Dynamic Matchings in Convex Bipartite Graphs

Gerth Stølting Brodal[1⋆], Loukas Georgiadis[2], Kristoffer Arnsfelt Hansen[3], and Irit Katriel[4⋆⋆]

[1] University of Aarhus, Århus, Denmark. gerth@daimi.au.dk
[2] Hewlett-Packard Laboratories, Palo Alto, CA, USA. loukas.georgiadis@hp.com
[3] University of Chicago, Chicago, IL, USA. arnsfelt@cs.uchicago.edu
[4] Brown University, Providence, RI, USA. irit@cs.brown.edu

Abstract. We consider the problem of maintaining a maximum matching in a convex bipartite graph $G = (V, E)$ under a set of update operations which includes insertions and deletions of vertices and edges. It is not hard to show that it is impossible to maintain an explicit representation of a maximum matching in sub-linear time per operation, even in the amortized sense. Despite this difficulty, we develop a data structure which maintains the set of vertices that participate in a maximum matching in $O(\log^2 |V|)$ amortized time per update and reports the status of a vertex (matched or unmatched) in constant worst-case time. Our structure can report the mate of a matched vertex in the maximum matching in worst-case $O(\min\{k \log^2 |V| + \log |V|, |V| \log |V|\})$ time, where k is the number of update operations since the last query for the same pair of vertices was made. In addition, we give an $O(\sqrt{|V|} \log^2 |V|)$-time amortized bound for this pair query.

1 Introduction

Let $G = (V, E)$ be a finite undirected graph. A subset $M \subseteq E$ of the edges is a *matching* if each vertex is incident with at most one edge in M. A matching of maximum cardinality is called a *maximum matching*. In this paper we study the problem of maintaining a *maximum matching* in a *convex bipartite* graph; G is bipartite if V can be partitioned to subsets X and Y such that $E \subseteq X \times Y$. A bipartite graph $G = (X \cup Y, E)$ is convex if there is a linear arrangement $y_1, \ldots, y_m$ of the nodes of Y such that the neighborhood of every node in X is an interval of the nodes of Y. Given a linear order for Y we can represent G in $O(|V|) = O(n+m)$ space (instead of $O(|V| + |E|)$), where $n = |X|$ and $m = |Y|$, since it suffices to specify the smallest and largest neighbor in Y for each $x \in X$.

Matchings in convex bipartite graphs correspond to a basic variant of the ubiquitous job scheduling problem. Namely, the case of unit-length tasks that need to be scheduled on a single disjunctive resource (i.e., one at a time) and

⋆ MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.
⋆⋆ Supported in part by NSF award DMI-0600384 and ONR Award N000140610607.

where we are given, for each job, a release time and a deadline. Another application for convex matching is in constraint programming, where filtering algorithms for the *AllDifferent* constraint [21] essentially identify edges in a bipartite graph which do not belong to any maximum matching. The vertices of X then represent variables of the program while the vertices of Y represent the values in their domains and the edges connect every variable with every value in its domain. In some cases, the domains are intervals and the graph is convex. In other cases, the constraint solver *assumes* that they are intervals as a heuristic that speeds up computations. Motivated by this application, and to avoid confusion, in the rest of the paper we call the vertices of X *variables* and the vertices of Y *values*. Furthermore, we sometimes refer to the interval of neighbors of a variable as its *domain*.

The problem of finding a maximum matching has a rich history [5]. Currently, the best upper bound for both bipartite and general matchings is randomized $O(|V|^\omega)$ [13], where ω is the exponent of matrix multiplication ($\omega < 2.376$ [2]). A simpler randomized algorithm with the same bound was presented recently [9]. The best deterministic bound is $O(|E|\sqrt{|V|})$ both for bipartite graphs [10] and general graphs [12].

Finding maximum matchings in convex bipartite graphs has also been extensively studied. Glover [7] showed that a maximum matching in such a graph can be found by traversing the values in increasing order and matching each value $y \in Y$ with the variable whose domain ends earliest, among those that have y in their domain and were not matched with smaller values, breaking ties according to a predetermined lexicographic order of the variables. Such a matching is called a *greedy matching*. This algorithm runs in $O(m + n \log\log n)$ time with the use of a fast priority queue [20]. Lipski and Preparata [11] presented an $O(n + m\alpha(m))$-time algorithm. (Here α is a functional inverse of Ackermann's function.) This running time was reduced to $O(n+m)$ in [4]. Further research [6, 18] aimed at decreasing the dependence on m, finally leading to an $O(n)$-time algorithm by Steiner and Yeomans [19].

In this paper we study the problem of maintaining a maximum matching in a dynamic convex bipartite graph. In the job scheduling view, this problem amounts to maintaining an optimal schedule for a dynamic set of unit-length jobs. Notice that Glover's algorithm can easily be adapted to operate in a dynamic setting where a job may be deleted or its release time and deadline may be available before the job is actually released. But still we cannot infer the service time of a given job (or even whether the job will eventually be serviced) before it is either actually dispatched or its deadline elapses.

Intuitively it is clear that maintaining a maximum matching dynamically must be hard; even a single edge insertion can change all the edges in the matching. Apparently, due to this difficulty, this problem has not received much attention. To the best of our knowledge, the only related previous work is presented in [16, 17]. In [16], Sankowski considers dynamic graphs, modified by edge insertions and deletions, that maintain the invariant that a *perfect matching* exists, i.e., all vertices are matched. He shows how to update the graph in $O(n^{1.495})$

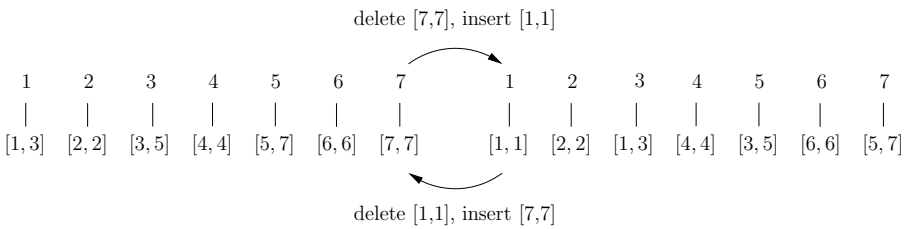

Fig. 1. Sequence of operations inducing linear number of changes in any explicit representation of a convex matching. Only edges in the maximum matching are shown.

time per operation and support queries asking whether a given edge is contained in any perfect matching. The same asymptotic update bound is achieved in [17] for maintaining the size of a maximum matching in general dynamic graphs (modified by edge insertions and deletions). The running time in both results depends on the value of the the exponent ω of matrix multiplication.

The situation is similar in the case of convex bipartite graphs; it is not hard to construct examples where a single update changes all the matching edges. Still, due to the special structure of the convex graphs it is less obvious that *any* representation of a maximum matching can change dramatically due to a small change in the input. For example, since the values in Y are ordered, a matching can be described by giving a corresponding order for X (together with a dummy variable ϵ for each unmatched value). To measure the difference between two convex matchings more accurately we can count the number of inversions between pairs of successive variables in the two corresponding orderings of X. Figure 1 shows that even with respect to this measure, the difference between two matchings after $O(1)$ updates can be $\Theta(|V|)$, also in the amortized sense. Such problematic cases rule out the possibility of maintaining any explicit representation of a maximum matching in sub-linear time, even for convex bipartite graphs. On the other hand, it is easy to verify that the sets of matched vertices in X (and Y) can change only by one element per update. (This fact holds also for general graphs modified by edge insertions and deletions, since a matching M is maximum iff there is no augmenting path relative to M [1].) The structure we develop is based on the previous observation and on a parallel algorithm of Dekel and Sahni for convex bipartite matchings [3]. We review their algorithm in Section 2.1.

2 Dynamic Convex Matchings

Recall that Y is totally ordered and the input graph $G = (X \cup Y, E)$ is represented succinctly by giving an interval $[s(x), t(x)]$, for each $x \in X$; $s(x)$ ($t(x)$) is the first (last) vertex in Y adjacent to x. For convenience, we identify the vertices in Y by their rank in the underlying linear arrangement, and henceforth assume

$Y = \{1, \ldots, m\}$. We allow G to be modified by a collection $\mathcal{U}$ composed of the following update operations:

- Insert a vertex x into X together with an interval $[s(x), t(x)]$; make x adjacent to every $y \in Y$ such that $s(x) \leq y \leq t(x)$.
- Delete a vertex x from X; remove all edges incident to x.
- Insert a vertex y into Y; make y adjacent to any $x \in X$ such that $s(x) < y < t(x)$.
- Delete a vertex y from Y if there is no $x \in X$ such that $s(x) = y$ or $t(x) = y$; remove all edges adjacent to y.

Clearly, the above operations maintain the convexity of G, and also maintain the given order of Y. As a result, we do not need to test if G remains convex after each operation (in fact we are not aware of any efficient dynamic algorithm for this test), but this also implies that we cannot support arbitrary edge updates. Restricted edge updates are possible by deleting a variable and re-inserting it with a different interval. Given a dynamic convex bipartite graph G that is modified subject to $\mathcal{U}$, we wish to support a collection $\mathcal{Q}$ of the following query operations:

- Given a vertex $x \in X$ or $y \in Y$, return the status *matched* or *unmatched* of this vertex.
- Given a matched vertex u ($u \in X \cup Y$), return its mate w in the maximum matching; the pair $\{u, w\}$ is an edge of the maximum matching.

Our results are summarized in the following theorems.

Theorem 1. *We can maintain a maximum matching of a convex bipartite graph $G = (X \cup Y, E)$, under the update operations in $\mathcal{U}$, in $O(\log^2 |X|)$ amortized time per update and $O(|Y| + |X| \log |X|)$ space. A status query can be answered in constant worst-case time. A pair query for a vertex u can be answered in $O(\min\{k \log^2 |X| + \log |X|, |X| \log |X|\})$ worst-case time, where k is the number of update operations that occurred since the last pair query for u or its mate.*

Theorem 2. *The amortized running time for a pair query is $O(\sqrt{|X|} \log^2 |X|)$. Also, there is an update sequence which forces our structure to spend $\Omega(\sqrt{|X|})$ amortized time for each pair query.*

2.1 The Dekel-Sahni algorithm

Let $S = \{s_1, \ldots, s_k\}$ be the set of start-points of the n variable domains ($s_i = s(x_i)$). The base structure in the algorithm of Dekel and Sahni [3] is a balanced binary search tree $\mathcal{T}$ over S. Each leaf s_i is associated with the interval of values $[s_i, s_{i+1} - 1]$; each internal node is associated with the interval of values formed by the union of the intervals of its children in $\mathcal{T}$. We denote the interval of values associated with a node P in $\mathcal{T}$ by *values*(P); we let $s(P)$ and $t(P)$ be, respectively, the smallest and largest value in *values*(P). We let *variables*(P) denote the set of variables $x \in X$ such that $s(x) \in$ *values*(P). Also, define

matched(P) to be the set of variables that belong to the greedy matching of the subgraph induced by *variables*$(P) \cup$ *values*(P). Let *transferred*(P) be the variables x in *variables*$(P) \setminus$ *matched*(P) such that $t(x) > t(P)$, i.e., their endpoints extend beyond *values*(P); these variables, although not matched at P, may become matched at an ancestor of P in $\mathcal{T}$ and thus participate in the final (global) matching, computed at the root of $\mathcal{T}$. The remaining variables in *variables*(P) form a set *infeasible*(P) and are discarded from later computations.

Now let P be an internal node of $\mathcal{T}$ with left child L and right child R. Assume that *matched*(L), *transferred*(L), *matched*(R) and *transferred*(R) were already computed. Dekel and Sahni show that *matched*(P) and *transferred*(P) can be computed as follows: Let *variables'*(R) = *transferred*$(L) \cup$ *matched*(R). First compute a greedy matching in the subgraph G'_R induced by *variables'*$(R) \cup$ *values*(R). Let *matched'*(R) be the subset of *variables'*(R) in that matching, and let *transferred'*$(R) = \{x \mid x \in$ *variables'*$(R) \setminus$ *matched'*(R) and $t(x) > t(R)\}$. Then, *matched*(P) = *matched*$(L) \cup$ *matched'*(R) and *transferred*(P) = *transferred*$(R) \cup$ *transferred'*(R). The key theorem of Dekel and Sahni [3, Theorem 2.1], shows that there is a simple way to compute *matched'*(R): It is safe to assume that the start-point of *every* interval in *variables'*(R) is $s(R)$. Although the matching produced this way may not be feasible in G'_R, the set of matched variables that are identified is correct, i.e., there is a greedy matching in this graph that matches exactly this set of variables.

After identifying the matched vertices in X, the algorithm of Dekel and Sahni performs a top-down phase that constructs a feasible matching. For each node P of $\mathcal{T}$, it constructs the set *matching*(P), containing the variables that are matched with a value in *values*(P). Although *matched*(P) is a subset of *variables*(P), this is not true, in general, for *matching*(P). Clearly, if P is the root of $\mathcal{T}$, *matching*(P) = *matched*(P). The *matching* sets for the children L and R of P are constructed by partitioning *matching*(P) as follows. Let W be the subset of *matching*(P) consisting of variables x such that $s(x) < s(L)$ or $x \in$ *matched*(P). (Note that W does not contain any variable in *transferred*(L).) If $|W| \leq |$*values*$(L)|$, then set *matching*$(L) = W$. Otherwise, pick $|$*values*$(L)|$ variables $x \in W$ with the smallest $t(x)$ and insert them in *matching*(L). The remaining variables are inserted into *matching*(R). Finally, we notice that for any leaf F, each $x \in$ *matching*(F) satisfies $s(x) \leq s(F)$. (Because we cannot have $s(F) < s(x) \leq t(F)$ for any $x \in X$.) Hence, sorting *matching*(F) by increasing right endpoint, gives us the matching at F; the variable with rank i in the sorted order is matched to the ith value.

2.2 Overview of the dynamic algorithm

We achieve an efficient dynamization of the Dekel-Sahni algorithm by showing (1) how to perform logarithmic-time local updates of the sets *matched*(P) and *transferred*(P), for each node $P \in \mathcal{T}$, and (2) that each operation in $\mathcal{U}$ requires a logarithmic number of local updates. These updates will allow us to report the status of any vertex in $O(1)$ worst-case time. The construction of this data structure is described in Section 3. In order to report the mate of a matched

vertex we also need to update the *matching* sets. These updates are performed just-in-time, as required by a pair query. We analyze the performance of this method in Section 4.

3 Data Structure supporting Status Queries

Here we present a structure supporting the update operations in $\mathcal{U}$ in $O(\log^2 n)$ amortized time, and answers status queries in $O(1)$ worst-case time. Due to the limited space, we only consider the case of the insertion of a new variable. The remaining update operations are performed in a similar manner. Our first building block is a structure which allows us to maintain a greedy matching in $O(\log n)$ time per update, for the special case where all intervals have the same starting point. We describe this in Section 3.1. Then, in Section 3.2, we show how an update can propagate bottom-up in $\mathcal{T}$ and give the new set of matched vertices in X. Given this set, we can update the set of matched vertices in Y in $O(\log n)$ time, using the same ideas as in Section 3.1. (Due to limited space we omit this last part.)

3.1 The Equal Start-Points Case

Suppose we wish to maintain a greedy matching when all intervals have the same start-point. That is, the input graph $G = (X \cup Y, E)$ is such that $s(x) = 1$ for all $x \in X$, and $Y = \{1, \ldots, m\}$. We say that an $x \in X$ has *higher priority* than an $x' \in X$ if $t(x) < t(x')$. Define n_i as the number of variables whose interval is $[1, i]$. Also define a_i to be the number of matched variables $x \in X$, such that $t(x) \leq i$. These quantities satisfy the recurrence $a_i = \min\{a_{i-1} + n_i, i\}$, for $1 \leq i \leq m$ and $a_0 = 0$. Inserting (deleting) a variable x with domain $[1, k]$ amounts to incrementing (decrementing) n_k. Let n'_i (a'_i) be the value of n_i (a_i) after the update (insertion or deletion of $[1, k]$).

Consider the insertion of the variable $[1, k]$. For the given k we have $n'_k = n_k + 1$ and for $i \neq k$, $n'_i = n_i$. Let j be the smallest integer such that $j \geq k$ and $a_j = j$, if such a j exists, and let $j = m + 1$ otherwise. If $j = k$ then all the first k values are matched with variables that have the same priority as x or higher, and nothing changes. Otherwise, $a_k < k$ and the new variable will be matched; this happens because we always maintain a greedy matching and $[1, k]$ has higher priority than any interval that ends after k. Thus, $a'_i = a_i + 1$ if $k \leq i < j$, and $a'_i = a_i$ otherwise. The remaining update operations are analyzed similarly. In every update operation, we are interested in the quantities $b_j = j - a_j - n_{j+1}$ for all $j \in Y$. When we insert a variable with domain $[1, k]$, we need to locate the first $j \geq k$ with $a_j = j$. Then, by the definition of a_j we have $a_{j-1} + n_j \geq j$, so $j' - a_{j'} - n_{j'+1} \leq -1$ for $j' = j - 1$. Similar inequalities are derived for the remaining update operations.

We construct a data structure that maintains the b_i quantities implicitly. This implicit representation is necessary because a single update can change many b_i's. The data structure consists of a balanced binary search tree T (e.g.,

a red-black tree [8]) over Y with each leaf corresponding to a value. Leaf i stores n_i and a number $\hat{b}_i$; an internal node v stores the numbers $add(v)$ and $min(v)$. We define these quantities later. Given a node or leaf u and an ancestor v of u in T, define $sum(u, v)$ to be the sum of $add(w)$ for all nodes w on the tree path from u to v, excluding v. Given a node or leaf u, define $sum(u)$ to be $sum(u, root)$, where $root$ is the root of T. Let $leaves(v)$ be the set of leaves contained in the subtree rooted at v. The numbers stored in the tree will satisfy the invariants: (a) $b_j = \hat{b}_j + sum(j)$, and (b) $min(v) = \min\{\hat{b}_j + sum(j, v) \mid j \in leaves(v)\}$. Combining (a) and (b) we get $\min\{b_j \mid j \in leaves(v)\} = \min\{\hat{b}_j + sum(j, v) \mid j \in leaves(v)\} + sum(v) = min(v) + sum(v)$. Initially $add(v) = 0$ for all nodes $v \in T$. As T is modified after an update operation the values $add(v)$ keep track of the changes that have to be applied to the affected b_j values. By its definition $add(v)$ affects the b_j values of all the leaves $j \in leaves(v)$. Also, $\hat{b}_j$ is simply the value of b_j before the update. We note that the numbers stored at the internal nodes are easy to maintain in constant time for each restructuring operation of the balanced tree (e.g., rotation).

We sketch how T is updated when we insert a variable with domain $[1, k]$. After locating leaf k we increment n_k. Since this affects b_{k-1}, we increment $\hat{b}_{k-1}$ and update $min(v)$ for the ancestors v of $k-1$ in T bottom-up. As we ascend the path toward the root we calculate $sum(k-1, v)$ at each node v, which takes constant time per node. Next we locate the first leaf $j \geq k$ such that $b_j < 0$. To that end we perform a top-down search on the appropriate subtree of T. Note that as we descend the tree we can compute $sum(v)$ at each node v that we visit, and thus compute $min(v) + sum(v)$ (in constant time per node). So, let $P = (v_0 = root, v_1, \ldots, k)$ be the search path for k in T. Let $P' = (u_1, u_2, \ldots)$ be the subsequence of P composed of the nodes v_i such that v_{i+1} is the left child of v_i; let $r_{i'}$ be the right child of $u_{i'}$. Then, j is located at the subtree rooted at r_{i^*} where i^* is the largest index i' that satisfies $min(r_{i'}) + sum(r_{i'}) < 0$. We can locate r_{i^*} in $O(\log m)$ time. Then the search for j proceeds top-down in the subtree of r_{i^*} as follows: Let v be the current node, and let u be its left child. If $min(u) + sum(u) < 0$ then the search continues in the subtree of u. Otherwise, we move to the right child of v. The last step is to subtract one from b_i, for $k \leq i \leq j$. Since the involved leaves may be too many, we perform the decrements implicitly using the add quantities. Let η be the nearest common ancestor of k and j in T; we can locate η easily in $O(\log m)$ time since T is balanced. Let P_k be the path from η to k, excluding the end-points, and let P_j be the path from η to j excluding the end-points. Let v be a node on P_k such that its right child u is not on P_k. If u is a leaf then decrement $\hat{b}_u$; otherwise decrement $add(u)$. Perform the symmetric steps for P_j (visiting the left children of nodes on this path). Finally, decrement $\hat{b}_k$ and $\hat{b}_j$ and update min for all $O(\log m)$ nodes on P_k, P_j and on the path from $root$ to η; each update takes constant time if we process the paths bottom-up.

Notice that by storing at each leaf i the variables x with $t(x) = i$, we can locate a variable that is replaced in the maximum matching by the newly inserted variable. This is a variable stored in the leaf j that satisfies $\alpha_j = j$ and $j \geq k$.

Similarly, after a deletion we can locate a variable that replaces the deleted one in the maximum matching. We will use this capability in the next section. Also, we note that by grouping successive empty leaves (i.e., leaves j with $n_j = 0$) we can reduce the number of leaves in T to $\min\{m, n\}$. Thus, we get:

Lemma 1. *Updates of variables and values in the equal start-points case take worst-case $O(\log n)$ time.*

3.2 The Complete Data Structure

The complete data structure maintains the *matched* and *transferred* sets of all nodes of $\mathcal{T}$. Its building block is the special-case structure of Section 3.1 which maintains these sets, assuming that all start-points of *variables* are equal. By [3, Theorem 2.1], such a data structure can be used to maintain the matching *matched'*(R) for every internal node and the complete matching for every leaf. Hence, each node P in $\mathcal{T}$ is associated with such a local structure T_P, as follows. First, the interval $[s(R), t(R)]$ of available values for matching with *variables'*(R) is translated to $[1, s(R)-t(R)+1]$, and every variable in *variables'*(R) is assumed to have start-point equal to one. Also, each such variable with end-point $t(x) > t(R)$ is stored in the leaf of T_P that corresponds to value $s(R)-t(R)+1$. When we have an update in *variables'*(R) we use the structure T_P to update *matched'*(R) accordingly. As noted in Section 3.1, this process will also return the variables that enter or leave *matched'*(R), so we can propagate the changes towards the root of $\mathcal{T}$.

For simplicity, we assume that the base tree $\mathcal{T}$ is fixed (i.e., we do not insert and delete leaves). Given this, we show that the global data structure can be updated upon the insertion or deletion of a variable or value in $O(\log^2 n)$ time, by proving the next lemma.

Lemma 2. *Each operation in $\mathcal{U}$ causes $O(\log n)$ operations on local data structures of the nodes of $\mathcal{T}$.*

The fully-dynamic data structure can be obtained by applying standard techniques, e.g., maintaining $\mathcal{T}$ as a (weight-balanced) BB[α]-tree [14, 15]. As a result, however, our update bounds become amortized.

Insertion of a variable. Assume that we insert a new variable x with domain $[s, t]$ $(s, t \in Y)$. Then s determines the leaf F of $\mathcal{T}$ where the variable is inserted. After inserting x into T_F, the sets *matched*(F) and *transferred*(F) may change. The changes to these sets need to be propagated to the parent of F in $\mathcal{T}$, in turn causing new changes to the parent of the parent of F, and so on; potentially, changes occur at all nodes on the path from F to the root of $\mathcal{T}$. We will show that at each node, only a constant number of update operations is necessary.

Consider the two possible results of inserting x to the leaf F. The first is that x entered *matched*(F) and, possibly, some other variable x' left *matched*(F). The insert operation on T_F will report if x has entered *matched*(F) and will return x' if it exists. If this x' exists, it either entered *transferred*(F) or not,

depending on its end-point, which can be tested in constant time. The second case is that x did not enter *matched*(F), and depending on its end-point either entered *transferred*(F) or not. In both cases, the information that needs to be propagated to the parent of F in $\mathcal{T}$ is a triplet of the form (a, b, c), where each of a and b is either a variable or the symbol ϵ which is a place-holder that represents nothing, and c is either *transferred* or *infeasible*. Such a triplet is interpreted to mean "a is inserted into *matched*(F), b is deleted from *matched*(F) and inserted into c". Note that a and b are not necessarily distinct. For instance, the case in which the inserted variable x is not matched and is inserted into *transferred*(F) is encoded as $(x, x, \textit{transferred})$.

We now show by induction on the height of the nodes in $\mathcal{T}$, that the information that needs to be propagated from an internal node to its parent can also be represented as a triplet of this form. To that end, let P be a node of $\mathcal{T}$ with left child L and right child R. By the induction hypothesis, P received a triplet (a, b, c) from one of its children, indicating the changes that have occurred at this child. We analyze the two possible cases separately.

First, assume that it was the left child who sent the triplet (a, b, c). This implies a was inserted into *matched*(L) and b was removed from *matched*(L) and inserted into c. The effects of these changes on *matched*(P) and *transferred*(P) are as follows. The first part of *matched*(P) is simply *matched*(L), so it changes in the same manner. The second half is the matching obtained from *variables'*(R). If c is equal to *infeasible*, neither of these sets has changed and the changes that P needs to report to its parent are $(a, b, \textit{infeasible})$. If c is equal to *transferred*, then b may belong to the second part of *matched*(P), possibly replacing some other variable b' which was matched there before. Again, notice that b' is returned after the update operation on T_P. Now, the important point to note is that P does not need to report anything about b to its parent; b was and remains in *matched*(P). The changes that P needs to report are summarized in the triplet (a, b', c'), where c' is *transferred* or *infeasible*, depending on the end-point of b'.

Now, assume that the triplet (a, b, c) was reported to P from its right child. In this case a was inserted into *matched*(R) and may or may not need to be inserted into *matched*(P). If not, then b was not in *matched*(P) before and will not be in *matched*(P) afterwards; since a replaced b in *matched*(R), we know that its end-point is smaller than b's, hence, it cannot be that $b \in$ *matched*(P) if a was not inserted. Therefore, P does not report any change to its parent. On the other hand, if a was inserted into *matched*(P), then if b was in *matched*(P) before the update it becomes unmatched, and if b was not in *matched*(P) before the update then some other variable b' may have become unmatched to make space for a. In the first case, the change reported by P to its parent is the triplet (a, b, c') where c' depends on the end-point of b. In the second case, P sends the triplet (a, b', c') where c' depends on the end-point of b', and b' is found during the update in T_R. It is important to note that even if b was inserted into *transferred*(R) after the update, it is not inserted into *transferred*(P). This is implied by the fact that b is not in *matched*(P): If b is in *transferred*(P) after the update then b must have already been in *transferred*(P) before the update.

4 Pair Queries

Now we augment the structure of Section 3 in order to support the pair query. As Figure 1 suggests, we cannot afford to change all the affected *matching* sets after an update operation. (In this example we would need to change $O(n)$ such sets.) However, we can get a more efficient solution, based on the following observation:

Observation 3 *An update operation in $\mathcal{U}$ can change only one member of a matching(P) set, for any $P \in \mathcal{T}$.*

A pair query will need to visit the nodes of a root-to-leaf path $\mathcal{T}_q$ of $\mathcal{T}$, and our plan is to update the *matching*(P) sets of the nodes P on this path. If the query vertex is $y \in Y$ then $\mathcal{T}_q$ is simply the path from the root to the leaf corresponding to y. Otherwise, the query vertex is $x \in X$ and $\mathcal{T}_q$ is discovered as we decide at each node P if x belongs to *matching*(L) or *matching*(R), where L and R are the children (left and right respectively) of P. To make this decision fast, we store at node P a list $\ell(P)$ representing the set W defined in Section 2.1. This list is sorted by the end-points $t(x)$ of the variables $x \in W$. A variable $x \in$ *matching*(P) belongs to *matching*(L) iff it appears in $\ell(P)$ and its rank in W is at most $|$*values*$(L)|$. Otherwise, $x \in$ *matching*(R). Hence the decision of whether to follow the left or the right child of P can be made in $O(\log n)$ time if the lists *matching*(P) and $\ell(P)$ are updated. To handle the updates in *matching*(P) (and $\ell(P)$), node P also maintains a list of variables *update*(P), which keeps track of the operations that need to be performed to bring *matching*(P) back to date. This list is arranged in the order the variables were inserted. The meaning of $x \in$ *update*(P) is that x has to be inserted into *matching*(P) if it does not already appear there, otherwise x has to be deleted. Notice that we can construct *matching*(P) in $O(|$*values*$(P)|)$ time, using the lists maintained at its parent in $\mathcal{T}$. Therefore, in our analysis we can assume that the size of *update*(P) is $O(|$*values*$(P)|)$. We use this assumption in the amortized analysis in Section 4.1.

Now consider what happens after an update operation. Observation 3 implies that at most one variable has to be inserted into *matching*(P) and at most one variable has to be deleted. If P is the root of $\mathcal{T}$ then these are exactly the changes in *matched*(P). Hence, the updates can be performed immediately at the root. Let L and R be, respectively, the left and right child of P. Using $\ell(P)$ we can find the variables that change in *matching*(L) and *matching*(R) and insert them into *update*(L) and *update*(R), all in $O(\log n)$ time. (At most two variables are inserted in each of these lists.) To update *matching*(P) for any non-root node of $\mathcal{T}$, we process the variables in *update*(P) one by one. Each variable can be processed in the same way an update for the root of $\mathcal{T}$ was performed. After each update we insert the appropriate variables in the *update* lists of the children of P in $\mathcal{T}$, and so the changes propagate downwards the tree. As noted earlier, the time spent per variable in *update*(P) is $O(\log n)$, and every *update* list contains a number of variables at most twice the number of update operations in $\mathcal{U}$. To get a bound on the pair query that is always $O(n \log n)$, we note that we can rebuilt the whole structure for $\mathcal{T}$ in $O(n \log n)$ time. If we do that after $O(n/\log n)$ total updates, each *update* list will have $O(n/\log n)$ variables at any time. Rebuilding

$\mathcal{T}$ results to an additional $O(\log^2 n)$ amortized cost for the updates in $\mathcal{U}$, so the asymptotic update bound is not affected and the bounds stated in Theorem 1 follow. Next we sketch the proof of Theorem 2.

4.1 Amortized cost of pair queries

Lower bound. We begin with $\mathcal{T}$ being a full binary tree with n leaves, where each leaf stores a unique variable: The ith leaf stores a variable with domain $[i, n]$. For simplicity we take $m = n$. Next we insert $\sqrt{n}$ variables with domains $[1, j]$, $j = 1, \ldots, \sqrt{n}$. These variables will shift the matching by $\sqrt{n}$ positions, and the last $\sqrt{n}$ variables will become infeasible. Consider the $\sqrt{n}$ nodes of $\mathcal{T}$ at height $\log \sqrt{n}$. For each such node P_i we make a pair query for a leaf in P_i's subtree. The corresponding query paths intersect only above the P_i's. To get the claimed bound it suffices to count the work done at each P_i. When the pair query algorithm visits P_i, $update(P_i)$ will have $\Theta(\sqrt{n})$ variables, so it will spend $\Omega(\sqrt{n})$ time to process these variables. Hence the total work for all nodes at height $\log \sqrt{n}$ is $\Omega(n)$. By deleting the variables $[1, j]$ and reinserting them, we can repeat this process $\sqrt{n}$ times. This implies an amortized cost of $\Omega(\sqrt{n})$ for a pair query.

Upper bound. For simplicity, we consider $\mathcal{T}$ to be a full binary tree with n leaves, each storing a single variable. Informally, having more variables in some leaves cannot increase the asymptotic cost bound because both the height of the tree decreases and also each pair query updates the matching for more than one variable. We analyze the worst case running time of a sequence of operations composed of ν updates, followed by μ pair queries. This will give us the worst-case scenario, because the total cost for querying a set of paths is maximized after all updates are performed. At any node P, processing $update(P)$ takes $O(2^i \log n)$ if the height i of P is at most $\log \nu$, and $O(\nu \log n)$ for $i > \log \nu$. We calculate separately the cost for processing the nodes at height above $\log \nu$ (top subtree) and below $\log \nu$ (bottom subtrees). First, for the top subtree of $\mathcal{T}$ we observe that each pair query visits $\log n/\nu$ of its nodes and at each node it spends at most $O(\nu \log n)$ time. Hence, if $\mu \leq n/\nu$, the total cost is $O(\mu\nu \log n/\nu \log n) = O(\mu\nu \log^2 n)$, so the amortized cost is $O(\frac{\mu\nu}{\nu+\mu} \log^2 n) = O(\sqrt{n} \log^2 n)$. For $\mu > n/\nu$, the total cost is $O(n \log n + \mu \log n)$, because the top subtree has $O(n/\nu)$ nodes. So, in this case, the amortized cost is $O(\frac{n}{\nu+\mu} \log n + \frac{\mu}{\nu+\mu} \log n) = O(\frac{n\nu}{\nu^2+n} \log n + \log n)$, which is $O(\sqrt{n} \log n)$. We now turn to the bottom subtrees. The contribution of a path inside a bottom subtree is $O(\nu \log n)$. Hence, if $\mu \leq n/\nu$, the total cost is $O(\mu\nu \log n)$, which gives $O(\sqrt{n} \log n)$ amortized cost. The total contribution of a bottom subtree to a pair query is $O(\nu \log n)$, because the total number of variables in all *update* lists is $O(\nu)$. Since we have $O(n/\nu)$ bottom subtrees, for $\mu > n/\nu$ the total cost is $O(n \log n + \mu \log n)$, i.e., $O(\sqrt{n} \log n)$ amortized cost.

Acknowledgement We thank Willem-Jan van Hoeve for useful references.

References

1. C. Berge. Two theorems in graph theory. *Proc. Nat. Acad. Sci. U.S.A.*, 43:842–844, 1957.
2. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.
3. E. Dekel and S. Sahni. A parallel matching algorithm for convex bipartite graphs and applications to scheduling. *Journal of Parallel and Distributed Computing*, 1:185–205, 1984.
4. H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–21, 1985.
5. Z. Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Comput. Surv.*, 18(1):23–38, 1986.
6. G. Gallo. An $O(n \log n)$ algorithm for the convex bipartite matching problem. *Operations Research Letters*, 3(1):31–34, 1984.
7. F. Glover. Maximum matching in convex bipartite graphs. *Naval Research Logistic Quarterly*, 14:313–316, 1967.
8. L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. on Foundations of Computer Science*, pages 8–21, 1978.
9. N. J. A. Harvey. Algebraic structures and algorithms for matching and matroid problems. In *Proc. 47th IEEE Symp. on Foundations of Computer Science*, pages 531–542, 2006.
10. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.
11. W. Lipski and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.
12. S. Micali and V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximal matching in general graphs. In *Proc. 21st IEEE Symp. on Foundations of Computer Science*, pages 17–27, 1980.
13. M. Mucha and P. Sankowski. Maximum matchings via gaussian elimination. In *Proc. 45th IEEE Symp. on Foundations of Computer Science*, pages 248–255, 2004.
14. J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. In *Proc. 4th ACM Symp. on Theory of Computing*, pages 137–142, 1972.
15. J. Nievergelt and C. K. Wong. Upper bounds for the total path length of binary trees. *Journal of the ACM*, 20(1):1–6, 1973.
16. P. Sankowski. Dynamic transitive closure via dynamic matrix inverse. In *Proc. 45th IEEE Symp. on Foundations of Computer Science*, pages 509–517, 2004.
17. P. Sankowski. Faster dynamic matchings and vertex connectivity. In *Proc. 18th ACM-SIAM Symp. on Discrete Algorithms*, pages 118–126, 2007.
18. M. G. Scutellà and G. Scevola. A modification of Lipski-Preparata's algorithm for the maximum matching problem on bipartite convex graphs. *Ricerca Operativa*, 46:63–77, 1988.
19. G. Steiner and J. S. Yeomans. A linear time algorithm for determining maximum matchings in convex, bipartite graphs. *Computers and Mathematics with Applications*, 31(12):91–96, 1996.
20. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.
21. W.-J. van Hoeve. The AllDifferent Constraint: A Survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.