Algorithmica (1992) 8: 89–101

Algorithmica


# Path-Matching Problems[1]

Sun Wu[2] and Udi Manber[2]

**Abstract.** The notion of matching in graphs is generalized in this paper to a set of paths rather than to a set of edges. The generalized problem, which we call the *path-matching problem*, is to pair the vertices of an undirected weighted graph such that the paths connecting each pair are subject to certain objectives and/or constraints. This paper concentrates on the case where the paths are required to be edge-disjoint and the objective is to minimize the maximal cost of a path in the matching (i.e., the bottleneck version). Other variations of the problem are also mentioned. Two algorithms are presented to find the best matching under the constraints listed above for trees. Their worst-case running times are $O(n \log d \log w)$, where $d$ is the maximal degree of a vertex, $w$ is the maximal cost of an edge, and $n$ is the size of the tree, and $O(n^2)$, respectively. The problem is shown to be NP-complete for general graphs. Applications of these problems are also discussed.

**Key Words.** Algorithm, Bottleneck, Graphs, Matching, Paths, Trees.

**1. Introduction.** Given an undirected graph $G = (V, E)$, a *matching* is a set of edges no two of which have a vertex in common. Each edge in the matching connects two vertices that are said to be *matched.* One is usually interested in finding maximum matchings, that is, matchings that include as many edges as possible. Sometimes the edges have associated weights and one is interested in finding maximum (or minimum) weight matchings. Problems involving matching occur in many situations (see [3] or [4]). Workers may be matched to jobs, machines to parts, players to teams, etc. Furthermore, many problems that seem unrelated to matching have equivalent formulations in terms of matching problems.

In this paper we introduce a generalization of the matching problem, show its usefulness, and solve several algorithmic problems associated with it. Let $G = (V, E)$ be an undirected weighted graph with $2n$ vertices. A *path-matching* in $G$ is a set of simple paths with distinct end vertices. A regular matching is thus a special case of a path-matching in which all the paths consist of exactly one edge. A *perfect path-matching* in $G$ is a set of $n$ paths such that each vertex in $G$ is the end vertex of exactly one path. (If the number of vertices is odd, we will call a matching that leaves exactly one vertex unmatched perfect.) Not all graphs contain a perfect (regular) matching. The next lemma shows that all connected graphs with even

[1] Udi Manber was supported in part by an NSF Presidential Young Investigator Award (Grant DCR-8451397), with matching funds from AT&T.

[2] Department of Computer Science, University of Arizona, Tucson, AZ 85721, USA.

Received August 20, 1989; revised March 23, 1990. Communicated by Nimrod Megiddo.

number of vertices contain a perfect path-matching, even if the paths are restricted to be edge-disjoint. Path-matchings may be useful in cases where there is no perfect matching and we can be satisfied with paths instead of edges. We present one such example next. We will generally be interested in optimal matchings, therefore, we will assume that the number of vertices is even. For simplicity of notation, we will call a path-matching a *P-matching*, and assume that a P-matching is perfect. A P-matching with edge-disjoint paths will be called a *DP-matching*. Unlike regular matching, it is easy to determine which graphs have a DP-matching, as is seen in the following simple lemma.

LEMMA 1. *Every connected undirected graph with an even number of vertices contains at least one DP-matching.*

PROOF. It suffices to prove the lemma for trees since every connected graph contains a spanning tree. The proof is by induction on the number of vertices. Since the number of vertices is even, the induction starts at $n = 2$ and advances in steps of 2. The base case is trivial. Let $T_i$ be a tree with $n$ vertices, and consider it as a rooted tree with (an arbitrary) root $r$. Let $v$ be a leaf in $T_i$ that is farthest away from the root, and let $w$ be its parent. If $w$ has other children besides $v$, then they must be leaves; in that case we match $v$ with one of these children, and remove the path. The remaining graph is still a tree, hence, by induction, it has a DP-matching, and we are done. If $v$ is $w$'s only child, then we match $v$ with $w$ and remove the edge. Again, we are done by induction. □

The following application motivated our study. Suppose that $G$ models a network of computers such that each vertex corresponds to a computer and each edge corresponds to a link of communication. Each link is associated with a cost of using that link (e.g., load, tariff, delay). Suppose furthermore that we want to organize a *tournament* among the computers such that each computer is paired with another one and they perform some competition together. The competition may correspond, for example, to some computation task that both computers are involved in. We would like to pair the computers so that communication is minimized. A minimum cost perfect matching would be the best solution, but there may not be one. A P-matching is thus required. One may want to insist on edge-disjoint paths to parallelize the communication as much as possible, or, if the bandwidth is high enough, edge-disjoint paths may not be required. One objective may be to minimize the total cost, which corresponds to the sum of the costs of all involved edges; we call this variation of the problem the *min-sum* problem. Another objective may be to minimize the delay (parallel time), which corresponds to the maximal cost of a path in the P-matching; we call this variation of the problem the *min-max* problem (such problems are also called *bottleneck problems*). We will address all these variations, but we concentrate on the min-max variation. Algorithms for some min-max (regular) matchings are presented in [1].

We start with some basic definitions. The input is a weighted undirected graph $G = (V, E)$ with positive weights and even number of vertices. The *length* of a path is the number of edges it contains. The *cost* of a path is the sum of the weights

of its edges. We use the term *longest–path* to denote the path having maximal number of edges, and *max–path* to denote the path with maximum cost. Let $k$ be an integer; we say that a P-matching is bounded by $k$ if the cost of its *max–path* is at most $k$. We say that a (D)P-matching is maximal for a graph with odd number of vertices, if only one vertex is left unmatched.

The following three facts are easy to verify. For brevity sake, we leave the proof to the reader (see also [5]). We mention these facts for completeness; we do not rely on them in this paper.

FACT 1. There exists an $O(n)$ algorithm for finding min-sum P-matching for trees (the algorithm can be directly obtained from Lemma 1).

FACT 2. For general graphs, there is a min-sum P-matching that contains only paths with at most two edges each. This P-matching can be obtained by computing all shortest paths (with at most two edges) and finding a minimum-cost matching where the costs correspond to path weights.

FACT 3. There exists an $O(\max(n^{2.5} \log w, n^3))$ algorithm for finding min-max P-matching for integer-weighted general graphs, where $w$ is the weight of the *max–path*.

We concentrate in this paper on min-max DP-matchings. We give three results. First, an $O(n \log d \log w)$ algorithm for finding min-max DP-matching for integer-weighted trees, where $d$ is the maximal degree of a vertex and $w$ is the maximal cost of an edge; second, an $O(n^2)$ algorithm for finding min-max DP-matching for trees with arbitrary costs; and third, a proof that the min-max DP-matching problem for general graphs is NP-complete.

**2. An Algorithm for Integer-Weighted Trees.** If the weights are integers, then we can use binary search (on the weights) to reduce the problem to the following decision problem, which we call a *feasibility test*: Given a weighted tree $T$ and a bound $B$, determine whether there exists a DP-matching whose max cost is $\leq B$. We solve the decision problem for $B = 1, 2, 4 \ldots$, until the answer is positive for, say, $B = 2^k$, and then apply regular binary search in the range $2^{k-1} + 1$ to $2^k$. Consider a tree $T$ rooted at $r$ as shown in Figure 1, and suppose that the feasibility test is attempted with bound $B$. If any of $T$'s subtrees $T_i$ is an even subtree (a subtree with even number of vertices), then the DP-matching of vertices of $T_i$ must be within $T_i$, because only one edge leaves $T_i$. Therefore, we have to verify only that the cost of the DP-matching in $T_i$ is at most $B$. However, if $T_i$ is an odd subtree, then one of its vertices must remain unmatched; the problem is to determine which vertex should be left unmatched. We call the path from the unmatched vertex $v_j$ in $T_i$ to $r_i$ an *active path*. (The active path is an empty path if the unmatched vertex is $r_i$.) For each odd subtree we will find the DP-matching with cost at most $B$ that *minimizes* the active path. We will call such a DP-matching a *minimal* DP-matching for the subtree. Our only interest in the feasibility test is whether

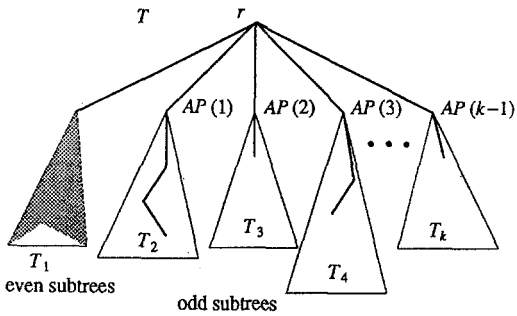



**Fig. 1.** The recursive structure of the algorithm.

there is a DP-matching with cost at most $B$. Therefore, we need not minimize the cost of the DP-matching in the subtree, as long as it is within the bound of $B$.

We find minimal DP-matchings by a bottom-up traversal. We assume that all the subtrees of $T_i$ have been traversed, and either minimal DP-matchings (in the case of even subtrees) or the minimal active paths (in the case of odd subtrees) for them have been found. If any of the partial DP-matchings cost more than $B$, then the feasibility test obviously fails. Our task is to match the active paths (each of which corresponds to an unmatched vertex) in the best possible way. Since $r$ has to be matched as well, we consider it as an active path with a cost of 0. We denote the active path of the odd subtree $T_j$ by $AP(j)$, and we add to all active paths the cost of the edge connecting the corresponding $r_i$ to $r$ (which the active path must use). Suppose that the active paths $AP(j)$ are sorted in increasing order of their cost.

The easy case is again the case where $T$ is an even subtree. Clearly, the best DP-matching in $T$ results from pairing the largest active path with the smallest one (the root), the second largest with the second smallest, and so on. Assume now that $T$ is an odd subtree. We must find a pairing of the active paths with cost at most $B$ such that the unmatched active path (which will be the active path of $T$) has minimum cost. If we decide to choose the active path $AP(j)$, then the best pairing for the rest of the active paths can be easily determined (the largest with the smallest, second largest with second smallest, and so on). However, we do not have to consider all possibilities. The cost of pairing the active paths increases when a smaller active path is excluded. We can use binary search to find the smallest active path whose exclusion still results in a pairing of cost at most $B$.

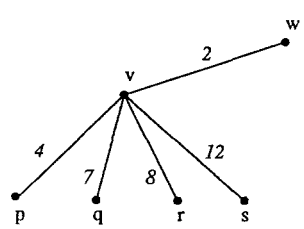
*Time Complexity.* The number of times the feasibility test for the whole tree is called is clearly $O(\log p)$, where $p$ is the cost of the *max–path* of the solution. However, by Lemma 1, in trees there is always a DP-matching that consists of paths of at most two edges. Therefore, the cost of the *max–path* is no more than twice the maximal cost of an edge, which we denote by $w$, and the feasibility test is called at most $O(\log w)$ times. Let the degree of root $r_i$ be $d_i$. The feasibility test for each odd subtree requires $O(d_i \log d_i)$ steps for the sorting, and $O(d_i)$ steps for finding the best pairing for each attempt to pair. There are $O(\log d_i)$ such attempts (using binary search). Therefore, the overall running time is

$O(\sum_{i \text{ is a vertex in } T} d_i \log d_i)$. Since $\sum d_i = n$ it is easy to see that the sum above is $O(n \log d)$, where $d = \max(d_i)$. The overall running time of this algorithm is thus $O(n \log d \log w)$.

The algorithm above performs well when the weights are integers (and they are not too large). But if the weights are not integers or if they are very large integers, then binary search is not a good approach. The next algorithm eliminates the need for binary search with increased time complexity.

**3. An Algorithm for Trees with Abrbitrary Weights.** In this section we present an efficient algorithm to solve the min-max DP-matching problem for trees with arbitrary positive weights. Throughout this section we consider the tree to be a rooted tree by choosing an arbitrary node $r$ to be the root. Each edge in a tree separates the tree into two subtrees. If these subtrees are even, then, as we have already mentioned, each subtree must be matched within itself. The difficult case is thus when the trees are odd.

The problem with an odd subtree is that it has an unmatched vertex, and we have to treat it in a special way. This is the major difficulty in the algorithm. Consider a vertex $v$ which is a root of $d$ odd subtrees. We would like to be able to combine the solutions for the subtrees to a solution for the whole tree. But, there is not just one "best" solution for each subtree; it depends on the unmatched vertex, as is shown in Figure 2. Since there are several possible "best matchings" for each subtree, we will keep track of all of them. The problem thus becomes one of combining sets of all possible matchings of the subtrees to a set of all possible matchings of the whole tree. A particular maximal matching of an odd subtree is denoted by a record $p$ that has two components: the cost of the matching $p.C$, and the cost of the active path $p.AP$. We call such a record a *point*. For each subtree, we will find a set of all possible *minimal points* in the following way. Suppose that $p$ and $q$ correspond to two different valid matchings for $T$. We say that $p \leq q$ if $p.AP \leq q.AP$ and $p.C \leq q.C$. In this case, $q$ can be discarded since $p$ is better than $q$ in all respects. Let $S$ be a set of points. A point $p$ is called a *minimal point* in $S$ if no point in $S$ is smaller than $p$. The matchings that correspond to minimal points are those that we are interested in. We will find all the minimal points associated with every odd tree. Let us first see how to find the set of all minimal points corresponding to an odd tree of height 1.




| matching | active path | cost |
|---|---|---|
| p--s q--r | v--w | |
| 16 15 | 2 | 16 |
| q--r s--v | p--w | |
| 15 12 | 6 | 15 |
| p--r v--s | q--w | |
| 12 12 | 9 | 12 |

**Fig. 2.** The possible *best* solutions for a subtree.

Consider an odd subtree of height 1 with a root $v$ and $d$ leaves with weights $c_1 \leq c_2 \leq \cdots \leq c_d$ on the corresponding edges (notice that $d$ is even). If the active path includes leaf $i$ then its cost is $c_i$, and it is easy to see that the best matching among the rest of the vertices is $v$ with leaf $d$ (or $d-1$ in case $i = d$), the maximal remaining leaf with the minimal, and so on. For example, let the cost be 1, 4, 6, 7, 8, 11 and let $i = 2$, then the best matching is $v$ with $v_6$, $v_1$ with $v_5$, and $v_3$ with $v_4$, whose max cost is 13. We can compute these costs for all possible active paths and then discard all points that are not minimal. Finding all minimal points can be done by sorting all points according to increasing active paths, then scanning the sorted sequence, and leaving only those points with costs that are smaller than those of the previous points. In the example above there are three minimal points: they correspond to picking the root for the active path (in which case the active path has cost 0, and the matching has cost 13), picking the third leaf for the active path (in which case the active path has cost 6, and the matching has cost 11), and picking the sixth leaf for the active path (in which case the active path has cost 11, and the matching has cost 10). Generally, few of the leaves will correspond to minimal points, but in the worst case, there could be $d$ minimal points.

Let $T$ be a tree with $d-1$ odd subtrees $T_1, T_2 \ldots, T_{d-1}$, each associated with a set of minimal points. We are now left with the problem of combining the lists of minimal points of the subtrees of $T$ into one list of minimal points for $T$. (From now on we will call the minimal points of the subtrees simply *points*, and reserve the term *minimal points* to the combined minimal points for $T$.) We denote the set of points of subtree $i$ by $p_1^i, p_2^i, \ldots, p_{k(i)}^i$, and we assume that they are sorted in increasing order of active paths (namely, for all $i$ and $j$, $p_j^i.AP < p_{j+1}^i.AP$). We add to the active paths of each of these points the cost of the edge to the root of $T$ (which the active paths must use). Furthermore, since the root of $T$ has to be matched as well (or be used as an active path) we consider it as a subtree with one minimal point (whose active path and cost are 0). From now we will talk about $d$ subtrees and include the root as a subtree.

At most one active path can continue up from $T$, hence the rest of the active paths must be matched. To find the minimum DP-matching for $T$, we need to consider all possible matchings of the active paths corresponding to the points of the subtrees. Let $Q = (q_1, q_2, \ldots, q_d)$ be a *tuple* such that each $q_i$ is a representative point from a distinct subtree. We assume that the points in $Q$ are sorted in increasing order of their active paths, so that $q_1.AP \leq q_2.AP \leq \cdots \leq q_d.AP$. If $d$ is even, which is the easier case since $T$ is an even tree, then the best matching of the points in $Q$ is clear. It is the same as if the points corresponded to leaves. We match the point with the largest active path with the point with the smallest, the second largest with the second smallest, and so on. If $d$ is odd then one active path must leave $T$. In that case, we consider each point $q_i$ separately, and find the best matching (as above) without it. This procedure leads to at most $d$ combined minimal points (one for each $q_i$ serving as an active path). The active path of the point corresponding to choosing $q_i$ is the active path for $q_i$. The cost of choosing $q_i$ as the active path will be the higher of the maximal cost of pairing the rest of the active paths or the maximal cost of all the points in $Q$.

Thus, for each tuple $Q$ we can compute combined minimal points for the whole tree. The problem is that there are too many possibilities. There are $k(1) \times k(2) \times \cdots \times k(d-1)$ possible tuples (the root corresponds to only one point). We cannot afford to check all of them. However, we are only interested in finding *minimal* combined points. Next, we show how to consider only a small set of the tuples which will be sufficient to find all combined minimal points.

Let us start with the easier case of even $d$. For each tuple $Q$ we denote by $Q.Pair$ the minimum cost of pairing the points in $Q$ (the cost of a pairing is the maximal, over all matched pairs, of the sum of their active paths). Since $d$ is even there is one unique minimum cost of pairing its points (the point with largest active path with the point with the smallest active path, the second largest with the second smallest, and so on), thus $Q.Pair$ is uniquely defined. We denote by $Q.Max$ the maximal cost among the points in $Q$. We start with $Q = (p_1^1, p_1^2, \ldots, p_1^d)$, namely, we choose from each subtree the point with the smallest active path. We match these points as described above and find $Q.Pair$. If $Q.Pair \geq Q.Max$ then we claim that this matching is the minimum in $T$. This is so because any other tuple will have larger active paths and thus larger sums. On the other hand, if $Q.Pair < Q.Max$ then the cost of this matching remains $Q.Max$, and there is a hope of finding a better matching since $Q'.Max$ may be smaller than $Q.Max$ for another tuple $Q'$. (Recall that, for each subtree, the points are ordered in increasing active paths and thus decreasing costs.) In other words, other tuples have larger active paths, but they may also have smaller costs, and if the pairing cost associated with $Q$ was due to $Q.Max$ then there is still hope.

We find the best matching for $T$ as follows. Let $p_1^i$ be the point with the maximal cost among the points in $Q$, namely, $p_1^i.C = Q.Max$. We replace $p_1^i$ with $p_2^i$ in the tuple, and reorder the points if necessary (so that they remain sorted). We then pair the points as above, and continue in the same manner. In each step, we compare $Q.Pair$ to $Q.Max$. there are two cases:

(1) $Q.Pair \geq Q.Max$—in this case we are done (by the same argument as above), and
(2) $Q.Pair < Q.Max$—we replace the maximal cost point with the point next in its list.

This process terminates either by encountering case 1, or by trying to replace a point which is last on its list.

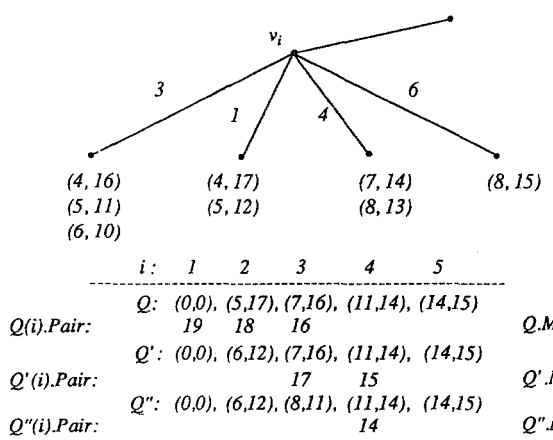
For each tuple $Q$ that we consider we find $Q.Pair$ and $Q.Max$. The cost of the corresponding DP-matching is $\max(Q.Pair, Q.Max)$. At the end we choose the tuple $Q$ that corresponds to the minimal DP-matching. We have to prove that the tuples that we have not considered cannot lead to a better matching. The reason for that is that replacing the point with the maximal cost is mandatory. All the tuples $Q'$, which use this point and which we have not considered, include points further down in the corresponding lists. Since the points are sorted according to their active paths and $Q'.Max = Q.Max$ (the point with $Q.Max$ is not changed), the cost of matching in $Q'$ is no lower than that of $Q$. If there is a total of $n$ points in all the subtrees of $T$ then we need to consider at most $n$ tuples.

The number of minimal points associated with a vertex $v$ in the tree cannot exceed the number of descendants of $v$ since a point corresponds to an active path.

We now consider the harder case of an odd $d$. This case is harder because there is now an active path coming out of $T$, and thus we can no longer determine the best DP-matching for $T$. We have to find the set of all minimal points of $T$. However, the procedure for finding the set of all minimal points is very similar to the procedure for the even case. Let $Q$ be again the tuple $(p_1^1, p_1^2, \ldots, p_1^d)$. We can no longer find the "best" pairing for it since it depends on the chosen active path. Instead, we will do the same thing we did for the case of a tree of height 1.

We denote by $Q(i).Pair$ the cost of pairing all points in $Q$ except for $q_i$, and by $Q.Max$ the maximal cost of all points in $Q$. If $Q$ is the tuple used for the matching, then the total cost cannot be lower than $Q.Max$. We will compute all $Q(i).Pair$ for those $i$ that lead to minimal points. The fact that the points of $Q$ are sorted by their active paths implies that $Q(1).Pair \geq Q(2).Pair \geq \cdots \geq Q(d).Pair$. We compute $Q(i).Pair$ for $i = 1, 2, \ldots$ until we find an $i$ such that $Q(i).Pair \leq Q.Max$. In that case, there is no need to continue checking other active paths in $Q$ since we cannot achieve a better cost than $Q.Max$ and we might as well use the smallest active path to achieve it (in other words, we cannot get more minimal combined points). We now continue in a similar fashion to the even case by replacing a point with the next one on its list. But first, let us see an example.

Suppose that there are four subtrees with the following minimal points (after we added the edge cost to the active path; see Figure 3): [(7, 16), (8, 11), (9, 10)], [(5, 17), (6, 12)], [(11, 14), (12, 13)], and [(14, 15)]. Of course, we also have to add the "point" (0, 0) corresponding to the root. The first $Q$ consists of the points (0, 0), (5, 17), (7, 16), (11, 14), and (14, 15). $Q.Max = 17$. $Q(1).Pair = 19$ (pairing 7 to 11 and 5 to 14), with the active path of cost 0. $Q(2).Pair = 18$ (pairing 7 to 11 and 0 to 14), with the active path of cost 5. $Q(3).Pair = 16$ (pairing 5 to 11 and 0 to 14), with the active path of cost 7. We can now stop considering $Q$ since other active paths will be larger and the cost cannot be lower than




*The minimal combined points generated are (0, 19), (5, 18), (7, 17), (11, 15).*

**Fig. 3.** An example.

$Q.Max = 17$. So far we have found three minimal combined points—(0, 19), (5, 18), and (7, 17). We now replace in $Q$ the point (5, 17), which sets the value of $Q.Max$, with the next point in the corresponding list, (6, 12). The new tuple $Q'$ consists now of the points (in increasing active path order) (0, 0), (6, 12), (7, 16), (11, 14), and (14, 15). $Q'.Max = 16$.

Notice that there is no need to check $Q'(1).Pair$ and $Q'(2).Pair$. This is so because $Q'(1).Pair$ ($Q'(2).Pair$) is $\geq Q(1).Pair$ ($Q(2).Pair$) (we replaced one point with another with larger active path), and since $Q(1).Pair$ ($Q(2).Pair$) $\geq Q.Max \geq Q'.Max$, they cannot lead to any minimal combined point. So next, we check $Q'(3).Pair$, which is 17. This leads to a combined point (7, 17) which is no better than the one we had before ((7, 17)), hence it is not a minimal combined point. We continue with $Q'(4).Pair = 14$, and generate a new minimal combined point, (11, 16). Since $Q'(4).Pair$ is $\leq Q'.Max$, we continue our process by replacing the point that sets the current bound, i.e., (7, 16), with its next point, (8, 11).

The new tuple $Q''$ now consists of the points (0, 0), (6, 12), (8, 11), (11, 14), and (14, 15). $Q''.Max$ is now equal to 15. Again, there is no need to check $Q''(i).Pair$ for $i < 4$ (by the same argument as above). We find $Q''(4).Pair$ to be 14, which is $< Q''.Max$. Since the new point (11, 15) has lower cost than that of the previous point (11, 16), we replace the previous point with this new point. We now continue the process by replacing the point that sets the bound, i.e., (14, 15) with its next point. However, there is no next point in that set, and the process is thus completed. The minimal combined points are [(0, 19), (5, 18), (7, 17), (11, 15)].

We now discuss the implementation of the algorithm and its complexity. The algorithm proceeds in a bottom-up fashion. The information collected at each vertex is either the best DP-matching of the tree rooted at that vertex (in case this tree is even), or a set of minimal points for that tree (in case it is odd). Since the whole tree is even, the information gathered at the root will be the best DP-matching for the tree. It is sufficient to consider the actions taken at an arbitrary vertex $v$. We assume that the sets of minimal points (each set corresponds to one child of $v$) are given to us each in increasing order of active paths. We describe only the case of $d$ odd; the even case is simpler.

Let $Q$ denote the current tuple, and $Q.Max$ the maximal cost of its points. Initially, $Q$ consists of the first minimal point from each set (i.e., the minimal point with the smallest active path). We first sort the points in the tuple according to their active paths. Let $i$ denote the index of the child that we currently try to use as an active path ($i$ is initially 1). For a given $Q$ and $i$ we find $Q(i).Pair$ (by pairing the smallest active path with the largest active path, the second smallest with the second largest, and so on). This step requires $O(d)$ time since the minimal points in the tuple are sorted according to their active paths (we will have to make sure this sorted order is maintained throughout the algorithm). If $Q(i).Pair > Q.Max$, we check whether the combined point ($q_i.AP$, $Q(i).Pair$) is a new combined minimal point by comparing it to the previous combined minimal point and add it to the list of minimal points if it is. (Notice that the values of the $q_i.AP$'s are increasing; hence, we collect the set of combined minimal points in increasing order of active paths.) We then increment $i$ and continue (without changing $Q$). If $i$ becomes larger than $d$ we terminate.

The harder case is when $Q(i).Pair \leq Q.Max$. We again check to see whether the combined point $(q_i.AP, Q.Max)$ is a new combined minimal point and add it if it is. We then replace the point $p_j^k$ whose cost is $Q.Max$ with the next point $p_{j+1}^k$ from the same set (if there is no next point in that set, we are done). We put $p_{j+1}^k$ in $Q$ in the appropriate place so that $Q$ remains in sorted order of active paths. Therefore, $p_{j+1}^k$ may not be in the same position in $Q$ as $p_j^k$ was. Putting $p_{j+1}^k$ in its right position in $Q$ can be done in $O(\log d)$ steps by using a balanced tree data structure. In the same time complexity we can also update $Q.Max$. We now have a new $Q$, and we continue in the same way. Notice that $i$ is not changed when $Q$ is changed.

*Time Complexity.* We now prove that the complexity of the algorithm is $O(n^2)$, where $n$ is the number of vertices in the tree. Since each point can add at most one new combined minimal point (and each leaf can add one minimal point), the number of minimal points associated with any vertex is bounded by the number of descendants of that vertex. The complexity of the algorithm for one vertex $v$ is bounded by $O(dD)$ where $D$ is the number of descendants of $v$ and $d$ is the degree of $v$. This is so because the most expensive step involving one particular $Q$ and $i$ is the computation of $Q(i).Pair$ with is $O(d)$ ($d$ is the size of the tuple $Q$). Since $\sum_{i=1}^{n} d_i \leq n$, and $D_i \leq n$, we have $\sum_{i=1}^{n} d_i D_i \leq n^2$. Thus, the overall worst-case complexity is $O(n^2)$.

**4. The DP-Matching Problem for General Graphs is NP-Complete.** In this section, we prove that the min-max DP-matching problem and a variation of a discrete multicommodity flow problem are NP-complete.

The *discrete multicommodity flow* problem (DMF) is the following. Given an undirected graph $G = (V, E)$ and a set of source-sink pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, determine whether there exists a set of $k$ vertex-disjoint paths, the $i$th of which connects $s_i$ to $t_i$. The DMF problem is NP-complete [2]. We are interested in a variation of this problem (denoted by DMF′), which is defined as follows. Let $G = (V, E)$ be a weighted graph, $K$ be an integer, and $(s_1, t_1), (s_2, t_2), (s_3, t_3), \ldots, (s_k, t_k)$ be a collection of distinct vertex pairs of $G$. The problem is to determine whether there exist $k$ *edge*-disjoint paths, the $i$th of which connects $s_i$ to $t_i$, such that the weight of each path is at most $K$. We do not allow the path from $s_i$ to $t_i$ (for any $i$) to include any $s_j$ or $t_j$ for $i \neq j$ (because the reduction to the DP-matching problem becomes easier); the problem remains NP-complete without this restriction.

THEOREM 1. *The DMF′ problem is NP-complete.*

PROOF. The proof is similar to the NP-completeness proof for the regular DMF problem. It is easy to see that DMF′ belongs to NP. We prove that DMF′ is NP-complete by a transformation from the satisfiability problem (SAT). Let the CNF expression $E = C_1 \cdot C_2 \cdot C_3 \cdot \ldots \cdot C_k$ be an arbitrary instance of SAT, where $C_i$ is a clause consisting of variables $x_1, x_2, \ldots, x_n$. We associate a pair of vertices

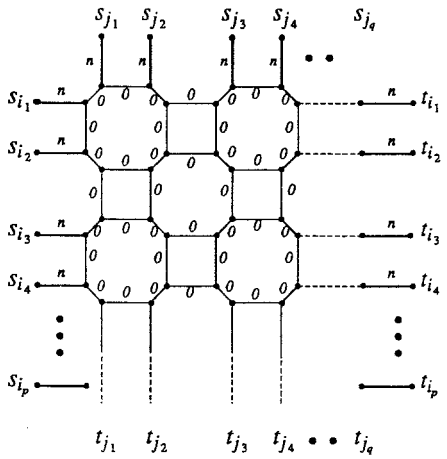


**Fig. 4.** A variable subgraph of $x_m$.

$(s_i, t_i)$ with each clause $C_i$. We call these vertices the *clause vertices.* For each variable $x_m$ we built a subgraph connecting the vertices corresponding to the clauses containing $x_m$ in the following way. If $x_m$ appears in clauses $C_{i_1}$, $C_{i_2}$, $C_{i_3}$, ..., $C_{i_p}$, and $\overline{x_m}$ appears in $C_{j_1}$, $C_{j_2}$, $C_{j_3}$, ..., $C_{j_q}$, we connect the corresponding clause vertices as is shown in Figure 4. All the edges incident to $s_i$ or $t_i$ are assigned weight $n$, and all the other edges are assigned weight 0. The new vertices forming the octagons in the middle are called *connecting vertices.* The subgraph is called a *variable subgraph.*

The variable subgraph is designed such that it is possible to join all the $s_{i_l}$'s to the corresponding $t_{i_l}$'s by edge-disjoint *horizontal* paths (paths that use no vertical edges), or to joint all the $s_{j_m}$'s to their corresponding $t_{j_m}$'s by edge-disjoint *vertical* paths (paths that use no horizontal edges). However, every horizontal path separates all the vertical paths and vice versa. Therefore, either the paths that connect clauses containing $x_m$ can be used, or paths that connect clauses containing $\overline{x_m}$ can be used. The graph for $E$ is obtained by adding the variable subgraphs for all variables (using new connecting vertices for each variable, but the same clause vertices) together.

We now prove that there is a truth assignment satisfying $E$ if and only if $G$ has $k$ edge disjoint paths, with cost at most $2n$, connecting the pairs of clause vertices. Given a truth assignment satisfying $E$, we obtain the edge-disjoint paths by selecting the horizontal paths in the subgraph corresponding to $x_m$ if $x_m$ is true, or the vertical paths otherwise. Since each clause must be satisfied, it is possible to connect the corresponding clause vertices. Furthermore, all horizontal and vertical paths have cost $2n$. On the other hand, suppose that $G$ has $k$ edge-disjoint paths of cost at most $2n$ connecting the $s_i$'s to the corresponding $t_i$'s. Then, any of these paths must use only one specific variable subgraph, since otherwise their cost would be more than $2n$. Since a subgraph allows only one type of path, connecting $s_i$ to $t_i$ determines the truth value of some variable that satisfies $C_i$.

(The variables whose truth assignment is not determined by the procedure above can be assigned arbitrarily.) It is clear that the transformation can be done in polynomial time. □

THEOREM 2. *Let $G = (V, E)$ be a weighted undirected graph with $2n$ vertices, and let $K$ be a positive integer. The problem of determining whether the cost of a min-max DP-matching for $G$ is at most $K$ is NP-complete.*

PROOF. It is easy to see that the problem belongs to NP. We prove that the problem is NP-complete by a transformation from DMF′. Let the graph $G = (V, E)$, with the designed vertex pairs $(s_1, t_1), (s_2, t_2), (s_3, t_3), \ldots, (s_k, t_k)$, and an integer $K$, be an arbitrary instance of the DMF′ problem. We construct a graph $G'$ from $G$ as follows. The construction for a vertex that is one of the vertices in the designated pairs is shown in Figure 5(a), and the construction for other (undesignated) vertices is shown in Figure 5(b). A designated vertex $s_i$ is augmented by adding $d_i$ *bridge vertices* together with $d_i$ *image vertices*, where $d_i$ is the degree of vertex $s_i$. The same construction is applied to the $t_i$'s, except that the weight of the additional edges are different. The construction involves four types of additional edges:

(1) edges connecting the $s_i$'s and their bridge vertices; their weight (denoted by $a$ in Figure 5) is $iM$, where $M$ is equal to the sum of all weights in $G$;
(2) edges connecting the $t_i$'s and their bridge vertices; their weight is $(2k - i)M$;
(3) edges connecting the bridge vertices and their image vertices; their weight (denoted by $b$ in Figure 5) is $(2kM + K)$; and

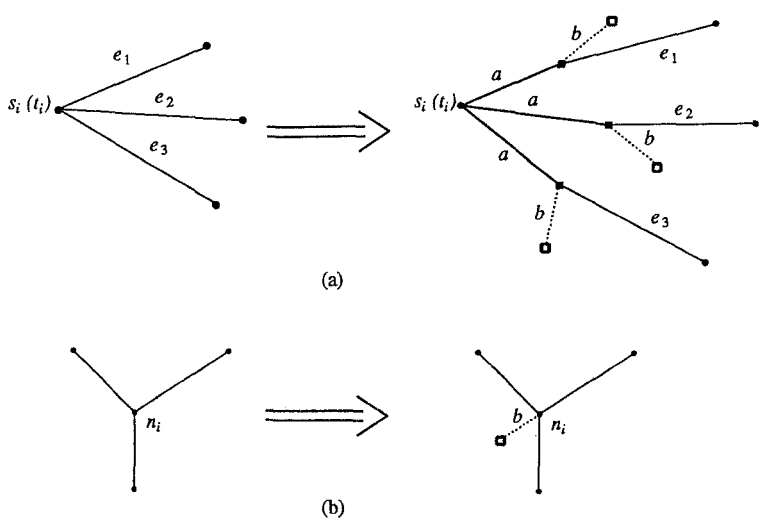



**Fig. 5.** The construction of $G'$ (the costs $a$ and $b$ are defined in the text). (a) Designated vertices, (b) undesignated vertices. ■, bridge vertex; □, image vertex.

(4) edges connecting undesignated vertices to their image vertex; their weight is also $(2kM + K)$.

The construction described above guarantees that $G'$ has a complete DP-matching bounded by $(2kM + K)$ if and only if $G$ has a set of edge-disjoint paths connecting the designated vertex pairs with weight $\leq K$. If $G$ has a set of paths, bounded by $K$, connecting the designated vertex pairs, then we can achieve a DP-matching, bounded by $(2kM + K)$, by matching the $s_i$'s to the $t_i$'s, the bridge vertices to their corresponding image vertices, and the undesignated vertices to their corresponding image vertices. Conversely, if there is a DP-matching bounded by $(2kM + K)$, then we first claim that image vertices cannot match designated vertices or other image vertices. This is so because the only edge coming out of an image vertex has already the maximal possible weight and we made sure that there are no zero-weight edges leading to designated vertices (there may be zero-weight edges in $G$). Since there is an equal number of image vertices and bridge vertices plus undesignated vertices, the match must be between them; that leaves the designated vertices to match among themselves. Next, we claim that $s_i$ must be matched with $t_i$, for all $i$ $1 \leq i \leq k$. The match involving each $s_i$ must contain an edge with weight $iM$, and the match involving each $t_i$ must contain an edge with weight $(2k - i)M$. Therefore, the sum of the weights in all the paths involves in the matching is at least $2k^2M$. The average weight of a path is at least $2kM$. If there is a path with weight $\leq (2k - 1)M$, then there must be another path with weight at least $(2k + 1)M > 2kM + K$, which is contrary to our goal. Therefore, all paths must have weights between $2kM$ and $(2k + 1)M$, which implies that each of the $s_i$ must be matched to its corresponding $t_i$. Since the paths between the $s_i$'s and the $t_i$'s are bounded by $(2kM + K)$ and mutually edge-disjoint, the corresponding paths in $G$ have costs bounded by $K$ and are mutually edge-disjoint. Thus, if $G'$ has a DP-matching bounded by $2kM + K$, then the original graph $G$ has $k$ edge-disjoint paths with cost bounded by $K$ the $i$th of which connects $s_i$ to $t_i$. The transformation can obviously be done in polynomial time. □

**Acknowledgment.** We thank Peter Downey for helpful discussions.

## References

[1] Gabow, H. N. and R. E. Tarjan, Algorithms for two bottleneck optimization problems, *J. Algorithms*, **9**, (1988), 411–417.

[2] Karp, R. M., On the complexity of combinatorial problems, *Networks*, **5** (1975), 45–68.

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

[4] Papadimitriou, C. H. and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[5] Wu S. and U. Manber, Algorithms for generalized matching, Technical Report, TR 88-39, Department of Computer Science, University of Arizona (November 1988).