A Fast Implementation of FR-Dijkstra

Ryan Lester
Department of Computer Science
Brown University
Providence, RI
ryan_d_lester@brown.edu

1 Introduction

The shortest path problem is the problem of finding the shortest distance from a specified source node to all other nodes in the graph. The best known algorithm for arbitrary graphs with real-valued weights is known as Bellman-Ford, and runs in $O(nm)$ time, where m and n are the number of vertices and nodes in the graph, respectively. For an arbitrary graph with non-negative weights, the best known algorithm is Dijkstra's algorithm, which runs in $O(m + n \log n)$ time.

However, better algorithms exist for special subclasses of graphs. The most important subclass is planar graphs. A planar graph is a graph with an embedding where no edges cross, or more formally, where the Euler characteristic of the graph is equal to $2k$, where k is the number of connected components. This paper looks at an algorithm known as FR-Dijkstra [2] that takes advantage of the non-crossing property of planar graphs to efficiently solve the distance problem. The algorithm can find the distance between two nodes of a graph in time $O(n \log^2 n / \log \log n)$. Alternatively, the core data structure used in FR-Dijkstra can be used as a distance oracle. This oracle can be contructed in time $O(n \log^2 n / \log \log n)$, can answer distance queries or change edge weights in time $O(n^{2/3} \log^{7/3} n)$ and requires space $O(n \log n)$.

The key insight of the paper is that the non-crossing property of planar graphs is similar to the Monge property of matrices. While these two properties are not directly related, there exists a decomposition of planar graphs that produces Monge matrices. Algorithms for quickly searching Monge matrices can then be applied to speed up the search for the shortest path.

In this paper, we present a new, efficient implementation of the core data structure, the Monge heap, used in FR-Dijkstra.

2 Algorithm

2.1 Online Monge searching

Unlike many approaches to describing FR-Dijkstra, the presentation here operates in reverse order. We start with the Monge property and the solution to the "online Monge matching problem", and then build to their application in solving the shortest path problem.

A matrix M is a Monge matrix if for every pair of rows $i < j$ and pair of columns $k < l$, $M_{ik} + M_{jl} \leq M_{il} + M_{jk}$, and is an inverse Monge matrix if the reverse condition holds. The property was initially recognized in geometry, where the property holds for computing distances between two ordered sets of points A and B on the left and right sides of an axis-aligned rectangle. Intuition for the property comes from the fact that, for $i < j \in A$ and $k < l \in B$ any path from point i to point l that lies entirely inside the rectangle must cross any path from j to k that also lies inside the rectangle. The Monge property can be derived from this fact with the use of the triangle inequality. The most famous algorithm taking advantage of this property is the SMAWK algorithm [1], which can find the minimum or maximum value of all rows of an mxn Monge matrix in time $O(m + n)$.

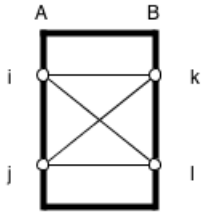


Figure 1: An illustration of the Monge property. Note that the paths between i and l and j and k must either cross one another or the boundary of the rectangle.

An oracle for answering submatrix maximum or minimum queries in Monge matrices also exists, and can be constructed in time $O(n \log n)$, queried in time $O(\log^2 n)$ and requires $O(n \log n)$ space [4].

The Monge matching problem, given a complete ordered bipartite graph $G = (A, B, E)$ with edge weights d satisfying the Monge property and offset distances D for the nodes in A, is to find a parent $p(v) \in A$ for all $v \in B$ that minimizes $D(p(v)) + d(p(v), v)$. If $|A| = |B| = n$, then SMAWK can be used to solve this problem in time $O(n)$. In the online version of this problem, the offset distances are initially set to ∞, then updated ("activated") once online. Additionally, nodes $v \in B$ are "extracted" online and added to a subset $S \subseteq B$; intuitively, these nodes can be thought of as ones for which the correct best match has already been found. and the updated set of best parents for all $v \in B$ must be found after each update of the offset distances. The problem is now to maintain a parent $p(v) \in A$ for all $v \in B \backslash S$ over all activations and extractions. Once again, SMAWK can be used to solve this problem, but because the entire algorithm has to be re-run after each activation or extraction, each activation or extraction takes $O(n)$ time, and since there are at most $2n$ activations and extractions, total running time is $O(n^2)$.

A better solution is suggested by two properties of the problem that follow from the Monge property:

1. The set of nodes in B for which the same node $u \in A$ is the parent must be consecutive in B.
2. If node a' is after a in A, then the set of nodes for which a' is the parent must also be after the set of nodes for which a is the parent in B.

From these, we arrive at the bipartite Monge Subheap data structure. This data structure supports three operations:

- ActivateLeft(u, d_u): for $u \in A$, set $D(u) = d_u$.
- FindMin(): return a node $v \in B \backslash S$ such that $v = \arg\min_{v \in B \backslash S} \min_{u \in A} D(u) + d(u, v)$.
- ExtractMin(): let $v =$ FindMin(), add v to S, and return v.

ActivateLeft and ExtractMin represent the activations and extractions discussed above; FindMin is not directly of use in solving the online Monge matching problem but will be of use later. The data structure maintains the current offset distance for all $u \in A$ and the subset $S \subseteq B$. Internally, the data structure builds a submatrix maximum oracle to represent d, and maintains:

- For all $a \in A$, a bit indicating if the node has been activated.
- A balanced binary tree N containing, for all activated $a \in A$, tuples of the form (a, b_{min}, b_{max}), where b_{min} and b_{max} are the range of nodes in B for which a is currently the best parent; these tuples are ordered lexicographically.
- A min-heap H containing, for all $(a, b_{min}, b_{max}) \in N$, the edge (a, b) for b between b_{min} and b_{max}, inclusive, with minimum distance.

While the ranges in N initially cover all nodes in B, this should no longer be the case when a node is extracted. As such, a can appear more than once in N, but the tuples containing a cannot contain overlapping b_{min} and b_{max}.

The operations are implemented as follows:

- FindMin(): Return the edge at the root of H.
- ExtractMin(): Pop the edge (a, b) off H and mark it as removed. Then, search N for the tuple (a, b_{min}, b_{max}) where b is between b_{min} and b_{max}. Let b_- and b_+ be the nodes immediately preceding and following b, respectively. We now remove (a, b_{min}, b_{max}) from N and replace it with two new tuples, (a, b_{min}, b_-) and (a, b_+, b_{max}), in a sense splitting the old tuple in half at b. For each of these, we find a new edge (a, b') lying in the associated range and insert it into H. Finally, we return (a, b).
- ActivateLeft(u, d_u): We begin by setting $D(u) = d_u$.

  Next, we need to find the new children of u in B. If u is the first node activated in A, then this is trivial; we can simply set all nodes in B as a child of u and update N and H accordingly. Otherwise, we need to find if the parent of any nodes in B will switch to u. To do so, we can take advantage of the properties mentioned above and check the children of the nodes immediately preceeding u, continuing to iterate downwards until we find a node in B that does not switch parents. We then do the same for the nodes following u. We here describe the procedure for searching the nodes preceeding u; the procedure for the following nodes is similar.

  First, we search the tuples preceeding u in N. We start by searching N for the last tuple (v, w_{min}, w_{max}) such that v preceeds u in A. If such a tuple does not exist, u has no children currently associated with nodes preceeding it in A, and we are done searching above. Otherwise, we traverse the tuples in T backwards until we run out of tuples. We check if $D(a') + d(a', b'_{min}) > D(u) + d(u, b'_{min})$. If this is true, then u can steal all the children of a', and we continue our traversal unless the node in B immediately preceeding b'_{min} has already been extracted. Otherwise, we next check if $D(a') + d(a', b'_{min}) > D(u) + d(u, b'_{min})$. If this is true, then the first node in B whose parent changes to u lies between b'_{min} and b'_{max}, and we can find it with a binary search. Otherwise, if the current tuple is not (v, w_{min}, w_{max}), the first node in B whose parent changes to u is the first node in the previous tuple in the iteration. If the current tuple is (v, w_{min}, w_{max}), then none of the nodes in tuples whose parents preceed u in A switch parents.

  Let u_{min} and u_{max} be the first and last nodes in B who switch parents to u. We remove from N all tuples containing nodes between u_{min} and u_{max}, and remove from H the edges contributed by the removed tuples. If u_{min} is not the first node in B in the tuple containing it, we re-add the same tuple containing it, but with b_{max} set to the node immediately preceeding u_{min}, and update H accordingly. Similarly, if u_{max} is not the last node in B in the tuple containing it, we re-add the same tuple containing it but with b_{min} set to the node immediately following u_{max}, and update H accordingly. Finally, we add the new tuple (u, u_{min}, u_{max}) to N and update H.

FindMin() takes $O(1)$ time. Because both ExtractMin() and ActivateLeft(u, d_u) both insert a constant number of tuples into T and H, these both take $O(\log n)$ time. Therefore, run time of activation and extraction of all nodes in A and B takes time $O(n \log n)$.

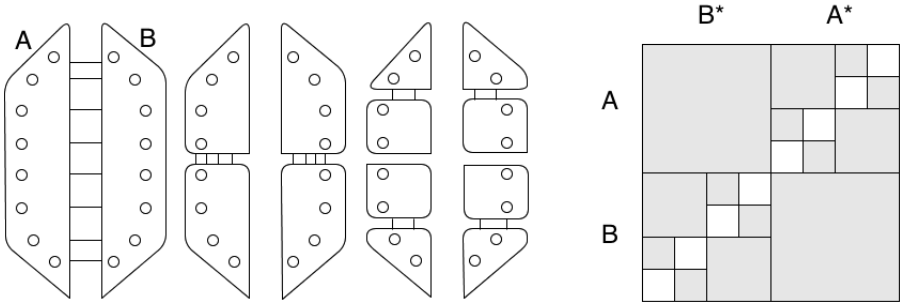
2.2 The Dense Distance Graph and Monge heaps

As mentioned above, the distances in a planar graph do not obey the Monge property, and planar graphs are most certainly not complete bipartite graphs. So then what does the online Monge matching problem have to do with computing distances? To understand this, we will need to examine two new data structures, the dense distance graph and the Monge heap.

First, we assume wlog that the planar graphs examined are simple, embedded in the plane, and not containing negative weight cycles. The first two can hold for all planar graphs with some amount of preprocessing required to convert an arbitrary planar graph into the correct form. If the latter is not true, the shortest path algorithm will detect it and abort.

Let n be the number of nodes in G. A piece of G is an edge induced subgraph. For some parameter $r < n$, an r-division is a partition of E into $O(n/r)$ pieces such that each piece contains $O(r)$ nodes, with nodes belonging to more than one piece identified as a boundary node, and each piece has $O(\sqrt{r})$ boundary nodes. Such a partition can be computed in $O(n \log r + \frac{n}{\sqrt{r}} \log n)$ time [3], using the planar separator theorem [5]. To simplify our presentation of the algorithm, we will assume that all pieces are connected.

To construct the dense distance graph (DDG) of a graph G, we start by computing an r-division of G. Then, for each piece P, for each pair of boundary nodes u, v, we add an edge (u, v) to a new graph G', and set $d(u, v)$ to the shortest distance between the two within P. The resulting graph, a union of the cliques formed by the boundaries of the pieces, is a dense distance graph.



(a) Graph decomposition (b) Matrix decomposition

Figure 2: Various steps in the decomposition of the boundary nodes into Monge arrays. At each step, the existing pieces (starting with the full set of nodes) are divided into two contiguous pieces; the full set of edges between the two pieces (illustrated here with only a small subset drawn) obey the Monge property. Note that the column entries of the corresponding distance matrix run in reverse order of the rows, meaning row i corresponds to the same node as $n - 1$; this is necessary for the Monge property to hold.

The full set of edges in the DDG do not obey the Monge property, as paths between some arbitrary nodes i, j, k, l do not necessarily cross. However, we can decompose the edges in a single piece of the DDG to form subgraphs that do obey the non-crossing property. To do so, we divide the full set of boundary nodes into two contiguous subsets; the resulting bipartite graph formed by the edges between (but not within) the two subsets obey the non-crossing property and the Monge property. We can recursively subdivide each subset until there is only a single node in a subset; the edges crossing each division all form bipartite graphs whose distances obey the Monge property. In total, we construct $O(\log r)$ Monge arrays per piece.

We can apply the bipartite Monge subheap to these subgraphs, suggesting a shortest path algorithm that, upon reaching a vertex u, activates u with its distance from the source in all the bipartite Monge subheaps containing u, then extracts a new minimum edge to find the next node to activate. To manage all the bipartite Monge subheaps in a single piece, we use a Monge heap data structure. This data structure contains all the Monge subheaps corresponding to an individual piece, along with a min-heap H_P that stores the minima of the min-heaps H of the bipartite Monge subheaps. This data structure, like the subheap, supports three operations:

- FindMinInPiece(): Return the minimum in H_P.
- ExtractMinInPiece(): Pop the minimum from H_P, and call ExtractMin() on the bipartite Monge subheap to which the minimum belonged. Then, update H_P with the new minimum of that subheap.
- ActivateInPiece(u, d_u): Call ActivateLeft on all of the $O(\log r)$ bipartite Monge subheaps containing u as an element in A. Afterwards, check if the minimum of each subheap called has changes; if it has, update H_P with the new edge.

FindMinInPiece() takes $O(1)$ time. Since ExtractMinInPiece only calls ExtractMin() on a single subheap, it takes $O(\log r)$ time. ActivateInPiece(u, d_u), on the other hand, has to call ActivateLeft(u, d_u) $O(\log r)$ times, so the total amortized running time is $O(\log^2 r)$.

2.3 FR-Dijkstra

We now have all the data structures required to build the FR-Dijkstra distance oracle. To construct the oracle, we compute the r-division and DDG of the input graph G, and build Monge heaps for

each piece in the DDG. This takes $O(n \log n)$ time, if r is chosen to be $n^{2/3} \log^{2/3} n$. To answer a query for the distance between two nodes s and t, we perform the following algorithm:

1. Let P_s and P_t be the piece of the r-division of G containing s and t respectively. We begin the algorithm by finding the distance from s to all the nodes in P_s using Dijkstra's. If $P_s = P_t$, we will find $d_P(s,t)$ during this step, which is not necessarily the same as $d(s,t)$
2. We now run a modified version of Dijkstra's on the DDG, using the Monge heaps to quickly relax edges. To do so, we maintain a global heap H_G, which contains the minimum edges of the Monge heaps. To initialize the heap, however, we insert, for all boundary nodes of P_s, the edges (s,u), with distance label $d(s,u)$. We also keep track of which boundary nodes have been visited, since a boundary node appears in multiple pieces and Monge heaps and could therefore appear multiple times in H_G. We use this record to prevent travelling to the same node multiple times.

   In each iteration of the modified Dijkstra's, we pop the minimum element (u,v) off H_G. If this heap entry was contributed by a Monge heap (ie. $u \neq s$), we call ExtractMinInPiece() on the corresponding Monge heap and insert the new minimum into H_G. If we have already visited v, we continue to the next iteration. Otherwise, we mark v as visited and set $d(s,v)$ to the value of $D(u) + d(u,v)$ in the corresponding Monge heap (note that $D(s) = 0$). For every piece that contains v, we now call ActivateInPiece(v, d(s,v)) on the corresponding Monge heap, and update the entries the Monge heap contribute to H_G if its minimum has changed or it is the first time a node in the Monge heap has been activated.

   We terminate this step once all boundary nodes u of P_t have been visited. If t is a boundary node of P_t, then we are finished.
3. Finally, we compute the distance from the boundary nodes of P_t to t inside P_t. To do so, we create an artificial source s' connected to each boundary node u with distances $d(s',u) = d(s,u)$, and run Dijkstra's on the resulting graph to obtain a final distance $d_{DDG}(s,t)$. We return this unless $P_s = P_t$, in which case we return $\min d_P(s,t), d_{DDG}(s,t)$.

The first and the last steps are simply runs of Dijkstra's on subgraphs of size r, so both steps run in time $O(r \log r)$. The middle step takes time $O((n/\sqrt{r}) \log n \log r)$. This follows from the fact that, since there are $O(n/r)$ pieces and $O(n/\sqrt{r})$ total boundary nodes, the maximum number of elements in H_G is $O(n/\sqrt{r})$. Therefore, pop and insertion take $O(\log(n/\sqrt{r})) = O(\log n)$ time. Since each boundary node may be inserted into H_G $O(\log r)$ times, the total time spent manipulating H_G is $O((n/\sqrt{r}) \log r \log n)$. This dominates the $O((n/\sqrt{r}) \log^2 r)$ time spent on calls to ExtractMinInPiece and ActivateInPiece. Therefore, the overall running time is $O((n/\sqrt{r}) \log r \log n)$, which gives $O(n^{2/3} \log^{5/3} n)$ for our choice of $r = n^{2/3} \log^{2/3} n$.

3 Implementation

While the first and last steps of the above algorithm can be run with existing implementations of Dijkstra's, no implementation exists of the second step, which requires the Monge heap and subheap data structures, and the modified version of Dijkstra's that works with the dense distance graph. In this paper, we present an novel implementation of the data structures that, provided with a dense distance graph, can build the required Monge heaps and subheaps and run the modified Dijkstra's algorithm.

Our implementation is written in C++, and designed for high performance. To implement the Monge subheap, we build off of Raphael Bost's implementation of the Monge submatrix query data structure, modified to support several additional features necessary to our implementation, such as finding the location of the maximum value in addition to the value itself, and support for building the data structure over Monge slices of a larger non-Monge matrix. We use this as our range-search data structure. For the heap H, we also use a min-Heap implementation by David Eisenstat, again modified to support fast location and removal of items given the value (but not the key) of the item in advance. This allows us to augment the tree N of tuples with the corresponding items in the heap H, enabling fast calls to ExtractMin().

Our Monge heap takes a single piece of the dense distance graph as input and manages its decomposition into Monge subpieces, for each of which a Monge subheap is created. Our main Oracle

class takes a dense distance graph as input and constructs a Monge heap for each piece of the graph, and can then be queried for distances between two nodes. To use the Oracle for a single run of FR-Dijkstra, the Oracle simply needs to be constructed, queried once in step two of the algorithm, then destroyed.

Although our implementation of the Oracle is not yet completed, we have run an initial set of benchmarks for our implementation of Monge heaps. These consisted of building the Monge heap with a dense distance graph containing N boundary nodes, then running N ActivateInPiece and N ExtractMin operations on the heap. Because each piece of an r-divison has $\sqrt{r}$ boundary nodes, our benchmark for $N = 16380$ boundary nodes corresponds to the case where there are on the order of $10^1 1$ nodes total for our choice of r. The times for buliding the data structure and querying it are reported separately; both are averaged over 5 runs. The benchmarks were run on a Macbook Pro with an Intel Core i7 2.3 GHz CPU and 8 GB of memory. The results in Table 1 suggests our data structure runs quickly and scales well. Constructing the Monge heaps took the vast bulk of the time, with a full set of queries always completing in under a tenth of a second. This suggests the usefulness of our data structure as a distance oracle, although we cannot truly analyze its use in computing distances until the implementation of the Oracle is completed.

Table 1. Benchmarks for Monge heaps

# of nodes	Build Time (ms)	Query Time (ms)
252	5.4586	0.4158
508	14.5558	0.9962
1020	43.7896	2.2692
2044	140.488	4.9726
4092	492.465	10.9832
8188	1810.46	24.522
16380	6934.1	53.458

Once the Oracle implementation is finished, our code could easily work with any graph library capable of computing r-divisions and dense distance graphs. A future project is to implement those algorithms in Philip Klein's "planarity" library for planar graphs.

4 Acknowledgements

I would like to thank Philip Klein for his advice, and for introducing me to the theory of planar graphs.

References

[1] Alok Aggarwal, MariaM. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

[2] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868 – 889, 2006. Special Issue on FOCS 2001.

[3] Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[4] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in monge matrices and monge partial matrices, and their applications. In *SODA*, pages 338–355, 2012.

[5] Richard J Lipton and Robert Endre Tarjan. Applications of a planar separator theorem. *SIAM journal on computing*, 9(3):615–627, 1980.