Fast Scheduling of Weighted Unit Jobs with Release Times and Deadlines

C. Greg Plaxton*

University of Texas at Austin
1 University Station C0500
Austin, Texas 78712–0233
plaxton@cs.utexas.edu

Abstract. We present a fast algorithm for the following classic scheduling problem: Determine a maximum-weight schedule for a collection of unit jobs, each of which has an associated release time, deadline, and weight. All previous algorithms for this problem have at least quadratic worst-case complexity. This job scheduling problem can also be viewed as a special case of weighted bipartite matching: each job represents a vertex on the left side of the bipartite graph; each time slot represents a vertex on the right side; each job is connected by an edge to all time slots between its release time and deadline; all of the edges adjacent to a given job have weight equal to the weight of the job. Letting U denote the set of jobs and V denote the set of time slots, our algorithm runs in $O(|U| + k \log^2 k)$ time, where $k \leq \min\{|U|, |V|\}$ denotes the cardinality of a maximum-cardinality matching. Thus our algorithm runs in nearly linear time, a dramatic improvement over the previous quadratic bounds.

1 Introduction

We address the classic scheduling problem in which the input is a collection of jobs, each with an associated release time, deadline, and weight, and our objective is to schedule a maximum-weight subset of the jobs. For the reader who is familiar with Graham's notation for scheduling problems, this problem is denoted $(1 \mid r_j;\ p_j = 1 \mid \sum w_j U_j)$, where U_j is a 0-1 variable indicating whether job j is successfully scheduled. A number of special cases and variants of this scheduling problem have been studied for many years; see the scheduling survey of Graham et al. [8] for additional pointers to the early literature in this area. Various well-known algorithms textbooks include a detailed treatment of the special case in which all jobs are released at time zero, for which it is relatively straightforward to design a nearly linear time algorithm [1, pp. 207–214], [3, pp. 399–401], [9, pp. 161–168].

The unweighted version of the above scheduling problem is equivalent to a special case of bipartite matching called "convex" bipartite matching. Since we find it convenient to present our results in the matching framework, we now develop the necessary definitions. A subset A of a totally ordered set $(S, \leq)$ is said to be *convex* if the following condition holds for all x, y, and z in S: If x and y belong to A and $x \leq z \leq y$, then

* Supported by NSF Grants ANI–0326001 and CCF–0635203.

z belongs to A. A *convex bipartite graph* is a bipartite graph together with a bipartition (U, V) of the vertices, and a total order $\leq$ on the vertices of V, such that the neighbors of any vertex u in U form a convex subset of V.

Glover presented a simple greedy algorithm [7] for maximum-cardinality convex bipartite matching that admits an $O(|V|+|U|\log|U|)$-time implementation using an elementary priority queue data structure. Later, van Emde Boas used a fast priority queue to obtain an $O(|V| + |U|\log\log|U|)$-time implementation of Glover's algorithm [15]. Lipski and Preparata [11] used Tarjan's fast union-find data structure [14] to devise a different algorithm running in time $O(|U|+|V|\alpha(|V|))$, where α is a functional inverse of Ackermann's algorithm. Gabow and Tarjan [5] show that this application of union-find falls into a category admitting a linear-time implementation, thereby reducing the Lipski-Preparata time bound to $O(|U| + |V|)$. Another line of work focused on eliminating the dependence of the running time on $|V|$ [6, 12]. This research culminated in the $O(|U|)$-time algorithm of Steiner and Yeomans [13].

The scheduling problem considered in this paper corresponds to the weighted variant of convex bipartite matching in which each vertex of U has an associated weight, and the weight of any edge is given by that of its endpoint in U. It is worth remarking that other weighted variants of the convex bipartite matching problem can be contemplated. One possibility is to allow each edge to have an arbitrary weight, but since the complete bipartite graph is convex, this weighted variant is equivalent to general weighted bipartite matching. A weighted variant of convex bipartite matching that is incomparable to the one considered in the present paper is obtained by associating a weight with each vertex in V, and defining the weight of each edge as that of its endpoint in V. For this "right-weighted" (i.e., V-weighted) variant, Katriel [10] has recently obtained an $O(|E| + |V|\log|U|)$-time algorithm to find a maximum-weight matching. Since the input size is $\Theta(|U| + |V|)$ words, and $|E|$ could be as large as $\Theta(|U| \cdot |V|)$, this algorithm has quadratic complexity. Another weighted variant of convex bipartite matching — this time more general than the one considered in the present paper — is obtained by associating a weight with each vertex in $U \cup V$, and defining the weight of each edge as the sum of the weights of its two endpoints. If the input graph admits a U-perfect matching, Katriel's $O(|E|+|V|\log|U|)$-time algorithm can be used to find a maximum-weight $|U|$-perfect matching. Since many bipartite graphs — including, for example, all bipartite graphs such that $|U| > |V|$ — do not admit a U-perfect matching, Katriel's algorithm addresses only a special case of this weighted variant.

Throughout the remainder of this paper, we focus on the "left-weighted" (i.e., U-weighted) variant of convex bipartite matching that corresponds to scheduling weighted unit jobs with release times and deadlines. Any left-weighted (or right-weighted) bipartite matching instance is well-known to form a matroid, where the independent sets of the matroid are those subsets U' of vertices on the left such that there exists a matching M that matches every vertex in U'. Thus, left-weighted convex bipartite matching can be addressed using the framework of the matroid greedy algorithm. Indeed, all efficient algorithms for left-weighted convex bipartite matching that have been proposed to date, including the algorithm presented in this paper, exploit this framework. In the next paragraph, we describe a simple $O(|U|\log|U| + |U| \cdot |V|)$-time algorithm of this sort. This algorithm, which we refer to in this paper as the *greedy algorithm*, produces a matching

that we call the *greedy matching*. The fast matching algorithm presented in this paper, which we refer to as the *hierarchically greedy algorithm*, maintains a hierarchical representation of a collection of matchings that includes the greedy matching. We establish the correctness of the hierarchically greedy algorithm by relating its behavior to that of the greedy algorithm.

Here is a description of the greedy algorithm. First, we sort the vertices in U in nonincreasing order of weight, initialize an independent set Z to the empty set, and initialize a matching M to the empty matching. Then, we iteratively attempt to grow the independent set Z by considering each successive vertex u in U (according to the sorted order previously determined) and adding it to Z if the resulting set remains independent. In order to simplify the task of determining whether u can be added to Z, we maintain the invariant that M is the greedy matching of Z that is produced by Glover's unweighted convex bipartite matching algorithm. Glover's algorithm attempts to match the vertices v in $|V|$, from lowest to highest, using a natural "earliest deadline" rule that the vertex in Z matched to v is the as yet unmatched vertex adjacent to v (if any) that has the smallest number of remaining opportunities to be matched. (If a tie occurs, it is broken according to a fixed ordering of the vertices in U.) Using a naive representation of the greedy matching M in the form of an array of length V, in $O(|V|)$ time we can determine whether Glover's algorithm successfully matches all vertices in $I \cup \{u\}$, and if so, update Z and M appropriately. Upon termination of the greedy algorithm, the greedy matching M is a maximum-weight matching.

Lipski and Preparata [11] use the matroid greedy framework to develop a left-weighted convex bipartite matching algorithm which, while somewhat different from the greedy algorithm described above, has a similar time complexity of $O(|U|^2 + |U| \cdot |V|)$. Dekel and Sahni [4] present a parallel algorithm for left-weighted convex bipartite matching that uses $O(|U|^2)$ processors and $O(\log^2 |U|)$ time, and which is based on a sequential algorithm with $O(|U|^2)$ complexity.

In this paper, we introduce a data structure that maintains a hierarchical representation of a collection of matchings that includes the greedy matching. This data structure allows us to implement each iteration of the matroid greedy algorithm in amortized polylogarithmic time. As a result, we obtain a nearly linear time algorithm for left-weighted convex bipartite matching. The remainder of this paper is organized as follows. Section 2 contains some preliminary definitions. Section 3 presents an efficient dynamic data structure for a special class of convex bipartite graphs. Section 4 presents our "hierarchically greedy" matching algorithm. Section 5 discusses the time complexity of this algorithm. Section 6 offers some concluding remarks.

2 Preliminaries

In this section, we specify the formal representation of a convex bipartite graph, or CBG, that will be used throughout the remainder of the paper. We also define certain special kinds of CBGs.

Instead of working with "jobs" and "time slots", we find it convenient to introduce more abstract types "ping" and "pong", which we define as follows. A *pong* is an element of some totally ordered universe, such as the integers. A *ping* u is charac-

terized by four attributes: pongs $u.\textit{first}$ and $u.\textit{last}$, a positive weight $u.\textit{weight}$, and a unique integer ID $u.\textit{id}$. We define three total orders over the set of all pings: under the *first-ID total order*, ping u is at most u' if $(u.\textit{first}, u.\textit{id})$ is lexicographically at most $(u'.\textit{first}, u'.\textit{id})$; under the *last-ID total order*, ping u is at most u' if $(u.\textit{last}, u.\textit{id})$ is lexicographically at most $(u'.\textit{last}, u'.\textit{id})$; under the *weight-ID total order*, ping u is at most u' if $(u.\textit{weight}, u.\textit{id})$ is lexicographically at most $(u'.\textit{weight}, u'.\textit{id})$. We primarily make use of the last-ID total order. For this reason, we adopt the convention that all ping comparisons are resolved according to the last-ID total order unless stated otherwise.

A pair (U, V), where U is a set of pings and V is a set of pongs, represents a CBG as follows: (1) we identify each ping in U with a vertex on the LHS; (2) we identify each pong in V with a vertex on the RHS; (3) there is an edge from ping u to pong v if and only if $u.\textit{first} \leq v \leq u.\textit{last}$.

A CBG (U, V) is *proper* if $\{u.\textit{first}, u.\textit{last}\} \subseteq V$ for all pings u in U. A CBG (U, V) is *simple* if $|U| \geq |V|$ and $v \leq u.\textit{last}$ for all pings u in U and all pongs v in V. A CBG (U, V) is *nice* if $|U| \leq |V| + 1$ and (U, V) is simple and admits a matching of cardinality $|V|$. A nice CBG (U, V) with $|U| = |V|$ is said to be *in-kilter*; otherwise, it is *out-of-kilter*.

3 A Dynamic Data Structure for Nice CBGs

In this section, we develop a dynamic data structure for maintaining a nice CBG subject to a collection of six operations. Three of these operations are applicable when the nice CBG is in an in-kilter state; the other three are applicable in out-of-kilter states.

The three in-kilter operations are as follows. The first is *pingAdd*(u), where ping u does not belong to U and $(U \cup \{u\}, V)$ is simple; this operation adds the ping u to U. The second is *pongDrop*(v), where pong v belongs to V; this operation removes the pong v from V. The third is *print*$()$, which prints out a perfect matching of (U, V).

The three out-of-kilter operations are as follows. The first is *pongAdd*(v), where pong v does not belong to V and $(U, V \cup \{v\})$ is simple and admits a perfect matching; this operation adds the pong v to V. The second is *pingDrop*(u), where ping u belongs to U and $(U \setminus \{u\}, V)$ admits a perfect matching; this operation removes the ping u from U. The third is *pingDrop*$()$, which takes no arguments; this operation removes from U the maximum ping u in U such that $(U \setminus \{u\}, V)$ admits a perfect matching.

Notice that the precondition of each of the above operations ensures that the CBG remains nice. Our goal is to implement the *print*$()$ operation in linear time, and each of the other five operations in logarithmic time. The only significant challenge is to implement the *pingDrop*$()$ operation efficiently. To do so, we first find it useful to introduce a few definitions and lemmas related to simple CBGs.

For any simple CBG (U, V), we define the following auxiliary functions: (1) let $A(U, V)$ denote $\{u.\textit{first} \mid u \in U\} \cup V$; (2) for any pong v in $A(U, V)$, let $f(U, V, v)$ denote the number of pings u in U such that $v \leq u.\textit{first}$ minus the number of pongs v' in V such that $v \leq v'$; (3) let $g(U, V)$ denote the maximum, over all pongs v in $A(U, V)$, of $f(U, V, v)$ (if $A(U, V)$ is empty, then $g(U, V)$ is zero). The proof of the following lemma is straightforward and is omitted.

Lemma 1. *For any simple CBG (U, V), the size of a maximum cardinality matching is $|U| - g(U, V)$.*

For any simple CBG (U, V) such that $A(U, V)$ is nonempty, let us define $h(U, V)$ as the maximum pong in $A(U, V)$ such that $g(U, V) = f(U, V, v)$. The proof of the next lemma follows easily from Lemma 1.

Lemma 2. *For any simple CBG (U, V) such that $|U| > |V|$, and any ping u in U, the size of a maximum cardinality matching of (U, V) is equal to that of $(U \setminus \{u\}, V)$ if and only if $h(U, V) \leq u$.*first.

Lemma 2 leads to the following useful characterization of the set of pings over which the maximization occurs in the definition of *pingDrop*().

Lemma 3. *For any out-of-kilter nice CBG (U, V), and any ping u in U, $(U \setminus \{u\}, V)$ admits a perfect matching if and only if $h(U, V) \leq u$.*first.

Lemma 3 suggests a two-phase approach for implementing *pingDrop*(): (1) compute the pong $h(U, V)$; (2) compute the maximum ping u in U such that $h(U, V) \leq u.\textit{first}$.

We now discuss how to implement the first phase in logarithmic time. By Lemma 1, for any out-of-kilter nice CBG (U, V), we have $g(U, V) = 1$, and hence $h(U, V)$ is the maximum pong in $A(U, V)$ such that $f(U, V, v) = 1$. To implement the first phase, we maintain an augmented red-black tree with a node for each pong v in $A(U, V)$; the key of each node is the associated pong. We augment a node x by maintaining the following three auxiliary fields: an integer "count" field equal to $|\{u \in U \mid u.\textit{first} = v\}| - \Delta$, where v is the key of node x and Δ is equal to 1 if v belongs to V, and 0 otherwise; an integer "sum" field that is equal to the sum of all count fields in the nodes of the subtree rooted at x; an integer "maximum suffix sum" field that is equal to the maximum, over all suffixes (including the empty suffix) of the key-ordered sequence of nodes in the subtree rooted at x, of the sum of the counts in the suffix. It is straightforward to argue that all of these fields can be maintained in logarithmic time whenever a ping is added to or removed from U, and whenever a pong is added to or removed from V. Furthermore, given the augmented red-black tree structure that we have just described, it is straightforward to determine the maximum pong in $A(U, V)$ such that $f(U, V, v) = 1$ in logarithmic time, and hence to implement the first phase in logarithmic time.

To implement the second phase, we maintain a second augmented red-black tree with a node for each ping in U. The nodes are sorted according to the first-ID ordering, that is, the key associated with the node for a ping u is $(u.\textit{first}, u.\textit{id})$. We augment each red-black tree node with a "max" field equal to the maximum ping (with respect to the last-ID ordering) in the nodes of the corresponding subtree. It is straightforward to maintain the max fields in logarithmic time whenever a ping is added to or removed from U. Furthermore, given this tree, and the pong threshold $h(U, V)$ determined in the first phase, it is straightforward to implement the second phase in logarithmic time.

As remarked in the preceding paragraphs, it is straightforward to maintain our two augmented red-black trees in logarithmic time whenever the nice CBG is modified via an operation that adds or drops a ping or pong. It remains to describe how to implement

the *print*() operation in linear time. One simple approach is to maintain a third red-black tree containing the pongs of V. A perfect matching of an in-kilter nice CBG can then be obtained by matching each ping in the second augmented red-black tree described above (the one sorted by the first-ID ordering) to the pong of equal rank in the third tree. In fact, it is not necessary to maintain such a third red-black tree, because we can use inorder traversals of the first and second red-black trees to produce a sorted list of the pongs in V in linear time. Thus our data structure for maintaining a dynamic nice CBG consists of just two augmented red-black trees. (Remark: The implementation of the *print*() operation described above does not, in general, produce the greedy matching. If we wish to produce the greedy matching, we can do so in $O(|U| \log |U|)$ time via a suitable linear sequence of calls to the *pongDrop*(v) and *pingDrop*() operations. To avoid modifying the data structure, we can first create a copy in linear time.)

4 A Hierarchically Greedy Algorithm

In this section we present a hierarchically greedy algorithm for computing a maximum-weight matching of a given CBG (U, V). It is convenient to assume that (U, V) is proper. In the context of establishing the upper bound of Theorem 1, our assumption that (U, V) is proper is made without loss of generality, since we can easily preprocess (U, V) in $O(|U| \log |U| + |V| \log |V|)$ time to obtain an equivalent CBG — with respect to the maximum-weight matchings — that is proper. The preprocessing phase removes all pings with degree zero, and for each of the remaining pings u, assigns $u.\textit{first}$ to the minimum pong v in V such that $u.\textit{first} \leq v$, and assigns $u.\textit{last}$ to the maximum pong v in V such that $v \leq u.\textit{last}$.

Like the greedy algorithm described in Section 1, our hierarchically greedy algorithm is based on the framework of the matroid greedy algorithm. As such, the algorithm iterates through the pings in decreasing order with respect to the weight-ID ordering. While the greedy algorithm maintains a specific matching — the greedy matching — at each iteration, our hierarchically greedy algorithm maintains a representation of a collection of matchings that includes the greedy matching. We say that an iteration of the greedy algorithm is successful if it adds a ping to the greedy matching; otherwise, it is unsuccessful. It turns out that, at any given iteration, all of the matchings in the collection maintained by the hierarchically greedy algorithm induce the same set of matched pings and pongs. It follows that an iteration of the hierarchically greedy algorithm successfully inserts the current ping into the set of matched pings if and only if the corresponding iteration of the greedy algorithm is successful.

It remains to describe how the hierarchically greedy algorithm performs the insertion attempt associated with a general iteration of the matroid greedy algorithm. It is straightforward to prove that the attempt to insert a ping u in a given iteration is unsuccessful if and only if there exist pongs v and v' in V such that $v \leq u.\textit{first} \leq u.\textit{last} \leq v'$ and the number of pings u' previously successfully inserted for which $v \leq u.\textit{first} \leq u.\textit{last} \leq v'$ is equal to the number of pongs in V that belong to the interval $[v, v']$. We refer to such an interval of pongs $[v, v']$ as a *tight interval*, and we say that a pong in V is *tight* if it belongs to some tight interval. Initially, none of the pongs in V are tight, but as more pings are successfully inserted, certain pongs become (and remain) tight.

Our hierarchically greedy algorithm maintains a conservative estimate (i.e., a subset) of the current set of tight pongs in V. The pongs associated with this estimate are said to be *marked tight*. Once a pong is marked tight, it continues to be marked tight thereafter. When we attempt to insert a ping u, we first determine whether all of the pongs v in V such that $u.\mathit{first} \leq v \leq u.\mathit{last}$ are marked tight. If so, we conclude that the insertion is unsuccessful, and proceed to the next iteration. We call such an unsuccessful insertion *good* because it is relatively inexpensive to process; other unsuccessful insertions are said to be *bad*.

4.1 An Augmented Binary Search Tree

Our hierarchical greedy algorithm makes use of an augmented BST with $|V|$ nodes, one for each pong in V. The key of each node α in the augmented BST, which is denoted $\alpha.\mathit{key}$, is the associated pong. Since V is not updated through the course of the algorithm, the structure of the augmented BST is static. In analyzing the performance of the algorithm, we assume only that the augmented BST has depth $O(\log |V|)$. We maintain several additional fields for each node α in the augmented BST: $\alpha.\mathit{left}$, which is a pointer to the left child of α ($\alpha.\mathit{left} = 0$ if there is no left child); $\alpha.\mathit{right}$, which is a pointer to the right child of α ($\alpha.\mathit{right} = 0$ if there is no right child); $\alpha.\mathit{parent}$, which is a pointer to the parent of α ($\alpha.\mathit{parent} = 0$ if α is the root); $\alpha.\mathit{keyMin}$, which is equal to the minimum, over all nodes β in the subtree rooted at α, of $\beta.\mathit{key}$; $\alpha.\mathit{size}$, which is equal to the total number of nodes in the subtree rooted at α; $\alpha.\mathit{occupant}$, which is of type "pointer to node", and which either points to some ancestor of α, or takes on one of two special values 0 and $+\infty$; $\alpha.\mathit{occupantMax}$, which is equal to the "maximum", over all nodes β in the subtree rooted at α, of $\beta.\mathit{occupant}$. In order to make the latter definition precise, we need to specify how to determine the maximum of two node pointers. We consider the special pointer value $+\infty$ to be the highest possible pointer value, and the special pointer value 0 to be the lowest possible pointer value. To compare two pointers to actual nodes α and β, we instead compare $\alpha.\mathit{size}$ with $\beta.\mathit{size}$, breaking ties arbitrarily.

Each node of the augmented BST also contains a nice CBG that is represented by two augmented red-black trees as discussed in Section 3. These nice CBGs are such that the total size of the augmented BST remains linear throughout.

As indicated earlier, the structure of the augmented BST is static. All occupant fields (and hence also the occupantMax fields) are initialized to $+\infty$ to indicate that all of the pongs are unmatched. The nice CBG associated with each node is initialized to the empty CBG. After initialization, the only attributes of a node that are subject to modification are the occupant and occupantMax fields, and the associated nice CBG.

4.2 Key Invariant

Due to space limitations, our proof of correctness is not included in this extended abstract. Instead we simply state a key invariant of the data structure, which holds in any *quiescent state*, that is, before and after any insertion attempt. The invariant completely characterizes the state of the augmented BST in terms of the set of pongs in V that have been marked tight and the state of the greedy algorithm after the same number of

insertion attempts. To define the invariant, we consider executing the greedy and hierarchically greedy algorithms side by side, one insertion attempt at a time. In any quiescent state, we define a mapping from the edges of the greedy matching to the nodes of the augmented BST as follows: Map edge (u, v) to the LCA of the node with key v and the node with key $u.last$. We claim that the set of pings (resp., pongs) of the nice CBG associated with any node α is exactly the set of ping (resp., pong) endpoints of the edges of the greedy matching that are mapped to node α. The preceding claim characterizes the set of $|V|$ nice CBGs associated with the nodes of our augmented BST in this quiescent state. It remains only to characterize the values of the occupant fields, since the occupant values determine the occupantMax values, and all of the other augmented BST fields are static. For any pong v in V, let α denote the augmented BST node with $\alpha.key = v$. If pong v is unmatched in the greedy matching, then $\alpha.occupant = +\infty$. If pong v is matched in the greedy matching and is marked tight, then $\alpha.occupant = 0$. If v is matched to ping u in the greedy matching and is not marked tight, then $\alpha.occupant$ is a pointer to the augmented BST node to which edge (u, v) is mapped, i.e., to the LCA of the node with key v and the node with key $u.last$.

The aforementioned invariant asserts a close correspondence between the successive quiescent states of the greedy and hierarchically greedy algorithms. The main idea underlying our full proof of correctness is to extend this correspondence to non-quiescent states. We do so by starting with the greedy algorithm and transforming it into the hierarchically greedy algorithm via a sequence of code transformations. A suitable correspondence is established between each pair of successive algorithms in this sequence.

4.3 Methods

In this section we describe the methods supported by the node object of the augmented BST introduced in Section 4.1.

The print method is invoked on the root node at the end of the hierarchically greedy algorithm in order to output a maximum-weight matching. When invoked on the root node, the print method traverses each node α of the augmented BST, and invokes the print operation of Section 3 on the nice CBG associated with α. Each of these $|V|$ print operations produces a piece of the overall output matching. The invariant stated in Section 4.2 ensures that the nice CBG associated with any augmented BST node is in-kilter in any quiescent state, and hence the nice CBG print operation is applicable in any such state. The overall running time of the print method is $O(|U| + |V|)$. (If we wish to produce the greedy matching, then we need to print the greedy matching of each nice CBG; by the analysis of the print operation in Section 3, this increases the running time by an $O(\log k)$ factor, where $k \leq \min\{|U|, |V\}$ denotes the maximum number of pings/pongs in any of the nice CBGs.)

The second major node method is insert. Pseudocode for the insert method is provided below. Each of the $|U|$ insertion attempts performed by the hierarchically greedy algorithm corresponds to an invocation of this method at the root of the augmented BST. The lone argument of the insert method is the ping associated with the current insertion attempt. The insert method is defined in terms of three other node methods: add, tight, and tighten. We discuss each of these three methods below.

```
insert(Ping u)
    Pong v := u.first
    if tight(v, u.last) = 0 then
        Node *p := add(u, v)
        if p ≠ 0 then
            tighten(v, p →key)
```

The tight method takes two pong arguments v and v', and returns a boolean value indicating whether every pong v'' in V such that $v \leq v'' \leq v'$ has been marked tight. By exploiting the occupantMax field of the augmented BST, this method is easy to implement in $O(\log |V|)$ time. But to establish our best time bound for left-weighted convex bipartite matching, we need a more efficient implementation of the tight method, so we maintain a separate union-find data structure [14]. The idea is to break the sorted sequence of pongs in V into maximal contiguous subsequences of pongs in which either: (1) all of the pongs in a subsequence have been marked tight, or (2) none of the pongs in a subsequence have been marked tight. Each such subsequence corresponds to a single set in our union-find data structure. Along with each set, we store a bit indicating whether the pongs in the set have been marked tight. The roots of the sets are linked together in a doubly-linked list, which is kept in sorted order based on the pong values. To implement the tight method, we simply look up the two pong arguments in the union-find data structure to determine whether they both belong to the same set, and if so, whether that set consists of pongs that have been marked tight. In the foregoing pseudocode for the insert method, we use a call to the tight method to determine whether to immediately reject a given insertion attempt.

The tighten method takes two pong arguments v and v'. For every pong v'' in V such that $v \leq v'' \leq v'$, this method marks v'' tight by ensuring that the occupant field in the node with key v'' is equal to zero (i.e., if it was not already zero, this method sets it to zero), and makes any necessary adjustments to the occupantMax fields. This method also performs corresponding unions to the union-find data structure mentioned in the preceding paragraph. (The sorted doubly-linked list of roots in the union-find data structure enables us to perform these union operations efficiently.) By exploiting the occupantMax field (i.e., there is no need for this method to descend into a subtree rooted at a node with occupantMax field equal to zero), we can easily execute an invocation of the tighten method in $O((k+1) \log |V|)$ time, where k denotes the number of nonzero occupant fields that are set to zero as a result of the invocation. In the above pseudocode for the insert method, we invoke the tighten method on the root node of the augmented BST to update our data structures after a bad unsuccessful insertion attempt. Accordingly, the quantity k in the preceding expression is guaranteed to be positive on any invocation of the tighten method. Furthermore, once we set the occupant field of some node to zero in tighten, it remains zero thereafter. It follows from the foregoing remarks that the total cost of all invocations of the tighten method is $O(|V| \log |V|)$.

The pseudocode for the add method is given below. If the insertion attempt associated with the add invocation is successful, add returns 0. Otherwise, add indicates failure by returning a pointer to a node in the augmented BST; this pointer is used to determine the second pong argument of the subsequent invocation of the tighten method. Recall that each augmented BST node has an associated nice CBG. In the pseudocode

for the add method, the calls to *pingAdd*(u) and *pingDrop*(u) act on the associated nice CBG. The same holds for the calls to *pongAdd*(v), *pingDrop*$()$, and *pingAdd*(u) appearing in the pseudocode for the resolve method below, and for the calls to *pongDrop*(v) and *pongAdd*(v) appearing in the pseudocode for the drop method below. Our proof of correctness establishes that each call acting on the associated nice CBG satisfies the required precondition specified in Section 3. The add method is defined in terms of the node method resolve, which we discuss next.

```
Node *add(Ping u, Pong v)
  if occupantMax = 0 then
    return this
  else if v > key then
    return right→add(u, v)
  else if u.last < key then
    return left→add(u, v)
  pingAdd(u)
  p := resolve(v)
  if p ≠ 0 then
    pingDrop(u)
  return p
```

The pseudocode for the resolve method is given below. Like the add method, resolve returns 0 on success, and a pointer to a node on failure. The resolve method is defined in terms of the node methods add, drop, occupy, and search. We have already defined the add method; drop, occupy, and search are defined below.

```
Node *resolve(Pong v)
  Node *p, *q := search(v)
  if q ≠ 0 then
    Pong v := q →key
    Node *r := q →occupant
    p := if r = +∞ then 0 else r →drop(v)
    if p = 0 then
      pongAdd(v)
      occupy(q)
  else
    Ping u := pingDrop()
    p := if key = u.last then this else right→add(u, right→keyMin)
    if p ≠ 0 then
      pingAdd(u)
  return p
```

When the search method is invoked at a node α, it takes as argument a pong v that is either equal to $\alpha.key$, or is equal to $\beta.key$ for some node β in the left subtree of α. The search method returns a pointer to a node, determined as follows. Let S denote the set of all nodes β in the subtree rooted at α such that $v \leq \beta.key \leq \alpha.key$, and $\beta.occupant$ is either $+\infty$ or the address of a proper ancestor of α. If S is empty, then 0 is returned.

Otherwise, a pointer to the node β in S minimizing $\beta.key$ is returned. By making use of the occupantMax field, it is straightforward to implement the search method in time proportional to the depth of the subtree rooted at node α.

When the occupy method is invoked at a node α, it takes as argument a nonzero pointer to a node returned by a just-completed call to the search method at node α. As such, the argument of the occupy method is a pointer to some β in the subtree rooted at node α such that $\beta.occupant$ is either $+\infty$ or the address of a proper ancestor of α. The occupy method sets $\beta.occupant$ to point to α, and then updates occupantMax fields as necessary, starting at β and repeatedly "bubbling up" to the parent until the occupantMax field is unchanged or we attempt to move to the parent of the root. The running time is proportional to the depth of the subtree rooted at node α.

The pseudocode for the drop method is given below. Like the add method, drop returns 0 on success, and a pointer to a node on failure. The drop method is defined in terms of the resolve method discussed above.

```
Node *drop(Pong v)
    pongDrop(v)
    Node *p := resolve(v)
    if p ≠ 0 then
        pongAdd(v)
    return p
```

5 Analysis

Due to space limitations, we are not able to include our analysis of the running time of the hierarchically greedy algorithm in this extended abstract. In the full version of the paper, we prove the following theorem. The proof uses separate arguments to bound the total cost of the successful, good unsuccessful, and bad unsuccessful insertions.

Theorem 1. *The hierarchically greedy algorithm of Section 4 computes a maximum-weight matching of a given CBG (U, V) in $O(|U| \log |U| + |V| \log^2 |V|)$ time using $O(|U| + |V|)$ space.*

With some extra preprocessing work, we can further improve the running time of the algorithm of Section 4. By employing a preprocessing phase based on the $O(|U|)$-time unweighted convex bipartite matching algorithm of Steiner and Yeomans [13], the $O(|V| \log^2 |V|)$ term can be improved to $O(k \log^2 k)$, where $k \leq \min\{|U|, |V|\}$ denotes the size of a maximum-cardinality matching. The following theorem can then be obtained via an $O(|U| + k \log^2 k)$-time preprocessing phase that discards all but $O(k \log k)$ of the pings in U, while ensuring that we can still produce a maximum-weight matching using the remaining pings. The complete details of the two preprocessing phases are nontrivial and are provided in the full paper.

Theorem 2. *A maximum-weight matching of a proper CBG (U, V) can be computed in $O(|U| + k \log^2 k)$ time and $O(|U| + |V|)$ space, where $k \leq \min\{|U|, |V|\}$ denotes the size of a maximum-cardinality matching.*

Notice that in order to achieve the time bound of Theorem 2 we need to assume — as in the work of Steiner and Yeomans [13] — that the input CBG (U, V) is proper.

6 Concluding Remarks

Recently, Brodal et al. [2] have designed a data structure based on the Dekel-Sahni algorithm for the problem of maintaining an unweighted maximum matching in a dynamic convex bipartite graph. It allows for vertices to be inserted or deleted from either U or V in $O(\log^2 |U|)$ amortized time. The interface supported by the data structure includes a constant-time "status query" that can be used to determine whether a given vertex is in the current maximum matching. Also, a "pair query" is provided that takes as argument a matched vertex and returns the current match of that vertex. The amortized cost of a pair query is shown to be $O(\sqrt{|U|}\log^2 |U|)$ and $\Omega(\sqrt{|U|})$. We plan to investigate whether the techniques of the present paper can be used to obtain improved bounds for some of the dynamic operations considered in [2].

References

1. G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, Englewood Cliffs, NJ, 1996.
2. G. S. Brodal, L. Georgiadis, K. A. Hansen, and I. Katriel. Dynamic matchings in convex bipartite graphs. In *Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science*, pages 406–417, August 2007.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, Cambridge, MA, 2nd edition, 2001.
4. 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.
5. 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*, 30:209–221, 1985.
6. G. Gallo. An $O(n \log n)$ algorithm for the convex bipartite matching problem. *Operations Research Letters*, 3:31–34, 1984.
7. F. Glover. Maximum matching in convex bipartite graphs. *Naval Research Logistic Quarterly*, 14:313–316, 1967.
8. R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, pages 287–326, 1979.
9. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, New York, 1978.
10. I. Katriel. Matchings in node-weighted convex bipartite graphs. *INFORMS Journal on Computing*. Published online in *Articles in Advance*, December 2007; print version to appear in 2008.
11. W. Lipski, Jr. and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.
12. 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.
13. G. Steiner and J. S. Yeomans. A linear time algorithm for determining maximum matchings in convex, bipartite graphs. *Computers and Mathematics with Applications*, 31:91–96, 1996.
14. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.
15. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.