DeltaSort: Incremental repair of sorted arrays with known updates

Shubham Dwivedi
Independent Researcher
shubd3@gmail.com

January 2026

Abstract

Maintaining sorted order under incremental updates is a common requirement in read-heavy systems, yet most production sorting routines are blind to which elements have changed and must repeatedly rediscover partial order from scratch. This paper explores an alternative model in which the sorting routine is explicitly informed of the indices updated since the previous sort. Under this model, we present *DeltaSort*, an incremental repair algorithm for arrays that batches multiple updates instead of applying them independently. *DeltaSort* reduces redundant comparison work and overlapping data movement by exploiting update locality and batching. An experimental evaluation of a Rust implementation shows that *DeltaSort* consistently outperforms repeated binary-insertion and blind native sorting for update batch sizes up to 25%, with multi-fold speedups and a clear crossover point depending on array size where full re-sorting becomes preferable. These results suggest that tighter integration between update pipelines and sorting routines can yield significant performance gains in real incremental-sorting workloads at the cost of increased memory usage to track updated indices.

1 Introduction

Sorting is among the most heavily optimized primitives in modern systems. Standard library implementations—TimSort [9], Introsort [7], and PDQSort [8]—deliver excellent performance for general inputs by exploiting partial order, cache locality, and adaptive strategies. However, these algorithms operate under a *blind* model: they rediscover structure dynamically rather than being explicitly informed about which elements have changed since the previous sort.

In many practical systems, this assumption is unnecessarily pessimistic. Sorted arrays are often maintained incrementally in read-heavy workloads where updates affect only a subset of elements and the indices of those updates can be easily tracked and known. Nevertheless, this information is typically not tracked or utilized, and systems fall back to blind re-sorting or independent incremental repairs using binary-insertion or extract-sort-merge. To address these limitations, this paper makes the following contributions:

1. **Update-aware sorting model:** We formulate an incremental sorting model in which the sorting routine is explicitly informed of the indices updated since the previous sort. Under this model, a natural baseline is repeated binary insertion, where each update is repaired independently using binary search. While correct, this approach exhibits efficiency limitations for batched updates due to repeated full-range searches and overlapping data movement.

2. **DeltaSort algorithm:** We present *DeltaSort*, an incremental repair algorithm for sorted arrays designed for the update-aware model. DeltaSort batches updates and repairs them

jointly, reducing redundant comparison work and overlapping element movement by exploiting update locality and batching. DeltaSort preserves correctness and does not improve asymptotic complexity, but achieves consistent practical performance improvements over repeated binary insertion and blind native sorting in relevant update regimes.

2 Related Work

Adaptive sorting algorithms exploit existing order in the input to improve performance on nearly sorted data. TimSort [9] and natural merge sort [5] identify monotonic runs dynamically and merge them efficiently, yielding improved performance when such structure is present. A substantial body of work formalizes measures of presortedness and studies sorting algorithms whose complexity depends on these measures rather than input size alone [6]. These approaches, however, must discover structure through full-array scans and do not assume explicit knowledge of which elements were modified.

Dynamic data structures offer a different trade-off. Self-balancing trees such as AVL trees [1], red–black trees [4], B-trees [2], and skip lists [10] support efficient ordered updates with logarithmic cost, but sacrifice contiguous memory layout and cache locality. Library sort [3] reduces insertion cost by maintaining gaps in the array, but addresses online insertion and incurs additional space overhead.

In contrast, this work focuses on maintaining sorted order in arrays under batched updates where the indices of modified elements are explicitly available. This update-aware model enables efficient repair without auxiliary data structures or additional memory, and distinguishes DeltaSort from prior adaptive and incremental approaches.

3 Problem Model

Definition 1 (Update-Aware Sorting)**.** Let $A[0..n-1]$ be an array sorted according to a strict weak ordering defined by a comparator `cmp`. Suppose a set of indices $D = \{d_1, \dots, d_k\} \subseteq \{0, \dots, n-1\}$ is given such that the values at these indices may have been arbitrarily modified, while values at all other indices remain unchanged.

The *update-aware sorting problem* is to restore A to a state that is sorted with respect to `cmp`, given explicit knowledge of the set D.

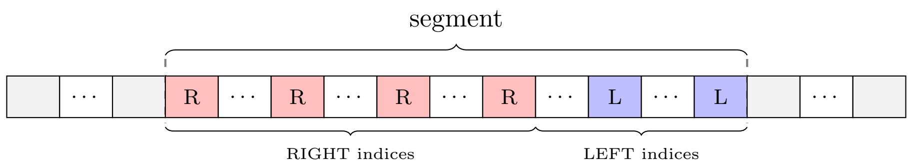
4 DeltaSort Algorithm

4.1 Overview

Definition 2 (Violation)**.** For a dirty index i, we classify its *violation* based on local order:

- **LEFT (L)**: Element **must** move left — $\texttt{cmp}(A[i-1], A[i]) > 0$ (for $i > 0$).
- **RIGHT (R)**: Element **may** move right or stay stable. Essentially, all dirty indices that are not LEFT.

Definition 3 (Segment)**.** A *segment* is a maximal group of dirty indices with structure $(\mathrm{R})^*\mathrm{L}^+$: zero or more RIGHT dirty indices followed by one or more LEFT dirty indices. Clean elements may be interspersed between dirty indices within a segment.



DeltaSort operates in two phases:

1. **Phase 1 (Preparation):** Extract dirty values, sort them, write back to dirty positions in index order. This establishes direction segments in the array which are disjoint and can be repaired independently.

2. **Phase 2 (Repair):** Repair each segment left-to-right, deferring RIGHT indices to a stack until the first LEFT index is encountered. Flush and repair the the RIGHT indexes in stack in LIFO order. Then repair all LEFT indices left-to-right.

4.2 Key Insight: Directional segmentation enables localized repair

The key insight behind DeltaSort is that pre-sorting dirty values induces a *directional segmentation* of updates. After Phase 1, dirty indices partition into disjoint segments of the form $(\mathrm{R})^*\mathrm{L}^+$.

Lemma 4 (Movement Confinement)**.** *Element movement is bounded within each segment: no element crosses a segment boundary.*

Proof. Let S be a segment with RIGHT indices $R_1, \ldots, R_m$ followed by LEFT indices $L_1, \ldots, L_p$. After Phase 1, dirty values are monotonically ordered by index, so $A[R_1] < \cdots < A[R_m] < A[L_1] < \cdots < A[L_p]$.

1. *RIGHT elements cannot pass the leftmost LEFT.* Since $A[R_i] < A[L_1]$ for all i.

2. *LEFT elements cannot pass the rightmost RIGHT.* Since $A[R_m] < A[L_j]$ for all j.

Since no element exits its segment, segments can be repaired independently. $\square$

4.3 Detailed Algorithm

Algorithm 1 DeltaSort

Require: Array $A[0..n-1]$, dirty indices D, comparator `cmp`
Ensure: A is sorted

```
Phase 1: Prepare
dirty ← sort(D)                                        ▷ Sort indices ascending
values ← [A[d] : d ∈ dirty]
values ← sort(values, cmp)
for i ← 0 to |dirty| − 1 do
    A[dirty[i]] ← values[i]
end for

Phase 2: Repair
pending ← []; leftBound ← 0
for p ← 0 to |dirty| − 1 do
    i ← dirty[p]
    if IsLeftViolation(A, i) then
        FlushPending(pending, i − 1)                    ▷ Fix pending before LEFT
        leftBound ← FixLeftViolation(A, i, leftBound) + 1
    else
        pending.push(i)                                 ▷ Defer RIGHT indices
    end if
end for

FlushPending(pending, n − 1)                            ▷ Fix remaining RIGHT violations
```

```
function IsLeftViolation(A, i)
    return i > 0 ∧ cmp(A[i − 1], A[i]) > 0
end function

function IsRightViolation(A, i)
    return i < n − 1 ∧ cmp(A[i], A[i + 1]) > 0
end function
```

```
function FixLeftViolation(A, i, leftBound)
    t ← BinarySearchLeft(A, A[i], leftBound, i − 1)
    Move(A, i, t)
    return t
end function
```

```
function FlushPending(pending, rightBound)
    while pending ≠ ∅ do
        s ← pending.pop()                                  ▷ Process in LIFO order
        if IsRightViolation(A, s) then
            t ← BinarySearchRight(A, A[s], s + 1, rightBound)
            Move(A, s, t)
        end if
    end while
end function
function Move(A, from, to)
    v ← A[from]
    if from < to then
        Shift A[from + 1..to] left by one
    else if from > to then
        Shift A[to..from − 1] right by one
    end if
    A[to] ← v
end function
```

4.4 Correctness Proof

Lemma 5 (Segment Boundary Invariant)**.** *Segment boundaries are never violated. For any two consecutive segments S_i and S_{i+1}, all values in S_i remain less than all values in S_{i+1} throughout the repair process.*

Proof sketch. After Phase 1, dirty values are monotonically ordered by index. A segment boundary occurs where a LEFT dirty index is followed by a RIGHT dirty index. At such a boundary, the RIGHT element satisfies $A[d_{i+1}] \geq A[d_{i+1} - 1]$, establishing sorted order. By Lemma 4, movement is confined within segments, so boundaries remain intact. A fully rigorous proof is deferred to future work. □

Lemma 6 (Fixing Does Not Introduce Violations)**.** *When a LEFT or RIGHT element is moved to its correct position via binary search, no new violations are introduced.*

Proof. LEFT fix: Binary search finds position $t < i$ where the element belongs. Elements in $A[t..i-1]$ shift right by one. The shift preserves relative order. By binary search, the moved element satisfies $A[t-1] \leq A[t] < A[t+1]$.

RIGHT fix: Binary search finds position $t > s$ where the element belongs. Elements in $A[s+1..t]$ shift left by one. The shift preserves relative order. By binary search, the moved element satisfies $A[t-1] < A[t] \leq A[t+1]$.

In both cases, no new violations are introduced. □

Theorem 7 (Correctness)**.** *DeltaSort produces a correctly sorted array.*

Proof. Phase 2 processes each dirty index exactly once, fixing its violation if any. By Lemma 6, each fix resolves a violation without introducing new ones. After all dirty indices are processed, each segment is internally sorted.

By Lemma 5, segment boundaries maintain sorted order throughout. Since each segment is internally sorted and boundaries preserve global order, the entire array is sorted. □

4.5 Algorithm Analysis

Theorem 8 (Time Complexity)**.** *DeltaSort runs in $O(k \log k + k \log n + M)$ time, where M is total movement.*

Proof. **Phase 1**: Sort k indices: $O(k \log k)$. Sort k values: $O(k \log k)$. Write back: $O(k)$.

Phase 2: Each dirty index: $O(1)$ direction check, $O(\log n)$ binary search. Total: $O(k \log n)$. Movement: $O(M)$. □

Theorem 9 (Space Complexity)**.** *DeltaSort uses $O(k)$ auxiliary space.*

Theorem 10 (Comparison Optimality)**.** *DeltaSort achieves $O(k \log n)$ comparisons, which matches the information-theoretic lower bound: each of k dirty elements can occupy any of n final positions, requiring $\log_2(n^k) = k \log n$ comparisons to distinguish all configurations.*

Remark 11 (Movement Efficiency)*.* While worst-case movement is $O(kn)$, the segmentation created by Phase 1 tends to reduce movement in practice in the average case. Empirical results in §5 demonstrate substantial speedups, validating that movement is typically much less than the worst case.

Here we compare algorithm complexity with other standard baseline approaches which we use in experiments to compare DeltaSort's performance:

- **NativeSort (NS)**: Re-sort the array using the natively available sort function. $O(n \log n)$ comparisons and movements.
- **Binary-Insertion-Sort (BIS)**: Extract dirty values, then for each: binary search for correct position, reinsert. $O(k \log n)$ comparisons, $O(kn)$ worst-case movement. Searches the full array range for each insertion.
- **Extract-Sort-Merge (ESM)**: Extract dirty values, sort them, merge with clean elements. $O(k \log k + n)$ comparisons, $O(n)$ movement. Requires $O(n)$ auxiliary space.

Table 1: Algorithm complexity comparison.

Algorithm	Comparisons	Movement	Space
NativeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Binary-Insertion-Sort	$O(k \log n)$	$O(kn)$	$O(1)$
Extract-Sort-Merge	$O(k \log k + n)$	$O(n)$	$O(n)$
DeltaSort	$O(k \log n)$	$O(kn)$*	$O(k)$

*Worst case; average case is closer to $O(n)$.

5 Experimental Evaluation

All experiments are run using a Rust implementation of DeltaSort on a synthetic dataset of user objects with composite keys (country, age, name) on a M3 Pro Macbook Pro with 18GB RAM. DeltaSort is compared against three baselines: Native sort (Rust's `sort_by`), Binary Insertion Sort (BIS), and Extract-Sort-Merge (ESM).

5.1 Correctness

Correctness is verified by an extensive set of randomized unit tests across various scales and update sizes. The test routine generates a sorted base array of size N, applies k random updates at random indices, runs DeltaSort, and asserts that the final array is sorted and contains all original elements with updated values. All tests pass successfully.

5.2 Execution Time

The graph below shows execution time (in microseconds) for $n = 50,000$ elements as a function of dirty count k. DeltaSort consistently outperforms all alternatives up to approximately $k = 15K$ (crossover point), achieving significant speedups of 6–20× over Native sort in the intermediate range. Also note the orders-of-magnitude gap between DeltaSort and the baseline incremental algorithms (BIS and ESM) across the full range of k values.

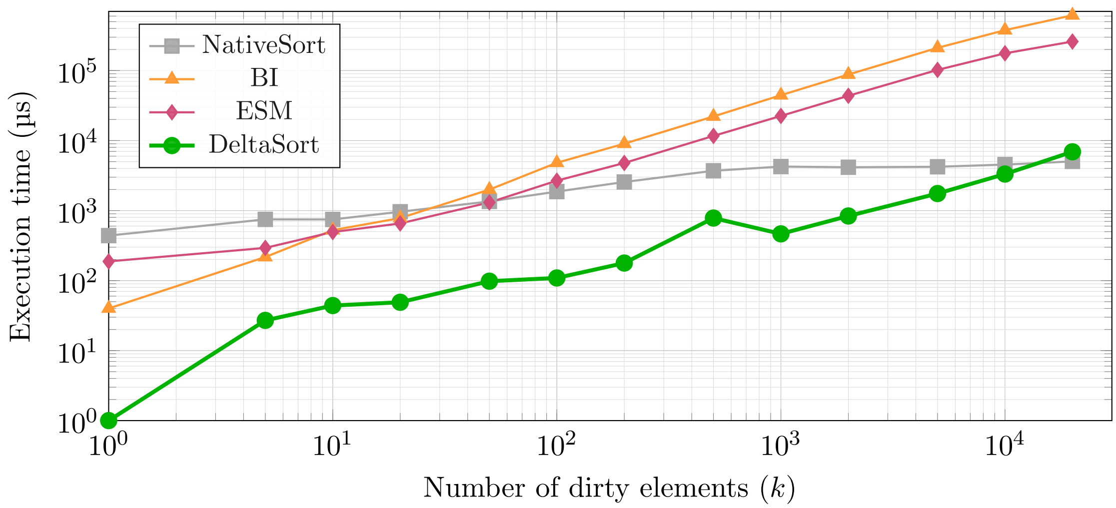


Figure 1: Execution time comparison of all algorithms for $n = 50,000$ (log-log scale)

5.3 Comparator Invocation Count

The following chart shows the number of comparator invocations for each algorithm. DeltaSort and Binary Insertion both achieve $O(k \log n)$ comparisons, substantially fewer than Native sort's $O(n \log n)$ and ESM's $O(k \log k + n)$. The comparison counts for DeltaSort are 10-40% more than BI , confirming that DeltaSort's performance advantage comes from reduced data movement.

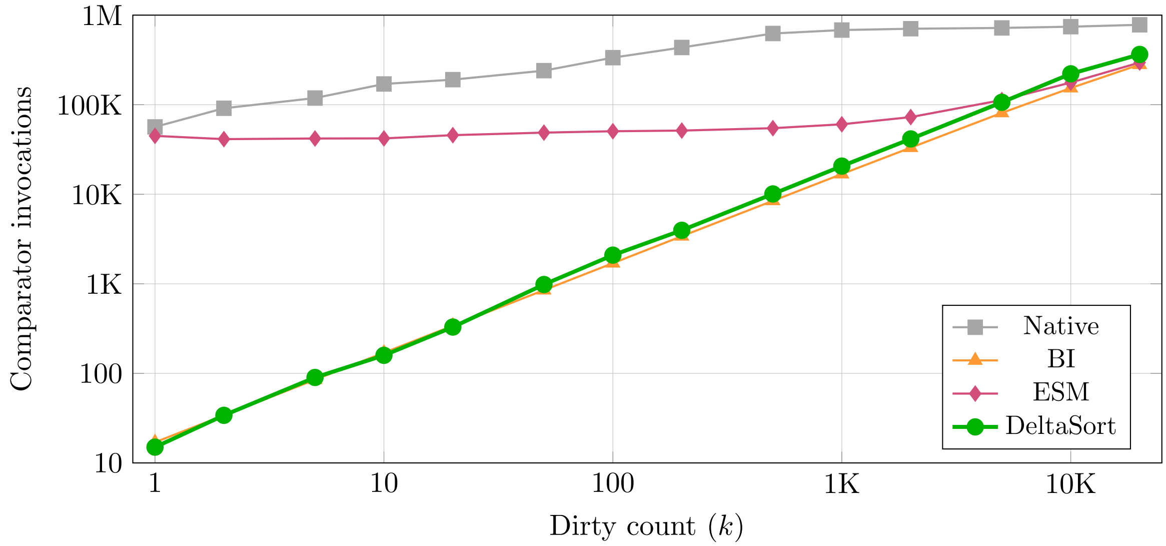


Figure 2: Comparator invocation count for n = 50,000 elements. DeltaSort and BI both achieve $O(k \log n)$ comparisons, while Native uses $O(n \log n)$ regardless of k, and ESM uses $O(k \log k + n)$. The similar comparison counts for DeltaSort and BI confirm that DeltaSort's speedup derives from movement efficiency, not fewer comparisons.

5.4 Crossover Threshold Analysis

A key practical question is: at what delta size should one switch from DeltaSort to Native sort? A binary search was conducted for the crossover point k_c across array sizes from 1K to 10M elements.

Figure 3 visualizes how the crossover ratio k_c/n varies with array size. The ratio peaks around 31% for medium-sized arrays ($n \approx 50$K) and declines for very large arrays, suggesting that DeltaSort's advantage narrows arrays grow very large. More study is needed to understand this trend fully.

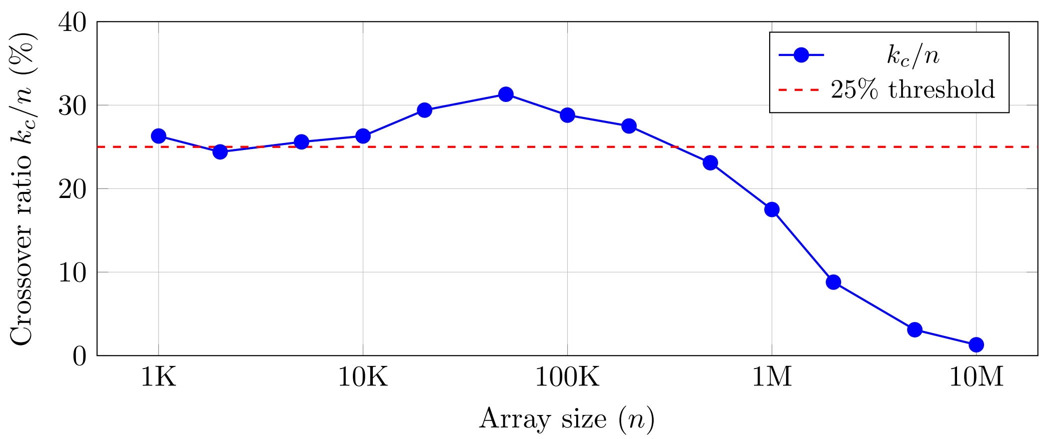


Figure 3: Crossover ratio k_c/n as a function of array size. DeltaSort outperforms Native sort when the dirty fraction is below this curve. The 25% rule of thumb (dashed line) is conservative for small-to-medium arrays but optimistic for very large arrays where the crossover drops significantly.

The key takeway is that DeltaSort is beneficial for **large** ranges of update sizes. So DeltaSort remains a viable choice even when delta sizes are large. The exact threshold depends on the scenario specifics (array sizes, data types, comparator cost, etc.) but the results suggest that a **25% dirty fraction** is a reasonable rule of thumb for the upper limit of delta size before which DeltaSort remains advantageous.

5.5 Performance in managed execution environments

DeltaSort was also implemented in JavaScript running on Node.js v20 (V8 engine). While the implementation passes all correctness tests, the performance results show higher variance due to JIT compilation behavior, garbage collection pauses, and other engine-level effects. The JavaScript benchmarks will be refined in a future revision to provide more stable measurements. The Rust implementation provides the authoritative performance characterization.

6 Future Work

This work suggests several directions for future investigation:

- **Stronger theoretical characterization:** While the correctness argument and extensive randomized testing establish that DeltaSort produces a correctly sorted array, certain invariants—particularly those governing inter-segment boundaries—deserve a more formal treatment that could yield sharper theoretical guarantees.

- **Average-case movement bounds:** Beyond worst-case correctness and empirical evaluation, establishing average-case bounds on data movement under reasonable update distributions may help explain observed crossover behavior and clarify when coordinated repair is most effective.

- **Structured workload analysis:** The current evaluation relies on randomized updates, whereas many real workloads exhibit additional structure, such as gradual value changes in leaderboards or localized updates in interactive list views. Studying such patterns may reveal regimes where DeltaSort's advantages are amplified or diminished.

- **Runtime variance analysis.** Performance variance in managed runtimes warrants deeper investigation. Understanding the impact of factors such as memory allocation, garbage collection, and JIT compilation could improve result interpretability and guide implementation choices.

- **Block-structured storage.** Although this work focuses on in-memory arrays, the update-aware model naturally extends to block-structured storage. Exploring how DeltaSort-style coordination interacts with page- or block-based layouts may clarify its applicability to database and external-memory settings.

7 Conclusion

This paper introduced *DeltaSort*, an incremental repair algorithm for maintaining sorted arrays under batched updates. The central insight is that pre-sorting updated values induces *directional segmentation*: dirty elements naturally partition into segments that can be repaired independently.

DeltaSort leverages this segmentation through stack-based processing. LEFT-moving elements are repaired immediately with progressively narrowed search ranges, while RIGHT-moving elements are deferred and processed in reverse order to ensure stable target positions. This coordination avoids redundant comparisons and overlapping element movement that arise from repeated binary insertion.

An experimental evaluation in Rust demonstrates that DeltaSort consistently outperforms both blind native sorting and repeated binary insertion across a wide range of array sizes and update volumes. In practice, DeltaSort remains advantageous until approximately 25% of the array is dirty, providing a clear and actionable decision boundary.

More broadly, this work highlights the value of integrating application-level update information into core algorithms. When sorting routines are informed of which elements changed, coordination becomes possible, enabling performance improvements that blind algorithms cannot realize. DeltaSort illustrates how modest structural insight—segmentation combined with disciplined processing order—can yield substantial practical gains.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[3] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $o(n \log n)$. In *Proceedings of the 3rd International Conference on Fun with Algorithms*, pages 16–23, 2004.

[4] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[5] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[6] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 100(4):318–325, 1985.

[7] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[8] Orson R. L. Peters. Pattern-defeating quicksort. `https://github.com/orlp/pdqsort`, 2021.

[9] Tim Peters. Timsort. Python source code, 2002. Adopted by Java SE 7, Android, and V8.

[10] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.