Bipartite Matching with Linear Edge Weights[*]

Nevzat Onur Domaniç [†] Chi-Kit Lam [†] C. Gregory Plaxton [†]

October 2016

Abstract

Consider a complete weighted bipartite graph G in which each left vertex u has two real numbers *intercept* and *slope*, each right vertex v has a real number *quality*, and the weight of any edge (u, v) is defined as the intercept of u plus the slope of u times the quality of v. Let m (resp., n) denote the number of left (resp., right) vertices, and assume that $m \geq n$. We develop a fast algorithm for computing a maximum weight matching (MWM) of such a graph. Our algorithm begins by computing an MWM of the subgraph induced by the n right vertices and an arbitrary subset of n left vertices; this step is straightforward to perform in $O(n \log n)$ time. The remaining $m - n$ left vertices are then inserted into the graph one at a time, in arbitrary order. As each left vertex is inserted, the MWM is updated. It is relatively straightforward to process each such insertion in $O(n)$ time; our main technical contribution is to improve this time bound to $O(\sqrt{n} \log^2 n)$. This result has an application related to unit-demand auctions. It is well known that the VCG mechanism yields a suitable solution (allocation and prices) for any unit-demand auction. The graph G may be viewed as encoding a special kind of unit-demand auction in which each left vertex u represents a unit-demand bid, each right vertex v represents an item, and the weight of an edge (u, v) represents the offer of bid u on item v. In this context, our fast insertion algorithm immediately provides an $O(\sqrt{n} \log^2 n)$-time algorithm for updating a VCG allocation when a new bid is received. We show how to generalize the insertion algorithm to update (an efficient representation of) the VCG prices within the same time bound.

[*]This research was supported by NSF Grant CCF–1217980.

[†]Department of Computer Science, University of Texas at Austin, 2317 Speedway, Stop D9500, Austin, Texas 78712–1757. Email: {onur, geocklam, plaxton}@cs.utexas.edu

1 Introduction

Given an undirected graph $G = (V, E)$, a *matching* of G is a subset M of E such that no two edges in M share an endpoint. If G is a weighted graph, we define the weight of a matching as the sum of the weights of its constituent edges. The problem of finding a maximum weight matching (MWM) of a weighted bipartite graph, also known as the "assignment problem" in operations research, is a basic and well-studied problem in combinatorial optimization. A classic algorithm for the assignment problem is the Hungarian method [10], which admits an $O(|V|^3)$-time implementation. For dense graphs with arbitrary edge weights, this time bound remains the fastest known. Fredman and Tarjan [4] introduce Fibonacci heaps, and by utilizing this data structure to speed up shortest path computations, they obtain a running time of $O(|V|^2 \log |V| + |E| \cdot |V|)$ for the maximum weight bipartite matching problem. When the edge weights are integers in $\{0, \ldots, N\}$, Duan and Su [3] give a scaling algorithm with running time $O(|E|\sqrt{|V|} \log N)$. In this paper, we consider a restricted class of complete weighted bipartite graphs where the edge weights have a special structure. Both unweighted and weighted matching problems in restricted classes of bipartite graphs have been studied extensively. Gabow and Tarjan [5], Glover [6], Katriel [9], Lipski and Preparata [12], Steiner and Yeomans [17] study matching problems in convex bipartite graphs, the graphs in which the right vertices can be ordered in such a way that the neighbors of each left vertex are consecutive. Plaxton [13] studies vertex-weighted matchings in two-directional orthogonal ray graphs, which generalize convex bipartite graphs.

In the present paper, we develop a fast algorithm for computing an MWM of a complete weighted bipartite graph with the following special structure: there are m left vertices, each of which has two associated real values, a "slope" and an "intercept"; there are n right vertices, each of which has an associated real "quality"; for each left vertex u and right vertex v, the weight of edge (u, v) is given by the slope of u times the quality of v plus the intercept of u. Since the weight of any edge (u, v) is determined by evaluating the linear function specified by u (via the slope and intercept) on the quality of v, we refer to this problem as *bipartite matching with linear edge weights*. Assuming that $m \geq n$, we solve this problem in $O(m\sqrt{n} \log^2 n)$ time. We begin by solving the problem on a subgraph induced by the n right vertices and an arbitrary subset of n left vertices; this turns out to be easy to accomplish in $O(n \log n)$ time via sorting. We then insert the remaining left vertices one at a time, in arbitrary order. As each left vertex is inserted, we update the solution in $O(\sqrt{n} \log^2 n)$ time. It is relatively straightforward to process each such insertion in $O(n)$ time, yielding an overall $O(mn)$ time bound. Our algorithm provides a significant improvement over the latter bound, which is the fastest previous result that we are aware of.

In recent work that is closely related to the current paper, Domaniç and Plaxton [2] present a fast algorithm for bipartite matching with linear edge weights in the special case where the qualities of the right vertices form an arithmetic sequence. Assuming that $m \geq n$, their algorithm runs in $O(m \log m)$ time. Applying that algorithm to the scheduling domain directly solves the problem of scheduling unit jobs on a single machine with a common deadline where each job has a weight and a profit, and the objective is to minimize the sum of the weighted completion times of the scheduled jobs plus the sum of the profits of the rejected jobs. Domaniç and Plaxton [2] also provide an extension that preserves the $O(m \log m)$ time bound for the special case where the qualities correspond to the concatenation of two arithmetic sequences. This extension solves a more general scheduling problem that incorporates weighted tardiness penalties with respect to a common due date into the objective.

By removing the technical restrictions on the qualities imposed in [2], the algorithm of the present paper supports a richer edge weight structure, while continuing to admit a compact graph representation that uses space linear in the number of vertices. In terms of scheduling, the present algorithm addresses a broader class of problems than [2]; for example, it can handle symmetric earliness and tardiness penalties with respect to a common due date, and allows certain time slots to be marked as unavailable. Below we discuss another motivation for the present work, which is based on its connection to unit-demand auctions.

In a unit-demand auction of a collection of items, each bidder submits a bid that specifies a separate offer on each item, which may or may not be equal to the private valuation that the bidder has for that item [1]. The outcome of the unit-demand auction is a pricing of the items and an allocation of each bidder to at most one item. In mechanism design, it is known that the VCG mechanism is the only mechanism for unit-demand auctions that achieves the desired properties of being efficient, strategyproof, and envy-free [7, 11]. Such an auction can be modeled as a bipartite graph in which each left vertex represents a bid, each right vertex represents an item, and the weight of the edge from a bid u to an item v represents the offer of the bid u on item v. Then, a VCG allocation corresponds to an MWM of such bipartite graph, and the VCG prices correspond to the dual variables computed by the Hungarian method, i.e., they correspond to the prices having the minimum sum among the ones that are the solutions to the dual of the linear program that solves the assignment problem encoding the auction.

The main motivation for our interest in the problem we consider in this paper, given the aforementioned desirable properties of the VCG mechanism, is to find frameworks to encode unit-demand auctions that are expressive enough to have suitable applications while being restrictive enough to yield efficient algorithms for finding VCG outcomes. For instance, consider a unit-demand auction for last-minute vacation packages in which some trusted third party (e.g., TripAdvisor) assigns a "quality" rating for each package and each bidder formulates a unit-demand bid for every package by simply declaring a linear function of the qualities of packages, i.e., determining the intercept and slope of this linear function. Within this context, we can formulate an auction as a complete weighted bipartite graph in the family that we consider in this paper. In some of the popular auction sites, e.g., eBay, bidding takes place in multiple rounds. eBay implements a variant of an English auction to sell a single item; the bids are sealed, but the second highest bid (plus one small bid increment), which is the amount that the winner pays, is displayed throughout the auction. We employ a similar approach by accepting the bids one-by-one and by maintaining an efficient representation of the tentative outcome for the enlarged set of bids. We show that we can process each bid in $\tilde{O}(\sqrt{n})$ time where n denotes the number of items in the auction. More precisely, we present a data structure that is initialized by the entire set of n items; the bids are introduced one-by-one in any order; the data structure maintains a compact representation of a VCG outcome (allocation and prices) for the bids introduced so far and for the entire set of items; it takes $O(\sqrt{n}\log^2 n)$ time to introduce a bid; it takes $O(n)$ time to print the outcome at any time.

Organization. In Sect. 2, we give the formal definition of the problem and introduce some useful definitions. In Sect. 3, we present an incremental framework for solving the problem. In Sect. 4, we present a basic algorithm within the framework of Sect. 3. Built on the concepts introduced in Sect. 4, we introduce a data structure and present our fast algorithm in Sect. 5. In Section 6, we extend the incremental framework to compute the VCG prices, and we present the algorithm within that framework.

2 Preliminaries

A *bid* is a triple $u = (slope, intercept, id)$ where $slope$ and $intercept$ are real numbers, and id is an integer. We use the notation $u.slope$ and $u.intercept$ to refer to the first and second components of a bid u, respectively. The bids are ordered lexicographically. An *item* is a pair $v = (quality, id)$ where $quality$ is a real number and id is an integer. We use the notation $v.quality$ to refer to the first component of an item v. The items are ordered lexicographically. For any bid u and any item v, we define $w(u, v)$ as $u.intercept + u.slope \cdot v.quality$.

For any set of bids U and any set of items V, we define the pair (U, V) as a *unit-demand auction with linear edge weights* (*UDALEW*). Such an auction represents a unit-demand auction instance where the set of bids is U, the set of items is V, and each bid u in U offers an amount $w(u, v)$ on each item v in V.

A UDALEW $A = (U, V)$ corresponds to a complete weighted bipartite graph G where left vertices are U, right vertices are V, and the weight of the edge between a left vertex u and a right vertex v is equal to $w(u, v)$. Hence, for a UDALEW, we use the standard graph theoretic terminology, alluding to the corresponding graph. The family of all such graphs G corresponds to the general graph family introduced in [2].

A *matching* of a UDALEW (U, V) is a set M of bid-item pairs where each bid (resp., item) in M belongs to U (resp., V) and no bid (resp., item) appears more than once in M. The *weight* of a matching M, denoted $w(M)$, is defined as the sum, over all bid-item pairs (u, v) in M, of $w(u, v)$.

In this paper, we solve the problem of finding a VCG outcome (allocation and prices) for a given UDALEW A; a VCG allocation is any MWM of A, and we characterize the VCG prices in Sect. 6.2. We reduce the problem of finding an MWM to the problem of finding a maximum weight maximum cardinality matching (MWMCM) as follows: we enlarge the given UDALEW instance $A = (U, V)$ by adding $|V|$ dummy bids to U, each with intercept zero and slope zero; we compute an MWMCM M of the resulting UDALEW A'; we remove from M all bid-item pairs involving dummy bids.

We conclude this section with some definitions that prove to be useful in the remainder of the paper. For any totally ordered set S — such as a set of bids, a set of items, or an ordered matching which we introduce below — we make the following definitions: any integer i is an *index* in S if $1 \leq i \leq |S|$; for any element e in S, we define the *index of* e *in* S, denoted $index(e, S)$, as the position of e in the ascending order of elements in S, where the index of the first (resp., last) element, also called the *leftmost* (resp., *rightmost*) element, is 1 (resp., $|S|$); $S[i]$ denotes the element with index i in S; for any two indices i and j in S such that $i \leq j$, $S[i : j]$ denotes the set $\{S[i], \ldots, S[j]\}$ of size $j - i + 1$; for any two integers i and j such that $i > j$, $S[i : j]$ denotes the empty set; for any integer i, $S[\,: i]$ (resp., $S[i :\,]$) denotes $S[1 : i]$ (resp., $S[i : |S|]$); a subset S' is a *contiguous* subset of S if $S' = S[i : j]$ for some $1 \leq i \leq j \leq |S|$.

For any matching M, we define $bids(M)$ (resp., $items(M)$) as the set of bids (resp., items) that participate in M. A matching M is *ordered* if M is equal to $\bigcup_{1 \leq i \leq |M|} \{(U[i], V[i])\}$ where U denotes $bids(M)$ and V denotes $items(M)$. The order of the pairs in an ordered matching is determined by the order of the bids (equivalently, items) of those pairs.

3 Incremental Framework

In this section, we present an incremental framework for the problem of finding an MWMCM of a given UDALEW $A = (U, V)$. As discussed below, it is a straightforward problem if $|U| \leq |V|$. Thus, the primary focus is on the case where $|U| > |V|$. We start with a useful definition and a simple lemma.

For any set of bids U and any set of items V such that $|U| = |V|$, we define $\mathit{matching}(U, V)$ as the ordered matching $\{(U[1], V[1]), \ldots, (U[|U|], V[|U|])\}$.

Lemma 1 below shows how to compute an MWMCM of a UDALEW where the number of bids is equal to the number of items. The proof follows from the rearrangement inequality [8, Section 10.2, Theorem 368].

Lemma 1. For any UDALEW $A = (U, V)$ such that $|U| = |V|$, $\mathit{matching}(U, V)$ is an MWMCM of A.

Corollary 1. For any UDALEW $A = (U, V)$ such that $|U| \geq |V|$, there exists an ordered MWMCM of A.

If $|U| < |V|$ in a given UDALEW (U, V), then it is straightforward to reduce the problem to the case where $|U| = |V|$. Let U' (resp., U'') denote the set of the bids in U having negative (resp., nonnegative) slopes. Then we find an MWMCM M' of the UDALEW $(U', V[\,:|U'|])$ and an MWMCM M'' of the UDALEW $(U'', V[|V| - |U''| + 1 :\,])$, and we combine M' and M'' to obtain an MWMCM of (U, V).

It remains to consider the problem of finding an MWMCM of a UDALEW (U, V) where $|U| > |V|$. The following is a useful lemma. The proof is straightforward by an augmenting path argument; see [2, Lemma 7] for the proof of a similar claim.

Lemma 2. Let $A = (U, V)$ be a UDALEW such that $|U| \geq |V|$. Let u be a bid that does not belong to U. Let M be an MWMCM of A and let U' denote $\mathit{bids}(M)$. Then, any MWMCM of the UDALEW $(U' + u, V)$ is an MWMCM of the UDALEW $(U + u, V)$.

Lemma 2 shows that the problem of finding an MWMCM of a UDALEW (U, V) where $|U| = |V| + k$ reduces to k instances of the problem of finding an MWMCM of a UDALEW where the number of bids exceeds the number of items by one. Below we establish an efficient incremental framework for solving the MWMCM problem based on this reduction.

For any ordered matching M and any bid u that does not belong to $\mathit{bids}(M)$, we define $\mathit{insert}(M, u)$ as the ordered MWMCM M' of the UDALEW $A = (\mathit{bids}(M) + u, \mathit{items}(M))$ such that the bid that is left unassigned by M', i.e., $(\mathit{bids}(M) + u) \setminus \mathit{bids}(M')$, is maximum, where the existence of M' is implied by Corollary 1.

We want to devise a data structure that maintains a dynamic ordered matching M. When the data structure is initialized, it is given an ordered matching M', and M is set to M'; we say that the data structure has initialization cost $T(n)$ if initialization takes at most $T(|M'|)$ steps. Subsequently, the following two operations are supported: the *bid insertion* operation takes as input a bid u not in $\mathit{bids}(M)$, and transforms the data structure so that M becomes $\mathit{insert}(M, u)$; the *dump* operation returns a list representation of M. We say that the data structure has bid insertion (resp., dump) cost $T(n)$ if bid insertion (resp., dump) takes at most $T(|M|)$ steps.

Lemma 3. Let $\mathcal{D}$ be an ordered matching data structure with initialization cost $f(n)$, bid insertion cost $g(n)$, and dump cost $h(n)$. Let A be a UDALEW (U, V) such that $|U| \geq |V|$. Then an MWMCM of A can be computed in $O(f(|V|) + (|U| - |V|) \cdot g(|V|) + h(|V|))$ time.

Proof. Let U' be a subset of U such that $|U'| = |V|$. Let $\langle u_1, \ldots, u_{|U|-|U'|} \rangle$ be a permutation of the bids in $U \setminus U'$. For any integer i such that $0 \leq i \leq |U| - |U'|$, let U_i denote $U' \cup \{u_1, \ldots, u_i\}$. Remark: $U_0 = U'$ and $U_{|U|-|U'|} = U$. We now show how to use $\mathcal{D}$ to find an ordered MWMCM of the UDALEW $A = (U_{|U|-|U'|}, V)$. We initialize $\mathcal{D}$ with $M_0 = \mathit{matching}(U_0, V)$, which by Lemma 1 is an ordered MWMCM of the UDALEW (U_0, V). Then we iteratively insert bids $u_1, \ldots, u_{|U|-|U'|}$. Let M_i denote the ordered matching associated with $\mathcal{D}$ after i iterations, $1 \leq i \leq |U| - |U'|$. By the definition of bid insertion, M_i is an ordered MWMCM of the UDALEW $(\mathit{bids}(M_{i-1}) + u_i, V)$, and thus, is an MWMCM of the UDALEW (U_i, V) by induction on i and Lemma 2. Thus, a dump on $\mathcal{D}$ after completing all iterations returns an ordered MWMCM of A. The whole process runs in the required time since we perform one initialization, $|U| - |U'|$ bid insertions, and one dump. $\square$

In Sect. 4, we give a simple linear-time bid insertion algorithm assuming an array representation of the ordered matching. Building on the concepts introduced in Sect. 4, Sect. 5 develops an ordered matching data structure with initialization cost $O(n \log^2 n)$, bid insertion cost $O(\sqrt{n} \log^2 n)$, and dump cost $O(n)$ (Theorem 2). The results of Sect. 5, together with Lemma 3, yield the $O(m\sqrt{n} \log^2 n)$ MWMCM time bound claimed in Sect. 1.

Looking from an auction perspective, as discussed in Sect. 2, our goal is to compute a VCG allocation and pricing given a UDALEW. In Sect. 6, we show how to extend the data structure of Sect. 5 to maintain the VCG prices as each bid is inserted. The asymptotic time complexity of the operations remain the same; the additional computation for maintaining the VCG prices takes $O(\sqrt{n})$ time at each bid insertion, where n denotes the size of the matching maintained by the data structure.

4 A Basic Bid Insertion Algorithm

In this section, we describe a linear-time implementation of $\mathit{insert}(M, u)$ given an array representation of the ordered matching. The algorithm described here is not only useful because it introduces the concepts that the fast algorithm we introduce in Sect. 5 is built on, but also the same approach is used in certain "block scan" computations of that fast algorithm. We first introduce two functions that, in a sense evident by their definitions, restrict $\mathit{insert}(M, u)$ into two halves, left and right, of M split by u.

For any ordered matching M and any bid u that does not belong to $\mathit{bids}(M)$, we define $\mathit{insert}_L(M, u)$ (resp., $\mathit{insert}_R(M, u)$) as the ordered MCM M' of the UDALEW $A = (\mathit{bids}(M) + u, \mathit{items}(M))$ of maximum weight subject to the condition that the bid that is left unassigned by M', i.e., $(\mathit{bids}(M) + u) \setminus \mathit{bids}(M')$, is less (resp., greater) than u, where the ties are broken by choosing the MCM that leaves the maximum such bid unassigned; if no such MCM exists, i.e., u is less (resp., greater) than every bid in $\mathit{bids}(M)$, then $\mathit{insert}_L(M, u)$ (resp., $\mathit{insert}_R(M, u)$) is defined as M.

The following lemma characterizes $\mathit{insert}(M, u)$ in terms of $\mathit{insert}_L(M, u)$ and $\mathit{insert}_R(M, u)$; the proof directly follows from the definitions of $\mathit{insert}(M, u)$, $\mathit{insert}_L(M, u)$, and $\mathit{insert}_R(M, u)$.

Lemma 4. Let M be a nonempty ordered matching and let u be a bid that does not belong to $\mathit{bids}(M)$. Let M_L denote $\mathit{insert}_L(M, u)$ and let M_R denote $\mathit{insert}_R(M, u)$. Let W denote the maximum of $w(M_L)$, $w(M)$, and $w(M_R)$. Then,

$$\mathit{insert}(M,u) = \begin{cases} M_R & \text{if } w(M_R) = W \\ M & \text{if } w(M) = W > w(M_R) \\ M_L & \text{otherwise.} \end{cases}$$

We now introduce some definitions that are used in Lemma 5 below to characterize $\mathit{insert}_L(M, u)$ and $\mathit{insert}_R(M, u)$.

For any ordered matching M and any two indices i and j in M, we define M_i^j as $\mathit{matching}(U - U[i], V - V[j])$, where U denotes $\mathit{bids}(M)$ and V denotes $\mathit{items}(M)$.

Let M be a nonempty ordered matching, let U denote $\mathit{bids}(M)$, and let V denote $\mathit{items}(M)$. Then we define $\Delta_L(M)$ as $w(M_1^{|M|}) - w(M)$, and we define $\Delta_R(M)$ as $w(M_{|M|}^1) - w(M)$. It is straightforward to see that $\Delta_L(M[i : j])$ and $\Delta_R(M[i : j])$ can be computed for any $1 \le i \le j \le |M|$ by the recurrences

$$\Delta_L(M[k-1 : j]) = \Delta_L(M[k : j]) + w(U[k], V[k-1]) - w(U[k-1], V[k-1]) \tag{L1}$$
$$\Delta_R(M[i : k+1]) = \Delta_R(M[i : k]) + w(U[k], V[k+1]) - w(U[k+1], V[k+1]) \tag{R1}$$

with base cases $\Delta_L(M[j]) = -w(U[j], V[j])$ and $\Delta_R(M[i]) = -w(U[i], V[i])$.

Let M be a nonempty ordered matching. Letting W denote $\max_{1 \le i \le |M|} w(M_i^{|M|})$, we define $\Delta_L^*(M)$ as $W - w(M)$, and we define $\mathit{loser}_L(M)$ as $\max\left\{i \mid w(M_i^{|M|}) = W\right\}$. Symmetrically, letting W' denote $\max_{1 \le i \le |M|} w(M_i^1)$, we define $\Delta_R^*(M)$ as $W' - w(M)$, and we define $\mathit{loser}_R(M)$ as $\max\{i \mid w(M_i^1) = W'\}$. By Lemma 1 and by the definitions of $\Delta_L(M)$ and $\Delta_R(M)$, it is straightforward to see that $(\Delta_L^*(M), \mathit{loser}_L(M)) = \max_{1 \le i \le |M|}(\Delta_L(M[i : \,]), i)$ and $(\Delta_R^*(M), \mathit{loser}_R(M)) = \max_{1 \le i \le |M|}(\Delta_R(M[\, : i]), i)$ (the pairs compare lexicographically). Hence, $\Delta_L^*(M[i : j])$, $\mathit{loser}_L(M[i : j])$, $\Delta_R^*(M[i : j])$, and $\mathit{loser}_R(M[i : j])$ can be computed for any $1 \le i \le j \le |M|$ by the recurrences

$$(\Delta_L^*(M[k-1 : j]), \mathit{loser}_L(M[k-1 : j])) = \max\{(\Delta_L^*(M[k : j]), \mathit{loser}_L(M[k : j]) + 1), (\Delta_L(M[k-1 : j]), 1)\} \tag{L2}$$

$$(\Delta_R^*(M[i : k+1]), \mathit{loser}_R(M[i : k+1])) = \max\{(\Delta_R^*(M[i : k]), \mathit{loser}_R(M[i : k])), (\Delta_R(M[i : k+1]), k+2-i)\} \tag{R2}$$

with base cases $\Delta_L^*(M[j]) = -w(U[j], V[j])$, $\Delta_R^*(M[i]) = -w(U[i], V[i])$, and $\mathit{loser}_L(M[j]) = \mathit{loser}_R(M[i]) = 1$.

Lemma 5. Let M be a nonempty ordered matching, let U denote $\mathit{bids}(M)$, let V denote $\mathit{items}(M)$, let u be a bid that does not belong to U, let k denote $\mathit{index}(u, U + u)$, let M_L denote $\mathit{insert}_L(M, u)$, and let M_R denote $\mathit{insert}_R(M, u)$. If $k > 1$, then M_L is equal to $M_i^{k-1} + (u, V[k-1])$ and $w(M_L) = w(M) + \Delta_L^*(M[\, : k-1]) + w(u, V[k-1])$ where i denotes $\mathit{loser}_L(M[\, : k-1])$; otherwise, $M_L = M$. If $k \le |M|$, then M_R is equal to $M_j^k + (u, V[k])$ and $w(M_R) = w(M) + \Delta_R^*(M[k : \,]) + w(u, V[k])$ where j denotes $\mathit{loser}_R(M[k : \,]) + k - 1$; otherwise, $M_R = M$.

Algorithm 1 A linear-time implementation of bid insertion. The difference of the weight of an MWMCM of the UDALEW $A = (\mathit{bids}(M) + u, \mathit{items}(M))$ and that of M is equal to δ, and the maximum bid in $\mathit{bids}(M) + u$ that is unmatched in some MWMCM of A is u^*.

```
Input: $M$ is an ordered matching and $u$ is a bid that does not belong to $bids(M)$.
Output: $insert(M, u)$.
 1: Let $U$ denote $bids(M)$ and let $V$ denote $items(M)$
 2: $C \leftarrow \{(0, u)\}$
 3: $k \leftarrow index(u, U + u)$
 4: if $k > 1$ then
 5:     for $i = k - 1$ down to $1$ do
 6:         Compute $\Delta_L(M[i : k-1])$ via (L1)
 7:         Compute $\Delta^*_L(M[i : k-1])$ and $loser_L(M[i : k-1])$ via (L2)
 8:     end for
 9:     $C \leftarrow C + (w(u, V[k-1]) + \Delta^*_L(M[\,: k-1]), U[i])$ where $i = loser_L(M[\,: k-1])$
10: end if
11: if $k \leq |M|$ then
12:     for $i = k$ to $|M|$ do
13:         Compute $\Delta_R(M[k : i])$ via (R1)
14:         Compute $\Delta^*_R(M[k : i])$ and $loser_R(M[k : i])$ via (R2)
15:     end for
16:     $C \leftarrow C + (w(u, V[k]) + \Delta^*_R(M[k :\,]), U[j])$ where $j = loser_R(M[k :\,]) + k - 1$
17: end if
18: $(\delta, u^*) \leftarrow$ the lexicographically maximum pair in $C$
19: return $matching(U + u - u^*, V)$
```

Proof. We address the claim regarding M_L; the claim regarding M_R is symmetric. There is nothing to prove if $k = 1$, so assume that $k > 1$. Since both M and M_L are ordered, and since each bid in M that is greater than u is in M_L, it is easy to see that $M[k :\,] = M_L[k :\,]$, and thus $w(M_L) - w(M) = w(M_L[\,: k-1]) - w(M[\,: k-1])$. Then, since M_L is ordered, u is matched to $V[k-1]$ in M_L, and thus M_L is equal to $M_i^{k-1} + (u, V[k-1])$ for some $i < k$. The observations in the preceding two paragraphs and the definitions of M_L, Δ^*_L, and loser_L imply that $w(M_L) - w(M) = w(u, V[k-1]) + \Delta^*_L(M[\,: k-1])$ and the index i in the preceding sentence is equal to $\mathit{loser}_L(M[\,: k-1])$. □

Lemmas 4 and 5, together with (L1), (R1), (L2), and (R2), directly suggest a linear-time computation of $\mathit{insert}(M, u)$, as shown in Algorithm 1. If $\mathit{insert}_L(M, u)$ (resp., $\mathit{insert}_R(M, u)$) is not equal to M, then the algorithm computes the difference $w(\mathit{insert}_L(M, u)) - w(M)$ (resp., $w(\mathit{insert}_R(M, u)) - w(M)$) and adds a pair at line 9 (resp., line 16) to a set C where the first component is this difference, and the second component is the bid in $\mathit{bids}(M) + u$ that is left unassigned by $\mathit{insert}_L(M, u)$ (resp., $\mathit{insert}_R(M, u)$). Then by Lemma 4, the algorithm correctly returns $\mathit{insert}(M, u)$ by choosing the maximum pair of C at line 18.

5 A Superblock-Based Bid Insertion Algorithm

In this section, we describe an ordered matching data structure based on the concept of a "superblock", and we show how to use this data structure to obtain a significantly faster bid insertion algorithm than that presented in Sect. 4. Before beginning our formal presentation in Sect. 5.1, we provide a high-level overview of the main ideas. A reader interested in only the formal presentation may proceed to Sect. 5.1 without loss of continuity.

Recall that an ordered matching data structure maintains a dynamic ordered matching M. Let n denote $|M|$. We maintain a partition of the bids of M into contiguous "groups" of size $\Theta(\ell)$, where ℓ is a parameter to be optimized later. The time complexity of Alg. 1 is linear because the **for** loops starting at lines 5 and 12 process bid-item pairs in M sequentially. Our rough plan is to accelerate the computations associated with this pair of loops by proceeding group-by-group. We can process a group in constant time if we are given six "auxiliary values" that depend on the "submatching" M' of M associated with the bids in the group, namely: $\Delta_L(M')$, $\Delta_R(M')$, $\Delta^*_L(M')$, $\Delta^*_R(M')$, $loser_L(M')$, and $loser_R(M')$. The auxiliary values associated with a group can be computed in $\Theta(\ell)$ time. A natural approach is to precompute these auxiliary values when a group is created or modified, or when the set of matched items associated with the group is modified. Unfortunately, a single bid insertion can cause each bid in a contiguous interval of $\Theta(n)$ bids to have a new matched item. For example, if a bid insertion introduces a "low" bid u and deletes a "high" bid u', then each bid between u and u' gets a new matched item one position to the right of its old matched item. Since a constant fraction of the groups might need to have their auxiliary values recomputed as a result of a bid insertion, the overall time complexity remains linear.

The preceding discussion suggests that it might be useful to have an efficient way to obtain the new auxiliary values of a group of bids when the corresponding interval of matched items is shifted left or right by one position. To this end, we enhance the precomputation associated with a group of bids as follows: Instead of precomputing only the auxiliary values corresponding to the group's current matched interval of items, we precompute the auxiliary values associated with shifts of $0, \pm 1, \pm 2, \ldots, \pm\Theta(\ell)$ positions around the current matched interval. That way, unless a group of bids is modified (e.g., due to a bid being deleted or inserted) we do not need to redo the precomputation with the group until it has been shifted $\Omega(\ell)$ times. Since the enhanced precomputation computes $\Theta(\ell)$ sets of auxiliary values instead of one set, a naive implementation of the enhanced precomputation has $\Theta(\ell^2)$ time complexity, leading once again to linear worst-case time complexity for bid insertion. We obtain a faster bid insertion algorithm by showing how to perform the enhanced precomputation in $O(\ell \log^2 \ell)$ time.

Our $O(\ell \log^2 \ell)$-time algorithm for performing the enhanced precomputation forms the core of our fast bid insertion algorithm. Here we briefly mention the main techniques used to perform the enhanced precomputation efficiently; the reader is referred to Sect. 5.3.1 for further details. A divide-and-conquer approach is used to compute the auxiliary values associated with the functions $loser_L$ and $loser_R$ in $O(\ell \log \ell)$ time; the correctness of this approach is based on a monotonicity result (see Lemmas 8 and 9). A convolution-based approach is used to compute the auxiliary values based on Δ_L and Δ_R in $O(\ell \log \ell)$ time (see Lemma 7). The auxiliary values based on $loser_L$ (resp., $loser_R$) are used within a divide-and-conquer framework to compute the auxiliary values based on Δ^*_L (resp., Δ^*_R); in the associated recurrence, the overhead term is dominated by the cost of evaluating the same kind of convolution as in the computation of the auxiliary values based on Δ_L and Δ_R. As a result, the overall time complexity for computing the auxiliary values based on Δ_L^* and Δ_R^* is $O(\ell \log^2 \ell)$.

Section 5.1 introduces the concept of a "block", which is used to represent a group of bids together with a contiguous interval of items that includes all of the items matched to the group. Section 5.3.1 presents a block data structure. When a block data structure is "initialized" with a group of bids and an interval of items, the enhanced precomputation discussed in the preceding paragraph is performed, and the associated auxiliary values are stored in tables. A handful of "fields" associated with the block are also initialized; these fields store basic information such as the number of bids or items in the block. After initialization, the block data structure is read-only: Whenever a block needs to be altered (e.g., because a bid needs to be inserted/deleted, because the block needs to be merged with an adjacent block), we destroy the block and create a new one. The operations supported by a block may be partitioned into three categories: "queries", "lookups" and "scans". Each query runs in constant time and returns the value of a specific field. Each lookup runs in constant time and uses a table lookup to retrieve one of the precomputed auxiliary values. Each of the two linear-time scan operations (one leftgoing, one rightgoing) performs a naive emulation of one of the **for** loops of Alg. 1; in the context of a given bid insertion, such operations are only invoked on the block containing the insertion position of the new bid.

Section 5.1 defines the concept of a superblock, which is used to represent an ordered matching as a sequence of blocks. A superblock-based ordered matching data structure is introduced in Sect. 5.3.2, where each of the constituent blocks is represented using the block data structure alluded to in the preceding paragraph. In Sect. 5.3, we simplify the presentation by setting the parameter ℓ to $\Theta(\sqrt{n})$. For this choice of ℓ, we show that bid insertion can be performed using $O(1)$ block initializations, $O(\sqrt{n})$ block queries, $O(\sqrt{n})$ block lookups, at most two block scans, and $O(\sqrt{n})$ additional overhead, resulting in an overall time complexity of $O(\sqrt{n} \log^2 n)$. In terms of the parameters ℓ and n, the approach of Sect. 5.3 can be generalized to perform bid insertion using $O(\lceil n/\ell^2 \rceil)$ block initializations, $O(n/\ell)$ block queries, $O(n/\ell)$ block lookups, at most two block scans, and $O(n/\ell)$ additional overhead; it is easy to verify that setting ℓ to $\Theta(\sqrt{n})$ minimizes the overall time complexity.

5.1 Blocks and Superblocks

We define a *block* B as a UDALEW (U, V) where $|U| \leq |V|$. For any block $B = (U, V)$, we define $\mathit{shifts}(B)$ as $|V| - |U| + 1$. For any block $B = (U, V)$ and any integer t such that $1 \leq t \leq \mathit{shifts}(B)$, we define $\mathit{matching}(B, t)$ as $\mathit{matching}(U, V[t : t + |U| - 1])$.

Let M be a nonempty ordered matching, let U denote $\mathit{bids}(M)$, and let V denote $\mathit{items}(M)$. Let m be a positive integer, and let $\langle a_0, \ldots, a_m \rangle$, $\langle b_1, \ldots, b_m \rangle$, and $\langle c_1, \ldots, c_m \rangle$ be sequences of integers such that $a_0 = 0$, $a_m = |U|$, and $1 \leq b_i \leq a_{i-1} + 1 \leq a_i \leq c_i \leq |U|$ for $1 \leq i \leq m$. Let B_i denote the block $(U[a_{i-1} + 1 : a_i], V[b_i : c_i])$ for $1 \leq i \leq m$. Then the list of blocks $S = \langle B_1, \ldots, B_m \rangle$ is a *superblock*, and we make the following additional definitions: $\mathit{matching}(S)$ denotes M; $\mathit{size}(S)$ denotes $|M|$; $\mathit{bids}(S)$ denotes U; $\mathit{items}(S)$ denotes V; $\mathit{shift}(S, i)$ and $\mathit{shift}(S, B_i)$ both denote $b_i - a_{i-1}$ for $1 \leq i \leq m$; $\mathit{sum}(S, i)$ denotes a_i for $0 \leq i \leq m$; the leftmost block B_1 and the rightmost block B_m are the *boundary blocks*, the remaining blocks $B_2, \ldots, B_{m-1}$ are the *interior blocks*. Remark: For any superblock S, $\mathit{matching}(S) = \bigcup_{1 \leq i \leq |S|} \mathit{matching}(S[i], \mathit{shift}(S, i))$.

5.2 Algorithm 2

We obtain a significantly faster bid insertion algorithm than Alg. 1 by accelerating the computations associated with the **for** loops starting at lines 5 and 12. Recall that the first loop computes $\Delta^*_L(M[\,:\,k-1])$ and $\mathit{loser}_L(M[\,:\,k-1])$, and the second one computes $\Delta^*_R(M[k\,:\,])$ and $\mathit{loser}_R(M[k\,:\,])$. These two loops process a trivial representation of M pair-by-pair using the recurrences (L1), (R1), (L2), and (R2). We start by generalizing these recurrences; these generalizations allow us to compute the aforementioned values more efficiently by looping over a superblock-based representation of the matching block-by-block, instead of pair-by-pair.

Let M denote $\mathit{matching}(U,V)$, and let i, j, and k be three indices in M such that $i \le j < k$. Then the following equation generalizes (L1), and it is straightforward to prove by repeated application of (L1).

$$\Delta_L(M[i:k]) = \Delta_L(M[j+1:k]) + w(U[j+1],V[j]) + \Delta_L(M[i:j]). \tag{L1$'$}$$

We also give a generalization of (L2), where the proof follows from the definitions of Δ^*_L and loser_L.

$$(\Delta^*_L(M[i:k]), \mathit{loser}_L(M[i:k])) = \max\left\{\begin{matrix}(\Delta^*_L(M[j+1:k]), \mathit{loser}_L(M[j+1:k]) + j + 1 - i), \\ (\Delta^*_L(M[i:j]) + w(U[j+1],V[j]) + \Delta_L(M[j+1:k]), \mathit{loser}_L(M[i:j]))\end{matrix}\right\} \tag{L2$'$}$$

Let M denote $\mathit{matching}(U,V)$, and let i, j, and k be three indices in M such that $i < j \le k$. Symmetrically, the following equations generalize (R1) and (R2).

$$\Delta_R(M[i:k]) = \Delta_R(M[i:j-1]) + w(U[j-1],V[j]) + \Delta_R(M[j:k]) \tag{R1$'$}$$

$$(\Delta^*_R(M[i:k]), \mathit{loser}_R(M[i:k])) = \max\left\{\begin{matrix}(\Delta^*_R(M[i:j-1]), \mathit{loser}_R(M[i:j-1])), \\ (\Delta^*_R(M[j:k]) + w(U[j-1],V[j]) + \Delta_R(M[i:j-1]), \mathit{loser}_R(M[j:k]) + j - i)\end{matrix}\right\} \tag{R2$'$}$$

We use (L1$'$) and (L2$'$) (resp., (R1$'$) and (R2$'$)) within a loop that iterates over a superblock-based representation of the matching block-by-block. In each iteration of the loop, we are able to evaluate the right-hand side of (L1$'$) and (L2$'$) (resp., (R1$'$) and (R2$'$)) in constant time because the terms involving $M[j+1:k]$ (resp., $M[i:j-1]$) are carried over from the previous iteration, and the terms involving $M[i:j]$ (resp., $M[j:k]$) are already stored in precomputed tables associated with the blocks of the superblock.

The high-level algorithm is given in Alg. 2. The input is a superblock S that represents an ordered matching, denoted M (i.e., $\mathit{matching}(S) = M$), and a bid u that does not belong to $\mathit{bids}(S)$. The output is a superblock representing $\mathit{insert}(M,u)$. The unique bid u^* that is unmatched in $\mathit{insert}(M,u)$ is identified using the block-based framework alluded to above. After identifying u^*, if $u^* \neq u$, the algorithm invokes a subroutine $\textsc{Swap}(S,u^*,u)$ which, given a superblock S, a bid u^* that belongs to $\mathit{bids}(S)$, and a bid u that does not belong to $\mathit{bids}(S)$, returns a superblock that represents $\mathit{matching}(\mathit{bids}(S)+u-u^*, \mathit{items}(S))$. We present our implementation of Swap and analyze its time complexity in Sections 5.3.3 and 5.3.4. The correctness of Alg. 2 is established in Lemma 6, where it is shown that Alg. 2 emulates the behavior of Alg. 1.

Algorithm 2 A high-level bid insertion algorithm using the superblock-based representation of an ordered matching.

```
Input: $S$ is a superblock and $u$ is a bid that does not belong to $bids(S)$.
Output: A superblock $S'$ such that $matching(S') = insert(matching(S), u)$.
 1: Let $M$ denote $matching(S)$, let $U$ denote $bids(S)$, and let $V$ denote $items(S)$
 2: Let $S[i]$ be $(U_i, V_i)$ for $1 \leq i \leq |S|$
 3: $\sigma(i) \leftarrow sum(S, i)$ for $0 \leq i \leq |S|$
 4: $C \leftarrow \{(0, u)\}$
 5: $\ell \leftarrow |\{(U', V') \mid (U', V') \in S$ and $U'[1] < u\}|$
 6: $k \leftarrow$ if $\ell < 1$ then $1$ else $index(u, U_\ell + u) + 1 + \sigma(\ell - 1)$
 7: if $k > 1$ then
 8:     for $i = k - 1$ down to $\sigma(\ell - 1) + 1$ do
 9:         Compute $\Delta_L(M[i : k - 1])$ via (L1)
10:         Compute $\Delta^*_L(M[i : k - 1])$ and $loser_L(M[i : k - 1])$ via (L2)
11:     end for
12:     for $i = \ell - 1$ down to $1$ do
13:         Compute $\Delta_L(M[\sigma(i - 1) + 1 : k - 1])$ via (L1$'$)
14:         Compute $\Delta^*_L(M[\sigma(i - 1) + 1 : k - 1])$ and $loser_L(M[\sigma(i - 1) + 1 : k - 1])$ via (L2$'$)
15:     end for
16:     $C \leftarrow C + (w(u, V[k - 1]) + \Delta^*_L(M[\,: k - 1]), U[i])$ where $i = loser_L(M[\,: k - 1])$
17: end if
18: if $k \leq |M|$ then
19:     for $i = k$ to $\sigma(\ell)$ do
20:         Compute $\Delta_R(M[k : i])$ via (R1)
21:         Compute $\Delta^*_R(M[k : i])$ and $loser_R(M[k : i])$ via (R2)
22:     end for
23:     for $i = \ell + 1$ to $|S|$ do
24:         Compute $\Delta_R(M[k : \sigma(i)])$ via (R1$'$)
25:         Compute $\Delta^*_R(M[k : \sigma(i)])$ and $loser_R(M[k : \sigma(i)])$ via (R2$'$)
26:     end for
27:     $C \leftarrow C + (w(u, V[k]) + \Delta^*_R(M[k :\,]), U[j])$ where $j = loser_R(M[k :\,]) + k - 1$
28: end if
29: $(\delta, u^*) \leftarrow$ the lexicographically maximum pair in $C$
30: return if $u^* \neq u$ then SWAP$(S, u^*, u)$ else $S$
```

Lemma 6. Algorithm 2 is correct.

Proof. Assume that, given a superblock S, a bid u^* that belongs to $bids(S)$, and a bid u that does not belong to $bids(S)$, SWAP(S, u^*, u) correctly returns a superblock that represents $matching(bids(S) + u - u^*, items(S))$. Let M denote $matching(S)$, let U denote $bids(S)$, and let V denote $items(S)$, as in the algorithm. First, the algorithm performs a scan over the blocks to compute an integer ℓ at line 5 so that each bid in each block $S[i]$ for $1 \leq i < \ell$ (resp., $\ell < i \leq |S|$) is less (resp., greater) than the new bid u. Then it is easy to see that the integer k computed at line 6 is equal to $index(u, U + u)$, as in Alg. 1. It remains to show that $\Delta^*_L(M[\,: k - 1])$, $loser_L(M[\,: k - 1])$,

$\Delta^*_R(M[k:\,])$, and $loser_R(M[k:\,])$ are computed correctly so that the set C is populated with the same pairs as in Alg. 1, and thus Lemma 4 implies that the bid u^* in $bids(M)+u$ that is left out by $insert(M,u)$ is correctly identified by choosing the maximum pair of C at line 29, as in Alg. 1, and that the superblock returned at line 30 represents $insert(M,u)$.

If $k > 1$ (resp., $k \le |M|$), the algorithm proceeds to emulate the **for** loop of Alg. 1 that starts at line 5 (resp., line 12) to compute $\Delta^*_L(M[\,:\,k-1])$ and $loser_L(M[\,:\,k-1])$ (resp., $\Delta^*_R(M[k\,:\,])$ and $loser_R(M[k\,:\,])$). We first discuss the emulation of the loop of Alg. 1 that starts at line 5; this emulation is performed by two **for** loops in Alg. 2. The first loop in Alg. 2, which starts at line 8, is identical to the loop of Alg. 1, except that it stops when the boundary of the submatching represented by block $S[\ell]$ is reached. Thus, by repeated application of (L1) and (L2), upon completion of this first loop, we have computed $\Delta_L(M')$, $\Delta^*_L(M')$, and $loser_L(M')$ where M' denotes $M[\sigma(\ell-1)+1 : k-1]$. Then the second **for** loop, which starts at line 12, resumes where the first one left off; however, it utilizes the superblock-based representation of the ordered matching to loop block-by-block. During the iterations of the second loop, for $i = \ell - 1$ down to 1, (L1$'$) and (L2$'$) are invoked by setting i, j, and k in these equations to $\sigma(i-1)+1$, $\sigma(i)$, and $k-1$, respectively; thus the submatching $M[i\,:\,j]$ in these equations corresponds to the submatching that the block $S[i]$ represents in S, i.e., $matching(S[i], shift(S,i))$. During such an iteration i, for $i = \ell - 1$ down to 1, the terms involving $M[i\,:\,j]$ in (L1$'$) and (L2$'$), i.e., the terms that are equal to $\Delta_L(matching(S[i], shift(S,i)))$, $\Delta^*_L(matching(S[i], shift(S,i)))$, and $loser_L(matching(S[i], shift(S,i)))$, are fetched from the precomputed tables associated with the block $S[i]$. Note that all the terms in these equations involving $M[j+1 : k]$ are carried over from the previous iteration, except for the first iteration, where they are already computed by the first **for** loop. Thus, upon completion of these two loops, we have computed $\Delta^*_L(M[\,:\,k-1])$ and $loser_L(M[\,:k-1])$.

The emulation of the second loop of Alg. 1 (starting at line 12) that computes $\Delta^*_R(M[k:\,])$ and $loser_R(M[k\,:\,])$ can be argued symmetrically, where (R1$'$) and (R2$'$) are invoked in the **for** loop at line 23 by setting i, j, and k in these equations to k, $\sigma(i-1)+1$, and $\sigma(i)$, respectively. Then the submatching $M[j\,:\,k]$ in these equations corresponds to the submatching that the block $S[i]$ represents in S, i.e., $matching(S[i], shift(S,i))$. □

5.3 Fast Implementation of Algorithm 2

In this section, we complete the discussion of our fast bid insertion algorithm by describing two data structures, giving the implementation details, and analyzing the running time. First, in Sect. 5.3.1, we present a block data structure that precomputes the auxiliary tables mentioned in Sect. 5.2 in quasilinear time, thus allowing lines 13, 14, 24, and 25 of Alg. 2 to be performed in constant time. Then, in Sect. 5.3.2, we introduce a superblock-based ordered matching data structure that stores the blocks using the block data structure, where the sizes of the blocks are optimized to balance the cost of SWAP with that of the remaining operations in Alg. 2. Finally, in Sections 5.3.3 and 5.3.4, we present our efficient implementation of SWAP, which constructs only a constant number of blocks, and analyze its time complexity.

5.3.1 Block Data Structure

Let S be a superblock on which a bid insertion is performed, let B be a block in S, and let M_t denote $\mathit{matching}(B,t)$ for $1 \leq t \leq \mathit{shifts}(B)$. The algorithm may query $\Delta_L(M_t)$, $\Delta_R(M_t)$, $\Delta^*_L(M_t)$, $\Delta^*_R(M_t)$, $\mathit{loser}_L(M_t)$, and $\mathit{loser}_R(M_t)$ for $t = \mathit{shift}(S,B)$. If B is part of the superblocks for a series of bid insertions, then these queries may be performed for various t values. For a fast implementation of Alg. 2, instead of individually computing these quantities at query time, we efficiently precompute them during the construction of the block and store them in the following six lists. We define $\Delta_L(B)$ as the list of size $\mathit{shifts}(B)$ such that $\Delta_L(B)[t]$ is equal to $\Delta_L(M_t)$ for $1 \leq t \leq \mathit{shifts}(B)$. We define the lists $\Delta_R(B)$, $\Delta^*_L(B)$, $\Delta^*_R(B)$, $\mathit{loser}_L(B)$, and $\mathit{loser}_R(B)$ similarly. The representation of a block $B = (U,V)$ simply maintains each of the following explicitly as an array: U, V, $\Delta_L(B)$, $\Delta_R(B)$, $\Delta^*_L(B)$, $\Delta^*_R(B)$, $\mathit{loser}_L(B)$, and $\mathit{loser}_R(B)$. In what follows, we refer to that representation as the *block data structure for* B. The block data structure is an integral part of the superblock-based ordered matching data structure which we introduce in the following section.

The main technical contribution of this paper is that we can compute the aforementioned lists efficiently as stated in the following theorem.

Theorem 1. The block data structure can be constructed in $O(|V|\,(\log \mathit{shifts}(B) + \log^2 |U|))$ time for any block $B = (U,V)$.

The proof of Theorem 1 follows directly from Lemmas 7, 10, and 11 below.

Lemma 7. Let B be a block (U,V). Then $\Delta_L(B)$ and $\Delta_R(B)$ can be computed in $O(|V|\log|U|)$ time.

Proof. We address the computation of $\Delta_L(B)$, the computation of $\Delta_R(B)$ is symmetric. Let β denote $\Delta_L(B)$. We define the following two real-valued functions on the set of integers. Let $x(n)$ be $V[n+1].\mathit{quality} - V[n+2].\mathit{quality}$, if $0 \leq n < |V|-1$; 0, otherwise. Let $h(n)$ be $U[|U|-n].\mathit{slope}$, if $0 \leq n < |U|-1$; 0, otherwise. Let $y(n)$ denote the discrete convolution $(x * h)(n) = \sum_m h(m) \cdot x(n-m)$. Then, for $1 \leq t \leq \mathit{shifts}(B)$,

$$\begin{aligned}
\beta[t] &= \sum_{1<i\leq|U|} w(U[i], V[i+t-2]) - \sum_{1\leq i\leq|U|} w(U[i], V[i+t-1]) \\
&= -w(U[1], V[t]) + \sum_{1<i\leq|U|} U[i].\mathit{slope} \cdot (V[i+t-2].\mathit{quality} - V[i+t-1].\mathit{quality}) \\
&= -w(U[1], V[t]) + \sum_{1<i\leq|U|} h(|U|-i) \cdot x(i+t-3) \\
&= -w(U[1], V[t]) + \sum_{0\leq m<|U|-1} h(m) \cdot x(|U|-m+t-3) \\
&= -w(U[1], V[t]) + y(t+|U|-3)
\end{aligned}$$

where the convolution $y(n) = (x * h)(n)$ can be computed in $O(|V|\log|U|)$ time by computing $\Theta(|U|)$-size segments of $y(n)$ using fast circular convolution, and concatenating the segments together [14]. □

The next two lemmas establish a monotonicity result that is used to prove Lemma 10.

Lemma 8. Let B be a block (U, V) and let α denote $\mathit{loser}_L(B)$. Then for any integer t such that $1 \le t < \mathit{shifts}(B)$, $\alpha[t] \ge \alpha[t+1]$.

Proof. For the sake of contradiction, suppose $\alpha[t] < \alpha[t+1]$ for some t such that $1 \le t < \mathit{shifts}(B)$. Let M denote $\mathit{matching}(B, t)$ and let M' denote $\mathit{matching}(B, t+1)$. Let i denote $\alpha[t] = \mathit{loser}_L(M)$ and let i' denote $\alpha[t+1] = \mathit{loser}_L(M')$. Since $i = \mathit{loser}_L(M) < i'$, we have $\Delta_L(M[i:\,]) > \Delta_L(M[i':\,])$, which, together with (L1′), implies $\Delta_L(M[i : i']) > \Delta_L(M[i'])$, and hence

$$\sum_{i<\ell\le i'} w(U[\ell], V[\ell+t-2]) - \sum_{i\le\ell<i'} w(U[\ell], V[\ell+t-1]) > 0. \tag{1}$$

Since $i' = \mathit{loser}_L(M')$, we have $\Delta_L(M'[i:\,]) \le \Delta_L(M'[i':\,])$, which, together with (L1′), implies $\Delta_L(M'[i : i']) \le \Delta_L(M'[i'])$, and hence

$$\sum_{i<\ell\le i'} w(U[\ell], V[\ell+t-1]) - \sum_{i\le\ell<i'} w(U[\ell], V[\ell+t]) \le 0. \tag{2}$$

Subtracting (1) from (2), we get

$$\begin{aligned}
0 &> \sum_{i<\ell\le i'} [w(U[\ell], V[\ell+t-1]) - w(U[\ell], V[\ell+t-2])] \\
&\quad - \sum_{i\le\ell<i'} [w(U[\ell], V[\ell+t]) - w(U[\ell], V[\ell+t-1])] \\
&= \sum_{i\le\ell<i'} U[\ell+1].\mathit{slope} \cdot (V[\ell+t].\mathit{quality} - V[\ell+t-1].\mathit{quality}) \\
&\quad - \sum_{i\le\ell<i'} U[\ell].\mathit{slope} \cdot (V[\ell+t].\mathit{quality} - V[\ell+t-1].\mathit{quality}) \\
&= \sum_{i\le\ell<i'} (U[\ell+1].\mathit{slope} - U[\ell].\mathit{slope})(V[\ell+t].\mathit{quality} - V[\ell+t-1].\mathit{quality}),
\end{aligned}$$

which contradicts the way that the bids in U and the items in V are ordered. □

Lemma 9. Let B be a block (U, V) and let α denote $\mathit{loser}_R(B)$. Then for any integer t such that $1 \le t < \mathit{shifts}(B)$, $\alpha[t] \le \alpha[t+1]$.

Proof. Symmetric to the proof of Lemma 8. □

We now introduce two definitions that return "subblocks" of a block and that are useful in the proofs of Lemmas 10 and 11 which give divide-and-conquer algorithms.

For any block $B = (U, V)$ and for any two indices i and i' such that $1 \le i \le i' \le |U|$, we define $\mathit{subBids}(B, i, i')$ as the block B' such that $\mathit{shifts}(B') = \mathit{shifts}(B)$ and $\mathit{matching}(B', t) = M_t[i : i']$ for $1 \le t \le \mathit{shifts}(B)$, where M_t denotes $\mathit{matching}(B, t)$; it is straightforward to see that $\mathit{subBids}(B, i, i') = (U[i : i'], V[i : |V| - |U| + i'])$.

For any block $B = (U, V)$ and for any two integers t and t' such that $1 \le t \le t' \le \mathit{shifts}(B)$, we define $\mathit{subShifts}(B, t, t')$ as the block B' such that $\mathit{shifts}(B') = t' - t + 1$ and $\mathit{matching}(B', t'') = \mathit{matching}(B, t'' + t - 1)$ for $1 \le t'' \le \mathit{shifts}(B')$; it is straightforward to see that $\mathit{subShifts}(B, t, t') = (U, V[t : |V| - \mathit{shifts}(B) + t'])$.

Lemma 10. Let B be a block (U, V). Then $loser_L(B)$ and $loser_R(B)$ can be computed in $O(|V| \log shifts(B))$ time.

Proof. We address the computation of $loser_L(B)$, which relies on Lemma 8. The computation of $loser_R(B)$ is symmetric, and relies on Lemma 9. We begin by stating a useful claim.

Let M be an ordered matching and let j be an index in M. Then we claim that, $loser_L(M) \leq j$ implies $loser_L(M) = loser_L(M[\,:\,j])$; similarly, $loser_L(M) \geq j$ implies $loser_L(M) = loser_L(M[j\,:\,]) + j - 1$. The proof of the claim is immediate from (L2′).

Let α denote $loser_L(B)$. We give a divide-and-conquer algorithm that computes α. If $|U| = 1$, then $\alpha[t] = 1$ for any item t and we are done; otherwise, we proceed as follows. Let t^* denote $\lceil shifts(B)/2 \rceil$ and let m denote $\alpha[t^*]$. We first compute m in $O(|U|)$ time using (L1) and (L2). Let B_1 denote the block $subBids(B, 1, m)$ and let B_2 denote the block $subBids(B, m, |U|)$. Let α_1 denote $loser_L(B_1)$ and let α_2 denote $loser_L(B_2)$. Then, by Lemma 8 and by the claim of the preceding paragraph,

$$\alpha[t] = \begin{cases} \alpha_1[t] & \text{if } t^* < t \leq shifts(B) \\ m & \text{if } t = t^* \\ \alpha_2[t] + m - 1 & \text{if } 1 \leq t < t^*. \end{cases}$$

Thus, it remains to compute $\alpha_1[t^* + 1 : shifts(B)]$ and $\alpha_2[1 : t^* - 1]$. Note that $\alpha_1[t^* + 1 : shifts(B)]$ is equal to $loser_L(B_1')$ for the block $B_1' = subShifts(B_1, t^* + 1, shifts(B))$, so we compute it recursively. Similarly, $\alpha_2[1 : t^* - 1]$ is equal to $loser_L(B_2')$ for the block $B_2' = subShifts(B_2, 1, t^* - 1)$, so we compute it recursively.

The overall running time satisfies the recurrence

$$T(n, s) \leq T(m, s/2) + T(n - m + 1, s/2) + O(n + s),$$

where n denotes $|U|$ and s denotes $shifts(B)$ for the input block $B = (U, V)$. Solving this recurrence, we obtain the desired running time. □

Lemma 11. Let B be a block (U, V). Then $\Delta_L^*(B)$ can be computed in $O(|V| \log^2 |U|)$ time given $loser_L(B)$. Similarly, $\Delta_R^*(B)$ can be computed within the same time bound given $loser_R(B)$.

Proof. We address the computation of $\Delta_L^*(B)$, which relies on Lemma 8. The computation of $\Delta_R^*(B)$ is symmetric, and relies on Lemma 9.

Let α denote $loser_L(B)$ and let β^* denote $\Delta_L^*(B)$. We now give a divide-and-conquer algorithm that computes β^*. If $|U| \leq 2$, then β^* can be computed trivially in $O(shifts(B))$ time; otherwise, we proceed as follows. Let m denote $\lceil |U|/2 \rceil$. Let B_1 denote the block $subBids(B, 1, m)$ and let B_2 denote the block $subBids(B, m+1, |U|)$. Let α_1 denote $loser_L(B_1)$ and let β_1^* denote $\Delta_L^*(B_1)$. Let α_2 denote $loser_L(B_2)$, let β_2^* denote $\Delta_L^*(B_2)$, and let β_2 denote $\Delta_L(B_2)$. Lemma 8 and the claim in the beginning of the proof of Lemma 10 imply that there exists an integer t' such that $0 \leq t' \leq shifts(B)$, $\alpha[t] = \alpha_1[t]$ for $t' < t \leq shifts(B)$, and $\alpha[t] = \alpha_2[t] + m - 1$ for $1 \leq t \leq t'$; in what follows let t^* denote the largest such integer. Then, by (L2′),

$$\beta^*[t] = \begin{cases} \beta_1^*[t] + \beta_2[t] + w(U[m+1], V[m+t-1]) & \text{if } t^* < t \leq shifts(B) \\ \beta_2^*[t] & \text{if } 1 \leq t \leq t^*. \end{cases}$$

Thus, it remains to compute $\beta_1^*[t^*+1 : \mathit{shifts}(B)]$, $\beta_2^*[1 : t^*]$, and $\beta_2[t^*+1 : \mathit{shifts}(B)]$. Note that $\beta_1^*[t^*+1 : \mathit{shifts}(B)]$ is equal to $\Delta_L^*(B_1')$ for the block $B_1' = \mathit{subShifts}(B_1, t^*+1, \mathit{shifts}(B))$, so we compute it recursively. Similarly, $\beta_2^*[1 : t^*]$ is equal to $\Delta_L^*(B_2')$ for the block $B_2' = \mathit{subShifts}(B_2, 1, t^*)$, so we compute it recursively. Finally, $\beta_2[t^*+1 : \mathit{shifts}(B)]$ is equal to $\Delta_L(B_2'')$ for the block $B_2'' = \mathit{subShifts}(B_2, t^*+1, \mathit{shifts}(B))$, and we compute it in $O(|V| \log |U|)$ time by Lemma 7.

The overall running time satisfies the recurrence

$$T(n, s) \le T(n/2, t) + T(n/2, s-t) + O((n+s)\log n),$$

where n denotes $|U|$ and s denotes $\mathit{shifts}(B)$ for the input block $B = (U, V)$. Solving this recurrence, we obtain the desired running time. $\square$

5.3.2 Superblock-Based Ordered Matching

In this section, we introduce a data structure called a *superblock-based ordered matching* (*SOM*). A SOM represents an ordered matching M by maintaining a superblock S such that $\mathit{matching}(S) = M$, where S is stored as a list of block data structures as described in Sect. 5.3.1.

Theorem 2. The SOM has initialization cost $O(n \log^2 n)$, bid insertion cost $O(\sqrt{n} \log^2 n)$, and dump cost $O(n)$.

Theorem 2 states the main result of our paper, and is proved in Sect. 5.3.4. We first briefly mention key performance-related properties of the SOM that are used for our efficient implementation of Alg. 2. Throughout the rest of this paragraph, let n denote the size of the matching represented by the SOM. We group the operations performed during Alg. 2 into three categories: $\Delta_L(M'')$, $\Delta_R(M'')$, $\Delta_L^*(M'')$, $\Delta_R^*(M'')$, $\mathit{loser}_L(M'')$, and $\mathit{loser}_R(M'')$ queries for submatchings M'' of M; the remaining operations performed in lines 1 through 29; the SWAP operation. It is easy to see that Alg. 2 does not modify the superblock, except during SWAP at line 30. When SWAP modifies the superblock, existing blocks are not modified; rather, some existing blocks are deleted, and some newly constructed blocks are inserted. Since the SOM stores the superblock as a list of block data structures as described in Sect. 5.3.1, all the values in the auxiliary tables $\Delta_L(B)$, $\Delta_R(B)$, $\Delta_L^*(B)$, $\Delta_R^*(B)$, $\mathit{loser}_L(B)$, and $\mathit{loser}_R(B)$ are available for each block B. Thus, the queries for $\Delta_L(M'')$, $\Delta_R(M'')$, $\Delta_L^*(M'')$, $\Delta_R^*(M'')$, $\mathit{loser}_L(M'')$, and $\mathit{loser}_R(M'')$ for each $1 \le i \le |S|$ can be answered in constant time, where M'' denotes $\mathit{matching}(S[i], \mathit{shift}(S, i))$. It is easy to see by inspecting the code of Alg. 2 that the number of such queries is proportional to the number of blocks in the superblock. Furthermore, the running time of all the remaining operations performed in lines 1 through 29 is proportional to the maximum of (1) the number of blocks in the superblock, and (2) the maximum number of bids in any single block. We define the blocks in a SOM so that each block has $\Theta(\sqrt{n})$ bids and $\Theta(\sqrt{n})$ items, yielding a $\Theta(\sqrt{n})$-time implementation of lines 1 through 29, and so that SWAP can be implemented by constructing at most a constant number of blocks. Later in this section, we formally define the SOM, and we introduce two invariants that are related to these requirements. Then in Sections. 5.3.3 and 5.3.4, we present a detailed $O(\sqrt{n} \log^2 n)$-time implementation of SWAP for the SOM that constructs at most a constant number of blocks. We begin with some useful definitions.

For any non-empty ordered matching M, we define $\mathit{slice}(M)$ as $\lceil \sqrt{n} \rceil$ where n denotes $|M|$, and we find it convenient to overload slice so that $\mathit{slice}(S)$ denotes $\mathit{slice}(\mathit{matching}(S))$.

Let S be a superblock and let B be a block that belongs to S. Then we define $time(S, B)$ as $\min(\mathit{shift}(S, B), \mathit{shifts}(B) - \mathit{shift}(S, B) + 1)$. Observe that $matching(B, t)$ is well-defined for all t such that $\mathit{shift}(S, B) - time(S, B) < t < \mathit{shift}(S, B) + time(S, B)$.

We now introduce two invariants that we maintain regarding the performance-related concerns mentioned above. The first invariant ensures that the number of bids in each block is at least $\sqrt{n}$ and less than $2\lceil\sqrt{n}\rceil$, thus there are at most $\sqrt{n}$ blocks, where n denotes $size(S)$. The second invariant ensures that there are not too many blocks B in the superblock S whose time $time(S, B)$ is low, thus SWAP does not require more than a constant number of block constructions. We now formally define these two invariants.

For any superblock S, we define the predicate $P(S)$ to hold if for each block (U, V) in S, $\mathit{slice}(S) \le |U| < 2 \cdot \mathit{slice}(S)$.

For any superblock S, we define the predicate $Q(S)$ to hold if for any ℓ such that $1 \le \ell \le \mathit{slice}(S)$, there are at most ℓ interior blocks B of S such that $time(S, B) \le \ell$.

We say that a superblock S is ***nice*** if $P(S) \wedge Q(S)$.

We say that an ordered matching data structure $\mathcal{D}$ is a *superblock-based ordered matching* (***SOM***) if it represents an ordered matching M by maintaining a nice superblock S such that $matching(S) = M$, and the superblock S is stored as a list of block data structures that are described earlier in Sect. 5.3.1.

5.3.3 Block-Level Operations

Having defined the SOM, it remains to show how to implement SWAP efficiently on the SOM. We describe SWAP by means of four kinds of block-level operations: $refresh$, $split$, $merge$, and $exchange$. As stated earlier, the primary goal of SWAP is to update the matching, and $exchange$ establishes that. The other three operations, $refresh$, $split$, and $merge$, do not alter the matching; the purpose of these operations is to maintain the two invariants defined in Sect. 5.3.2. The common goal of the $split$ and $merge$ operations is to keep the number of bids in each block of a superblock S within a constant factor of $\mathit{slice}(S)$, and to keep $|S|$ at most $\mathit{slice}(S)$. In addition, for any block B that is created by any of these four block-level operations on a superblock S, $\mathit{shifts}(B)$ is within a constant factor of $\mathit{slice}(S)$. We define here what each of these operations establishes and we outline how these operations are chained together in order to achieve a SWAP implementation, we defer the analysis of the running times to Sect. 5.3.4. We start with some useful definitions.

All of the four block-level operations operate by replacing one or two existing blocks with one or two new blocks. We now outline how the new blocks are chosen by these operations. The choice of the bid set of a new block directly depends on the type of the operation: it is equal to the bid set of the block to be replaced by a $refresh$, or it is one of the two halves of the bid set of the block to be replaced by a $split$, or it is the union of the bid sets of the two blocks to be replaced by a $merge$, or one bid is removed and/or one bid is added to the bid set of a block to be replaced by a $exchange$. The exact details are given in the paragraphs below that introduce the individual operations. Given the bid set of the block that is to be created, the choice of the item set of the block depends only on the matching that the superblock resulting from the operation represents. Thus, any new block that the block-level operations create can be expressed as a function of the resulting matching M and the set U of the bids that are involved in the block. This function $\mathit{fresh}(M, U)$ is defined as follows.

Let M be an ordered matching and let U be a contiguous subset of $\mathit{bids}(M)$. Let i denote

$\mathit{index}(U[1], \mathit{bids}(M))$ and let V denote $\mathit{items}(M)$. Let $U_{\prec}$ denote $\{u' \mid u' \in \mathit{bids}(M) \wedge u' < U[1]\}$ and let $U_{\succ}$ denote $\{u' \mid u' \in \mathit{bids}(M) \wedge u' > U[|U|]\}$ (note that $|U_{\prec}| = i - 1$ and $|U_{\succ}| = |M| - |U| - |U_{\prec}|$). Then, we define $\mathit{fresh}(M, U)$ as the block $(U, V[i - r : i + |U| - 1 + r])$ where r denotes $2 \cdot \min(\mathit{slice}(M), |U_{\prec}|, |U_{\succ}|)$. For any superblock S and any block B in S that is equal to $\mathit{fresh}(\mathit{matching}(S), U)$ for some U, we make the following two observations: (1) $\mathit{time}(S, B) = \max_{S'} \mathit{time}(S', B)$ where the maximum is taken over all possible superblocks that B can be a part of; (2) $\mathit{time}(S, B) = 1 + \mathit{slice}(S)$ unless the degenerate condition $\min(|U_{\prec}|, |U_{\succ}|) < \mathit{slice}(S)$ holds, where $U_{\prec}$ and $U_{\succ}$ are defined as earlier in this paragraph. It will be explained later that, at the end of each bid insertion, the degenerate condition mentioned in the preceding observation only holds for the boundary blocks.

We are now ready to introduce the four block-level operations that the SOM performs to modify the superblock that it maintains. In order to define what these operations establish, we introduce a function for each operation that takes a superblock as input (with additional arguments for $\mathit{exchange}$) and returns another one.

For any superblock S, we define $\mathit{refresh}(S)$ as the superblock that is identical to S except that, if it exists, the block $B = (U, V)$ among all interior blocks with the lowest $\mathit{time}(S, B)$, breaking ties by choosing the block with the lowest index, is replaced with $\mathit{fresh}(\mathit{matching}(S), U)$.

For any superblock S, we define $\mathit{split}(S)$ as the superblock that is identical to S except that, if it exists, the block $B = (U, V)$ with the lowest index among the ones satisfying $|U| \geq 2 \cdot \mathit{slice}(S)$ is replaced with two blocks $\mathit{fresh}(\mathit{matching}(S), U[\,:m])$ and $\mathit{fresh}(\mathit{matching}(S), U[m + 1:\,])$, where m denotes $\lceil |U| / 2 \rceil$.

For any superblock S, we define $\mathit{merge}(S)$ as the superblock that is identical to S except that, if it exists, the block $B = (U, V)$ with the lowest index among the ones satisfying $|U| < \mathit{slice}(S)$, and the block $B' = (U', V')$ with the lowest index among the at most two that are adjacent to B, are replaced with a single block $\mathit{fresh}(\mathit{matching}(S), U \cup U')$.

It is easy to see that $\mathit{matching}(\mathit{refresh}(S))$, $\mathit{matching}(\mathit{split}(S))$, and $\mathit{matching}(\mathit{merge}(S))$ are all equal to $\mathit{matching}(S)$. However, the following function returns a superblock that represents a matching that is different than the one its input represents, by exchanging an existing bid for a new bid.

Let S be a superblock such that $\mathit{time}(S, B) > 1$ for each interior block B in S, let u^* be a bid that belongs to $\mathit{bids}(S)$, let u be a bid that does not belong to $\mathit{bids}(S)$, and let M denote $\mathit{matching}(\mathit{bids}(S) - u^* + u, \mathit{items}(S))$. Then we define $\mathit{exchange}(S, u^*, u)$, which returns a superblock that represents M and that is identical to S with the exception of at most two blocks, as follows: let $B^* = (U^*, V^*)$ denote the block in S that contains u^*; let $B^\dagger = (U^\dagger, V^\dagger)$ denote the block with the lowest index among the ones in S satisfying that $U^\dagger + u$ is a contiguous subset of $\mathit{bids}(S) + u$; if $B^* = B^\dagger$, then $\mathit{exchange}(S, u^*, u)$ is identical to S except that B^* is replaced with $\mathit{fresh}(M, U^* - u^* + u)$, otherwise, $\mathit{exchange}(S, u^*, u)$ is identical to S except that B^* is replaced with $\mathit{fresh}(M, U^* - u^*)$ and $B^\dagger$ is replaced with $\mathit{fresh}(M, U^\dagger + u)$.

In order to justify that $\mathit{exchange}(S, u^*, u)$ returns a valid superblock that represents the desired matching, we now compare $\mathit{matching}(S)$ with the desired matching from the perspectives of bids and blocks. In what follows, let S be a superblock and let u^* and u be two bids such that $\mathit{exchange}(S, u^*, u)$ is well-defined, and let S' denote $\mathit{exchange}(S, u^*, u)$. Let U denote $\mathit{bids}(S)$, let V denote $\mathit{items}(S)$, let M denote $\mathit{matching}(S)$, and let M' denote the desired matching $\mathit{matching}(U - u^* + u, V)$. Let k^* denote $\mathit{index}(u^*, U)$ and let k denote $\mathit{index}(u, U + u)$. Let B^* and $B^\dagger$ be the blocks defined as in the preceding paragraph, let ℓ^* denote $\mathit{index}(B^*, S)$, and let

$\ell^\dagger$ denote $\mathit{index}(B^\dagger, S)$. Comparing M with M' from the perspective of bids, it is straightforward to see that, if $k^* < k$ (resp., $k^* \geq k$) then u is assigned to $V[k-1]$ (resp., $V[k]$) in M', and for each i such that $k^* < i \leq k-1$ (resp., $k \leq i < k^*$), the bid $U[i]$, which is assigned to $V[i]$ in M, is ***shifted left*** (resp., ***right***) ***by*** $\mathit{exchange}(S, u^*, u)$, i.e., is assigned to $V[i-1]$ (resp., $V[i+1]$) in M'. Each bid that belongs to U but that is neither shifted left nor right by $\mathit{exchange}(S, u^*, u)$ is assigned to the same item in both M and M', except for u^* which is unassigned in M'. Comparing M with M' from the perspective of the blocks, it is easy to see that, if $\ell^* < \ell^\dagger$ (resp., $\ell^* > \ell^\dagger$), then each bid in each block $S[i]$ with a block index $\ell^* < i < \ell^\dagger$ (resp., $\ell^* > i > \ell^\dagger$) is shifted left (resp., right), hence, we say that the block $S[i]$ is ***shifted left*** (resp., ***right***) ***by*** $\mathit{exchange}(S, u^*, u)$. For each block B that is shifted left (resp., right) by $\mathit{exchange}(S, u^*, u)$, $\mathit{shift}(S', B) = \mathit{shift}(S, B) - 1$ (resp., $+1$), and for each block B that belongs to both S and S' but that is neither shifted left nor right, $\mathit{shift}(S', B) = \mathit{shift}(S, B)$. Thus, since B^* and $B^\dagger$ are replaced with new blocks in S' and since $\mathit{time}(S, B) > 1$ for each interior block B that belongs to S, $\mathit{matching}(B, \mathit{shift}(S', B))$ is well-defined for each block B that belongs to S'. Hence, S' is a valid superblock and it is easy to see that $\mathit{matching}(S') = M'$.

We now define a function via $\mathit{refresh}$, $\mathit{exchange}$, split, and merge that proves to be useful in our goal of efficiently implementing SWAP on the SOM.

For any nice superblock S, any bid u^* that belongs to $\mathit{bids}(S)$, and any bid u that does not belong to $\mathit{bids}(S)$, we define $\mathit{swap}(S, u^*, u)$ as $\mathit{split}(\mathit{merge}(\mathit{split}(\mathit{exchange}(\mathit{refresh}(S), u^*, u))))$. The following lemma suggests using $S = \mathit{swap}(S, u^*, u)$ as an implementation of SWAP(S, u^*, u) since it modifies the superblock as desired while maintaining the predicates $P(S)$ and $Q(S)$.

Lemma 12. Let S be a nice superblock, let u^* be a bid that belongs to $\mathit{bids}(S)$, and let u be a bid that does not belong to $\mathit{bids}(S)$. Then $\mathit{swap}(S, u^*, u)$ is a nice superblock, and $\mathit{matching}(\mathit{swap}(S, u^*, u)) = \mathit{matching}(\mathit{bids}(S) + u - u^*, \mathit{items}(S))$.

Proof. Let S_1 denote $\mathit{refresh}(S)$. By the definition of $\mathit{refresh}$, $\mathit{matching}(S_1) = \mathit{matching}(S)$. Since S is nice, $P(S)$ and $Q(S)$ holds. Then, since $\mathit{refresh}$ does not change the bid partitioning implied by the superblock, $P(S_1)$ holds. And since $\mathit{refresh}$ only replaces an interior block B with the lowest $\mathit{time}(S, B)$, if it exists, with a block B' having $\mathit{time}(S_1, B') = \mathit{slice}(S_1) + 1$, it is straightforward to see that a stronger $Q(S_1)$ (thus implying $Q(S_1)$) holds, which we define next as $Q^+(S_1)$.

For any superblock S, we define the predicate $Q^+(S)$ to hold if for any ℓ such that $1 \leq \ell \leq \mathit{slice}(S)$, there are at most $\ell - 1$ interior blocks B of S such that $\mathit{time}(S, B) \leq \ell$.

Let S_2 denote $\mathit{exchange}(S_1, u^*, u)$. Since $Q^+(S_1)$ implies that $\mathit{time}(S_1, B) > 1$ for each interior block B in S_1, $\mathit{exchange}(S_1, u^*, u)$ is well-defined, and hence $\mathit{matching}(S_2) = \mathit{matching}(\mathit{bids}(S) + u - u^*, \mathit{items}(S))$. Now, if only one block is replaced during $\mathit{exchange}(S_1, u^*, u)$, then the following claims hold: $P(S_2)$ since the replaced block has the same number of bids; $Q^+(S_2)$, and thus $Q(S_2)$ since no blocks are shifted; hence, $\mathit{split}(\mathit{merge}(\mathit{split}(S_2)))$ is equal to S_2, which is a nice superblock with the desired matching, and we are done. If two blocks are replaced, then it is straightforward to see that the following claims hold by the definition of $\mathit{exchange}$: $Q(S_2)$ since $Q^+(S_1)$ and the fact that $|\mathit{shift}(S_2, B) - \mathit{shift}(S_1, B)| \leq 1$ for each surviving block B, where the latter fact is a result of the shifts, as described while arguing the correctness of $\mathit{exchange}$; $P(S_2)$ except that one block may be undersized by one bid and one block may be oversized by one bid.

Let S_3 denote $\mathit{split}(S_2)$. It is straightforward to see that the following claims hold by the definition of split: $\mathit{matching}(S_3) = \mathit{matching}(S_2)$; $Q(S_3)$; $P(S_3)$ except that one block may be undersized by one bid.

Let S_4 denote $\mathit{merge}(S_3)$. It is straightforward to see that the following claims hold by the definition of merge: $\mathit{matching}(S_4) = \mathit{matching}(S_3)$; $Q(S_4)$; $P(S_4)$ except that one block may be oversized with total number of bids at most $3 \cdot \mathit{slice}(S) - 2$.

Let S_5 denote $\mathit{split}(S_4)$. It is straightforward to see that the following claims hold by the definition of split: $\mathit{matching}(S_5) = \mathit{matching}(S_4)$; $Q(S_5)$; $P(S_5)$.

By the preceding observations, we see that S_5, which is equal to $\mathit{swap}(S, u^*, u)$, is a nice superblock and $\mathit{matching}(S_5) = \mathit{matching}(S_2) = \mathit{matching}(\mathit{bids}(S) + u - u^*, \mathit{items}(S))$, as required. □

5.3.4 Implementation of SWAP and Time Complexity

We now complete the discussion of the fast bid insertion on the SOM by describing how to efficiently implement SWAP as $S = \mathit{swap}(S, u^*, u)$, as described in the preceding section, and by proving Theorem 2, which summarizes our results. Recall that the goal of SWAP(S, u^*, u) is, given a superblock S, a bid u^* that belongs to $\mathit{bids}(S)$, and a bid u that does not belong to $\mathit{bids}(S)$, to return a superblock that represents $\mathit{matching}(\mathit{bids}(S) + u - u^*, \mathit{items}(S))$; since a SOM always maintains a nice superblock, we require the input S and the returned superblock to be nice.

Lemma 13. Let $\mathcal{D}$ be a SOM, let S denote the superblock maintained by $\mathcal{D}$, and let n denote $\mathit{size}(S)$. Then, SWAP(S, u^*, u) on $\mathcal{D}$ can be implemented as $S = \mathit{swap}(S, u^*, u)$ in $O(\sqrt{n}\log^2 n)$ time.

Proof. Lemma 12 implies that $S = \mathit{swap}(S, u^*, u)$ is a correct implementation of SWAP(S, u^*, u), and it satisfies the requirement that $\mathcal{D}$ maintains a nice superblock. We now argue the running time. It is straightforward to see that each of the operations $\mathit{refresh}$, $\mathit{exchange}$, split, and merge can be implemented in $O(\sqrt{n}\log^2 n)$ time; it takes $O(|S|) = O(\sqrt{n})$ time to identify the block/blocks to be replaced, since $P(S)$ implies that $|S|$ is $\Theta(\sqrt{n})$; it takes $O(\sqrt{n}\log^2 n)$ time to construct each block $B = (U, V)$ by Theorem 1, since $P(S)$ and the definition of fresh implies that $|U|$, $|V|$, and $\mathit{shifts}(B)$ are $O(\sqrt{n})$; there are at most two block constructions per operation. □

Proof of Theorem 2. When initialized with an ordered matching M with size n, the SOM constructs $\Theta(\sqrt{n})$ blocks, each taking $O(\sqrt{n}\log^2 n)$ time.

Bid insertion on the SOM can be implemented as Alg. 2 in $O(\sqrt{n}\log^2 n)$ time since Lemma 13 shows that SWAP(S, u^*, u) can be implemented in $O(\sqrt{n}\log^2 n)$ time, and as argued in Sect. 5.3.2, the remaining operations in Alg. 2 can be implemented in $O(\sqrt{n})$ time, where n denotes the size of the superblock that the SOM maintains.

It is straightforward to implement dump by scanning over all the blocks and constructing a list representation of the matching in $O(n)$ time where n denotes the size of the matching.

□

We conclude the discussion of the fast bid insertion on the SOM with some remarks. It is possible to support constant-time queries that return the bid matched to a given item with some additional bookkeeping. Queries to find whether a bid is matched or not, and if so, to return the matched item, can be implemented in logarithmic time by performing binary search. Finally, it is possible to initialize the SOM with a matching consisting of all dummy bids, each with intercept zero and slope zero, in linear time, since all of the weights involving those bids are zero, and thus it is trivial to construct the blocks.

6 Computation of the VCG Prices

In this section, we show how to extend the SOM to maintain the VCG prices as each bid is inserted. Section 6.1 introduces some useful definitions. Section 6.2 extends the incremental framework of Sect. 3 to compute the VCG prices. Section 6.3 presents a basic algorithm within the framework of Sect. 6.2. Section 6.4 describes how to extend the data structure of Sect. 5 and presents a fast emulation of the algorithm of Sect. 6.3.

6.1 Preliminaries

In many settings including the unit-demand auction settings, every efficient and strategyproof mechanism is a so-called Groves mechanism, one that aligns the incentives of all players with the goal of maximizing social welfare [7]. In addition to being efficient and strategyproof, the special case of the Groves mechanism that employs the Clarke pivot rule for payments, what is commonly referred to as the VCG mechanism, is individually rational and has no positive transfers, i.e., the mechanism does not make payments to the bidders. The VCG allocation is an assignment that maximizes the social welfare, and hence corresponds to an MWM of the bipartite graph. The VCG prices can be characterized in various equivalent ways. In this section, we use the characterization that identifies the VCG prices as the minimum stable price vector [11]. Another characterization of the VCG prices follows directly from the Clarke pivot rule; the price of the item that is assigned to a bidder u is the decrease in the social welfare of others caused by the participation of u.

We begin by reviewing some standard definitions and results that prove to be useful. We state these results for UDALEWs; however, they hold for general unit-demand auctions. (The reader is referred to [15, Chapter 8] for a thorough discussion and omitted proofs.)

For a UDALEW $A = (U, V)$, a *surplus vector* s assigns a real value $s[i]$ to each bid $U[i]$ in U, a *price vector* p assigns a real value $p[j]$ to each item $V[j]$ in V, and an *outcome* is a triple (M, s, p) such that s is a surplus vector, p is a price vector, and M is a matching of A.

An outcome (M, s, p) of a UDALEW (U, V) is *feasible* if $\sum_{1\le i\le |U|} s[i] + \sum_{1\le j\le |V|} p[j] = w(M)$. For any feasible outcome (M, s, p), we say that the pair of vectors (s, p) and the matching M are *compatible*.

Let $A = (U, V)$ be a UDALEW. We say that a bid $U[i]$ (resp., item $V[j]$) blocks an outcome (M, s, p) of A if $s[i] < 0$ (resp., $p[j] < 0$). We say that a bid-item pair $(U[i], V[j])$ blocks an outcome (M, s, p) of A if $s[i] + p[j] < w(U[i], V[j])$. If no bid, item, or bid-item pair blocks an outcome (M, s, p) of A , then we say that the outcome (M, s, p) is *stable*, and that the payoff (s, p) is *stable* with M. For any stable outcome (M, s, p) of A, the following are known: M is an MWM of A; $s[i] + p[j] = w(U[i], V[j])$ for all $(U[i], V[j])$ matched in M; $s[i] = 0$ for all $U[i]$ unmatched in M; $p[j] = 0$ for all $V[j]$ unmatched in M. It is also known that any MWM is compatible with any stable payoff. Thus, given the price vector p of a stable outcome of A, the corresponding surplus vector s is uniquely determined by the following equation, where M denotes an arbitrary

MWM of A:

$$s[i] = \begin{cases} w(U[i], V[j]) - p[j] & \text{if } V[j] \text{ is assigned to } U[i] \text{ in } M \\ 0 & \text{if } U[i] \text{ is left unassigned in } M. \end{cases} \tag{3}$$

For any stable payoff (s, p) of A, we say that p is a *stable price vector* of A.

In the remainder of the paper, we write an outcome as a pair (M, p) rather than a triple (M, s, p), and it is understood that the associated surplus vector s is given by (3).

It is known that the stable price vectors of a UDALEW form a lattice [16]. Hence, there is a unique stable price vector that is componentwise less than or equal to any other stable price vector; this minimum stable price vector corresponds to the VCG prices [11]. Thus, for a UDALEW A, we refer to a stable outcome (M, p) of A as a *VCG outcome of* A if p is the VCG prices. In the remainder of the paper, the inequality operators denote componentwise inequalities when they are used on price vectors.

6.2 Incremental Framework with Prices

In this section, we present an incremental framework for the problem of finding a VCG outcome of a UDALEW; we follow the approach of Sect. 3. In order to utilize the algorithms of Sections 4 and 5, we assume that the UDALEW (U, V) for which we seek a VCG outcome is enlarged by adding $|V|$ dummy bids, each with intercept zero and slope zero, so that, by Corollary 1, we can restrict our attention to ordered MWMCMs. Hence, in the remainder of the paper, for any outcome (M, p) of a UDALEW A, we impose the condition that M is an ordered MWMCM of A.

Let $A = (U, V)$ be a UDALEW such that $|U| \geq |V|$. Then for any VCG outcome (M, p) of A and any bid u that does not belong to U, we define $\mathit{insert}(M, p, u)$ as the stable outcome (M', p') of the UDALEW $A' = (\mathit{bids}(M) + u, \mathit{items}(M))$ where M' is $\mathit{insert}(M, u)$ and p' is the minimum stable price vector of A' such that $p' \geq p$; the existence and uniqueness of such p' is implied by the lattice property of the stable price vectors.

The following lemma is at the core of our incremental framework. The proof follows from [15, Proposition 8.17] and from Lemma 2.

Lemma 14. Let $A = (U, V)$ be a UDALEW such that $|U| \geq |V|$ and let u be a bid that does not belong to U. Then for any VCG outcome (M, p) of A, $\mathit{insert}(M, p, u)$ is a VCG outcome of the UDALEW $(U + u, V)$.

We want to devise a data structure that maintains a dynamic outcome (M, p). The data structure is initialized with a VCG outcome (M', p') of some UDALEW. The characterization of the data structure is analogous to that of Sect. 3, except that bid insertion transforms the data structure to represent $\mathit{insert}(M, p, u)$, and dump returns a list representation of both M and p.

Lemma 15. Let $\mathcal{D}$ be an outcome data structure with initialization cost $f(n)$, bid insertion cost $g(n)$, and dump cost $h(n)$. Let A be a UDALEW (U, V). Then a VCG outcome of A can be computed in $O(f(|V|) + (|U| - |V|) \cdot g(|V|) + h(|V|))$ time.

Proof. Let U' be a set of $|V|$ dummy bids, each with intercept zero and slope zero. Let $\langle u_1, \ldots, u_{|U|} \rangle$ be an arbitrary permutation of the bids in U. For any integer i such that $0 \leq i \leq |U|$, let U_i denote

$U' \cup \{u_1, \ldots, u_i\}$. Remark: $U_0 = U'$ and $U_{|U|} = U \cup U'$. We now show how to use $\mathcal{D}$ to find a VCG outcome of the UDALEW $(U_{|U|}, V)$, which is also a VCG outcome of A. We initialize $\mathcal{D}$ with the outcome consisting of the ordered matching $M_0 = \mathit{matching}(U_0, V)$ and the all-zeros price vector p_0; note that (M_0, p_0) is a VCG outcome of the UDALEW (U_0, V). Then we iteratively insert bids $u_1, \ldots, u_{|U|}$. Let (M_i, p_i) denote the outcome associated with $\mathcal{D}$ after i iterations, $1 \leq i \leq |U|$. Then, by induction on i, Lemma 14 and the definition of bid insertion together imply that (M_i, p_i) is a VCG outcome of the UDALEW (U_i, V). Thus, a dump on $\mathcal{D}$ after completing all iterations returns a VCG outcome of A. The whole process runs in the required time since we perform one initialization, $|U|$ bid insertions, and one dump. $\square$

In Sect. 6.3, we give a linear-time bid insertion algorithm assuming an array representation of the ordered matching and the price vector. Building on the concepts introduced in Sect. 6.3 and the SOM of Sect. 5, Sect. 6.4 develops an outcome data structure with initialization cost $O(n \log^2 n)$, bid insertion cost $O(\sqrt{n} \log^2 n)$, and dump cost $O(n)$. The results of Sect. 6.4, together with Lemma 15, imply an $O(m\sqrt{n} \log^2 n)$ time bound for computing a VCG outcome.

6.3 A Basic Algorithm with Prices

In this section, we describe a linear-time implementation of $\mathit{insert}(M, p, u)$ given an array representation of the ordered matching M and the price vector p. In Sect. 6.3.1, we give a characterization of the price component of $\mathit{insert}(M, p, u)$; in Sect. 6.3.2, we show how to compute $\mathit{insert}(M, p, u)$ based on this characterization. We start with some useful definitions and lemmas.

For any ordered matching M, we make the following definitions, where U denotes $\mathit{bids}(M)$ and V denotes $\mathit{items}(M)$: $V[j]$ is the *match of* $U[j]$ *in* M for $1 \leq j \leq |M|$; $V[j-1]$ is the *left-adjacent item of* $U[j]$ *in* M for $1 < j \leq |M|$; $V[j+1]$ is the *right-adjacent item of* $U[j]$ *in* M for $1 \leq j < |M|$; a bid-item pair consisting of a bid and its left-adjacent (resp., right-adjacent) item in M, i.e., $(U[j], V[j-1])$ (resp., $(U[j], V[j+1])$), is a *left-adjacent* (resp., *right-adjacent*) *pair in* M; a left-adjacent or a right-adjacent pair is also called an *adjacent pair*.

The following lemma plays a key role in our algorithm; it suggests that we focus on adjacent pairs to obtain a stable price vector.

Lemma 16. Let $A = (U, V)$ be a UDALEW such that $|U| \geq |V|$ and let M be an ordered MWMCM of A. Let p be a price vector such that no adjacent pair in M blocks the outcome (M, p) of A. Let u be a bid in U.

1. For any index i such that $1 \leq i < |M|$ and $u.\mathit{slope} \leq U[i].\mathit{slope}$, if $(u, V[i])$ does not block (M, p), then $(u, V[i+1])$ does not block (M, p).

2. For any index i such that $1 < i \leq |M|$ and $u.\mathit{slope} \geq U[i].\mathit{slope}$, if $(u, V[i])$ does not block (M, p), then $(u, V[i-1])$ does not block (M, p).

Proof. We prove the first claim; the second claim is symmetric. Let i be an index such that $1 \leq i < |M|$, $u.\mathit{slope} \leq U[i].\mathit{slope}$, and $(u, V[i])$ does not block (M, p). Since $V[i+1].\mathit{quality} \geq V[i].\mathit{quality}$ and $U[i].\mathit{slope} \geq u.\mathit{slope}$, we have

$$\begin{aligned} w(U[i], V[i+1]) - w(U[i], V[i]) &= U[i].\mathit{slope} \cdot (V[i+1].\mathit{quality} - V[i].\mathit{quality}) \\ &\geq u.\mathit{slope} \cdot (V[i+1].\mathit{quality} - V[i].\mathit{quality}) \\ &= w(u, V[i+1]) - w(u, V[i]). \end{aligned} \tag{4}$$

Since $(U[i], V[i+1])$ does not block (M, p), we have $p[i+1] - p[i] \geq w(U[i], V[i+1]) - w(U[i], V[i])$, and by (4), $w(u, V[i]) - p[i] \geq w(u, V[i+1]) - p[i+1]$. Since $(u, V[i])$ does not block (M, p), we know that the surplus of u is at least $w(u, V[i]) - p[i]$; combining this with the inequality established in the preceding sentence, we deduce that the surplus of u is at least $w(u, V[i+1]) - p[i+1]$, as required. □

For any outcome (M', p'), we make the following definitions, where U' denotes $bids(M')$ and V denotes $items(M')$: a bid u in U' is *left-tight* (resp., *right-tight*) if it is indifferent between being assigned to its match in M' or being assigned to its left-adjacent (resp., right-adjacent) item in M'; for any two indices j_1 and j_2 such that $1 \leq j_1 < j_2 \leq |M'|$, the interval $[j_1, j_2]$ of (M', p') is *left-tight* if each bid $U'[j]$ for $j_1 < j \leq j_2$ is left-tight, and symmetrically, the interval $[j_1, j_2]$ of (M', p') is *right-tight* if each bid $U'[j]$ for $j_1 \leq j < j_2$ is right-tight.

For any outcome (M', p'), it is straightforward to observe the following, where U' denotes $bids(M')$ and V denotes $items(M')$: if a bid $U'[j]$ is left-tight, then

$$p'[j-1] = p'[j] - w(U'[j], V[j]) + w(U'[j], V[j-1]), \tag{5}$$

and hence, if an interval $[j_1, j_2]$ is left-tight, then

$$p'[j_1] = p'[j_2] + \Delta_L(M'[j_1 : j_2]) + w(U'[j_1], V[j_1]); \tag{6}$$

symmetrically, if a bid $U'[j]$ is right-tight, then

$$p'[j+1] = p'[j] - w(U'[j], V[j]) + w(U'[j], V[j+1]), \tag{7}$$

and hence, if an interval $[j_1, j_2]$ is right-tight, then

$$p'[j_2] = p'[j_1] + \Delta_R(M'[j_1 : j_2]) + w(U'[j]_2, V[j_2]). \tag{8}$$

For any ordered matching M' and any real value t, we define $tight_L(M', t)$ (resp., $tight_R(M', t)$) as the price vector p' of the UDALEW (U, V) such that $p'[|V|] = t$ (resp., $p'[1] = t$), and for $j = |V| - 1, \ldots, 1$ (resp., for $j = 2, \ldots, |V|$), $p'[j]$ is defined by (5) (resp., by (7)), where U' denotes $bids(M')$ and V denotes $items(M')$.

Let M' be an ordered matching, let V denote $items(M')$, let p be a price vector for V, and let u^* be a bid that does not belong to $bids(M')$. Then we define $reach_L(M', p, u^*)$ and $reach_R(M', p, u^*)$ as follows. Let j^* denote $index(u^*, bids(M') + u^*)$. If there exists an index j such that $1 \leq j < j^*$ and $p[j : j^* - 1] \leq tight_L(M'[j : j^* - 1], w(u^*, V[j^* - 1]))$, then $reach_L(M', p, u^*)$ is defined as the minimum such j; otherwise, $reach_L(M', p, u^*)$ is defined as j^*. Symmetrically, if there exists an index j such that $j^* \leq j \leq |M|$ and $p[j^* : j] \leq tight_R(M'[j^* : j], w(u^*, V[j^*]))$, then $reach_R(M', p, u^*)$ is defined as the maximum such j; otherwise, $reach_R(M', p, u^*)$ is defined as $j^* - 1$.

In the next section, we use the concepts introduced above to characterize the prices after bid insertion.

6.3.1 Characterization of the Prices After Bid Insertion

In this section, we fix an arbitrary outcome (M, p) that is stable for the UDALEW $(bids(M), items(M))$ and an arbitrary bid u that does not belong to $bids(M)$. In what follows, let U denote $bids(M)$, let V denote $\mathit{items}(M)$, let A denote the UDALEW $(U + u, V)$, let M' denote $\mathit{insert}(M, u)$, let U' denote $\mathit{bids}(M')$, let u^* denote $(U + u) \setminus U'$, let j^* denote $\mathit{index}(u^*, U + u)$, let $j^\dagger_L$ denote $\mathit{reach}_L(M', p, u^*)$, and let $j^\dagger_R$ denote $\mathit{reach}_R(M', p, u^*)$. We first introduce two useful lemmas and then, in Theorem 3, we characterize the prices after bid insertion.

Lemma 17. $M'[\,:j^\dagger_L - 1] = M[\,:j^\dagger_L - 1]$ and $M'[j^\dagger_R + 1:\,] = M[j^\dagger_R + 1:\,]$.

Proof. We prove by contradiction that $M'[\,:j^\dagger_L - 1] = M[\,:j^\dagger_L - 1]$; the proof of the other statement is symmetric. Assume that $M'[\,:j^\dagger_L - 1] \neq M[\,:j^\dagger_L - 1]$ and let k denote the least index such that $M'[k] \neq M[k]$; thus k is less than $j^\dagger_L$, which by definition is at most j^*. We consider two cases.

Case 1: $u \geq u^*$. Then $U'[\,:j^* - 1] = U[\,:j^* - 1]$, which implies $M'[\,:j^* - 1] = M[\,:j^* - 1]$, contradicting our assumption that $M'[\,:j^\dagger_L - 1] \neq M[\,:j^\dagger_L - 1]$.

Case 2: $u < u^*$. Then $u^* = U[j^* - 1]$, $u = U'[k]$, and $U'[k + 1 : j^* - 1] = U[k : j^* - 2]$ since M and M' are ordered matchings with $|M| - 1$ common bids and since $k < j^*$. The claim established below implies that $j^\dagger_L \leq k$, which contradicts $k < j^\dagger_L$, thereby completing the proof.

Claim: $p[k : j^* - 1] \leq \mathit{tight}_L(M'[k : j^* - 1], w(u^*, V[j^* - 1]))$. In what follows, let $q[j^* - 1]$ denote $w(u^*, V[j^* - 1])$ and let $q[j]$ denote $q[j+1] - w(U'[j+1], V[j+1]) + w(U'[j+1], V[j])$ for $k \leq j \leq j^* - 2$. In the remainder of the proof, we show by reverse induction on j that $p[j] \leq q[j]$ for $k \leq j < j^*$; then the claimed inequality follows immediately since the right-hand side is a vector with components $q[k], \ldots, q[j^* - 1]$.

Base case: $j = j^* - 1$. Since $u^* = U[j^* - 1]$ and since (M, p) is stable for the UDALEW $(\mathit{bids}(M), \mathit{items}(M))$, we have $p[j^* - 1] \leq w(u^*, V[j^* - 1]) = q[j^* - 1]$.

Induction step. Let j be an integer such that $k < j < j^*$ and assume $p[j + 1] \leq q[j + 1]$. Since the pair $(U[j], V[j + 1])$ does not block (M, p), we know that $p[j] \leq p[j + 1] - w(U[j], V[j + 1]) + w(U[j], V[j])$. Then, by our assumption that $p[j + 1] \leq q[j + 1]$ and since $U'[j + 1] = U[j]$, we deduce that $p[j] \leq q[j + 1] - w(U'[j + 1], V[j + 1]) + w(U'[j + 1], V[j]) = q[j]$. $\square$

Lemma 18. For any item index j such that $j < j^*$, we have $j^\dagger_L \leq j$ if and only if $p[j] \leq w(u^*, V[j^* - 1]) + \Delta_L(M'[j : j^* - 1]) + w(U'[j], V[j])$. Symmetrically, for any item index j such that $j \geq j^*$, we have $j^\dagger_R \geq j$ if and only if $p[j] \leq w(u^*, V[j^*]) + \Delta_L(M'[j^* : j]) + w(U'[j], V[j])$.

Proof. We only prove the first claim; the proof of the second claim is symmetric. It is easy to see by the definition of $\mathit{reach}_L(M', p, u^*)$ that the claim holds for $j = j^* - 1$. We now show that if the claim holds for some item index j such that $1 < j < j^*$, then it holds for $j - 1$. In what follows, let $q[j]$ denote $w(u^*, V[j^* - 1]) + \Delta_L(M'[j : j^* - 1]) + w(U'[j], V[j])$ for $1 \leq j < j^*$. Let j be an item index such that $1 < j < j^*$ and assume that the claim holds for this index, i.e., $j^\dagger_L \leq j$ if and only if $p[j] \leq q[j]$. In what follows, let p' denote $\mathit{tight}_L(M'[j - 1 : j^* - 1], w(u^*, V[j^* - 1]))$. Note that $q[j] = p'[2]$ by (6), and $q[j - 1] = q[j] - w(U'[j], V[j]) + w(U'[j], V[j - 1]) = p'[1]$ by (L1$'$) and (6). We consider two cases.

Case 1: $j^\dagger_L \leq j$ and $p[j] \leq q[j]$. Since $j^\dagger_L \leq j$, we know that $p[j : j^* - 1] \leq p'[2:\,]$. Thus, $p[j - 1] \leq q[j - 1]$ if and only if $p[j - 1 : j^* - 1] \leq p'$. Hence, $j^\dagger_L \leq j - 1$ if and only if $p[j - 1] \leq q[j - 1]$ by the definition of $\mathit{reach}_L(M', p, u^*)$.

Case 2: $j^\dagger_L > j$ and $p[j] > q[j]$. Then Lemma 17 implies that $U'[j] = U[j]$. The stability of (M, p) implies that $p[j-1] \geq p[j] - w(U[j], V[j]) + w(U[j], V[j-1])$. Then, since $p[j] > q[j]$ and $U'[j] = U[j]$, we conclude that $p[j-1] > q[j] - w(U'[j], V[j]) + w(U'[j], V[j-1]) = q[j-1]$. $\square$

We now characterize a certain price vector given the stable outcome (M, p) and the new bid u, and then state in Theorem 3 that this price vector is the price component of any VCG outcome after insertion of u. We define $\mathit{grow}(M, p, u)$ as the price vector p' of A such that the following conditions hold: $p'[\,:j_L^\dagger - 1] = p[\,:j_L^\dagger - 1]$; if $j_L^\dagger < j^*$, then $p'[j_L^\dagger : j^* - 1] = \mathit{tight}_L(M'[j_L^\dagger : j^* - 1], w(u^*, V[j^* - 1]))$; if $j_R^\dagger \geq j^*$, then $p'[j^* : j_R^\dagger] = \mathit{tight}_R(M'[j^* : j_R^\dagger], w(u^*, V[j^*]))$; $p'[j_R^\dagger + 1 :\,] = p[j_R^\dagger + 1 :\,]$.

Theorem 3. For any VCG outcome (M_0, p_0) of a UDALEW (U_0, V_0) such that $|U_0| \geq |V_0|$ and for any bid u_0 that does not belong to U_0, $\mathit{insert}(M_0, p_0, u_0) = (\mathit{insert}(M_0, u_0), \mathit{grow}(M_0, p_0, u_0))$.

Proof. The proof follows from Lemma 14, which summarizes our incremental framework, and from Lemma 19 below. □

The remainder of this section states and proves Lemma 19 which is used in the proof of Theorem 3. In what follows, let p' denote $\mathit{grow}(M, p, u)$.

Lemma 19. The following claims hold: (1) $p' \geq p$; (2) p' is a stable price vector of A; (3) for any stable price vector p'' of A such that $p'' \geq p$, $p' \leq p''$.

We state two useful lemmas before proving Lemma 19.

Lemma 20. $p'[j] \leq w(U'[j], V[j])$ for $1 \leq j \leq |V|$.

Proof. Since (M, p) is stable for the UDALEW (U, V), we know that $p[j] \leq w(U[j], V[j])$ for $1 \leq j \leq |V|$. Then, $p'[j] \leq w(U'[j], V[j])$ for $1 \leq j < j_L^\dagger$ (resp., $j_R^\dagger < j \leq |V|$) since $p'[\,:j_L^\dagger - 1] = p[\,:j_L^\dagger - 1]$ (resp., $p'[j_R^\dagger + 1 :\,] = p[j_R^\dagger + 1 :\,]$) by the definition of $\mathit{grow}(M, p, u)$, and since $U'[\,:j_L^\dagger - 1] = U[\,:j_L^\dagger - 1]$ (resp., $U'[j_R^\dagger + 1 :\,] = U[j_R^\dagger + 1 :\,]$) by Lemma 17. It remains to show that the claim holds for $j_L^\dagger \leq j \leq j_R^\dagger$. If $j_L^\dagger < j^*$ (resp., if $j_R^\dagger \geq j^*$), then $p'[j^* - 1] = w(u^*, V[j^* - 1]) \leq w(U'[j^* - 1], V[j^* - 1])$ (resp., $p'[j^*] = w(u^*, V[j^*]) \leq w(U'[j^*], V[j^*])$) where the equality holds by the definition of $\mathit{grow}(M, p, u)$ and the inequality holds by the fact that u^* is not matched by the MWMCM M'. Then it is straightforward to see that $p'[j] \leq w(U'[j], V[j])$ for $j = j^* - 2, \ldots, j_L^\dagger$ (resp., $j = j^* + 1, \ldots, j_R^\dagger$) since $p'[j]$ is defined by (5) (resp., by (7)). □

Lemma 21. No adjacent pair in M' blocks the outcome (M', p') of A.

Proof. We start the proof by showing in the following three paragraphs that no adjacent pair in $M'[\,:j^* - 1]$ blocks the outcome (M', p'). The task of showing that no adjacent pair in $M'[j^* :\,]$ blocks (M', p') is symmetric. Then, we complete the proof by showing that if $1 < j^* \leq |M'|$, then neither of the two adjacent pairs in $M'[j^* - 1 : j^*]$ blocks the outcome (M', p').

First, we argue about the adjacent pairs in $M'[\,:j_L^\dagger - 1]$, which is nonempty only if $j_L^\dagger > 1$. Assume that $j_L^\dagger > 1$. Since $p'[\,:j_L^\dagger - 1] = p[\,:j_L^\dagger - 1]$ by the definition of $\mathit{grow}(M, p, u)$ and since $M'[\,:j_L^\dagger - 1] = M[\,:j_L^\dagger - 1]$ by Lemma 17, the stability of (M, p) implies that no pair (adjacent or not) in $M'[\,:j_L^\dagger - 1]$ blocks (M', p').

Second, we argue about the two adjacent pairs in $M'[j_L^\dagger - 1 : j_L^\dagger]$ when $j_L^\dagger < j^*$. Assume that $j_L^\dagger < j^*$. Since $p'[j_L^\dagger - 1] > p'[j_L^\dagger] - w(U'[j_L^\dagger], V[j_L^\dagger]) + w(U'[j_L^\dagger], V[j_L^\dagger - 1])$ by the definition of $j_L^\dagger$, the left-adjacent pair $(U'[j_L^\dagger], V[j_L^\dagger - 1])$ does not block (M', p'). Since $U'[j_L^\dagger - 1]$ is matched to the same item ($V[j_L^\dagger - 1]$) in M and in M' by Lemma 17, and since the right-adjacent pair

$(U'[j_L^\dagger - 1], V[j_L^\dagger])$ does not block (M, p) (by the stability of (M, p)), we conclude that the same pair does not block (M', p') because the definition of $j_L^\dagger$ implies that $p'[j_L^\dagger - 1] = p[j_L^\dagger - 1]$ and $p'[j_L^\dagger] \geq p[j_L^\dagger]$.

Third, we argue about the adjacent pairs in $M'[j_L^\dagger : j^* - 1]$, which is nonempty only if $j_L^\dagger < j^* - 1$. Assume that $j_L^\dagger < j^* - 1$. Let j be an arbitrary index such that $j_L^\dagger < j < j^*$. It is easy to see that the left-adjacent pair $(U'[j], V[j-1])$ does not block (M', p') since $p'[j-1] = p'[j] - w(U'[j], V[j]) + w(U'[j], V[j-1])$ by (5). Since $U'[j-1] < U'[j]$, by an argument similar to the one that is used to derive (4), we deduce that $w(U'[j-1], V[j]) - w(U'[j-1], V[j-1]) \leq w(U'[j], V[j]) - w(U'[j], V[j-1])$, which combined with the equality in the preceding sentence implies that the right-adjacent pair $(U'[j-1], V[j])$ does not block (M', p').

Finally, we complete the proof by showing that if $1 < j^* \leq |M'|$, then neither of the two adjacent pairs in $M'[j^* - 1 : j^*]$ blocks the outcome (M', p'). Assume that $1 < j^* \leq |M'|$. Then at least one of the following conditions hold: (1) $u \leq u^* < U'[j^*] = U[j^*]$; (2) $u \geq u^* > U'[j^* - 1] = U[j^* - 1]$. We argue about condition (1); the argument about condition (2) is symmetric. We start with some useful observations. Since $U'[j^* - 1] \leq u^* < U[j^*]$, we deduce the following two inequalities by an argument similar to the one that is used to derive (4):

$$w(U'[j^* - 1], V[j^*]) - w(U'[j^* - 1], V[j^* - 1]) \leq w(u^*, V[j^*]) - w(u^*, V[j^* - 1]); \quad (9)$$

$$w(u^*, V[j^*]) - w(u^*, V[j^* - 1]) \leq w(U[j^*], V[j^*]) - w(U[j^*], V[j^* - 1]). \quad (10)$$

Stability of (M, p) implies the following two inequalities:

$$w(U[j^* - 1], V[j^*]) - w(U[j^* - 1], V[j^* - 1]) \leq p[j^*] - p[j^* - 1]; \quad (11)$$

$$p[j^*] - p[j^* - 1] \leq w(U[j^*], V[j^*]) - w(U[j^*], V[j^* - 1]). \quad (12)$$

It is easy to see that $U'[j^* - 1] \leq U[j^* - 1]$ because either $M = M'$ or $U[j^* - 1] = u^*$; hence

$$w(U'[j^* - 1], V[j^*]) - w(U'[j^* - 1], V[j^* - 1]) \leq w(U[j^* - 1], V[j^*]) - w(U[j^* - 1], V[j^* - 1]), \quad (13)$$

by an argument similar to the one that is used to derive (9).

With these observations in mind, we want to show the stability of the right-adjacent pair in $M'[j^* - 1 : j^*]$, i.e.,

$$w(U'[j^* - 1], V[j^*]) - w(U'[j^* - 1], V[j^* - 1]) \leq p'[j^*] - p'[j^* - 1], \quad (14)$$

and the stability of the left-adjacent pair in $M'[j^* - 1 : j^*]$, i.e.,

$$p'[j^*] - p'[j^* - 1] \leq w(U[j^*], V[j^*]) - w(U[j^*], V[j^* - 1]), \quad (15)$$

where $p'[j^* - 1] = \max(p[j^* - 1], w(u^*, V[j^* - 1]))$ and $p'[j^*] = \max(p[j^*], w(u^*, V[j^*]))$. We consider three cases.

Case 1: $p'[j^* - 1] = w(u^*, V[j^* - 1])$ and $p'[j^*] = w(u^*, V[j^*])$. Then, (9) implies (14) and (10) implies (15).

Case 2: $p'[j^* - 1] = p[j^* - 1] \geq w(u^*, V[j^* - 1])$ and $p'[j^*] = w(u^*, V[j^*]) > p[j^*]$. Then $p[j^*] - p[j^* - 1] < p'[j^*] - p'[j^* - 1] \leq w(u^*, V[j^*]) - w(u^*, V[j^* - 1])$. The first inequality in the preceding sentence, (11), and (13) imply (14); the second inequality in the preceding sentence and (10) imply (15).

Case 3: $p'[j^*-1] = w(u^*, V[j^*-1]) > p[j^*-1]$ and $p'[j^*] = p[j^*] \geq w(u^*, V[j^*])$. Then $w(u^*, V[j^*]) - w(u^*, V[j^*-1]) \leq p'[j^*] - p'[j^*-1] < p[j^*] - p[j^*-1]$. The first inequality in the preceding sentence and (9) imply (14); the second inequality in the preceding sentence and (12) imply (15). □

Proof of Lemma 19. It is easy to see that claim (1) holds by the definition of $grow(M, p, u)$.

No item blocks the outcome (M', p') since the stability of p and claim (1) together imply that no price in p' is negative. No bid blocks the outcome (M', p') by Lemma 20. In order to prove claim (2), it remains to show that no bid-item pair in A blocks the outcome (M', p'). Observe that Lemmas 16 and 21 directly imply that no bid-item pair involving a bid in U' blocks (M', p'). Now, if $j^* > 1$ (resp., $j^* \leq |V|$), then it is easy to see that $(u^*, V[j^*-1])$ (resp., $(u^*, V[j^*])$) does not block (M', p') since $p'[j^*-1] \geq w(u^*, V[j^*-1])$ (resp., $p'[j^*] \geq w(u^*, V[j^*])$); thus, Lemmas 16 and 21 imply that no bid-item pair involving u^* blocks (M', p').

We now prove claim (3). Assume that there exists a stable price vector of A, denoted p'' in what follows, such that $p'' \geq p$ and $p''[j] < p'[j]$ for at least one item index j. We show a contradiction if $p''[j] < p'[j]$ for some $j \geq j^*$; the argument for the case where $j < j^*$ is symmetric. Assume that $p''[j] < p'[j]$ for some $j \geq j^*$ and let j' denote the minimum such j. Since $p'[j'] > p''[j'] \geq p[j']$, we conclude that $j^\dagger_R \geq j'$. We consider two cases.

Case 1: $j' = j^*$. Then $p''[j^*] < p'[j^*] = w(u^*, V[j^*])$, and thus the bid-item pair $(u^*, V[j^*])$ blocks the outcome (M', p'') since u^* is not matched by M', contradicting the stability of p''.

Case 2: $j' > j^*$. Then $p''[j'] < p'[j'] = p'[j'-1] - w(U'[j'-1], V[j'-1]) + w(U'[j'-1], V[j'])$, where the equality holds by (7). Then, since the definition of j' implies $p'[j'-1] \leq p''[j'-1]$, we conclude that $p''[j'] < p''[j'-1] - w(U'[j'-1], V[j'-1]) + w(U'[j'-1], V[j'])$, and thus that the bid-item pair $(U'[j'-1], V[j])$ blocks the outcome (M', p''), contradicting the stability of p''. □

6.3.2 Computing Prices after Bid Insertion

In this section, we show how to compute $insert(M, p, u)$ in linear time. In what follows, let (M, p) be an arbitrary outcome that is stable for the UDALEW $(bids(M), items(M))$, let u be an arbitrary bid that does not belong to $bids(M)$, let M' denote $insert(M, u)$, let U denote $bids(M)$, let V denote $items(M)$, let U' denote $bids(M')$, let u^* denote $(U + u) \setminus U'$, let j^* denote $index(u^*, U + u)$, let $j^\dagger_L$ denote $reach_L(M', p, u^*)$, let $j^\dagger_R$ denote $reach_R(M', p, u^*)$, and let $p^\dagger$ denote $grow(M, p, u)$.

Algorithm 3 first computes M' and identifies the bid u^* that is not matched by M', using Alg. 1 of Sect. 4. Then, Alg. 3 computes $j^\dagger_L$ (resp., $j^\dagger_R$) in lines 6 through 15 (resp., 17 through 26), which we refer to as the *left-scan* (resp., *right-scan*) in what follows, by initializing the program variable j_L to j^* (resp., j_R to j^*-1), and then decrementing j_L (resp., incrementing j_R) until j_L is equal to $j^\dagger_L$ (resp., j_R is equal to $j^\dagger_L$). The prices $p^\dagger[j^\dagger_L : j^\dagger_R]$ are also computed on the fly during the left-scan (resp., right-scan) and stored in the array $p'[j^\dagger_L : j^\dagger_R]$. We begin our discussion by introducing two useful definitions.

We define the state predicate P_L to hold if $j^\dagger_L \leq j_L$ and $p'[j_L : j^*-1] = p^\dagger[j_L : j^*-1]$. Symmetrically, we define the state predicate P_R to hold if $j^\dagger_R \geq j_R$ and $p'[j^* : j_R] = p^\dagger[j^* : j_R]$.

Algorithm 3 A linear-time implementation of $insert(M, p, u)$.

```
Input: $(M, p)$ is an outcome that is stable for the UDALEW $(bids(M), items(M))$, and $u$ is a bid
    that does not belong to $bids(M)$.
Output: $insert(M, p, u)$.
 1: Let $U$ denote $bids(M)$ and let $V$ denote $items(M)$
 2: $M' \leftarrow insert(M, u)$
 3: $u^* \leftarrow (U + u) \setminus bids(M')$
 4: $j^* \leftarrow index(u^*, U + u)$
 5: $U' \leftarrow bids(M')$
 6: $j \leftarrow j_L \leftarrow j^*$
 7: while $j > 1$ do
 8:     $j \leftarrow j - 1$
 9:     $t \leftarrow$ if $j = j^* - 1$ then $w(u^*, V[j])$ else $p'[j+1] - w(U'[j+1], V[j+1]) + w(U'[j+1], V[j])$
10:     if $p[j] \leq t$ then
11:         $p'[j] \leftarrow t$
12:         $j_L \leftarrow j$
13:     else break
14:     end if
15: end while
16: $p'[\,:j_L - 1] = p[\,:j_L - 1]$
17: $j \leftarrow j_R \leftarrow j^* - 1$
18: while $j < |V|$ do
19:     $j \leftarrow j + 1$
20:     $t \leftarrow$ if $j = j^*$ then $w(u^*, V[j])$ else $p'[j-1] - w(U'[j-1], V[j-1]) + w(U'[j-1], V[j])$
21:     if $p[j] \leq t$ then
22:         $p'[j] \leftarrow t$
23:         $j_R \leftarrow j$
24:     else break
25:     end if
26: end while
27: $p'[j_R + 1 :\,] = p[j_R + 1 :\,]$
28: return $(M', p')$
```

Lemma 22. The following claims hold: (1) the predicate P_L is an invariant of the **while** loop in the left-scan; (2) upon completion of the left-scan, $j_L = j_L^\dagger$.

Proof. The predicate P_L trivially holds upon execution of line 6, and thus at the beginning of the first iteration of the **while** loop, by the definition of $j_L^\dagger$. We prove by induction on the number of iterations that P_L holds at the end of each iteration. Consider an arbitrary iteration of the loop and assume that P_L holds at the beginning of the iteration. After executing line 8, let p'' denote $tight_L(M'[j : j^* - 1], w(u^*, V[j^* - 1]))$. If this is the first iteration of the loop, it is trivial to see that the variable t computed at line 9 is equal to $p''[1]$. Otherwise, since P_L holds at the beginning of the iteration, $p'[j + 1]$ is equal to $p''[2]$, and thus the variable t computed at line 9 is equal to $p''[1]$ by (5). Then it is easy to see that $j_L^\dagger \leq j$ if and only if the condition $p[j] \leq t$ at the **if** statement at line 10 holds, since $p[j+1 : j^* - 1]$ is less than or equal to $p''[2:]$ by our assumption that P_L holds at the beginning of the iteration. Then it is easy to see that P_L holds after execution of the **if** statement, and thus at the end of the iteration. It is also easy to see that $j_L = j_L^\dagger$ upon the termination of the loop: if the loop terminates via the **break** statement, then the inequality $j_L^\dagger > j = j_L - 1$ and P_L implies $j_L = j_L^\dagger$; if the loop terminates because $j = 1$ at line 7, then P_L implies $j_L = j_L^\dagger = j = 1$ since $j_L^\dagger \geq 1$ by definition. □

Lemma 23 below is symmetric to Lemma 22, and so its proof is omitted.

Lemma 23. The following claims hold: (1) the predicate P_R is an invariant of the **while** loop in the right-scan; (2) upon completion of the right-scan, $j_R = j_R^\dagger$.

Finally, the unchanged prices, i.e., $p^\dagger[: j_L^\dagger - 1]$ (resp., $p^\dagger[j_R^\dagger + 1 :]$), are copied into $p'[: j_L^\dagger - 1]$ (resp., $p'[j_R^\dagger + 1 :]$) after the left-scan (resp., right-scan) at line 16 (resp., line 27).

Lemma 24. Algorithm 3 is correct.

Proof. By Lemmas 22 and 23, and lines 16 and 27, the array p' is equal to $p^\dagger$ upon termination of the algorithm. The correctness follows by Theorem 3. □

6.4 Superblock-Based Price Computation

We obtain a fast emulation of Alg. 3 by employing a superblock-based outcome representation and by extending the faster superblock-based bid insertion algorithm of Sect. 5.

For any nonempty ordered matching M, we say that π is a *price-block* for M if π is an array of real values of size $|M|$, or π is a pair (D, q) where $D \in \{L, R\}$ and q is a real value.

Let M be a nonempty ordered matching and let π be a price-block for M. Let U denote $\mathit{bids}(M)$ and let V denote $\mathit{items}(M)$. If π is an array, then we say that π explicitly represents prices, or π is an explicit price-block, and it is understood that $\pi[j]$ is the price of $V[j]$ for $1 \leq j \leq |V|$. If π is a pair (L, q) (resp., (R, q)), then we say that π compactly represents left-tight (resp., right-tight) prices, or π is a compact price-block, and it is understood that q is the price of $V[|V|]$ (resp., $V[1]$), and the prices of other items are defined by (6) (resp., (8)). If the price-block π compactly represents left-tight (resp., right-tight) prices, it is straightforward to see that the following claims hold: for any j such that $1 \leq j \leq |V|$, the price of $V[j]$ can be computed in $O(\min(j, |V| - j))$ time via (5) and (6) (resp., via (7) and (8)) given $\Delta_L(M)$ (resp., $\Delta_R(M)$); π can be converted to an explicit price-block in $O(|V|)$ time.

For any superblock S and any sequence $\Pi = \langle \pi_1, \ldots, \pi_{|S|} \rangle$ of price-blocks, we say that (S, Π) is a *superblock-based outcome* if π_i is a price-block for $\mathit{matching}(S[i], \mathit{shift}(S, i))$ for $1 \leq i \leq |S|$. It is understood that (S, Π) represents the outcome (M, p) where $M = \mathit{matching}(S)$ and the price vector p is represented by the individual price-blocks π_i. Thus, for any superblock-based outcome (S, Π) and any bid u that does not belong to $\mathit{bids}(S)$, we use the notation $\mathit{insert}(S, \Pi, u)$ to denote $\mathit{insert}(M, p, u)$, where (M, p) is the outcome that (S, Π) represents.

In order to obtain a fast implementation of $\mathit{insert}(M, p, u)$, we enhance the data structure of Sect. 5.3.2 so that we maintain a superblock-based outcome. We momentarily describe the necessary modifications in the implementation of the four block-level operations of Sect. 5.3.3, but we first characterize what our fast implementation computes.

Let (S, Π) be a superblock-based outcome and let u be a bid that does not belong to $bids(S)$. Let M' denote $insert(matching(S), u)$ and let u^* denote the bid $(bids(S) + u) \setminus M'$. Then for any superblock S' such that $matching(S') = M'$, $|S| = |S'|$, and $|sum(S, i) - sum(S', i)| \leq 1$ for $0 \leq i \leq |S|$, we define $prices(\Pi, S, u^*, S')$ as the set of all price-block sequences Π' such that the superblock-based outcome (S', Π') represents $insert(S, \Pi, u)$. Note that there may be more than one Π' in $prices(\Pi, S, u^*, S')$, each representing the same price vector, and we are only interested in computing an arbitrary one. We describe how to compute a price-block sequence that belongs to $prices(\Pi, S, u^*, S')$ in Sect. 6.4.2 by emulating Alg. 3, starting with line 4.

6.4.1 Block-Level Operations

We are now ready to describe our efficient implementation of $insert(S, \Pi, u)$, which is based on an enhanced SOM. The input is a superblock-based outcome (S, Π) and a bid u that does not belong to $bids(S)$. We proceed as in Alg. 2 of Sect. 5 and identify u^* at line 29. We enhance the four block-level operations, $refresh$, $split$, $merge$, and $exchange$, so that they operate on superblock-based outcomes, and thus SWAP at line 30 modifies the superblock-based outcome (S, Π).

If the matching does not change, i.e., $u^* = u$ at line 30, we simply set the superblock-based outcome that the data structure maintains to (S', Π') where $S' = S$ and Π' is a price-block sequence that belongs to $prices(\Pi, S, u^*, S')$, and we are done. Otherwise, the enhanced block-level operations are performed during SWAP(S, u^*, u), as described next.

The $refresh$ operation does not change the matching and does not change the partitioning of bids (by the blocks of S), so the price-block sequence Π does not need to be modified.

The $split$ and $merge$ operations do not change the matching, but they potentially change the partitioning of the bids, by either splitting a block or merging two blocks, respectively. If $split$ replaces a block $S[i]$ with two blocks, then we perform the following two steps: if the corresponding price-block $\Pi[i]$ is compact, we convert it to explicit form; we split the array $\Pi[i]$ into two halves. If $merge$ replaces two blocks $S[i]$ and $S[i+1]$ with a single block, then we perform the following two steps: if the corresponding price-block $\Pi[i]$ (resp., $\Pi[i+1]$) is compact, we convert $\Pi[i]$ (resp., $\Pi[i+1]$) to explicit form; we merge the arrays $\Pi[i]$ and $\Pi[i+1]$ into a single array.

The most complicated operation is $exchange$ because it alters the matching. It is also at the end of the enhanced $exchange$ where we emulate Alg. 3 (starting with line 4) to compute the prices, as described in Sect. 6.4.2. The goal is to transform a superblock-based outcome (S, Π) into another superblock-based outcome (S', Π') such that $S' = exchange(S, u^*, u)$ (as described in Sect. 5.3.3) and Π' belongs to $prices(\Pi, S, u^*, S')$. First, the operation $exchange(S, u^*, u)$ is carried out as described in Sect. 5.3.3 to obtain the superblock S', except that the replaced blocks are not discarded until the whole process we describe next is finished, since the blocks corresponding to the old superblock will be useful. Therefore, in what follows, we assume that both S and S' are available. Note that $|sum(S, i) - sum(S', i)| \leq 1$ for $0 \leq i \leq |S|$, as required by the definition of $prices$, where the difference of one is caused by the shifts and by the fact that one block may lose a bid and another block may gain a bid. Second, we compute a price-block sequence Π' that belongs to $prices(\Pi, S, u^*, S')$, and we set the superblock-based outcome that the data structure represents to (S', Π').

In the next section, we describe how to compute a price-block sequence that belongs to $prices(\Pi, S, u^*, S')$ in $O(\sqrt{n})$ time, where n denotes $size(S)$. It is straightforward to see that all the additional computations described above associated with enhanced $refresh$, $split$, $merge$, and $exchange$ operations take $O(\sqrt{n})$ time.

6.4.2 Fast Update of Price-Blocks

In order to complete the description of our efficient implementation of $insert(S, \Pi, u)$, we now show how to compute a price-block sequence that belongs to $prices(\Pi, S, u^*, S')$ in $O(\sqrt{n})$ time. Our approach is to emulate Alg. 3, starting with line 4, on input M, p, and u where (M, p) is the outcome that (S, Π) represents. In what follows, when we refer to the execution of Alg. 3, we mean the execution associated with this input. In the remainder of this section, let M' denote $matching(S')$, which is equal to the ordered matching that the program variable of the same name stores during the execution of Alg. 3, let U' denote $bids(M')$, let V denote $items(M')$, let $j_L^\dagger$ (resp., $j_R^\dagger$) denote $reach_L(M', p, u^*)$ (resp., $reach_R(M', p, u^*)$), which is equal to the value of j_L (resp., j_R) at the end of the execution of Alg. 3, and let $p^\dagger$ denote $grow(M, p, u)$. Before beginning our formal presentation, we present the key idea underlying our fast emulation.

Recall that Alg. 3 computes $j_L^\dagger$ (resp., $j_R^\dagger$) by initializing the program variable j_L to j^* (resp., j_R to $j^* - 1$), and then decrementing j_L (resp., incrementing j_R) by one at each iteration in a loop until j_L is equal to $j_L^\dagger$ (resp., j_R is equal to $j_L^\dagger$). Our fast emulation of Alg. 3 relies on the following lemma, which allows us to "jump over" the blocks, i.e., decrement j_L by one plus the size of a block when we are at a block boundary.

Lemma 25. Let j_1 and j_2 be indices in M such that $j_1 < j_2$ and $j_L^\dagger \leq j_2$ (resp., $j_R^\dagger \geq j_1$). Then, we can decide whether $j_L^\dagger \leq j_1$ (resp., $j_R^\dagger \geq j_2$) holds and we can compute $p^\dagger[j_1]$ (resp., $p^\dagger[j_2]$) in constant time given $\Delta_L(M'[j_1 : j_2])$, $p[j_1]$, and $p^\dagger[j_2]$ (resp., $\Delta_R(M'[j_1 : j_2])$, $p[j_2]$, and $p^\dagger[j_1]$).

Proof. We address the claims regarding $j_L^\dagger$ and the computation of $p^\dagger[j_1]$; the claims regarding $j_R^\dagger$ and the computation of $p^\dagger[j_2]$ are symmetric. Let $\overline{p}$ denote $p^\dagger[j_2] + \Delta_L(M'[j_1 : j_2]) + w(U'[j_1], V[j_1])$, which can be computed in constant time given $p^\dagger[j_2]$ and $\Delta_L(M'[j_1 : j_2])$. Since $j_L^\dagger \leq j_2$, the definition of $p^\dagger$ implies that $p^\dagger[j_2] = w(u^*, V[j^* - 1]) + \Delta_L(M'[j_2 : j^* - 1]) + w(U'[j_2], V[j_2])$, which is equal to $w(u^*, V[j^* - 1]) + \Delta_L(M'[j_2 + 1 : j^* - 1]) + w(U'[j_2 + 1], V[j_2])$ by (L1). Then, $\overline{p}$ is equal to $w(u^*, V[j^* - 1]) + \Delta_L(M'[j_1 : j^* - 1]) + w(U'[j_1], V[j_1])$ by (L1′), and Lemma 18 implies that $j_L^\dagger \leq j_1$ if and only if $p[j_1] \leq \overline{p}$. Finally, the definition of $p^\dagger$ implies that $p^\dagger[j_1]$ is equal to $\overline{p}$ if $j_L^\dagger \leq j_1$, to $p[j_1]$ otherwise. □

Before presenting Alg. 4, our fast implementation of computing a price-block sequence that belongs to $prices(\Pi, S, u^*, S')$, we introduce several useful definitions.

For any superblock-based outcome (S, Π) and any index ℓ such that $1 \leq \ell \leq |S|$, we define $explicit(S, \Pi, \ell)$ as the explicit price-block (an array of real values) that stores the same prices of $items(matching(S[\ell], shift(S, \ell)))$ as in $\Pi[\ell]$. Note that $explicit(S, \Pi, \ell)$ can be computed in $O(sum(S, \ell) - sum(S, \ell - 1))$ time.

For any superblock-based outcome (S, Π), any index ℓ such that $1 \leq \ell \leq |S|$, and any integer $offset$ such that $\max(-1, 1 - \ell) \leq offset \leq 1$, we define $boundary_L(S, \Pi, \ell, offset)$ as follows: if $offset \geq 0$, then $boundary_L(S, \Pi, \ell, offset)$ is the price of the item $V[1 + offset]$ represented by $\Pi[\ell]$, where V denotes $items(matching(S[\ell], shift(S, \ell)))$; if $offset = -1$, then $boundary_L(S, \Pi, \ell, offset)$ is the price of the item $V[|V|]$ represented by $\Pi[\ell - 1]$, where V denotes $items(matching(S[\ell - 1], shift(S, \ell - 1)))$. Note that $boundary_L(S, \Pi, \ell, offset)$ is the price of the item $V[\mathit{sum}(S, \ell - 1) + 1 + \mathit{offset}]$ represented by Π, where V denote $\mathit{items}(S)$, and it can be computed in constant time.

Symmetrically, for any superblock-based outcome (S, Π), any index ℓ such that $1 \leq \ell \leq |S|$, and any integer *offset* such that $-1 \leq \mathit{offset} \leq \min(1, |S| - \ell)$, we define $\mathit{boundary}_R(S, \Pi, \ell, \mathit{offset})$ as follows: if $\mathit{offset} \leq 0$, then $\mathit{boundary}_R(S, \Pi, \ell, \mathit{offset})$ is the price of the item $V[|V| + \mathit{offset}]$ represented by $\Pi[\ell]$, where V denotes $\mathit{items}(\mathit{matching}(S[\ell], \mathit{shift}(S, \ell)))$; if $\mathit{offset} = 1$, then $\mathit{boundary}_R(S, \Pi, \ell, \mathit{offset})$ is the price of the item $V[1]$ represented by $\Pi[\ell + 1]$, where V denotes $\mathit{items}(\mathit{matching}(S[\ell+1], \mathit{shift}(S, \ell+1)))$. Note that $\mathit{boundary}_R(S, \Pi, \ell, \mathit{offset})$ is the price of the item $V[\mathit{sum}(S, \ell) + \mathit{offset}]$ represented by Π, where V denote $\mathit{items}(S)$, and it can be computed in constant time.

Our efficient computation of a price-block sequence that belongs to $\mathit{prices}(\Pi, S, u^*, S')$ is shown in Alg. 4. First, the algorithm performs a scan over the blocks of S' to compute a block index ℓ^* at line 6 so that each bid in each block $S[i]$ for $1 \leq i < \ell^*$ (resp., $\ell^* < i \leq |S|$) is less (resp., greater) than u^*. Then it is easy to see that the integer j^* computed at line 7 is equal to $\mathit{index}(u^*, U' + u^*)$, as in Alg. 3.

We compute $j_L^\dagger$ and the price-blocks that represent $p^\dagger$ for the items $V[\,:j^* - 1]$ in three stages. The computation of $j_R^\dagger$ and the price-blocks for the items $V[j^* :\,]$ is symmetric and omitted. We first give some useful definitions that are analogous to those of P_L and P_R introduced in Sect. 6.3.2, and then we describe the three stages of Alg. 4.

We define the state predicate Q_L (resp., Q_R) to hold if $j_L^\dagger \leq j_L$ (resp., $j_R^\dagger \geq j_R$) and for each item index j' such that $j_L \leq j' \leq j^* - 1$ (resp., $j^* \leq j' \leq j_R$), either (1) $p'[j']$ is assigned $p^\dagger[j']$, or (2) $p'[j']$ remains uninitialized and the price of $V[j']$ represented by Π' is equal to $p^\dagger[j']$.

The first stage (lines 13 through 15) is identical to the left-scan of Alg. 3 except that it is confined within the block $S'[\ell^*]$. The goal of the first stage is to fill $p'[\sigma'(\ell^* - 1) + 1 : j^* - 1]$ which, together with $p'[j^* : \sigma'(\ell^*)]$ that is computed in the symmetric stage (i.e., the first stage associated with the computation of $j_R^\dagger$ and the price-blocks for the items $V[j^* :\,]$), constitutes the explicit price-block $\Pi'[\ell^*]$ built at line 41. Before executing the **while** loop at lines 13 through 15 that decrements j_L and fills p', we ensure at lines 10 through 12 that the input prices (prices represented by Π) for the items that are matched in $S'[\ell^*]$ (items in $V[\sigma'(\ell^* - 1) + 1 : \sigma'(\ell^*)]$) are available in the array p. Then, Lemma 26 below implies that $j_L = \max(j_L^\dagger, \sigma'(\ell^* - 1) + 1)$ and $p'[j_L : j^* - 1] = \mathit{tight}_L(M'[j_L : j^* - 1], w(u^*, V[j^* - 1]))$ upon completion of the **while** loop. Hence, if the inequality checked at line 16 holds, we have $j_L = j_L^\dagger$, and we copy the unchanged prices for the remaining items into $p'[\sigma'(\ell^* - 1) + 1 : j_L - 1]$ from $p[\sigma'(\ell^* - 1) + 1 : j_L - 1]$ at line 17, and we are done. Otherwise, we proceed to the next stage to see whether $j_L^\dagger < \sigma'(\ell^* - 1) + 1$.

Lemma 26. The following claims hold: (1) the predicate Q_L is an invariant of the **while** loop at lines 13 through 15; (2) upon termination of the **while** loop at lines 13 through 15, if $j_L > \sigma'(\ell^* - 1) + 1)$ then $j_L = j_L^\dagger$.

Proof. The predicate Q_L trivially holds upon execution of line 7, and thus at the beginning of the first iteration of the **while** loop, by the definition of $j_L^\dagger$. The rest of the proof is identical to that of Lemma 22, with Q_L replacing P_L, except that if the **while** loop terminates because $j = \sigma'(\ell^* - 1) + 1$ at line 13, then Q_L only implies $j_L = j = \sigma'(\ell^* - 1) + 1 \geq j_L^\dagger$, but not necessarily $j_L = j_L^\dagger$. □

The second stage (lines 19 through 31) identifies the blocks that reside in the left-tight interval

Algorithm 4 Fast implementation of computing a price-block sequence that belongs to $\mathit{prices}(\Pi, S, u^*, S')$.

```
Input: $\Pi$, $S$, $u^*$, and $S'$ such that $\mathit{prices}(\Pi, S, u^*, S')$ is well-defined.
Output: $\Pi'$ such that the outcome $(S', \Pi')$ represents $\mathit{insert}(S, \Pi, u)$, where $u$ denotes $\mathit{bids}(S') \setminus \mathit{bids}(S)$.
 1: Let $U'$ denote $\mathit{bids}(S')$, let $V$ denote $\mathit{items}(S')$, and let $M'$ denote $\mathit{matching}(S')$
 2: Let $S'[i]$ be $(U'_i, V'_i)$ for $1 \le i \le |S'|$
 3: Let $p$ and $p'$ be two uninitialized arrays of $|M'|$ real values
 4: $\sigma(i) \leftarrow \mathit{sum}(S, i)$ for $0 \le i \le |S|$
 5: $\sigma'(i) \leftarrow \mathit{sum}(S', i)$ for $0 \le i \le |S'|$
 6: $\ell^* \leftarrow \max(1, |\{(U', V') \mid (U', V') \in S' \text{ and } U'[1] < u^*\}|)$
 7: $j \leftarrow j_L \leftarrow j^* \leftarrow \mathit{index}(u^*, U'_{\ell^*} + u^*) + \sigma'(\ell^* - 1)$
 8: if $j^* > 1$ then
 9:     $\ell \leftarrow \ell^*$
10:     $p[\sigma(\ell - 1) + 1 : \sigma(\ell)] \leftarrow \mathit{explicit}(S, \Pi, \ell)$
11:     $p[\sigma'(\ell - 1) + 1] \leftarrow \mathit{boundary}_L(S, \Pi, \ell, \sigma'(\ell - 1) - \sigma(\ell - 1))$
12:     $p[\sigma'(\ell)] \leftarrow \mathit{boundary}_R(S, \Pi, \ell, \sigma'(\ell) - \sigma(\ell))$
13:     while $j > \sigma'(\ell - 1) + 1$ do
14:         Perform lines 8 through 14 of Alg. 3
15:     end while
16:     if $j_L > \sigma'(\ell - 1) + 1$ then
17:         $p'[\sigma'(\ell - 1) + 1 : j_L - 1] \leftarrow p[\sigma'(\ell - 1) + 1 : j_L - 1]$
18:     else
19:         $\ell \leftarrow \ell - 1$
20:         while $\ell \ge 1$ do
21:             $t \leftarrow p'[j] - w(U'[j], V[j]) + w(U'[j], V[j - 1])$
22:             $j_1 \leftarrow \sigma'(\ell - 1) + 1$
23:             $p[j_1] \leftarrow \mathit{boundary}_L(S, \Pi, \ell, \sigma'(\ell - 1) - \sigma(\ell - 1))$
24:             if $p[j_1] \le t + \Delta_L(M'[j_1 : j - 1]) + w(U'[j_1], V[j_1])$ then
25:                 $p'[j_1] \leftarrow t + \Delta_L(M'[j_1 : j - 1]) + w(U'[j_1], V[j_1])$
26:                 $\Pi'[\ell] \leftarrow (L, t)$
27:                 $j \leftarrow j_L \leftarrow j_1$
28:                 $\ell \leftarrow \ell - 1$
29:             else break
30:             end if
31:         end while
32:         if $\ell \ge 1$ then
33:             Perform lines 10 through 15 above
34:             $p'[\sigma'(\ell - 1) + 1 : j_L - 1] \leftarrow p[\sigma'(\ell - 1) + 1 : j_L - 1]$
35:             $\Pi'[\ell] \leftarrow p'[\sigma'(\ell - 1) + 1 : \sigma'(\ell)]$
36:         end if
37:     end if
38:     $\Pi'[\,:\ell - 1] \leftarrow \Pi[\,:\ell - 1]$
39: end if
40: Compute $j_R$ and the price-blocks for the items $V[j^* :\,]$ that represent the prices in $p^\dagger$, using code
    symmetric to lines 7 through 39
41: $\Pi'[\ell^*] \leftarrow p'[\sigma'(\ell^* - 1) + 1 : \sigma'(\ell^*)]$
42: return $\Pi'$
```

of $(M', p^\dagger)$ by utilizing the constant-time computation of Lemma 25. The prices corresponding to these blocks are represented using compact price-blocks. More precisely, for each block index i such that $1 \leq i < \ell^*$ and $j_L^\dagger \leq \sigma'(i)$, the **while** loop at lines 20 through 31 sets the price-block $\Pi'[i]$ corresponding to block $S'[i]$ to a compact price-block representing the prices in $p^\dagger$. Upon termination of the **while** loop, the program variable ℓ is equal to $\min\{i \mid 1 \leq i \leq \ell^* \text{ and } j_L^\dagger \leq \sigma'(i)\} - 1$. The following lemma establishes useful properties of the second stage.

Lemma 27. The following claims hold: (1) the predicate Q_L is an invariant of the **while** loop at lines 20 through 31; (2) upon termination of the **while** loop at lines 20 through 31, we have $j_L = \sigma'(\ell) + 1$, and if $\ell \geq 1$ then $j_L^\dagger > \sigma'(\ell - 1) + 1$.

Proof. The predicate Q_L holds upon execution of line 19, and thus at the beginning of the first iteration of the **while** loop, by the first claim of Lemma 26. It is easy to see that the second claim holds if the loop never iterates, i.e., if $\ell = 0$ upon execution of line 19. We prove by induction on the number of iterations that Q_L holds at the end of each iteration. Consider an arbitrary iteration of the loop and assume that Q_L holds at the beginning of the iteration. The variable j_1 is set at line 22 to the index of the item that is matched to the leftmost bid in $S'[\ell]$; the following line sets $p[j_1]$ to the price of that item as represented by the input Π. Then, the algorithm determines whether $j_L^\dagger \leq j_1$ at line 24 by utilizing the constant-time computation of Lemma 25, where j plays the role of j_2: it is easy to see that the right-hand side of the condition of the **if** statement at line 24 is equal to $p'[j] + \Delta_L(M'[j_1 : j]) + w(U'[j_1], V[j_1])$ by (L1), and thus the condition holds if and only if $j_L^\dagger \leq j_1$. If $j_L^\dagger \leq j_1$, then Q_L and the definition of $p^\dagger$ imply that the price assigned at line 25 is equal to $p^\dagger[j_1]$, the price stored in the variable t is equal to $p^\dagger[j]$, and $p^\dagger[j_1 : j]$ is equal to $\mathit{tight}_L(M'[j_1 : j], t)$; thus, the predicate Q_L is maintained after setting $\Pi'[\ell]$ to the left-tight compact price-block at line 26 and upon executing line 28. If $j_L^\dagger > j_1$, it is easy to see that Q_L continues to hold and the second claim of the lemma holds. □

The third stage (line 33) is similar to the first stage: it is identical to the left-scan of Alg. 3 except that it is confined within the block $S'[\ell]$. The goal of the third stage is to fill $p'[\sigma'(\ell - 1) + 1 : \sigma'(\ell)]$ which subsequently forms the explicit price-block $\Pi'[\ell]$ at line 35. Before executing the **while** loop that decrements j_L and fills p', we ensure that the input prices (prices represented by Π) for the items that are matched in $S'[\ell]$ (items in $V[\sigma'(\ell - 1) + 1 : \sigma'(\ell)]$) are available in the array p. By the second claim of Lemma 27 and by Lemma 28 below, it is easy to see that the **while** loop terminates with $j_L = j_L^\dagger$, and we have $p'[j_L : j^* - 1] = \mathit{tight}_L(M'[j_L : j^* - 1], w(u^*, V[j^* - 1]))$.

Lemma 28. The following claims hold: (1) the predicate Q_L is an invariant of the **while** loop associated with line 33; (2) upon termination of the **while** loop associated with line 33, $j_L = j_L^\dagger$.

Proof. The predicate Q_L holds right before executing line 33, and thus it holds at the beginning of the first iteration of the **while** loop, by the first claim of Lemma 27. The rest of the proof is identical to that of Lemma 22, with Q_L replacing P_L, except that it is guaranteed by the second claim of Lemma 27 that the **while** loop terminates via the **break** statement, and thus, the inequality $j_L^\dagger > j = j_L - 1$ and Q_L imply that $j_L = j_L^\dagger$ holds upon termination of the loop. □

Corollary 2. Upon execution of line 39, $j_L = j_L^\dagger$.

Lemma 29. Algorithm 4 is correct.

Proof. It is sufficient to prove that the price-block sequence Π' represents the prices in $p^\dagger$. In what follows, let $\ell^\dagger$ denote $\min\{i \mid 1 \leq i \leq \ell^*$ and $j_L^\dagger \leq \sigma'(i)\} - 1$. As discussed earlier, upon termination of the **while** loop at lines 20 through 31, the program variable ℓ is equal to $\ell^\dagger$.

Lemma 26 and line 17 imply that $p'[\sigma'(\ell^* - 1) + 1 : j^* - 1] = p^\dagger[\sigma'(\ell^* - 1) + 1 : j^* - 1]$. The symmetric results and code associated with the computation of $j_R^\dagger$ and the price-blocks for the items $V[j^* :]$ imply that $p'[j^* : \sigma'(\ell^*)] = p^\dagger[j^* : \sigma'(\ell^*)]$. Then line 41 implies that $\Pi'[\ell^*]$ represents $p^\dagger[\sigma'(\ell^* - 1) + 1 : \sigma'(\ell^*)]$.

If $\ell^\dagger < \ell^* - 1$, then Lemma 27 and line 26 imply that $\Pi'[\ell^\dagger + 1 : \ell^* - 1]$ represents $p^\dagger[\sigma'(\ell^\dagger) + 1 : \sigma'(\ell^* - 1)]$.

If $\ell^\dagger \geq 1$, then Lemma 28 and lines 34 and 35 imply that $\Pi'[\ell^\dagger]$ represents $p^\dagger[\sigma'(\ell^\dagger - 1) + 1 : \sigma'(\ell^\dagger)]$.

If $\ell^\dagger > 1$, then Corollary 2 and line 38 imply that $\Pi'[: \ell^\dagger - 1]$ represents $p^\dagger[: \sigma'(\ell^\dagger - 1)]$, as those prices are unaffected by bid insertion.

If $\ell^* < |S'|$, the symmetric results and code associated with the computation of $j_R^\dagger$ and the price-blocks for the items $V[j^* :]$ imply that $\Pi'[\ell^* + 1 :]$ represents $p^\dagger[\sigma'(\ell^*) + 1 :]$. □

It is easy to see that the three stages associated with the computation of $j_L^\dagger$ and the price-blocks that represent $p^\dagger$ for the items $V[: j^* - 1]$ run in $O(\sqrt{n})$ time where n denotes $size(S)$: for the first and third stages, the number of iterations of each loop is at most the number of bids in the associated block, which is $O(\sqrt{n})$; for the second stage, the number of iterations of the loop is at most the number of blocks in S', which is $O(\sqrt{n})$; each iteration of each loop takes constant time; all the remaining operations can be performed in $O(\sqrt{n})$ time. By a symmetric argument, the three stages associated with the computation of $j_R^\dagger$ and the price-blocks for the items $V[j^* :]$ also run in $O(\sqrt{n})$ time.

References

[1] G. Demange, D. Gale, and M. A. O. Sotomayor. Multi-item auctions. *The Journal of Political Economy*, 94:863–872, 1986.

[2] N. O. Domaniç and C. G. Plaxton. Scheduling unit jobs with a common deadline to minimize the sum of weighted completion times and rejection penalties. In *Proceedings of the 25th International Symposium on Algorithms and Computation*, pages 646–657, 2014.

[3] R. Duan and H.-H. Su. A scaling algorithm for maximum weight matching in bipartite graphs. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1413–1424, 2012.

[4] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[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] F. Glover. Maximum matching in a convex bipartite graph. *Naval Research Logistics Quarterly*, 14:313–316, 1967.

[7] J. Green and J.-J. Laffont. Characterization of satisfactory mechanisms for the revelation of preferences for public goods. *Econometrica*, 45:427–438, 1977.

[8] G. H. Hardy, J. E. Littlewood, and G. Pólya. *Inequalities*. Cambridge University Press, 2nd edition, 1952.

[9] I. Katriel. Matchings in node-weighted convex bipartite graphs. *INFORMS Journal on Computing*, 20:205–211, 2008.

[10] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[11] H. B. Leonard. Elicitation of honest preferences for the assignment of individuals to positions. *The Journal of Political Economy*, 91:461–479, 1983.

[12] 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.

[13] C. G. Plaxton. Vertex-weighted matching in two-directional orthogonal ray graphs. In *Proceedings of the 24th International Symposium on Algorithms and Computation*, pages 524–534, 2013.

[14] L. R. Rabiner and B. Gold. *Theory and application of digital signal processing*. Englewood Cliffs, NJ, Prentice-Hall, Inc., 1975.

[15] A. E. Roth and M. A. O. Sotomayor. *Two-sided matching: A study in game-theoretic modeling and analysis*. Cambridge University Press, 1992.

[16] L. S. Shapley and M. Shubik. The assignment game I: The core. *International Journal of Game Theory*, 1:111–130, 1971.

[17] G. Steiner and J. S. Yeomans. A linear time algorithm for maximum matchings in convex, bipartite graphs. *Computers and Mathematics with Applications*, 31:91–96, 1996.