. □

We now present a more detailed proof of the lemma. Consider the search graph $\overline{\mathcal{S}}$ immediately before the last dual adjustment. $\overline{\mathcal{S}}$ is a subgraph of H. Define a path form in H similar to the lemma:

$$P, Q, P'$$

forming an even-length alternating path where

P has even length and goes from a free vertex to an outer vertex of $\overline{\mathcal{S}}$, $P \subseteq T'$;

Q has the form of the lemma;

P' has odd length, its last edge is matched, and last vertex is inner in $\overline{\mathcal{S}}$, $P' \subseteq T'$.

In contrast to the lemma, P' starts at the end of Q (i.e., no implicit reversal this time).

Let A be an arbitrary alternating even-length path in H that starts at a free vertex. We claim that A is a prefix of the above form. Clearly this claim forces *find_ap* to find a path with the structure of the lemma.

We prove the claim inductively. Suppose an even length prefix A' of A ends at vertex u, and the next two edges of A are uv, vv' with $uv \notin M \ni vv'$.

If A' has length 0 then u is free. A' has the form P.

Suppose A' has form P. There are three possibilities:

Subcase *v is inner in $\overline{\mathcal{S}}$*: Its mate v' is outer, so form P holds for the longer prefix.

Subcase *v is outer in $\overline{\mathcal{S}}$*: uv joins two outer vertices of $\overline{\mathcal{S}}$. Thus $uv \in D_2$. The matched edge vv' joins an outer vertex with an inner, so v' is inner. So the new prefix of A has form P, Q, P' for $P' = (vv')$.

Subcase $v \notin \overline{\mathcal{S}}$: This makes $uv \in D_1$. The new prefix has the form P, g_1, e with g_1, e as in Q.

Now suppose A' has form P, g_1, e with g_1, e as in Q. Since $e \notin \overline{\mathcal{S}}$ and uv is tight, $uv \in D_1$. (uv is unmatched with $u \notin \mathcal{S}$. It is not tight if $v \notin \mathcal{S}$.) So v is outer. Thus v' is inner. The new prefix has form P, Q, P' ($P' = (vv')$).

Finally suppose A' has form P, Q, P'. No end of an edge of $D_1 \cup D_2$ is inner. Since u is inner this makes $uv \in T'$. So Proposition B.2 makes v outer. The new prefix ends with edge $vv' \in M$ and v' inner. Thus it has form P, Q, P'. The induction is complete.

The lemma opens up the possibility of having a Phase 2 search start from an edge e of type Q. DDFS uses this strategy.

The Micali-Vazirani algorithm uses DDFS in Phase 1 as well. This depends on the fact that blossoms have a starting edge e similar to the lemma. (More precisely suppose a blossom step creates a blossom B with base b, with $v \in B$ a new outer vertex. Then $P(v, b)$ contains a unique subpath of form Q of the lemma. This is easily proved as above, e.g., use the second argument, traversing the path $P(v, b)$ starting from b.)

Using DDFS in both Phases 1 and 2 makes the Micali-Vazirani algorithm elegant and avoids any overhead in transitioning to Phase 2.

The proof of [3] that DDFS is correct is involved. Possibly it could be simplified using the lemmas we have presented, as well as other structural properties that weighted matching makes clear. The following aspects of the finer structure of H are not needed for our development but are used in [3].

[3] defines *evenlevel*(x) as the length of a shortest even alternating path from a free vertex to x. The proof of Lemma 3.1 shows any even alternating fx-path has weight $\leq y(f) - y(x)$, i.e., length $\geq y(x) - y(f)$. Furthermore it shows that an outer vertex x has $evenlevel(x) = y(x) - y(f) = |P(x)|$.

oddlevel(x), the length of a shortest odd alternating path from a free vertex to x, has a similar characterization, e.g., any odd alternating fx-path has weight $\leq y(f) + y(x) + \sum z(B)$, i.e., length $\geq 1 - y(x) - y(f) - \sum z(B)$, where the sum is over blossoms B with base vertex b and $x \in B - b$.

Finally [3] divides the edges of H into *bridges* and *props*. This is due to the fact that an edge e of form Q can trigger an initial blossom step, which can be followed by blossom steps triggered by unmatched edges of T'. e is a bridge and the other triggers are props. (In the precise blossom structure stated above, e is the Q edge and the prop triggers are in T'.)

Acknowledgments

The author thanks two anonymous referees for a number of suggestions that helped clarify the presentation.

References

[1] Micali S, Vazirani VV. An $O(\sqrt{|v|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In: Proc. 21st Annual Symp. on Found. of Comp. Sci. 1980 pp. 17–27. doi:10.1109/SFCS.1980.12 .

[2] Vazirani VV. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. Combinatorica, 1994;14(1):71–109.

[3] Vazirani VV. A simplification of the MV matching algorithm and its proof. CoRR, abs/1210.4594v5. Also "A proof of the MV matching algorithm", manuscript, May 13, 2014, 42 pages.

[4] Gabow HN, Tarjan RE. Faster scaling algorithms for general graph matching problems. J. ACM, 1991;38(4):815–853. doi:10.1145/115234.115366.

[5] Edmonds J. Maximum matching and a polyhedron with 0,1-vertices. J. Res. Nat. Bur. Standards, 1965;69B:125–130. URL `https://www.bibsonomy.org/bibtex/2a3a3ef104190926e124d4126d2bfc919/ytyoun`.

[6] Cook WJ, Cunningham WH, Pulleyblank WR, Schrijver A. Combinatorial Optimization. Wiley and Sons, New York, 1998. ISBN-10:0486402584, 13:978-0486402581.

[7] Lawler EL. Combinatorial Optimization: Networks and Matroids. Holt, Rinehart and Winston, New York, 1976. ISBN-0486414531, 9780486414539.

[8] Papadimitriou CH, Steiglitz K. Combinatorial Optimization: Algorithms and Complexity. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982. ISBN-0486402584, 9780486402581.

[9] Schrijver A. Combinatorial Optimization: Polyhedra and Efficiency. Springer, New York, 2003. ISBN-10:3540443894, 13:978-3540443896.

[10] Hopcroft JE, Karp RM. An $n^{2.5}$ algorithm for maximum matchings in bipartite graphs. SIAM J. Comput. 1973;2(4):225–231. URL https://doi.org/10.1137/0202019.

[11] Karzanov AV. On finding maximum flows in network with special structure and some applications. Math. Problems for Production Control. Moscow State University Press, Moscow 1976;5:81–94.

[12] Goldberg AV, Karzanov AV. Maximum skew-symmetric flows and matchings. Math. Program., Series A. 2004;100(3):537–568. doi:10.1007/s10107-004-0505-z.

[13] Gabow HN, Tarjan RE. A linear–time algorithm for a special case of disjoint set union. J. Comp. and System Sci., 1985;30(2):209–221. URL https://doi.org/10.1016/0022-0000(85)90014-5.