I/O-efficient shortest path algorithm for simple polygons

Gijs Kwaks ICA-4133080

November 2020

Frank Staals Maarten Löffler

Department of Information and Computing Sciences
Utrecht University

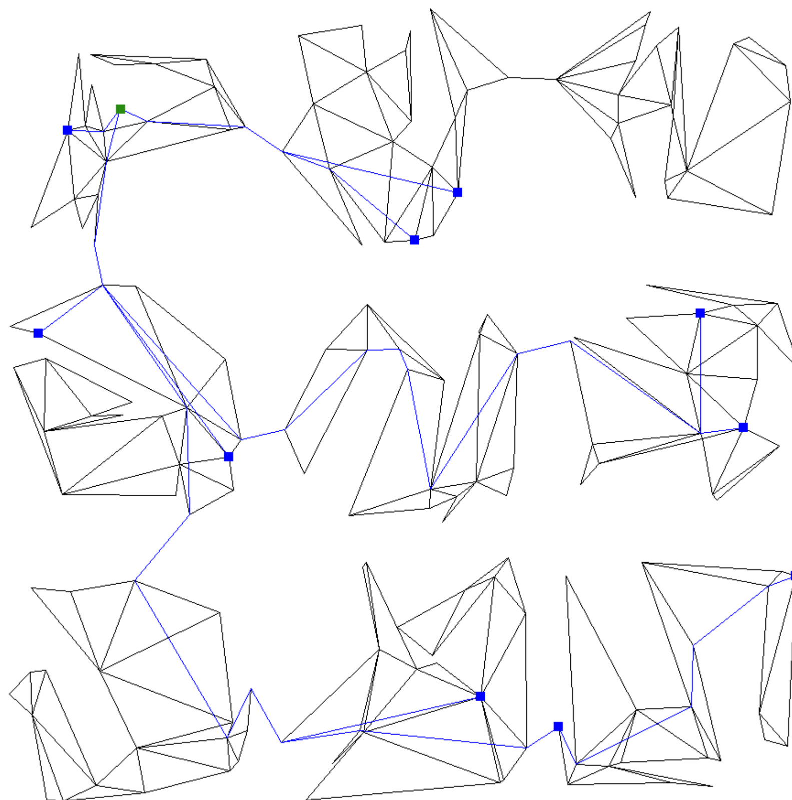


Figure 1: Some example shortest paths in a generated simple polygon

Contents

1 Introduction **1**

2 Internal shortest path algorithm **4**
2.1 Funnels 4
2.2 Shortest path algorithm, Hershberger, Snoeyink [1] 5

3 I/O-efficient preliminaries **8**
3.1 I/O-efficient stacks and deques 8
3.2 I/O-efficient tree traversal 9
3.3 I/O-efficient sorting 10

4 I/O-efficient shortest path algorithm **11**
4.1 I/O-efficient sparse shortest path tree 11
4.2 Divide and conquer 12

5 Implementation **14**
5.1 Internal funnel 15
5.2 I/O-efficient funnel 15
5.3 Internal shortest path tree 15
5.4 Sparse shortest path tree 16

6 Experiment **18**

7 Results **20**
7.1 Single shortest path experiment 20
7.2 (Sparse) shortest path tree experiment 21

8 Discussion **23**
8.1 Limitations 23
8.2 Evaluation of research questions 24

9 Conclusion **26**

1 Introduction

Recently a new I/O-efficient algorithm has been developed for computing shortest path problems in simple polygons [2]. In this thesis we explore the practical applicability of this algorithm. Normally, algorithms are optimized to use as few as possible CPU cycles to complete. However, when the data on which the algorithm operates does not fit into fast main memory the execution time of the algorithm will be dominated by the time it takes to transfer data between slow hard drives and fast main memory:

"Because mechanical movement is involved, the typical read or write time is on the order of milliseconds. By comparison, the typical transfer time of main memory is a few nanoseconds—a factor of 10^6 faster!" [3]

An I/O-efficient algorithm is an algorithm that is optimized to use as few as possible I/O operations. The standard model to analyse I/O-efficient algorithms is the I/O model introduced by Aggarwal and Vitter [4]. This model abstracts a computer in three components: a CPU, a limited amount of main memory and a conceptually infinite amount of disk space. The data on which I/O-efficient algorithms operate generally does not fit in the limited main memory. When data is requested that is not stored in the main memory it is transferred from the disk to the main memory in fixed-size blocks. Each such transfer is called an I/O operation or simply an I/O. The performance of an algorithm is measured in how many I/O operations performed. The following variables are used to quantify this measure.

N = amount of records;
M = records that can fit into internal memory;
B = records that can be transferred in a single block;
D = the number of disks, for simplicity this is usually kept at one disk of conceptually unlimited size;

Two common notations in the I/O model are the scanning and sorting of records. In the I/O model N records can be sequentially scanned in $O(\frac{N}{B})$ I/Os, denoted as $Scan(N)$. The optimal bound of sorting is $O(\frac{N\,log(N/B)}{B\,log(M/B)})$ I/Os, denoted as $Sort(N)$ [4].

A simple polygon P with N vertices is a polygon without any loops or holes and has a clear definition of what is inside and outside of the polygon. All the approaches assume the polygon is saved as a triangulation T. A triangulation is a way to divide a simple polygon into triangles without introducing new vertices, it can be calculated in $O(N)$ time as described by Chazelle [5]. The triangulation T is saved as a binary tree where each node is a triangle and each edge is a link to a neighbouring triangle, see figure 2. The I/O-efficient algorithm assumes T is saved in post order for I/O-efficient access. A shortest path is defined as the path that covers the smallest possible distance between two points while being

entirely within the polygon. Most approaches to solving shortest path problems pre-process the polygon in order to create a shortest path tree. A shortest path tree is a data structure containing every vertex of the polygon where each vertex has a reference to the next vertex on the shortest path towards a single vertex, the source vertex. This means it contains the shortest paths from every vertex in the polygon to a single starting point. The I/O-efficient approach uses a sparse shortest path tree which is a data structure that only contains the shortest paths from a limited number of vertices to the source vertex.

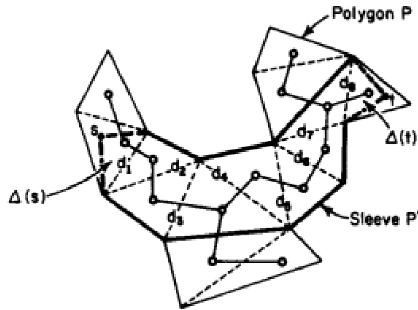


Figure 2: Triangulation of a simple polygon and its dual tree, taken from [6]

In order to determine whether the new approach of Agarwal et al. [2] has practical applicability it will be compared to a similar existing algorithm. Three other approaches will be considered by Guibas et al. [7], Hershberger and Snoeyink [1] and Guibas and Hershberger [8]. The approach of Guibas et al. pre-processes the polygon to create a shortest path tree in $O(N)$ time where N is the amount of vertices in the polygon. The approach makes use of a more complex data structure, the finger tree data structure [9]. The approach of Hershberger and Snoeyink is very similar to that of Guibas et al. It has the same algorithmic complexity of $O(N)$ and creates the same shortest path data structure but does not require the complicated finger tree data structure. The last approach by Guibas and Hershberger also has the same algorithmic complexity of $O(N)$ but results in a different data structure. Instead of a shortest path tree it splits the polygon into subpolygons and prepares data for these subpolygons to efficiently be able to query shortest paths. The major contribution of this approach is that it works for any source vertex. Where a shortest path tree only supports querying for a single source vertex this approach can query shortest paths between any two points within the polygon. The downside of the approach is that it uses a lot of extra and very complicated data structures in order to split the polygon in the subpolygons and to store the pre-processed data for these subpolygons. In this thesis we will be comparing the new approach of Agarwal et al. to that of Hershberger and Snoeyink because these are more similar and not overly complex to implement. Both the approaches result in a shortest path tree and only allow querying for one source vertex.

The algorithm proposed by Agarwal et al. [2] is a purely theoretical algo-

rithm. It is unclear whether it is also efficient in practice. In order to conclude whether it is efficient in practice we can answer the following three questions.

> *1: Does the theoretical algorithmic complexity of creating the sparse shortest path tree of the proposed I/O-efficient algorithm hold up in practice.*
>
> *2: Does the theoretical algorithmic complexity of creating the shortest path tree of competing state of the art internal memory algorithms hold up in practice.*
>
> *3: Does the I/O-efficient algorithm become faster than the internal memory algorithms for larger problems.*

In the research questions we mention testing the performance for creating the (sparse) shortest path tree. The experiment will be extended to also test the performance of calculating a single shortest path without any pre-processing for the I/O-efficient method and the internal memory method.

The algorithmic complexity of the proposed I/O-efficient algorithm is designed with large problems in mind, so the hypothesis is that this complexity will hold up in practice. We expect this is not the case for the internal memory variants. The expectations are that these algorithms will become significantly slower when they require more memory than there is RAM. The proposed algorithm will have practical value when it is faster than competing internal memory algorithms for certain sized problems. Usually I/O-efficient algorithms are slower than similar internal memory algorithms for small problems because the constant cost of operations is much higher and algorithmic complexity is often worse. However if the hypotheses for research questions 1 and 2 are correct we expect that for some problem size the I/O-efficient algorithm will become faster than its internal memory competitors.

Besides finding out how the proposed algorithm of Agarwal et al. behaves in practice this thesis makes another contribution by improving their approach slightly. As mentioned the approach uses a sparse shortest path tree which allows a limited number of shortest paths to be stored. The amount is limited by how much main memory is available. Agarwal et al. reach a limit of $O(\sqrt{\frac{M}{B}})$, the way we propose to implement it we can get to $O(\frac{M}{B})$ shortest paths.

First we will discuss the concepts relevant to shortest path algorithms and the approach of Hershberger and Snoeyink in section 2 and then some preliminary I/O-efficient concepts in section 3. In section 4 we will describe the I/O-efficient approach by Agarwal et al. Then in section 5 we will highlight implementation specific details of the approaches. The experiment specific details will be given in section 6 and then finally the results, discussion of the results and the conclusion will be given in section 7, 8 and 9.

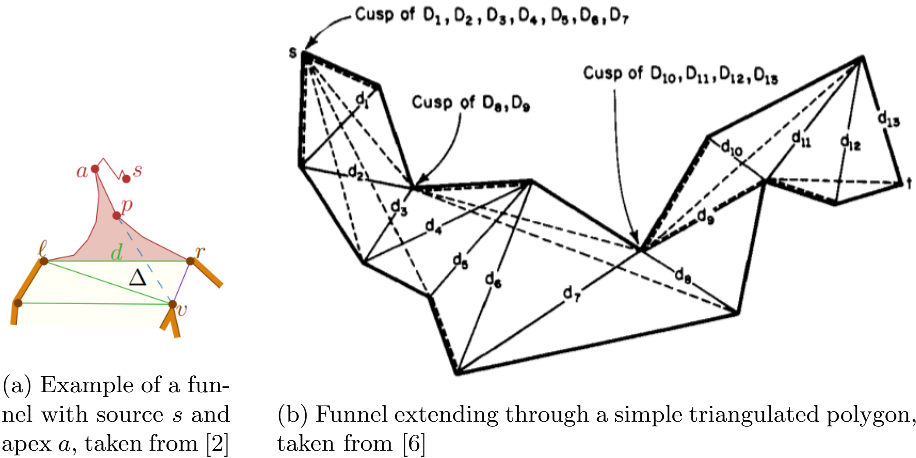
2 Internal shortest path algorithm

In this section we will discuss the concepts used by shortest path algorithms. Given a polygon P with N vertices, a triangulation T and a source vertex s the goal is to create a shortest path tree containing shortest paths from every vertex towards s. Firstly we will discuss funnels, the key concept used by all the shortest path algorithms. Then we will go into more detail about the shortest path algorithm proposed by Hershberger and Snoeyink [1].

2.1 Funnels

The concept of funnels is introduced by Lee and Preparata [6] and is used to find a shortest path, from a source point s to a target point t, using the triangles of T. The points s and t are both vertices of P. It is easy to extend funnels to support source and target points in the interior of P but this is not necessary to understand the concept of funnels. A funnel is a region inside P that starts at a vertex called the apex and two chains extending left and right from the apex. The left and right chain extend to both end points of a diagonal d that is an edge of a triangle in T. Initially the apex is s and the left and right chains will extend to the other two vertices of the triangle containing s. See figure 3a for an example of a funnel.

The initial case of a funnel, a single triangle, is very simple to understand. One of the vertices is s, the other two vertices are the end points of d. The left and right chains are the edges from s to the end points of d. A funnel can be extended with a triangle that shares edge d. Now d becomes an edge of the newly added triangle that is closer to t, it is assumed it is known which edge of the new triangle to choose, this is the responsibility of the shortest path algorithm and not of the funnel. Since d is updated, the left and right chain need to be updated, this is symmetrical for both end points. This is done by scanning the existing funnel for a vertex that is tangent with the new end point, a point for which the line through itself and the end point touches the chain only in that point and never crosses the chain, again see figure 3a for an example of a tangent line (the blue dotted line). Start by scanning from the outer side of the left chain towards the apex, if it contains a point p that is tangent with the end point all the scanned points are removed from the left chain and the new end point is added with p as its predecessor. If the left chain does not contain a tangent point continue with scanning the right chain away from the apex until a tangent point p is found. This means all the shortest paths from s to the current d are identical up to p and diverge there, thus p is the new apex of the funnel, both chains are updated accordingly. A simple example of a funnel starting as a triangle and extending multiple steps through a triangulated polygon is given in figure 3b. The thick line segments are the boundary of P, the line segments labelled d_x are the diagonals added to the funnel and the dotted line segments are the chains showing the shortest path from the apex to the end points of the diagonal. The process described here gives a method to calculate the shortest path from s to t in $O(N)$ time given the order of triangles to traverse.



(a) Example of a funnel with source s and apex a, taken from [2]

(b) Funnel extending through a simple triangulated polygon, taken from [6]

Figure 3

2.2 Shortest path algorithm, Hershberger, Snoeyink [1]

Remember the definitions of a simple polygon P with N vertices and a source vertex s on the boundary of P. The shortest path from s to v is denoted by $\pi(s, v)$ where v is a vertex of P. The goal of this paper [1] is to construct a data structure, a shortest path tree, in $O(N)$ time that can be used to query $\pi(s, v)$ for any v on P in $O(\log_2 N + k)$ time. $O(\log_2 N)$ time for locating v and $O(k)$ time to report the path from v to s by traversing through the k vertices where the path makes a turn. The key concept used in this algorithm is a funnel. As seen before a funnel can be used to find a shortest path by starting at s and extending it towards a target point t. In the process of finding one shortest path the funnel actually finds all the shortest paths to the vertices it passes. This means it is possible to find all the shortest paths by making the funnel branch through the entirety of P, and thus T passing over all N vertices. This can be done by rooting T at the triangle containing s and visiting every node of the tree. An efficient way to save a shortest path is to save the predecessor on the shortest path towards s for every vertex in P. The complete shortest path can then be reconstructed by recursively moving backwards from t to its predecessor until it reaches s, this is the factor k in the query time. Saving all the shortest paths in this way constructs a shortest path tree where every node is a vertex with a link to its parent in the tree, which is the next vertex on the shortest path towards s.

A funnel is initialised with apex s and both chains pointing towards an arbitrary adjacent vertex v_1. The initial edge (s, v_1) is treated like a diagonal. The funnel is then extended recursively. Every step the funnel is extended by taking the most recently added diagonal and finding an adjacent vertex so that it forms a complete triangle that has not been processed yet. Note that this adds two new edges to the funnel, one edge from both the end points of the diagonal

to the adjacent vertex. In order to do this the funnel is split into two separate funnels every step. Then both new funnels recursively extend themselves only if the newly added edge is a diagonal in P and not a boundary edge of P. This means for every degree-2 node in T the funnel is split into two, but one of the new funnels will be a boundary edge and will not recurse further. For every degree-3 node in T the funnel is split and both new funnels will recurse further so the entire polygon will be processed. See figure 4 for an example how a funnel can be extended. The only not trivial part of this algorithm is how to split funnels in an efficient way.

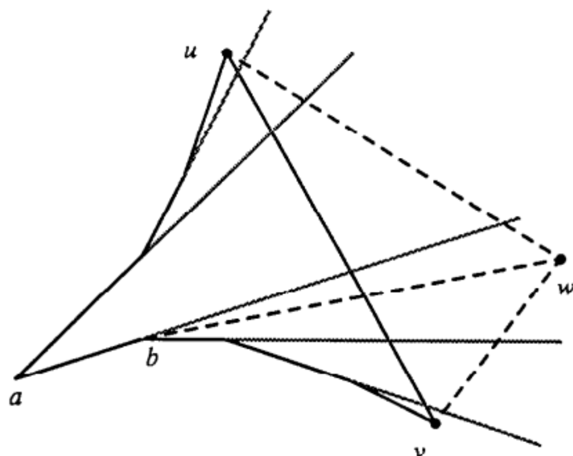


Figure 4: Illustrates how a funnel can be split, taken from [1]

Hershberger and Snoeyink [1] propose a way to split a funnel using very basic data structures, a stack and a double-ended queue (deque) both implemented using an array for direct access. The stack is used to track the history of the funnel, allowing it to be restored to previous states. The deque stores the vertices of the funnel and supports five operations:

Length(deque)	Return the number of items in the deque.
Index(deque, i)	Return the ith item in the deque.
Add(f, deque, x)	Add the item x to the f (front) or b (back) of the deque.
Split(f, deque, i)	Return the items in f or b of and including item i and discard the other half of the deque.
Undo(deque)	Undo the most recent **Add()** or **Split()** operation.

Code examples for the front of the deque can be found in figure 5, it also shows that all the operations can be performed in constant time. The algorithm

Length(*deque*) return *last* − *first* + 1 **Add**(f, *deque*, *x*) decrement *first* push (**add**, f, *deque* [*first*]) to stack set *deque*[*first*] ← *x* **Split** (f, *deque*, *i*) check $0 \leqslant i <$ **Length**(*deque*) push (**split**, f, *last*) to stack set *last* ← *i* + *first*	**Index**(*deque*, *i*) check $0 \leqslant i <$ **Length** (*deque*) return *deque*[*i* + *first*] **Undo** (*deque*) if stack top is (**add**, f, *x*) set *deque*[*first*] ← *x* increment *first* else stack top is (**split**, f, *i*) set *last* ← *i*

Figure 5: Table that shows code examples for the front of the deque, taken from [1]

starts at the root of T and covers the entire tree in a DFS manner. Going downwards toward the leafs of the tree the funnel is extended by an **Add()** followed by a **Split()**. Each add operation adds a history node to the stack containing the previous value (vertex) on the place in the deque where the new vertex is added. For every node visited the splitting index i is calculated once. This is the point of tangency of the funnel and the newly added end point as discussed when introducing funnels. The splitting index i is the predecessor of the end points used to construct the shortest path tree. This splitting index i can be found efficiently by using exponential search on the funnel until it passes the new end point and then performing binary search between the two last indexes to find the exact tangency point. Each split also adds a history node to the stack containing the index i where the funnel has been split, effectively removing a part of the funnel. When the algorithm reaches a leaf it proceeds to backtrack using the **Undo()** operation until it reaches a degree-3 node where it has not yet explored the right child. This time it will traverse the right child until it reaches a leaf. In this fashion the entirety of T (and thus P) will be processed and for every vertex a predecessor on the shortest path towards s will be known, forming the shortest path tree. Undoing an add operation restores one element in the deque containing the funnel to a previous state. Then undoing a split restores the boundaries of the deque used to represented the funnel to a previous state.

This results in a shortest path tree constructed in $O(N)$ time. In order to query a shortest path in the promised $O(\log_2 N + k)$ time a point location data structure is needed that finds target points in $O(\log_2 N)$ time. This has been extensively researched by Kirkpatrick [10] and Edelsbrunner et al [11]. Actually reporting the shortest paths is then trivial. Use the point location data structure to locate the desired vertex in the shortest path tree and then recursively call

its predecessor until reaching the source point s.

3 I/O-efficient preliminaries

In this section we will discuss some I/O-efficient techniques that are relevant to the I/O-efficient shortest path algorithm proposed by Agarwal et al [2]. First we will discuss I/O-efficient stacks and deques that are necessary for representing an I/O-efficient funnel. Then some techniques for I/O-efficient tree traversal will be discussed, this is necessary for efficient traversal of the triangulation and for reporting shortest paths from a shortest path tree. Then finally we will discuss I/O-efficient sorting.

3.1 I/O-efficient stacks and deques

A stack is the most simple data structure that allows values to be pushed and popped from the top in constant time. The internal memory variant is usually implemented with an array and an index counter. A push inserts a value at the index counter and increase the index counter by one. A pop will decrease the index counter by one and read the value at the index counter. This can obviously be done in constant time per push or pop. The only issue arises when too many items are added and the underlying array needs to be resized, this needs $O(N)$ time, amortised over N inserts makes this implementation $O(1)$ amortised. This data structure is trivial to make I/O-efficient. Remember that memory is read and written from and to disk in blocks of size B at a time. Two buffers of size B are kept in main memory. These buffers allow inserts and removes without any I/Os. There are two cases where an I/O happens. When a push is done and both buffers are full the first is written to disk and when a pop is done when both buffers are empty the previous buffer is loaded from disk. Two buffers are necessary otherwise a worse case scenario of pushes and pops could trigger a read/write on every operation. This gives an amortised performance of $O(\frac{1}{B})$ I/Os for inserting and removing. Note that a resize of an underlying array should never be necessary since the disk is of conceptually unlimited size, enough dish space should be allocated when initialising the stack.

A deque (double-ended queue) is essentially a stack that can be accessed from both sides, the top and bottom, or in queue terminology the front and back. In internal memory this can be implemented using two stacks that have one underlying array. One stack represents the front of the queue and the other the back. The underlying array is filled starting in the middle and moving towards both ends. When one of the stacks reaches the end of the array a resize will be performed. The performance for each operation is identical to that of the stack. A push and pop can be done in amortised $O(1)$ time, this includes the occasional resizing. Making a deque I/O-efficient is also similar to that of the stack. Four buffers will be kept in main memory, two for front access and two for back access of the deque. An I/O happens when an push is done on a full buffer, then it is written to disk, or a pop is done on an empty buffer,

a previous buffer is read from disk. Again, no resizing of the underlying array has to be done due to the conceptually unlimited size of the disk. Performance remains $O(\frac{1}{B})$ I/Os for pushes and pops.

3.2 I/O-efficient tree traversal

There are many variants of trees used in computer science, to keep this subsection relevant, only the techniques necessary for the I/O-efficient shortest path algorithm will be discussed. First we will discuss why the triangulation of a polygon should be in post order. Then we will discuss I/O-efficient DFS (depth-first search) and I/O-efficient node to root traversal. The DFS is used as a pre-processing step and therefore it may take up to $O(Sort(N))$ I/Os. The node to root traversal is used to output the answer to a query and should only take $O(Scan(k))$ I/Os, where k is the length of the path. For this to be possible, the traversed tree needs to be formatted which can be done as a pre-processing step which costs $O(Sort(N))$ I/Os.

The underlying data structure of a tree is generally an array and when using the hard disk a file is very similar to an array. There are a few different methods of ordering the nodes of the tree in this array which greatly impacts the amount of I/O operations necessary to traverse a tree. Post order traversal is done by visiting the left child first, then the right child and lastly the node itself. Applying this recursively means the left most leaf will be outputted first and after that the leaf to the immediate right. The very last node outputted will be the root of the tree. The reason we want our triangulation T of the polygon to be in post order is because when we scan T it will visit the triangles in a certain order, without any jumps costing additional I/O's. This order ensures that when we are scanning T from a certain triangle towards the root we will visit every triangle on the path between that triangle and the root in the correct order to form a shortest path.

DFS is a tree traversal algorithm that traverses the tree in a specific order, often used for greedily searching. From the root it will traverse toward the leafs by always choosing the leftmost child. When it reaches a leaf and has not yet terminated it will backtrack to a previous node that has unvisited children and traverse the leftmost child it has not yet visited. In this manner the entirety of the tree will be traversed. An I/O-efficient DFS is described by Chiang et al [12]. The described method is designed for a directed graph but also works for trees. Let G be a graph with V vertices (nodes in a tree) and E edges (edges are directed away from the root) represented by three arrays. An array A of size E containing all the edges, sorted by source. Two arrays, $Start[i]$ and $Stop[i]$ giving the starting index and ending index of all vertices (nodes) in A that are children of i. So vertex i has the following children $A[j] \mid Start[i] \leq j \leq Stop[i]$. The method uses a stack of vertices to maintain the path from the root to the current vertex. As expected for DFS, it starts at the root and retrieves the list of children, the leftmost unvisited child is the next active vertex. When all children of a vertex have been visited it is removed from the stack and the previous vertex on the stack is the active vertex. Using this method the entirety

of G will be traversed using $O((1 + \frac{V}{M})Scan(E) + V)$ I/Os.

An I/O-efficient node to root traversal method that fits the allowed I/O restraints is given by Zeh [13]. A tree T of height h is split up in layers of height $h' = \log_2 B$ resulting in layers $L_0, ..., L_{\lceil h/h' \rceil - 1}$. Each layer is a forest of subtrees of T, see figure 6. Each subtree fits in a single memory block B and allows traversal of an entire layer. This allows node to root traversal in the desired $Scan(k)$ I/Os, where k is the depth of the starting node. The decision to store each subtree in a single memory block might result in very sparsely populated memory blocks. To limit the necessary storage space to a constant factor sparsely populated memory blocks are merged together. Pairs of memory blocks with less than $B/2$ vertices are merged together, at most one such block can remain unmerged. In a worse case scenario the size of T roughly doubles.

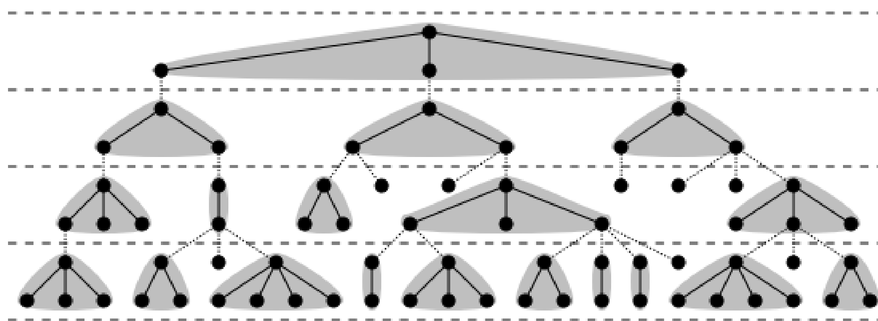


Figure 6: A tree divided in layers and highlighted subtrees, taken from [13]

3.3 I/O-efficient sorting

I/O-efficient sorting is not explicitly used in the I/O-efficient shortest path algorithm, but the resulting term $Sort(N)$ that is used often deserves some explanation. Ignoring a rare exception all the I/O-efficient sorting algorithms are either based on merge sort or distribution sort. Merge sort is a bottom up approach where small subsets are sorted and later all the subsets are recursively merged together. Distribution sort is a top down approach where median points are computed and the data is then put in buckets based on the median values. These buckets are then recursively sorted, the results are added back together, they can simply be appended, there is no need for merging. A simulation done by Vengroff et al. [14] shows that merge sort is overall the faster algorithm and the focus of this subsection will be on an I/O-efficient merge sort algorithm.

An optimal I/O-efficient merge sort is given by Aggarwal et al [4]. The approach closely resembles the internal memory variant of the merge sort algorithm. Start by sorting subsets of the data and later merge these together. Logically the size of these initially sorted subsets is M which means after this initial phase there are $\frac{N}{M}$ sorted subsets on the disk having used $\frac{N}{B}$ I/Os. The merging phase merges $\frac{M}{B} - 1$ subsets together by loading the first B values of

each subset in main memory. A buffer of size B is initialised to keep track of the output. The buffer is filled with the lowest B values of all the subsets in main memory, in a sorted manner, and then written back to disk. If the B values of a subset in main memory are all processed it loads the next batch of that subset until all the values are processed and all the loaded subsets are completely merged. The result of this merging phase is a new, much larger, subset. This merging phase is done recursively until the result is a single fully sorted set. This approach results in an optimal sorting algorithm that uses $O(\frac{N \log_2(N/B)}{B \log_2(M/B)})$ I/Os often written as $Sort(N)$.

4 I/O-efficient shortest path algorithm

In this section an I/O-efficient algorithm proposed by Agarwal et al. [2] will be discussed. The well known funnel concept is converted to use I/O efficient data structures in order to find a shortest path. A key problem for creating a shortest path tree is the inability to efficiently split a funnel at a degree-3 triangle. Guibas et al. [7] use a finger tree which no I/O-efficient variant is known for. The approach of Hershberger and Snoeyink [1] uses data structures that have I/O-efficient equivalents except that their approach requires a deque that allows reading of a specific index in constant time, which is not supported by the I/O-efficient deque. The approach of Guibas and Hershberger. [8] requires efficient splitting and concatenation which also has no I/O efficient variant. The algorithm proposed by Agarwal et al. [2] is a divide and conquer algorithm that recursively partitions P into smaller subpolygons, these smaller subpolygons can be solved in main memory and the results combined. The key component to efficiently combine the results is a sparse shortest path tree, that is a tree that stores the shortest path from a source point s to $k = O(\sqrt{\frac{M}{B}})$ points, note that we will improve this to $O(\frac{M}{B})$ in the next section. First a $O(Scan(N))$ I/O algorithm to compute such a sparse shortest path tree will be given and then the divide and conquer algorithm will be discussed in more detail. Then we will discuss how to process the smaller subpolygons in main memory and combine them into a shortest path tree. Finally the shortest path tree can be used to compute a shortest path map M that supports shortest path length queries in $O(\log_B N)$ I/Os and the shortest path itself in an additional $O(Scan(k))$ I/Os.

4.1 I/O-efficient sparse shortest path tree

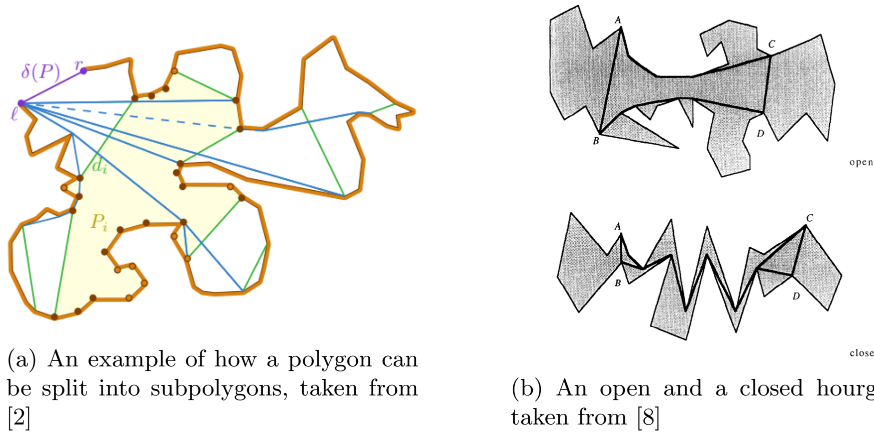
Given a polygon P and the triangulation stored as a post order dual tree T a shortest path can be found using funnels in much the same way as in section 3. T is used to find the triangles the shortest path must traverse and a funnel is maintained throughout that path using a stack and deques. With this approach, using I/O-efficient stacks and deques, a shortest path can be found in $O(Scan(N))$ I/Os. As we have seen in section 2.2 the approach of Guibas et al. [7] extends this algorithm to compute the shortest path tree. However,

as already mentioned, this requires an efficient way to split funnels at degree-3 triangles and there is no known way to do this. Instead of the full shortest path tree the goal of this step of the algorithm is to create a sparse shortest path tree that stores the shortest paths from a source point s to $k = O(\sqrt{\frac{M}{B}})$ target points, this can be done in $O(Scan(N))$ I/Os. The triangles traversed on the paths from s to every k_i can, again, be found using T, this takes $O(Scan(N))$ I/Os. During this scan all the degree-3 triangles where the funnel has to be split are marked. For each of these degree-3 triangles a vertex where the funnel has to be split is stored as a finger ordered along the funnel in main memory where it can be accessed without any I/Os. The vertices between two fingers will be stored in a I/O-efficient deque. Traversing a degree-2 triangle is straightforward, both ends of the funnel are modified as usual and added to the correct deque between the fingers, if necessary the apex is changed. In order to traverse a degree-3 triangle the corresponding finger is located without any I/Os. The funnel is split by creating new fingers and new empty deques in between them. This requires at most k new deques and since there are at most k degree-3 triangles this requires at most $O(k^2)$ I/Os and $O(M)$ memory in total. During this process the resulting sparse shortest path tree T_s is created. T_s only stores degree-3 nodes and stores the path between these nodes as arrays. To build T_s while extending the funnel an in memory tree $T_s{}'$ is maintained that stores the same degree-3 nodes as T_s and has I/O-efficient stacks to represent paths between them. Since $T_s{}'$ only has k nodes this requires k - 1 buffers of the I/O-efficient stacks, using $O(\sqrt{\frac{M}{B}})$ memory. Whenever the apex of a funnel is changed another part of the shortest path between two degree-3 nodes is known, these vertices are pushed to the correct stack in $T_s{}'$, when that stacks buffer is full it is saved to T_s on disk.

4.2 Divide and conquer

The divide and conquer algorithm splits P into $O(\sqrt{\frac{M}{B}})$ subpolygons of roughly equal size. This can be done efficiently by computing the centroid decomposition [12] and transforming that into a balanced hierarchical decomposition as described by Chazelle [15]. This is done recursively until a subpolygon fits into main memory, then all the local shortest path info can be computed without any I/Os. An example of this can be seen in figure 7a.

All the subpolygons defined by the green diagonals have to cross door diagonal $\delta(P)$ to reach s. All the orange vertices have a parent on their shortest path towards $\delta(P)$ that lies within its own subpolygon and can therefore already be calculated without any I/Os. All the brown vertices can see some part of the diagonal (green) that is closest to $\delta(P)$. For these vertices an hourglass $H(d_i, \delta(P))$ is constructed. Hourglasses are first described by Guibas and Hershberger [8]. Hourglasses are simply two funnels concatenated together. It can either be open or closed, if it is open neither funnel has an apex and if it is closed the funnels have an apex and possibly vertices between both apexes, see figure



(a) An example of how a polygon can be split into subpolygons, taken from [2]

(b) An open and a closed hourglass, taken from [8]

Figure 7

7b. If the hourglass is closed it is clear that the vertex cannot see $\delta(P)$ and a parent is assigned, if the hourglass is open there is a possibility the vertex can see part of $\delta(P)$. The part of $delta(P)$ the vertex can see is restrained by two vertices, the left and right restrainer. If the vertex cannot see $delta(P)$ a parent is assigned otherwise a left and right restrainer. Agarwal et al. argue that it is possible to compute these values with a constant amount of information at every vertex. The hourglasses necessary for these computations consist of a left and right chain. This is where the sparse shortest path tree defined above comes into play. See figure 7a, the blue solid and dashed lines are the left chains from every d_i to $\delta(P)$. These blue lines will be part of the sparse shortest path tree and can be easily retrieved at this point. Right chains can be constructed in the same manner. In order to write the following concisely lets define some notations. $\overrightarrow{xy}$ denotes a line segment from x to y. L and R are the left and right chain from a d_i to $\delta(P)$. l and r are the left and right endpoint of $\delta(P)$. p_l and p_r are the left and right successor of vertex v on L and R. $w_l(v)$ and $w_r(v)$ are the left and right restrainer of v to d_i. $C^{\mathrm{d}}(v)$ is the cone restricted by $\overrightarrow{vw_l(v)}$ and $\overrightarrow{vw_r(v)}$. B is the subpolygon in which v lies and A is the other subpolygon. Now, these rules apply to assign $w_l(v)$, $w_r(v)$ and $p(v)$ appropriately.

$w_l(v) = p_l$ and $w_r(v) = p_r$ if p_l lies left of $\overrightarrow{vp_r}$, and $p(v) =$

$$\begin{cases} p_l & \text{if } p_l = p_r \in A \\ p_l & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_l \in B, \text{ and } p_r \in A \\ p_r & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_r \in B, \text{ and } p_l \in A \\ p_l & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_l{,}p_r \in A,\ p_l \in \triangle(v, p_r, r), \text{ and } p_l \in C^{\mathrm{d}}(v) \\ w_r(v) & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_l{,}p_r \in A,\ p_l \in \triangle(v, p_r, r), \text{ and } p_l \notin C^{\mathrm{d}}(v) \\ p_r & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_l{,}p_r \in A,\ p_r \in \triangle(v, p_l, r), \text{ and } p_r \in C^{\mathrm{d}}(v) \\ w_l(v) & \text{if } p_l \text{ lies right of } \overrightarrow{vp_r},\ p_l{,}p_r \in A,\ p_r \in \triangle(v, p_l, r), \text{ and } p_r \notin C^{\mathrm{d}}(v) \end{cases}$$

These rules apply for open hourglasses, closed hourglasses can be seen as open hourglasses when l and r are both at the apex of the closed hourglass. These rules use a constant amount of information to calculate its parent on the shortest path towards s, this means it is done in $O(N)$ I/Os. Now the full shortest path tree can be constructed that contains the shortest path from every vertex in P to s. This is done by locating the triangle that contains s and using the divide and conquer algorithm as described above starting from every edge of that triangle. If a vertex v does not have a parent assigned but instead has w_l and w_r then the parent is assigned by checking whether s lies on the left, on the right or inside the cone constructed by w_l, w_r and v. Now the length of all the shortest paths can be computed by using DFS as described in section 3.2 using $O(Sort(N))$ I/Os. The results are stored with the vertices so they can be reported in $O(1)$ I/Os. In order to efficiently report the full shortest path from a vertex to the source the shortest path tree is stored in the efficient node-to-root path data structure also discussed in section 3.2. In order to support queries that ask the shortest path from s to any point within P the shortest path tree is converted to a shortest path map M. All the chains stored in the shortest path tree are extended until they hit the boundary of P dividing P into smaller triangles, for all the points in such a triangle their parent on the shortest path towards s is known. Now M can be stored in an I/O-efficient point location data structure. One such point location data structure that allows querying a triangle in $O \log_B N$ I/Os is presented by Arge et al [16]. This triangle has a reference to its parents vertex, the length of the path can be reported in $O(1)$ I/Os and the full path in an additional $O(Scan(k))$ I/Os using the node-to-root data structure.

5 Implementation

In the previous sections we have discussed the shortest path algorithms that will be tested in the experiment. However, not everything will be implemented and some parts of the algorithms might be implemented slightly different than originally proposed. In this section we will discuss what is implemented and all the changes made to the algorithms while implementing them. For the I/O-efficient shortest path algorithm we had to skip a lot of features due to very high implementation complexity and the amount of steps and other algorithms

it depends upon. We have decided to focus on the key concept of the algorithm, the sparse shortest path tree. This means that the divide and conquer part of the polygon is not implemented, neither are the I/O-efficient point location algorithm and the I/O-efficient tree traversals.

5.1 Internal funnel

For the internal algorithm the entire triangulation of the simple polygon is read from file at the start of execution. Then the triangulation is scanned to find the triangle containing the target point. The root of the triangulation is the source triangle. The path through the triangulation can easily be computed by starting at the target triangle and visiting the parent, do this recursively until the root is found. The funnel starts at the source so the calculated path is reversed. The funnel is implemented exactly as described in section 2 with a stack, a deque and an apex. Every time a diagonal is added one of the vertices of the diagonal is already a vertex at the end of either the left or right chain of the funnel. The other vertex of the diagonal should be added to the opposite side of the funnel. Lets say a new vertex v is added at the left side of the funnel, it is symmetrical for the other case. Now we need to calculate the point of tangency of v with the funnel. This is done by checking whether v and the two last vertices of the left chain make a left or right turn. In this case the point of tangency is found if a left turn is found. If this is not the case the last vertex of the left chain is removed and the process is repeated. As soon as we pass the apex the check needs to be inverted. A point of tangency is found when a right turn is found. When the apex has been passed and vertices are removed from the funnel they are saved onto the stack representing the shortest path found so far.

5.2 I/O-efficient funnel

There are only two differences between the I/O-efficient funnel and the internal funnel. The stack and deque data structures are replaced by their I/O-efficient variants and the process of finding the shortest path within the triangulation is different. Instead of loading the entire triangulation it is only scanned one block at a time to find the triangle containing the target. Then the path is constructed in the same manner by recursively visiting the parent until the root is found. In order to do this I/O-efficiently the triangulation T has to be saved in post order. The rest of the process is exactly identical.

5.3 Internal shortest path tree

As mentioned before, for the internal memory variant of creating a shortest path tree we have implemented the algorithm proposed by Hershberger and Snoeyink [1]. This choice has been made because the algorithm has competitive algorithmic complexity with other state of the art algorithms, a lower implementation complexity and the capabilities of the algorithm are similar to that of the I/O-efficient variant. Furthermore Hershberger and Snoeyink provide a

detailed description of how to implement their algorithm with small pieces of pseudo code shown in figure 5. The pseudo code really shows the core of how the algorithm works and we have used this code in our implementation with one exception. Their code for splitting a funnel when a vertex is added to the front of the deque seems to be mirrored for our implementation. To keep everything consistent throughout our implementation we have used their pseudo code for splitting the front of a funnel for splitting the back of a funnel. The same change has been made for undoing a split. The rest of the algorithm has been implemented exactly as described by Hershberger and Snoeyink.

5.4 Sparse shortest path tree

The sparse shortest path tree has not been described in great detail by Agarwal et al [2]. In order to implement it a lot of details still had to be figured out. The final implementation is described below and is a slightly improved version of what is described by Agarwal et al. As mentioned before we have improved the amount of target points the sparse shortest path tree can contain from $k = O(\sqrt{\frac{M}{B}})$ to $O(\frac{M}{B})$.

Given, are a binary tree T representing the triangulation of the polygon P with N vertices in post order where the root contains the source point s and a set of target points t that are inside P or a vertex of P. The first step is to extract the smallest subtree of T that contains all the points of t and source point s. Source point s is located at the root and all the points in t are found by scanning through T. If a target point is a vertex of P it can be part of multiple triangles, to ensure the one closest to s is found T is scanned from front to back. Given the fact that T is in post order this ensures the triangle containing a target point that is closest to s is visited last and will overwrite any previously found triangles. The triangles containing target points are kept in memory, again using the fact that T is in post order we can scan through T from front to back and visit every triangle in the correct order from bottom to top to create paths from the targets to s. At some triangles two paths merge, these are called degree-3 triangles. These are kept in memory and the split vertex is defined, the vertex where an incoming funnel will be split. These degree-3 triangles form a binary tree, so it is known at each split which degree-3 triangles are down the left and right path.

In order to trace the funnel through the polygon two data structures are used. The funnel data structure that is represented by $k+1$ deques, where k is the amount of degree-3 triangles, at worst, equal to the amount of targets. The parent of a split vertex $P(v)$ is placed between each deque, initially these are empty. A $P(v)$ is a vertex that is part of the funnel that is point of tangency of the split vertex and the funnel. Figure 8 illustrates an example funnel on the left and the described data structure on the right, where dots are the $P(v)$ and the lines are deques, both with assigned vertices. In practice the funnel is represented by two arrays, one of all the deques and the other with all the $P(v)$. The second data structure is a binary tree of degree-3 triangles with a

constant amount of extra data in each node to help tracking which deques and parents of split vertices are part of the current funnel. When a degree-3 triangle is encountered the funnel will be split into a left and right funnel. Both these funnels will only operate on a subset of the deques and parents of split vertices. This subset will always be consecutive entries in the two arrays of the funnel data structure and can thus be represented as a min and max index value in the binary tree. The binary tree is pre-processed so every time a split is made it is known on which deques and $P(v)$ to operate. It is a matter of assigning which $P(v)$ belongs to a degree-3 triangle and the left and right most deque it is allowed to use, all of these values are indices in the arrays of the funnel data structure. See figure 8c to see what this binary tree looks like. Using the example illustrated in figure 8, the first degree-3 triangle encountered will be $sv(1)$ and using the min and max deque saved in the binary tree the funnel will use all of the deques and parents of split vertices. After the first split the next degree-3 triangle in the left funnel will be $sv(0)$, using the binary tree the funnel will only operate on the deques in the array indices 0 and 1 containing $v(0)$ and $v(2)$ and on the $P(v)$ containing $v(1)$. Note that there is always one less parent of a split vertex in a funnel than there are deques, this discrepancy can easily be handled which will be shown in a code example later. After the first split the $P(v)$ containing $v(3)$ is not part of either of the funnels, but $v(3)$ should be part of both outgoing left and right funnels, thus it is added to both funnels on the appropriate side. Splitting a funnel represented this way costs two deque adds and the funnel uses $O(k * B)$ memory, where B is the size of a disk block. This is an improvement on the reported $O((k * B)^2)$ by Agarwal et al [2]. When a degree-3 triangle is reached that is a leaf in the binary tree the split is done as before except that there is no new node in the binary tree to retrieve information from on which deques to operate. After this split no more degree-3 triangles will be encountered which means the funnel can be represented with a single deque. Therefore the left funnel at such a split can operate on the minDeque of the previous degree-3 triangle and the right funnel on the maxDeque, these values are consecutive.

Extending a funnel represented by this data structure is fairly easy. Normally a funnel is represented by a single deque. Since ours is represented by multiple deques and k parents of split vertices the peek, pop and push operations to the front and back should be modified to reflect these changes. See Algorithm 1 for an example of how a front pop on this funnel would work. min and max deque values are taken from the binary tree data structure for the current degree-3 triangle we are traversing towards. Deques and splitVertex are the arrays mentioned in the funnel data structure. The rest of the operations can be modified in similar ways.

The sparse shortest path tree is created I/O-efficiently by tracing the above described funnel through the subset of T that contains all the targets. This can be done in $Scan(N)$ I/Os by starting at the back of the file and scanning towards the front since the it is stored in post order. This means the funnel starts at the source vertex s and extends towards the targets, every split the right path is visited first according to the post-order layout of T. When a leaf in

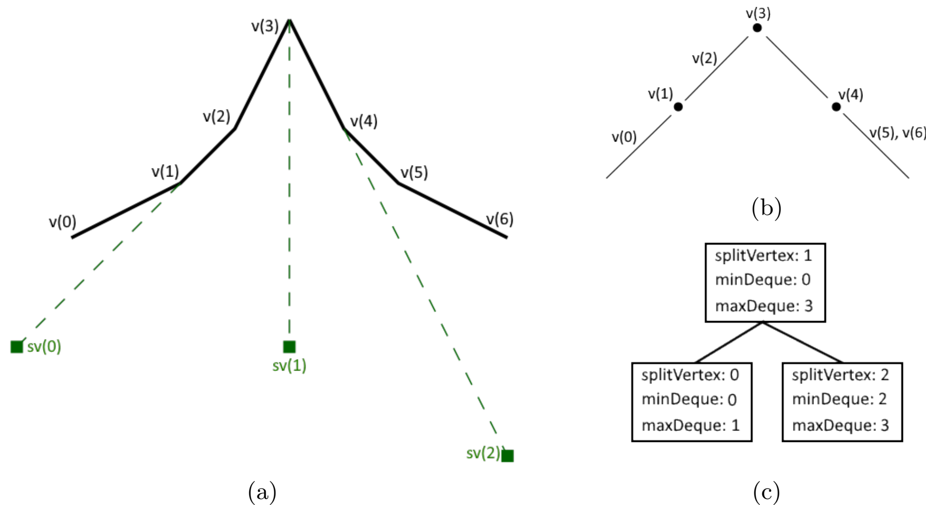


Figure 8: Example funnel and internal representation of funnel

T has been reached it is trivial to return to the last split and continue extending the left outgoing funnel. All that has to be done is set the correct min and max deque that represent that funnel.

6 Experiment

The experiment is split into two parts. The first part is comparing how fast both algorithms can compute a single shortest path given a triangulated polygon in post order without any further pre-processing. In the second part we compare the pre-processing steps of the algorithms. For the internal algorithm of Hershberger and Snoeyink [1] we compute a complete shortest path tree. For the I/O-efficient algorithm we only compute the sparse shortest path tree, a shortest path tree that contains only paths to a predefined set of targets, this is due to time constraints and very high implementation complexity of the algorithm. A consequence of this choice is that the comparison between the algorithms in the second part is not totally fair. It will however still give us answers to at least research questions one and two, whether the theoretical algorithmic complexities of the algorithms hold up in practice when applied to large polygons.

The experiment described above has two requirements. One, it requires a limitation on the size of the main memory, this is necessary to reduce the total time it takes to perform the experiment. Generating polygons of sizes greater than 16gb would take days. Two, it requires very large polygons such that they do not fit in limited main memory. The first requirement is met by using the cgroups (control groups) [17] feature of linux. It limits the resource usage of CPU, memory, disk I/O, etc for processes. Using this feature main memory is limited to 128mb and swap memory (on disk) is limited to 10gb,

```
for i ← minDeque to maxDeque do
    if deques[i] has elements then
        return deques[i].pop()
    end
    // Skip the last P(v), necessary because there is always
       one more deque than P(v) in the funnel.
    if i not is maxDeque then
        if splitVertex[i] is not empty then
            return splitVertex[i]
        end
    end
end
```

Algorithm 1: Deque pop front

which is practically unlimited for the polygons used. Since writing data to disk is such an expensive operation it would be a very stalling operation to do this when main memory is full. Note that most operating systems including linux have something called a swappiness constant. Therefore when the swappiness constant is reached (linux default is 60%) it starts writing data from main memory to swap memory on disk. For this experiment the swappiness constant is left at the default 60%.

For the second requirement, the large polygons, a framework called CGAL (Computational Geometry Algorithms Library) is used [18]. It has a functionality to generate simple polygons within a unit square with a variable amount of vertices. However the algorithmic complexity for generating a simple polygon with CGAL is $O(N^4 \log_2(N))$ where N is the amount of vertices, this is not suitable for generating large polygons. In order to speed up this process to an $O(N)$ time algorithm a large grid where each cell contains a small polygon of 200 vertices is generated. Generating many small polygons is much faster than generating one very large polygon. All the cells are connected to each other using infinitely small corridors creating one large simple polygon P, see fig 9, this is done in one scan of the entire polygon. Then CGAL is used to create a triangulation of the final polygon in post order which is used as input for all the shortest path algorithms. The size of the generated polygons is easily adjusted by increasing the size of the grid and thus the amount of small polygons generated.

Finally, the experiment has been executed on a system with the following specs:

- CPU, Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz (4CPUs)
- Motherboard, G1.Sniper B5
- RAM, 16384MB
- OS, Windows 10 Home 64-bit

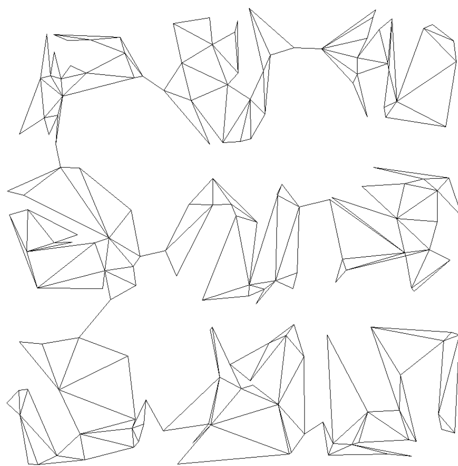


Figure 9: An example of a simple polygon generated using a 3x3 grid where each cell contains a 20 vertices polygon

On this windows 10 PC Ubuntu-18.04 is ran on a virtual machine using Oracle VM VirtualBox with the following settings:

- CPUs, 4
- CPU execution cap, 100%
- RAM, 8192MB

7 Results

In this section we present the results obtained from running the experiment described in the previous section. Remember that for all the experiments the main memory is limited to 128 mb using cgroups. Our hypothesis is that the algorithms designed without I/O-efficiency in mind will be faster on small polygons but fall off as soon as too much main memory is consumed.

7.1 Single shortest path experiment

The first part of the experiment is finding single shortest paths through simple polygons without any pre-processing. The input is a triangulation of the simple polygon in post order, a source point and a target point. The output is a list of vertices denoting the full path from source to target. The targets are uniformly randomly selected vertices of the polygon. In a single run on one polygon a total of 10 shortest paths will be computed. In order to prevent outliers where, by chance, the shortest path to the randomly selected targets is very short a total of 10 runs will be executed per polygon size. This means that a total of

100 shortest paths are computed for every polygon size. The plotted results are the average time in seconds it takes for a single shortest path to be computed. The results for this part of the experiment are split into four graphs in order to clearly show the relevant results to answer the research questions, see figure 10. In figure 10a all the data is shown. The I/O-efficient funnel results are hard to read, figure 10b zooms in on the results of the I/O-efficient funnel. Then figure 10c shows a subset of the results where the polygon sizes are relatively small and the performance of the I/O-efficient funnel and the internal funnel are comparable. Figure 10d displays the average amount of vertices in a shortest path for all the polygon sizes. This has been done to check whether our approach of using randomly selected vertices of the polygon is suitable.

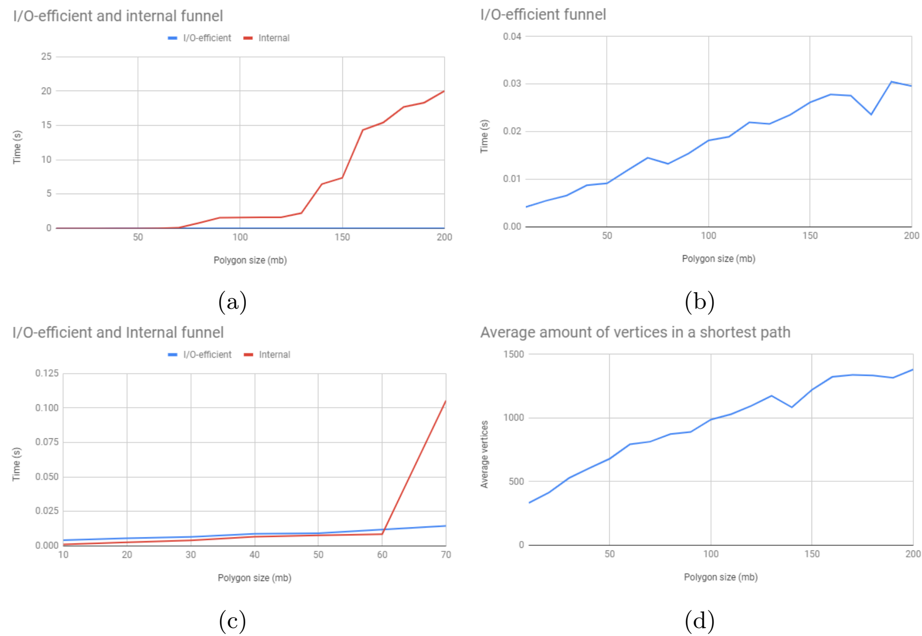


Figure 10: Results of single shortest paths through simple polygons

7.2 (Sparse) shortest path tree experiment

The second part of the experiment uses the same polygons, but instead of computing single shortest paths we now focus on pre-processing the polygons to allow efficient shortest path querying. The I/O-efficient algorithm creates a sparse shortest path tree containing the shortest paths to k targets, where k has been chosen to be 10, similar to the amount of paths calculated in the previous experiment. The internal memory algorithm calculates the full shortest path tree containing the paths to all N vertices of P. Note that this is a significant difference in the amount of work each algorithm has to perform and the results

should not be compared in absolute values. We can however report the results to get a general idea how the algorithms behave. Again the experiment is ran multiple times to eliminate potential outliers caused by the randomly selected targets. This should only affect the I/O-efficient algorithm since the internal algorithm calculates a full shortest path tree it does not need random targets. The reported result is the average value for a single run with a total of 10 runs performed. The results are split into three graphs where figure 11a shows all the data, figure 11b zooms in on the I/O-efficient algorithm and 11c only shows the part where the performance is relatively close together for the smaller polygons.

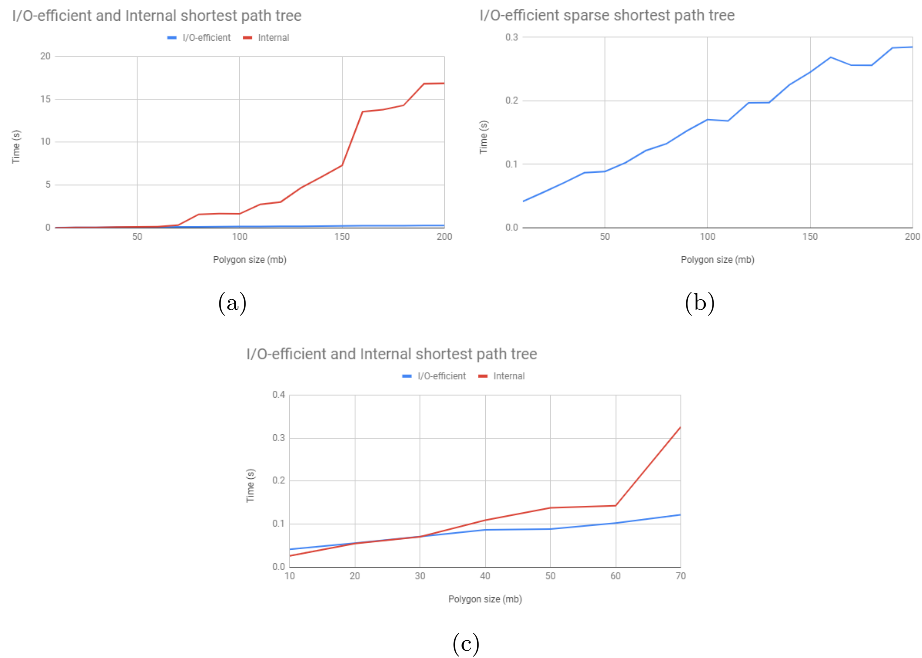


Figure 11: Results of creating (sparse) shortest path trees

Then finally something that caught our attention during implementation. The size of the funnel remains very small throughout all the experiments. To be able to discuss about this topic we have measured the absolute maximum size of the funnels, so the maximum amount of vertices ever present in a funnel, for every size of polygon. This can be seen in figure 12.

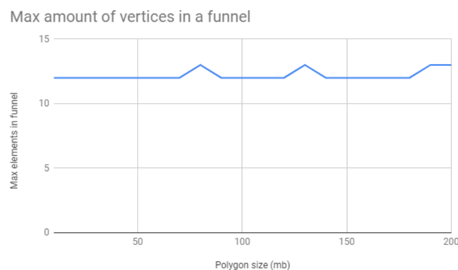


Figure 12

8 Discussion

In this section we will first discuss two important points that limit how our results can be interpreted. Then we will interpret our results and attempt to answer our research questions.

8.1 Limitations

Firstly, as mentioned in the implementation section, the final I/O-efficient shortest path tree has not been fully implemented. Only the core principle, the sparse shortest path tree, has been implemented. This step in the algorithm has an algorithmic complexity of $O(Scan(N))$ I/Os. The full implementation would have an algorithmic complexity of $O(Sort(N))$ I/Os. Therefore it will not be possible to give a conclusive answer on research question three, whether the I/O-efficient algorithm performs better than the internal memory algorithm. However it is still possible to give a conclusive answer on the first two research questions and give our hypothesis for research question three.

The second point is about the final note in the results section. Conceptually, in the worse case scenario, a funnel can grow to contain every vertex of a polygon. Since the internal memory algorithms assume memory is not an issue they allocate enough memory for the worse case, so at least N vertices fit into the funnel. Since a funnel is represented by a deque that starts in the middle of the allocated memory and can extend into both directions, but in the worse case scenario only extends into one direction, the allocated memory should be extended to $2N$. For the I/O-efficient variant of a funnel (deque) this is less of a problem since it uses four buffers of size B and the size of the underlying file does not really matter. As mentioned in the results section we noticed during implementation that the funnels used in our generated polygons never grew beyond 12 vertices. This is due to how the polygons are generated. Two factors play a role. As mentioned in section 6, generating very large polygons takes much too long to be feasible, therefore we sped up the process by generating many small polygons in a grid and concatenating them together with infinitely small corridors. A side effect of this is that the funnel will always collapse when going

through such a corridor, resetting it to two vertices before extending through the next small polygon. The second factor is how CGAL generates its polygons, we do not know the full implementation but the resulting polygons have very sharp corners which also causes the funnel to collapse frequently within a small polygon. We are unsure what the effect of this observation is on the overall performance of the implemented algorithms. The internal memory algorithms are allocating a lot of unused memory and the I/O-efficient algorithms are never actually filling their buffers so no I/O operations are performed for the funnel itself. As mentioned, the effect of this on the performance is unclear. It also hard to say whether this also happens in polygons that are used for practical applications. These can take all forms and shapes, so some of them will have similar size funnels and others might have very large funnels. It would be very interesting to set up a separate experiment to test how the shape of the polygon (determines the size of the funnel) influences the performance of extending a funnel through the polygon. An easy to generate worse case example where the funnel will never collapse is two quarter circles positioned to touch each other that can be infinitely densely sampled, see figure 13.

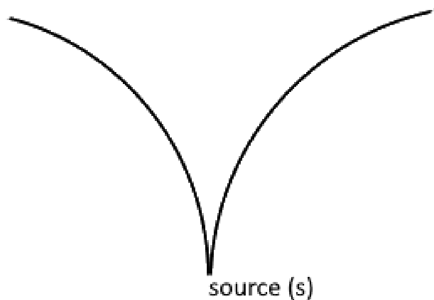


Figure 13

8.2 Evaluation of research questions

Finally, to answer our research questions:

> *1: Does the theoretical algorithmic complexity of creating the sparse shortest path tree of the proposed I/O-efficient algorithm hold up in practice.*

The algorithmic complexity of the implemented I/O-efficient algorithm is $Scan(N)$ which means there is a linear relation between the polygon size and time it takes to compute the sparse shortest path tree. Figure 11b clearly shows this linear relation. Therefore we conclude that the proposed algorithmic complexity of creating the sparse shortest path tree does hold up in practise. As

mentioned when introducing the research questions we have extended the algorithm to also test the performance of computing single shortest paths without any pre-processing. These results are shown in figure 10b and also show a clear linear relation between polygon size and computation time.

> *2: Does the theoretical algorithmic complexity of creating the shortest path tree of competing state of the art internal memory algorithms hold up in practice.*

The implemented internal memory algorithm is that of Hershberger and Snoeyink [1]. They promise a linear relation between polygon size and computation time for both a single shortest path and a shortest path tree. Figure 10 shows the results for a single shortest path. In figure 10c we can see that for polygon sizes between 10 mb and 60 mb the relation is linear but for larger polygons the computation time rises very fast. This is clearly the point where the main memory, limited to 128 mb, becomes too small to contain both the polygon and the data structures for the funnel. In figure 10a it becomes very clear that the relation between polygon size and computation time is not linear for polygon sizes greater than 70 mb. Where computing a shortest path in a 50 mb polygon takes 0.01 seconds, computing one in a 200 mb polygon takes 20 seconds. The exact same thing can be seen in the results of the shortest path experiment, see figure 11a and 11c. For polygons of sizes between 10 mb and 60 mb the relation is linear and for bigger polygons the computation time rises very fast. Therefore we conclude that the proposed algorithmic complexity of creating a shortest path tree with an internal memory algorithm does not hold up in practice for larger polygons.

> *3: Does the I/O-efficient algorithm become faster than the internal memory algorithms for larger problems.*

As already mentioned multiple times, this research question can not be answered conclusively but it is possible to speculate. There is a difference between the I/O-efficient algorithm for computing a single shortest path and computing the sparse shortest path tree. The algorithm for computing a single shortest path has been fully implemented and has an algorithmic complexity of $O(Scan(N))$ which is equal to that of its internal memory competitor. The algorithm for computing the sparse shortest path tree has an algorithmic complexity of $O(Scan(N))$ but in order to extend it to a shortest path tree it would become $O(Sort(N))$ which is a factor $O(\log(N))$ worse than its internal memory competitor. For a single shortest path the results are comparable and figure 10c shows that they are very close together for small polygons smaller than 60 mb. However figure 10a shows that for polygons greater than 70 mb the I/O-efficient single shortest path algorithm is significantly faster. For the (sparse) shortest path tree experiment the results are similar, for small polygons between 10 mb and 60 mb the performance is very close together but for the larger polygons the I/O-efficient approach is significantly faster. It is hard to predict what the exact results of a fully implemented I/O-efficient shortest path tree would be

but the reported results of the internal memory competitor seem to be worse than a factor $O(log(N))$.

9 Conclusion

The goal of this thesis is to determine whether the proposed I/O-efficient shortest path algorithm of Agarwal et al. [2] has practical applicability. In order to determine this an experiment has been performed that compares the performance of the proposed algorithm with that of a state of the art internal memory shortest path algorithm proposed by Hershberger and Snoeyink [1]. Two components have been compared, computing a single shortest path and computing a (sparse) shortest path tree. The first component, computing a single shortest path, has a conclusive answer. For larger polygons where the main memory cannot contain both the polygon and the data structures necessary for the algorithm the I/O-efficient approach of Agarwal et al. [2] becomes significantly faster and thus has practical application. For the second component, the (sparse) shortest path tree, it is more complicated. Due to the complex implementation and many supporting algorithms that are necessary we were unable to fully implement the algorithm. Instead we have chosen to test the key concept used by the algorithm, the sparse shortest path tree which contains k shortest paths, 10 in this experiment, where the internal memory algorithm computes shortest paths to all N vertices of P. Due to this difference in what is computed we cannot give a conclusive answer which algorithm performed better. However as we have discussed in section 8 it seems like the impact of I/O operations on the internal memory algorithm for large polygons is greater than the expected increase in computation time for fully implementing the I/O-efficient approach. This makes it likely the I/O-efficient approach will outperform the internal memory algorithm in practice for large polygons and thus give it practical applicability.

References

[1] John Hershberger and Jack Snoeyink. Computing minimum length paths of a given homotopy class. *Computational geometry*, 4(2):63–97, 1994.

[2] Pankaj K Agarwal, Lars Arge, Yujin Shin, and Frank Staals. I/o-efficient algorithms for shortest path problems in simple polygons. 2019.

[3] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing i/o-efficient data structures using tpie. In *European Symposium on Algorithms*, pages 88–100. Springer, 2002.

[4] Alok Aggarwal and S Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[5] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(3):485–524, 1991.

[6] Der-Tsai Lee and Franco P Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[7] Leonidas Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert E Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2(1-4):209–233, 1987.

[8] Leonidas J Guibas and John Hershberger. Optimal shortest path queries in a simple polygon. *Journal of Computer and System Sciences*, 39(2):126–152, 1989.

[9] Leo J Guibas, Edward M McCreight, Michael F Plass, and Janet R Roberts. A new representation for linear lists. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 49–60, 1977.

[10] David Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12(1):28–35, 1983.

[11] Herbert Edelsbrunner, Leonidas J Guibas, and Jorge Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15(2):317–340, 1986.

[12] Yi-Jen Chiang, Michael T Goodrich, Edward F Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. 1995.

[13] Norbert Zeh. An external-memory data structure for shortest path queries. *Master's thesis, Friedrich-Schiller-Universität Jena, Germany*, 1998.

[14] Darren Erik Vengroff and Jeffrey Scott Vitter. I/o-efficient scientific computation using tpie. 1996.

[15] Bernard Chazelle. A theorem on polygon cutting with applications. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 339–349. IEEE, 1982.

[16] Lars Arge, Andrew Danner, and Sha-Mayn Teh. I/o-efficient point location using persistent b-trees. *Journal of Experimental Algorithmics (JEA)*, 8:1–2, 2003.

[17] Cgroups v2 documentation. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`. Accessed: 2020-13-10.

[18] The computational geometry algorithms library. `https://www.cgal.org/`. Accessed: 2020-13-10.