Submatrix maximum queries in Monge matrices: an implementation

Raphaël Bost*†

Abstract

We propose an implementation of the data structure presented by Kaplan, Mozes, Nussbaum, and Sharir in [KMNS12] for submatrix maximum queries in Monge matrices. The implementation shows that the average running time of the algorithm is similar to the proved worse case running time: $O(\log^3 n)$. We also propose a new efficient and low-memory consuming procedure to generate random Monge matrices.

Introduction

Monge matrices are objects that we naturally encounter when working on planar graph algorithms, especially when dealing with shortest paths as suggested in [FR01]. Accordingly, the principal motivation for implementing this algorithm was to provide a working implementation of the quasi-linear multiple source multiple sink max flow algorithm of Borradaile *et al.* in [BKM+11] which heavily relies on Monge matrices properties to achieve the quasi-linear running time.

A critical operation on Monge matrices in the algorithm of [BKM+11] is the determination of the minimum value of a submatrix of a given Monge matrix. In [KMNS12], Kaplan, Mozes, Nussbaum, and Sharir introduce a data structure that answers such queries in amortized $O(\log^2 n)$ time with a $O(n \log^2 n)$ preprocessing time for a square Monge matrix of size $n \times n$. Consequently we decided to implement this data structure as a building brick of the bigger max flow algorithm.

The goal of our implementation is to provide some fast and easy to use code for the data structure. Among the important criteria that governed our work were correctness, efficiency, portability, and re-usability. It has been written in C++ and does not rely on any library but the C++ standard library. The code is templated so it can use any arithmetic type without casting.

In order to complete some tests and benchmarks, we also had to develop a method to build sample Monge matrices. We will see why this step is very important to check the correctness of the implementation and evaluate its efficiency. The different benchmarks we will present give a new insight on the algorithm and its practical use.

*Department of Computer Science, Brown University

†Direction Générale pour l'Armement

1 The algorithm

In this section we quickly recall the algorithm for submatrix maximum queries in Monge matrices as described in [KMNS12].

1.1 Preliminaries

1.1.1 Monge matrices

A matrix $M = (M_{ij})$ is a *totally monotone* matrix if for every pair of rows $i < j$ and every pair of columns $k < l$, $M_{ik} \leq M_{il}$ implies $M_{jk} \leq M_{jl}$

A matrix M is a *Monge* matrix (resp. an *inverse Monge* matrix) if for every pair of rows $i < j$ and every pair of columns $k < l$, $M_{ik} + M_{jl} \leq M_{il} + M_{jk}$ (resp. $M_{ik} + M_{jl} \geq M_{il} + M_{jk}$). It is easy to show that, if M is inverse Monge, then M and M^t are totally monotone.

1.1.2 Pseudo-lines and envelopes

We call a set L of curves in the plane a family of pseudo-lines if every pair of curves intersect in at most one point and the two curves cross each other transversally there (there is no "tangency" point). We can think of a pseudo-line l as a map $l : I \to \mathbb{R}$ where $I \subseteq \mathbb{R}$, I is an interval. Two pseudo-lines l_1, l_2 of the same family are such that if $\exists x_0 \in I$ s.t. $l_1(x_0) = l_2(x_0)$ then $\forall x_1 < x_0 < x_2,\ (l_1(x_1) - l_2(x_1))(l_1(x_2) - l_2(x_2)) < 0$ or said otherwise $l_1(x_1) < l_2(x_1) \Rightarrow l_1(x_2) > l_2(x_2)$ and $l_1(x_1) > l_2(x_1) \Rightarrow l_1(x_2) < l_2(x_2)$.

The *upper envelope* of a set of pseudo-lines L is the function

$$\mathcal{E}_L(x) = \max_{l \in L} l(x)$$

A *breakpoint* in the upper envelope of a set of pseudo-lines is an intersection point of two pseudo-lines on $\mathcal{E}_L$. We define the *size* of an envelope $\mathcal{E}_L$ as the number of its breakpoints. Since each pseudo-line in L can be maximal among all the other pseudo-lines in a single interval at most, the size of $\mathcal{E}_L$ is at most $|L| - 1$.

We can apply these notions of pseudo-lines and envelopes to inverse Monge matrices: as an inverse Monge matrix is also totally monotone, we can interpret its rows as a family of pseudo-lines. Namely for an inverse Monge matrix M of size $m \times n$, we can define $\hat{M}_i : [1, n] \to \mathbb{R}$ by $\hat{M}_i(k) = M_{ik}$ for $k \in \{1, \ldots, n\}$ and by linearly interpolating between these points. Moreover, if $i < j$, the pseudo-line corresponding to the row j lies under the one corresponding to row i before the intersection point (cf. the definition of monotone matrices).

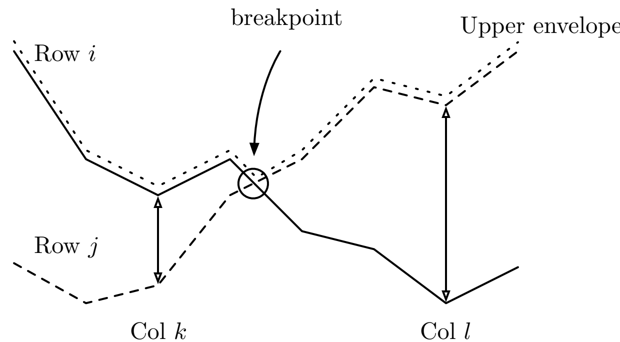


Figure 1: Pseudo-lines and envelope in a matrix

1.2 The data structure

1.2.1 The model

We suppose that we are given an oracle access to the matrix M. That is, we can access any entry M_{ij} in $O(1)$ time (also, we do not have to keep the entire matrix M in memory. The matrix M can be too big to entirely fit in memory).

Previously, we supposed M to have real entries. Actually, the data structure will work if the entry space E is an arbitrary set given a total order and if M is inverse Monge with respect to that order.

In the following, M will be an $m \times n$ inverse Monge matrix.

1.2.2 Maximum for one column

As we can see with the definitions in 1.1.2, the upper envelope of a set of rows consists of the column maxima for these rows. So the idea is to compute well chosen upper envelopes once and for all, and then search among theses envelopes for the maximum. However, if we explicitly represent the envelopes, we will need $O(n)$ values for each of them as, whereas an envelope is entirely defined by its breakpoints. So we need $O(k)$ records to represent an envelope, where k is the number of pseudo-lines in the envelope. So we use this succinct and implicit representation that saves a lot of space when the number of rows in the envelope is smaller than the number of columns in the matrix.

To find the maximum of a column π among a set of rows L whose upper envelope is known, we just have to find the breakpoints located just before and after π in the envelope representation using a binary search and once we have found this interval, we know to which pseudo-line (say the one corresponding to row ρ) it belongs to. Then the maximum is just $M_{\rho\pi}$.

We still need to construct the implicit representation of the upper envelopes. To do so, we build a balanced binary tree T over the rows of M. For each node u of T, the upper envelope of the pseudo-lines representing the rows in the subtree rooted at u is computed rootward and stored in u. The envelope of a leaf is trivial as it only represents a row. For a non-leaf node u, we compute its envelope by merging the envelopes of its children w_1 and w_2 (w_1 is the node with lower indices, w_2 with higher indices): by the monotonicity of M, the envelope of u starts with rows in w_1, reaches a new breakpoint, where w_2's upper envelope is over w_1's one and ends with a suffix of w_2's envelope. So we first find the location of the breakpoint between the envelopes of w_1 and w_2 and then concatenate the prefix of w_1's envelope, the new breakpoint, and the suffix of w_2's envelope. In the following, when it exists, we will call the newly inserted breakpoint the *crossing breakpoint* of u.

A query to this data structure consists in a column π and a range of rows $[\rho, \rho']$ and the output is the maximum value of M is column π between rows ρ and ρ'. To answer it, we first retrieve the *canonical nodes* of T with respect of $[\rho, \rho']$ *i.e.* the minimal set of nodes whose set of rows are disjoint and cover $[\rho, \rho']$ (the set of rows of each of this node is contained in $[\rho, \rho']$ but this is not the case for its parents, *cf.* Figure 2). For each of these nodes, we compute the maximum for the queried column over the rows represented by the node and we take the greatest of these maxima.

It has been shown (*cf.* [KMNS12]) that this data structure can be built in $O(m(\log m + \log n))$ time and that a query takes $O(\log^2 m)$ time. One can also consider the transposed matrix and build a similar data structure over the columns of a matrix and answer efficiently to a maximum query over a row and a range of columns.

1.2.3 Maximum for a submatrix

To construct a data structure that answers efficiently maximum queries over submatrices of a matrix, we use the previous one as a base component. We build the tree $\mathcal{T}$ over the rows of M as before. We also construct the "flipped" data structure *i.e.* the tree $\mathcal{B}$ over the columns that can answer to maximum queries over a row and a range of columns.

Then we *extend* the tree $\mathcal{T}$ by adding some extra informations in its nodes: for every node u, we find the maxima for every interval of the upper envelope stored in u. As each interval in the envelope corresponds

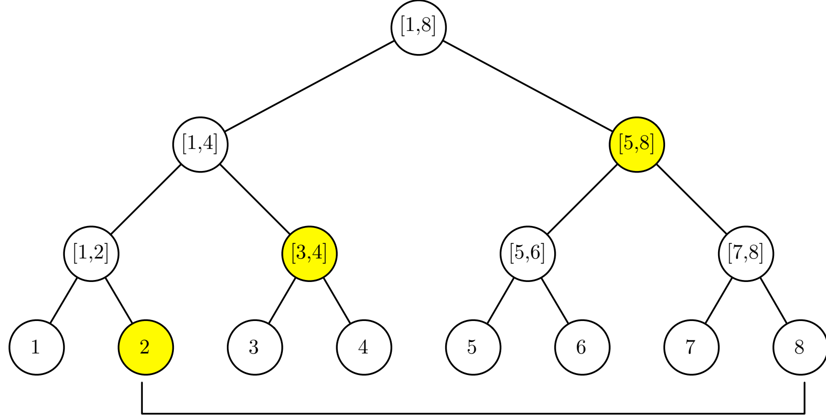


Figure 2: Binary tree for with 8 leaves and canonical nodes for range $[2, 8]$

indeed to a single row (the greatest row in that interval among the rows represented by the node) and to a range of columns (the interval where the row is maximal), we use the flipped tree $\mathcal{B}$ find efficiently all the maxima. Finally, we store them in a range-maximum-query data structure.

A query consists in a range of consecutive rows R and a range of consecutive columns C and the output is the maximum value for M in the submatrix defined by R and C. As before, we find the canonical nodes of $\mathcal{T}$ with respect to R and for each node we find the maximum of the upper envelope in the range C.

To do so, we find the maximum set $\mathcal{I}$ of intervals of the upper envelope that are entirely contained in C. The maximum value among this set is given by the range maximum query data structure. We still have to find the maxima of the *prefix* and of the *suffix* of C to return the maximum over C. The prefix p of C that is not covered by $\mathcal{I}$ is, by construction of $\mathcal{I}$ contained in an interval of the upper envelope. Thus, p corresponds to a single row in the upper envelope and hence we can retrieve the maximum of the upper envelope within p using a query to $\mathcal{B}$. We use the same idea to find the maximum with the suffix s of C (not covered by $\mathcal{I}$).

Constructing the data structure for an $m \times n$ inverse Monge matrix takes $O(m \log m \log^2 n + n(\log m + \log n))$ time and a query takes $O(\log m(\log m + \log^2 n))$ (we refer the reader to [KMNS12] for the analysis[1]).

2 Implementation

2.1 Implementation choices

Our goals when implementing the algorithm was to provide code that is efficient (both in terms of computation time and memory use), portable, and easy to reuse. Thus we made some choices for the coding environment:

- **Language:** We decided to have an Object-Oriented approach for the development: a non-object oriented language would have forced to do a lot of undesirable code duplication.

  Because the speed of the implementation was one of our first concerns, we immediately considered C++ as our first choice for a language compared to other OO languages like Java. C++ also provides us a low memory print, at the cost of manual memory management. But in our situation, memory management is not a big problem as we mostly work with trees and arrays.

- **Libraries:** Using or not external libraries, different from the C++ standard library was not a real concern because all we needed for classes and data structures was already build in the standard library.

[1] A careful reader will observe that the complexity result is sightly better than the one we mention here. This is because we did not use fractional cascading while they do.

Actually, the only class of the `stdlib` that we use is the `vector` class.[2] In the same vein, the code does not use C++11 features.

This makes the code very portable: it is compiling and running without any modification on Linux and MacOS X machines using `gcc`, `clang` and `icc`.

- **Building:** We used `make` in the first stage of this work, but we finally switched to `SCons`[3] for a construction tool because we wanted a tool that easily takes the platform into account and that can be parametrized from the command line (for example to choose the compiler).
- **License:** For maximum reusability among the community, the code has been published under the MIT License[4] and is available on GitHub[5].

2.2 The code

2.2.1 Paradigms of conception

The two main ideas we kept in mind while implementing the algorithm were *code reusability* and *"small is beautiful"*: we tried to minimize code duplication (*e.g.* using common superclasses) and to create small functions instead of big blocks of code.

Similarly, as we wanted to be as general as possible, we decided to use templates. So, our classes depend on a type that is the type of the matrix data. Thus, C++ operator overloading mechanism allows us to use any type as long as a comparison operator is defined and that without changing the code at all.

Because we wanted high reusability, we also chose a specific design for the input data. As we mentioned in 1.2.1, we use an "oracle model" to access the matrix data. So we provide an abstract class `Matrix` that defines all the methods the data structure needs to be constructed and a user of the data structure only has to subclass it in order to create a interfacing layer between the data structure and its data.[6]

2.2.2 Classes

In this paragraph, we only cover the core classes, not the code developed for testing or benchmarking purposes.

`Matrix`: As we said before, `Matrix` is an abstract class that defines all the methods the data structure needs to construct itself given an input matrix. In particular, we need to have access to the number of rows and columns and to the value stored in a cell, so this class defines these accessors. It also defines convenience methods to check for Monge or inverse Monge property.

`MaxValue`: This is a simple wrapper to store a maximum value and some additional informations about this maximum (in our case, the position of the maximum in the matrix).

`BasicRQNode`: We define a range-maximum-query data structure based on binary trees. `BasicRQNode` is the class for the nodes of these trees. It stores the minimum and maximum index of the subtree rooted at the node as well as its maximum value and pointers to its children. It also defines a `query` method that takes a range as input and outputs the maximum value stored in the tree for nodes contained in this range.

Actually, a `BasicRQNode` is constructed taking a comparison function as argument, so we do not necessary build a tree for a range maximum data structure but we can also build a range minimum data structure or we can consider any total order we want to build the tree.

[2] And we could do without it. The present version of the core code (*i.e.* the code without the test and benchmarking features) could be easily rewritten without using any class of the standard library

[3] `http://www.scons.org`

[4] `http://opensource.org/licenses/MIT`

[5] `https://github.com/rbost/SubmatrixQueries`

[6] This `Matrix` abstract class acts pretty much like an interface in Java or a protocol in Objective-C.

`Breakpoint:` The class that represents breakpoints. It simply store the couple of indices representing the position of the breakpoint in the matrix.

`Envelope`, `RowEnvelope` and `ColumnEnvelope`: `Envelope` is an abstract class that represents an upper envelope. We created this class to *factor* the code of its two concrete subclasses that are `RowEnvelope` and `ColumnEnvelope`. This allows us to have a single method to locate a breakpoint in an envelope and a single function that merges two envelopes (instead of having two, one for each class).

`EnvTreeNode`, `RowNode` and `ColNode`: In the same vein than for envelopes, `EnvTreeNode` is an abstract class for representing an envelope tree as defined in 1.2.2, whose concrete subclasses are `RowNode` and `ColNode`. In particular, if `RowNode` and `ColNode` are built to provide correct initialization and correct access to the datas, all the hard work is done by `EnvTreeNode`. All the code that answers to a maximum query is located in that class.

Actually, there even are three sightly different implementations of the query algorithm. We will come back to that point later.

`ExtendedRowNode` This subclass of `RowNode` implements the envelope trees augmented with interval maxima, as described in 1.2.3. More in detail, it has an extra field that is a range maximum query (in our case an instance of `BasicRQNode`) that is filled using a “flipped tree” (an instance of `ColNode`).

`SubmatrixQueriesDataStructure:` This is a wrapper that initializes and stores an instance of `ExtendedRowNode` (the extended tree over the rows) and an instance of `ColNode` (“the flipped tree” over the columns) given a `Matrix` object given as input of its constructor. This class also implements the code needed to answer maximum submatrix queries.

`SubmatrixQueriesDataStructure` has two different implementations of the querying algorithm (we will describe the differences later).

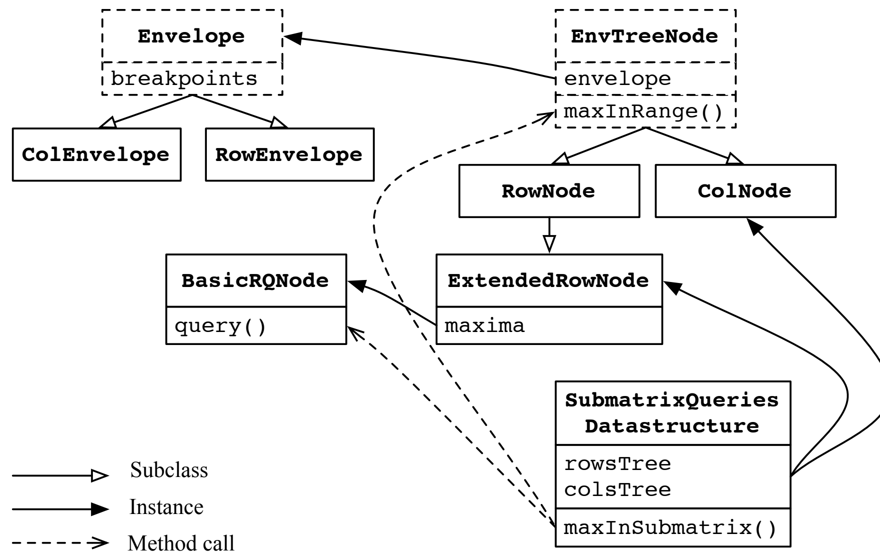


Figure 3: Graphic representation of the core code

2.2.3 Different implementations of the queries

While we stuck to the algorithm as described above, we had some liberty for the implementation of some operations. For instance concerning the canonical nodes: our first implementation explicitly created the set of canonical nodes for a query, and then enumerated the set to get the maximum; our final implementation recursively traversed the envelope tree to locate the canonical nodes in a depth-first search and update the maximum value once a canonical node has been found. We implemented this subtlety for both the queries on one row (or column) and a submatrix. They are referred in the sequel as the *explicit nodes* and *implicit nodes* implementations.

Furthermore, for the single row (or column) queries, we improved the implicit node implementation by noticing that we can avoid to search for some breakpoints in some conditions. Given a column c, suppose we found the breakpoint b located just before c in the envelope of node u. Then, for one of the children of u, we do not have to search for the breakpoint located before c in its envelope: suppose indeed b is in the part of u's envelope which is before the crossing breakpoint, *i.e.* the part of the envelope corresponding to the low indices rows and that is an exact copy of a prefix of the envelope of the low indices child u^- of u; then, b is also the breakpoint located just before c is u^-'s envelope.

This remark does not change the overall complexity of the algorithm but saves us a big number of binary searches among the envelopes (half of them in average).

We called this implementation the *simple cascading* implementation[7].

2.3 Testing the code

Once the code was written, we wanted to make sure that the algorithm is correctly implemented and then to do some benchmarks to get an idea of the implementation performances. We will present here the methods we used to check for the validity of the code.

2.3.1 Sampling Monge matrices

Because we did not want to make tests on too specific classes of matrices that could have hidden some bugs in the data structure, we imagined a way to generate random Monge matrices. Moreover, we tried to minimize the memory required to use these matrices in order to run tests on (very) big matrices.

Let $\{s_i\}_{i=1}^m$ be a set of sorted elements of $\mathbb{R}$ ($\forall i < j,\ s_i \leq s_j$). Let $M = (m_{i,j})$ the $m \times n$ matrix defined

$$m_{i,j} = j.s_i - i$$

Then M is an inverse Monge matrix: indeed $\forall i < j$ and $k < l$, we have

$$\begin{aligned} M_{ik} + M_{jl} \geq M_{il} + M_{jk} &\Leftrightarrow M_{ik} - M_{il} \geq M_{jk} - M_{jl} \\ &\Leftrightarrow (k-l).s_i \geq (k-l).s_j \\ &\Leftrightarrow s_i \leq s_j \end{aligned}$$

The idea is to take the values of the matrices along lines (real lines, not pseudo-lines) with a slope that increases with the index of the row (*cf.* Figure 4) To have a good distribution of slopes, instead of taking them uniformly in a big range, we pick angle α uniformly in the open interval $]-\frac{\pi}{2}, \frac{\pi}{2}[$ and set the slope to $\tan(\alpha)$.

It turns out that these matrices generate a *good* envelope tree for the rows but not for the columns: we have a lot of breakpoints in the envelopes corresponding to the nodes of the row tree but only one or two for the nodes of the column tree. To work around that issue, we just have to notice that, by considered the

[7] We used this name because in reference to Chazelle's and Guibas' *fractional cascading*, *cf.* [CG86]

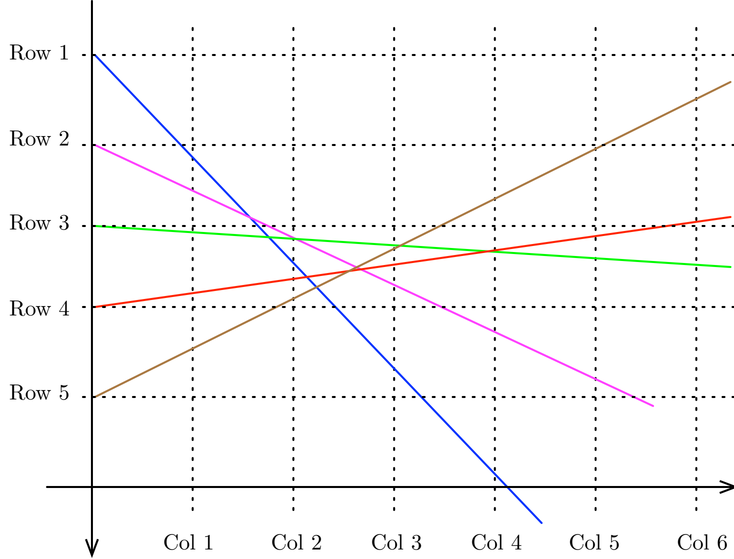


Figure 4: Generating inverse Monge matrices

transposition, we can construct a matrix whose envelopes for the column tree are now good and that the sum of two inverse Monge matrices is an inverse Monge matrix.

So, to summarize, we generate two ordered lists $\{\theta_i\}_{i=1}^m$ and $\{\theta'_j\}_{j=1}^n$ of elements picked uniformly at random in $]-\frac{\pi}{2}, \frac{\pi}{2}[$ and we set $m_{i,j}$ to

$$m_{i,j} = j.\tan(\theta_i) - i + i.\tan(\theta'_j) - j = j(\tan(\theta_i) - 1) + i(\tan(\theta'_j) - 1)$$

2.3.2 Testing

To check the validity of the implementation, we just coded the naive algorithm that compares all the values in the queried column/submatrix and returns the maximum. Then we checked whether its results were consistent with the algorithm's implementation on random inverse Monge matrices and random queries.

3 Benchmarking and practical considerations

3.1 Benchmarking protocol

For all our benchmarks, we timed the code we wanted to benchmark on a large number of random queries and on several sample matrices to get an average running time of our implementation.

Our goal was first to find the fastest implementation of our algorithms and then to determine the average complexity of the algorithms we described before (we already have the worst case complexity thanks to the analysis but not the average case complexity). We used `Gnuplot` to generate our graphics and to fit the test data.

We ran our benchmarks on square matrices. Accordingly, in this section all the matrices we consider are square matrices whose size is n. A very important parameter in the complexity is also the number of breakpoints in the envelopes: as we are doing binary searches on the envelopes and as the complexity of the binary searches is one of the bottleneck of the queries (with the search of canonical nodes), it is really important that the upper envelope of k rows (or columns) contains $O(n^c)$ breakpoints for a constant c (we know that $c \leq 1$, *cf.* 1.1.2) so the binary searches still take $\Theta(\log n)$ time.

Figure 5 shows that the matrices generated using the algorithm described in 2.3.1 are good for our benchmarks because, as we can see, the size of the upper envelope for a matrix of size n is $O(n^c)$ with $c = 0.35$.

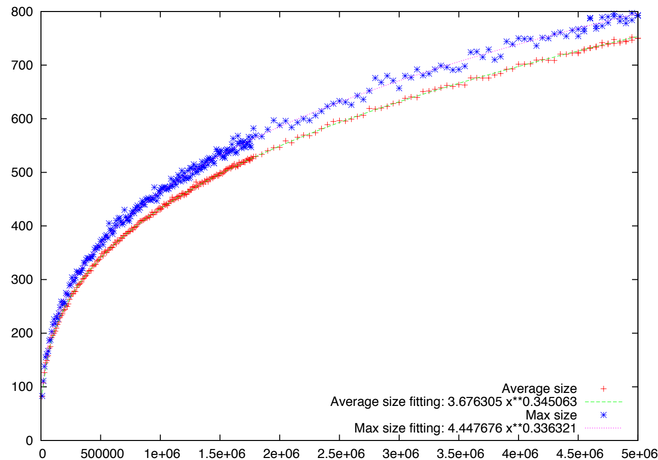


Figure 5: Statistics of envelopes sizes for sample inverse Monge matrices

3.2 Comparing implementations

Were compare here the different implementations of the one row (or one column) maximum queries - the *position* queries - and the submatrix maximum queries. For small matrix sizes, we also show the timing datas for the naive algorithms.

For these benchmarks, for n, we sampled 50 inverse Monge matrices of size $n \times n$ and we ran on each sample 1000 times each position query implementation and 100 times each submatrix query implementation with random query parameters. For small sizes ($n \leq 5000$), we increased n by steps of 50, by steps of 500 for big sizes ($n \leq 50000$).

3.2.1 Position queries

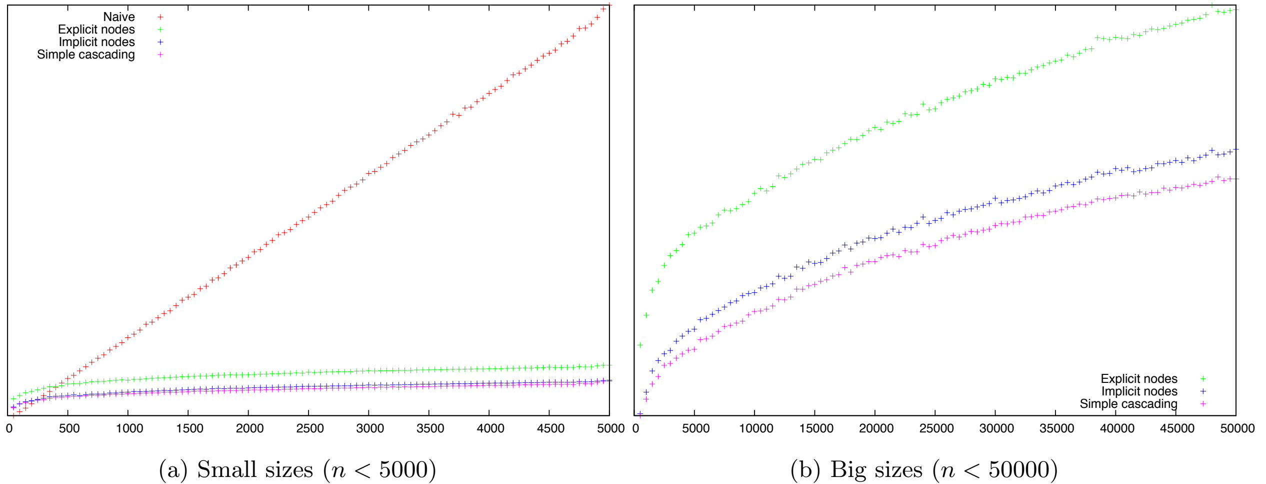
Figure 6 shows a big speed up with our algorithm compared to the naive algorithm. Moreover, it shows that the fastest implementation is the “implicit nodes” implementation. We can clearly see in (a) that we moved from a linear running time to a sublinear one.

(b) tells us which of the implementation is the fastest. If the overall complexity is similar of all three, the implementation that explicitly gets the canonical nodes is really less efficient than the “implicit” implementations. Finally, doing the simple cascading saves some running time without changing the complexity, exactly as expected.

3.2.2 Submatrix queries

For submatrix maximum queries, the speedup brought by our algorithm is huge: again we see sublinear complexity for the fast queries and polynomial complexity for the naive algorithm.

As for position queries, the implicit nodes implementation is faster than the explicit nodes implementation.



(a) Small sizes ($n < 5000$)

(b) Big sizes ($n < 50000$)

Figure 6: Benchmarks of position queries

3.3 Benchmarks and theory

Now that we know that our sampling method is valid and which of the implementations is the fastest, we can try to find the average running time of the algorithm.

For these benchmarks, for n, we sampled 20 inverse Monge matrices of size $n \times n$ and we ran on each sample 1000 times the simple cascading position query implementation and 100 times the implicit nodes submatrix query implementation with random query parameters. For $0 < n \leq 5.10^4$, we increased n by steps of 500, for $5.10^4 < n \leq 1.10^5$ by steps of 5000 and for $5.5\ .10^5 < n \leq 5.10^6$ by steps of 5.10^4. It gives us 208 measures for both position and submatrix queries.

Figure 8 shows that the performances of the implementation are consistent with the complexity given by the analysis of the algorithm in [KMNS12] *i.e.* $O(\log^2 n)$ for position queries and $O(\log^3 n)$ for submatrix queries.

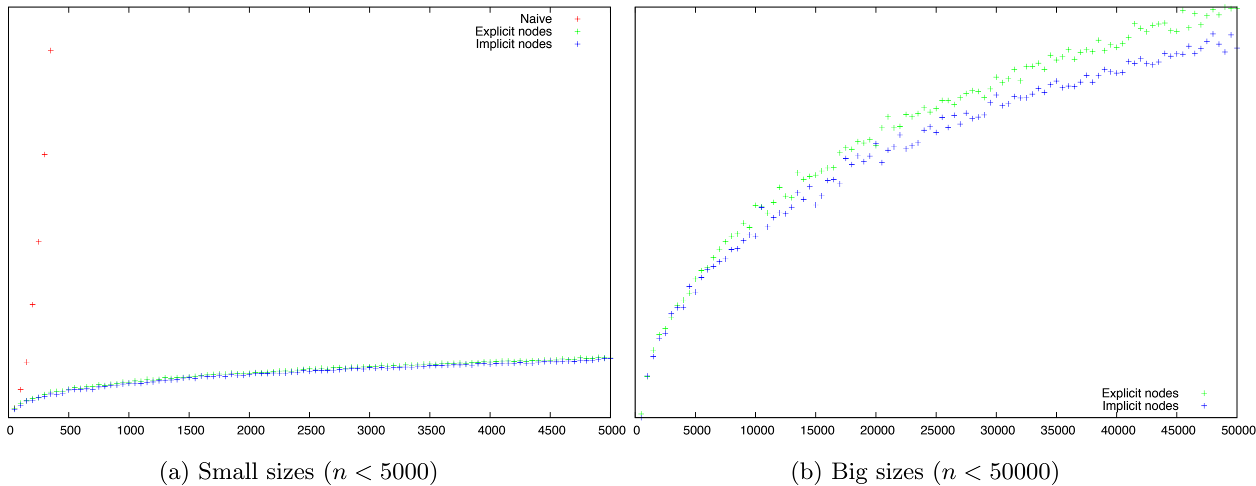
4 Possible implementation improvements

4.1 Queries

In [KMNS12], the fractional cascading technique is used to remove a logarithmic factor in the time complexity of the queries. This technique, described in [CG86], allows us to do a single binary search on the root of the envelope tree in order to find the breakpoints located just before a column in each node of the tree in constant time, and this by multiplying only by 2 the number of records stored in the tree's nodes.

If this technique should bring a theoretical improvement, we however decided not to implement it for two reasons: firstly because we saw our *simple cascading* technique only brought a small practical improvement to our implementation and thus we thought that the gain in time will not be substantial with respect to the additional complexity of the code; secondly because the algorithm's memory usage is already big and we didn't want to increase it.

An other possible way to improve the submatrix queries performances would be to use a range maximum query which has a constant amortized query time with linear space complexity (compared to our RMQ that has a $O(\log n)$ amortized query time and linear space complexity). We also do not think this would be an interesting optimization since the search in the RMQ is not a bottleneck, and, when we profile the code, the time spent querying the RMQ is negligible compared with the time spent by position queries.



(a) Small sizes ($n < 5000$)

(b) Big sizes ($n < 50000$)

Figure 7: Benchmarks of submatrix queries

4.2 Preprocessing

When we profiled the preprocessing step, we found that the most time-consuming step was computing the interval maxima and, in that routine, the position queries to the flipped tree was the bottleneck. So if we can improve significantly the speed of those queries, we should be able to improve the data structure initialization running time too.

An other tip we can follow to improve the performances would be using multithreading: when creating the data structure, we can construct the two envelope trees in parallel and then construct the interval maxima data structure for the nodes of the row tree. We can even do this in parallel too (at least partly). If this will not change the complexity of the algorithm, it would however be interesting for a practical use when dealing with big matrices.[8]

[8]We did not comment on this approach in the previous section because the time necessary for a single request is already very small – even for big matrices – and setting a multithreading environment will undoubtedly increase the running time.

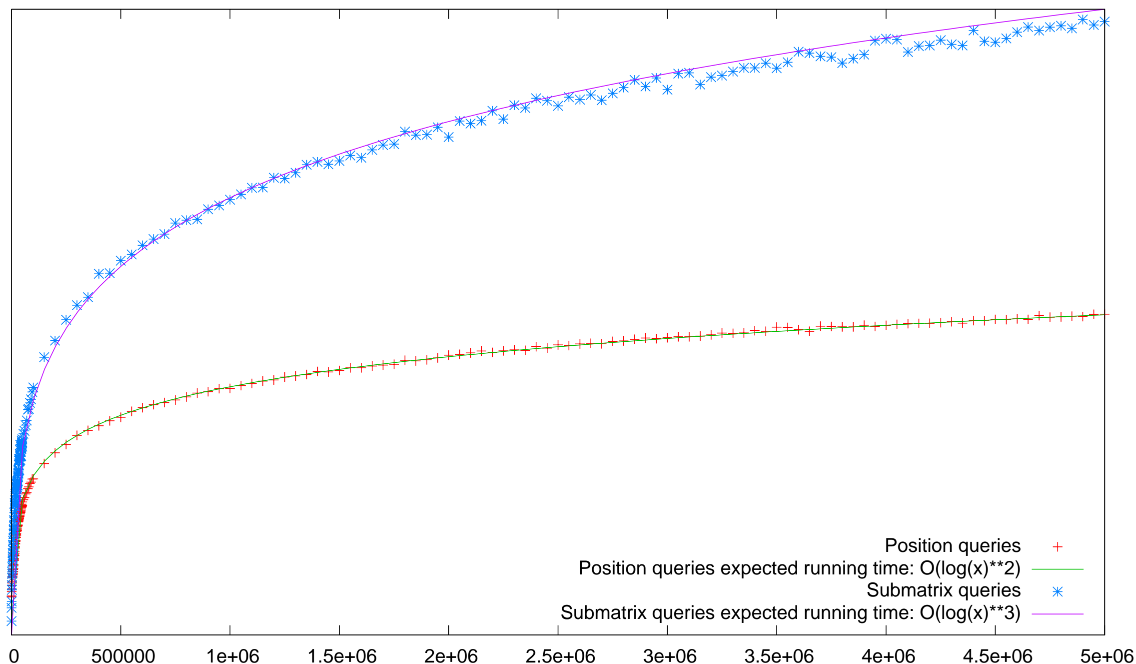


Figure 8: Average running time of maximum queries

Acknowledgements

The author would like to thank Shay Mozes for his very valuable assistance and encouragement throughout the development of this implementation and professor Philip Klein for his suggestions and his help.

References

[BKM+11] Glencora Borradaile, Philip N Klein, Shay Mozes, Yahav Nussbaum, and Christian Wulff-Nilsen. Multiple-source multiple-sink maximum flow in directed planar graphs in near-linear time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 170–179. IEEE, 2011.

[CG86] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986.

[FR01] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*, pages 232–241. IEEE, 2001.

[KMNS12] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and Monge partial matrices, and their applications. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 338–355. SIAM, 2012.

A Getting and compiling the code

The code is available on GitHub and can be copied using `git`:

```
git clone git://github.com/rbost/SubmatrixQueries.git
```

Once downloaded, you can compile the code using `SCons`[9] just by running `scons` on the command line in the directory where you put the sources.

You can choose the compiler you wish to use to build the source using the building script: `gcc` is used by default, run `scons compiler=clang` to compile the sources using `clang` and `scons compiler=icc` if you want to use the Intel C compiler. These compilers need to be installed and put in the `PATH` to be used.

B Using the command line tool

Compiling the code produces a command line tool called `test_queries`, which can run several tests and benchmarks.

`./test_queries` m n [`--mode` i][`--minSize` minM minN] [`--step` s] [`--samples-per-size` nSamples] [`-v` | `--verbose`] [`--naive`] [`-o` fileName]

m n : The size of the matrices to test/benchmark

`--mode` i : Set the running mode of the testing utility:

0. Benchmarks the initialization of the data structure for a m×n matrix.
1. Runs the algorithm validity tests on a sample m×n matrix.
2. Benchmarks all the position queries implementations.
3. (default) Benchmarks all the submatrix queries implementations.
4. Benchmarks the fastest implementation of the position queries and the fastest implementation of submatrix queries.
5. Benchmark all the queries implementations for a single sample matrix of size m×n.
6. Benchmark all the queries implementations.
7. Generate statistics on the size of the upper envelopes.

`--minSize` minM minN : Set the minimal size of matrices to benchmark to minM×minN (0×0 by default).

`--step` s: Set the step to go from the minimal size to the maximal size when benchmarking to s (50 by default).

`--samples-per-size` nSamples: Set the number of sample matrices to nSamples for every benchmark size (50 by default).

`-v` | `--verbose`: Verbose mode.

`--naive`: Run benchmarks of the naive algorithm when possible.

`-o` fileName: Output the benchmarking data in the specified file for modes 3, 4, 6, and 7.

[9] `http://www.scons.org`

C Using the code in your software

If you wish to use the submatrix maximum queries for inverse Monge matrices, you will need to put the files `debug_assert.h, envelope_tree.h, envelope.h, matrix.h, max_value.h, range_query.h, range.h` with your sources.

To provide access to your datas, you can either implement your own concrete subclass of `Matrix` that will act as an *interaction layer* between the data structure and your data, or use the already implemented concrete subclasses `SimpleMatrix` or `ComplexMatrix`. We discourage the use of these classes for non testing purposes as they create a copy of the data they are given as input for initialization.

Once you have your matrix, you can create an envelope tree over the rows, an envelope tree over the columns or the entire submatrix queries data structure. To query an envelope tree, just call `fastestMaxInRange()` with the column (or row) and range of your query. To query the submatrix queries data structure, call `fastestMaxInSubmatrix()` with the query ranges.

D Benchmarks

You can find the original PDF figures for the benchmarks, the data sets and the `gnuplot` files that were used for this document at the following URL: `http://cs.brown.edu/~bost/Files/SubmatrixQueries/Benchmarks/`