MASTER

I/O efficient single source shortest path algorithms for massive grid based graphs

van Maanen, W.

Award date:
2014

I/O efficient single source shortest path algorithms for massive grid based graphs

Wilco van Maanen
Supervisor: Herman Haverkort

August 1, 2014

Contents

Abstract

This report addresses the problem of computing single-source shortest paths for massive grid-based graphs, where n is the number of vertices in the grid and non of the edge-weights are negative. We define a simple block-based algorithm called Square . Square works well in practice and has an I/O bound of $O(\frac{n}{\sqrt{B}})$. This algorithm is based on a single-source shortest paths algorithm from Henzinger et al, which runs in $O(n \cdot \log\log n)$ time in the RAM model. Both of these algorithms are compared with Dijkstra's algorithm and with a separator-based algorithm from Hazel et al.

The separator-based algorithm from Hazel et al runs in $O(n \cdot \sqrt{R} \cdot \log n)$ time, where R is the chosen size for the regions. If $R = B^2$ then the optimal I/O bound is achieved. We introduce two improvements for Hazel et al's separator-based algorithm. For the first improvement a topology tree is used to achieve a running time of $O(n \cdot \log n)$. For the second improvement two levels of priority queues are used to ensure that the priority queue always fits in memory as long as $c \cdot M > B^2$ holds for some $c > 1$.

Our new block-based algorithm and the $O(n \cdot \log\log n)$-time algorithm from Henzinger et al work well even if the input data set is several times larger than available memory. Hazel et al's separator-based algorithm scales well and has a better I/O bound of $O(sort(n))$ or with our improvements $O(\frac{n}{B})$, but still does not perform nearly as well as the Square algorithm due to constant factors for data sets up to 6 Gigabytes on a machine of 384 Megabytes of memory.

We introduce a new property for graphs called the augmented highway dimension. This definition is based on the highway dimension, which is a property that some realistic graphs might have. But even graphs with a constant highway dimension do not have a better I/O bound then worst case graphs. For graphs with a constant augmented highway dimension however we were able to proof an I/O bound of $\Theta(\frac{n}{B})$ for the Square algorithm. The I/O bounds of the other algorithms were not improved even if the input graph has a constant augmented highway dimension.

1 Introduction

Efficiently solving single source shortest paths problems on massive data sets is a key challenge in a wide spectrum of different areas, such as geographic information systems, finding shortest paths in road networks and as a part of other algorithms. The standard algorithms for computing single-source shortest paths are not made to handle massive data sets which do not fit in memory and thus, for these data sets, the number of hard disk read and write operations (I/O) becomes the bottleneck.

The single source shortest path problem, SSSP, is the problem where given a weighted graph G without negative weights and a source vertex s we would like to find all minimal distances from s to all other vertices in

$G = (V, E)$. Let $v \in V$ be a vertex and $P \subset E$ be a path from s to v then the distance from s to v along this path is:

$$d(v) = \sum_{e \in P} w(e)$$

The minimal distance denoted by $d_{\min}(v)$ is defined as the minimal $d(v)$ if all possible paths, P, from s to v are tried.

In this paper we will deduce theoretically and experimentally why certain algorithms, Dijkstra's [5] and Henzinger et al's [9], do work well in memory, but fail in the case of massive data. We will deduce when these algorithms do still work and why this is the case. For all theoretically and experimentally results we will limit ourselves to the domain of grid graphs with non-negative edge weights. Moreover we will introduce and analyse a new algorithm, Square , which we expect to work well for massive data. In order to give a proper comparison we will also compare all algorithms with a previously developed I/O optimal algorithm, Terracost, which is defined by Hazel et al [8].

1.1 Model of computation

Most algorithms are made in such a way that they try to minimize the number of CPU operations made, but there are other factors to be considered. Such as the amount of swapping done by the algorithm. This swapping occurs once the amount of used memory is greater then the amount of available memory and the hard drive has to be used to store some amount of the required data. Since the hard drive is relatively slow compared to internal memory it would be smart to minimize the amount of swapping which is needed while running an algorithm. This becomes especially a factor once data sets are much larger then the memory itself and most of the time is wasted on reading and writing to and from the hard drive. In order to reason about this we calculate the number of I/O operations in O or Θ notation needed by the algorithm.

It is important to note here that the standard paging policy applied by most standard operating systems is the least-recently-used policy (LRU), which, once a page has to be written back to the hard drive, will always pick the least-recently-used one. This policy requires at most a factor two more I/Os and memory then the optimal paging policy, which means that in O notation there is no difference as long as the total available memory is also doubled. From a practical viewpoint it seems reasonable. We refer to Sleator et al [17] for a proof and further information about this subject.

In order to reason about the number of I/Os two variables are introduced:

- M, which defines the total amount of available memory (RAM).

- B, which defines the block size on the hard drive. This block size tells us how much data can be read or written to or from the hard drive each time it is accessed.

Then there are two I/O models to be considered, where the first is the cache-oblivious model and the second is the cache-aware model. When using the cache-oblivious model the values of M and B are not known by the algorithm, while for the cache-aware model they are. Within this paper for all our proofs we assume that the cache-aware model is used. Further information on the cache-oblivious model is found in Frigo et al [6] and for the cache-aware model we refer to Aggarwal and Vitter [2].

1.2 Literature study

Haverkort [7] introduces several algorithms, which have good I/O bounds. —In particular for the single source shortest paths problem an algorithm is presented which requires $O(scan(n) + sort(\frac{n}{\sqrt{M}}))$ I/Os, where M is the total amount of available memory. The algorithms for breadth first search traversal, creating minimum-spanning trees and topological sorting are solved with $O(scan(n))$ I/Os for grid graphs.

An in memory optimal algorithm which solves the SSSP problem in linear time is defined in the paper of Henzinger et al [9], where a theoretical linear time algorithm is presented for planar graphs and a more practical $O(n \cdot \log \log n)$ algorithm is presented as well. But there has been little to no investigation on how this algorithm described in Henzinger et al [9] works on massive data sets, while it already creates regions within the graph and minimizes the number of vertices on the boundary of a region, which is one of the general ways to tackle I/O.

In Maheshwari and Zeh [11] an I/O optimal separator based algorithm is presented for solving the single source shortest paths problem on planar graphs. In order to get a better CPU bound this algorithm uses Henzinger et al [9] Linear time algorithm for solving SSSP. Besides this they also derive an I/O optimal breadth-first search and depth-first search algorithms.

The paper of Hazel et al [8] gives us another approach, TerraCost, for solving single sources shortest paths problems for planar grid based graphs. The TerraCost algorithm defined in Hazel et al [8] is a separator based algorithm. This algorithm is I/O optimal, but has a CPU bound of $\Theta(n \cdot \sqrt{R} \cdot \log n)$, where R is the size of regions and in order to have an I/O bound of $O(sort(n))$ we need $R = B^2$. Where B is the block size of the hard drive. It is important to understand here that in order to achieve this I/O bound we need a rather large R, which thus means that the running time becomes large. Which in practice means that a balance needs to be found between having a large R to achieve the proper I/O bound and a small R to have a smaller CPU bound.

In Klein [10] we are introduced to an algorithm, which efficiently solves the multiple-source shortest paths problem in planar graphs. The algorithm takes $O(n \cdot \log n)$ time to construct a data structure, which supports queries which solve a distance query between any vertex on the boundary of the infinite face and any other vertex. The infinite face for a grid graph would be the vertices on the outside of the grid. Any query takes $O(\log n)$ time to complete. The interesting part for us here is that with this algorithm one can lower the CPU bound of the TerraCost algorithm defined in Hazel et al [8] to $\Theta(n \cdot \log n)$. This data structure which allows this improvement is an topology tree, which is defined in depth in papers Sleator et al [16] and Sleator et al [15].

In Arge et al [4] we are revisited to an I/O efficient solution to the single source shortest paths problem on planar graphs. They present an algorithm which achieves a $O(sort(n))$ I/O bound and requires $O(n \cdot \log n)$ time to complete. The algorithm presented in Arge et al [4] uses the framework presented in Klein [10] in order to give an extension of computing simple cycle separators that partition the graph into more then two pieces as presented in Miller [13]. This extension is then used to reduce the running time of the algorithm, while maintaining the I/O bound of $O(sort(n))$.

In Meyer and Zeh [12] an I/O efficient single-source shortest paths algorithm for undirected graphs is presented, with an I/O bound of $O(\sqrt{\frac{n \cdot m \cdot \log L}{B}} + MST(n, m))$, where m is the number of edges and edges have lengths between 1 and L. $MST(n, m)$ is the I/O cost related to building a minimum spanning tree.

2 Input

In this section we will explain what graphs are, which graphs are used, how they are stored, what regions are, what a highway dimension is and what the augmented highway dimension is.

2.1 Grid graph

A graph, G, is a tuple of a set vertices, V, and a set of edges, E, and is often denoted as $G = (V, E)$. Each $v \in V$ represents a point, while each $e \in E$ represents a connection between two vertices in V. Hence $e = (u, v)$, where $u, v \in V$ and $e \in E$.

If the graph is undirected then $(v, u) \in E \implies (u, v) \in E$ and if the graph is directed then $(v, u) \in E$ does not imply $(u, v) \in E$.

In addition to the standard graph it is possible to have a weighted graph, where each edge $e \in E$ also has a value connected to it. We denote this value with:

$$w(e)$$

A path P within a graph is a set of edges in E where $(v, u) = P_i$ and $(u, o) = P_{i+1}$ holds for all $0 \leq i \leq |P| - 1$.

Grid Graph A regular grid graph, also called grid graph or just grid, is a graph where all the vertices can be put next to each other with equal distances and none of the edges in the graph will cross an example of this can be seen figure 1. As can be seen in figure 1 each vertex has two, three of four ingoing and outgoing edges. The total number of vertices in the grid is n. In this paper we also assume that the number of vertices along both sides of the graph is $\sqrt{n}$ and that $\sqrt{n}$ is a power of 2. The total number of edges is $4 \cdot 2 + 3 \cdot (\sqrt{n} - 2) + 4 \cdot (\sqrt{n} - 2)^2 \leq 4 \cdot n$.

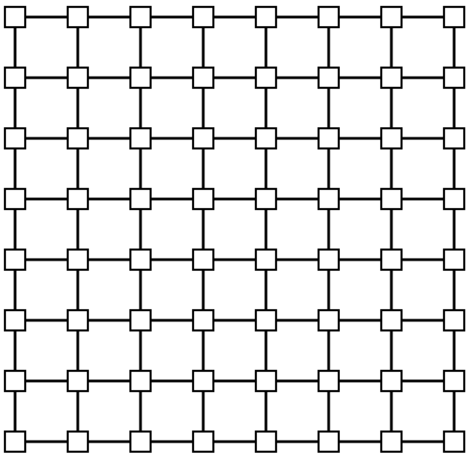


Figure 1: A figure of a grid graph, where the rectangular objects are the vertices and the lines between them are the edges in the regular grid graph.

2.2 Storage

In this section the details on how the grid is stored will be discussed. With a vertex we will always store all its outgoing edges, but the order in which the vertices can be stored can be different. This is the case because in memory there is only one dimension, while in the grid there are two (namely left to right and top to bottom). Differently put most vertices have four neighbors, while in memory each element can only have two. We discus three ways to solve this problem in the following subsections.

2.2.1 Row and column order

Row and column order is the standard way of storing the input data and usually are the way in which the input data is given. Figures 2 and 4 give an example of two grids where one is stored in row order and the other is stored in column order.

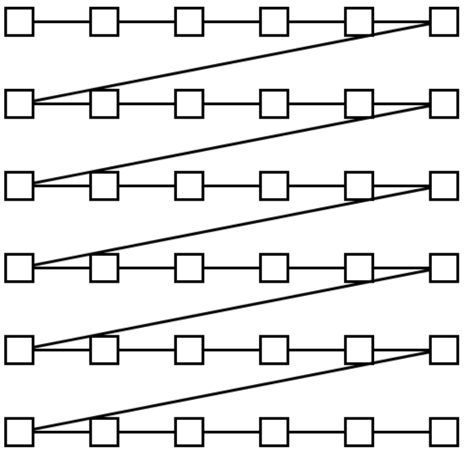


Figure 2: A grid stored in row order. The edges between the vertices represent the order in which they are stored.

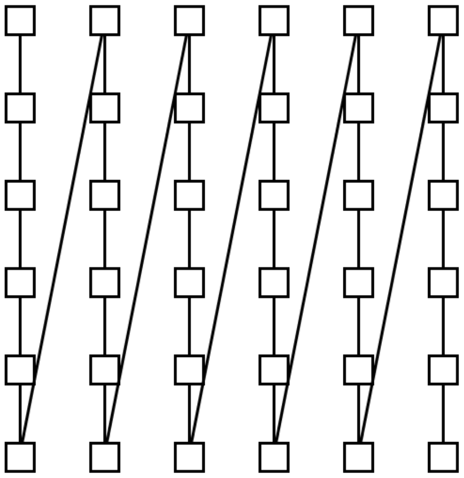


Figure 3: A grid stored in column order. The edges between the vertices represent the order in which they are stored.

2.2.2 Z-order

Another way of storing the data is in Z-order. An example of this ordering is giving in figure 4. The idea behind this is that vertices which have a small number of edges in the graph between them should on often also be close to each other in the input file. The reason why this is relevant for us is because we are trying to minimize the number of made I/Os, hence applying Z-order instead of the standard row or column orderings might be a way to do this. We refer to Morton [14] for more information on the Z-order curve and storing data with this ordering.

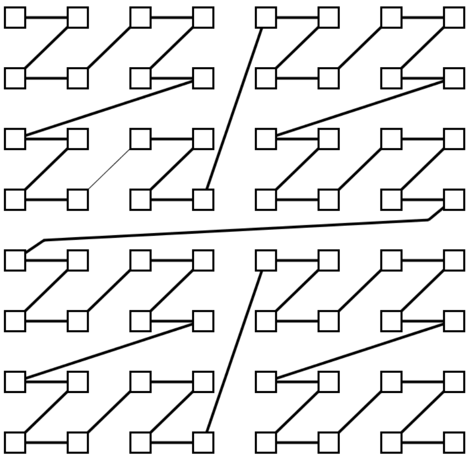


Figure 4: A grid stored in Z-order. The edges between the vertices represent the order in which they are stored.

2.3 Regions

In order to properly understand the descriptions of the algorithms presented below it is important to understand what a region is. A region is sub part of the total graph. Such a region can be a square containing for example a 4×4 part of the graph, but it can also take other forms. See figure 5 for a standard 8×8 grid graph. Figure 6 shows a grid graph with several 8×1 regions and figure 7 shows a grid graph with several 4×4 regions.

Each region has some number of boundary vertices, where a boundary vertex is vertex which has an edge going from its region into another region. As can be seen in 6 each region here has 8 boundary vertices and in 7 each region has 7 boundary vertices. In general using a 4×4 sized region each region would have 12 boundary vertices. For our algorithms it is important to minimize the number of boundary vertices for a given R in order to get nice I/O bounds and nice CPU bounds. R here denotes the number of vertices inside a region. Since we have regular grid graphs we know that the minimum number of boundary nodes is $\Theta(\sqrt{R})$ given that R is not close to n or 1. This implies that a square of size $\sqrt{R} \times \sqrt{R}$ is a good way to achieve this. Figure 7 is an example of this with $R = 16$.

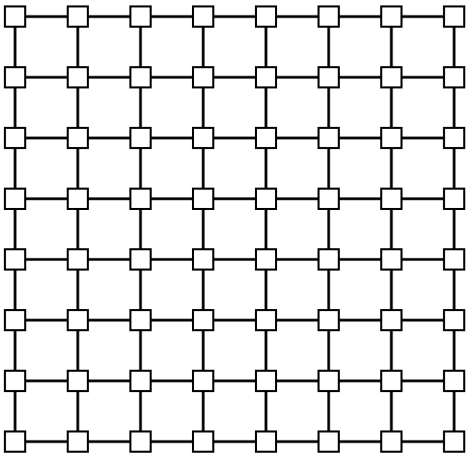


Figure 5: A grid graph without defined regions.

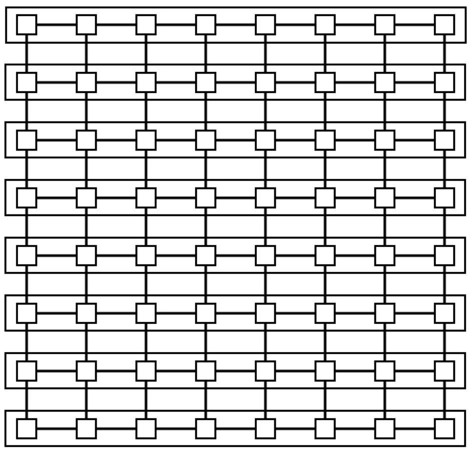


Figure 6: A grid graph with rectangular regions.

2.4 Highway dimension

From Abraham et al [1] we get the concept of the highway dimension, which applies to real road networks and other real data sets. We considered using

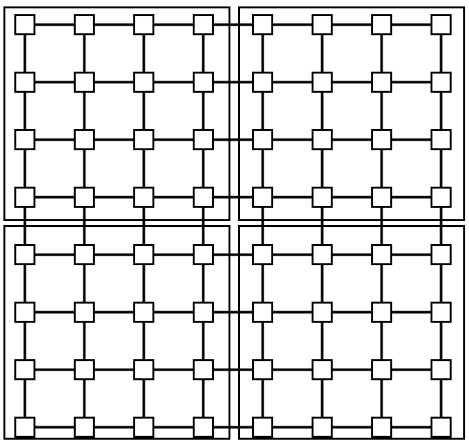


Figure 7: A grid graph with 4×4 square regions.

the highway dimension to get nice I/O bounds for a sub-set of real world data sets.

The highway dimension is defined as follows:

$$\forall r \in \Re^{+}, \forall u \in V, \exists S \subseteq B_{u,4\cdot r} : |S| \leq h$$

such that:

$$\forall v, w \in B_{u,4\cdot r} : (|P(v,w)| > r \wedge P(v,w) \subseteq B_{u,4\cdot r}) \Rightarrow (P(v,w) \cap S \neq \emptyset)$$

Where:

- $P(v,w)$ is the shortest path from v to w.
- $|P(v,w)|$ is the sum of all edges in $P(v,w)$.
- $B_{u,r} = \{v \| \in V \wedge |P(u,v)| \leq r\}$

If h is constant then the highway dimension is constant.

2.5 Augmented highway dimension

In section 2.4 we have introduced the concept of highway dimension, but since this might not be enough to get an improvement on the I/O bound we introduce the concept of augmented highway dimension.

The augmented highway dimension is defined as follows:

$$\forall r \in \Re^{+}, \forall u \in V, \exists S \subseteq B_{u,4\cdot r} : |S| \leq h$$

Such that:

$$\forall v, w \in B_{u,4\cdot r} : (|D(v,w)| > r \wedge P(v,w) \subseteq B_{u,4\cdot r}) \Rightarrow (P(v,w) \cap S \neq \emptyset)$$

Where:

- $P(v, w)$ is the shortest path from v to w within the weighed graph.
- $D(v, w)$ is the shortest path from v to w within the unweighed graph.
- $|D(v, w)|$ is the number of edges in path $D(v, w)$.
- $B_{u,r} = \{v \| v \in V \wedge |D(u, v)| \leq r\}$

If h is constant then the augmented highway dimension is constant.

3 Algorithms

We have implemented several algorithms such that several known algorithms can be tested against new or untested algorithms in the field of I/O and massive data. The chosen algorithms are listed below:

- Dijkstra's algorithm, which is a widely used SSSP algorithm, but for which it is known to have a worst case $\Theta(n)$ I/O bound. The running time of Dijkstra's algorithm is $\Theta(n \cdot \log n)$. Dijkstra's algorithm is defined in Introduction to Algorithms [5].
- Quasi-linear , which is based on a linear time SSSP algorithm defined in Henzinger et al [9]. The version which was implemented is more practical and has an CPU bound of $O(n \cdot \log(\log(n)))$. We will show that the worst case I/O bound for this algorithm is $\Theta(\frac{n}{\sqrt{B}})$.
- Square , which is a newly developed algorithm and is similar to Quasi-linear , but is made such that it should be optimized for I/O. The CPU bound of Square is $\Theta(n \cdot \sqrt{R} \cdot \log n)$ and the I/O bound is $O(\frac{n}{\sqrt{B}})$.
- TerraCost, which is a separator based approach developed by Hazel et al [8] and which has a theoretical near-optimal I/O bound of $O(sort(n))$. The CPU bound of this algorithm is $\Theta(n \cdot \sqrt{R} \cdot \log n)$. We introduce two improvements, which lower CPU bound to $\Theta(n \cdot \log n)$ and the I/O bound to $\Theta(\frac{n}{B})$.

3.1 Priority queue

Dijkstra's algorithm and similar algorithms all use a priority queue, which is a simple data structure which maintains the minimal value among the values stored inside it. The functions it must support are:

- Push x with key k into pq

    - k is the key value on which the minimum or maximum is determined.
    - x is the satellite-data which in our case points to the vertex or region it is related to.
    - pq is the priority queue into which (x, k) is inserted.

- $x := pq.extract\_minKey()$
    - x is the satellite-data which in our case points to the vertex or region it is related to. The queue guarantees that x has he minimal key value among all key values in the queue and that x is removed from the priority when it is returned.
    - pq is the priority queue into which (x, k) is inserted.

Besides this it is required that each vertex or region, denoted in the functions with x, is only stored once inside the priority queue. Meaning that each element inside the queue has a unique satellite-data identifier. If an element is pushed with the same identifier, x, as one already inside the queue, then only the one with lowest key value is stored.

For more details on how to implement this we refer to Introduction to Algorithms [5] and for a detailed explanation on how to make such a structure I/O efficient we refer to Arge et al [3].

3.2 Dijkstra's Algorithm

Dijkstra's algorithm, which is explained in Introduction to Algorithms [5]. Dijkstra's algorithm given a source and a graph with non-negative edge weights, solves the single source shortest path problem in $\Theta(n \cdot \log n)$ time. This CPU bound is good, since it is only a $\log(n)$ factor from the optimum $\Theta(n)$. Moreover the algorithm is easy to implement and it is widely used, but the algorithms I/O behavior is suboptimal. It can be proven that the I/O bound is $\Theta(n)$ even if an I/O efficient priority queue is used.

3.2.1 Description

In this section Dijkstra's algorithm will be described and the concept of relaxing edges will be discussed. The pseudo code for Dijkstra's algorithm:

```
   $Dijkstra.Process(G, s)$

   Comment: $G$ is the input grid.

   Comment: $s$ is the source vertex.

P1 For all $v \in G$ let $d(v) := \infty$

P2 Let $pq$ be an I/O efficient priority queue.
```

```
P3 Push $s$ with key 0 into $pq$.
    P4 While $pq$ is not empty do:
        P5 $v := pq.extract\_minKey()$
        P6 For each outgoing edge $(v, w)$ of $v$ do:
            P7 If $d(v) + length(v, w) < d(w)$ then
                P8 $d(w) := d(v) + length(v, w)$
                P9 Push $w$ with key $d(w)$ into $pq$
```

The algorithm will continue as long as there is an element in the priority queue, pq. Once pq is empty we know that all vertices have been relaxed and all $v \in V$ have their final lowest value, $d_{min}(v)$. Lines P7 and P8 entail the relaxation of an edge (v, w), whereby a new and lower value for $d(w)$ is found if P7 yields true. It is possible that this new found $d(w)$ is also $d_{min}(w)$. Whenever a vertex, x, is extracted from the priority queue, then we know that $d(x) = d_{min}(x)$ since there are no non-negative edges and $d(x)$ is the lowest value in the priority queue. Hence whenever a vertex is extracted from the priority queue we know it will have its final value and it won't be pushed into the priority queue again.

3.2.2 Running time (CPU)

The running time for Dijkstra's algorithm is $\Theta(n \cdot \log n)$, where the $\log n$ factor is related to the size of the priority queue. For a proof of this we refer to Introduction to Algorithms [5].

3.2.3 I/O-efficiency in the worst case

In general Dijkstra's algorithm is not I/O efficient and it can be proven to need $\Theta(n)$ I/Os even when using an I/O efficient priority queue. There are data sets however where one would expect Dijkstra's algorithm to need a lot less I/Os then $\Theta(n)$.

3.2.4 I/O-efficiency in grids with constant highway dimension

The highway dimension will not improve the I/O bound for Dijkstra's algorithm for a proof of this we refer to 3.4.4. The fact that Dijkstra's algorithm does not use regions will not invalidate this proof.

3.2.5 I/O-efficiency in grids with constant augmented highway dimension

The augmented highway dimension will not improve the I/O bound for Dijkstra's algorithm for a proof of this we refer to 3.4.5. The fact that Dijkstra's algorithm does not use regions will not invalidate this proof.

3.3 I/O-efficiency in grids with random weights

For some graphs the I/O bound of Dijkstra's algorithm might be better then $\Theta(n)$ I/Os.

Lets assume we have a given grid which is fully random with weights between zero and one then intuitively we would expect that the algorithm slowly moves outward from the source vertex, which means that you would expect the algorithm to go by the vertices as Breadth first search would (BFS). Figure 8 visualizes several intermediate results for an experiment with Dijkstra's algorithm, where all the weights are random. The figure shows that even though Dijkstra's algorithm does not behave exactly like BFS would it still enforces the idea that the number of grey vertices is limited, where the grey vertices are the vertices in the priority queue. Given a proper grid and using the idea above we would expect that the number of vertices in the priority queue should be at the most $O(\sqrt{n})$. In section 4.1.3 we present some additional experimental results which support this idea.

Theorem If the number of vertices in the priority queue is at most $O(\sqrt{n})$, the grid is stored in Z-order and $M = B^2 > c \cdot \sqrt{n} \cdot B$ holds for a $c > 1$ then at the most $O(\frac{n}{\sqrt{B}})$ I/Os are used.

Proof Whenever a vertex, v, is taken from the priority queue and its neighboring vertices are relaxed then we know that the number of edges between v and its neighbors is at most one.

Since the grid is stored in Z-order it takes at least $\sqrt{B}$ Dijkstra steps to go from any vertex v to a block which is not adjacent to block Q_v. In figure 9 an example of this is given, where v is the black vertex, u is the grey vertex and the dotted blocks are the blocks adjacent to Q_v. Hence as long as Q_v and its adjacent blocks stay in memory then we know that we can do at least $\sqrt{B}$ Dijkstra steps before we need to use an I/O.

Since it is possible to have a maximum of $O(\sqrt{n})$ vertices stored in the priority queue and we know that Z-order storage is used then loading in any vertex and its neighbors takes at most $O(1)$ I/Os. Now let a vertex, v, be taken from the priority queue and let v be on the boundary of block T_v on disk. If we now assume we try to reach a block which is not adjacent to T_v then this would require at least $\sqrt{B}$ steps from the initial vertex v. So if the size of the priority queue is at most $O(\sqrt{n})$ and $O(\sqrt{n})$ blocks and their adjacent blocks fit in memory then we are able to relax at least $\sqrt{B}$ vertices before we need to use one I/O. We also know that once we use one I/O to load in the next block and its adjacent neighbors that again we will be able to do at least $\sqrt{B}$ Dijkstra steps before we need to use a new I/O. Thus each I/O yields at least $\sqrt{B}$ Dijkstra steps and also $\sqrt{B}$ vertices get their final value.

Using the above we can calculate a new I/O bound:

$$O(1) \cdot \frac{n}{\sqrt{B}}$$
$$=$$
$$O(1 \cdot \frac{n}{\sqrt{B}})$$
$$=$$
$$O(\frac{n}{\sqrt{B}})$$

We still need to show that we have enough memory to hold all the blocks in memory such that $O(\frac{n}{\sqrt{B}})$ I/Os are needed at the most. Let $c > 1$ then we have:

$$M > c \cdot \sqrt{n} \cdot B = O(\sqrt{n} \cdot B)$$

Let $M = B^2$ then we get:

$$M = B^2 > c \cdot \sqrt{n} \cdot B = c \cdot \sqrt{n} \cdot B$$
$$\Longrightarrow$$
$$\frac{B}{c} > \sqrt{n}$$

We now have an I/O bound of $O(\frac{n}{\sqrt{B}})$ I/Os under the assumptions that $\frac{B}{c} > \sqrt{n} \Longrightarrow M = B^2 > c \cdot \sqrt{n} \cdot B$ holds and that the grid is stored in Z-order. This is exactly what the theorem said thus proven.

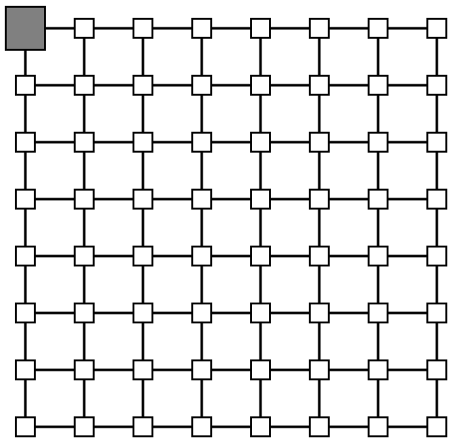
$$\blacksquare$$

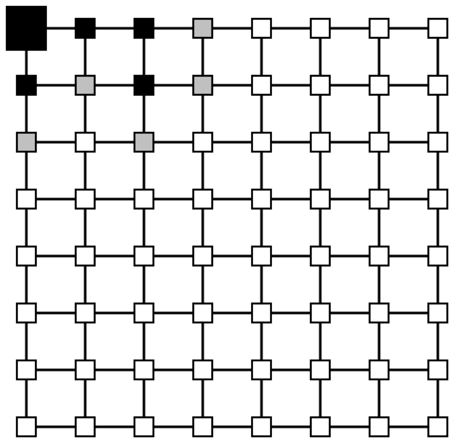
Remarks If the grid is stored in row order then the I/O bound of $O(\frac{n}{\sqrt{B}})$ will not be achieved since:

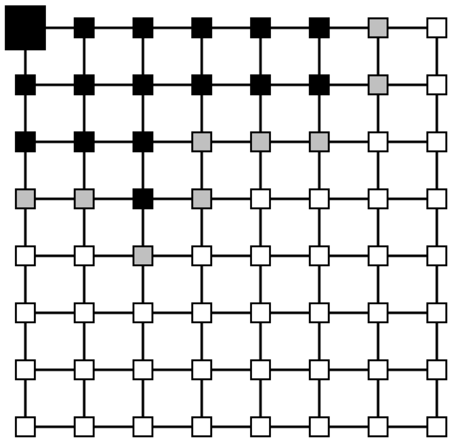
Given a vertex v which is extracted from the priority and a grid which is stored in row order, then generally two of the four neighbors of v will be stored in a different block then v. Let u be one of those two neighbors, where $B_u \neq B_v$. Let u be added to the priority queue and $d(u)$ be the minimal value in the priority queue. Now since $d(u)$ is the lowest value in the priority queue and u is selected next, then generally at least one neighbor of u is in a different block on disk then u and then v, which means that each vertex selected from the priority queue might require an I/O even if vertices neighboring each other in the grid are selected in order.

If $\frac{M}{c} > \sqrt{n} \cdot B \Longleftrightarrow False$ then depending on the order in which vertices are added and extracted from the priority queue we will again use one I/O per dijkstra step and get an I/O bound of $O(n)$.

Thus all three assumption in the theorem are necessary. For some extra information and some experimental results related to the assumptions required to let this theorem work we refer to section 4.1.3


(a) Starting situation with the source vertex top left


(b) After five steps of the algorithm


(c) After 16 steps of the algorithm

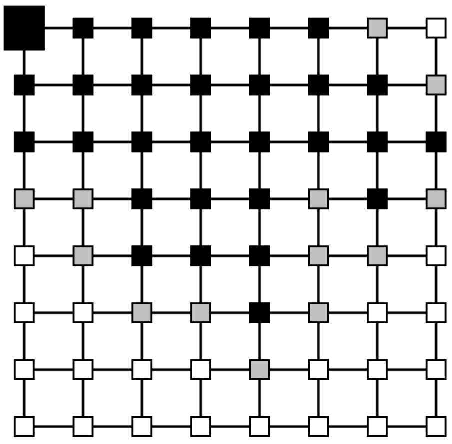

(d) After 29 steps of the algorithm

Figure 8: A way Dijkstra's algorithm could behave on a grid with random edge weights between zero and one. The bigger vertex top left is the source. Grey vertices are currently in the priority queue, black vertices are done and have their lowest possible value and white vertices have not been accessed yet.

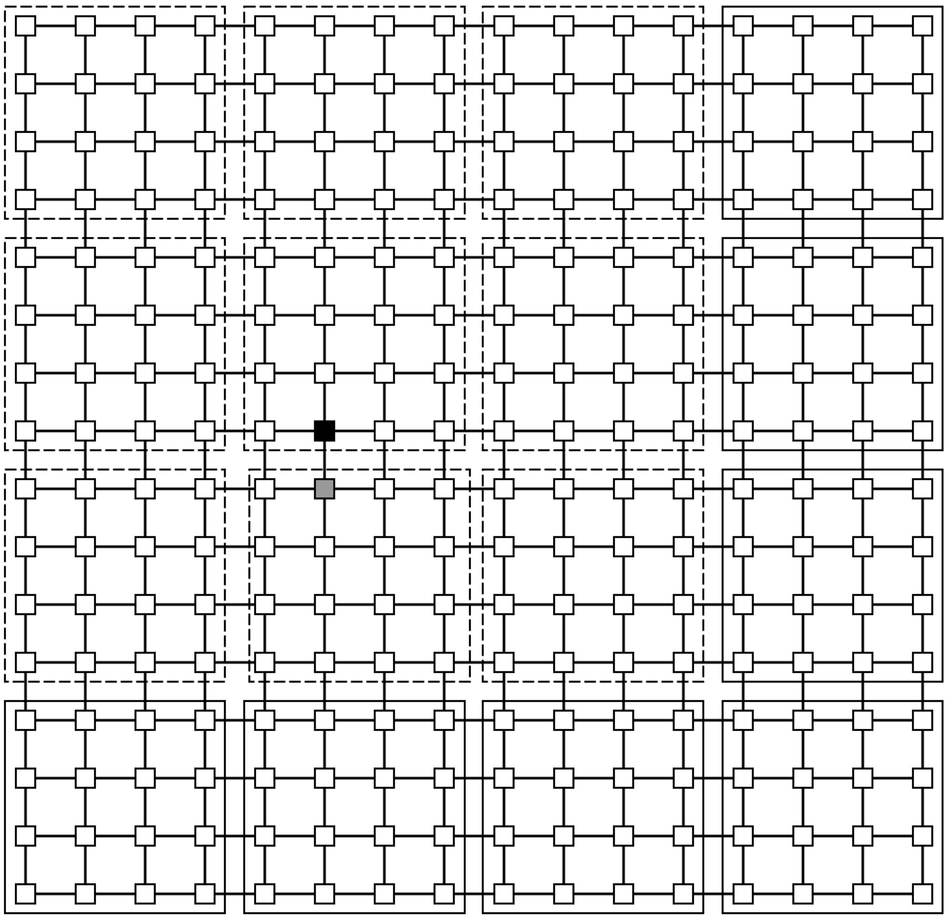


Figure 9: A 16 by 16 grid stored in Z-order with $B = 16 = 4^2$.

3.4 Quasi-linear

The linear time algorithm is a single source shortest path algorithm which theocratically achieves a running time of $O(n)$ CPU operations for the specifics we refer to Henzinger et al [9]. Besides this optimal linear time algorithm they also propose another more practical algorithm with a running time of $O(n \cdot \log(\log(n)))$. This second algorithm is the one we decided to implement and test against the other algorithms in this paper.

In section 3.4.3 we will explore in depth how I/O efficient the algorithm is in the worst case. For the CPU bound and the exact description of algorithm we refer to Henzinger et al [9].

3.4.1 Description

For the specifics on the Quasi-linear algorithm we refer to Henzinger et al [9], but we will outline the general idea in this section and why we think this would also work well in an I/O setting.

The idea behind the algorithm is that it chops the grid up into regions, where each region has at most $O(\sqrt{R})$ boundary vertices and then one priority queue is maintained for the regions and a set of priority queues is maintained for the regions (one queue for each region). In each iteration the region with lowest key value is selected from the main priority queue and α Dijkstra steps are performed on this region. Every time a Dijkstra step is performed the related queues are updated and this process continues until the main queue is empty and thus also all the queues for each region is empty.

The idea here was that this algorithm creates regions and does some amount of work within each region before it continues with another region. Hence when a proper R and α would be chosen, then one would expect it to be I/O efficient, at least to some extent.

3.4.2 Running time (CPU)

The running time of the Quasi-linear algorithm is $O(\log \log n)$, which is very close to the optimum of $O(n)$. For a proof on this we refer to Henzinger et al [9].

3.4.3 I/O-efficiency in the worst case

In this section we will prove that the I/O bound will be $\Theta(\frac{n}{\sqrt(B)})$ if the input is stored in Z-order and $\Theta(n)$ if the input is stored in column or row order. This will be done by first proving that an upper bound and secondly giving an example which yields the lower bound. Hence resulting in the tight bound given here.

Let each region have size R, then there are $O(\sqrt{R})$ boundary vertices and let $\alpha = \infty$. Meaning that the entire region will always be done completely.

Whenever a lowest element is extracted from the main priority queue and α Dijkstra steps are performed on the corresponding region $(4+1) \cdot \Theta(x + \frac{R}{B})$ I/Os are used. $\Theta(x + \frac{R}{B})$ is the number of I/Os needed to read one region and additional data structures into memory. The total number of regions which might have to be accessed is the region itself and its 4 adjacent neighboring regions. The factor x denotes the minimal number of I/Os needed to access any region. If the grid is stored in row or column order then $x = \sqrt{R}$ and if the grid is stored in Z-order then $x = 1$.

Theorem (upper bound) Let $R = B$ and $M > c \cdot \frac{n}{B}$ for $c > 1$, then the I/O upper bound will be $O(\frac{n}{\sqrt{(B)}})$ if the input is stored in Z-order and $O(n)$ if the input is stored in column or row order.

Proof (upper bound) The sub-queues from each of the regions only contain elements on the boundary of that region. This is the case because when a region, q, is extracted from the main priority queue, then α Dijkstra steps are applied to that block. Since $\alpha = \infty$ all vertices in the block are relaxed and the sub-queue of q is empty. The sub-queue's in regions neighboring q can still contain unrelaxed vertices, but the added or changed vertices in these sub-queue's can only be boundary vertices. Hence each sub-queue only contains boundary vertices of the corresponding region, which means that each region in the main queue is only in the main queue, because some boundary vertex of that region has not been relaxed yet.

Besides this in essence the algorithm simply runs Dijkstra and each region extracted from the main queue has a vertex which is the lowest possible at that time. Since there are no negative edges we know that vertices cannot be made any lower, hence once this region is extracted and the lowest vertex in this region is relaxed we know it will not be relaxed again.

This implies that each region is extracted at most once from the main queue for each boundary vertex it contains. Resulting in $O(\sqrt{R} \cdot \frac{n}{R})$ or less times that a region is extracted from the main queue.

Let $M > O(\frac{n}{R})$, meaning that the main priority queue fits in memory at all times.

We can combine the previous parts to calculate the total number of I/Os needed to run the algorithm:

$$O(\sqrt{R} \cdot \frac{n}{R}) \cdot O(x + \frac{R}{B})$$

$$=$$

$$O(\sqrt{R} \cdot \frac{n}{R} \cdot (x + \frac{R}{B}))$$

$$=$$

$$O(x \cdot \sqrt{R} \cdot \frac{n}{R} + \sqrt{R} \cdot \frac{n}{B})$$

$$=$$

$$O(n \cdot (\frac{x}{\sqrt{R}} + \frac{\sqrt{R}}{B}))$$

From this equation it is already visible that if row or column order are used and $x = \sqrt{R}$ that then: $O(n \cdot (\frac{x}{\sqrt{R}})) = O(n \cdot (\frac{\sqrt{R}}{\sqrt{R}})) = O(n)$ I/Os are needed. Thus we assume Z-order is used and $x = 1$.

$$O(n \cdot (\frac{1 \cdot B + R}{\sqrt{R} \cdot B}))$$

$$=$$

$$O(n \cdot (\frac{B + R}{\sqrt{R} \cdot B}))$$

When $B, R > 1$, then the minimum for $\frac{B+R}{\sqrt{R} \cdot B}$ is found if $B = R$.

$$O(n \cdot (\frac{B + B}{\sqrt{B} \cdot B}))$$

$$=$$

$$O(n \cdot (\frac{2}{\sqrt{B}}))$$

$$=$$

$$O(\frac{n \cdot 2}{\sqrt{B}})$$

$$=$$

$$O(\frac{n}{\sqrt{B}})$$

Hence the upper bound for a grid which is stored in row and column order is $O(n)$ I/Os and the upper bound for a grid which is stored in Z-order is $O(\frac{n}{\sqrt{B}})$.

In order to complete this part we still need to show that the assumption that the main priority queue fits in memory is fair. Since $B = R$ was chosen we know that the size of the main queue is at most $O(\frac{n}{B})$, which means that: $M > c \cdot \frac{n}{B}$ must hold for some large enough $c > 1$. Hence this assumption should be no problem.

■

Theorem (lower bound) A lower I/O bound will be $\Omega(\frac{n}{\sqrt{(B)}})$ if the input is stored in Z-order and $\Omega(n)$ if the input is stored in row or column order.

Proof (lower bound) We will proof this lower bound by giving an example where the lower bound of $\Omega(\frac{n}{\sqrt{B}})$ is encountered:

Assume we have an undirected graph where $v_{i,j}$ is vertex i, j in the grid and $w(v, u)$ the cost to use the edge between v and u:

- For all i and j let $w(v_{i,j}, v_{i+1,j}) = 0$.
- For $i = 0$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = 0$
- For $i = 0$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = \infty$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = 0$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = \infty$
- Let the source vertex be $v_{0,0}$

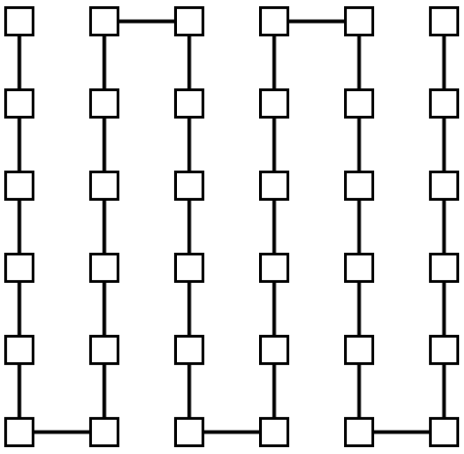


Figure 10: An undirected grid graph where the missing edges have weight ∞.

This input is very similar to the worst case input data as defined in section 4.2.1 an example is given in figure 10.

Given this input when a region is accessed at least $\sqrt{R}$ vertices are relaxed to their final value. And given a $R < n$ at most $2 \cdot \sqrt{R}$ vertices are relaxed to their final value. Which means that each region has to be accessed $\Theta(\frac{R}{\sqrt{R}})$ times.

Remember that whenever a region is accessed at least $\Theta(x + \frac{R}{B})$ I/Os are used.

Due to the way the algorithm works the path will simply be followed, hence solving one column of the input will cost the following number of

I/Os:

$$\Theta(\frac{R}{\sqrt{R}}) \cdot \Theta(x + \frac{R}{B})$$

Since there are $\Theta(\sqrt{n})$ columns, the total I/O bound becomes:

$$\Theta(\frac{n}{\sqrt{R}}) \cdot \Theta(x + \frac{R}{B})$$
$$=$$
$$\Theta(\frac{n \cdot x}{\sqrt{R}} + \frac{n}{B})$$
$$=$$
$$n \cdot \Theta(\frac{x}{\sqrt{R}} + \frac{1}{B})$$

From this equation it is already visible that if row order is used and thus $x = \sqrt{R}$ that then: $n \cdot \Theta(\frac{x}{\sqrt{R}}) = \Theta(\frac{\sqrt{R}}{\sqrt{R}}) = \Theta(n)$ I/Os are needed. Thus we assume Z-order is used and $x = 1$:

$$n \cdot \Theta(\frac{1 \cdot B + \sqrt{R}}{\sqrt{R} \cdot B})$$
$$=$$
$$n \cdot \Theta(\frac{B + \sqrt{R}}{\sqrt{R} \cdot B})$$

When $B, R > 1$, then the minimum for $\frac{B+R}{\sqrt{R} \cdot B}$ is found if $B = R$.

$$n \cdot \Theta(\frac{B + \sqrt{B}}{\sqrt{B} \cdot B})$$
$$=$$
$$n \cdot \Theta(\frac{\sqrt{B} + 1}{B})$$
$$=$$
$$n \cdot (\Theta(\frac{\sqrt{B}}{B}) + \Theta(\frac{1}{B}))$$
$$=$$
$$n \cdot \Theta(\frac{\sqrt{B}}{B})$$
$$=$$
$$\Theta(\frac{n}{\sqrt{B}})$$

Hence for this example we found that the total number of I/Os used for accessing the regions is $\Theta(\frac{n}{\sqrt{B}})$ if Z-order is used and $\Theta(n)$ if row or column is used (a similar example can be constructed for column order). This also means that these bounds are lower bounds in general.

■

Proof (tight bound) Thus in the above two theorems we have proven an upper and a lower bound which show that the number of I/Os which are needed is $\Theta(n)$ if row or column order are used and $\Theta(\frac{n}{\sqrt{B}})$ if the grid is stored in Z-order.

■

3.4.4 I/O-efficiency in grids with constant highway dimension

In this section we will show that if the input data set has a constant highway dimension h then the I/O bound will not be improved.

Theorem Given an input graph with a constant highway dimension h then the I/O bound will be $O(\frac{n}{\sqrt{B}})$.

Proof We use the following example to show that a constant highway dimension does not improve the I/O bound given in section 3.4.3.

Let us have an un-directional grid graph where $v_{i,j}$ is vertex i, j in the graph and $w(v, u)$ the cost to use the edge between v and u:

- For any i, j let $w(v_{i,j}, v_{i+1,j}) = 1$
- For $i = 0$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = 1$
- For $i = 0$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = \infty$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = 1$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = \infty$
- For $0 < i < \sqrt{n} - 1$ and any j let $w(v_{i,j}, v_{i,j+1}) = \infty$
- Let the source vertex be $v_{0,0}$

Figure 11 shows an example of the graph as it is defined above.

Now given some r and some vertex u, we can build a set S of size 16 or less where the elements in S are:
Let $S_0 = u$.
$S_i = x$, where: $\frac{r}{2} \leq |P(x, S_{i-1})| \leq r$. There are at least two possible solutions for x, but we choose the x furthest from u. Since it is possible to go in two directions from u, we assume that this is done. This means that two times at most 8 elements are added to S. Since $\frac{r}{2} \cdot 8 = 4 \cdot r$, which is the maximum range.

It is not hard to verify now that every r distance or less along the path from u outward at least one vertex in S is encountered. Which means that according to the definition of the highway dimension we have a constant highway dimension of 16 or less for this given data set.

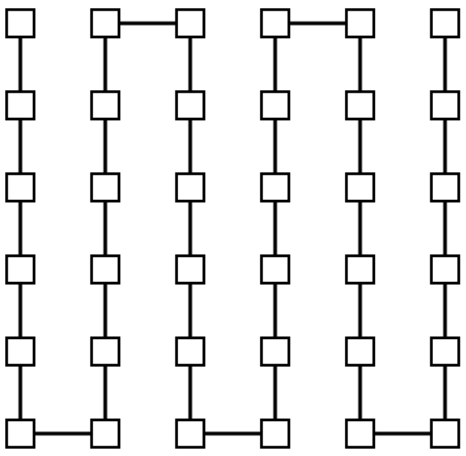


Figure 11: An un-directional grid graph where the missing edges have weight ∞ and the other edges have weight 1.

The question which remains is does this improve the I/O bound found in 3.4.3. No it does not since the data set is practically the same as the constructed data set that gives a bound of $O(\frac{n}{\sqrt{B}})$. The only difference is that the edges here along the path are 1 instead of 0 and that several vertices are not connected, but in essence this graph still has one long path which is only a constant factor smaller then n. Hence the highway dimension does not help us to find a better I/O bound.

■

3.4.5 I/O-efficiency in grids with constant augmented highway dimension

In this section we will show that a graph with an constant augmented highway dimension will not yield an improvement upon the I/O bound for the Quasi-linear algorithm.

Theorem Given an input graph with a constant augmented highway dimension h then the I/O bound will be $\Theta(\frac{n}{\sqrt{B}})$.

Proof We will proof this by taking some graph, G, which has a constant augmented highway dimension h and build a new graph, G_{new} from G which also has a constant augmented highway dimension, h_{new}. Graph G_{new} will have a lower bound on the number of I/Os of $\Omega(\frac{n}{\sqrt{R}} + \frac{n}{\sqrt{B}})$.

Let graph, G, with a constant augmented highway dimension, h, where all edge weights are in [0..1]. Now we will create a new graph, G_{new}, where we replace each vertex in G by a set of 5×5 vertices as follows:

Generate one 5×5 region, Q_i, for each vertex, $v_i \in V$, where the edges between the 16 boundary vertices in Q_i have weight zero. The other $4 \cdot 3 \cdot 3 = 36$ internal edges in Q_i have a random weight greater than $25 \cdot n$ and smaller than $100 \cdot n$. The weights for all of the five edges from Q_i to Q_j are $w((v_i, v_j))$ for any Q_i and Q_j where v_i and v_j are adjacent to each other in the original grid graph, G. We refer to figure 12 for a visualization of this transformation for a 3×3 grid graph.

Since the new grid graph, G_{new}, is larger and has new edges with random weights in it we will choose a new h:

$$h_{new} = h \cdot 25$$

By the definition of the augmented highway dimension we can state that G_{new} must also have a bounded augmented highway dimension of h_{new}. This is true because the edges between the boundary vertices of each 5×5 region are zero.

Because of the fact that R is always a power of two we can infer that $\Theta(\frac{n}{R})$ regions are breaking the internal edges within a 5×5 region Q_i. Remember here that the internal edges are the dotted edges with random weight visible in figure 12. Since these edges are random and greater then $n \cdot 25$ we know that first all vertices on the boundary of each 5×5 region will be relaxed and given their final value and only after that the internal vertices of each 5×5 region will be extracted from the priority queue. The order in which these vertices are extracted form the priority queue is random and since $\Theta(\frac{n}{R})$ regions are broken by internal edges connected to these internal vertices in each region we know that most regions might be accessed $\Theta(\sqrt{R})$ times. The reason for this is that because every time when a region boundary is crossed the region on the other side of that boundary will be added into the priority queue and this region will be extracted again. Since each access to a region requires $\Theta(1 + \frac{R}{B})$ I/Os (remember that the order is unknown) we know that at least $\Omega(\frac{n}{\sqrt{R}} \cdot (1 + \frac{R}{B})) = \Omega(\frac{n}{\sqrt{R}} + \frac{n \cdot \sqrt{R}}{B})$ I/Os are needed. Thus the I/O bound will not be improved by using the augmented highway dimension. We refer to section 3.4.3 for the general proof which also gives an upper bound on the number of I/Os for a graph with a constant augmented highway dimension.

$\blacksquare$

3.5 Square

The Square algorithm is similar to the Quasi-linear algorithm defined in section 3.4, but the α value is always set to be ∞ and instead of only processing one region at a time a 3×3 square of 9 regions is processed in its entirety. Figure 13 gives an example of this.

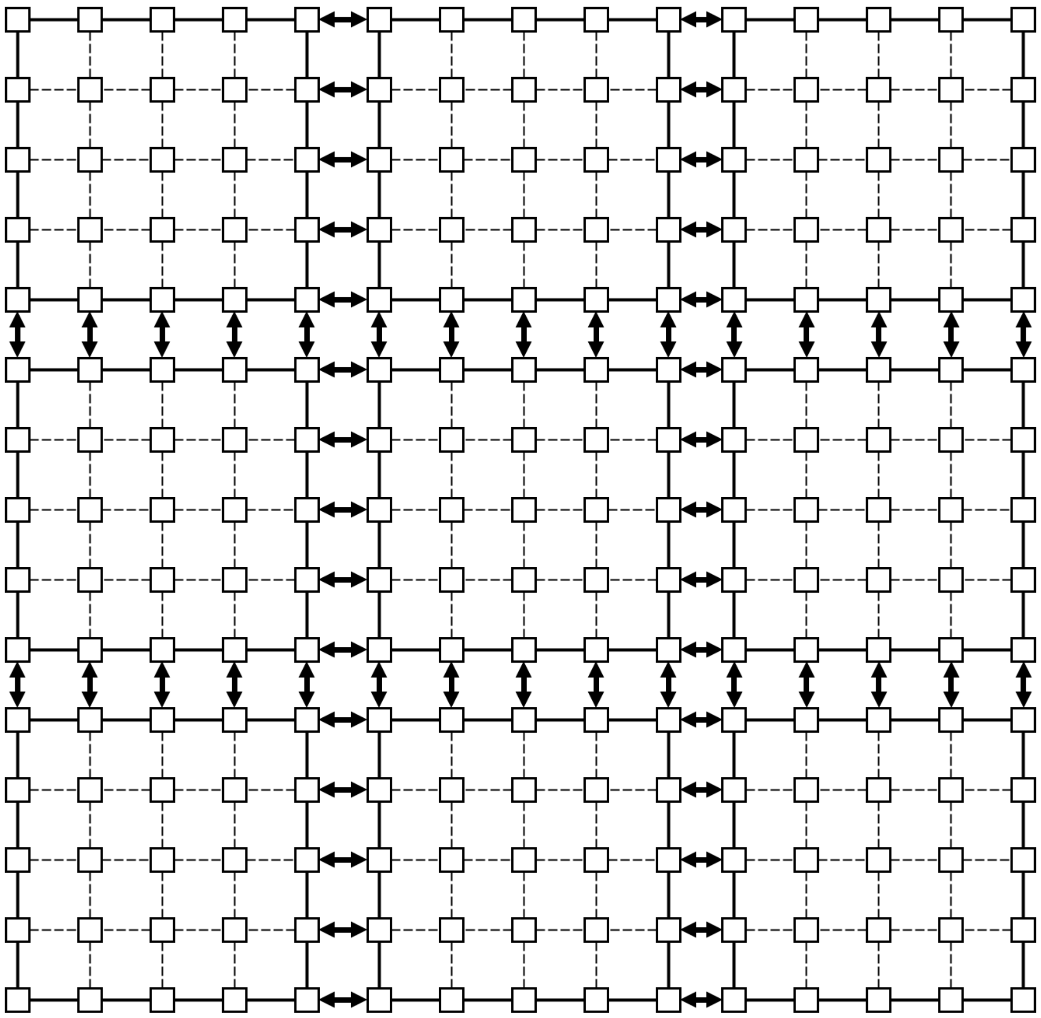


Figure 12: The resulting grid graph when a 3×3 grid graph is transformed by replacing each vertex by 5×5 region. The dotted lines have random weight, the normal lines have weight zero and the lines with arrows have the weight equal to that in the original graph.

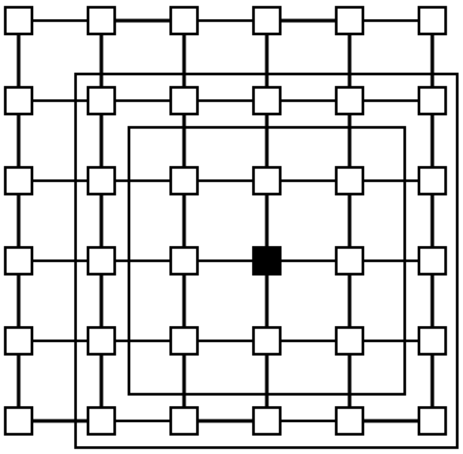


Figure 13: In this figure a graph where each small square represents one of the 36 regions. The filled square represents the region which was extracted from the main priority queue. The rectangle containing 9 squares shows the 9 regions which will be processed and the rectangle containing 25 squares shows the 25 regions which might be accessed during the sub procedure $Square.Process(T, pq, main\_pq, sub\_pq)$.

3.5.1 Description

The algorithm is defined with the help of two function in the pseudo code below. The first function $Square.Calc\_sssp(G, s)$ is the entry function which calculates the the shortest distance to each $v \in V$ this distance is stored in $d(v)$. The second function $Square.Process(T, pq, main\_pq, sub\_pq)$ is a sub function and processes a 3×3 square of 9 regions within the grid. The total number of regions accessed is at most $9 + 12 = 21$, namely the 3×3 square and 3 neighboring regions on each side (if diagonal edges are allowed then the total number of regions accessed is 25). The result of the second function is the shortest distance stored in $d(v)$ for all the vertices in T using paths and known distances within T. Hence these found shortest distance might not be final.

The function $r(v)$ return the region which contains v.

The pseudo code:

```
$Square.Calc\_sssp(G, s)$

Comment: $G$ is the grid graph.

Comment: $s$ is the source vertex.

 A1 Let $main\_pq$ be the main priority queue.
```

```
    Comment: $main\_pq$ will accept region id's such as $r(v)$ with some
    key value $d(v)$. Where $r(v)$ is the region of vertex $v$. If there are
    two entries in $main\_pq$ with the same $r(v)$ value then the one
    with the lowest key value is stored and the other is removed (or
    not added).
A2  Let $sub\_pq(R)$ be the priority queue for region $R$.
    Comment: The sub priority queue, $sub\_pq(R)$, will only contain
    vertices from the boundary of $R$ and possible the source.
A3  Let $pq$ be a priority queue which contains vertices and there key
    values.
A4  For all $v \in G$ let $d(v) := \infty$
A5  Let $d(s) := 0$
A6  Push $r(s)$ with key $d(s)$ into $main\_pq$
A7  Push $s$ with key $d(s)$ into $sub\_pq(r(s))$
A8  While $main\_pq$ is not empty do:
    A9  $R := pq.extract\_minKey()$
    A10 Let $T_{0..8}$ be $R$ and its 8 surrounding regions.
    A11 For $i := 0..8$ do:
        A12 While $sub\_pq(T_i)$ is not empty do:
            A14 Push $v$ with key $d(v)$ into $pq$
    A15 $Square.Process(T, pq, main\_pq, sub\_pq)$
```

$Square.Process(T, pq, main\_pq, sub\_pq)$

Comment: T are the 9 regions which are being processed.

Comment: pq is the priority queue which is used.

Comment: $main\_pq$ is the priority queue to maintain the minimum among the regions.

Comment: sub_pq are the priority queues which contains the minimum for the vertices within the boundary of a region.

```
P1  While $pq$ is not empty do:
    P2  $v := pq.extract\_minKey()$
    P3  For each outgoing edge $(v, w)$ of $v$ do:
        P4  If $d(v) + length(v, w) < d(w)$ then
            P5  $d(w) := d(v) + length(v, w)$
            P6  If $r(w) \in T$ then
                P7  Push $w$ with key $d(w)$ into $pq$
            P8  Else
                P9  Push $w$ with key $d(w)$ into $sub\_pq(r(w))$
                P10 Push $r(w)$ with key $d(w)$ into $main\_pq(r(w))$
```

3.5.2 Running time (CPU)

The CPU bound for this algorithm is $\Theta(n \cdot \sqrt{R} \cdot \log R + n \cdot \log \frac{n}{R})$. The proof for this bound will be similar to the I/O bound proven in section 3.4.3. First we will prove the upper bound of $O(n \cdot \sqrt{R} \cdot \log R)$ and then we will give an example which encounters the $\Omega(n \cdot \sqrt{R} \cdot \log R)$. $\Theta(n \cdot \log \frac{n}{R})$ is the cost to extract and push regions into the main queue. If R is small or $R = 1$ then the CPU bound in essence becomes $\Theta(n \cdot \log n)$, which is the CPU bound of Dijkstra's algorithm

Theorem (upper bound) Let R be the size of a region then the CPU upper bound for the Square algorithm is

$$O(\sqrt{R} \cdot \log R \cdot n)$$

Proof (upper bound) Given some R and some grid we know that a region is calculated at most $O(\sqrt{R})$ times. The reason for this is that a region only has $O(\sqrt{R})$ boundary vertices and each boundary vertex can only be selected from the main queue once. Since once it has been selected from the main queue it will have the minimal possible value, this is similar to the invariant which Dijkstra's algorithm maintains. The fact that this algorithm processes a 3×3 block in each sub step does not affect this. Hence each region is processed at most $O(\sqrt{R})$ times.

Each time when a region is selected from the main queue at most $O(R \cdot \log R)$ operations are performed. Remember here that the sub queue has size at most $O(R)$ and each operation on it takes $O(\log R)$ time if it is full.

Now adding the previous two parts together and remembering that there are $\frac{n}{R}$ regions we can calculate an upper bound of:

$$O(\sqrt{R}) \cdot O(R \cdot \log R) \cdot O(\frac{n}{R})$$
$$=$$
$$O(\sqrt{R} \cdot R \cdot \log R \cdot \frac{n}{R})$$
$$=$$
$$O(\sqrt{R} \cdot \log R \cdot n)$$

This is exactly what we were looking for.

■

Theorem (lower bound) Let R be the size of a region then the CPU upper bound for the Square algorithm is

$$\Omega(\sqrt{R} \cdot \log R \cdot n)$$

Proof (lower bound) The second part consists of giving an example which uses at least $\Omega(\sqrt{R} \cdot \log R \cdot n)$ CPU operations. We construct the following input grid:

- For all i and j let $w(v_{i,j}, v_{i+1,j}) = 0$.
- For $i = 0$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = 0$
- For $i = 0$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = n$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 0$ let $w(v_{i,j}, v_{i,j+1}) = 0$
- For $i = \sqrt{n} - 1$ and $j \mod 2 = 1$ let $w(v_{i,j}, v_{i,j+1}) = n$
- Let the source vertex be $v_{0,0}$

If we assume here that $4 \cdot n > R$, then whenever a region is selected in the main priority queue at the most $2 \cdot \sqrt{R}$ vertices get their final value. This means that each region has to selected at least $\Omega(\frac{R}{\sqrt{R}}) = \Omega(\sqrt{R})$ times. Using this we can calculate the minimal required number of CPU operations:

$$\Omega(2 \cdot \sqrt{R} \cdot R \cdot \log R \cdot \frac{n}{R})$$

$$=$$

$$\Omega(\sqrt{R} \cdot \log R \cdot n)$$

■

Proof (tight bound) Combining the upper and lower bound we get a tight bound of

$$\Theta(n \cdot \sqrt{R} \cdot \log R)$$

To get the final bound we need to add the cost of using the main priority queue, which is the main cost for small R. The CPU bound is:

$$\Theta(n \cdot \sqrt{R} \cdot \log R + n \cdot \log \frac{n}{R})$$

■

3.5.3 I/O-efficiency in the worst case

The worst case I/O bound is also $\Theta(\frac{n}{\sqrt{R}})$. The reasoning behind this is the same as that for the Quasi-linear algorithm and the fact that this algorithm processes a 3×3 square instead of just 1 region in every sub step does not change this. See section 3.4.3 for the details on this.

3.5.4 I/O-efficiency in grids with constant highway dimension

If the input grid has a constant highway dimension then the I/O bound for the Square algorithm will still be $\Theta(\frac{n}{\sqrt{B}})$. We refer to section 3.4.4 for an example which yields the $\Omega(\frac{n}{\sqrt{B}})$. Note here that this example in section 3.4.4 also works for the Square algorithm.

3.5.5 I/O-efficiency in grids with constant augmented highway dimension

Using the concept of augmented highway dimension we can proof an improved I/O bound for the Square algorithm.

Let $r = \frac{\sqrt{R}}{2}$ and choose for each Q a set S_Q such that:

$$\forall v, w \in B_{u,4\cdot r} : (|D(v,w)| > \frac{\sqrt{R}}{2} \wedge P(v,w) \subseteq B_{u,4\cdot\frac{\sqrt{R}}{2}}) \Rightarrow (P(v,w) \cap S_Q \neq \emptyset)$$

Where $|S_Q| \leq h$

Now let us define T_{inner} to be the set of vertices of all the boundary vertices of Q and let T_{outer} be the set of vertices of all the boundary vertices of the nine region which make up the 3×3 square region where region Q is the center region.

By the above chosen values for the augmented highway dimension we know that any shortest path from $u \in T_{inner}$ to $v \in T_{outer}$ and vice versa, where $P(u,v) \subseteq B_{u,4\cdot\frac{\sqrt{R}}{2}}$ has to contain a vertex from $S_{T_{inner}}$.

We assume that the source vertex, s, is always in S_Q as long as $s \in B_{u,4\cdot r}$. This assumption will not change the fact that $|S_Q|$ is constant and lower or equal to $h + 1$. Moreover it will ensure that we will not need to consider the special case where the source is nearby.

Now whenever a region, I_0, is selected from the main priority queue then some vertex, v_0, on the boundary of I_0 has the lowest value. Let $d(v)$ be the current value of vertex v. In order for I_0 with value $d(v_0)$ to be added to the main priority queue all vertices on a path from s to v_0 through $v_s \in S_{I_0}$ thus have its final value too. The reason why we can say this is because $r = \frac{\sqrt{R}}{2} < \sqrt{R}$. Thus v_s is also set to its final value.

When processing region I_0 and its adjacent regions each shortest path from s to any vertex $v_i \in T_{inner}$ containing v_s will also be relaxed to its final value as long as $P(v_i, v_s) \subseteq B_{v_i,4\cdot\frac{\sqrt{R}}{2}}$ holds.

Due to the above we know that each boundary vertex of Q is only reachable by going through a vertex in S_Q, which implies that each region is accessed at most $\Theta(h)$ times instead of the standard $\Theta(\sqrt{R})$ times. This means that if $h < \sqrt{B}$ then a new I/O bound can be deduced namely:

$$\Theta(\frac{n}{B} \cdot h)$$

3.6 TerraCost

The TerraCost algorithm by Hazel et al, with the separators and optimized topology tree. The definition and analysis can be found in Hazel et al [8]. We do however explain in short how the algorithm works and for the special case of grid graphs improve the I/O bound to $O(\frac{n}{B})$ and the CPU bound of Step 1 (Intra-Tile Dijkstra) to $O(n \cdot \log R)$.

3.6.1 Description

The terra-cost algorithm uses the well known concept of separators to limit the number of required I/Os. The algorithm goes through four steps to solve the single source shortest path problem and the algorithm is given in figure 14. Step 2 (Sorting) is not applied, since with a regular grid it is easy to store all the edges in Step 1 (Intra-Tile Dijkstra) in a sorted manure (meaning that Step 2 (Sorting) should take no time).

We propose and have implemented two modifications. The first modification improves on the CPU bound in Step 1 (Intra-Tile Dijkstra) . The second modification improves on the I/O bound of Step 3 (Inter-Tile Dijkstra) .

The modification in Step 1 (Intra-Tile Dijkstra) only replaces the way in which the shortest distances from and to each boundary vertex within a region are calculated. Instead of running Dijkstra's algorithm $O(\sqrt{R})$ times we implemented a multiple-source shortest path algorithm, which is explained in Klein [10]. The used topology tree is explained in depth in Sleator et al [16] and Sleator et al [15]. This modification changes the CPU bound of Step 1 (Intra-Tile Dijkstra) from $O(n \cdot \sqrt{R} \cdot \log R)$ to $O(\frac{n}{R} \cdot R \cdot \log R) = O(n \cdot \log R)$.

The modification in Step 3 (Inter-Tile Dijkstra) is related to the priority queue. Instead of using one large priority queue of size $O(\frac{n}{R} \cdot \sqrt{R}) = O(\frac{n}{\sqrt{R}})$ we propose to use one main queue of size $O(\frac{n}{R})$ and a set of sub queue. Where each sub queue is related to its own region and has size at most $O(\sqrt{R})$. The reasoning behind this is that if R is large enough then $O(\frac{n}{R})$ will fit in memory and thus no I/Os would be wasted on this. This is much the same as the reasoning presented in 3.4.3.

The new pseudo code for modified Step 3 (Inter-Tile Dijkstra) is given below:

```
$Terra.Step3(G, s, main\_pq, sub\_pq)$

Comment: $s$ is the source vertex and $G$ is the modified graph created
in Step 1 (Intra-Tile Dijkstra) and sorted in step 2.

Comment: $main\_pq$ is the priority queue to maintain the minimum
among the regions.
```

Comment: *sub_pq* are the priority queues which contains the minimum for the vertices within the boundary of a region.

```
S3.1 For all $v \in G$ let $d(v) := \infty$
S3.2 Push $r(s)$ with key 0 into $main\_pq$
S3.3 Push $s$ with key 0 into $sub\_pq(r(s))$
S3.4 While $main\_pq$ is not empty do:
    S3.5 $t := main\_pq.extract\_minKey()$
    S3.6 $v := sub\_pq(t).extract\_minKey()$
    S3.7 For each outgoing edge $(v, w)$ of $v$ do:
        S3.8 If $d(v) + length(v, w) < d(w)$ then
            S3.9 $d(w) := d(v) + length(v, w)$
            S3.10 Push $r(w)$ with key $d(w)$ into $main\_pq$
            S3.11 Push $w$ with key $d(w)$ into $sub\_pq(r(w))$
            S3.12 If $sub\_pq(t)$ is not empty then
                S3.13 v := $sub\_pq(t).extract\_minKey()$
                S3.14 Push $r(v)$ with key $d(v)$ into $main\_pq$
                S3.15 Push $v$ with key $d(v)$ into $sub\_pq(r(v))$
```

The result of this step is still that for all $v \in G$ the shortest distance is stored in $d(v)$.

3.6.2 Running time (CPU)

The CPU bound for Step 1 (Intra-Tile Dijkstra) using the optimized topology tree is $O(n \cdot \log R)$. The prove for this new bound is given in Klein [10].

The total CPU bound now becomes $O(n \cdot \log R) + O(n \cdot \log \frac{n}{R}) = O(n \cdot \log R + n \cdot \log \frac{n}{R}) = O(n \cdot \log n)$. $O(n \cdot \log \frac{n}{R})$ is derived from the cost of using the main priority queue defined used in Step 3 (Inter-Tile Dijkstra) .

3.6.3 I/O-efficiency in the worst case

The bottleneck for the I/O bound is found in Step 3 (Inter-Tile Dijkstra) of the algorithm. Since Step 1 (Intra-Tile Dijkstra) and Step 4 (Final Dijkstra) are already optimal with $\Theta(\frac{n}{B})$ I/Os.

The I/O cost for Step 3 (Inter-Tile Dijkstra) given in Hazel et al [8] is $O(sort(n))$, which comes from the use of an I/O efficient priority queue. We propose to remove this I/O efficient priority queue and to replace the queue with one main queue and a set of sub queues. The main queue would only hold one key value for each region, which means that it would require $O(\frac{n}{R})$ space. The sub queues, one for each region, would hold only keys for vertices within that region, since only boundary vertices of regions exist in this step we know that each sub queue has size at most $O(\sqrt{R})$.

The different steps of the TerraCost algorithm as given in Hazel et al [8]:

- ***Step 1 (Intra-Tile Dijkstra)*** *First we partition the grid into tiles of size R and compute (an edge-list representation of) the substitute graph S. If there are any sources in a tile, we construct one additional vertex s in that tile (as in the single-source version); this vertex, however, now represents all sources inside the tile. We then run Dijkstra's algorithm from each of the sources, and for each boundary vertex v of the tile we construct exactly one edge (v, s) that corresponds to the least-cost path of that vertex to any one of the sources and is weighted with the cost of this path. This means that we always output at most $4 \cdot \sqrt{R}$ source-to-boundary edges when processing a single tile, irrespective of how many sources are in that tile. We also run Dijkstra's starting from each boundary vertex and reaching out to all other boundary vertices. Each least-cost path $\delta_S(u, v)$ computed in this step corresponds to an edge $(u, v, \delta_S(u, v))$ in the substitute graph. All edges are written to one of two streams, one for the source-to-boundary edges, the other one for boundary-to-boundary edges.*

- ***Step 2 (Sorting)*** *We sort the boundary-to-boundary stream created in Step 1 (Intra-Tile Dijkstra) such that all edges originating from the same vertex will be contiguous. This allows Step 3 (Inter-Tile Dijkstra) to efficiently index into this stream and to load the $O(\sqrt{R})$ neighbors on any vertex using $O(\frac{\sqrt{R}}{B})$ I/Os. We separate this step out because the substitute graph is large (has $O(n)$ edges), resides on disk, and sorting it takes a significant amount of time; also this step lends itself to future improvements.*

- ***Step 3 (Inter-Tile Dijkstra)*** *We compute the least-cost paths to all the boundary vertices using the substitute graph S. We run Dijkstra's using an I/O-efficient priority queue that is initialized with all the least-cost paths from sources to the boundary computed in Step 1 (Intra-Tile Dijkstra) . As vertices are settled, we load the edges adjacent to the current vertex by indexing into the edge-list representation of S (sorted boundary-to-boundary stream)*

- ***Step 4 (Final Dijkstra)*** *For each tile, we compute the least-cost paths to all internal points by running Dijkstra's starting at the boundary points along with any internal source points.*

Figure 14: The different steps in the TerraCost algorithm copied from **Terracost: A Versatile and scalable approach to Computing Least-Cost-Path Surfaces for Massive Grid-Based Terrains**. For the original paper we refer to Hazel et al [8].

Remember that each vertex in Step 3 (Inter-Tile Dijkstra) has $O(\sqrt{R})$ outgoing edges. Hence whenever a vertex in Step 3 (Inter-Tile Dijkstra) is chosen to be relaxed we know that this requires $O(1 + \frac{\sqrt{R}}{B})$ I/Os, which means that accessing the sub queue only increases the number of I/Os with a constant factor.

As long as $c \cdot M > \frac{n}{R}$ holds for some $c > 1$ we know that the main queue fits in memory and thus no I/Os are needed for using the main priority queue.

Theorem If $c \cdot M > \frac{n}{R}$ holds for some $c > 1$ and sub and main priority queues are used then the I/O bound for grid graphs will be $\Theta(\frac{n}{B})$

Proof There are $O(\frac{n \cdot \sqrt{R}}{R})$ vertices in Step 3 (Inter-Tile Dijkstra) . Thus using this and using that each vertex has at the most $O(\sqrt{R})$ outgoing edges needing $O(1 + \frac{\sqrt{R}}{B})$ I/Os whenever a vertex is selected to be relaxed we get:

$$O(\frac{n \cdot \sqrt{R}}{R}) \cdot O(1 + \frac{\sqrt{R}}{B})$$

$$=$$

$$O(\frac{n \cdot \sqrt{R}}{R} + \frac{n \cdot \sqrt{R}}{R} \cdot \frac{\sqrt{R}}{B})$$

$$=$$

$$O(\frac{n \cdot \sqrt{R}}{R} + \frac{n}{\sqrt{R}} \cdot \frac{\sqrt{R}}{B})$$

$$=$$

$$O(\frac{n}{\sqrt{R}} + \frac{n}{B})$$

Which means if $R = B^2$ then we get an optimal I/O bound of:

$$\Theta(\frac{n}{\sqrt{B^2}} + \frac{n}{B}) = \Theta(\frac{n}{B})$$

$$\blacksquare$$

Remarks Its interesting to note here that no assumptions on the data storage are made, hence it not only works for Z-order, but also for row and column order.

4 Experimental set-up

In this section we will explain the different grid graphs which will be generated and we will point out the relevant implementation details, which might effect the running time of our algorithms.

4.1 Implementation details

In order to verify and properly understand some of the numbers which are used in the calculations in this paper we need to say a bit more about the actual implementation. Firstly we will explain the sizes of different types of variables, then we will explain how the grid is implemented and what details should be known about the priority queue.

The main different variable types with their sizes are listed below:

- An integer is 32 bits.
- A pointer to another variable or array is 64 bits.
- Floating point numbers are 32 bits when they are stored in data structures which are swapped in out of memory. Otherwise they are 32 or 64 bits depending on the task which is performed.

Its important to note that in some cases its better to use 64 bits for floating point numbers, especially if precision is important.

4.1.1 Grid

As explained in section 2.2 there are several orderings in which the vertices of the grid can be stored, but this doesn't explain how a vertex is stored exactly. The structure which is used to store one vertex, v, is given below:

- w_{left}, a floating point number which defines the weight from v to the vertex left of it within the grid.
- w_{right}, a floating point number which defines the weight from v to the vertex to the right of it within the grid.
- w_{above}, a floating point number which defines the weight from v to the vertex above it within the grid.
- w_{below}, a floating point number which defines the weight from v to the vertex below it within the grid.
- $d_{min}(v)$, a floating point number which stores the minimal shortest distance from s to v.

If for instance v has no left neighbor within the grid, then $w_{left} = \infty$. The same holds for w_{right}, w_{above} and w_{below}. If v would be unreachable then $d_{min}(v) = \infty$.

The total size of this structure is $4 \cdot 5 = 20$ bytes.

4.1.2 Priority queue

Elements which are added to the priority queue have the following form:

- v, an integer which identifies the vertex within the grid related to $d(v)$.
- R_v, an integer which identifies in which region vertex v resides. This field is omitted if it has no meaning.
- $d(v)$, the current shortest known distance from s to v. $d(v)$ is also used as the key where we minimize the priority queue on.

This means that each element in the priority queue is $3 \cdot 4 = 12$ bytes general and $2 \cdot 4 = 8$ bytes in other cases such as if there is no region identifier or if the entire queue is associated with one particular region.

4.1.3 Dijkstra's Algorithm

Dijkstra's algorithm was implemented in accordance with the pseudo code given in section 3.2.1. For the experiments which run fully in memory we used our own standard implement priority queue as explained in section 4.1.2. In the I/O case we choose to use the I/O efficient priority queue from the Standard Template Library for Extra Large Data Sets (STXXL). This way we can ensure that if Dijksra's algorithm fails to run properly when running large data sets in an I/O setting we know that it is not the priority queue which is failing. This is important since the priority queue given a proper data set has size $\Theta(n)$.

Size of the Priority queue In section 3.2.3 we state that the theoretical I/O bound of Dijkstra's algorithm is $\Theta(\frac{n}{\sqrt{B}})$ if the number of vertices in the priority queue is at most $O(\sqrt{n})$, Z-order storage is used and $M = B^2 > c \cdot \sqrt{n} \cdot B$ holds for a $c > 1$.

Figures 15 and 16 show the number of vertices in the priority queue for an experiment of the worst case data set with 10% distortion and the fully random data set. It is visible that it is never greater then $c \cdot \sqrt{n} = 3 \cdot \sqrt{n}$. Tables 15 and 16 also show that the difference between the number of vertices in the priority queue and the number of blocks in which these vertices reside is approximately $O(\sqrt{B})$ (for example $1129 \cdot \sqrt{B} = 1129 \cdot \sqrt{256} = 1129 \cdot 16 = 18064 \approx 17241$).

In figure 17 we show for several different B the amount of memory which would be needed to achieve the I/O bound of $O(\frac{n}{\sqrt{B}})$ for Dijkstra's algorithm. If we would take $B = 262144$ bytes then we would need 195.82 MB to get the I/O bound of $O(\frac{n}{\sqrt{B}})$. As can be seen in figures 15 and 16 it is expected that a grid which is four times as big only needs two times as much memory to get the I/O bound of $c \cdot \frac{n}{\sqrt{B}} = c \cdot \frac{n}{\sqrt{262144}} = c \cdot \frac{n}{512}$, where $c > 1$ is some reasonable constant. This would mean that a grid of 32768 × 32768 only needs $4 \cdot 195.82$ MB = 783.28 MB of memory, which is a lot less then the $32768^2 \cdot 20 = 20$ Gigabytes needed to store the grid itself. Also the constant factor of nine in table 17 might be to pessimistic in practice and it might be closer to three or four.

Block size (B) in vertices	Number of blocks which have a vertex in the priority queue for: Worst case with 10% distortion and grid size: 1024 × 1024	2048 × 2048	4096 × 4096	8192 × 8192
Dijkstra[Z-order,$B = 1$]	5415	7106	12301	22410
Dijkstra[Z-order,$B = 256$]	173	384	505	892
Dijkstra[Z-order,$B = 1024$]	67	149	211	383
Dijkstra[Z-order,$B = 4096$]	32	65	96	174
Dijkstra[Z-order,$B = 16384$]	16	25	46	83
Dijkstra[Z-order,$B = 65536$]	8	11	23	41

Figure 15: The maximum number of different blocks which represent one or more vertices in the priority queue during the execution of Dijkstra's algorithm on the worst case with 10% distortion data set defined in section 4.2.2. The source vertex is chosen to be in the top left corner. The first line with $B = 1$ represents the number of vertices in the priority queue.

Block size (B) in vertices	Number of blocks which have a vertex in the priority queue for: Fully random and grid size: 1024 × 1024	2048 × 2048	4096 × 4096	8192 × 8192
Dijkstra[Z-order,$B = 1$]	2053	3725	7448	17241
Dijkstra[Z-order,$B = 256$]	147	281	538	1129
Dijkstra[Z-order,$B = 1024$]	71	137	262	541
Dijkstra[Z-order,$B = 4096$]	34	66	130	264
Dijkstra[Z-order,$B = 16384$]	16	34	65	131
Dijkstra[Z-order,$B = 65536$]	8	17	33	66

Figure 16: The maximum number of different blocks which represent one or more vertices in the priority queue during the execution of Dijkstra's algorithm on the fully random data set defined in section 4.2.3. The source vertex is chosen to be in the top left corner. The first line with $B = 1$ represents the number of vertices in the priority queue.

	Maximum total required memory size:	
Block size (B) in bytes	Worst case with 10% distortion	Fully random
Dijkstra[Z-order,$B = 256 \cdot 16 = 4096$]	$892 \cdot B \cdot 9 \leq 32.9$ MB	$1129 \cdot B \cdot 9 \leq 41.7$ MB
Dijkstra[Z-order,$B = 1024 \cdot 16 = 16384$]	$383 \cdot B \cdot 9 \leq 56.5$ MB	$541 \cdot B \cdot 9 \leq 79.8$ MB
Dijkstra[Z-order,$B = 4096 \cdot 16 = 65536$]	$174 \cdot B \cdot 9 \leq 102.7$ MB	$264 \cdot B \cdot 9 \leq 155.8$ MB
Dijkstra[Z-order,$B = 16384 \cdot 16 = 262144$]	$83 \cdot B \cdot 9 \leq 195.9$ MB	$131 \cdot B \cdot 9 \leq 309.1$ MB
Dijkstra[Z-order,$B = 65536 \cdot 16 = 1048576$]	$41 \cdot B \cdot 9 \leq 387.0$ MB	$66 \cdot B \cdot 9 \leq 622.9$ MB

Figure 17: The required memory size to achieve the $O(\frac{n}{\sqrt{B}})$ I/O bound for Dijkstra's algorithm on 10% distortion 4.2.2 and the fully random 4.2.3 data set. The size of the grid was 8192 $\times$ 8192. The source vertex chosen to be in the top left corner.

4.1.4 Quasi-linear

The Quasi-linear algorithm is implemented as it is explained in section 3.4.1. The priority queues which are used are not I/O efficient, but the main queue will always fit in memory and if the sub queues are needed then an entire region is also loaded in from disk. Thus this does not asymptotically add any I/Os.

4.1.5 Square

The Square algorithm is implemented as it is explained in section 3.5.1. The priority queues which are used are not I/O efficient, but the main queue will always fit in memory and if the sub queues are needed then an entire region is also loaded in from disk. Thus this does not asymptotically add any I/Os.

4.1.6 Terra-simple

The Terra-simple algorithm is implemented in accordance with the Terracost algorithm as explained Hazel et al [8]. This algorithm is not used in the I/O case, hence we choose not to use an I/O efficient priority queue, but just the normal queue which is explained in section 3.2.1.

4.1.7 Terra-efficient

The Terra-efficient algorithm is implemented in accordance with the modifications to the Terracost algorithm as defined in section 3.6.1. The topology tree is implemented as defined in Sleator et al [16] and Sleator et al [15]. It is important to note here that the topology tree requires a lot of memory in order to be used. For example if $R = 65536$ then the topology tree uses 80 Megabytes of memory. This relation is linear hence double R then the amount of memory needed is also doubled. This also means that each vertex in R needs approximately 1280 bytes to information about it, which seems to be a very large number if you consider what is actually stored. This is also a place where further improvements should be possible.

Topology tree In order to run the multiple-source shortest path algorithm, which is explained in Klein [10] we also had to transform all the weights from floating point numbers to integers. The reason for this is that floating point numbers are imprecise and might give a different result to an equation if numbers are added together in a different order. This fact might seem trivial, but when working with a complex concept of topology trees in the area of multiple-source shortest path algorithms as explained in Klein [10] it can give rise to problems. Such as wrongly adding edges to the topology tree and since the topology tree is rather complex by itself it is not easy to generally fix this. Hence we choose to do it the easy way and just drop the concept of floats while inside the multiple-source shortest path algorithm.

Since there is always the question about loss of precision when transforming a value from a floating point number to an integer we experimented a bit and came to the conclusion that 64 bit integers would be good enough for the grid graphs on which we where running experiments. In order to convince ourselves that this precision will not give a problem at some point we also tried to use larger integers, which yielded that larger integers gave better and better precision. Hence in essence using integers is not a problem even though it might require slightly more memory, but since floating point numbers are 32 bits then an increase to 64 bit integers is hardly the problem. Besides this these 64 bits where only used in Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm, which runs completely in memory.

The S^* graph created in Step 1 (Intra-Tile Dijkstra) The Terracost algorithm creates a new graph, S^* from the input graph and uses it to get a nice I/O bound in Step 3 (Inter-Tile Dijkstra) of the algorithm. This S^* graph is stored in a way which is as compact as possible, namely each internal vertex, v, in S^* has approximately $4 \cdot \sqrt{R}$ outgoing edges. These edges are stored in a list where each weight has size of the floating point number. Thus this means that each vertex in S^* will need $4 \cdot 4 \cdot \sqrt{R} = 16 \cdot \sqrt{R}$ bytes of memory. The source and destination vertices of these edges are retrieved using a simple table of size $O(\sqrt{R})$ and for all vertices the same table is used.

4.2 Data-sets

In the following subsections the data sets which were used to test the implemented algorithms are defined. The total size of a data set is always denoted in the number of vertices within the grid, n, and each grid always is a square where each row and column contain $\sqrt{n}$ vertices. To make make the implementation and storage of the grid easier we assume that $\sqrt{n}$ is always a power of 2. Besides this the grid graph is directed and each vertex has two, three or four ingoing and outgoing edges, namely one edge for each vertex above, below, right or left of it.

4.2.1 Worst case

The worst case data set is made in such a way such that all algorithms we test perform asymptotically worst on it. We define the data set as follows:

- For all i and $j \mod 3 \in \{0, 2\}$ let $w(v_{i,j}, v_{i+1,j}) = 0$ and $w(v_{i+1,j}, v_{i,j}) = 0$.
- For all i and $j \mod 3 = 1$ let $w(v_{i,j}, v_{i+1,j}) = x$ and $w(v_{i+1,j}, v_{i,j}) = y$ where x and y are two random values between 0 and 1
- For $i = 0$ and $j \mod 4 \in \{0, 1\}$ let $w(v_{i,j}, v_{i,j+1}) = x$ and $w(v_{i,j+1}, v_{i,j}) = y$ where x and y are two random values between 0 and 1
- For $i = 0$ and $j \mod 4 \in \{2, 3\}$ let $w(v_{i,j}, v_{i,j+1}) = 0$ and $w(v_{i,j+1}, v_{i,j}) = 0$
- For $i = \sqrt{n} - 1$ and $j \mod 4 \in \{2, 3\}$ let $w(v_{i,j}, v_{i,j+1}) = x$ and $w(v_{i,j+1}, v_{i,j}) = y$ where x and y are two random values between 0 and 1
- For $i = \sqrt{n} - 1$ and $j \mod 4 \in \{0, 1\}$ let $w(v_{i,j}, v_{i,j+1}) = 0$ and $w(v_{i,j+1}, v_{i,j}) = 0$

A visual representation of the above data set is given in figure 18.

4.2.2 Worst case with distortion rate $\varphi\%$

Because the worst case data set defined in 4.2.1 might not be very realistic we thought of taking a worst case data set and adding some amount of random distortion to it.

The way how this was implemented is that first a worst case grid graph is calculated after which $\varphi\%$ of the edges within the graph are chosen at random and their value is changed to a random value $x \in [0, 1]$ where x is independently and randomly chosen.

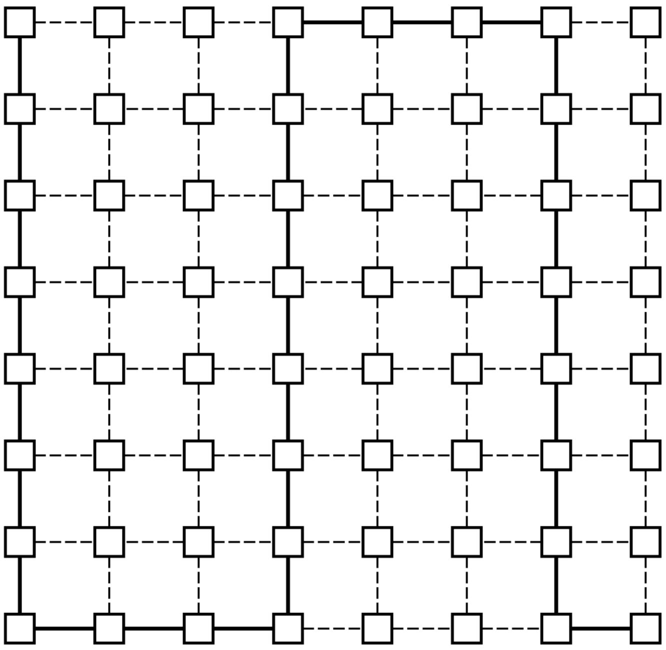


Figure 18: A 8×8 grid graph is shown here where the thick edges have weight 0 and the dotted edges have a random weight between 0 and 1.

4.2.3 Fully random data set

The fully random data set is defined as a graph where all edges have some value $x \in [0, 1]$ where x is independently and randomly chosen.

4.2.4 Highways and obstacles

The highways and obstacles data set is an other approach to try to get a random grid, which is expected to be closer to input data sets which are seen in practice. In this case we take a fully random grid graph as defined in section 4.2.3 and then modify it as follows:

- Generate $\delta \cdot \sqrt{n}$ random highways, where each highway is constructed by selecting two random vertices u and v and setting all outgoing weights on all vertices on a path from u to v with the minimal number of edges to a very low value γ.
- Generate $\delta \cdot \sqrt{n}$ random obstacles, where each obstacle is constructed by selecting two random vertices u and v and setting all outgoing weights on all vertices on a path from u to v with the minimal number of edges to a very high value β.

4.2.5 Real world data sets

The real data sets are derived from height maps, where a map is subdivided in equal-sized squares of some size and each square has a certain value (its height).

In order to create grid graphs from these height maps we use the following function:

$$length(v_{(i,j)}, v_{(k,l)}) = \frac{cost(v_{(i,j)}) + cost(v_{(k,l)})}{2}$$

Where $length(v_{(i,j)}, v_{(k,l)})$ is the weight of the edge between vertices $v_{(i,j)}$ and $v_{(k,l)}$. $cost(v_{(i,j)})$ is the height value at row i and column j within the height map.

Hence each square in the height map is transformed into a vertex and added to G and for each edge to a neighboring square in the height map a weight is calculated. Neighboring squares are either above, below, left or right of the current square (resulting in the fact that each vertex has at most four neighbors).

In order to finally use the resulting grid we clipped a part of the graph to make it a square, such that both sides have the same length, and to ensure that no height values which have an unknown value are represented in the final regular grid graph. This data set is stored as a directed grid even though $w((v,u)) = w((u,v))$ for all $u, v \in V$.

5 Results

In this section we will present the results for the different data sets as defined in section 4.2 for the case where the amount of memory is limited and when it is not limited.

5.1 Fully in memory

In the following sections we will go through the results for the different data sets as defined in section 4.2, with an unlimited amount of memory.

5.1.1 Machine

The hardware which was used to run the in memory experiments was:

- Intel i7-3770 CPU running at 3.40 GHz.
- 16 Gigabytes of internal memory.
- 64-bit operating system and executables.

It is made sure that in these experiments the hard drive is not used and that more then enough memory is available for the experiments, such that the hard drive is not needed.

5.1.2 Worst case

In this section we will present the results for the worst case data set as defined in section 4.2.1, where the entire grid and data structures fit in memory.

In figure 19 we see that Quasi-linear , Square and dijkstra's algorithm perform best and that Terra-simple and Terra-efficient perform worst in memory.

For algorithms Terra-simple and Terra-efficient we see that the minimal region size of $R = 4$ is chosen, which can be explained by looking at figures 22 and 23, since Step 1 (Intra-Tile Dijkstra) clearly requires the most execution time and the time taken for Step 1 (Intra-Tile Dijkstra) increases when R gets bigger.

For the Square algorithm we also see that the smallest R is chosen, which is supported by the CPU bound of the algorithm of $\Theta(n \cdot \sqrt{R} \cdot \log(R) + n \cdot log(\frac{n}{R}))$ and shown in figure 20.

The Quasi-linear algorithm performs best and unlike the other algorithms performs better with a somewhat larger region size, with a small α. This is to be expected when reading paper Henzinger et al [9]. Figure 21 shows the different running times for different R and α values.

Dijkstra's algorithm as seen in 19 doesn't perform badly, but one can see that Quasi-linear and Square are a bit better this is probably due to the fact that Dijkstra's algorithm does not have to have spacial or temporal locality when accessing different vertices in each Dijkstra step, while Square and Quasi-linear do a certain amount of work on vertices within the same region. Hence they do have a form of spacial and temporal locality. Besides this the priority queue of Dijkstra's algorithm has size $O(n)$, while for the other two algorithms this is at most $O(\frac{n}{R} + R)$.

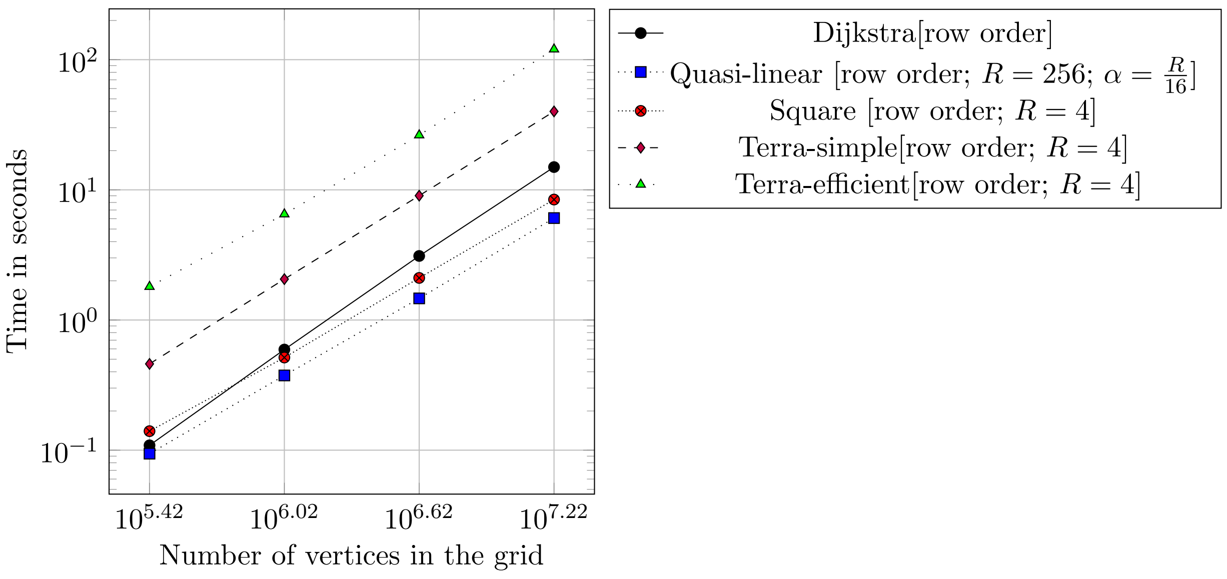


Figure 19: Best running time results for worst case data set in memory. For the experiments all the data structures and the input grid graph fit in memory.

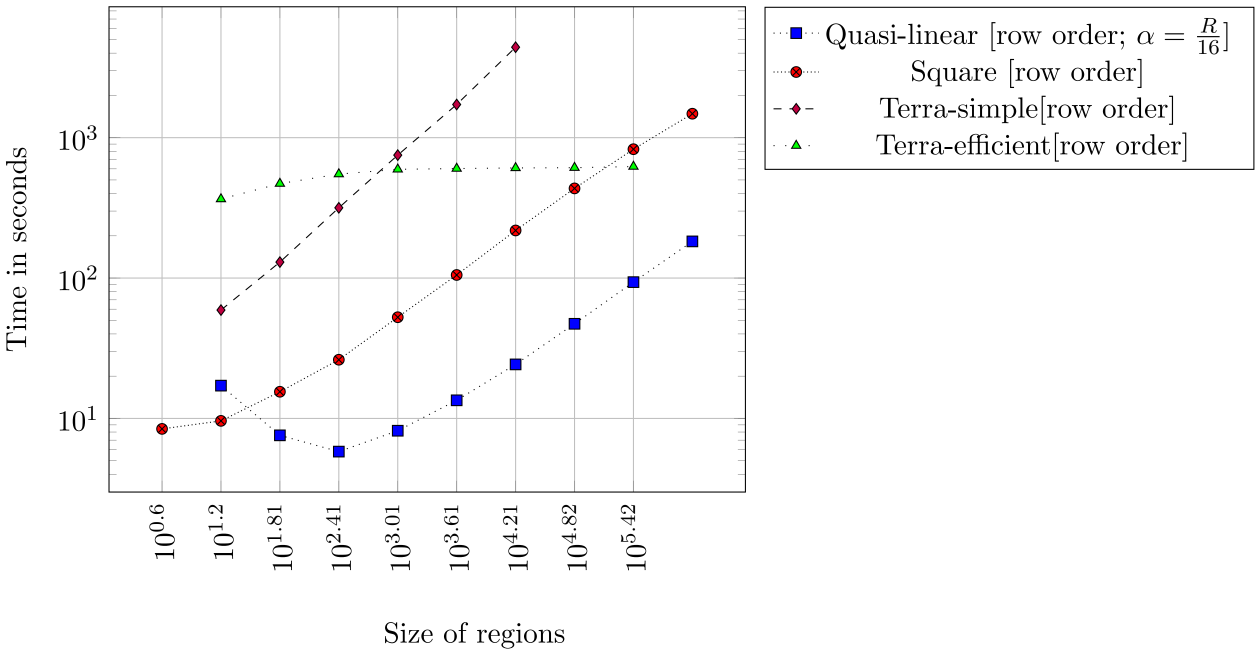


Figure 20: Running time results for the worst case data set in a 4096 $\times$ 4096 grid graph, where the region sizes are varied. For the experiments all the data structures and the input grid graph fit in memory.

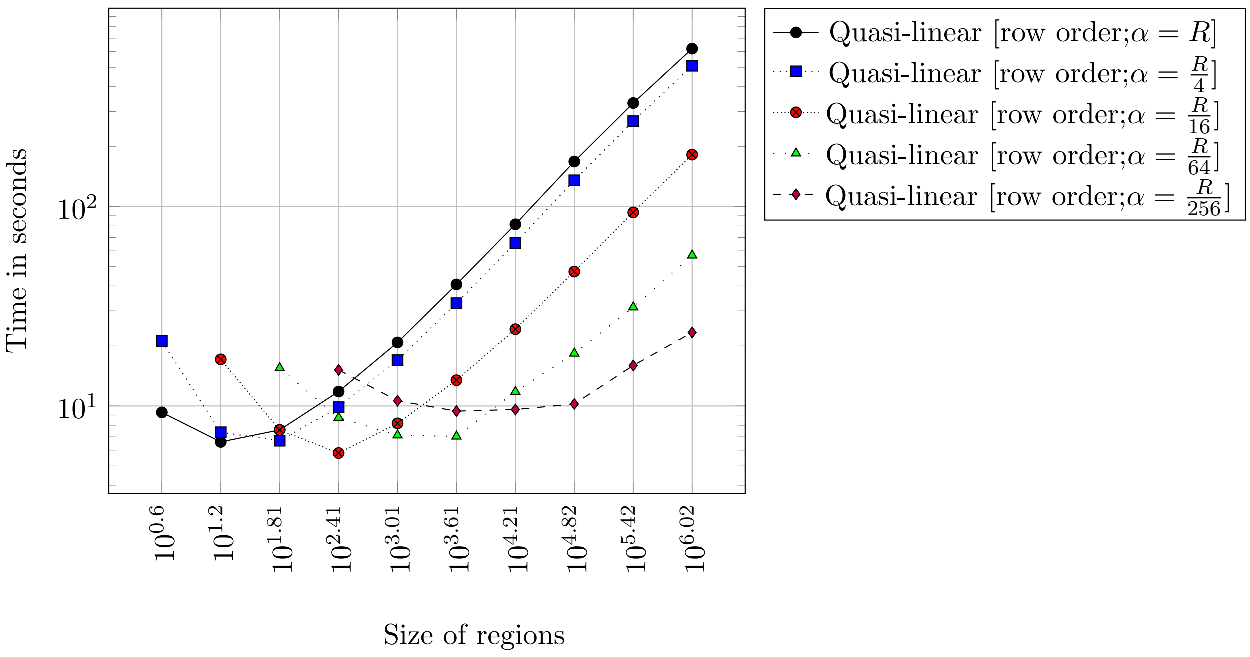


Figure 21: Running time results for the worst case data set in a 4096 × 4096 grid graph, where the region sizes and α values are varied for the Quasi-linear algorithm. For the experiments all the data structures and the input grid graph fit in memory.

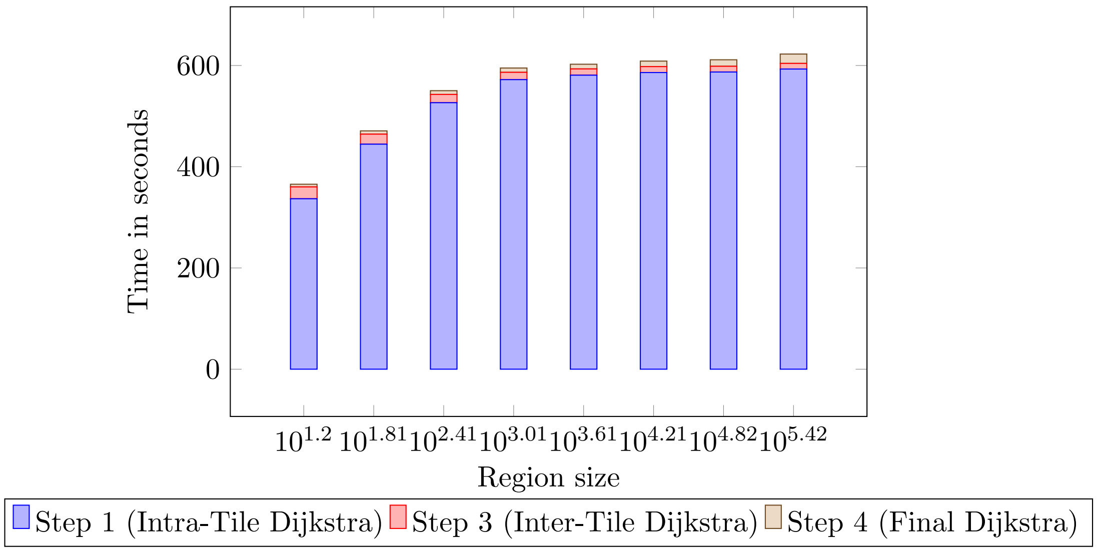


Figure 22: Running time results for the worst case data set in a 4096 × 4096 grid graph, where the region sizes are varied for the Terra-efficient algorithm. The running times for the different sub steps of the Terra-efficient algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

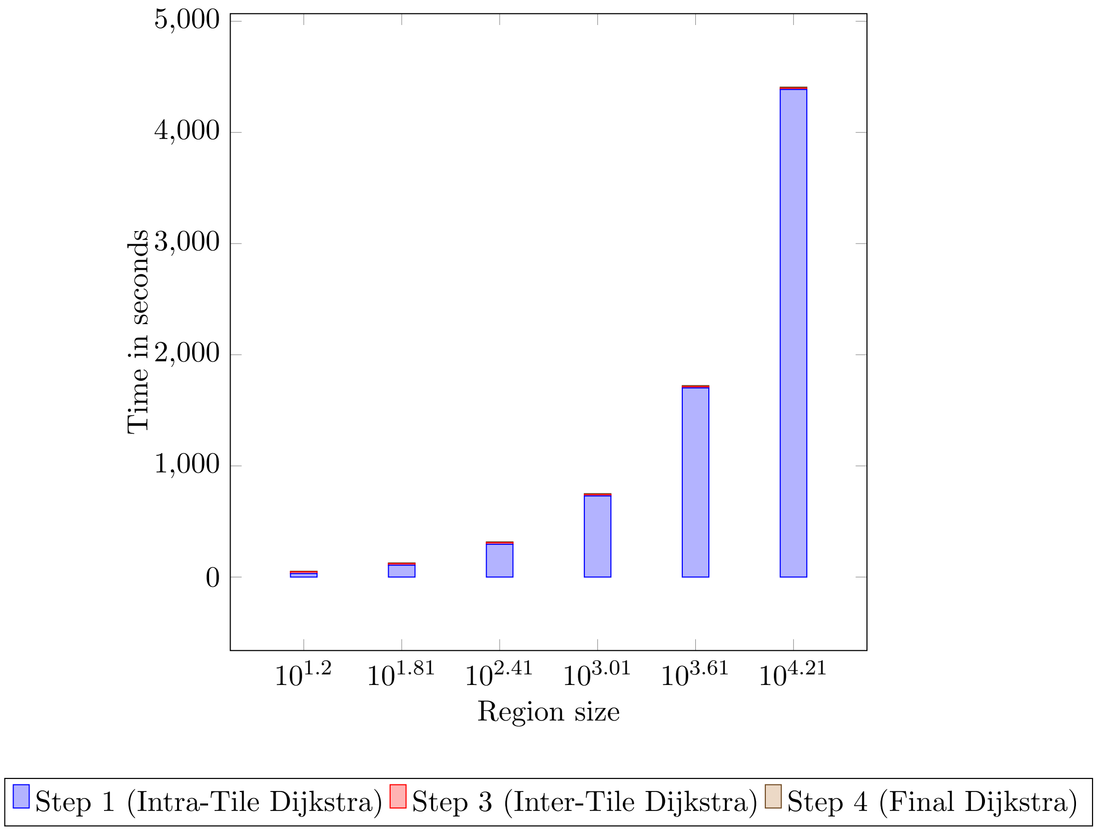


Figure 23: Running time results for the worst case data set in a 4096×4096 grid graph, where the region sizes are varied for the Terra-simple algorithm. The running times for the different sub steps of the Terra-simple algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

5.1.3 Worst case data set with 10% distortion

In this section we will present the results for the worst case data set with 10% distortion as defined in section 4.2.2, where the entire grid and data structures fit in memory.

In figure 24 we see that Quasi-linear , Square and Dijkstra's algorithm perform much better then Terra-simple and Terra-efficient . This is simply due to the fact that Step 1 (Intra-Tile Dijkstra) of Terra-simple and Terra-efficient is costly for larger R as seen in figures 27 and 28.

Figure 25 shows that the Square algorithm always performs about the same for any R, but for Quasi-linear there does seem to be a difference. The difference for Quasi-linear for different values of R and α is also clearly perceived in figure 26, where we see that $R = 10^{3.01} \approx 1024$ and $\alpha = \frac{R}{16} = \frac{1024}{16} = 64$ gives the best performance.

Comparison with the worst case data set If we compare this data set with the worst case data set, then we notice from figures 19 and 24 that Dijkstra's, Quasi-linear and Square perform slightly better for a data set with some distortion added, while for Terra-efficient and Terra-simple there is little to no difference noticeable. Besides this it is interesting to note that for Quasi-linear and Square larger value for R also perform well and for Quasi-linear we notice that a smaller α is still preferred. For Terra-efficient we notice in figures 27 and 22 that Step 1 (Intra-Tile Dijkstra) requires more time in the worst case with distortion then without. This difference is probably due to the fact that more mutations on the topology tree are required.

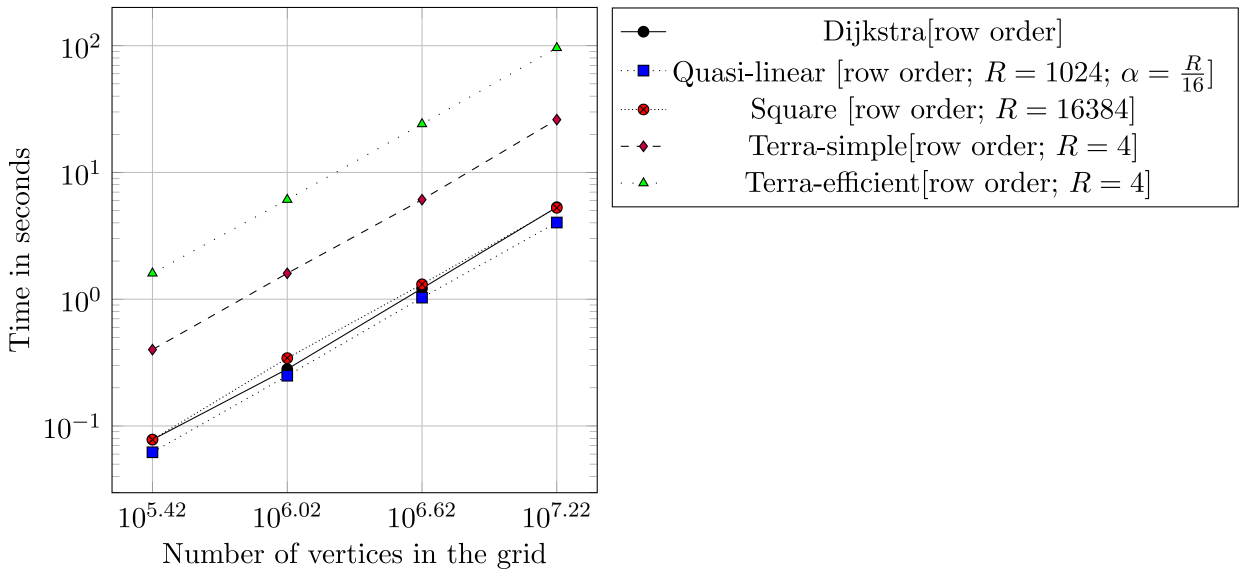


Figure 24: Best running time results for the worst case data with set 10% random distortion. For the experiments all the data structures and the input grid graph fit in memory.

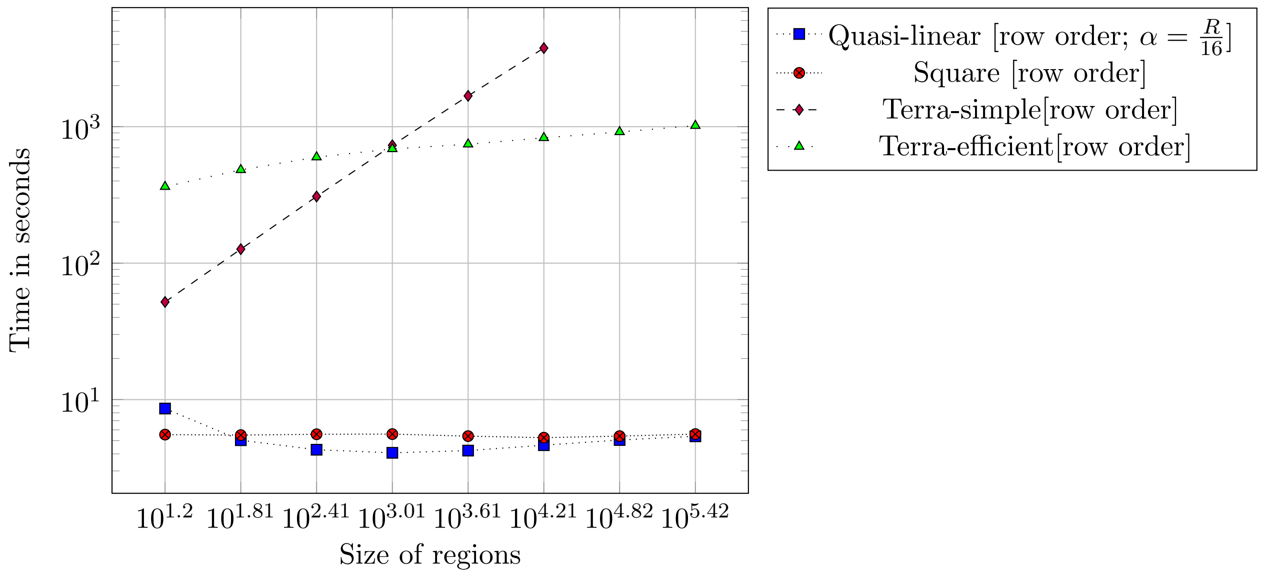


Figure 25: Running time results for the worst case data set with 10% distortion in a 4096 $\times$ 4096 grid graph, where region size are varied. For the experiments all the data structures and the input grid graph fit in memory.

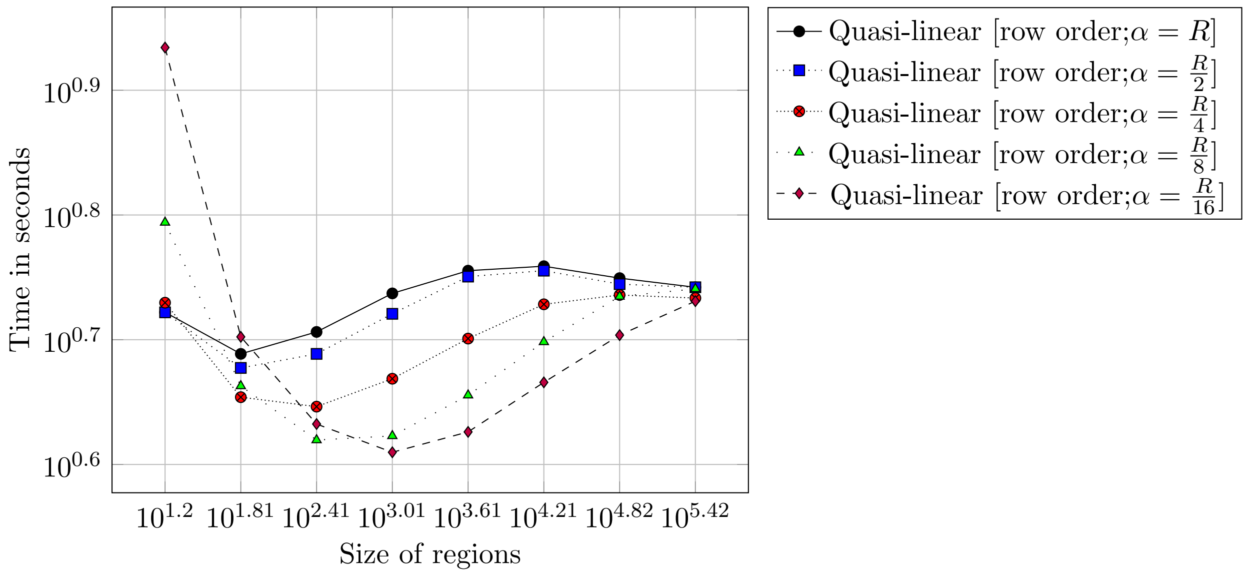


Figure 26: Running time results for the worst case data set with 10% distortion in a 4096 × 4096 grid graph, where region sizes and α values are varied for the Quasi-linear algorithm. For the experiments all the data structures and the input grid graph fit in memory.

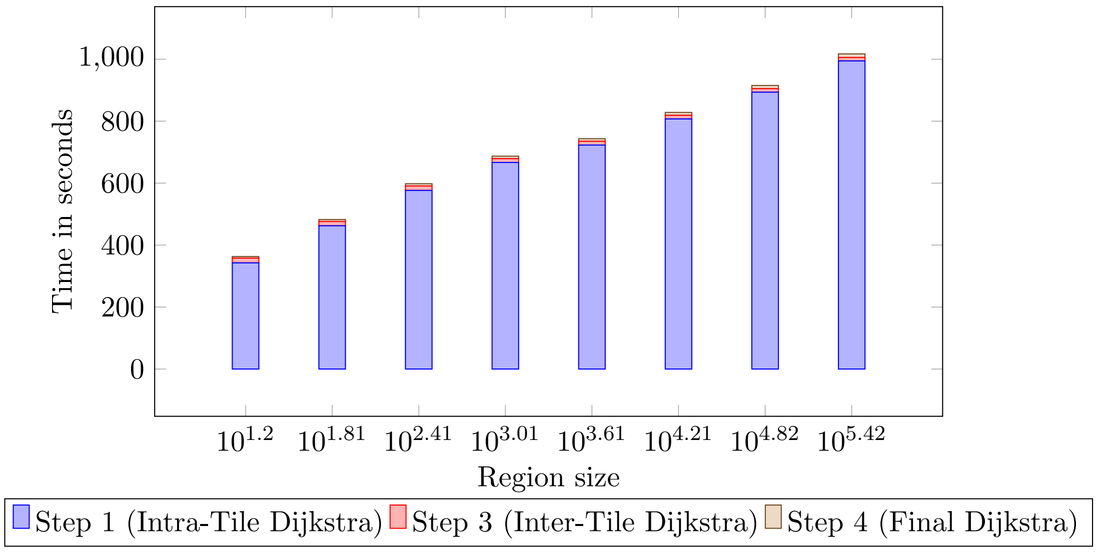


Figure 27: Running time results for the worst case data set with 10% distortion in a 4096 × 4096 grid graph, where the region sizes are varied for the Terra-efficient algorithm. The running times for the different sub steps of the Terra-efficient algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

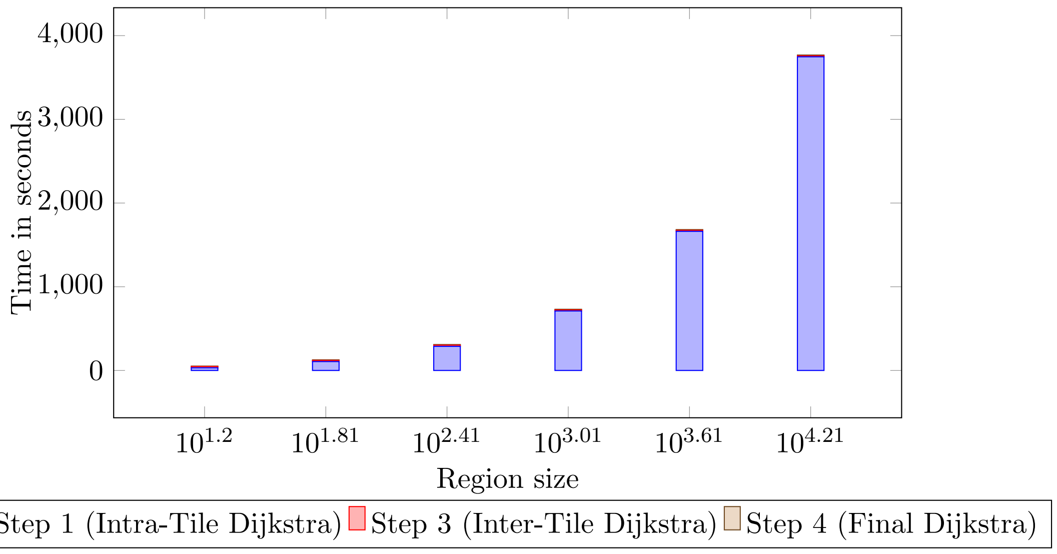


Figure 28: Running time results for the worst case data set with 10% distortion in a 4096 $\times$ 4096 grid graph, where the region sizes are varied for the Terra-simple algorithm. The running times for the different sub steps of the Terra-simple algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

5.1.4 Fully random

In this section we will present the results for the fully random data set as defined in section 4.2.3, where the entire grid and data structures fit in memory.

In figure 29 we see that Quasi-linear , Square and Dijkstra's algorithm perform much better then Terra-simple and Terra-efficient . This is simply due to the fact that Step 1 (Intra-Tile Dijkstra) of Terra-simple and Terra-efficient is costly for larger R as seen in 32 and 33.

Figure 30 shows that the Square and Quasi-linear algorithm always perform about the same for any R given a proper α. In 31 we see that larger R is preferred.

Comparison with the worst case data set with and without distortion As with the worst case data set with 10% distortion we see in 19 and 29 that the random graph is completed in less time then the worst case graph for Dijkstra's, Quasi-linear and Square , while for Terra-efficient and Terra-simple no real difference is seen. As with a worst case data set with 10% distortion we notice that larger R perform even better for Quasi-linear and Square for the fully random graphs. It is interesting to note here that for Quasi-linear a large α is now preferred over a smaller one for the other two data sets. For Terra-efficient we notice in figures 22, 27 and 32 that

Step 3 (Inter-Tile Dijkstra) takes even more time for fully random graphs, we again suspect that this is due to the number of modifications to the topology tree, which must be greater for fully random graphs.

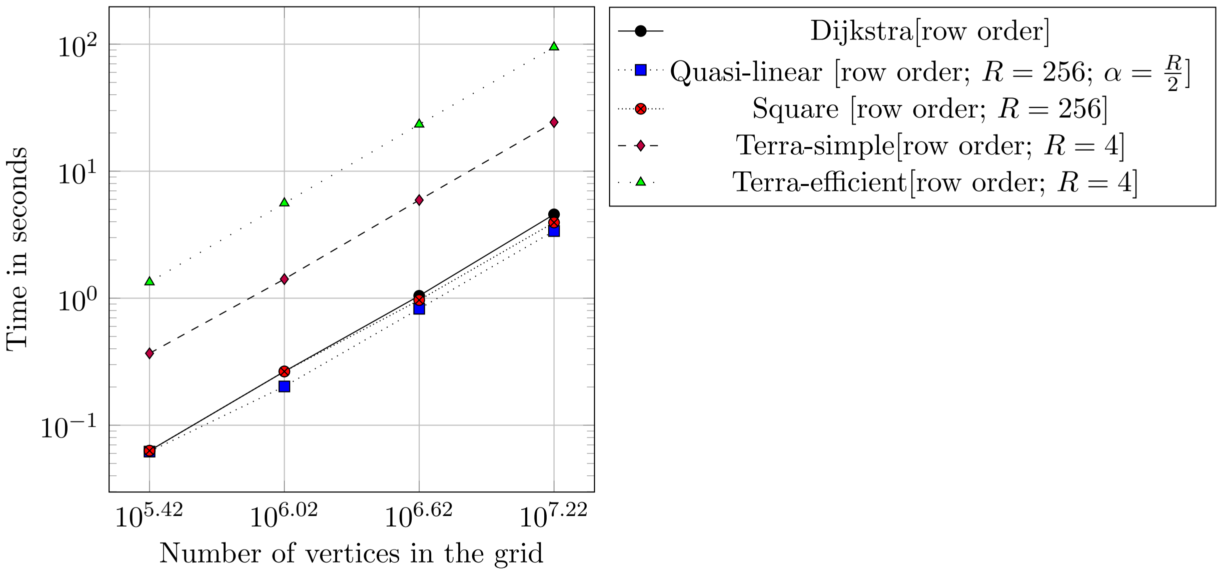


Figure 29: Best running time results for the fully random data set. For the experiments all the data structures and the input grid graph fit in memory.

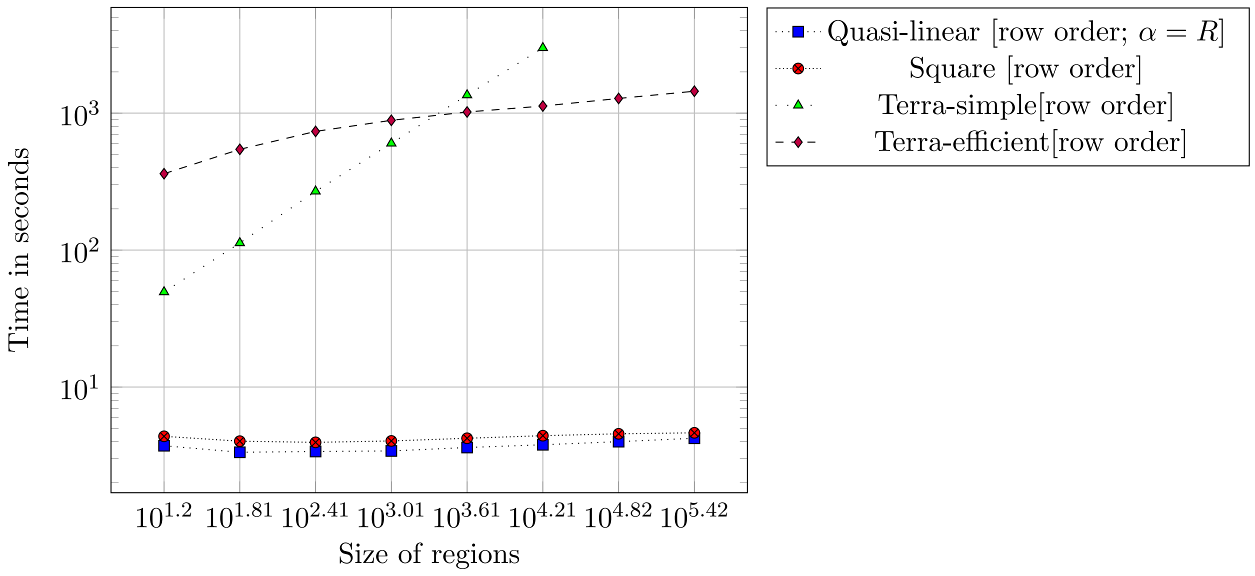


Figure 30: Running time results for the fully random data set in a 4096 × 4096 grid graph, where region sizes are varied. For the experiments all the data structures and the input grid graph fit in memory.

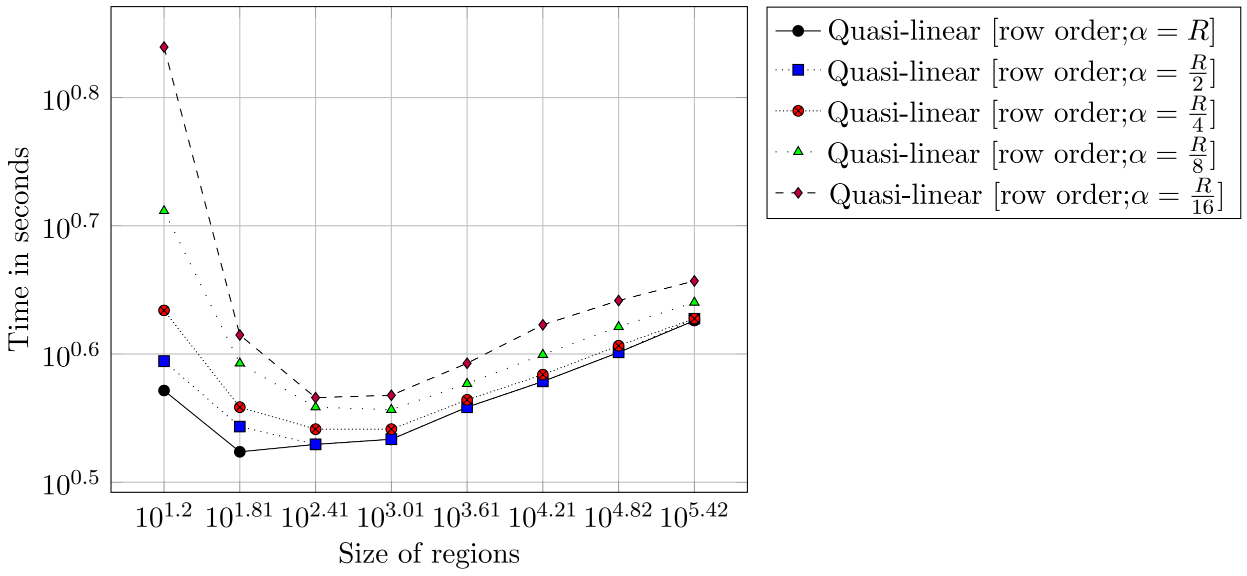


Figure 31: Running time results for the fully random data set in a 4096 × 4096 grid graph, where region sizes and α values are varied for the Quasi-linear algorithm. For the experiments all the data structures and the input grid graph fit in memory.

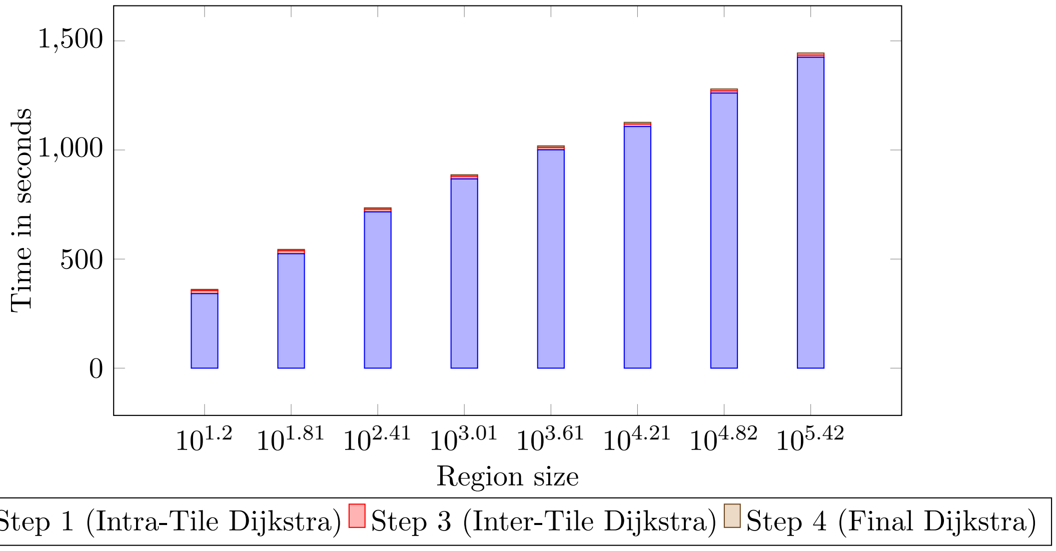


Figure 32: Running time results for the worst case data set with 10% distortion in a 4096 × 4096 grid graph, where the region sizes are varied for the Terra-efficient algorithm. The running times for the different sub steps of the Terra-efficient algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

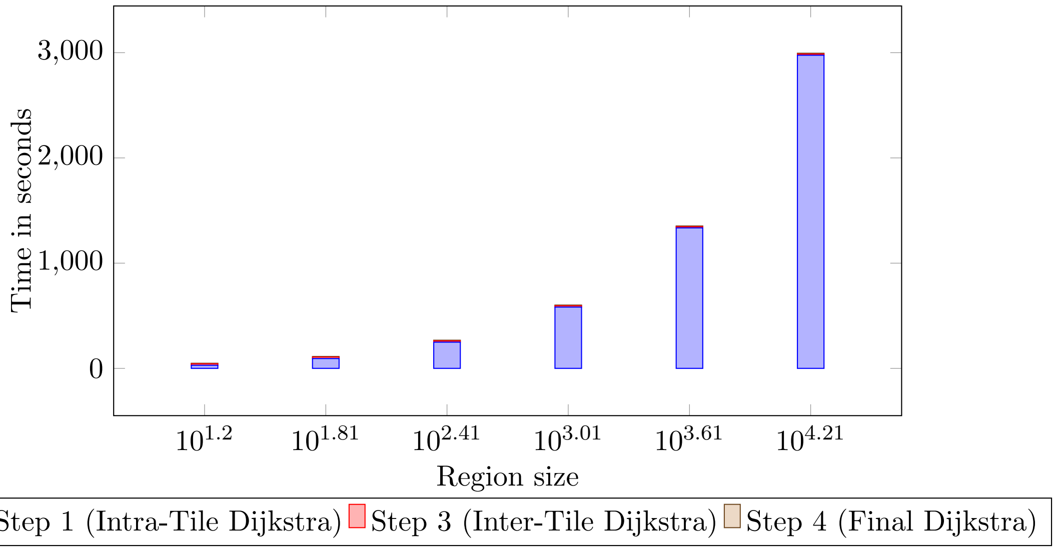


Figure 33: Running time results for the worst case data set with 10% distortion in a 4096 × 4096 grid graph, where the region sizes are varied for the Terra-simple algorithm. The running times for the different sub steps of the Terra-simple algorithm are also given. For the experiments all the data structures and the input grid graph fit in memory.

5.1.5 Highways and obstacles

In this section we present the results for the highways and obstacles data set, where all input data and data structures fit in memory. Figure 34 shows for each of the algorithms the best found running time. What we mainly notice is that the Square algorithm is slower then Dijkstra's and Quasi-linear , besides this we see that Dijkstra's algorithm is more vulnerable then the other two algorithm. With vulnerable we mean that a different δ value has a clear effect on the running time.

For the Quasi-linear algorithm there are very small differences between the different settings for R and α as can be seen in figures 35 and 36, but what we do notice is that with a larger R the running time increases by a seemingly constant factor and the same holds for a change in α even the difference between the lines is more vulnerable.

In figure 37 we see that for the Square algorithm the running times for the different δ and different R are small and nothing can really be deduced from this graph, besides the fact that a proper R must be chosen for a certain input graph.

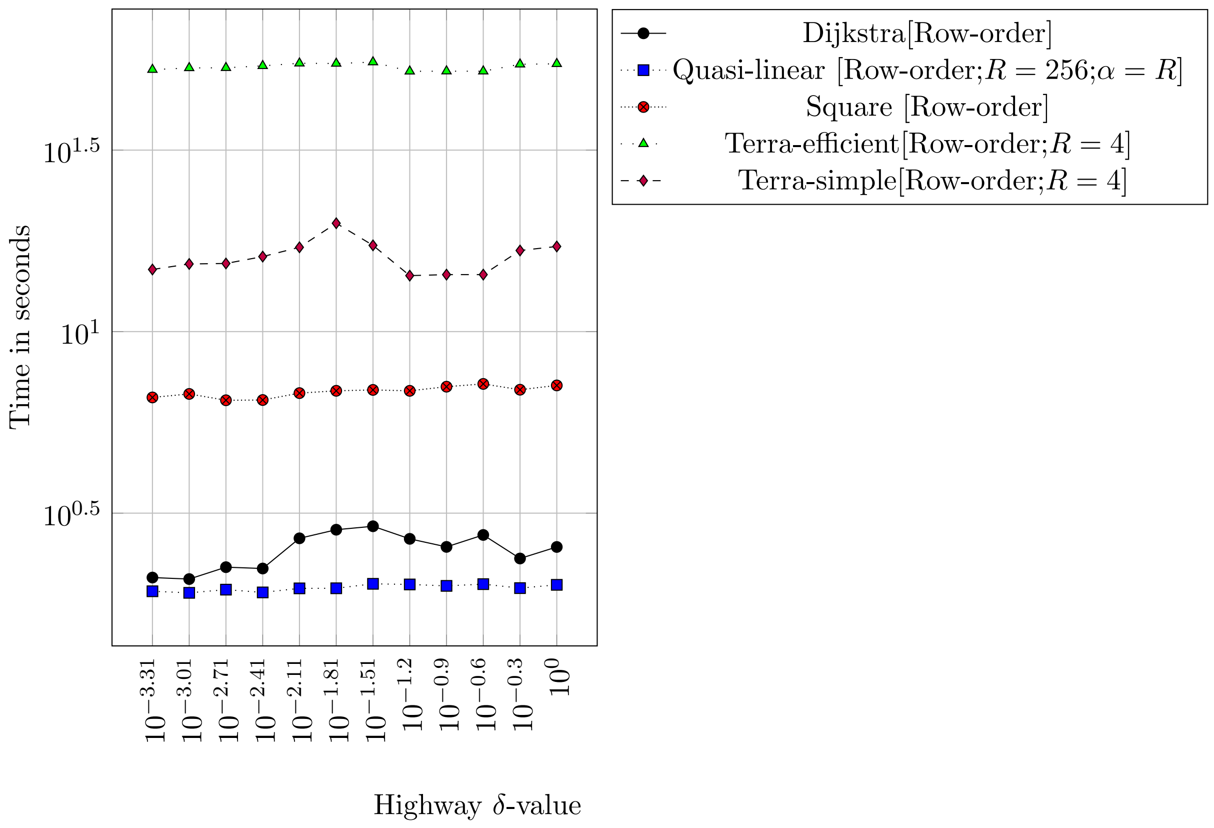


Figure 34: Best running time results for the highways and obstacles data set in a grid with 2048 × 2048 vertices. For the experiments all the data structures and the input grid graph fit in memory.

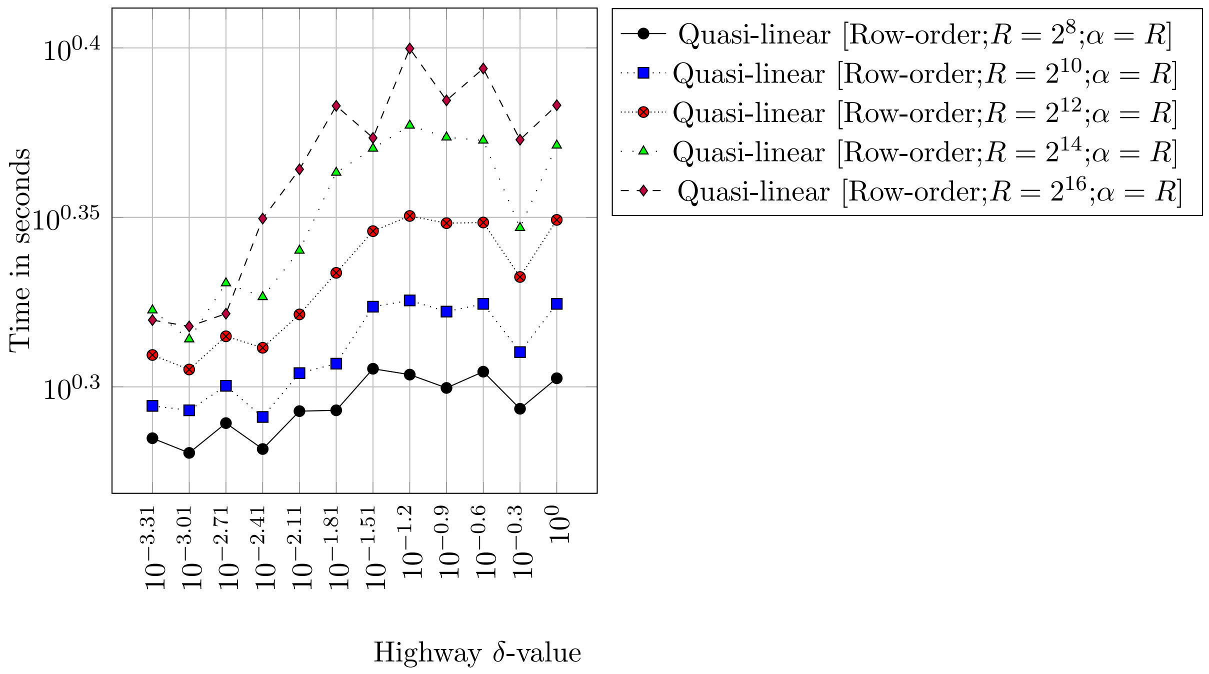


Figure 35: Best running time results for the highways and obstacles data set in a grid with 2048 × 2048 vertices for the Quasi-linear algorithm where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

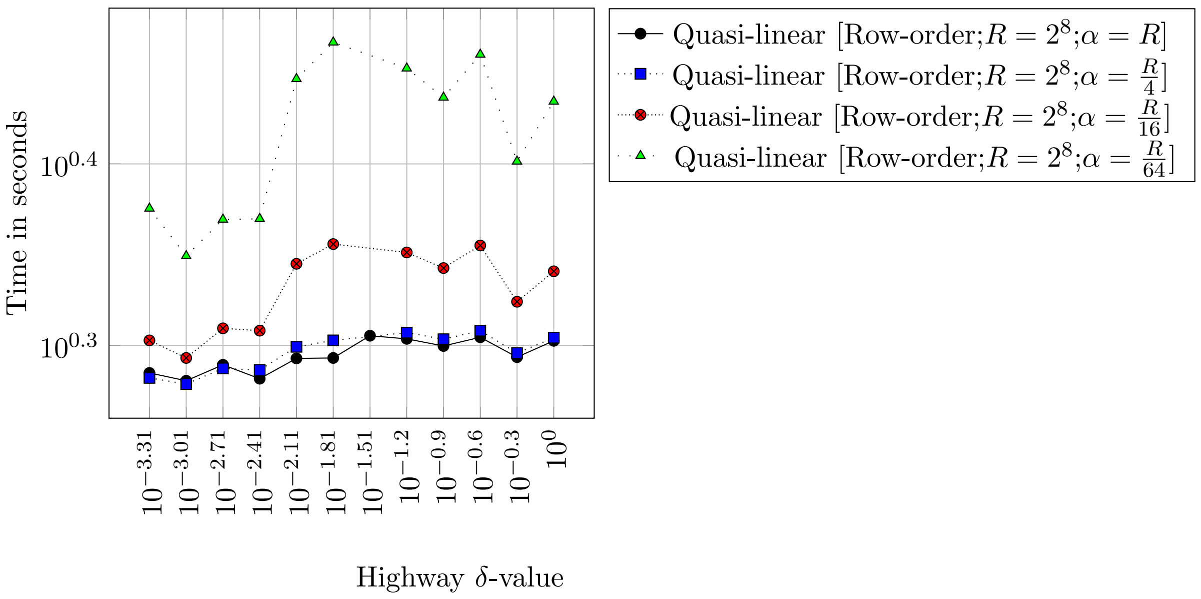


Figure 36: Best running time results for the highways and obstacles data set in a grid with 2048 × 2048 vertices for the Quasi-linear algorithm where α is varied. For the experiments all the data structures and the input grid graph fit in memory.

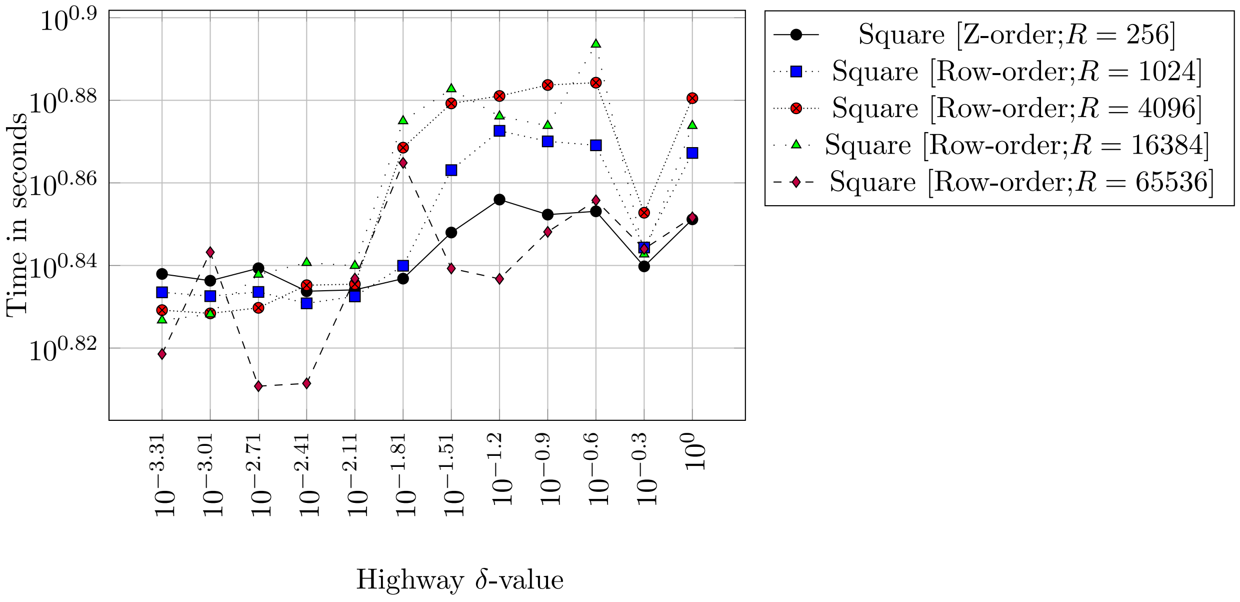


Figure 37: Best running time results for the highways and obstacles data set in a grid with 2048 × 2048 vertices for the Square algorithm where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

5.1.6 Real world data sets

In this section we present the results for the real world data set, where all input data and data structures fit in memory. Figure 38 shows for each of the algorithms the best running times for the different data sets. We notice here that Dijkstra's and Quasi-linear perform best, while Square , Terra-efficient and Terra-simple perform worse.

From figure 39 we conclude that a large α and a small R is preferred. For the α we notice that there is little difference between all $\frac{R}{64} \leq \alpha \leq \frac{R}{4}$. In figure 40 we notice that the region size for the Square algorithm does not seem to matter as much, while for Terra-efficient and Terra-simple it does, but for Terra-efficient and Terra-simple this is mainly due to Step 3 (Inter-Tile Dijkstra) of the algorithm as seen in figures 41 and 42.

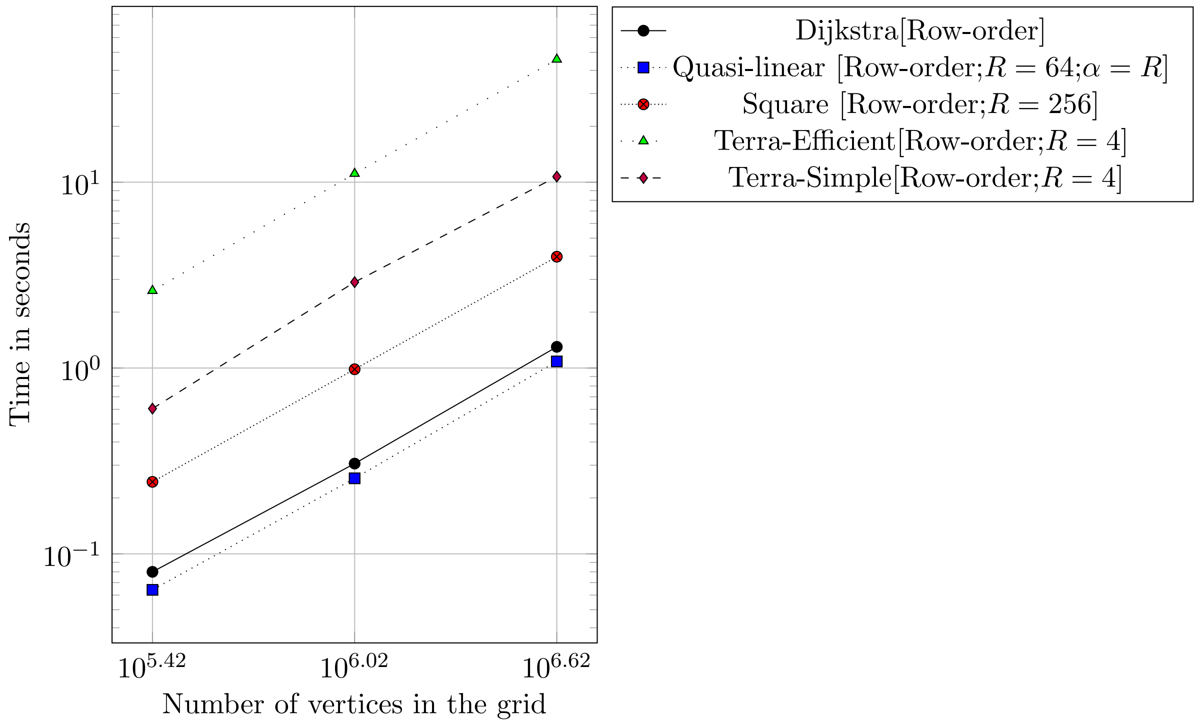


Figure 38: Best running time results for the real world data set. For the experiments all the data structures and the input grid graph fit in memory.

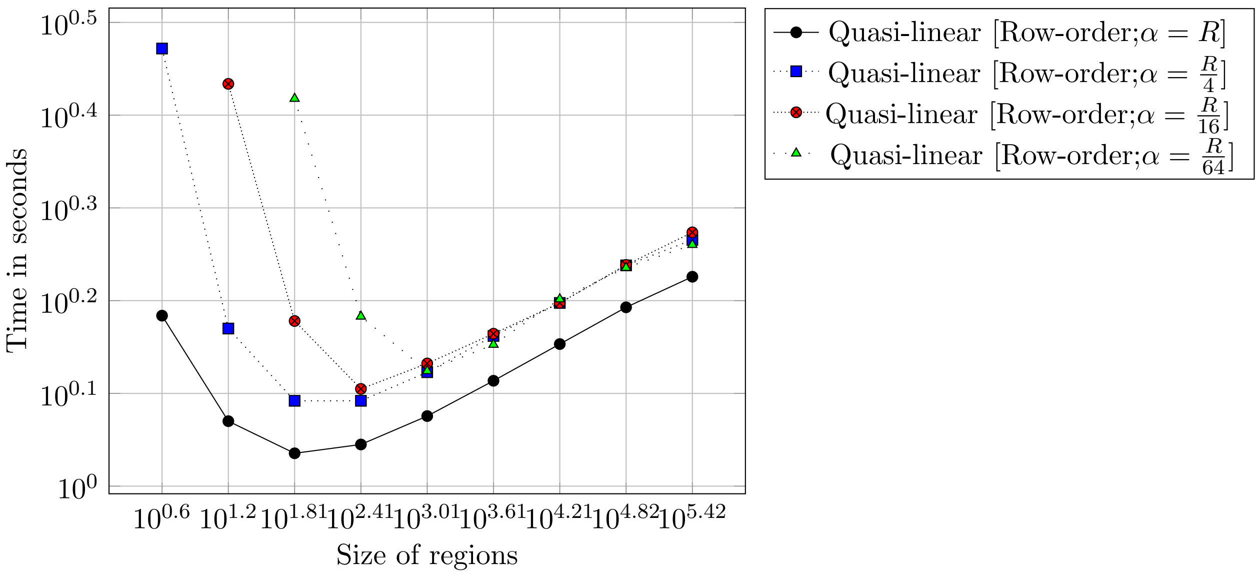


Figure 39: Best running time results for the real world data set in a grid with 2048× 2048 vertices for the Quasi-linear algorithm where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

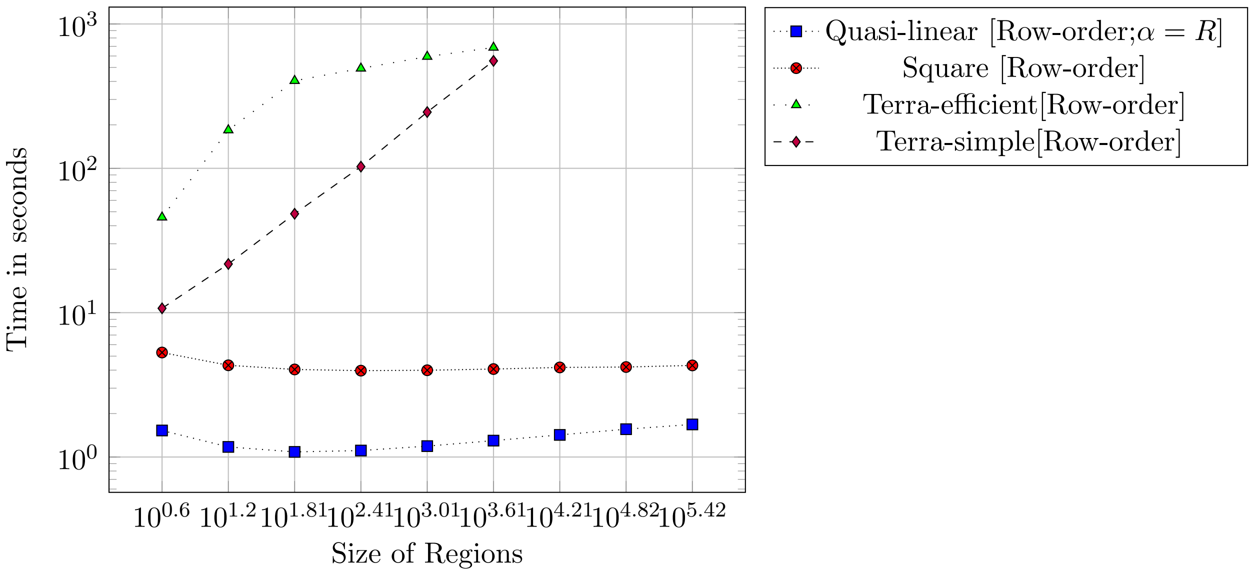


Figure 40: Best running time results for the real world data set in a grid with 2048 × 2048 vertices where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

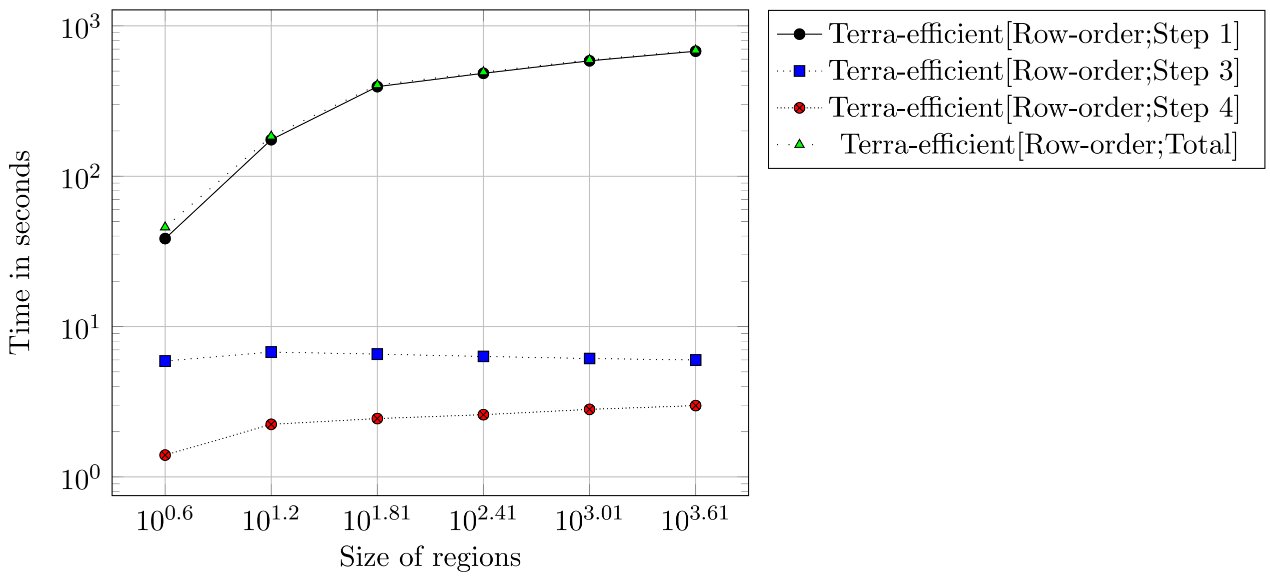


Figure 41: Best running time results for the real world data set in a grid with 2048 × 2048 vertices for the Terra-efficient algorithm. For the experiments all the data structures and the input grid graph fit in memory.

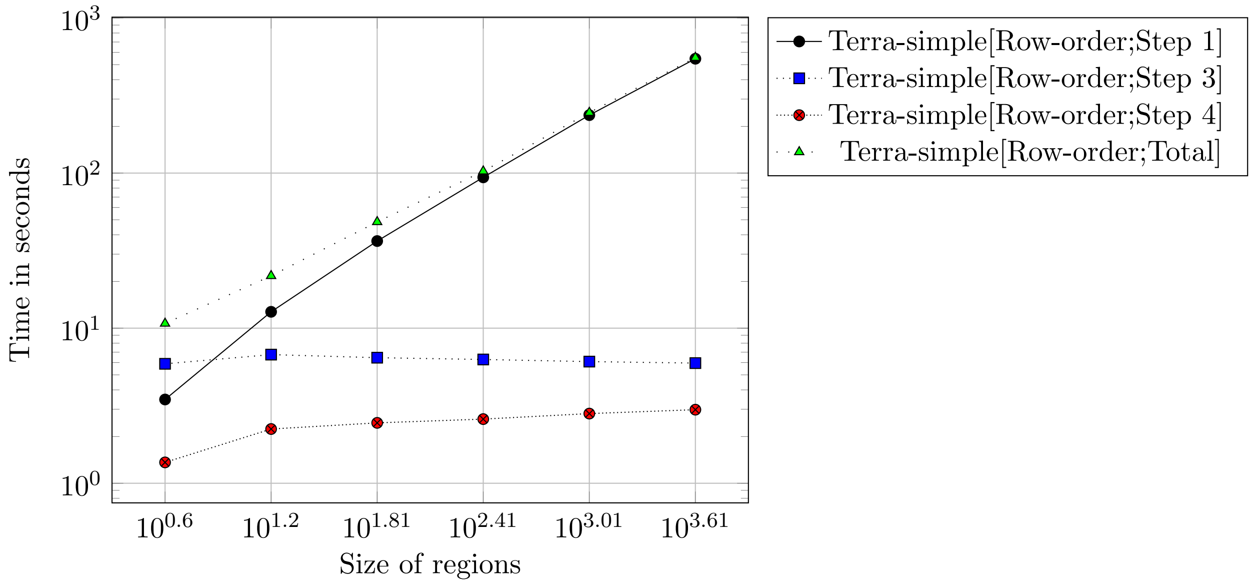


Figure 42: Best running time results for the real world data set in a grid with 2048 × 2048 vertices for the Terra-simple algorithm. For the experiments all the data structures and the input grid graph fit in memory.

5.1.7 Combined graphs and general thoughts

In figure 43 we see that for Square , Quasi-linear and Dijkstra's algorithm the worst case data set is the hardest to solve, while it does not seem to matter much for Terra-simple and Terra-efficient . The reason why the data set has little affect on Terra-simple and Terra-efficient is because Step 1 (Intra-Tile Dijkstra) takes by far the most time for both algorithms and since $R = 4$ is chosen the time to compute the SSSP for a 2×2 graph is practically constant. This observation is also supported by figures 22, 23 , 27, 28, 32 and 33, which are pretty much the same even if the data set is very different.

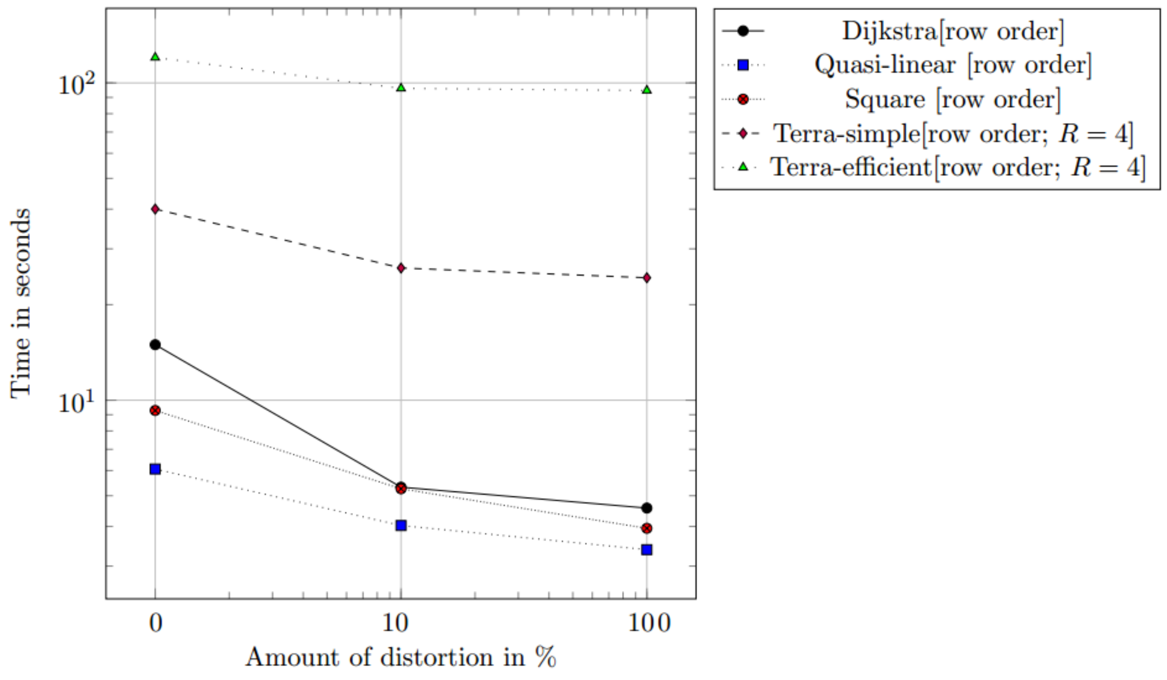
For the Square algorithm we see in figures 20, 25 and 30 that for the worst case data set a smaller region size performs better, while even if only 10% distortion is added the running time is seemingly independent from the chosen region size, which supports the idea that the worst case CPU bound for the Square algorithm should not be encountered for more practical data sets.

For the Quasi-linear algorithm we see three very different results for different R and α values as seen in figures 21, 26 and 31, but the best chosen R is always in the range of $256 \leq R \leq 1024$.

Dijkstra's algorithm performance suffers most from the worst case data set as seen in 43. This is in line with the argument made in section 5.1.2.

For the Quasi-linear and Square algorithm the chosen R and α for the real world and highways and obstacles data sets are similar to those of the worst case data set with 10% distortion and the fully random data set. Besides this we notice that Terra-efficient and Terra-simple still use up most of their time in Step 1 (Intra-Tile Dijkstra) for the real world and highways and obstacles data sets.

Z order versus row order In order to justify using Z order storage of the grid in an I/O setting we decided to add an experiment which shows the difference in running times for grid stored in Z order and grids stored in row order. Figures 44, 45 and 46 visualize several experiments where both grids in stored in row and Z order where tried out. We clearly see that for the worst case data set, seen in figure 44, there is very little to no difference between Z and row order for all algorithms. In figures 45 and 46 we see that for Dijkstra's algorithm there is no difference between storage in row or Z order and for Quasi-linear there is only little difference in most cases. For Square there is a clear cost of using Z order storage over that of row order, but this is just a small factor of the total running time. It is interesting to note here that for the Square algorithm a worst case data set with 10% distortion shows less difference between row and Z order then for a fully random data set. The reason for this is probably since the worst case data set with 10% distortion has vertical paths, while the fully random data set does not, hence probably Square with a Z order stored graph has more locality then Square with row order stored graph.



	Worst case	Worst case With 10% distortion	Fully random
Amount of distortion	0%	10%	100%
Quasi-linear [Z-order](R, α)	$(256, \frac{R}{16} = 16)$	$(1024, \frac{R}{16} = 64)$	$(256, \frac{R}{2} = 128)$
Square [Z-order](R)	16	16384	256

Figure 43: Best running time results for the worst case data set with 0%, 10% and 100% distortion (fully random) on a grid of size 4096 $\times$ 4096 grid. For the experiments all the data structures and the input grid graph fit in memory.

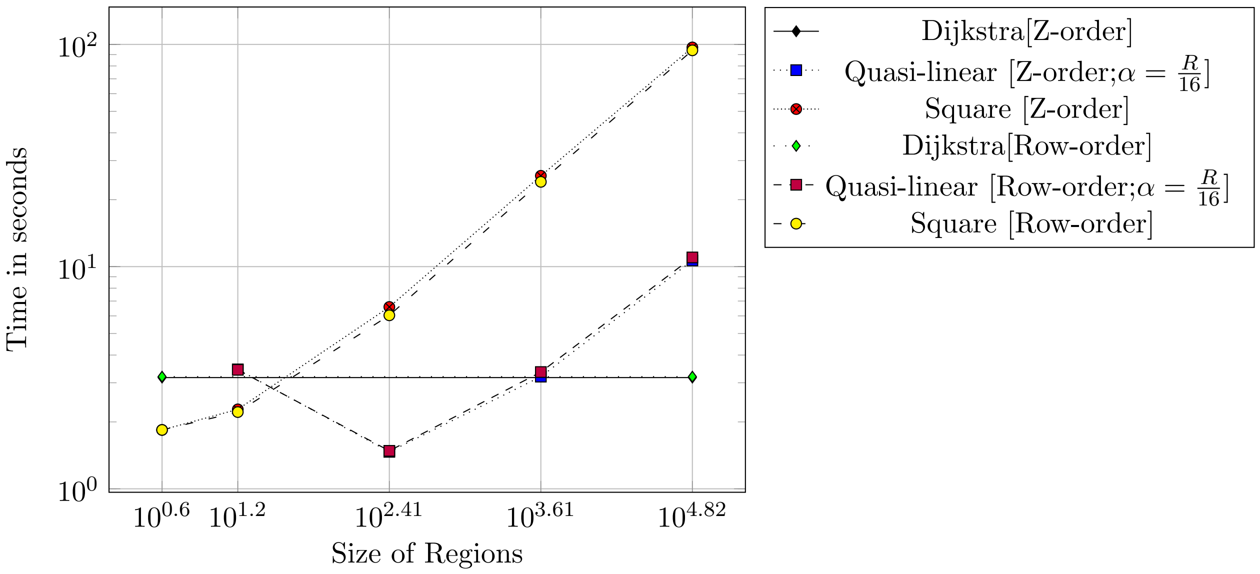


Figure 44: Running time results for the worst case data set in a grid with 2048 $\times$ 2048 vertices where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

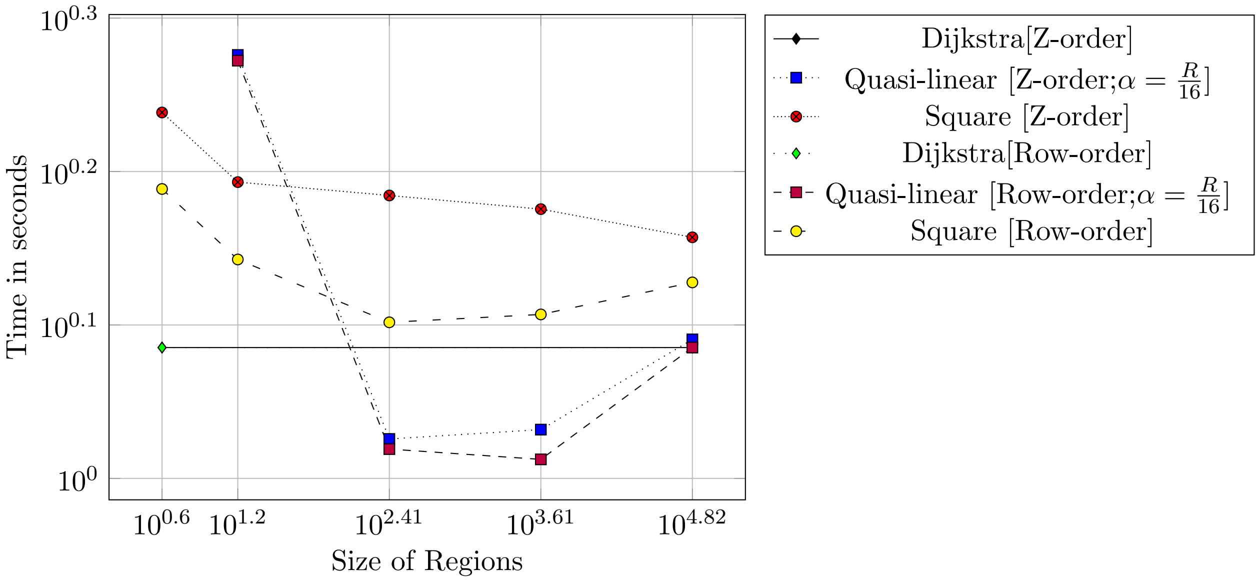


Figure 45: Running time results for the worst case data set with 10% distortion in a grid with 2048 $\times$ 2048 vertices where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

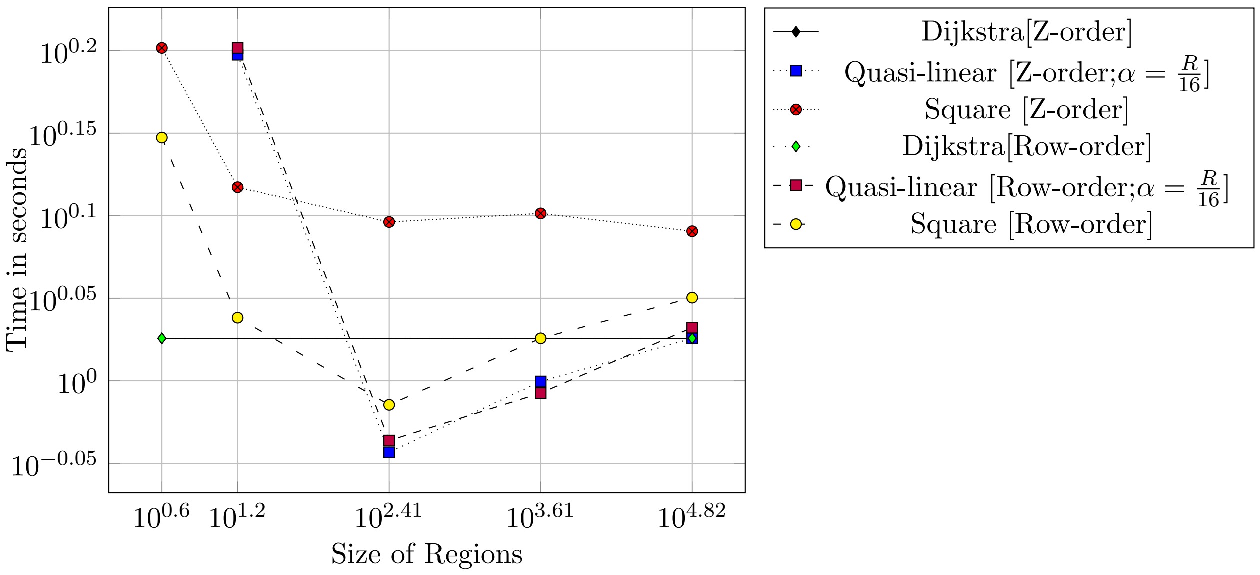


Figure 46: Running time results for the fully random data set in a grid with 2048 $\times$ 2048 vertices where R is varied. For the experiments all the data structures and the input grid graph fit in memory.

5.2 I/O

In the following sections we will present the results for the different data sets as defined in section 4.2, when the amount of available memory is limited. We have chosen to set the total amount of available memory to 384 Megabytes, which should be small enough if you consider the memory needed to store a grid as seen in figure 47. Figure 47 also clearly shows that swapping will occur at a grid of size 4096 $\times$ 4096 and should be clearly visible at a grid of size 8192 $\times$ 8192, since the amount of memory needed to store the grid is $\frac{1280}{384} > 3$ times greater than the amount of available memory.

5.2.1 Machine

The hardware which was used to run the I/O experiments presented in sections 5.2.3, 5.2.4 and 5.2.5 was:

- Unknown CPU ?
- 384 Megabytes of internal memory.
- 64-bit Linux operating system and executables.

Number of vertices (n)	Size in memory in Megabytes
$n = 10^{5.42}$ or 512×512	$\frac{n \cdot 20}{1024 \cdot 1024} = 5$ MB
$n = 10^{6.02}$ or 1024×1024	$\frac{n \cdot 20}{1024 \cdot 1024} = 20$ MB
$n = 10^{6.62}$ or 2048×2048	$\frac{n \cdot 20}{1024 \cdot 1024} = 80$ MB
$n = 10^{7.22}$ or 4096×4096	$\frac{n \cdot 20}{1024 \cdot 1024} = 320$ MB
$n = 10^{7.83}$ or 8192×8192	$\frac{n \cdot 20}{1024 \cdot 1024} = 1280$ MB
$n = 10^{8.43}$ or 16384×16384	$\frac{n \cdot 20}{1024 \cdot 1024} = 5120$ MB
$n = 10^{9.03}$ or 32768×32768	$\frac{n \cdot 20}{1024 \cdot 1024} = 20480$ MB

Figure 47: Amount of memory which is needed to store the grid graph. Each vertex requires 20 bytes of storage.

The hardware which was used to run the I/O experiments presented in sections 5.2.6 and 5.2.7 was:

- Intel core 2 E6600 at 2400 MHz.
- 384 Megabytes of internal memory.
- 64-bit Linux operating system and executables.

The reason for using two different systems for these different data sets was to be able to run all the experiments within the limited amount of time which was available. Just a note for the reader non of the experimental results between these two machines will be compared with each other.

5.2.2 Settings

In order to run the I/O experiments within the limited time we have we have chosen to run all I/O experiments on grid graphs which are stored in Z-order. As shown in section 5.1.7 we know that using Z-order only adds a little bit of extra computation cost in the worst case, while in our theoretical analysis we have concluded that Z-order storage has in most cases a better I/O bound then row order stored grid graphs.

For the Terracost algorithm we have chosen to only run Terra-efficient and not Terra-simple . We have done this because as we will show in the I/O experiments larger values for R are preferred in which case as we have seen in the in memory experiments Terra-efficient outperforms Terra-simple . Moreover we will show that this preference of larger R is due to Step 3 (Inter-Tile Dijkstra) of the algorithm, which means that the choice between Terra-simple and Terra-efficient does not affect this preference for larger R.

5.2.3 Worst case

In this section we will present the results for the worst case data set as defined in section 4.2.1. Figure 48 visualizes the best results for each of the algorithms for different grid sizes.

Dijkstra's algorithm One of the things which is noticeable is that Dijkstra's algorithm performs better for $n = 10^{6.62}$ then the Quasi-linear and Square algorithms, but blows up for $n = 10^{7.22}$. The reason why Dijkstra's algorithm first performs better is because for $n = 10^{6.62}$ the grid is about 80 Megabytes in memory and depending on the implementation the priority queue should not be much more then $16 \cdot n = 16 \cdot 10^{6.62} \approx 64$ Megabytes. This means that a total of $64 + 80 = 144$ Megabytes should be small enough to still be fully in memory, since the total available memory size is 384 Megabytes. While for $n = 10^{7.22}$ the grid itself already is 320 Megabytes, hence with the added priority queue I/Os will be needed to complete the algorithm and since the worst case data set ensures that after all the edges with weight zero have been relaxed there will be $O(n)$ vertices in the priority queue, which all might need an I/O to relax. They might need an I/O since the entire grid does not fit in memory and since the order in which they are relaxed is random and is not related with the order in which they are stored on disk.

Quasi-linear and Square For the Square and the Quasi-linear algorithms a small jump in running times is visible between $n = 10^{6.02}$ and $n = 10^{6.62}$ the reason for this is because if $n = 10^{6.02}$ then the grid is only 20 Megabytes in memory, while for $n = 10^{6.62}$ the grid is 80 Megabytes. Both of these values still fit in memory, but the additional data structures increase the total needed memory size slightly above the 384 Megabytes boundary, which means that some I/Os are needed for $n \geq 10^{6.62}$. The graph visualizes this clearly.

For the Quasi-linear algorithm a region size of $R = 2^{14} = 16384$ seems to yield the best results for larger grids. Besides this a small α is preferred. This is also confirmed by figure 49 in which we clearly see the sweet spot being at $2^{12} = 10^{3.61} \leq R \leq 10^{4.82} = 2^{16}$.

For the Square algorithm a best region size of $R = 2^{10} = 2^{3.01}$ is found for larger n. This is also confirmed by figure 50, where we see that smaller region sizes perform better then larger ones. The reason why smaller regions are preferred is probably because this decreases CPU bound, but also since this data set at the most sets $4 \cdot \sqrt{R}$ vertices to their final value, hence larger R means that more vertices are read in which are not relaxed to their final value and thus have to be relaxed again at a later stage. Besides that $\Theta(R) = \Theta(\sqrt{B})$ was suppose to give the best I/O bound as explained in 3.4.3. If we now forget about the constants and fill in $20 \cdot R = 20 \cdot 1024 = 20480 \approx \sqrt{B} \Rightarrow B = 20480^2$ then we see that this makes no sense since B can't be that large. But this is not the only factor to take into account, since if $\sqrt{B} \cdot \sqrt{n}$ fits in memory, then because of the way the worst case data set is constructed we know that $O(\frac{n}{B})$ I/Os are needed at the most. This is because the path of edges with value zero are relaxed first and once this is done everything else is relaxed, but these zero edges are one long line going up and down, hence if the height of the grid and the width of a block of vertices on disk fit in memory we get the optimal access pattern. If we now fill in the numbers we get:

$$\sqrt{n} \cdot \sqrt{B} \leq \sqrt{10^{8.43}} \cdot \sqrt{B} \leq 16384 \cdot \sqrt{B}$$

Hence if $16384 \cdot \sqrt{B} < M = 384$ Megabytes holds then $O(\frac{n}{B})$ I/Os should be needed. For if we assume some large $B = 2^{20}$ then we get:

$$16384 \cdot \sqrt{B} = 16384 \cdot \sqrt{2^{20}} = 16384 \cdot 2^{10} = 16 \cdot 2^{20} < M = 384 \cdot 2^{20}$$

$$\Longrightarrow$$

$$16 < M = 384$$

Hence everything should easily fit in memory to achieve the $O(\frac{n}{B})$ I/O bound. This same idea also holds for the Quasi-linear algorithm. The reason why Quasi-linear block size is about a factor $2^4 = 16$ larger then that of the Square algorithm is because Square always accesses from 9 up to 21 regions while Quasi-linear only accesses from 1 up to 5.

Terra-efficient The Terra-efficient algorithm as seen in figure 48 performs best for a large block size of $R = 65536$, this is because $R = B^2$ should give the optimal I/O bound of $O(\frac{n}{B})$ unfortunately it is doubtful that $B = 256$, even when adding a constant factor the B, is large enough to come close to the operating system block size on disk, which usually is at least 32 kilobytes. To get a nice running time you might even need larger block sizes then 32 kilobytes, but the problem is that a larger R is not possible since Step 1 (Intra-Tile Dijkstra) would not fit in memory any more and since the boundary to boundary calculation is not I/O efficient we would again get $O(n)$ I/Os from Step 1 (Intra-Tile Dijkstra) alone. The problem here is the same as with Dijkstra's algorithm, since $R < B^2$ we get that Step 3 (Inter-Tile Dijkstra) is not totally I/O efficient, since each vertex which is relaxed reads a total of $256 \cdot 4$ outgoing edges from this vertex. But since the next vertex which will be extracted from the priority queue can be anywhere inside the grid we get that for each vertex which is extracted we might need one I/O, which means that instead of a division by B we get a division by some amount significantly smaller then B (namely $256 \cdot 4 \cdot 4$).

For the Terra-efficient algorithm we see in figure 52 that the running time of Step 3 (Inter-Tile Dijkstra) is dependent on the region size, which is exactly what we would expect since how larger R is the closer it will get to B^2 (as we argued above). Note here that figure 52 are results for a 4096 $\times$ 4096 grid graph, while figure 53 are the results for a 16384 $\times$ 16384 grid graph. Step 1 (Intra-Tile Dijkstra) in figure 52 seems to not mind which region size is chosen. Figure 53 clearly shows where the most time is lost for Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm. To completely understand this we calculate how many vertices are part of the path with edge weight zero between them:

$$\frac{n}{R} \cdot \frac{4 \cdot \sqrt{R}}{3}$$
$$=$$
$$\frac{16384^2}{65536} \cdot \frac{4 \cdot \sqrt{65536}}{3}$$
$$=$$
$$4096 \cdot \frac{1024}{3}$$
$$=$$
$$\frac{4096 \cdot 1024}{3}$$
$$=$$
$$1398101$$
$$\approx$$
$$5.33 \cdot 10^{5.418}$$

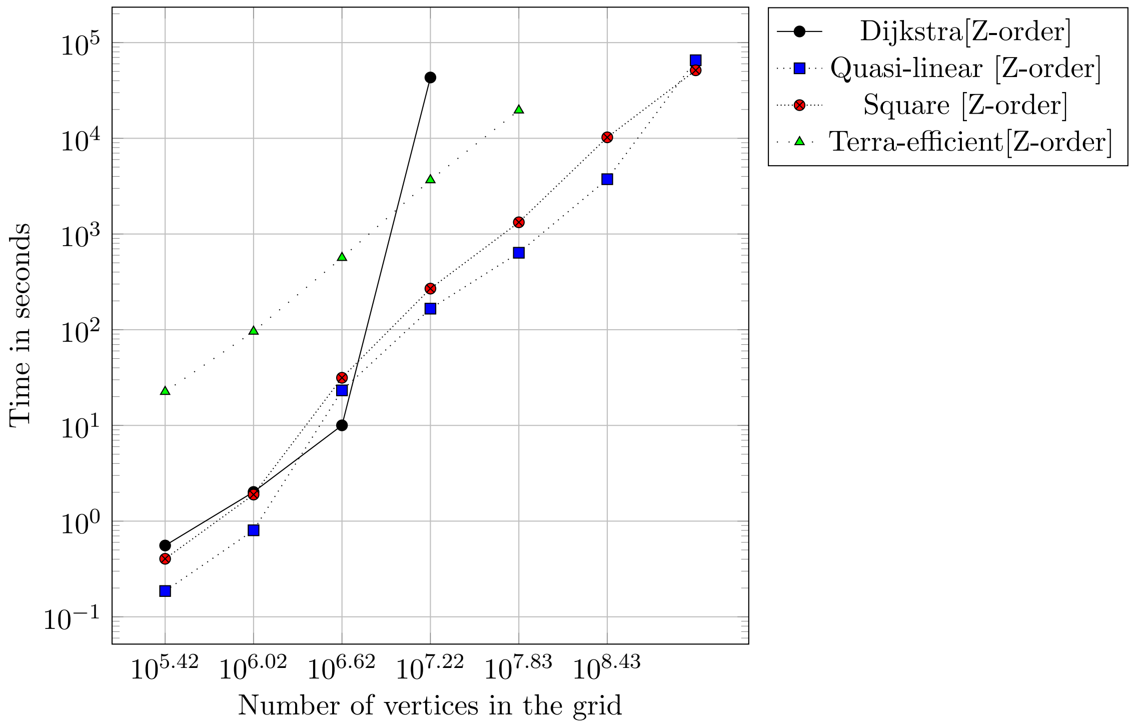
Where $\frac{n}{R}$ is the number of regions and $\frac{4 \cdot \sqrt{R}}{3}$ is about the number of vertices in each region which are part of this zero weight path. What is interesting to note here is that exactly between $5 \cdot 10^{5.418}$ and $6 \cdot 10^{5.418}$ the time required to calculate the next $10^{5.418}$ goes up drastically. If we assume that the algorithm won't just speed up and take the last couple of lines from figure 53 to be the trend which is followed until the program is done, then we can calculate the total running time for Step 3 (Inter-Tile Dijkstra) (as calculated in the description of figure 53 the total number of vertices to be relaxed is $16 \cdot 10^{5.418}$):

$$(16-9) \cdot (87511-64182)+87511 = 163303+87511 = 250814 \approx 3600 \cdot 69.67$$

Hence if the last couple of lines in figure 53 are correct, then it would take 69 days total to finish Step 3 (Inter-Tile Dijkstra) of the algorithm.

In order to further show that the problem for the Terra-efficient is the region size, which is limited by the amount of memory available to run Step 1 (Intra-Tile Dijkstra) we have done an additional experiment where we increased the memory to 2048 Megabytes. Figure 54 shows the time required to relax x vertices in a 16384×16384 grid graph in Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm. What is clearly noticed in this graph is that whichever value for R is chosen the time required to relax $10^6 \cdot 0.27$ vertices takes about the same amount of time once the first $\frac{1}{3}$ vertices have been relaxed. Figure 55 shows the same thing, but the x-axis is now the percentage of vertices done. The exact numbers for the speed up are given in figure 56 and clearly show what we saw in figure 55.

Due this we can conclude that in a normal setting where more memory would be available for Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm a larger region size could be chosen and the running time of Step 3 (Inter-Tile Dijkstra) of the algorithm will go down. Besides this we clearly see in 56 that a larger region size is far less costly for Step 1 (Intra-Tile Dijkstra) than the amount of time which is gained in Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm. Thus this also clearly shows that in order for the Terra-efficient algorithm to properly work for worst case data sets a topology tree must be used in order to reach the $B^2 = R$.



Number of vertices (n)	$10^{5.42}$	$10^{6.02}$	$10^{6.62}$	$10^{7.22}$	$10^{7.83}$	$10^{8.43}$
Quasi-linear [Z-order](R, α)	$(2^6, \frac{R}{4})$	$(2^6, \frac{R}{4})$	$(2^{12}, \frac{R}{2})$	$(2^{14}, \frac{R}{8})$	$(2^{16}, \frac{R}{32})$	$(2^{16}, \frac{R}{16})$
Square [Z-order](R)	2^{16}	2^{18}	2^{8}	2^{8}	2^{10}	2^{10}
Terra-eff[Z-order](R)	2^{6}	2^{16}	2^{16}	2^{16}	2^{16}	

Figure 48: Best running time results for worst case data set with 384 MB available memory.

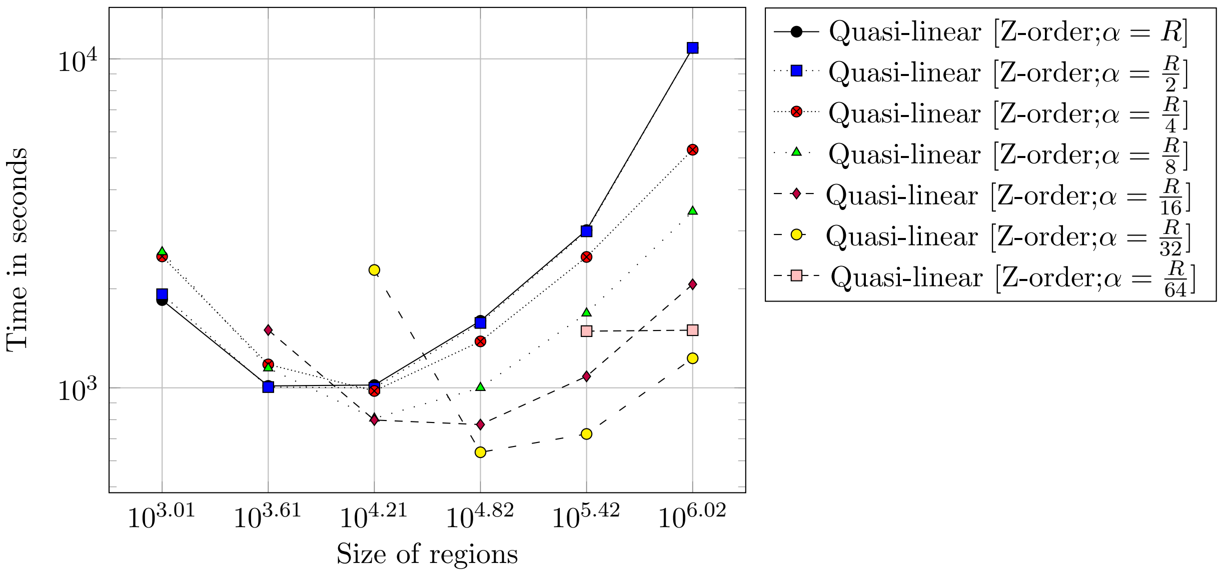


Figure 49: Running time results for the worst case data set in a 8192 $\times$ 8192 grid graph with only 384 MB memory available, where the region size and α values are varied for the Quasi-linear algorithm. For $\alpha = \frac{R}{64}$ and $R = 10^{4.82}$ the experiment did not finish within 1 hour or $3600 = 10^{3.55}$ seconds.

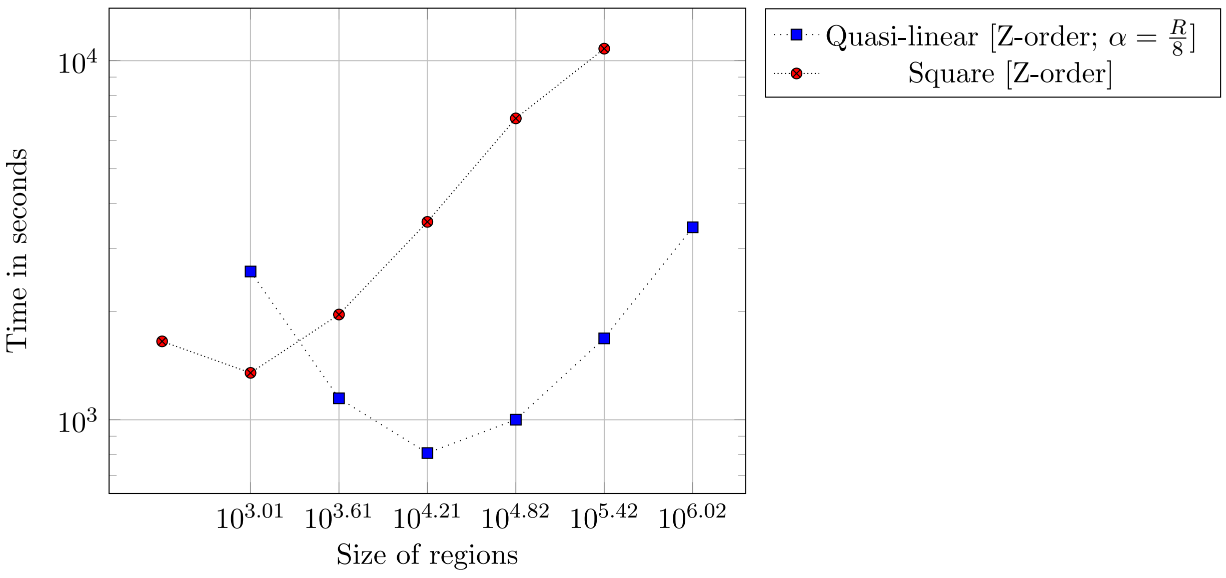


Figure 50: Running time results for the worst case data set in a 8192 $\times$ 8192 grid graph with only 384 MB memory available, where the region size is varied. Process is cut off at 3 hours = $10^{4.03}$ seconds.

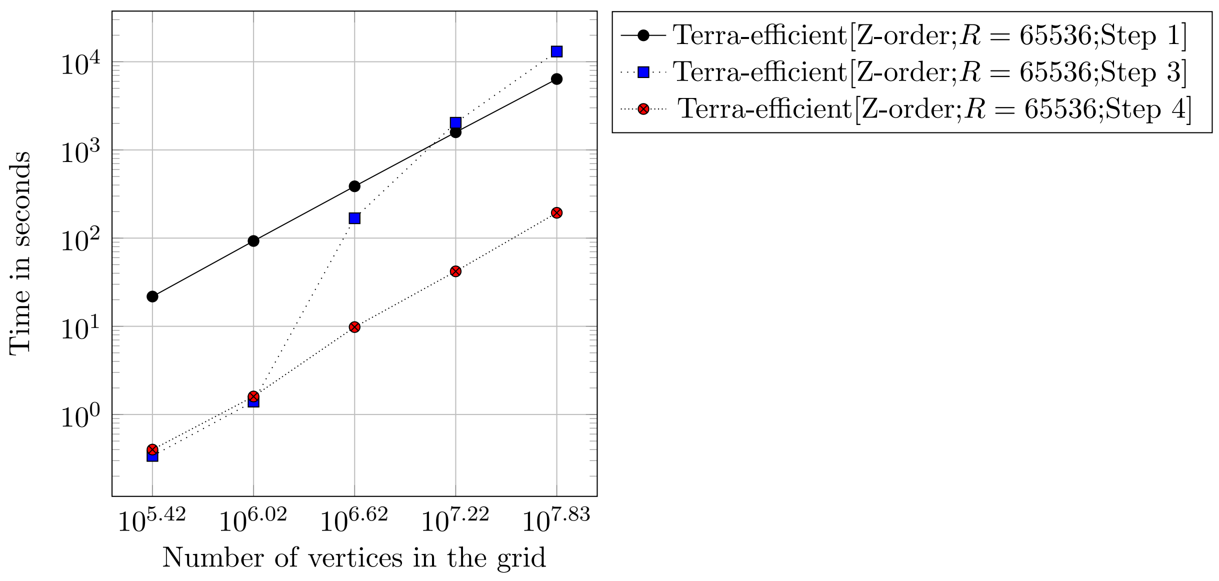


Figure 51: Running time results for the worst case data set in a 4096 × 4096 grid graph with only 384 MB of memory available, where the region sizes are varied for the different steps of the Terra-efficient algorithm.

Region size (R)	$10^{2.41}$	$10^{3.01}$	$10^{3.61}$	$10^{4.21}$	$10^{4.82}$	$10^{5.42}$
Terra-eff[Z-order,Step 1]	1497	1830	1644	1697	1697	∞
Terra-eff[Z-order,Step 3]	∞	∞	∞	11763	7665	-
Terra-eff[Z-order,Step 4]	-	-	-	110	121	-
Terra-eff[Z-order,Total]	∞	∞	∞	13570	9484	∞

Figure 52: Running time results in seconds for the worst case data set in a 4096 × 4096 grid graph, with only 384 MB memory available, when running the Terra-efficient algorithm. A ∞ denotes that the algorithm did not end within 4 hours or 14400 seconds. A $-$ denotes that a previous step didn't end within 4 hours, hence its unknown what running time it has.

Number of vertices extracted from priority queue (Meaning number of vertices which received their final value)	Total running time in seconds for Step 3 (Inter-Tile Dijkstra)
$1 \cdot 10^{5.418}$	1322
$2 \cdot 10^{5.418}$	2650
$3 \cdot 10^{5.418}$	4000
$4 \cdot 10^{5.418}$	5337
$5 \cdot 10^{5.418}$	6670
$6 \cdot 10^{5.418}$	19667
$7 \cdot 10^{5.418}$	42029
$8 \cdot 10^{5.418}$	64182
$9 \cdot 10^{5.418}$	87511
$10 \cdot 10^{5.418}$	∞

Figure 53: Running time results in seconds for the worst case data set in a 16384×16384 grid graph, with only 384 MB memory available, when running the Terra-efficient algorithm. The region size, R, was set to 65536. It shows the progression of Step 3 (Inter-Tile Dijkstra) of the algorithm. The algorithm was stopped at $10 \cdot 10^{5.418} = 2618183$, since Step 3 (Inter-Tile Dijkstra) alone was already running for 1 day and the total number of vertices which had to be given a final value was:
$\frac{16384^2}{R} \cdot (2 \cdot \sqrt{R} + 2 \cdot (\sqrt{R} - 2) = \frac{16384^2}{65536} \cdot (2 \cdot \sqrt{65536} + 2 \cdot (\sqrt{65536} - 2)$
$\approx \frac{16384^2}{65536} \cdot (4 \cdot \sqrt{65536}) = \frac{16384^2}{65536} \cdot (4 \cdot \sqrt{65536}) = 4194304$
$\approx 10^{6.62} \approx 16 \cdot 10^{5.418}$.
Step 1 (Intra-Tile Dijkstra) took 24889 seconds or nearly 7 hours to complete.

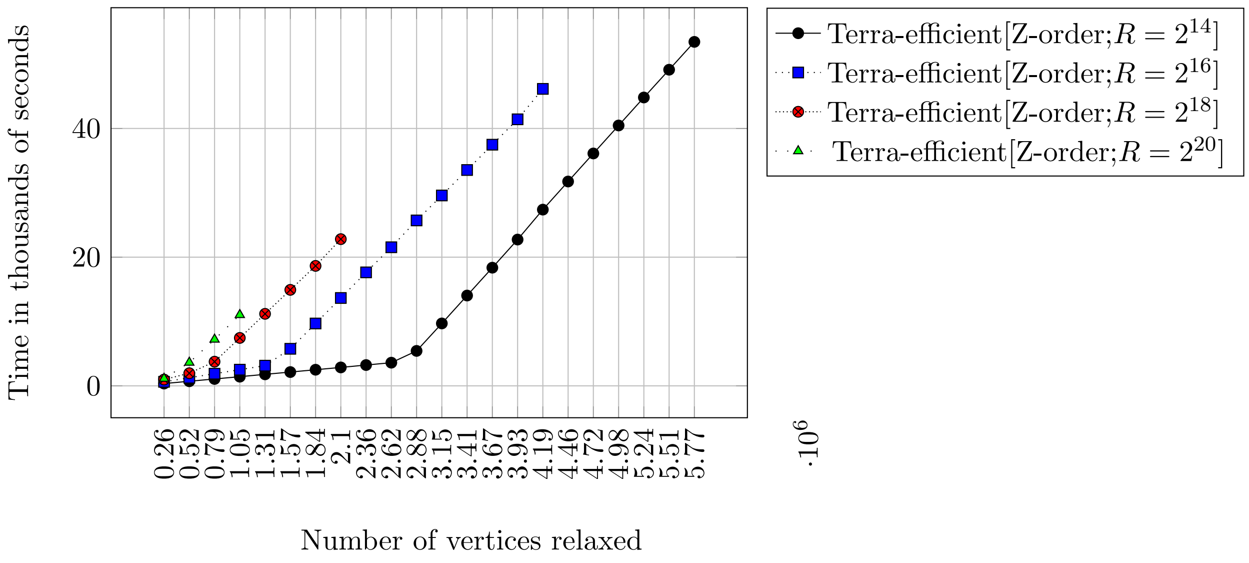


Figure 54: Time required to relax x vertices in a 16384×16384 grid for Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm, with only 2048 MB of memory available. The cut off time was 55000 seconds, meaning that Terra-efficient[Z-order;$R = 2^{14}$] is not done after it has relaxed $5.77 \cdot 10^6$ vertices.

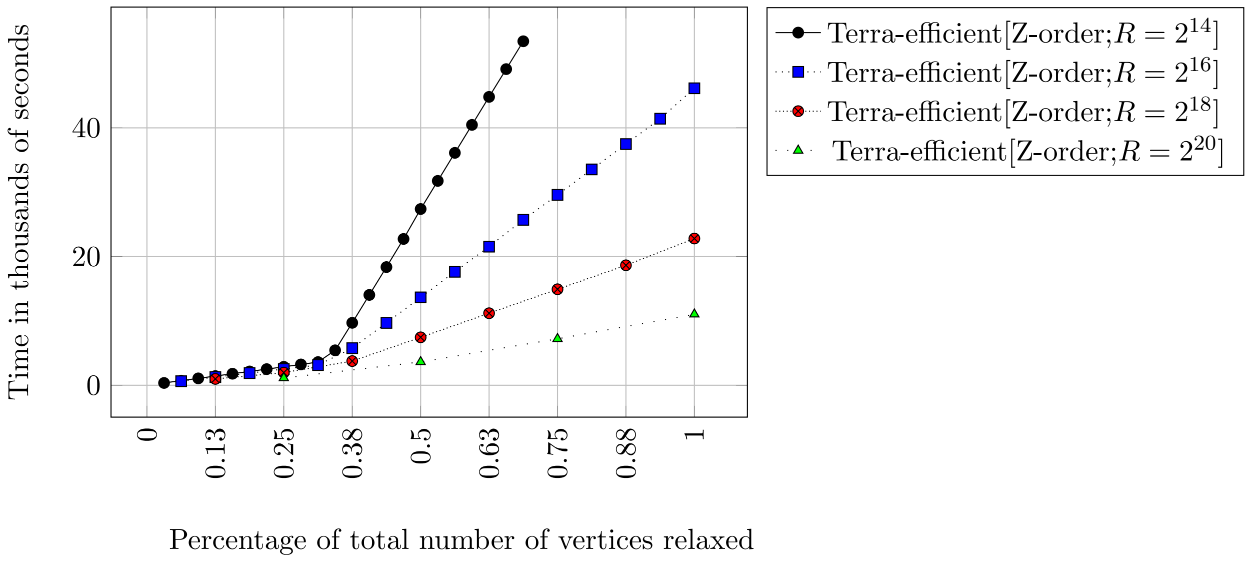


Figure 55: Time required to relax a percentage of the total number of vertices which have to be relaxed for Step 3 (Inter-Tile Dijkstra) the Terra-efficient algorithm in a 16384×16384 grid, with only 2048 MB of memory available. Note here that Terra-efficient[Z-order;$R = 2^{14}$] only relaxes 70% of all its vertices and thus not finish within the required amount of time.

Region size (R)	16384	65536	262144	1048576
Terra-eff[Z-order,Step 1]	27497	28059	29416	32236
Terra-eff[Z-order,Step 3]	∞	46150	22801	11030
Terra-eff[Z-order,Step 4]	-	916	1047	1333
Terra-eff[Z-order,Total]	∞	75125	53264	44599

Figure 56: Running time results for the worst case data set in a grid with 16384 $\times$ 16384 vertices for the Terra-efficient algorithm when R is varied, with 2048 MB memory available.

5.2.4 Worst case with 10% distortion

In this section we will present the results for the worst case data set with 10% distortion. Figure 57 shows the best results for each of the algorithms given different values of n.

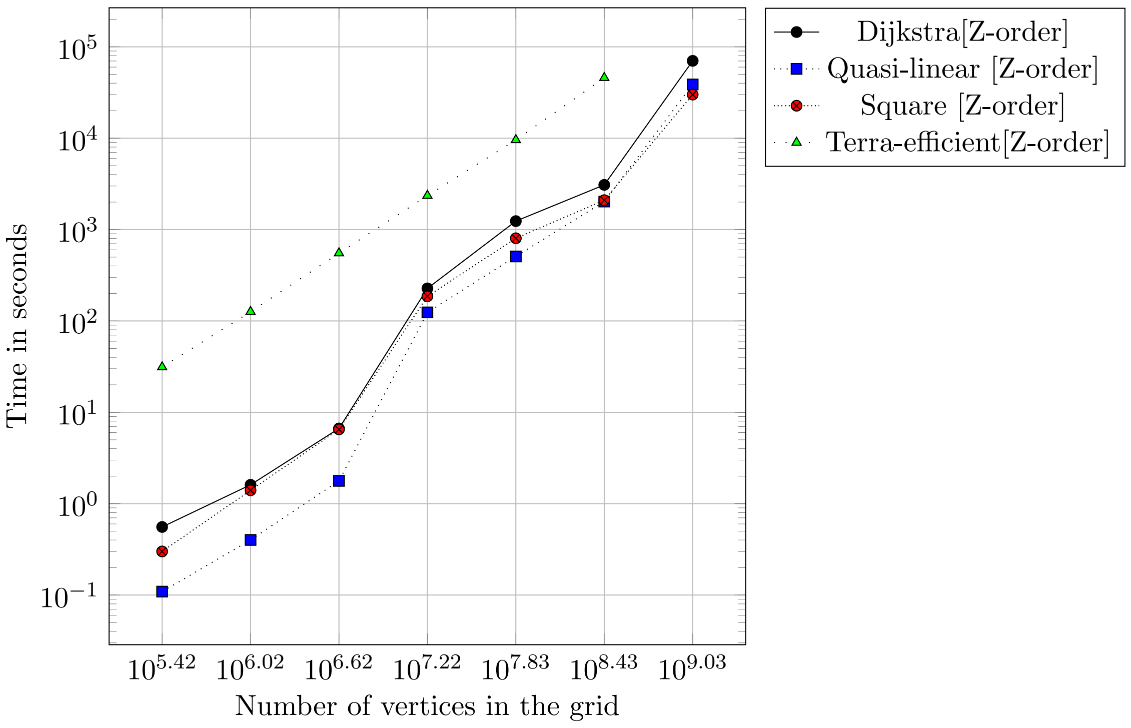
Dijkstra's algorithm The first thing we notice when looking at figures 48 and 57 is that Dijkstra's algorithm does not fail and is just slightly worse then Quasi-linear and Square , while it is better then the Terra-efficient algorithm.

Quasi-linear and Square We see figure 57 that Quasi-linear likes to have region size of 2^{12} and prefers α to be large. In contrary to what we see for the worst case data set in figure 48, where a smaller α is preferred. The Square algorithm prefers a region size of $R = 2^{10}$ which is little bit less then Quasi-linear , whereby the reasoning behind this is the same as that in section 5.2.3. It is also interesting to note that Quasi-linear performs a little bit worse then Square for the largest grid of 32768 $\times$ 32768 nodes. What we also notice is the for the worst case data set and the worst case data with 10% distortion the same region sizes are preferred.

Figure 58 clearly shows that a larger α is preferred for the Quasi-linear algorithm and that $R = 10^{5.42}$ performs best for a grid of $n = 2^{26}$ vertices.

Figure 59 shows that the trend for the different R for the Quasi-linear and the Square algorithm is the about the same it is just shifted somewhat in the X-axis direction.

Terra-efficient Figure 60 shows that Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm still takes up most time and that Step 3 (Inter-Tile Dijkstra) explodes far less then for the worst case data set without distortion as seen in section 5.2.3, but it still seems to be increasing for larger grids. In figure 61 we see that the region size has little effect on the total running time and on the running time of Step 1 (Intra-Tile Dijkstra) . Step 4 (Final Dijkstra) and mainly Step 3 (Inter-Tile Dijkstra) do seem to be affected significantly.



Number of vertices (n)	$10^{5.42}$	$10^{6.02}$	$10^{6.62}$	$10^{7.22}$	$10^{7.83}$	$10^{8.43}$	$10^{9.03}$
Quasi-linear [Z-order](R, α)	$(2^6, R)$	$(2^6, R)$	$(2^{12}, \frac{R}{2})$	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{12}, R)$	$(2^{12}, R)$
Square [Z-order](R)	2^{16}	2^{18}	2^{12}	2^{12}	2^{12}	2^{10}	2^{10}
Terra-eff[Z-order](R)	2^6	2^{16}	2^{16}	2^{16}	2^{16}	2^{16}	

Figure 57: Best running time results for worst case data set with 10% distortion with 384 MB available memory.

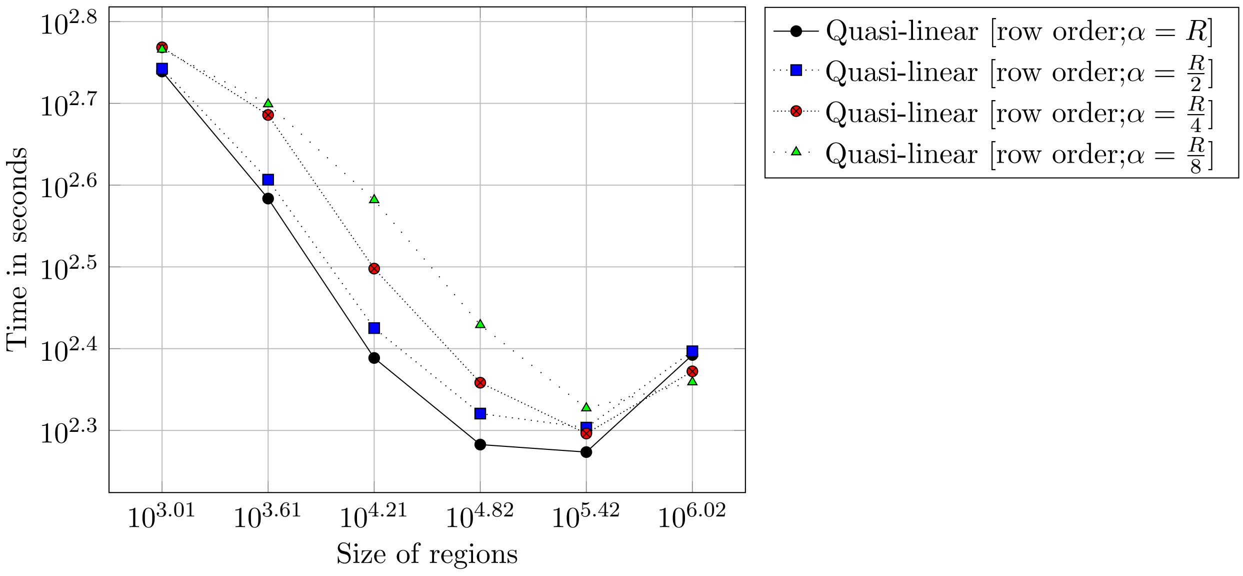


Figure 58: Running times results for the worst case data set with 10% distortion in a 8192 × 8192 grid graph, with 384 MB memory available, where the region size and α values are varied for the Quasi-linear algorithm.

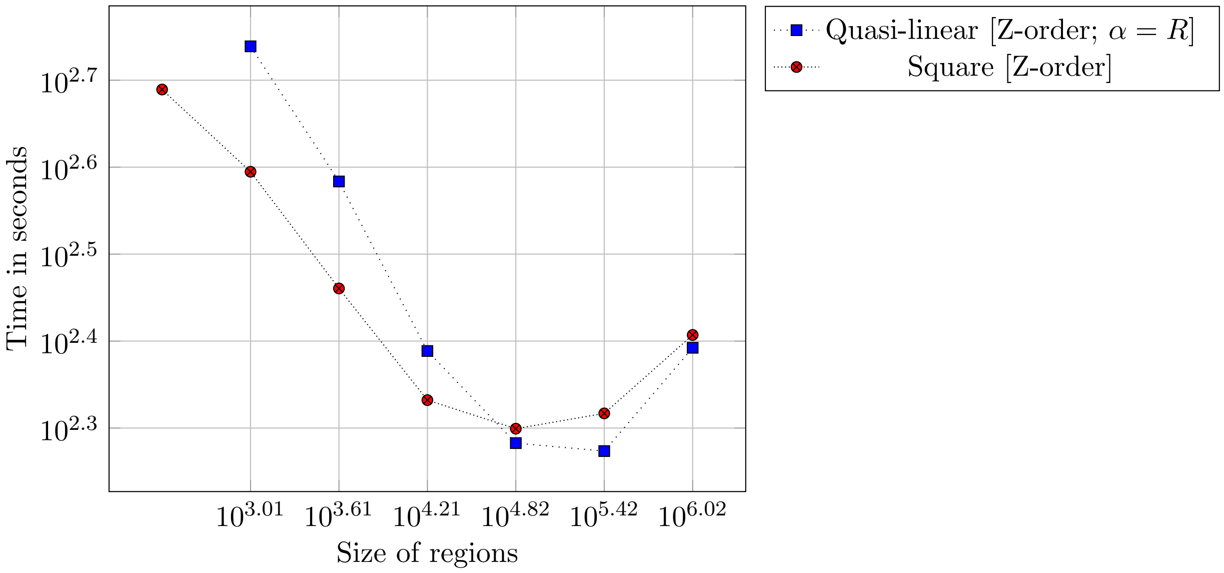


Figure 59: Running time results for the worst case data set with 10% distortion in a 8192 × 8192 grid graph, with 384 MB memory available, where the region size is varied.

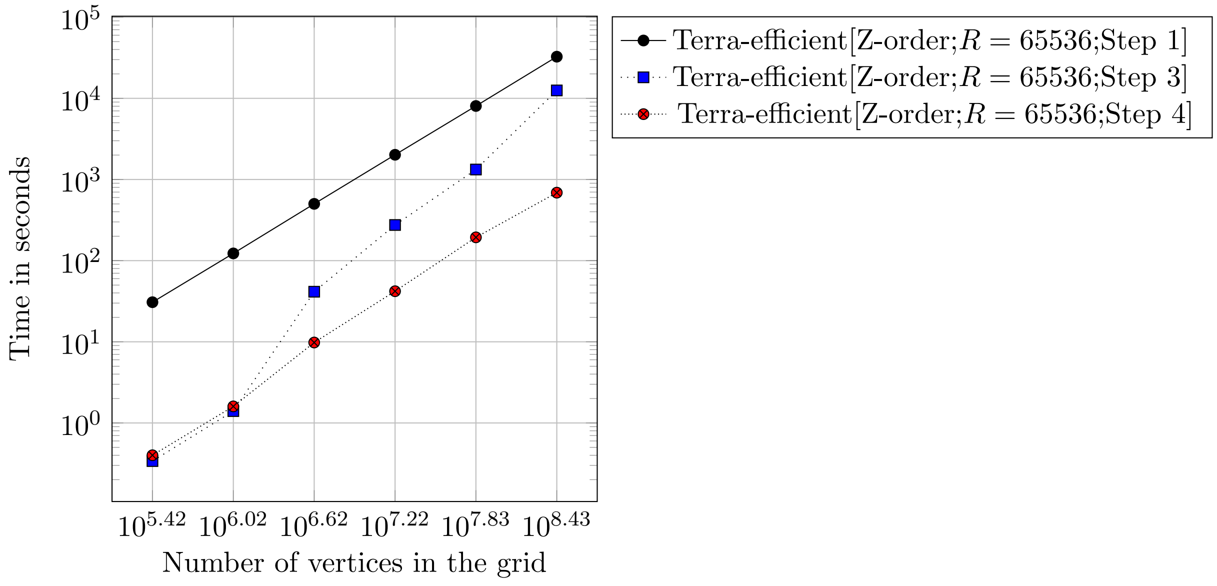


Figure 60: Running time results for a worst case data set with 10% distortion, with only 384 MB of memory available for the different steps of the Terra-efficient algorithm.

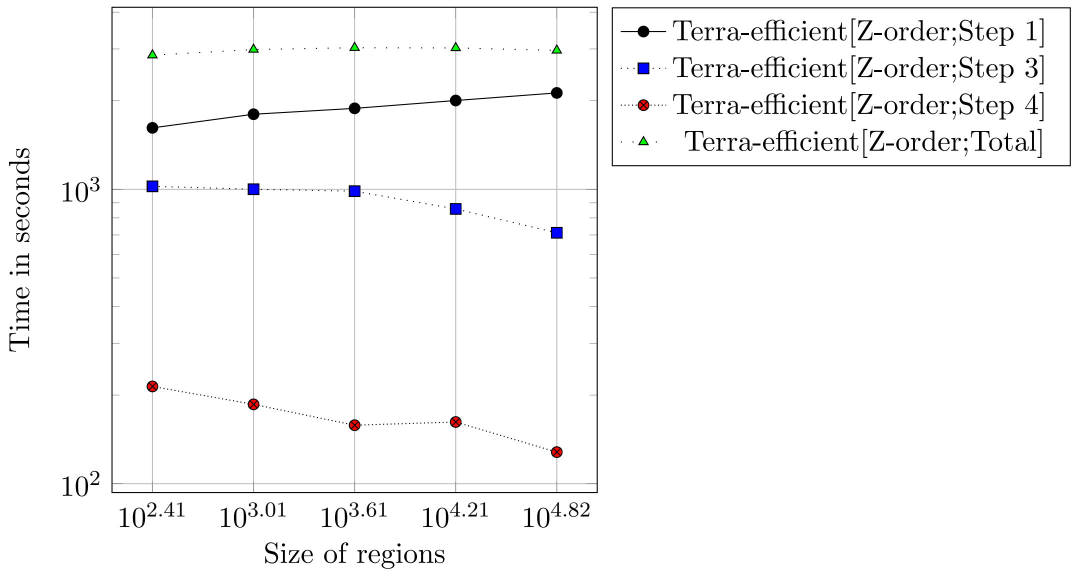


Figure 61: Running time results for a worst case data set with 10% distortion in a 4096×4096 grid graph, with only 384 MB of memory available, where the region sizes are varied for the Terra-efficient algorithm.

5.2.5 Fully random

In this section we will present the results for the fully random data set for which the best results are given in figure 62.

Dijkstra As seen in figures 48, 57 and 62 we notice that Dijkstra's algorithm performs similarly for a random graph as it does for a worst case graph with 10% distortion. This is not as bad as one might initially have thought it would be even for $n = 10^{9.23}$, which means that the grid is about 20 Gigabytes on disk. This is a lot more then the 384 Megabytes of memory which are available.

Quasi-linear and Square In figure 62 we see that the Quasi-linear and Square algorithms performs similarly for grids with up to 32768×32768 vertices.

In figure 63 we see that for the Quasi-linear algorithm large α are preferred and $R = 10^{5.42}$ works best for this grid size. This is in line with what we see in worst case data set with 10% distortion as presented in figure 58.

Figure 64 shows that Quasi-linear and Square perform about as well as the other.

Terra-efficient Figure 65 shows that Step 1 (Intra-Tile Dijkstra) takes the most time and that Step 3 (Inter-Tile Dijkstra) requires some amount of I/Os for $n \geq 10^{6.62}$, but after this point increases linearly with n until $10^{7.83} \leq n$ and for $n = 10^{8.43}$ it increases further again. The reason why this further increase in Step 3 (Inter-Tile Dijkstra) is made might be explained by looking at figure 77 where we can see that the total number of regions which are represented by vertices in the priority queue is 133, when now calculating how much memory would be needed to keep these 133 regions and their related data structures in memory then we get:

$$133 \cdot (4 \cdot 256) \cdot (4 \cdot 256) \approx 133 \cdot 2^{20}$$

Where $(4 \cdot 256)$ is approximately the number of vertices around the boundary of one region. The second $(4 \cdot 256)$ represents the number of outgoing edges from any vertex along the boundary to any other vertex along the boundary in that same region. If we now assume each edge weight has the size of one float then we get:

$$133 \cdot 2^{20} \cdot 4 = 532 \cdot 2^{20} > 384 \cdot 2^{20}$$

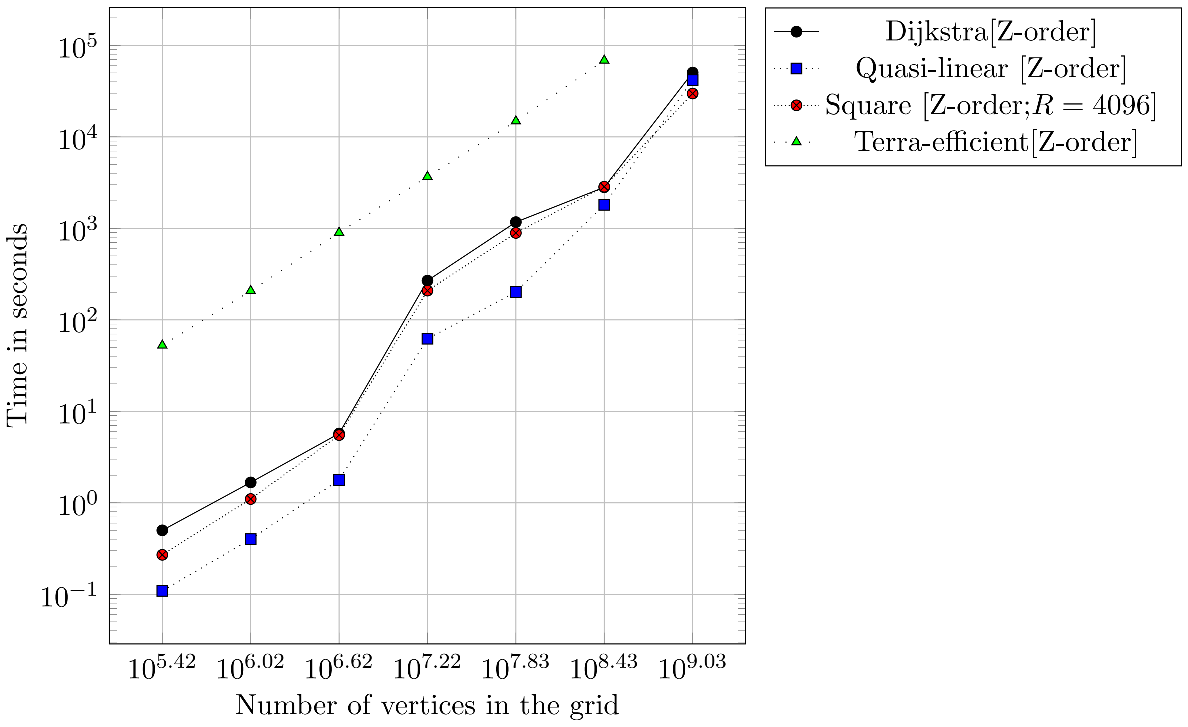
Which means there is a good chance that everything won't fit in memory (we can't know this for sure since it depends on the order in which the vertices are selected from the priority queue). If we on the other hand take the grid of size $n = 10^{7.83}$, then we see a total of 65 different regions with vertices in the priority queue, which calculates to:

$$65 \cdot (4 \cdot 256) \cdot (4 \cdot 256) \approx 65 \cdot 2^{20}$$

Thus we need about $65 \cdot 2^{20} \cdot 4 \approx 260$ Megabytes to hold all the outgoing edges from vertices in the priority queue. Next to this we also need some amount of memory to hold the other information related to each region, but this is far less then the amount of memory needed to store the edge lists. Hence we can say that for $10^{7.83} = n$ we need less then 384 MB, thus we would expect to see something which is close to $O(\frac{n}{B})$.

Figure 66 shows the progression of the Terra-efficient algorithm for the different sub steps on a grid of size 4096 $\times$ 4096.

Besides this we notice that the Terra-efficient algorithm performs the same for worst case graphs with 10% distortion as it does for fully random graphs and also prefers large R over smaller ones. Even though the cost for choosing a smaller R for Step 3 (Inter-Tile Dijkstra) is far less steep then for the worst case data set.



Number of vertices (n)	$10^{5.42}$	$10^{6.02}$	$10^{6.62}$	$10^{7.22}$	$10^{7.83}$	$10^{8.43}$	$10^{9.03}$
Quasi-linear [Z-order](R, α)	$(2^8, R)$	$(2^8, R)$	$(2^{14}, R)$	$(2^{16}, \frac{R}{2})$	$(2^{18}, R)$	$(2^{10}, R)$	$(2^{10}, R)$
Square [Z-order](R)	2^{12}	2^{12}	2^{12}	2^{12}	2^{12}	2^{12}	2^{12}
Terra-eff[Z-order](R)	2^{6}	2^{6}	2^{6}	2^{8}	2^{16}	2^{16}	2^{16}

Figure 62: Best running time results for the fully random data set with 384 MB memory available.

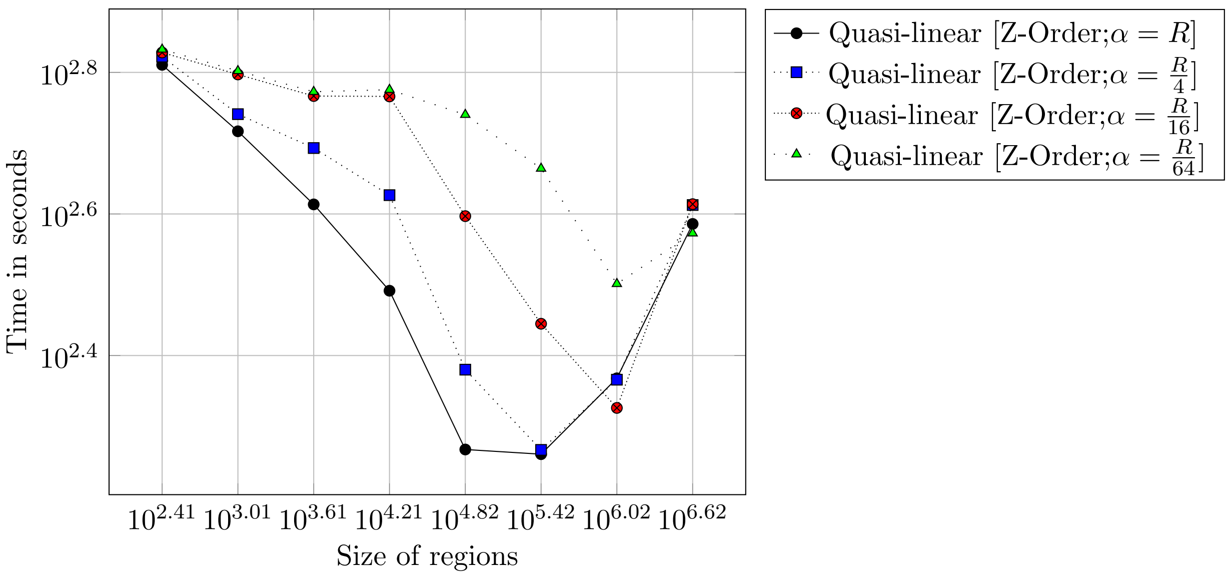


Figure 63: Running time results for the fully random data set in a 8192×8192 grid graph, with only 384 MB memory available, where the region size and α values are varied for the Quasi-linear algorithm.

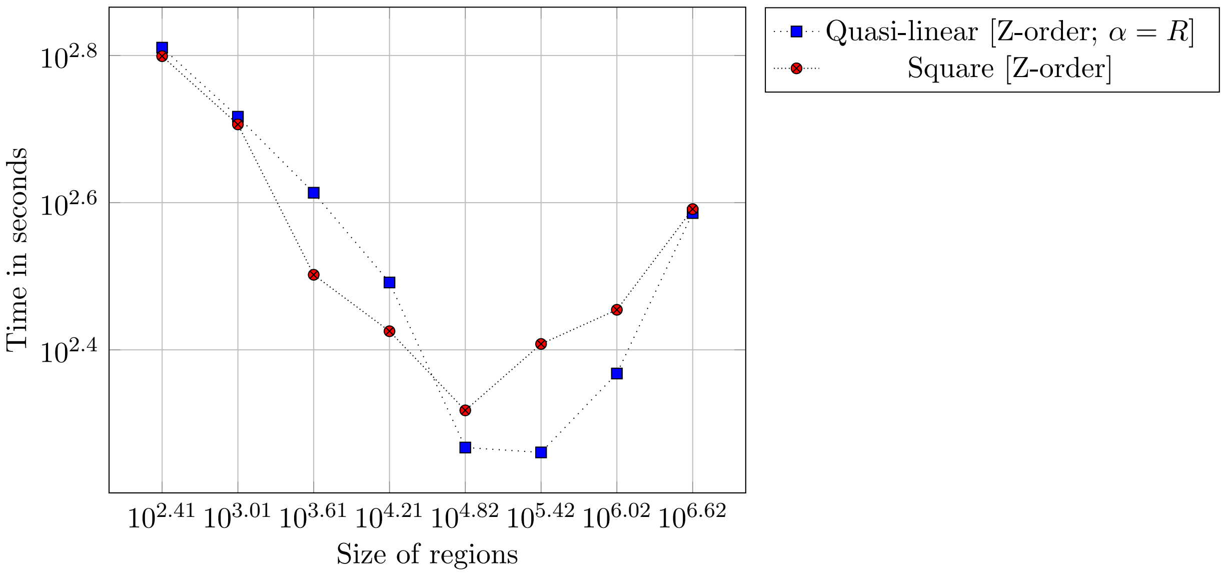


Figure 64: Running time results for the fully random data set in a 8192×8192 grid graph, with only 384 MB memory available, where the region size is varied.

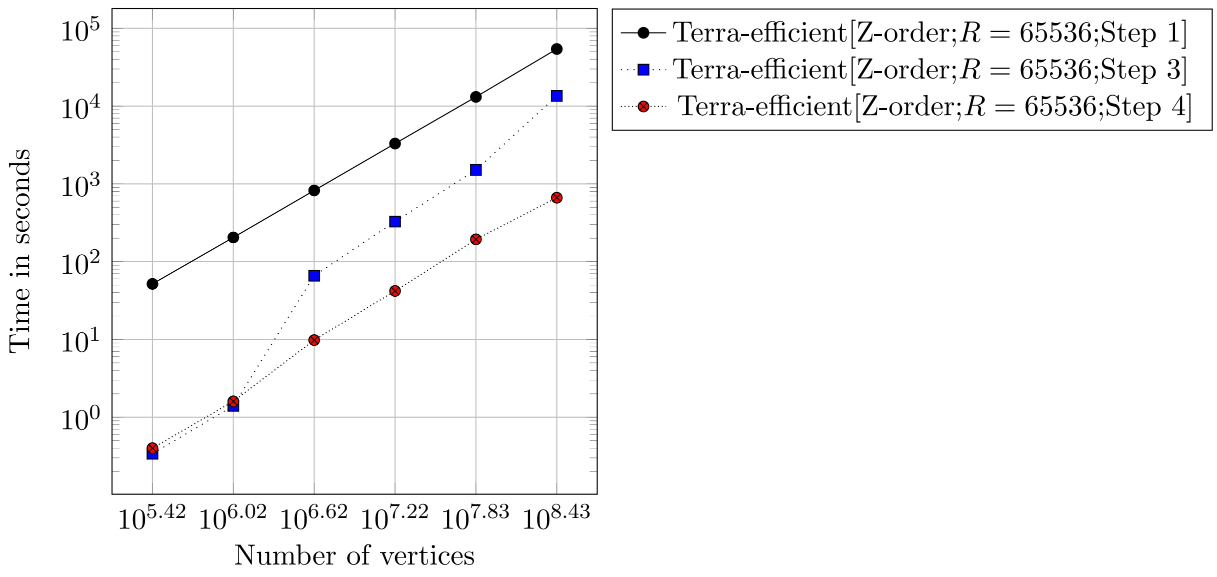


Figure 65: Running time results for the fully random data set, with only 384 MB memory available for the Terra-efficient algorithm.

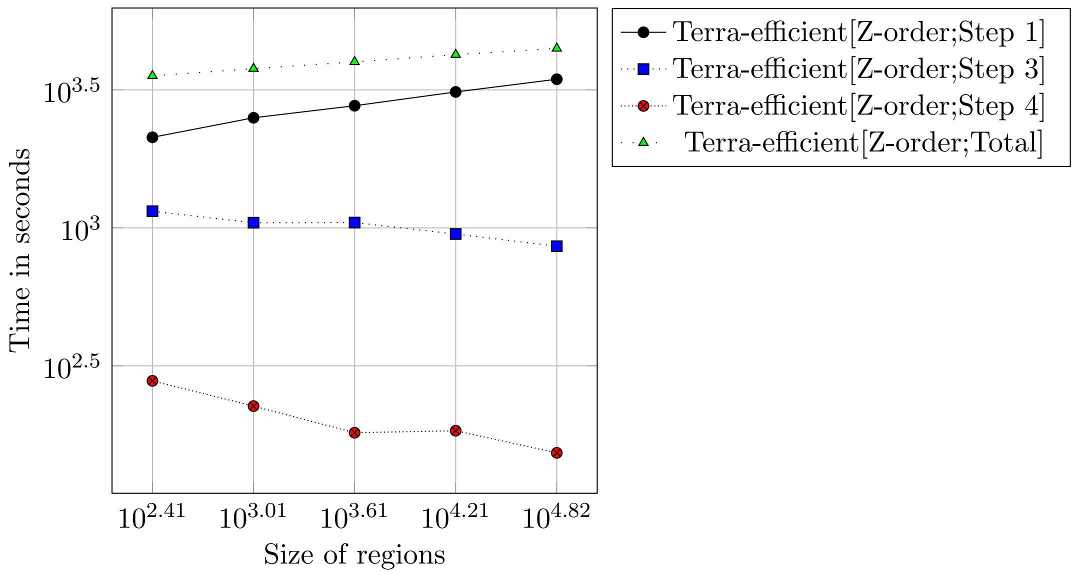


Figure 66: Running time results for the fully random data set in a grid of 4096×4096 vertices, with only 384 MB memory available for the Terra-efficient algorithm, where the running times of the different sub steps of the algorithm are given.

5.2.6 Highways and obstacles

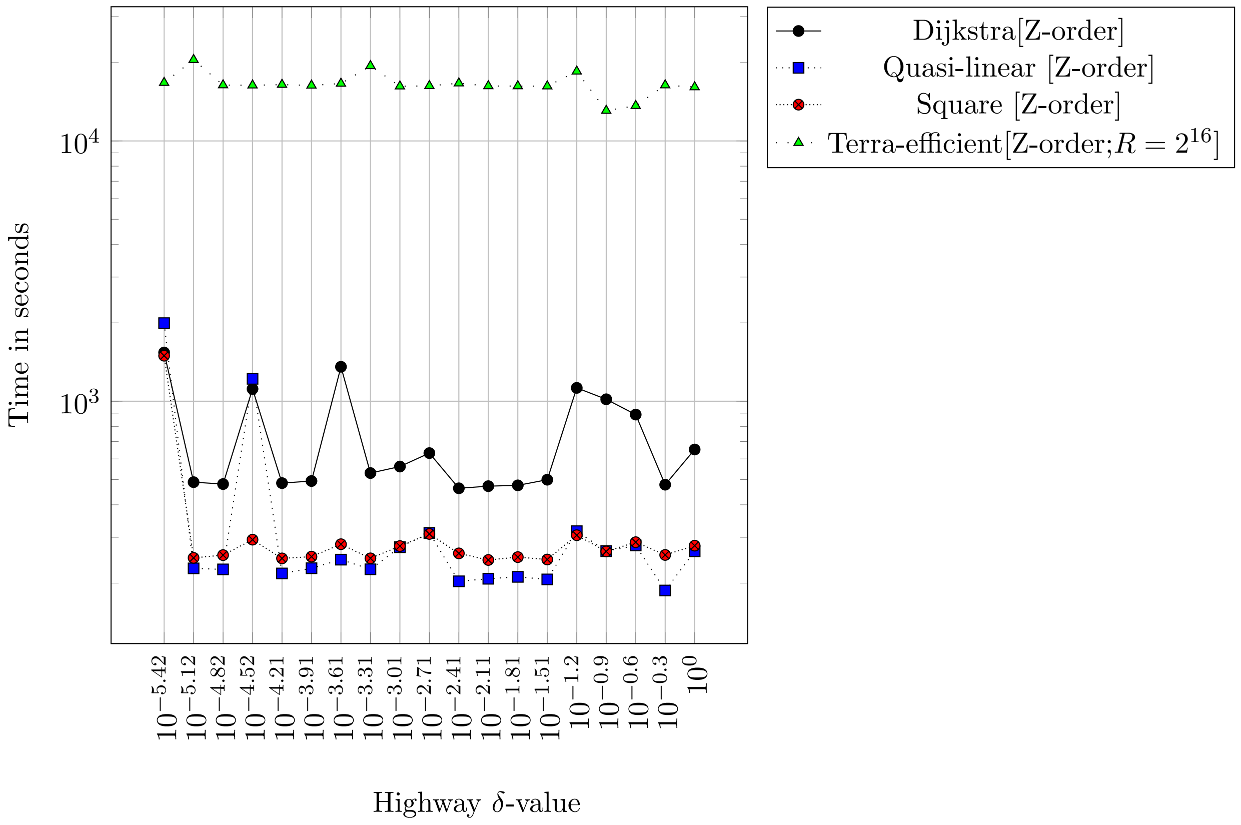
In this section we present the results for the highways and obstacles data set, where figure 67 visualizes the best results for each algorithm. What we mainly see here is that Quasi-linear and Square perform similarly, while Dijkstra performs worse, but it seems to be purely a constant factor. The Terra-efficient performs worst.

Figure 68 and 69 show the different running times for different δ and different R for the Square and Quasi-linear algorithms. Both figures show the same trends and only $\delta = 10^{-4.52}$ is somewhat strange for the Square algorithm the reason for this is unknown.

Figure 70 shows that the total running time of the Terra-efficient algorithm is mainly dependent on the Step 1 (Intra-Tile Dijkstra) of the algorithm, but that Step 3 (Inter-Tile Dijkstra) is also relevant.

Figure 71 shows for $\delta = 1.0$ that for the Terra-efficient algorithm there is very little difference in running time between different values for R. This is mainly the case because when Step 1 (Intra-Tile Dijkstra) takes longer and R is larger Step 3 (Inter-Tile Dijkstra) will take less time.

In general when looking at these different figures we notice that larger δ require a bit more time to calculate, while small δ seem more random as in sometimes it takes more time and sometimes it takes less this might be related to the fact for small δ we are closer to the fully random graph and thus anything can happen.



Highway δ-value	$10^{-5.42}$	$10^{-5.12}$	$10^{-4.82}$	$10^{-4.52}$	$10^{-4.21}$	$10^{-3.91}$
Quasi-linear (R, α)	$(2^{14}, R)$	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{14}, R)$	$(2^{18}, R)$	$(2^{18}, R)$
Square (R)	(2^{16})	(2^{16})	(2^{16})	(2^{16})	(2^{16})	(2^{16})

Highway δ-value	$10^{-3.61}$	$10^{-3.31}$	$10^{-3.01}$	$10^{-2.71}$	$10^{-2.41}$	$10^{-2.11}$	$10^{-1.81}$
Quasi-linear (R, α)	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{16}, R)$	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{18}, R)$
Square (R)	(2^{16})	(2^{16})	(2^{16})	(2^{16})	(2^{16})	(2^{16})	(2^{16})

Highway δ-value	$10^{-1.51}$	$10^{-1.2}$	$10^{-0.9}$	$10^{-0.6}$	$10^{-0.3}$	10^0
Quasi-linear (R, α)	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{16}, R)$	$(2^{18}, R)$	$(2^{18}, R)$	$(2^{18}, R)$
Square (R)	(2^{16})	(2^{16})	(2^{14})	(2^{16})	(2^{16})	(2^{16})

Figure 67: Best running time results for the highways and obstacles data set in a grid with 8192×8192 vertices, with 384 MB memory available.

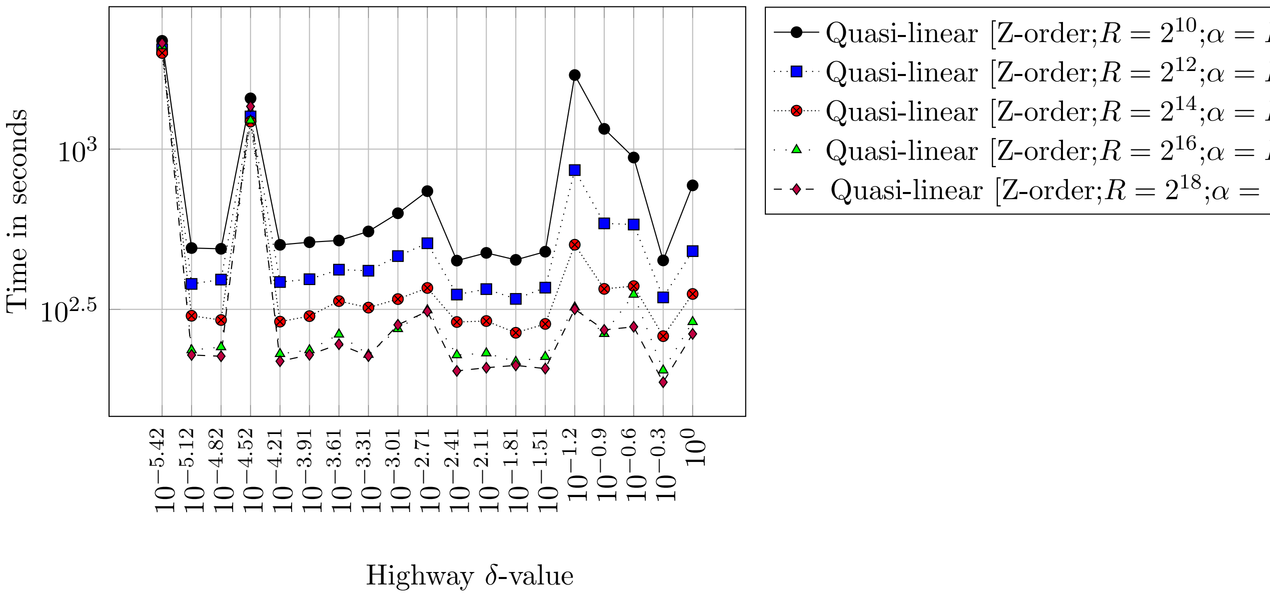


Figure 68: Best running time results for the highways and obstacles data set in a grid with 8192×8192 vertices for the Quasi-linear algorithm where R is varied, with 384 MB memory available.

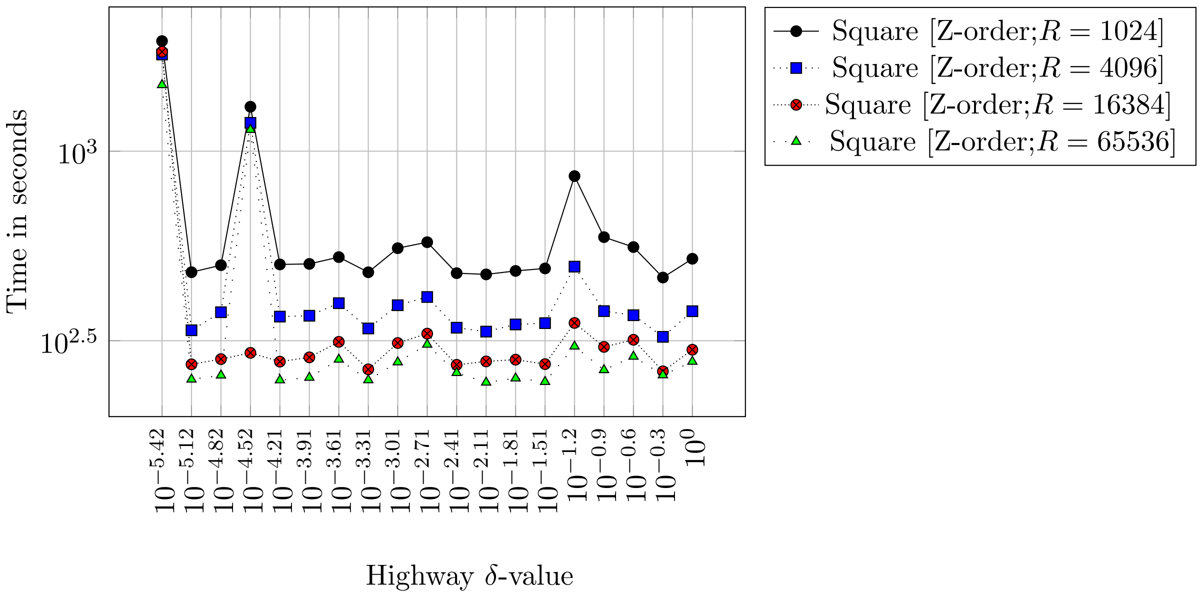


Figure 69: Best running time results for the highways and obstacles data set in a grid with 8192×8192 vertices for the Square algorithm where R is varied, with 384 MB memory available.

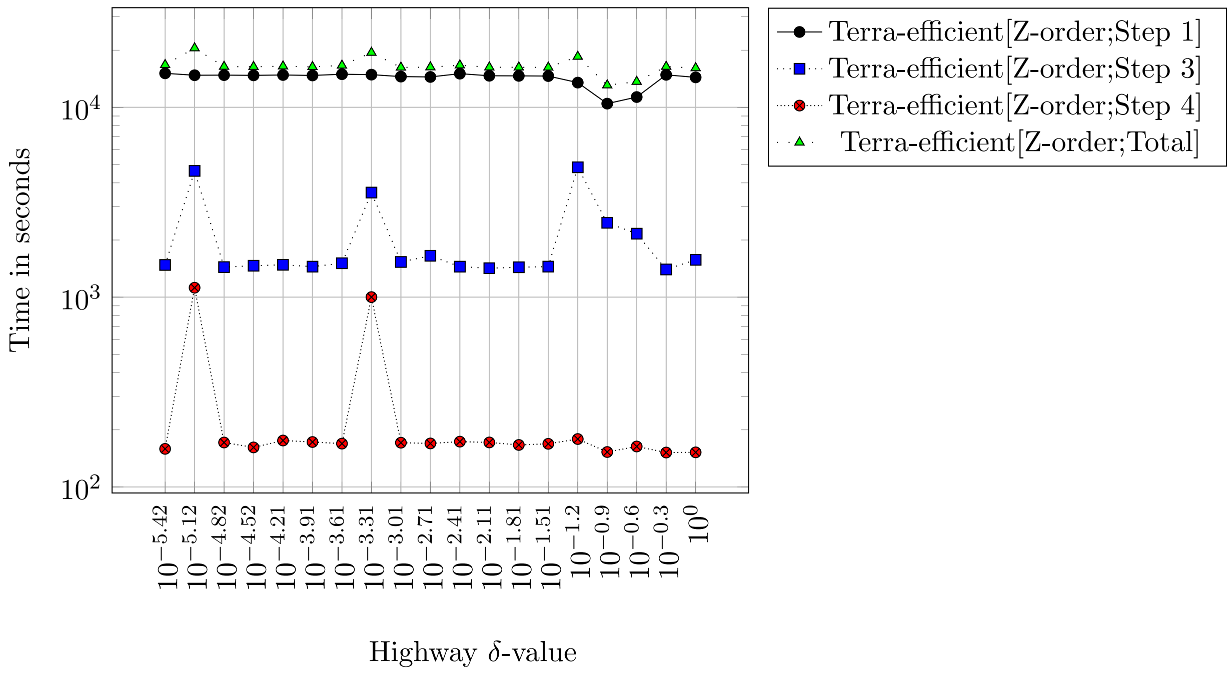


Figure 70: Best running time results for the highways and obstacles data set in a grid with 8192 × 8192 vertices for the Terra-efficient algorithm, with 384 MB memory available.

Region size (R)	4096	16384	65536
Terra-eff[Z-order,Step 1]	11618	12875	14380
Terra-eff[Z-order,Step 3]	3625	2313	1572
Terra-eff[Z-order,Step 4]	211	169	152
Terra-eff[Z-order,Total]	15455	15354	16104

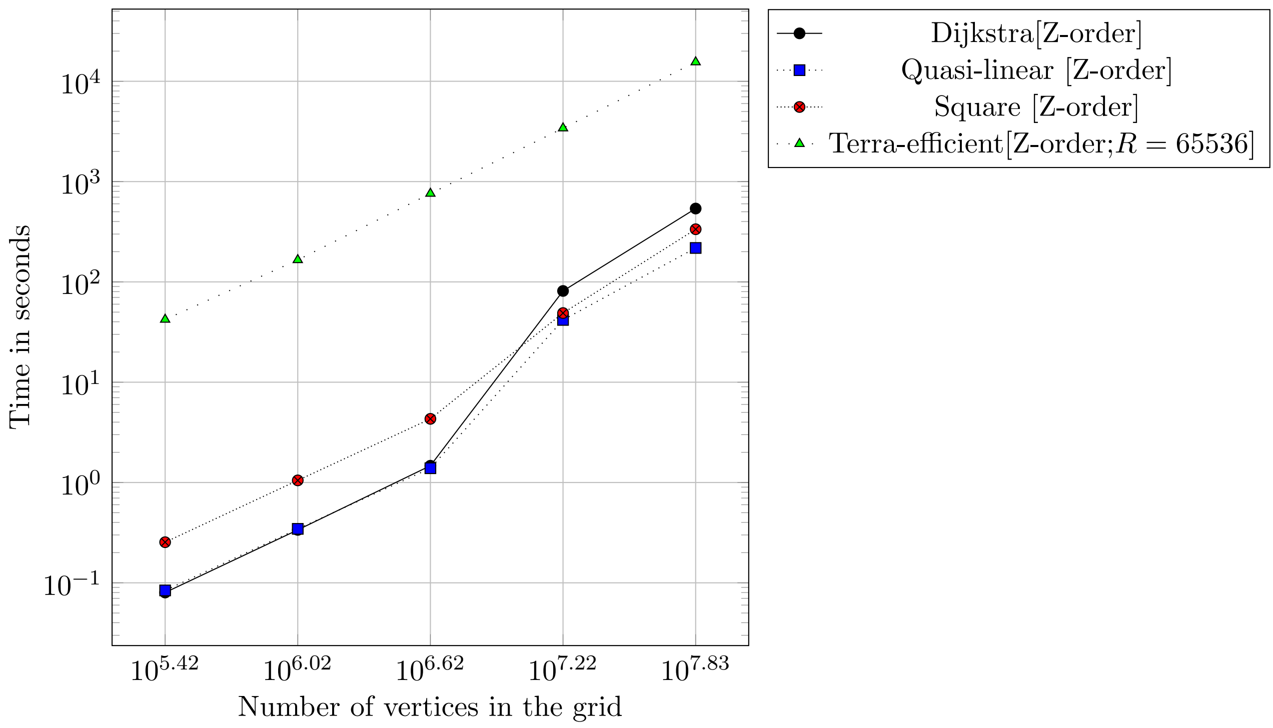
Figure 71: Running time results for the highways and obstacles data set in a grid with 8192 × 8192 vertices for the Terra-efficient algorithm when R is varied and $\delta = 1.0$, with 384 MB memory available.

5.2.7 Real world data sets

In this section we present the results for the real world data set, where figure 72 visualizes the best results for each algorithm. We clearly notice here that Dijkstra's algorithm performs about as well as Quasi-linear and Square , while Terra-efficient is worse.

Quasi-linear and Square From figure 73 we notice for the Quasi-linear algorithm that a large region size is preferred and $\alpha = R$ performs best, while in 74 for Square a much smaller R of 16384 is preferred. This is also as expected since Square always finishes 9 regions while Quasi-linear only does one. Notice here that $16384 \cdot 9 = 147456$ and $2^{17} < 147456 < 2^{18}$.

Terra-efficient In figure 75 we see that the cost in Step 1 (Intra-Tile Dijkstra) for a larger region size is small, while the benefit is greater in Step 3 (Inter-Tile Dijkstra) of the algorithm. One must note here that since the figure has logarithmic scale the benefit in Step 3 (Inter-Tile Dijkstra) might look larger then the cost in Step 1 (Intra-Tile Dijkstra) , but in essence its about the same as we can see by looking at the line which represents the total cost.



n	$10^{5.42}$	$10^{6.02}$	$10^{6.62}$	$10^{7.22}$	$10^{7.82}$
Quasi-linear (R, α)	(4096,$\alpha = R$)	(4096,$\alpha = R$)	(4096,$\alpha = R$)	(2^{18},$\alpha = R$)	(2^{18},$\alpha = R$)
Square (R)	16384	65536	256	262144	16384
Terra(R)	4096	4096	4096	4096	4096

Figure 72: Best running time results for the real world data set, with 384 MB memory available.

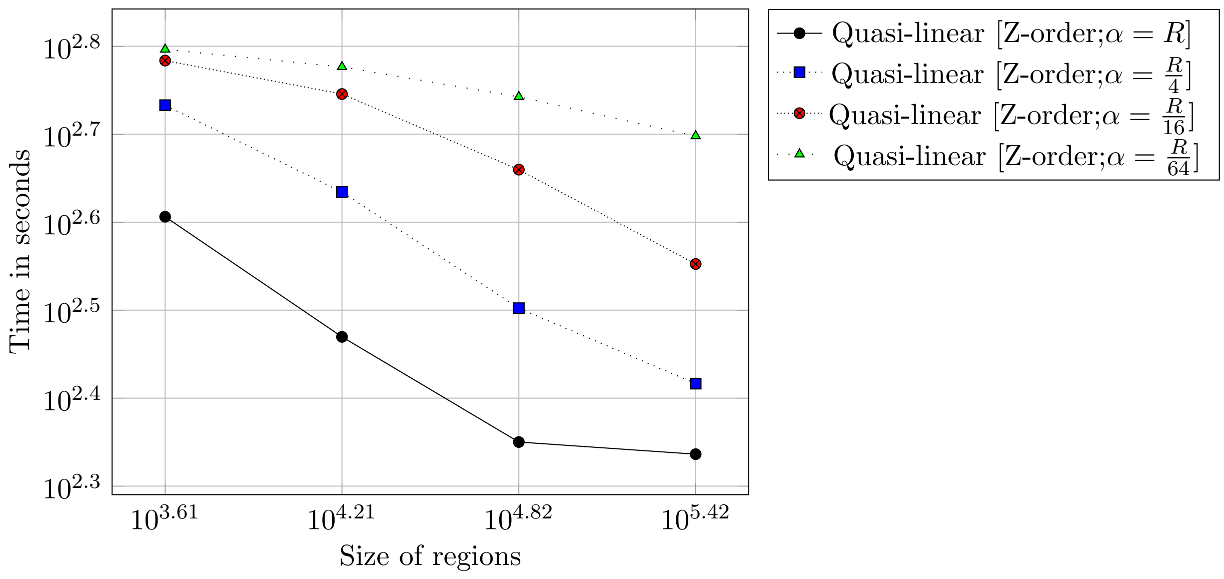


Figure 73: Best running time results for the real world data set in a grid with 8192 × 8192 vertices for the Quasi-linear algorithm where R is varied, with 384 MB memory available.

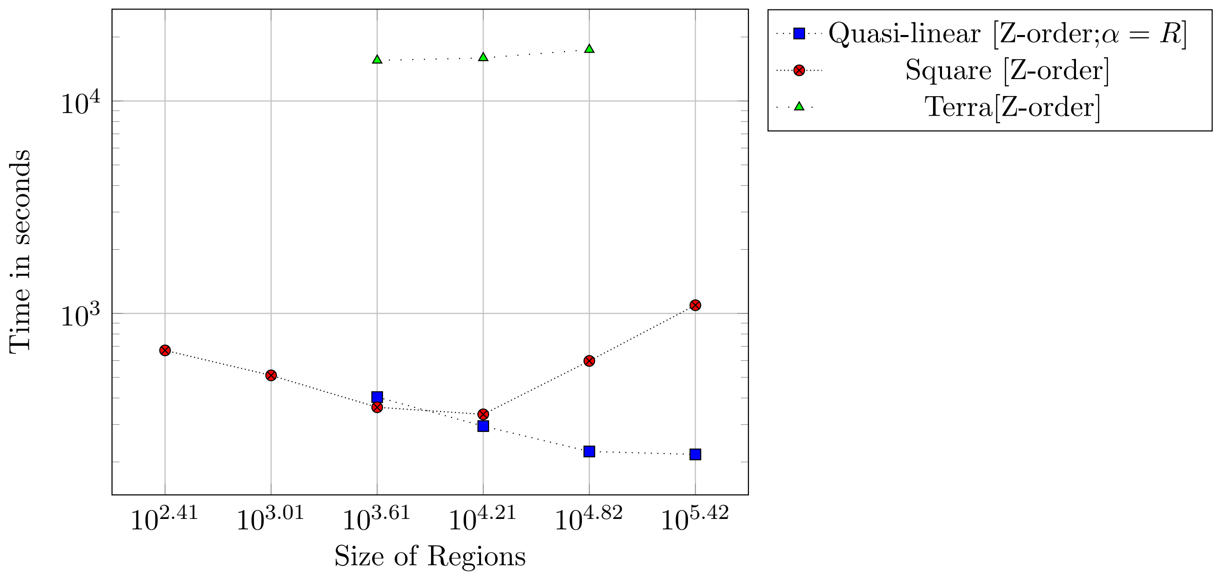


Figure 74: Best running time results for the real world data set in a grid with 8192 × 8192 vertices where R is varied, with 384 MB memory available.

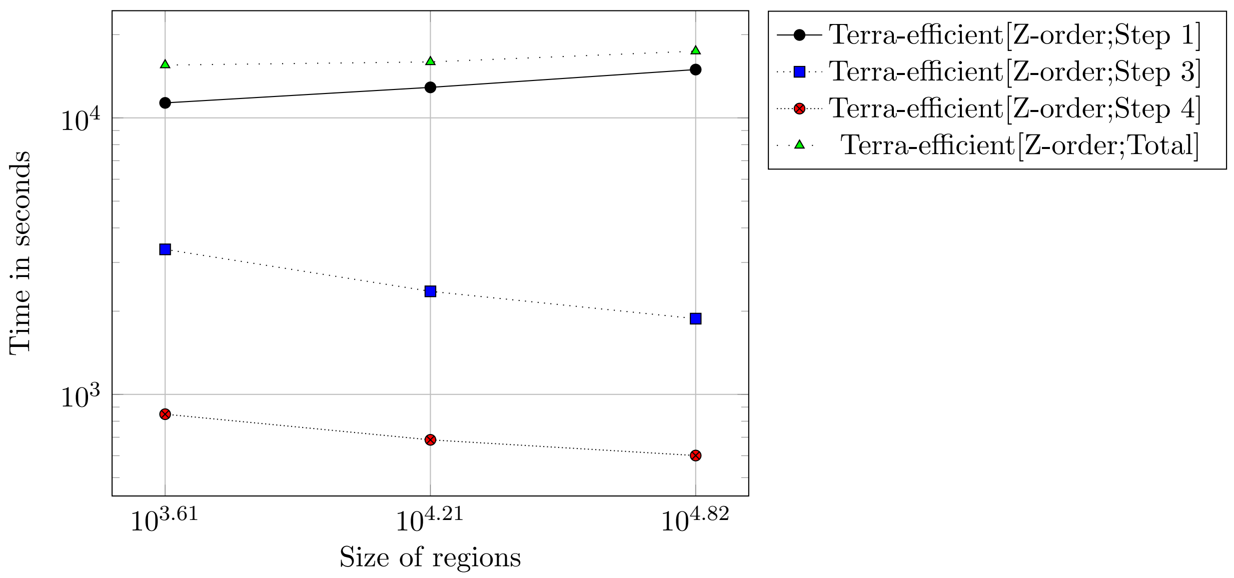


Figure 75: Best running time results for the real world data set in a grid with 8192 × 8192 vertices for the Terra-efficient algorithm, with 384 MB memory available.

5.2.8 Combined graphs and general thoughts

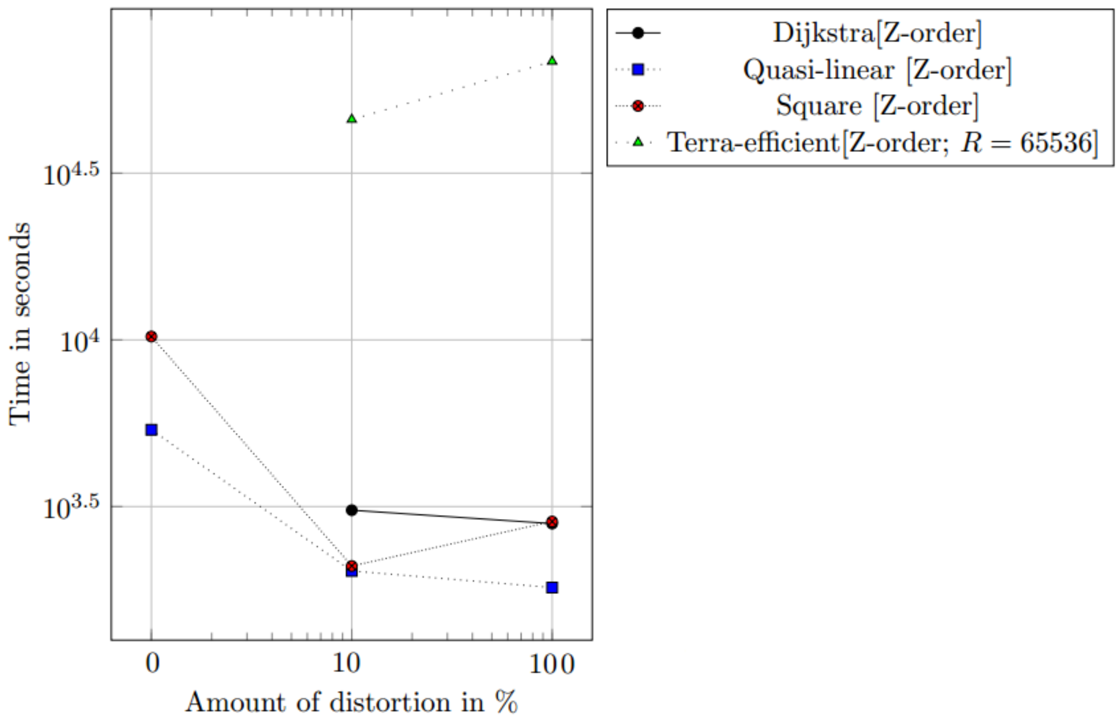
In this section we will present the differences in results between the worst case, worst case with 10% distortion and fully random data sets. Figure 76 visualizes these results and what we notice is that Dijkstra's and Terra-efficient algorithms don't finish within time for the worst case data structure. Besides this we see that Terra-efficient is always worse then all the other algorithms and that Dijkstra's algorithm is comparable with Quasi-linear and Square for the worst case with 10% distortion and fully random data sets.

The reason why the Square algorithm performs better for the worst case data set with 10% distortion then for the fully random data set is hard to properly explain, but it might have something to do with the fact that the worst case data set with 10% distortion still has some structure in side it, while the fully random data set does not. With structure we refer to the vertices with zero edge weights between them.

Figure 77 shows the number of vertices from different regions which are in the priority queue for the Terra-efficient algorithm.

Highways and obstacles For the highways and obstacles data sets we notice that Quasi-linear performs best with large $R = 2^{18}$ and $\alpha = R$ and for Square we see the same behavior since $R = 2^{16}$ is best. For the Terra-efficient algorithm we notice that Step 1 (Intra-Tile Dijkstra) still takes up most time and the δ value hardly effects the running time of Step 1 (Intra-Tile Dijkstra) of the algorithm. Moreover the chosen δ does effect Step 3 (Inter-Tile Dijkstra) and Step 4 (Final Dijkstra) of the Terra-efficient algorithm.

Real world For the real world data sets we notice that Dijkstra's algorithm performs about as well as Quasi-linear and Square , while Terra-efficient performs worst. For Quasi-linear we see that a larger R of 2^{18} and $R = \alpha$ work well. While Square we see that a region size of $R = 2^{14}$ works well. Just as with the highways and obstacles data set we notice that the Terra-efficient algorithm uses up most of its time in Step 1 (Intra-Tile Dijkstra) of the algorithm and we notice that a larger R is improves the time required for Step 3 (Inter-Tile Dijkstra) and Step 4 (Final Dijkstra) , while it slightly increases time running time of Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm.



	Worst case	Worst case With 10% distortion	Fully random
Amount of distortion	0%	10%	100%
Quasi-linear [Z-order](R, α)	$(16384, \frac{R}{8})$	$(4096, R)$	$(4096, R)$
Square [Z-order](R)	1024	1024	1024

Figure 76: Best running time results for the worst case data set, with 0%, 10% and 100% distortion (fully random) in a grid of size 16384 × 16384 grid, with only 384 MB of memory available. Dijkstra's and Terra-efficient running time for the worst case data set is ∞ (more then three days).

	Worst case	Worst case With 10% distortion	Fully random
Amount of distortion	0%	10%	100%
Terra[Z-order;$R = 65536$]($n = 10^{5.42}$)	4	3	4
Terra[Z-order;$R = 65536$]($n = 10^{6.02}$)	16	8	7
Terra[Z-order;$R = 65536$]($n = 10^{6.62}$)	64	12	18
Terra[Z-order;$R = 65536$]($n = 10^{7.23}$)	256	22	32
Terra[Z-order;$R = 65536$]($n = 10^{7.83}$)	1024	40	65
Terra[Z-order;$R = 65536$]($n = 10^{8.43}$)	4096	80	133

Figure 77: The maximum size of the priority queue in Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm, where the size is counted in different regions. Hence if two vertices from the same region are in the priority queue then it still only counts as one.

6 Conclusion

In this paper we have presented several known and a new algorithm to solve the single source shortest path problem within massive grid-based graphs.

We have proven that storing the grid-graph in Z-order instead of the standard row or column order pays off for Dijkstra's, Quasi-linear and Square algorithms. For Dijkstra's algorithm we prove that the I/O bound is not always $O(n)$, but if the number of vertices in the priority queue stays small, $O(\sqrt{n})$, and if the grid is stored in Z-order, then if $O(M) > O(\sqrt{n} \cdot \sqrt{B})$ then the I/O bound will be $O(\frac{n}{\sqrt{B}})$. For Quasi-linear and Square algorithms we prove that the I/O bound is $O(\frac{n}{\sqrt{B}})$, when the grid is stored in Z-order and $(R, \alpha) = (\sqrt{B}, \infty)$.

For the TerraCost algorithm we show that a slightly better I/O bound of $O(\frac{n}{B})$ can be achieved for grid-graphs. Besides this we have implemented a new version of the the Terracost algorithm, Terra-efficient , which has a CPU bound of $O(n \cdot \log R + n \cdot \log \frac{n}{R})$ this is a significant improvement to the prior CPU bound of $O(n \cdot \log R \cdot \sqrt{R} + n \cdot \log n)$. In our experiments we also see that for the worst case data set we need this new cpu bound to get the largest possible region in order to get the I/O bound of $O(\frac{n}{B})$.

We have theoretically proven that a constant highway dimension does not improve the I/O bound for any of our algorithms. For the augmented highway dimension we have shown that only for the Square algorithm an optimal I/O bound of $\Theta(\frac{n}{B})$ can be achieved, while Dijkstra's and Quasi-linear hold the same I/O bound as for worst case graphs.

6.1 In memory

In this section we will discuss the conclusion for the case where the grid and all data structures fit in memory we see little difference in running times between Dijkstra's, Quasi-linear and Square algorithms assuming that proper R and α are chosen. The general trend here is though that Quasi-linear performs best (with proper R, α) and that the running time for Dijkstra's algorithm grows faster then the Square algorithm for most data sets. The Terracost algorithm performs far worse, but this mainly since Step 1 (Intra-Tile Dijkstra) of the algorithm takes up most time. For the Terra-simple algorithm we know that the CPU bound is $O(n \cdot \log R \cdot \sqrt{R} + n \cdot \log \frac{n}{R})$, which explains it, while for the Terra-efficient algorithm it are the constant factors, which are to blame. To give an idea what i mean with these constant factors the Terra-efficient algorithm consists of about 6000 lines of programming code, while the Quasi-linear algorithm consists of less then 800 lines of programming code. Besides this maintaining a topology tree might be asymptotically optimal with a CPU bound of $O(R \log R)$, but in practice its worse. On the other hand if we compare Terra-efficient and Terra-simple for different values of R, then we notice that Terra-simple running doubles for every four times increase in R, while the Terra-efficient also requires more time when R gets larger, but this difference is closer to 10% whenever R gets four times larger (we refer to figures 22, 23, 27, 28, 32 and 33 for experimental results which support this).

6.2 I/O setting

In this section we will discuss the conclusion for the case where the grid does not fit in memory and the algorithm needs to use I/Os to solve the single source shortest path problem.

In general we notice that the Square and Quasi-linear algorithms both perform well for all data sets, where Quasi-linear is usually a bit better then Square .

Besides this Dijkstra's algorithm usually performs comparable to Quasi-linear and Square it just fails for the worst case data set. For the Terra-efficient algorithm we see that it is always slower then the other algorithms and that it also fails for the worst case data set. The reason why both Dijkstra and Terra-efficient fail for this data set is because the worst case data set after all the zero weight edges have been relaxed will always pick a random vertex in the graph to be relaxed.

For Quasi-linear we found that the best α is equal to R for all but the worst case data set, meaning that it performs best if the entire region is finished. The size of R is 16384 for the worst case data set and 4096 for the worst case with 10% distortion and the fully random data sets, while for the highway and obstacles data set an R of 2^{18} is usually preferred. The same trend we see for the Square algorithm where for the worst case, the worst case with 10% distortion and the fully random data sets a region size of 1024 is preferred, while for the highway and obstacles data set a much larger region size of 2^{16} is preferred.

Terra-efficient For the Terra-efficient algorithm we see very little difference between different values of R for all but the worst case data set, this is mainly since Step 1 (Intra-Tile Dijkstra) takes up most time and since if R is bigger then Step 1 (Intra-Tile Dijkstra) takes more time, while Step 3 (Inter-Tile Dijkstra) takes less time.

Moreover we have shown that the reason why Terra-efficient fails to work for the worst case data set is related to having to choose $R = 65536$. From the theoretical analysis of the TerraCost algorithm we know that $R = B^2$ gives the I/O bound of $\Theta(\frac{n}{B})$, but in practical setting we know that when $R = 65536$ then $B = 256$ can't be true. With an additional experiment we have shown that larger region sizes does significantly decrease the running time of Step 3 (Inter-Tile Dijkstra) of the Terra-efficient algorithm, while only slightly increasing the running time of Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm.

Thereby we must conclude that in order to run the TerraCost algorithm efficiently on a the worst case data a topology tree must be used. This conclusion extends to other data sets which have $\Theta(\frac{n}{\sqrt{R}})$ vertices in the priority queue and extract them in random order.

Within the paper of Hazel et al [8] it is concluded that Dijkstra's algorithm fails for real world data sets and even though our experiments are slightly different we must note that we notice nothing of the sort. The reason why it is expected for Dijkstra's algorithm to fail for the experiments in the paper of Hazel et al [8] is that these experiments are done on a graph with multiple sources. Let m be the number sources randomly distributed over a grid graph of sufficient size then we would expect that at least $\Omega(m \cdot \sqrt{n})$ vertices will be in the priority queue most of the time. Hence even if a random graph is used then our analysis of Dijkstra's algorithm would require a factor m more memory. If m is sufficiently large for a large graph then it is expected that Dijkstra's algorithm would always fail.

7 Future work

In general this document contains a large number of experimental results and perhaps not all implications of these experimental results are known at this point, hence this would be nice a starting point for a further in depth study. Moreover there are several questions which are still unanswered.

In memory For all algorithms it would be nice to see if the I/O setting and theoretical bounds could be applied to the memory and cache model. Moreover how does Z-order versus row order storage factor in?

I/O setting For Dijkstra's algorithm we wonder if we can show the breaking point in the experimental setting from graphs which have priority queues of size $\Theta(\sqrt{n})$.

For Quasi-linear and Square we wonder if Quasi-linear will stay about as good as Square or does Quasi-linear fail for even larger graphs as we somewhat see a glimpse of in the fully random graph of size 32786×32786. Moreover when does Square fail to work or is it always pretty good?

In addition to these experiments we wonder how Quasi-linear and Square perform in the case that multiple sources are added to a graph. For Quasi-linear and Square we would expect to still maintain the I/O bound of $\theta(\frac{n}{\sqrt{B}})$, but it would be nice to see some experimental results which support this expectation. For Dijkstra's algorithm as stated before we would expect it to fail for a sufficiently large number of sources, but it would still be nice to know when this happens and that it can be explained theoretically in combination with experimental results.

Moreover it would be nice to tailor a data set with constant augmented highway dimension, which fails for Dijkstra's, Quasi-linear and Terra-efficient with small region size, but still works for the Square algorithm. This would therefor show experimentally that in an I/O setting Square really adds stability over Quasi-linear .

For the TerraCost algorithm we wonder how well it will perform if the topology tree is further optimized, since as stated in the paper a region size of 65536 means approximately 80 Megabytes of memory is needed to complete Step 1 (Intra-Tile Dijkstra) of the Terra-efficient algorithm. 1280 bytes of memory per vertex seems somewhat unreasonable even if a complex data structure is used and also will such an optimized topology tree also significantly decrease the running time in the case where Terra-efficient runs fully in memory.

Appendices

A Notation

In this appendix some notation is defined which is used throughout the paper.

- M is the total available memory size.
- B is the size of one block on the hard drive.

- v is a vertex.
- $v_{i,j}$ is vertex with at location row i and column j within the grid.
- $d(v)$ is the current shortest distance from the source to vertex v.
- $r(v)$ is the region which contains v. Regions never overlap, hence only one region identifier is returned.
- n is the size of grid graph.
- $length(v, u)$ is the weight of the edge between v and u.
- R is the number of vertices contained in each region.
- $\Theta(\sqrt{R})$ is the number of boundary vertices for each region.
- α is the number of operations which are done in each sub step of the Quasi-linear algorithm.
- B_v is the block on disk which contains vertex v.
- R_v is the identifier for the region containing vertex v.

References

[1] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 782–793, Philadelphia, PA, USA, 2010.

[2] A. Aggarwal and J. S. Vitter. The I/O complexity of sorting and related problems. In *14th International Colloquium on Automata, Languages and Programming*, pages 467–478, London, UK, 1987. ACM vol 31 (1988).

[3] Lars Arge, Laura Toma, and Jeffrey Scott Vitter. I/O-efficient algorithms for problems on grid-based terrains. *J. Exp. Algorithmics*, 6, 2001.

[4] Lars Arge, Freek van Walderveen, and Norbert Zeh. Multiway simple cycle separators and i/o-efficient algorithms for planar graphs. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pages 901–918. SIAM, 2013.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[6] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, January 2012.

[7] Herman J. Haverkort. I/O-optimal algorithms on grid graphs. *CoRR*, abs/1211.2066, 2012.

[8] Tom Hazel, Laura Toma, Jan Vahrenhold, and Rajiv Wickremesinghe. Terracost: A versatile and scalable approach to computing least-cost-path surfaces for massive grid-based terrains.

[9] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3 – 23, 1997.

[10] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 146–155, Philadelphia, PA, USA, 2005.

[11] Anil Maheshwari and Norbert Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 372–381, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[12] Ulrich Meyer and Norbert Zeh. I/O-efficient shortest path algorithms for undirected graphs with random or bounded edge lengths. *ACM Trans. Algorithms*, 8(3):22:1–22:28, July 2012.

[13] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 376–382, New York, NY, USA, 1984. ACM.

[14] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. 1966.

[15] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, June 1983.

[16] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.

[17] Daniel Dominic Sleator and Robert E. Tarjan. Amortized efficiency of list update rules. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 488–492, New York, NY, USA, 1984.