FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Implementing and Investigating the Performance of an Approximate Distance Oracle for Planar Undirected Graphs

Bachelor's Thesis
June 2016

Author: Lukas S. Engedal
Department of Computer Science
University of Copenhagen

Supervisor: Mikkel Thorup
Department of Computer Science
University of Copenhagen

Abstract

We implement a stretch-$(1+\epsilon)$ approximate distance oracle for undirected, planar graphs based on the article [Tho04]. Our implementation can preprocess a graph with n vertices in $O(n(\log n)^3/\epsilon)$ time, creating a distance oracle structure using $O(n(\log n)^2/\epsilon)$ space that can estimate distances in $O((\log n)^2/\epsilon)$ time and within a factor of $(1+\epsilon)$.

We test the performance of our implementation on a set of graphs with one to five thousand vertices, by measuring run times, memory usage and oracle distance accuracy. The results of our tests of run times and memory usage are in accordance with our expectations, while the result of the accuracy test it not, which indicates there might be an error in either our implementation of the oracle or in the implementation of our performance tests.

Resumé

Vi implementerer et spænd-$(1+\epsilon)$ approksimativt afstands-orakel for ikke-orienterede, plane grafer baseret på artiklen [Tho04]. Vores implementation kan forarbejde en graf med n knuder på $O(n(\log n)^3/\epsilon)$ tid, hvilket producerer en afstands-orakel struktur der tager $O(n(\log n)^2/\epsilon)$ plads og kan estimere afstande inden for en faktor $(1+\epsilon)$ på $O((\log n)^2/\epsilon)$ tid.

Vi tester performance af vores implementation på et sæt af grafer med et til fem tusinde knuder, ved at måle køretider, pladsforbrug og afstands-præcision. Resultaterne af vores tests af køretider og pladsforbrug er i over-ensstemmelse med hvad vi forventer, hvorimod resultatet af vores test af afstands-præcision ikke er det, hvilket indikerer at der muligvis er en fejl enten i vores implementation af oraklet eller i vores implementation af performance-testene.

Contents

1 Introduction **1**

2 Helpful Algorithms and Data Types **2**

2.1 Data Types 2

2.2 Algorithms 5

3 Constructing the Distance Oracle **6**

3.1 The Planar Separator Theorem 8

3.2 Constructing the Distance Oracle 12

3.3 Preprocessing 14

3.4 Answering Queries 15

4 Performance Testing **17**

4.1 Graphs 18

4.2 Preprocessing 19

4.3 Querying 22

5 Conclusion **24**

5.1 Future Work 24

References **25**

1 Introduction

Having to determine the shortest path from point A to point B is a problem that one encounters under many different circumstances. You might be trying to traverse the surface of a plane such as Earth, to write pathfinding for an AI, or to determine routing for a set of routers on the Internet. All different situations with different requirements and expectations to a solution to the path finding problem.

One such class of solutions are known as *distance oracles*. They take a graph, consisting of edges and vertices, and preprocess it to produce a distance oracle structure, which they can then use to determine paths between pairs of vertices. One way to construct such a distance oracle is described in [Tho04], and it is this approach that we will base our implementation on.

We will start with a brief walk-through of the particular concepts from graph theory that we will need for our implementation. The section is not meant to be an introduction to graph theory for the uninitiated, as we expect our readers to already be familiar with the concepts described, but rather as a way of making sure that the reader knows exactly what we are referring to when speaking about trees, graphs, Dijkstra's, etc., and additionally to introduce the notation we will use for these. For a general introduction to data types and algorithms we recommend CLRS's *Introduction to Algorithms*[CLRS09].

We will then describe in detail the construction of a stretch-$(1+\epsilon)$ approximate distance oracle for undirected, planar graphs. Given a graph with n vertices this distance oracle can preprocess the graph in $O(n(\log n)^3/\epsilon)$ time, creating a distance oracle structure using $O(n(\log n)^2/\epsilon)$ space that can then be used to estimate distances between pairs of vertices within a factor of $(1+\epsilon)$ in $O((\log n)^2/\epsilon)$ time.

Finally we will then test the performance of our implementation, using a set of graphs that we generate ourselves. The focus will be on measuring run times, memory usage and estimate accuracy. We end by plotting and discussing the results of these tests, which are in accordance with our expectations for run times and memory usage, but not for accuracy.

2 Helpful Algorithms and Data Types

2.1 Data Types

The main data type used in our implementation is *graphs*. A graph consists of a set of points, called *vertices*, and a set of connections between pairs of vertices, called *edges*. We typically refer to vertices using letters such as v and u, and the edge between such a pair of vertices as (v, u). Furthermore, if G is graph, then $V(G)$ is the set of vertices associated with G, and $E(G)$ is the set of edges. An example of a graph is shown in Figure 1.

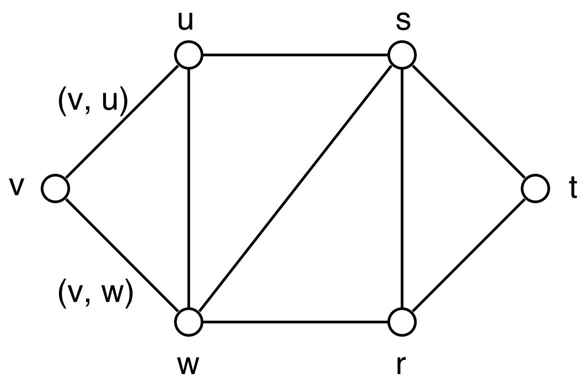

Figure 1: An example of a graph G with labeled vertices and a few labeled edges.

We differentiate between two kinds of edges, *directed* and *undirected*. A directed edge (v, u) allows you to go from vertex v to vertex u, but not from u to v. In the same way the directed edge (u, v) allows you to go from u to v, but not the other way around. On the other hand, an undirected edge (u, v)[1] allows you to go from both v to u as well as from u to v. We will be working exclusively with undirected graphs in our algorithm, meaning all edges in $E(G)$ are undirected.

Another attribute of the graphs that we will be working with, is *weighted* edges. This means that for each edge (v, u) there is a cost w associated with it. This is particularly important when working with shortest paths, since it means that the shortest path is not necessarily the one involving the fewest edges. For the graphs that we will be working with, all edges will have positive, finite weights.

A *path* Q is a list of vertices v_1 to v_n pairwise connected by edges, which means that it is possible to go from vertex v_0 to vertex v_n through vertices $v_1, v_2, ..., v_{n-1}$. For each vertex v_i we keep an integer i to indicate its position in the path, as well as an integer d to indicate the distance from the start of the path, v_0, to v_i. As

[1]We use the same notation for directed and undirected edges. Whether you are dealing with one or the other kind should be clear from the context.

we will only be working with undirected edges, all paths will also be undirected, i.e., it is possible to traverse them in both directions.

A third attribute that we will require from the graphs that we will be working with, is planarity. A *planar* graph is a graph that can be embedded in a 2D plane, that is, drawn on a 2D surface, in such a way that no two edges intersect, except at their endpoints. As an example, if we look at the graph in Figure 1, we see that no two edges intersect each other, apart from at their endpoints at the vertices. If we were to add an edge that went straight from vertex u to vertex r, it would intersect the edge (w, s) at its halfway point, and the embedding would no longer be planar. If we instead drew the edge from u around the outside of the graph and to r, it would not intersect any other edges, and the embedding would still be planar.

A neat feature of a planar embedding of a graph is, that for each vertex in the graph, it uniquely describes the ordering of all incident edges. This allows us to order our graph such that each edge has a pointer to the next edge clockwise in the ordering of the edges, as well as a pointer to the next edge counter-clockwise in the ordering. This is a feature that we will be using later on. Undirected edges have clockwise and counterclockwise pointers for both endpoints. An example is shown in Figure 2.

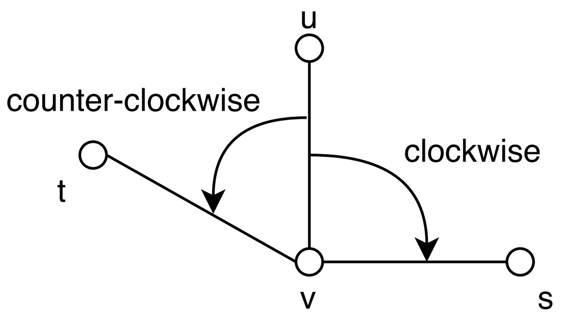

Figure 2: An example of the ordering of edges incident to a vertex v. The edge (v, u) has a clockwise pointer to (v, s) and a counter-clockwise pointer to (v, t).

Another feature of planar graphs is that they are *sparse*, meaning that the number of edges is roughly equal to the number of vertices, $|E| \approx |V|$. This is particularly important when dealing with algorithms such as Dijkstra's, where the run time is $O(|E| + |V| \log |V|)$. For a non-planar graph with edges between all vertices, this would be $O(|V^2|)$, where as for sparse graphs the run time is $O(|V| \log |V|)$. From now on, we will be using n when referring to either $|V|$ or $|E|$ in run time expressions, for instance the run time of Dijkstra's is then $O(n \log n)$.

A fourth and final attribute that we will require of the graphs that we will be working with is connectivity. A connected graph is one where for each vertex v there is a path to every other vertex in the graph. This is a requirement that comes from the planar separator theorem that we will be using.

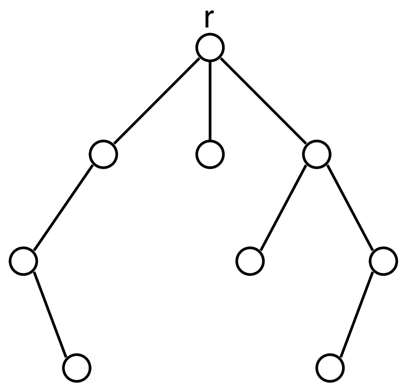

Figure 3: An example of a tree T with four levels and a root r of size nine.

Another data type that we will be using a lot is trees. A tree T consists of a set of nodes, with each node having a pointer to its parent node in the tree, as well as pointers to each of its children. The exception is the so-called root of the tree, which does not have a parent in the tree, only children. An example of a tree can be seen in Figure 3.

In the trees that we will be working with, each node in the tree also keeps track of a set of additional information, namely *level* and *size*. The level of a node n is the level of its parent plus one, which translates to the number of nodes between n and the root of the tree. The root has a level of zero, which gives its children a level of one, their children a level of two and so on. The size of a node n is the number of nodes in the subtree rooted at n, which is the number of children of n, plus the number of children for each of these, and so on.

We also use paths when dealing with trees, in particular root paths. The root path of a node n is the list of nodes from n to the root of the tree, including both. In order to create such a path, you can simply follow the parent pointers from n to the root.

We will regularly be constructing trees based on graphs, such that a vertex v in the graph G has an associated node n in the tree T. Such a tree is called a *spanning* tree if each vertex in the graphs has exactly one associated node in the tree. A tree edge then refers to an edge in the graph between two vertices v and u, with associated tree nodes n and m, where either n is the parent of m in the tree, or m is the parent of n. We will also be talking about root paths for vertices, by which we then actually mean the root path of the associated tree node in the tree.

2.2 Algorithms

In our implementation we will be using an array of different algorithms. We will not be describing how each of these works in detail, but rather just mention which they are, what they do, and what their run time is.

The first of these is *Dijkstra's*. Dijkstra's algorithm is a so-called *single source shortest path*(sssp) algorithm. Given a graph and a source vertex s, the algorithm determines the shortest path from s to all other vertices in the graph, including the target t. The way this is done is by visiting each vertex in the graph in a specific order[2], each time acting on all edges incident to the vertex. Our implementation of Dijkstra's, as well as heaps, is based on the pseudo code in [CLRS09], which is also what we refer to for more information. The run time of Dijkstra is $O(n \log n)$.

Another set of algorithms we will be using is for constructing various kinds of trees. A *shortest path tree* can be constructed from a graph using Dijkstra's, and is a tree where the root node r corresponds to the source vertex s in the graph, and each node n's root path corresponds to the shortest path from the source vertex s to the vertex v, associated with n, in the graph. The run time of this is the same as Dijkstra's.

Given a graph G we can also construct a tree using so-called *breadth-first search*. This is a technique where we start at a vertex s, associated with the root in the tree, and then traverse the graph one level of edges at a time. Each time we visit a new vertex, we check whether we have visited it before, and if not, add it to the tree. This way, the level of each node in the tree is equal to the number of edges between the associated vertex v and vertex s along the path with fewest edges, and its root path is equal to this path. This has a run time of $O(n)$.

Finally we will be using an algorithm to determine the *lowest common ancestor* (LCA) of a pair of nodes in a tree, that is, the node in the tree at which the pair of nodes root paths first intersect. This is simply done by comparing the root paths of the two nodes in order to find the last node they have in common, which is then the LCA. With Q being the longest root path among the two, this has a run time of $O(|Q|)$.

[2]The order in which to visit the vertices is typically determined using a MIN heap.

3 Constructing the Distance Oracle

The focus of our approximate distance oracle is determining the distance between point A and point B as effectively and accurately as possible. In particular we are working with graphs, and so the goal is to find the shortest distance between source vertex s and target vertex t. As we are working with weighted graphs, the shortest path is not necessarily the one containing the fewest edges, but rather the path where the total cost of the edges is lowest.

A simple way to do this is to use an algorithm such as Dijkstra's. Given a graph G with n vertices, Dijkstra's algorithm can determine the shortest path from s to v in $O(n \log n)$ time[3]. If someone then asks for the shortest path from s to another vertex v, we can use our previous calculation to answer the query in constant time. On the other hand, if someone asks for the shortest path from t to v, we would have to run Dijkstra's algorithm all over again. And so, in a situation with a graph with a lot of vertices that we want distances between, we would be spending a lot of time running Dijkstra's. In such a case we need to come up with a better solution.

One trick that we will be using is to assume that we are given the graph in advance and allowed to perform some kind of preprocessing on the graph, before then having to answer distance queries. The result of this preprocessing is a data structure that is often referred to as a *distance oracle*, as it is a construction that can typically answer distance queries in near constant time. As an example, one way to construct such an oracle is to simply run Dijkstra's algorithm on every single vertex in the graph, and then save the results in a big $n \times n$ table. Doing so would take $O(n^2 \log n)$ time and use $O(n^2)$ space, but we would then be able to answer any distance query in constant time.

Another trick that we will use is approximations. Instead of insisting on finding the absolute shortest path, we will settle for finding a path whose cost is close to that of the shortest path. We will be constructing *stretch-t* distance oracles, which given a pair of vertices provides an estimate of the distance between the two that is no more than t times bigger than the distance of the shortest path, and never smaller. We will set $t = (1 - \epsilon)$, and then use ϵ as the parameter that we use to indicate the amount of inaccuracy that we can tolerate.

The approach that we will be using in our implementation is from the article *Compact oracles for reachability and approximate distances in planar digraphs* by Mikkel Thorup from 2004, [Tho04]. In the article they first introduce an approach to constructing a reachability oracle for a directed planar graph, which can be used to determine whether there is a path from a vertex v to another vertex w. They

[3]Assuming that the graphs is sparse.

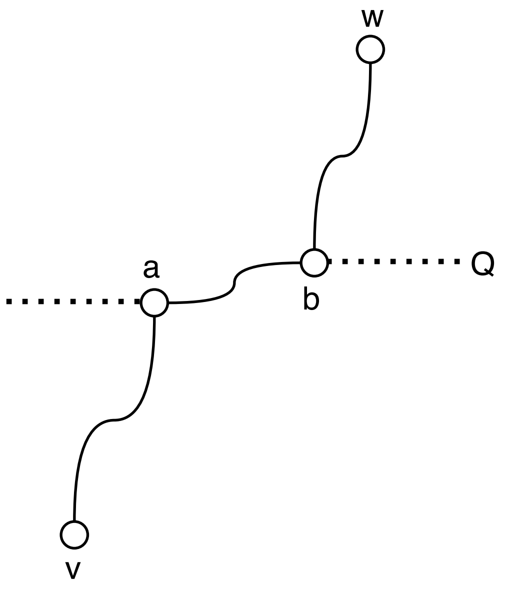

Figure 4: A path from vertex v to vertex w that runs along the separator path, Q, from vertex a to vertex b.

then describe a way to construct an approximate distance oracle for a directed planar graph using some of the same principles. Finally they briefly describe how to construct a similar distance oracle for undirected planar graphs, which is the part that we will focus on. The reason why we chose the undirected case, is that it is technically significantly simpler than the directed case, while it still follows the same basic principles and ideas.

The principle idea behind the approach is that we will construct a number of so-called *separator paths*, and then for each vertex in the graph we will determine a set of connections to these paths. When asked about the distance from one vertex to another, we can then simply add together the distance from the first vertex to one of these paths, the distance from the second vertex to the same path, and the distance between the points of the path where the two vertices connected. This is illustrated in Figure 4. The distance we find might very well be bigger than for the shortest path, but we will make sure that it is within our accepted range.

The main algorithm consists of two stages. The first stage is preprocessing the graph and constructing the distance oracle. The second stage is answering distance queries using our distance oracle.

The preprocessing stage can be further split into two parts. The first part takes a graph and then uses a planar separator theorem to determine three separator paths, whose removal splits the graph into a number of components each with no more than half as many vertices as the original graph. The procedure is then repeated on each of the component graphs, and then on each of the graphs resulting from this, and so on, until the graphs become so small that the separator theorem is no longer valid.

The second part is run on each of the the graphs after the separator theorem has been run on it, but before the graph is split into components. It takes one of the root paths as well as the graph, and then builds on the distance oracle structure that we are constructing. It does this by creating connections between the path and each of the vertices in the graph, which gives us an idea of the distance from each vertex in the graph to different parts of the path.

In the second stage we use the distance oracle structure that we have constructed to answer distance queries. We estimate the distance from vertex v to vertex w using the connections to the separator paths that we established during the preprocessing stage.

All of these stages and parts will be explained in detail below.

3.1 The Planar Separator Theorem

The planar separator theorem used in [Tho04] is based on a paper from 1979 by Lipton and Tarjan, [LT79]. In this paper they present a linear time algorithm that, given a planar graph with n vertices, can determine a separator C consisting of at most $O(\sqrt{n})$ vertices, and whose removal separates the graph into two disjoint components, each with at most $2n/3$ vertices. A lot of their focus is on the size of the separator C, and their algorithm has ten different steps that deals with an array of situations such as a graph that is not connected, by dividing the vertices into different kinds of levels and then using and manipulating these.

In [Tho04] they use a modified version of this algorithm, focusing on step 1, 9 and 10. Given a graph with n vertices, the goal is to find three paths whose removal splits the graph into a number of components, each containing at most $n/2$ vertices. The reason for this modification to the original algorithm is that it gives us a better constant in the recursion, as $2\log_{3/2}(n) \simeq 3.4\log_2(n) > 3\log_2(n)$. This is also the version of the planar separator theorem that we will be using.

The first step is to construct a planar embedding of our graph G. In particular we need to ensure that for each vertex all incident edges are sorted in a clockwise ordering. We have not actually implemented an algorithm for embedding a graph, as this is quite complicated. Instead we will be using graphs that are already embedded, i.e. have sorted edges.

We will then use this ordering of the edges to *triangulate* the graph. We do this by choosing an edge in the graph, follow it to one of its endpoints, choose the next edge in the clockwise ordering at the endpoint, repeat this twice, and then compare the edge we ended at to the one we started with. If they are the same, the three edges we visited form a triangle. If they are not, we add a new edge from the first vertex to the third, thus creating a triangle. An example is shown in Figure 5. We do this for both directions of the edge, and for both the clockwise and counter-clockwise order. By doing this for all edges in the graph, we ensure that all edges are part of exactly two triangles.

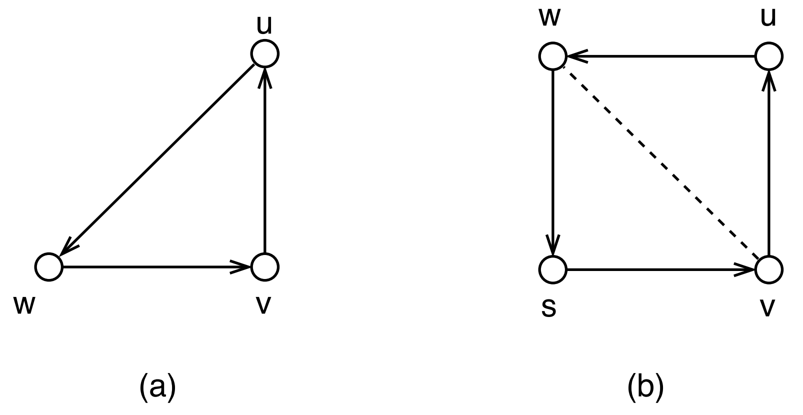

Figure 5: (a) We start with the edge (v, u), we then move to edge (u, w), then to (w, v) and finally once again to (v, u). We began and ended at the same edge, and so nothing needs to be done. (b) We start with edge (v, u), but end at (s, v), and therefore we add the edge (u, s) in order to create a triangle.

Next up we create a spanning tree T for the graph in the form of a shortest path tree with a random vertex as the root, and we choose a random triangle in the graph in the form of three edges. For each of these edges we determine the root path for each of the two endpoints, and join these two root paths together in order to form a cycle. This cycle splits the graph into two parts, an inside and an outside. We call the part containing the triangle the outside, and the other the inside.

We then want to determine the number of vertices contained in the inside-part of the cycle. We do this by traversing the cycle, one vertex v at a time, each time checking for any incident tree edges whose other endpoint w is inside the cycle. Each time one such edge is found, we then have to determine what number to add to the inside count. If v is the parent of w, we add w's size to the inside count as all of the subtree rooted at w must be inside the circle, since the circle itself consists of root paths and thus tree edges. If w is the parent of v, we add the total number of vertices n minus the size of w to the inside count, as in that case all the tree edges in the subtree rooted at w are either part of the cycle or outside the cycle. If the edge is a tree edge, then the inside of the circle contains zero edges, as the two root paths will be identical apart from one being a single vertex longer. When determining the number of vertices inside and outside of a cycle, we do not count the vertices on the cycle as being either, as they will be removed from the graph later on. An example is shown in Figure 6.

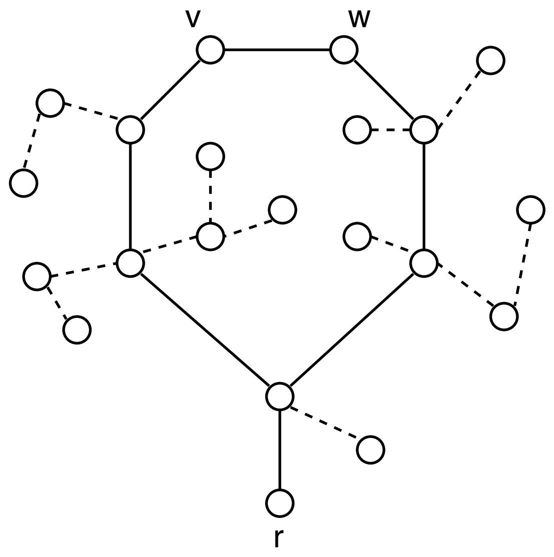

Figure 6: The cycle of the edge (v, w) is plotted with solid lines, and the tree edges incident to the cycle are shown as dotted lines. Only non-tree edge shown is (v, w). One side of the cycle contains five vertices, and the other eight.

By determining the cycle for each of the three edges in the triangle, we can then separate our graph into three different parts, each corresponding to the inside part of one of the cycles, as shown in Figure 7. We also know the number of vertices in each of these three parts, and if none of the parts contain more than $n/2$ vertices, then we are done.

If one of the cycles does contain more than $n/2$ vertices, then we need to select another triangle. If the previous triangle consisted of vertices v, w and u, and the edge (w, u) was the one whose cycle contained more than $n/2$ vertices, then we choose the triangle consisting of vertices w, u and v' instead. This triangle still contains the edge (w, u), but now the inside of the corresponding cycle is the previous outside, which we know contains at most $n/2$ vertices. Meanwhile, what was previously the inside of the (w, u) cycle is now split among the (w, v') and (u, v') cycles, apart from vertex v' which is now part of the triangle, and thus both cycles contain fewer vertices than the previous inside of (w, u). This does not guarantee that the three cycles now all contain less than $n/2$ vertices, but it does ensure that the number of vertices in the biggest cycle is decreased by at least one.

Thus we can simply repeat this until we find a suitable triangle. An example is shown in Figure 7.

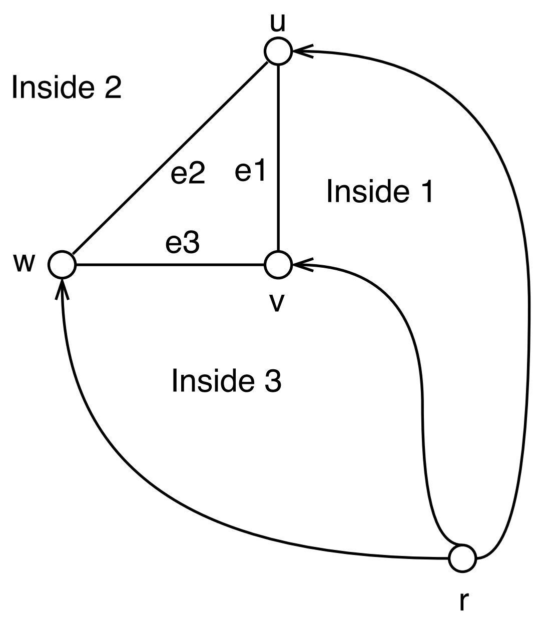

Figure 7: A triangle consisting of the vertices v, u and w, with the root path from root r shown for each, and edges $e1$, $e2$ and $e3$, with an indication of the inside of the cycle created by each of the triangle edges.

Once we have found a triangle that satisfies our demands, we then have our three separator paths in the form of the root paths of the three vertices that makes up the triangle. By following each path and removing all vertices encountered along the way from the graph, and deleting all edges incident to these vertices, we then split our graph into a number of components, each of which has at most half as many vertices as our original graph.

Unlike [LT79] out planar separator theorem requires that the graph is connected, otherwise it might be impossible to find a suitable triangle. Image a situation with a graph with n vertices that has two components, with one of the components having more than $n/2$ vertices. If we then at random choose our first triangle to be in the smaller of the two components, then no matter how much we move the triangle around we will never be able to find a suitable one, as we will never move the triangle to the bigger component.

In order to be able to even find a triangle, we need a graph with at least three vertices. Knowing that we will be using the root paths of the triangle vertices to determine the distance between vertices in the graph it also does not make much sense to run the separator theorem on a graph that does not have at least two vertices that are not part of the triangle. This gives us a lower limit on the graph size of five vertices. Thus we know that when running our planar separator theorem on a graph, and then on each of the component graphs created and so on, we can stop the recursion when the graphs reach a size of five or fewer vertices.

In the worst case scenario we might have to visit every single triangle in the graph before we find the one satisfying our demands, thus taking $O(\Delta)$ time, where Δ is the number of triangles in our graph. As we have triangulated our graph we know that every edge is part of two triangles, and every triangle consists of three edges, which gives us $3\Delta = 2|E|$, assuming $n \geq 3$ otherwise there are no triangles. Remembering that we are working with sparse graphs we then have a run time of $O(\Delta) = O(|E|) = O(n)$.

3.2 Constructing the Distance Oracle

The actual construction of the distance oracle data structure takes place after we have determined the three separator paths using our planar separator theorem, but before we use the paths to split up the graph. The construction described below is run for each of the three root paths separately.

For this recursion we will need to define a concept called ϵ-covering, which has to do with the amount of inaccuracy that we can tolerate in the distances determined. This is a concept that is defined and redefined multiple times in [Tho04]. The most basic definition is, given a vertex v and a path Q containing vertices a and b, we say that (v, a) ϵ-covers (v, b) if $\ell(v, a) + \delta(a, b) \leq \ell(v, b) + \epsilon\alpha$. Here $\ell(v, a)$ and $\ell(v, b)$ are the distances from v to a and b respectively, $\delta(a, b)$ is the shortest path between a and b, which we know from when we created the path, and α is an indicator of the amount of inaccuracy we can accept. What this means is that if the distance from vertex v to a vertex a on the path Q plus the distance from a to another vertex b on the path is shorter than the distance from v to b plus ϵ times some value, then we only need the connection (v, a) and not (v, b).

The definition we will use in our construction, is that we will say that (v, a) semi-ϵ-covers (v, b) if $\ell(v, b) \geq (1-\epsilon)\ell(v, a)+\delta(a, b)$. Here we see that our tolerance for inaccuracy is $\epsilon\ell(v, a)$, that is, ϵ times the distance from v to a.

We start by creating a covering set $C(Q, v)$ for each of the vertices in the graph H that we are currently working on, which is essentially a list connections between the vertex v and the given path Q. The lists start out being empty, and we will then fill in distances as we determine them during the recursion. In fact, as part of the initialization we determine the distances between each vertex v in the graph

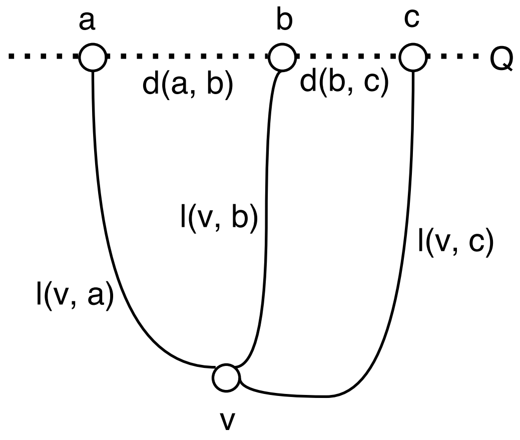

Figure 8: A vertex v and its connections to vertex a, b and c on the path Q, showing the length of the connections.

and the first vertex in the path, a, as well as the last vertex in the path, c. We determine the distances using Dijkstra's algorithm, and then add the distances to each vertex's covering set. Throughout this recursion, when we add distances to a vertex's covering set, we will make sure to keep it ordered according to the order of the vertices in the path[4]. We then start the recursion on our graph H and our path Q.

The recursion takes a graph H_0 and a path Q_0, and requires that we already know the distances from the first and the last vertices in the path, a and c respectively, to each vertex v in the graph.

It starts by determining b, the vertex at the middle of the path. If Q has q vertices, then b is the $\lceil q/2 \rceil$'th vertex in the path. We then use Dijkstra's algorithm to determine the distance from b to each vertex v in H_0, including the vertices in the path, and add the distance to each vertices covering set.

We then define two sets of vertices, U_1 and U_2. For each vertex v in the graph, excluding the vertices in the path, if one of (v, a) and (v, b) semi-ϵ-covers the other, we add v to U_1. Then if one of (v, b) and (v, c) semi-ϵ-covers the other we add v to U_2. We then create a copy of H_0 that we call H_1, where we remove all of the vertices in U_1, and in the same way we make a copy we call H_2 where we remove all of the vertices in U_2. We also split Q_0 in two parts, Q_1 and Q_2, where Q_1 is the part of Q before vertex b and Q_2 is the part after, both including b.

Finally we then recurse on (H_1, Q_1) and (H_2, Q_2).

[4]For instance the distance to the first vertex in the path, a, comes before the distance to the last vertex in the path, c.

If the recursion is run on a path with fewer than 3 vertices, then there is no new middle vertex b to determine distances to, and thus the recursion stops. In the same way, if the recursion is run on a graph with no vertices that are not part of the path, then there is nothing to determine distances to, and so the recursion stops.

When the recursion is complete we have created a ordered covering set for the given path Q for each vertex v in the given graph H. From Lemma 3.18 in [Tho04] we have that each vertex has $O((\log |V(Q)|)/\epsilon)$ connections to the path, as the recursion depth is at most $\log |V(Q)|$ and there are at most $O(1/\epsilon)$ minimal calls involving a vertex v, where a minimal call is a call that involves v with no descending calls involving v. As our separator paths are of unbounded length this gives us a size for each covering set of $O((\log n)/\epsilon)$.

In [Tho04] they use the linear time sssp algorithm from [Tho99] in Lemma 3.18, which gives them a run time for the recursion of $O(n(\log n)/\epsilon)$. We are using Dijkstra's as our sssp algorithm, and so the run time for our recursion is $O(n(\log n)^2/\epsilon)$.

3.3 Preprocessing

The actual preprocessing algorithm is then constructed using the two parts explained above. It takes a graph G, and initiates the distance oracle structure, where each vertex in the graph has a list of covering sets as well as a pointer to their final call node in the recursion tree. The distance oracle structure also has a list where all of the separator paths will be added. The recursion tree is then created with just the root, which has a value of three. The value of a node in the recursion tree indicates the total number of separator paths that have been created in that particular branch of the tree. The recursion is then started by handing the graph, the distance oracle structure and the tree root to the recursion algorithm.

The recursion takes a graph H, the distance oracle structure and a recursion tree node n. It starts by determining the three separator paths using the planar separator theorem, and adds them to the distance oracle structure's list of paths. For each of the separator paths it runs the algorithm that determines the covering set for each of the vertices in the graph, and adds these covering sets to the distance oracle structure as well. For each of the sets of covering sets it also attaches the ID of the path that was used, which is equal to the paths position in the list of paths.

It then removes the three separator paths from H, by removing all the vertices in the three paths as well as any incident edges, which splits the graph up into components of at most half the size of H. For each of the vertices thus removed it sets their final call as the node n, and for each of the new component graphs it creates a new tree node in the recursion tree, with n as the parent and with a value equal to the value of n plus 3. It then finally recurses on each of the graph components and their tree nodes.

As we are dividing the graph into components of at most half the size in each step of the recursion, the depth of the recursion tree for a graph G with n vertices is $O(\log n)$. Recalling that each step of the recursion takes $O(n(\log n)^2/\epsilon)$ time, this gives us a total run time of $O(n(\log n)^3/\epsilon)$.

With n vertices, a recursion depth of $O(\log n)$, and recalling that each covering set C has $O((\log n)/\epsilon)$ connections, we get a total size of $O(n(\log n)^2/\epsilon)$ for the covering sets. Each separator path has a size of $O(n)$, and with $O(\log n)$ separator paths this gives us a total size of $O(n \log n)$. In total the size of the distance oracle structure is then $O(n(\log n)^2/\epsilon)$.

3.4 Answering Queries

Once we have finished our preprocessing of the graph, we then have a distance oracle structure that can be used to estimate distances in the graph.

In order to determine the distance from vertex v to vertex w, we need to consider their connections to the various separator paths. v has a list of covering sets for different separator paths, as does w, and so we need to determine the separator paths that they have in common, as those are the ones that we are interested in. We do this by using the recursion tree. v and w both have a final call node in the tree, and so we determine the *lowest common ancestor*(LCA) of these two nodes by looking through the root path of each and determining the point where they first differ. The LCA of v and w is then the last call in the recursion tree where v and w were both part of the particular graph the recursion was running on. For this recursive call, and all previous calls, both v and w where part of the graph, and so covering sets will have been created for both the vertices for each of the three separator paths in each of the recursive calls. Beyond the LCA, v and w will have been part of different recursion branches in the tree, and so will have created covering sets for different separator paths, that we are then not interested in. Thus, if l is the LCA node of v and w's final call nodes, and l has a value of s, then we know that the first s covering sets in v and w's lists of covering sets will be for the same separator paths, and so these are the sets we are interested in.

For each of these separator paths, Q, we will have a covering set for v and a covering set for w. We then merge the two covering sets together, while ensuring that the connections are still ordered according to the ordering of the vertices in Q. Then we go through the list and consider each neighboring pair of elements.

If both are connections from v to Q or if both are from w to Q, we move on to the next pair. If one is a connection from v to vertex q_i in Q, and the other is a connection from w to q_j in Q[5], we then determine the distance from v to w via q_i and q_j as $\ell(v, q_i) + \delta(q_i, q_j) + \ell(q_j, w)$.

We do this for the entire merged list, which gives us the shortest path from v to w via Q. We then do this for every one of the separator paths that we found v and w to have in common, and the shortest distance from among these paths is the one we report.

Recalling that the depth of our preprocessing recursion was $O(\log n)$, determining the LCA for v and w's nodes takes $O(\log n)$ time. For each node from the LCA and to the root of the recursion tree, we then have three separator paths that we will have to consider. We also recall that each covering set had $O((\log n)/\epsilon)$ connections that we will have to consider, and so the total run time of our query implementation is $O((\log n)^2/\epsilon)$.

In [Tho04] they achieve a query time of $O(1/\epsilon)$ by improving this technique in a number of ways. One of the ways is to use what they refer to as *frames* during their preprocessing recursion, which is a way of manipulating the separator paths, which means that during querying we only have to consider a constant number of paths, rather than $O(\log n)$ paths. Another way is to use the LCA algorithm from [HT84], which preprocess the tree in order to answer LCA queries in constant time. A third way they use is to reduce the number of connections per covering set to $O(1/\epsilon)$, by removing connections based on a stronger ϵ-covering definition.

[5] Where q_i and q_j might or might not be the same vertex

4 Performance Testing

We have chosen to implement our approximate distance oracle in the programming language *C* [KR88]. We chose *C* as it allows us a great deal of control over our implementation, in particularly over the usage of memory, and as it is a very fast and efficient language, which allows us to make an implementation that runs very fast. We have implemented all of the parts that we need ourselves, including all the data types and algorithms, using only a few standard libraries for things such as I/O, run time measurements and calculating logarithms.

There are four different parameters that we will be measuring to determine performance. The first of these is the run time of the preprocessing algorithm. We measure this using the `clock_gettime()` function from the `time.h` library. This function has an option to use a special monotonic clock, so that even if the operating system suddenly decides to update the internal clock it does not interfere with our time measurements. The function claims to be able to accurately measure time on a nanosecond scale, but whether or not it really is that accurate is not critically important, as we are not as interested in the absolute run time values as we are in the relative difference in run time values for graphs of different sizes.

The second parameter we measure is the memory usage of the distance oracle structure created by the preprocessing. We determine this by simply going through the entire structure adding the sizes of all the different covering sets together. We measure it in numbers of 32 bit words, as that is the size of a normal integer variable such as a distance.

The third parameter is the distance oracle query time, that is, the time it takes for our distance oracle to estimate the distance between two vertices. We measure this in the same way as we measured the preprocessing time. The query time for a pair of vertices may vary based on the depth of the two vertices final call depths in the recursion tree, and so to account for this we measure the query time for all combinations of pairs of vertices in the graph and determine an average.

The fourth and final parameter is the accuracy of the distance estimates made by our oracle. We measure this by using our oracle to estimate the distance between all combinations of pairs of vertices in the graph. We then compare these distance with those we can get from using Dijkstra's, which, assuming our implementation of Dijkstra's is correct, should return the distance of the shortest path between each pair of vertices. By dividing the distance we get from our distance oracle with the distance we get using Dijkstra's, we get a ratio. This ratio should never be smaller than 1, and if for instance our distance oracle is a stretch-2 oracle, with an ϵ value of 1, then the ratio should be no bigger than 2.

As our performance test determines the distance between all combinations of pairs of vertices in our graph, it has a run time of $O(n^2)$. And so even though running the test on a graph with a thousand vertices only takes a couple of minutes, running it on a graph with five thousand vertices takes a couple of hours, which is why we have chosen to use graphs with sizes in the range of one to five thousand vertices.

The performance tests have been performed on a Lenovo Z500 laptop running Ubuntu 14.04 with an Intel I7-3632 2.20 GHz processor and 8 GBs of RAM.

4.1 Graphs

For testing the performance of our implementation we use sets of grid-based graphs that we have generated ourselves. The graphs consists of a grid of vertices x wide and y deep. All vertices, apart from those at the edge of the graph, have edges to the neighboring vertices above, below, to the left and to the right in the grid, as well as to the vertices at the north-west and south-east positions. After generating the graph we run our triangulation algorithm on it, in order to ensure that the borders of the graph are also properly triangulated. An example of a 3x3 graph is shown in Figure 9. All edges in the graph are assigned a random weight between 1 and some maximum value. This includes the potentially far-reaching edges on the border of the graph, which could potentially create some interesting shortcuts.

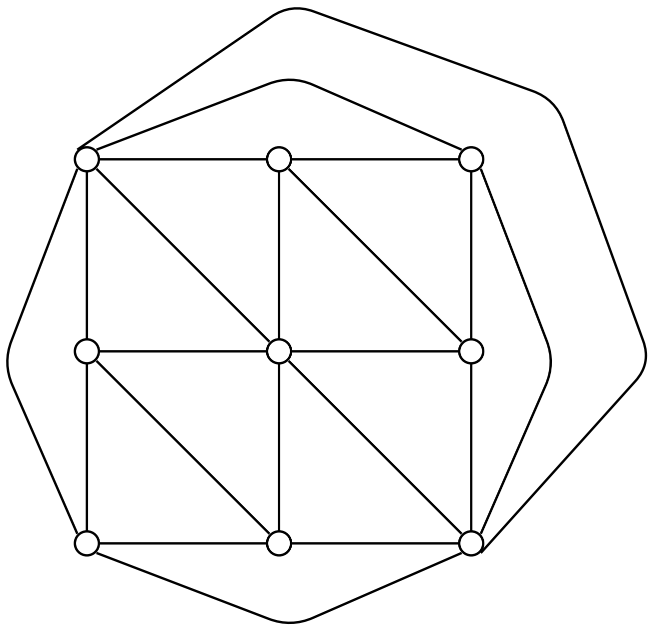

Figure 9: An example of a 3x3 grid graph as generated by our graph generator.

For testing we have then generated a set of graphs with the same specific number of vertices, measured the various quantities we are interested in for each of the graphs, and then determined an average over all the graphs. Specifically we have generated five different sizes of graphs, one with 990 vertices (30x33), one with 2000 vertices (40x50), one with 3000 vertices (60x50), one with 3960 vertices (60x66) and one with 4970 vertices (70x71). For each size of graph we have then generated 100 graphs, and for each of those we have measured the four different parameters.

4.2 Preprocessing

Figure 10 shows the results of our measurements of the preprocessing stage's run time. We have run the algorithm on the graphs described above, for values of ϵ of 0, 1 and 2.

As we expect a run time for the algorithm of $O(n(\log n)^3)$, we have fitted the data with a curve $y = a * x * (\log x)^3 + b$. Here a and b are then the constants that gets lost in the big-O notation, and so are the numbers that would be of particular interest if we were comparing our algorithm to another algorithm with a similar run time. For $\epsilon = 0$ we get a value of $a \simeq 500$, and for $\epsilon = 1$ and $\epsilon = 2$ we get $a \simeq 300$. For all three values of ϵ we get a value of b in the order of -10^9.

On Figure 10 we can see that the preprocessing run time increases with graph size, as we would expect, and the fit shows that our estimated run time of $O(n(\log n)^3)$ is not entirely unreasonable. Worth noting is the significant increase in run time from two to three thousand, which the fit also seems to have a hard time adjusting to. This might be due to the size of our data structures exceeding some internal values such as memory page size or TLB size, but it is difficult to say without further testing.

We also note that the run time increases as we decrease the value of ϵ, this is particularly evident for $\epsilon = 0$. This is in accordance with what we would expect, as a lower value of ϵ means that in the preprocessing stage we will have fewer cases of connections ϵ-covering each other, and thus we will need to add more connections to the distance oracle structure, which takes more time.

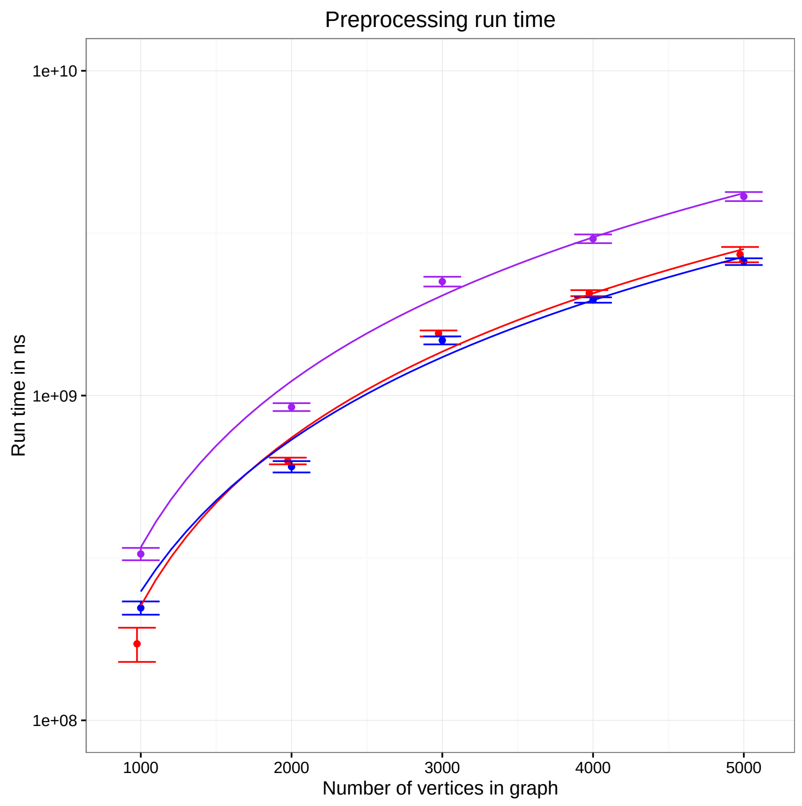

Figure 10: Plot of the run time in nanoseconds of the preprocessing algorithm for graphs of different sizes and for different values of ϵ. Purple is $\epsilon = 0$, red is $\epsilon = 1$ and blue is $\epsilon = 2$. Fitted lines are of the form $y = a * x * (\log x)^3 + b$.

Figure 11 shows the results of the measurements of the preprocess stage's memory usage. We have fitted the data with a curve $y = a * x * (\log x)^2 + b$, and get $a \simeq 1.1, b \simeq -28,000$ for $\epsilon = 0$ and $a \simeq 0.4, b \simeq 7,000$ for $\epsilon = 1$ and $\epsilon = 2$. Interestingly we get a negative b value for $\epsilon = 0$ but positive b values for $\epsilon = 1$ and $\epsilon = 2$.

Once again the results seem in accordance with our expectations, the fit is not unreasonable, and the values for $\epsilon = 0$ are once again significantly higher than for $\epsilon = 1$ and $\epsilon = 2$. The reason behind this difference is the same as for the run time, as a lower value of ϵ means more connections per vertex per path, and more connections require more memory.

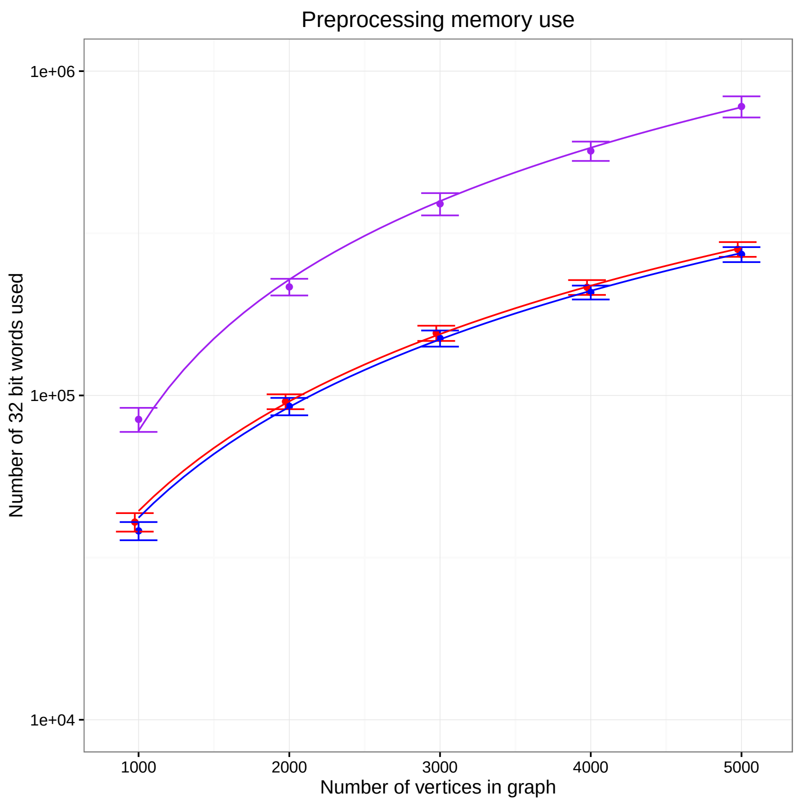

Figure 11: Plot of distance oracle structure memory usage in units of 32 bit words for graphs of different sizes and for different values of ϵ. Purple is $\epsilon = 0$, red is $\epsilon = 1$ and blue is $\epsilon = 2$. Fitted lines are of the form $y = a * x * (\log x)^2 + b$.

4.3 Querying

Figure 12 shows the results of the measurements of the distance oracle query time. We have fitted these with a curve $y = a * (\log x)^2 + b$, and we get $a \simeq 26, b \simeq -570$ for $\epsilon = 0$, $a \simeq 19, b \simeq -460$ for $\epsilon = 1$ and $a \simeq 20, b \simeq -690$ for $\epsilon = 2$. Once again we get relatively large, negative values of b.

The plot is very much in accordance with what we have seen previously, once again we see that one of the size 2000 data points is off curve, and the $\epsilon = 0$ values are again significantly higher than the others. The explanation is the same as before, that for lower values of ϵ we need more connections to each path, and the way the distance oracle determines distances is by looking through all the connections for different paths, so more connections leads to a longer query run time.

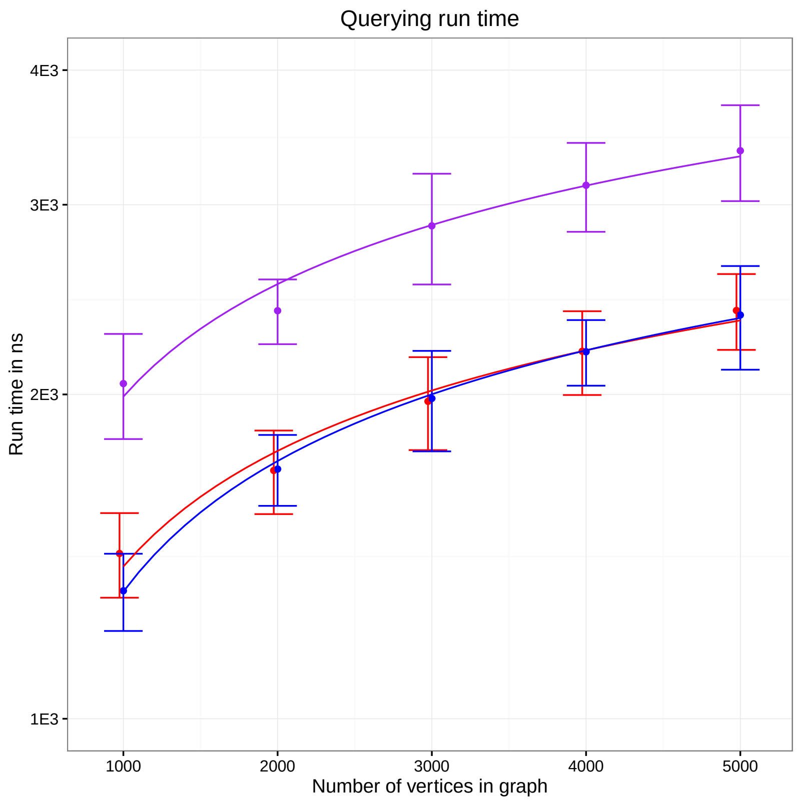

Figure 12: Plot of distance oracle query time in nanoseconds for graphs of different sizes and for different values of ϵ. Purple is $\epsilon = 0$, red is $\epsilon = 1$ and blue is $\epsilon = 2$. Fitted lines are of the form $y = a * (\log x)^2 + b$.

Finally figure 13 shows the results of our measurements of the distance oracle accuracy. Here we see that the data for the three different values of ϵ is more or less coinciding for all five different sizes of graphs, with a seemingly random ordering of the three data sets for each graph size.

This is very much not in accordance with our expectations, as it pretty much goes against the definition of a stretch $(1 + \epsilon)$ distance oracle. The data seems somewhat reasonable for $\epsilon = 1$, but for $\epsilon = 2$ we would then expect the accuracy to be lower, meaning the ratio between oracle distance and Dijkstra distance should be higher, and for $\epsilon = 0$ we would expect the ratio to be very close to one.

The fact that this is not the case, tells us that there is an error somewhere in our implementation, although whether it is an error in the actual implementation of the distance oracle or if it is an error in the implementation of the performance testing, we have not been able to conclusively determine.

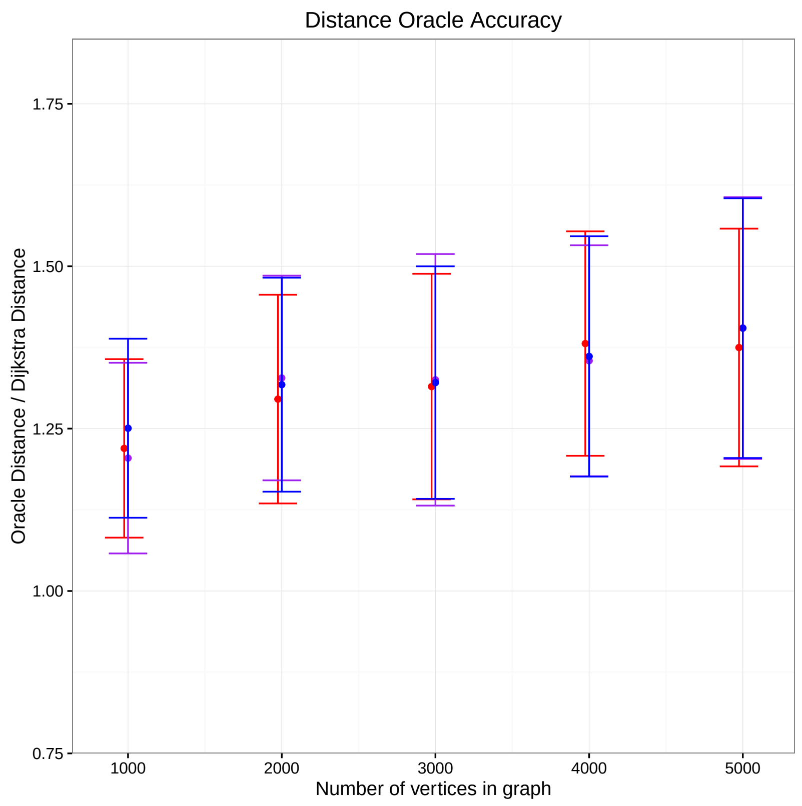

Figure 13: Plot of the ratio between the distances estimated by the distance oracle and the distances determined by Dijkstra's for graphs of different sizes and for different values of ϵ. Purple is $\epsilon = 0$, red is $\epsilon = 1$ and blue is $\epsilon = 2$.

5 Conclusion

We have implemented a stretch-$(1+\epsilon)$ approximate distance oracle for undirected, planar graphs, based on the article [Tho04]. Our theoretical discussion of the algorithm predicts that we can preprocess a graph with n vertices in $O(n(\log n)^3/\epsilon)$ time, creating a distance oracle structure using $O(n(\log n)^2/\epsilon)$ space that can answer distance queries in $O((\log n)^2/\epsilon)$ time.

We have implemented our distance oracle, including all needed data types and other algorithms, in C. We have generated sets of undirected, planar graphs based on grids of vertices, with graphs having sizes in the range of one to five thousand vertices. We have then tested the performance of our implementation using these graphs, by measuring run times, memory usage and oracle accuracy.

The results of the performance tests for run time and memory usage are in accordance with what we have predicted. The results of the accuracy test for our distance oracle is not in accordance with what we expect, and seems to indicate an error in either our distance oracle implementation or in our performance tests. We have not been able to conclusively determine the error.

5.1 Future Work

One obvious candidate for future work is to determine why the accuracy of our distance oracle does not seem to depend on the value of ϵ. This is either due to an error in the implementation of the preprocessing algorithm, or due to an error in our performance test implementation.

The performance test implementation is another candidate for future work, as it was very hastily constructed towards the end of the project, and has undergone very little testing or optimization. The way we measure the various parameters is not entirely obvious or optimal, and the way we save, process and plot the data has a lot of room for improvements.

Also due to time constraints the actual implementation of our distance oracle algorithm is mostly unoptimized, and in some places rather crude. Putting some more work into this, although it should not impact the overall results, could potentially improve run times.

Another obvious candidate would be to implement some of the more advanced tricks used in [Tho04] to reduce memory usage and/or query run time, such as frames and fast LCA and sssp algorithms.

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusetts, 3rd edition, 2009.

[HT84] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancester. *SIAM Journal on Computing*, 13(2):338–355, 1984.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[Tho04] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.