Offline sorting buffers on Line

Rohit Khandekar[1] and Vinayaka Pandit[2]

[1] University of Waterloo, ON, Canada. email: rkhandekar@gmail.com
[2] IBM India Research Lab, New Delhi. email: pvinayak@in.ibm.com

Abstract. We consider the *offline sorting buffers* problem. Input to this problem is a sequence of requests, each specified by a point in a metric space. There is a "server" that moves from point to point to serve these requests. To serve a request, the server needs to visit the point corresponding to that request. The objective is to minimize the total distance travelled by the server in the metric space. In order to achieve this, the server is allowed to serve the requests in any order that requires to "buffer" at most k requests at any time. Thus a valid reordering can serve a request only after serving all but k previous requests.
In this paper, we consider this problem on a line metric which is motivated by its application to a widely studied disc scheduling problem. On a line metric with N uniformly spaced points, our algorithm yields the first *constant-factor approximation* and runs in quasi-polynomial time $O(m \cdot N \cdot k^{O(\log N)})$ where m is the total number of requests. Our approach is based on a dynamic program that keeps track of the number of pending requests in each of $O(\log N)$ line segments that are geometrically increasing in length.

1 Introduction

The sorting buffers problem arises in scenarios where a stream of requests needs to be served. Each request has a "type" and for any pair of types t_1 and t_2, the cost of serving a request of type t_2 immediately after serving a request of type t_1 is known. The input stream can be reordered while serving in order to minimize the cost of type-changes between successive requests served. However, a "sorting buffer" has to be used to store the requests that have arrived but not yet served and often in practice, the size of such a sorting buffer, denoted by k, is small. Thus a legal reordering must satisfy the following property: any request can be served only after serving all but k of the previous requests. The objective in the sorting buffers problem is to compute the minimum cost output sequence which respects this sequencing constraint.

Consider, as an example, a workshop dedicated to coloring cars. A sequence of requests to color cars with specific colors is received. If the painting schedule paints a car with a certain color followed by a car with a different color, then, a significant set-up cost is incurred in changing colors. Assume that the workshop has space to hold at most k cars in waiting. A natural objective is to rearrange the sequence of requests such that it can be served with a buffer of size k and the total set-up cost over all the requests is minimized.

Consider, as another example, the classical disc scheduling problem. A sequence of requests each of which is a block of data to be written on a particular track is given. To write a block on a track, the disc-head has to be moved to that track. As discussed in [3], the set of tracks can be modeled by uniformly spaced points on a straight line. The cost of moving from a track to another is then the distance between those tracks on the straight line. We are given a buffer that can hold at most k blocks at a time, and the goal is to find a write-sequence subject to the buffer constraint such that the total head movement is minimized.

Usually, the type-change costs satisfy metric properties and hence we formulate the sorting buffers problem on a metric space. Let (V, d) be a metric space on N points. The input to the Sorting Buffers Problem (SBP) consists of a sequence of m requests, the ith request being labeled with a point $p_i \in V$. There is a server, initially located at a point $p_0 \in V$. To serve ith request, the server has to visit p_i. There is a sorting buffer which can hold up to k requests at a time. In a *legal schedule*, the ith request can be served only after serving at least $i-k$ requests of the first $i-1$ requests. More formally, the output is given by a permutation π of $\{1, \ldots, m\}$ where the ith request in the output sequence is the $\pi(i)$th request in the input sequence. Observe that a schedule π is legal if and only if it satisfies $\pi(i) \leq i+k$ for all i. The cost of the schedule is the total distance that the server has to travel, i.e., $C_\pi = \sum_{i=1}^{m} d(p_{\pi(i-1)}, p_{\pi(i)})$ where $\pi(0) = p_0$ corresponds to the starting point. The goal in SBP is to find a legal schedule π that minimizes C_π. In the online version of SBP, the ith request is revealed only after serving at least $i-k$ among the first $i-1$ requests. In the offline version, on the other hand, the entire input sequence is known in advance.

The car coloring problem described above can be thought of as the SBP on a uniform metric where all the pair-wise distances are identical while the disc scheduling problem corresponds to the SBP on a line metric where all the points lie on a straight line and the distances are given along that line.

1.1 Previous Work

On a general metric, the SBP is known to be NP-hard due to a simple reduction from the Hamiltonian Path problem. However, for the uniform or line metrics, it is not known if the problem remains NP-hard. In fact, no non-trivial lower bound is known on the approximation (resp. competitive) ratio of offline (resp. online) algorithms, deterministic or randomized. In [3], it is shown that the popular heuristics like shortest time first, first-in-first-out (FIFO) have $\Omega(k)$ competitive ratio on a line metric. In [5], it is shown that the popular heuristics like FIFO, LRU, and Most-Common-First (MCF) have a competitive ratio of $\Omega(\sqrt{k})$ on a uniform metric.

The offline version of the sorting buffers problem on any metric can be solved optimally using dynamic programming in $O(m^{k+1})$ time where m is the number of requests in the sequence. This follows from the observation that the algorithm can pick k requests to hold in the buffer from first i requests in $\binom{i}{k}$ ways when the $(i+1)$th request arrives.

The SBP on a uniform metric has been studied before. Räcke et al. [5] presented a deterministic online algorithm, called *Bounded Waste* that has $O(\log^2 k)$ competitive ratio. Englert and Westermann [2] considered a generalization of the uniform metric in which moving to a point p from any other point in the space has a cost c_p. They proposed an algorithm called Maximum Adjusted Penalty (MAP) and showed that it gives an $O(\log k)$ approximation, thus improving the competitive ratio of the SBP on uniform metric. Kohrt and Pruhs [4] also considered the uniform metric but with different optimization measure. Their objective was to maximize the reduction in the cost from that of the schedule without a buffer. They presented a 20-approximation algorithm for this variant and this ratio was improved to 9 by Bar-Yehuda and Laserson [1].

For SBP on line metric, Khandekar and Pandit [3] gave a polynomial time randomized online algorithm with $O(\log^2 N)$ competitive ratio. In fact, their approach works on a class of "line-like" metrics. Their approach is based on probabilistic embedding of the line metric into the so-called hierarchical well-separated trees (HSTs) and an $O(\log N)$-competitive algorithm for the SBP on a binary tree metric. No better approximations were known for the offline problem.

1.2 Our results

The first step in understanding the structure of the SBP is to develop offline algorithms with better performance than the known online algorithms. We provide such an algorithm. Following is our main theorem.

Theorem 1. *There is a constant factor approximation algorithm for the offline SBP on a line metric on N uniformly spaced points that runs in quasi-polynomial time: $O(m \cdot N \cdot k^{O(\log N)})$ where k is the buffer-size and m is the number of input requests.*

This is the first constant factor approximation algorithm for this problem on any non-trivial metric space. The approximation factor we prove here is 15. However we remark that this factor is not optimal and most likely can be improved even using our techniques. Our algorithm is based on dynamic programming. We show that there is a near-optimum schedule with some "nice" properties and give a dynamic program to compute the best schedule with those nice properties. In Section 2.1, we give an intuitive explanation of our techniques and the Sections 2.2 and 2.3 present the details of our algorithm.

2 Algorithm

2.1 Outline of our approach

We start by describing an exact algorithm for the offline SBP on a general metric on N points. As we will be interested in a line metric as in the disc scheduling problem, we use the term "head" for the server and "tracks" for the points. Since the first k requests can be buffered without loss of generality, we fetch and store them in the buffer. At a given step in the algorithm, we define a *configuration* (t, C) to be the pair of current head location t and an N-dimensional vector C that specifies the number of requests pending at each track. Since there are N choices for t and a total of k requests pending, the number of distinct configurations is $O(N \cdot k^N)$. We construct a dynamic program that keeps track of the current configuration and computes the optimal solution in time $O(m \cdot N \cdot k^N)$ where m is the total number of requests. The dynamic program proceeds in m levels. For each level i and each configuration (t, C), we compute the least cost of serving i requests from the first $i + k$ requests and ending up in the configuration (t, C). Let us denote this cost by $\mathtt{DP}[i, t, C]$. This cost can be computed using the relation

$$\mathtt{DP}[i, t, C] = \min_{(t', C')} (\mathtt{DP}[i - 1, t', C'] + d(t', t))$$

where the minimum is taken over all configurations (t', C') such that while moving the head from t' to t, a request at either t' or t in C' can be served and a new request can be fetched to arrive at the configuration (t, C). Note that it is easy to make suitable modifications to keep track of the order of the output sequence.

Note that the high complexity of the above dynamic program is due to the fact that we keep track of the number of pending requests at *each* of the N tracks. We now describe our intuition behind obtaining much smaller dynamic program for a line metric on N uniformly spaced points. Our dynamic program keeps track of the number of pending requests only in $O(\log N)$ segments of the line which are geometrically increasing in lengths. The key observation is as follows: if the optimum algorithm moves the head from a track t to t' (thereby paying the cost $|t - t'|$), a constant factor approximation algorithm can safely move an additional $O(|t - t'|)$ distance and clear all the nearby requests surrounding t and t'. We show that instead of keeping track of the number of pending requests at each track, it is enough to do so for the ranges of length $2^0, 2^1, 2^2, 2^3, \ldots$ surrounding the current head location t. For each track t, we partition the disc into $O(\log N)$ ranges of geometrically increasing lengths on both sides of t. The configuration (t, C) now refers to the current head location t and an $O(\log N)$-dimensional vector C that specifies number of requests pending in each of these $O(\log N)$ ranges. Thus the new dynamic program will have size $O(m \cdot N \cdot k^{O(\log N)})$.

To be able to implement the dynamic program, we ensure the property that the new configuration around t' should be easily computable from the previous configuration around t. More precisely, we ensure that the partitions for t and t' satisfy the following property: outside an interval of length $R = O(|t - t'|)$ containing t and t', the ranges in the partition for t coincide with those in the partition for t' (see Figure 1). Note however that inside this interval, the two partitions may not agree. Thus when the optimum algorithm moves the head from t to t', our algorithm starts the head from t, clears all the pending requests in this interval and rests the head at t' and updates the configuration from the

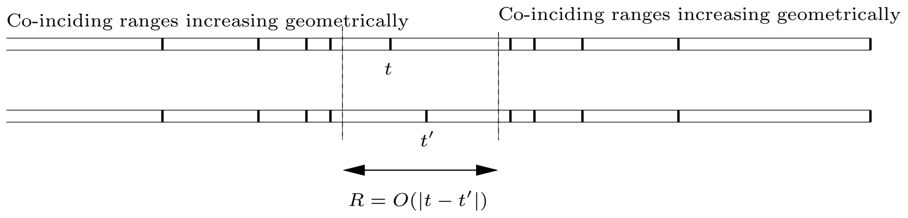
Co-inciding ranges increasing geometrically
Co-inciding ranges increasing geometrically
t
t'
$R = O(|t - t'|)$

Fig. 1. Division of the line into ranges for tracks t and t'

previous configuration. Since the length of the interval is $O(|t-t'|)$, our algorithm spends at most a constant factor more than the optimum.

2.2 Partitioning Scheme

Now we define a partitioning scheme and its properties that are used in our algorithm. Let us assume, without loss of generality, that the total number of tracks $N = 2^n$ is a power of two and that the tracks are numbered from 0 to $2^n - 1$ left-to-right. In the following, we do not distinguish between a track and its number. For tracks t and t', the quantity $|t-t'|$ denotes the distance between these tracks which is the cost paid in moving the head from t to t'. We say that a track t is to the right (resp. left) of a track t' if $t > t'$ (resp. $t < t'$).

Definition 1 (landmarks). *For a track t and an integer $p \in [1, n]$, we define pth landmark of t as $\ell_p(t) = (q+1)2^p$ where q is the unique integer such that $(q-1)2^p \le t < q2^p$. We also define $(-p)$th landmark as $\ell_{-p}(t) = (q-2)2^p$. We also define $\ell_0(t) = t$.*

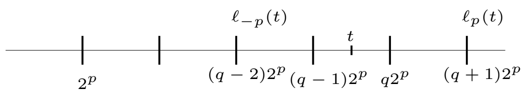

Fig. 2. The pth and $(-p)$th landmarks of a track t

It is easy to see that $\ell_{-n}(t) < \cdots < \ell_{-1}(t) < \ell_0(t) < \ell_1(t) < \cdots < \ell_n(t)$. In fact the following lemma claims something stronger and follows easily from the above definition.

Lemma 1. *Let $p \in [1, n-1]$ and $(q-1)2^p \le t < q2^p$ for an integer q.*

- *If q is even, then $\ell_{p+1}(t) - \ell_p(t) = 2^p$ and $\ell_{-p}(t) - \ell_{-p-1}(t) = 2^{p+1}$.*
- *If q is odd, then $\ell_{p+1}(t) - \ell_p(t) = 2^{p+1}$ and $\ell_{-p}(t) - \ell_{-p-1}(t) = 2^p$.*

In the following definition, we use the notation $[a, b) = \{t \text{ integer} \mid a \leq t < b\}$.

Definition 2 (ranges). *For a track t, we define a "range" to be a contiguous subset of tracks as follows.*

- *$[\ell_{-1}(t), \ell_0(t) = t)$ and $[\ell_0(t) = t, \ell_1(t))$ are ranges.*
- *for $p \in [1, n-1]$,* **if** *$\ell_{p+1}(t) - \ell_p(t) = 2^{p+1}$ and $\ell_p(t) - \ell_{p-1}(t) = 2^{p-1}$* **then** *$[\ell_p(t), \ell_p(t) + 2^p)$ and $[\ell_p(t) + 2^p, \ell_{p+1}(t))$ are ranges,* **else** *$[\ell_p(t), \ell_{p+1}(t))$ is a range.*
- *for $p \in [1, n-1]$,* **if** *$\ell_{-p}(t) - \ell_{-p-1}(t) = 2^{p+1}$ and $\ell_{-p+1}(t) - \ell_{-p}(t) = 2^{p-1}$* **then** *$[\ell_{-p-1}(t), \ell_{-p-1}(t) + 2^p)$ and $[\ell_{-p-1}(t) + 2^p, \ell_{-p}(t))$ are ranges,* **else** *$[\ell_{-p-1}(t), \ell_{-p}(t))$ is a range.*

The above ranges are disjoint and form a partition of the tracks which we denote by $\pi(t)$.

Note that in the above definition, when the difference $\ell_{p+1}(t) - \ell_p(t)$ and $\ell_{-p}(t) - \ell_{-p-1}(t)$ equals 4 times $\ell_p(t) - \ell_{p-1}(t)$ and $\ell_{-p+1}(t) - \ell_{-p}(t)$ respectively, we divide the intervals $[\ell_p(t), \ell_{p+1}(t))$ and $[\ell_{-p-1}(t), \ell_{-p}(t))$ into two ranges of length 2^p each. For example, in Figure 3, the region between $\ell_{p+2}(t)$ and $\ell_{p+3}(t)$ is divided into two disjoint ranges of equal size.

The following lemma proves a useful relation between the partitions $\pi(t)$ and $\pi(t')$ for a pair of tracks t and t': the ranges in the two partitions coincide outside the interval of length $R = O(|t - t'|)$ around t and t'. As explained in Section 2.1, such a property is important for carrying the information about the current configuration across the head movement from t to t'.

Lemma 2. *Let t and t' be two tracks such that $2^{p-1} \leq t' - t < 2^p$. The ranges in $\pi(t)$ and $\pi(t')$ are identical outside the interval $R = [\ell_{-p}(t), \ell_p(t'))$.*

Proof. First consider the case when $(q-1)2^p \leq t < t' < q2^p$ for an integer q, i.e., t and t' lie in the same "aligned" interval of length 2^p. Then clearly they also lie in the same aligned interval of length 2^r for any $r \geq p$. Thus, by definition, $\ell_r(t) = \ell_r(t')$ for $r \geq p$ and $r \leq -p$. Thus it is easy to see from the definition of ranges that the ranges in $\pi(t)$ and $\pi(t')$ outside the interval $[\ell_{-p}(t), \ell_p(t'))$ are identical.

Consider now the case when t and t' do not lie in the same aligned interval of length 2^p. Since $|t - t'| < 2^p$, they must lie in the adjacent aligned intervals of length 2^p, i.e., for some integer q, we have $(q-1)2^p \leq t < q2^p \leq t' < (q+1)2^p$ (See Figure 3). Let $q = 2^u v$ where $u \geq 0$ is an integer and v is an odd integer.

The following key claim states that depending upon how r compares with the the highest power of two that divides the "separator" $q2^p$ of t and t', either the rth landmarks of t and t' coincide with each other or the $(r+1)$th landmark of t coincides with the rth landmark of t'.

Claim. 1. $\ell_r(t) = \ell_r(t')$ for $r \geq p + u + 1$ and $r \leq -p - u - 1$.
2. $\ell_{r+1}(t) = \ell_r(t')$ for $p \leq r < p + u$,
3. $\ell_{-r}(t) = \ell_{-r-1}(t')$ for $p \leq r < p + u$,

4. $\ell_{p+u}(t') = \ell_{p+u}(t) + 2^{p+u}$ and $\ell_{p+u+1}(t) - \ell_{p+u}(t) = 2^{p+u+1}$,
5. $\ell_{-p-u}(t) = \ell_{-p-u-1}(t') + 2^{p+u}$ and $\ell_{-p-u}(t') - \ell_{-p-u-1}(t') = 2^{p+u+1}$,

Proof. The equation 1 follows from the fact that since 2^{p+u} is the highest power of two that divides $q2^p$, both t and t' lie in the same aligned interval of length 2^r for $r \geq p+u+1$.

The equations 2, 3, 4, and 5 follow from the definition of the landmarks and the fact that t and t' lie in the different but adjacent aligned intervals of length 2^r for $p \leq r < p+u$ (see Figure 3).

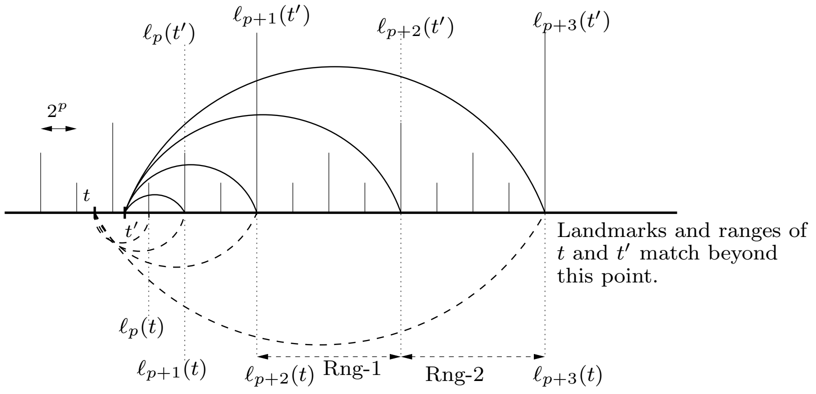

Fig. 3. Landmarks and ranges for tracks t and t' when $q = 4, u = 2$.

Claim 2.2 implies that all but one landmarks of t and t' coincide with each other. For the landmarks of t and t' that coincide with each other, it follows from the definition of the ranges that the corresponding ranges in $\pi(t)$ and $\pi(t')$ are identical.

The landmarks of t, t' that do not coincide are $\ell_{p+u}(t') = \ell_{p+u}(t) + 2^{p+u}$ and $\ell_{-p-u}(t) = \ell_{-p-u-1}(t') + 2^{p+u}$. But, note that the intervals $[\ell_{p+u}(t), \ell_{p+u+1}(t))$ and $[\ell_{-p-u-1}(t'), \ell_{-p-u}(t'))$ are divided into two ranges each: $[\ell_{p+u}(t), \ell_{p+u}(t) + 2^{p+u})$, $[\ell_{p+u}(t) + 2^{p+u}, \ell_{p+u+1}(t))$ and $[\ell_{-p-u-1}(t'), \ell_{-p-u-1}(t') + 2^{p+u})$, $[\ell_{-p-u-1}(t') + 2^{p+u}, \ell_{-p-u}(t'))$. These ranges match with $[\ell_{p+u-1}(t'), \ell_{p+u}(t'))$, $[\ell_{p+u}(t'), \ell_{p+u+1}(t'))$ and $[\ell_{-p-u-1}(t), \ell_{-p-u}(t))$, $[\ell_{-p-u}(t), \ell_{-p-u+1}(t))$ respectively. This follows again from the Claim 2.2 and the carefully chosen definition of ranges. Thus the proof of Lemma 2 is complete.

For tracks t and t', where $t < t'$, let $R(t, t') = R(t', t)$ be the interval $[\ell_{-p}(t), \ell_p(t'))$ if $2^{p-1} \leq t' - t < 2^p$. Note that the length of the interval $R(t, t')$ is at most $|\ell_{-p}(t) - \ell_p(t')| \leq 4 \cdot 2^p \leq 8 \cdot |t - t'|$. Thus the total movement in starting from t, serving all the requests in $R(t, t')$, and ending at t' is at most $15 \cdot |t - t'|$.

2.3 The Dynamic Program

Our dynamic program to get a constant approximation for the offline SBP on a line metric is based on the intuition given in Section 2.1 and uses the partition scheme given in Section 2.2. Recall that according to the intuition, when the optimum makes a move from t to t', we want our algorithm to clear all the requests in $R(t, t')$. This motivates the following definition.

Definition 3. *A feasible schedule for serving all the requests is said to be "locally greedy" if there is a sequence of tracks $t_1, \ldots, t_l$, called "landmarks", which are visited in that order and while moving between any consecutive pair of tracks t_i and t_{i+1}, the schedule also serves all the current pending requests in the interval $R(t_i, t_{i+1})$.*

Since the total distance travelled in a locally greedy schedule corresponding to the optimum schedule is at most 15 times that of the optimum schedule, the best locally greedy schedule is a 15-approximation to the optimum. Our dynamic program computes the best locally greedy schedule. For a locally greedy schedule, let a configuration be defined as a pair (t, C) where t is the location of the head and C is an $O(\log N)$-dimensional vector specifying the number of requests pending in each range in the partition $\pi(t)$. Clearly the number of distinct configurations is $O(N \cdot k^{O(\log N)})$.

The dynamic program is similar to the one given in Section 2.1 and proceeds in m levels. For each level i and each configuration (t, C), we compute the least cost of serving i requests from the first $i + k$ requests and ending up in the configuration (t, C) in a locally greedy schedule. Let $\mathtt{DP}[i, t, C]$ denote this cost. This cost now can be computed as follows. Consider a configuration (t', C') after serving $i - r$ requests for some $r > 0$ such that while moving from a landmark t' to the next landmark t,

1. the locally greedy schedule serves exactly r requests from the interval $R(t', t)$,
2. it travels a distance of D, and
3. after fetching r new requests, it ends up in the configuration (t, C).

In such a case,

$$\mathtt{DP}[i - r, t', C'] + D$$

is an upper bound on $\mathtt{DP}[i, t, C]$. Taking the minimum over all such upper bounds, one obtains the value of $\mathtt{DP}[i, t, C]$.

Recall that the locally greedy schedule clears all the pending requests in the interval $R(t', t)$ while moving from t' and t and also that the ranges in $\pi(t)$ and $\pi(t')$ coincide outside the interval $R(t', t)$. Thus it is feasible to determine if after serving r requests in $R(t', t)$ and fetching r new requests, the schedule ends up in the configuration (t, C).

The dynamic program, at the end, outputs $\min_t \mathtt{DP}[m, t, \mathbf{0}]$ as the minimum cost of serving all the requests by a locally greedy schedule. It is also easy to modify the dynamic program to compute the minimum cost locally greedy schedule along with its cost.

3 Conclusions

Prior to this work, any offline algorithms with better approximation factors than the corresponding online algorithms were not known for the sorting buffers problem on any non-trivial metric. We give the first constant factor approximation for the sorting buffers problem on the line metric improving the previously known $O(\log^2 N)$ competitive ratio. As the running time of our algorithm is quasi-polynomial, we suggest that there may be a polynomial time constant factor approximation algorithm as well. Proving any hardness results for the sorting buffers problem on the uniform or line metrics; or poly-logarithmic approximation results for general metrics remain as interesting open questions.

References

1. R. Bar-Yehuda and J. Laserson. 9-approximation algorithm for the sorting buffers problem. In *3rd Workshop on Approximation and Online Algorithms*, 2005.
2. M. Englert and M. Westermann. Reordering buffer management for non-uniform cost models. In *Proceedings of the 32nd International Colloquium on Algorithms, Langauages, and Programming*, pages 627–638, 2005.
3. R. Khandekar and V. Pandit. Online sorting buffers on line. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 616–625, 2006.
4. J. Kohrt and K. Pruhs. A constant approximation algorithm for sorting buffers. In *LATIN 04*, pages 193–202, 2004.
5. H. Räcke, C. Sohler, and M. Westermann. Online scheduling for sorting buffers. In *Proceedings of the European Symposium on Algorithms*, pages 820–832, 2002.