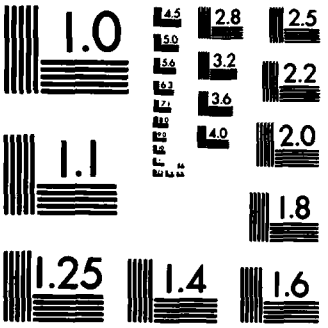
AD-A158 610 DYNAMIC MONOTONE PRIORITIES ON PLANAR SETS (EXTENDED ABSTRACT)(U) YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE M J FISCHER ET AL. JUL 85 YALEU/DCS/TR-415 1/1

UNCLASSIFIED N00014-82-K-0154 F/G 12/1 NL



DYNAMIC MONOTONE PRIORITIES ON PLANAR SETS
(EXTENDED ABSTRACT)

Michael J. Fischer and Michael S. Paterson

YALEU/DCS/TR-415
July, 1985

YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

DYNAMIC MONOTONE PRIORITIES ON PLANAR SETS
(EXTENDED ABSTRACT)

Michael J. Fischer and Michael S. Paterson

YALEU/DCS/TR-415
July, 1985

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: 415	2. GOVT ACCESSION NO.: AD-A158 610
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): DYNAMIC MONOTONE PRIORITIES ON PLANAR SETS (Extended Abstract)	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Michael J. Fischer and Michael S. Paterson	8. CONTRACT OR GRANT NUMBER(s): ONR: N00014-82-K-0154; and a Senior Fellowship from the Science and Engineering Research Council
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science/ Yale University Dunham Lab./ 10 Hillhouse Avenue New Haven, Connecticut 06520	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research 800 N. Quincy Arlington, VA 22217 ATTN: R.B. Grafton	12. REPORT DATE: July, 1985
	13. NUMBER OF PAGES: 7
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distributed unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

data structure
priority queue
maximum-finding
jump assembly

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A monotonic priority set is a new data structure which supports maximum-finding and deletions over a set of weighted points in the plane. Global updates to the weights can also be made, incrementing the weights of all points above a given threshold in one of the coordinates. The weights are assumed to be always monotonic in both coordinates. An efficient implementation of this structure is presented and two main applications are described. The first is to the problem of optimal assembly of code for computers with two kinds of jump instruction: long and short. The task in the second application is the implementation of a queuing discipline based on the ranks with respect to two different criteria.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Dynamic Monotone Priorities on Planar Sets[†]

(EXTENDED ABSTRACT)

Michael J. Fischer
Yale University
New Haven, Connecticut

Michael S. Paterson
University of Warwick
Coventry, England

Abstract

A monotonic priority set is a new data structure which supports maximum-finding and deletions over a set of weighted points in the plane. Global updates to the weights can also be made, incrementing the weights of all points above a given threshold in one of the coordinates. The weights are assumed to be always monotonic in both coordinates.

An efficient implementation of this structure is presented and two main applications are described. The first is to the problem of optimal assembly of code for computers with two kinds of jump instruction: long and short. The task in the second application is the implementation of a queuing discipline based on the ranks with respect to two different criteria.

1 Monotonic Priority Sets

Let P be a multiset of points $p = (x, y)$ in the plane. P is partially ordered by the product ordering defined by $(x_0, y_0) \leq (x_1, y_1)$ iff $x_0 \leq x_1$ and $y_0 \leq y_1$. Each point p has an associated weight, $w(p)$. This real-valued weight function w is monotonic in the sense that if $p_1 < p_2$ then $w(p_1) \leq w(p_2)$. We define the following operations on P.

- FIND_MAX returns a point in P of maximum weight.
- DELETE(p) removes p from P.
- V_UPDATE(y_0, a) adds a to the weight of each point (x, y) with $y \geq y_0$.
- H_UPDATE(x_0, a) adds a to the weight of each point (x, y) with $x \geq x_0$.

We assume that the updates preserve the monotonicity of w. (This is assured if every a is non-negative.) A data structure that supports these operations is called a *monotonic priority set*. In the next section, we describe an implementation of a monotonic priority set on n points in which the total cost for m operations is $O((n+m)\log(n+m))$. The remaining sections discuss applications of this data structure and a complementary NP-completeness result.

2 Efficient Implementation of Monotonic Priority Sets

A naive implementation of monotonic priority sets simply keeps a list of all points not yet deleted along with their current weights. Each operation is performed with an entire scan of the list.

A first idea for gaining efficiency is to associate the points with the leaves of a balanced binary tree and to store at each leaf the weight of the

[†]This work was supported in part by the Office of Naval Research under Contract Number N00014-82-K-0154 and by a Senior Fellowship from the Science and Engineering Research Council.

corresponding point and at each internal node the maximum weight from its subtree. This permits FIND_MAX and DELETE to be performed in time $O(\log n)$. Each H_UPDATE and V_UPDATE however might affect the weights of $O(n)$ leaves and hence require that much time when done in a straightforward way.

One way to update groups of leaves at once is to place an offset value ψ_i at internal node i to be regarded as an increment to all weights in subtree i. Thus, the current weight at any node is its initial weight plus the sum of the offsets on the path to the root. If the leaves were sorted by their x-coordinates, then an H_UPDATE could be effected by altering offsets on at most $\log n$ nodes. Similarly, V_UPDATEs would be easy if the leaves were sorted by their y-coordinates.

For a pair of points (x_0, y_0) and (x_1, y_1) incomparable in the product ordering, $x_0 \leq x_1$ iff $y_0 \geq y_1$. Any independent (i.e. pairwise incomparable) set of points can therefore be ordered (from left to right) so that the x-coordinates are non-decreasing and the y-coordinates are non-increasing, permitting both H_UPDATE and V_UPDATE operations to be performed efficiently as described above.

We choose for our independent set I a maximal independent set of (undeleted) points that are maximal in the product ordering. Since P is a multiset, I may not be unique, so we specify further that among the possibly several copies of the point (x, y), we choose one of largest weight for inclusion in I. By the monotonicity condition, I contains some point with maximum weight among all of the undeleted points in P. Thus, FIND_MAX and the two updates can all be performed on this structure in time $O(\log n)$.

However following a DELETE of some element from the set I, new elements may need to be brought into I to restore the maximality condition. This leaves us with two problems: (1) finding which elements to insert into I; (2) inserting them with their current weights.

Suppose a maximal point is deleted, and let (x_L, y_L) and (x_R, y_R) be its left and right neighbors in I, respectively. We introduce for convenience virtual end elements for I, $p_{-\infty} = (-\infty, +\infty)$ and $p_{+\infty} = (+\infty, -\infty)$, so that these neighbors are always defined. Any new point (x, y) to be added to I satisfies $x_L < x < x_R$ and $y_L > y > y_R$. To assist in finding all such points, we will have initially ordered the whole of P in lexicographically increasing order, first on x, then on y, and finally on the initial weights. Starting at (x_R, y_R), we search to the left in the lexicographic ordering for the first undeleted point (x_1, y_1) with $y_1 > y_R$. Starting at (x_1, y_1), we search again to the left for the first point (x_2, y_2) with $y_2 > y_1$. We continue in this way until we reach the point (x_L, y_L). The points $(x_1, y_1), (x_2, y_2), \ldots$ obtained are the new maximal points to insert into I. A data structure consisting of a balanced binary tree with the leaves so ordered and with the maximum y coordinate for each subtree stored at the corresponding internal node supports this search for each successive point efficiently.[1] Each new point is found in time $O(\log n)$, and deletion of any element from this set can be done in this time also. The set I is initialized by performing the search with $(x_L, y_L) = p_{-\infty}$ and $(x_R, y_R) = p_{+\infty}$.

To calculate the current weight for each new maximal element, we use two additional trees T_H and T_V to record the H_UPDATEs and V_UPDATEs respectively. The leaves of T_H are the pairs $\langle x_i, a_i \rangle$ corresponding to each H_UPDATE, ordered by the x_i's. Internal nodes store the sum of the a-values of the leaves in their subtrees. Similarly, T_V records the pairs $\langle y_j, b_j \rangle$ corresponding to V_UPDATEs. The current weight w of a point $p = (u, v)$ with initial weight w_0 is given by

$$w = w_0 + \sum_{i:\, x_i \leq u} a_i + \sum_{j:\, y_j \leq v} b_j.$$

Each sum can be computed in time proportional to the height of the tree. Using balanced tree techniques, the height is $O(\log m)$ after m updates, and the time to maintain the trees after each update is also $O(\log m)$.

[1] Knuth attributes the idea for such a data structure to McCreight [1].

3 Optimal Assembly of Jumps

3.1 Short Jump to Long Jump Conversion

Monotonic priority sets have application to a problem of assembling code for computers with two sizes of jump instructions. Short jumps execute quickly, but the distance they can jump, their *span*, is limited, for example, to about 128 bytes in either direction. Long jumps on the other hand are to an arbitrary destination, but are more costly, both in execution time and program size, since a long jump instruction needs more address bytes. Choosing which kind of jump instruction to use will normally be done by the assembler. The problem is a nontrivial one since making a given jump long increases the span of any jump traversing it. It is possible to construct chains of overlapping jump instructions such that each jump that is made long causes exactly one other to exceed the maximum span for a short jump, so the latter too must be made long.

It is easy to see that a minimal size program can be obtained by starting with all jumps short and traversing the program repeatedly, converting short jumps to long on each pass as necessary. A straightforward implementation of this strategy however is quite inefficient since up to $O(n)$ passes may be required for a program containing n jumps.

A more conservative approach is to check just those short jumps within a distance S of each new conversion, where S is the maximum span of a short jump, for these are the only new candidates for conversion. This approach leads to an $O(nS)$ time algorithm.

Using a monotonic priority set, we obtain an algorithm for optimal assembly of jumps that runs in time $O(n \log n)$, independent of S. A jump from x to y is represented by the point $(-\min(x,y), \max(x,y))$ with initial weight $|x-y|$. FIND_MAX returns the remaining short jump of largest span. It will need conversion to a long jump if any jump does. Assuming it does, we then delete it from the priority set and update the priorities of the remaining elements by executing V_UPDATE(z,b) and H_UPDATE$(-z,b)$. Here we assume that the conversion requires the insertion of b extra bytes at position z. (Typically, $z = x$.) The two given UPDATE's actually increment those jumps which span z by $2b$ and all other jumps by b, preserving the priorities as required. If we wish the weights to equal the current jump spans, then a further operation of say V_UPDATE$(-\infty, -b)$ could be used to restore the proper weights. The time bounds follow since in this case $m = O(n)$.

3.2 Multilevel Jumps

In a logical extension to the above problem we may have further levels of jumps, for example short, long, and huge. This problem is handled using two monotonic priority sets, one to deal with conversions from short to long and the other for conversions from long to huge. All jumps start as members of both sets. A jump is deleted from the first set when it is promoted from short to long, and it is deleted from the second set when it is made huge, but no jump is deleted from the second set until after it has been deleted from the first. In slightly more detail, at each stage an attempt is made to convert a short jump to long, and only if that fails is an attempt made to convert a long jump to huge. All updates are applied to both sets. Any finite number of jump levels can be handled in the same way, as can the case of short and long jumps in which the maximum span of a short jump depends on its direction, whether forward or backward in memory. A continuous analogue, in which a jump of distance d has size $\log d$, appears to be much more difficult, and we leave that problem open.

3.3 *NP*-Completeness for Shared Jumps

A reduction of program length may be possible if we allow a jump to a distant destination to be assembled as a chain of two jumps, a short followed by a long, for then several nearby short jumps could share the same long jump.

Though related to the problem of Section 3.1, optimization with shared jumps appears to be very hard. Some microprocessor codes have only

short conditional jumps, so that any long conditional jump always requires two instructions. We can show in this case that given a program and an integer k, the problem of testing whether the jumps can be assembled to yield a program of length at most k is $\mathcal{NP}$-complete. A proof of this result will appear in the final version of this paper.

4 Double Priority Queues

Suppose we are processing a set of elements ranked by two criteria, for example, expected run time and memory size of computer jobs. The next element to be processed and deleted is one for which the sum of its two ranks among the remaining elements is minimal. This queueing procedure can be implemented efficiently using priority sets. An element e is identified with the point $(-x, -y)$ in a priority set, where x and y are its initial ranks, and its initial weight is $-x-y$. When e is deleted, H_UPDATE$(-x,-1)$ and V_UPDATE$(-y,-1)$ are performed to make the weights reflect the sum of the current ranks. Even though the increments are negative, we can prove that the updates preserve monotonicity, so the priority set implementation of Section 2 can be applied.

5 Conclusion and Open Problems

Both the monotonicity requirement and the restriction to the plane appear essential to our approach. Relaxation of either constraint would seem to require new ideas.

The "log" factor in the time bound seems difficult to eliminate. For example, even when the points are given in order of decreasing initial weights, at least $\Omega(n \log n)$ comparisons may be needed in the case that the weights can be arbitrary real numbers. To see this, take n points, $p_1 = (1, b_1 - 2), \ldots, p_n = (n, b_n - 2n)$, in this order, where $0 < b_i < 1$ and $w(p_i) = b_i - i$, $1 \leq i \leq n$. Then perform H_UPDATE$(i, 1)$ for each $i \in \{1, \ldots, n\}$. Now the resulting weights are $b_1, \ldots, b_n$, and a sequence of n DELETE(FIND_MAX) operations removes the points from the priority set in order of decreasing value of b_i, thereby effecting a sort of $\{b_1, \ldots, b_n\}$.

Acknowledgement

We are grateful to Arnold Schönhage for stimulating our interest in the jump assembly problem and for contributing to its solution.

References

[1] Knuth, Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Massachusetts, 1973, Section 6.2.3, Exercise 30, p. 471, and solution, p. 678.

DISTRIBUTION LIST

Office of Naval Research Contract NQ0014-82-K-0154

Michael J. Fischer, Principal Investigator

Defense Technical Information Center
Building 5, Cameron Station
Alexandria, VA 22314
(12 copies)

Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Dr. R.B. Grafton, Scientific Officer (1 copy)

Information Systems Program (437)
(2 copies)

Code 200 (1 copy)
Code 455 (1 copy)
Code 458 (1 copy)

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106
(1 copy)

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
(1 copy)

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380
(1 copy)

Naval Ocean Systems Center
Advanced Software Technology Division
Code 5200
San Diego, CA 92152
(1 copy)

Mr. E.R. Gleissner
Naval Ship Research and Development Center
Computation and Mathematics Department
Bethesda, MD 20084
(1 copy)

Captain Grace M. Hopper
Naval Data Automation Command
Washington Navy Yard
Building 166
Washington, D.C. 20374
(1 copy)

Defense Advance Research Projects Agency
ATTN: Program Management/MIS
1400 Wilson Boulevard
Arlington, VA 22209
(3 copies)

END

FILMED

10-85

DTIC