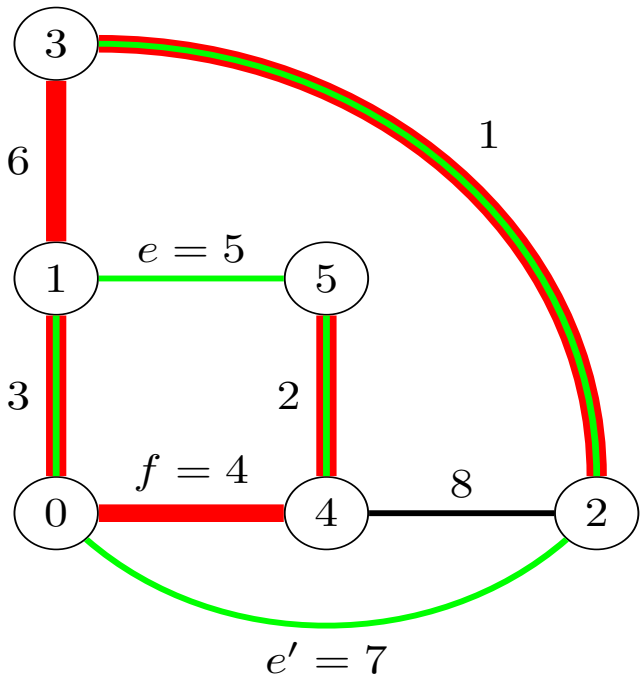
Department of Computer Science, Aarhus University

Bachelor report (15 ECTS) in Computer Science

Online graph exploration (Online graf udforskning)

Authors
Asger Klinkby Jørgensen, 201806325
Jacob Kjær Monsrud, 201808665

Supervisor
Prof. Gerth S. Brodal



June 8, 2021

AARHUS UNIVERSITET

Abstract

The online graph exploration problem asks for the minimum weight tour visiting all vertices and returning to the origin. Initially, only the origin vertex is known to the searcher. Exploring a new vertex will reveal its adjacent vertices and the weights of its incident edges. The searcher must then continuously decide its exploration strategy based on partial knowledge of the graph.

We study the competitive ratio of three different algorithms and related lower bounds. We describe the simple nearest neighbour algorithm which has a tight bound of $\Theta(\log n)$ on the competitive ratio on general graphs. We show that $\Omega(\log n)$ is a lower bound. Then for DFS, we show a that it obtains a competitive ratio of 2 on unweighted graphs. We then show that any algorithm has a lower bound on the competitive ratio of 2 for unweighted graphs. This concludes that DFS is a best online algorithm for unweighted graphs. The $\texttt{Blocking}_\delta$ algorithm is treated in depth. We provide examples and a description of the behaviour. We show that $\texttt{Blocking}_\delta$ is 16-competitive on weighted planar graphs. The proof we provide contains a correction of claim 1 in [15]. Additionally, we construct a planar graph for which our implementation of $\texttt{Blocking}_\delta$ obtains a competitive ratio of 5.25. This graph is a simple cycle with exponentially increasing edge weights which force $\texttt{Blocking}_\delta$ to backtrack all previously explored edges every time it visits a new vertex.

We also conduct a series of experiments with the $\texttt{Blocking}_\delta$ algorithm. These show that $\texttt{Blocking}_\delta$ performs well on small random planar graphs with a competitive ratio of around 2 on these. We find that the δ-parameter does not affect the competitive ratio much on these graphs. Increasing the number of vertices in the small random planar graphs increase the competitive ratio from an average of 1.35 on graphs with below 7 vertices to an average of 2.21 with $13 - 20$ vertices. We use a *MST* lower bound for the optimal tour to experiment on graphs with 20-200 vertices. This experiment indicates that the competitive ratio of $\texttt{Blocking}_\delta$ stagnates at an average of approximately 2.7 on the random graphs we were able to generate.

Table of Contents

Part I
Introduction

The travelling salesman problem asks us to find a shortest tour visiting all cities and returning to the origin. Normally we are given a map with all the distances between cities, but suppose we are in an unknown country without this map. The only information available is the city signs telling us how far there is to neighbouring cities. In the classical travelling salesman problem we are only allowed to visit each city once, but we will allow for multiple visits. We still want to visit all the cities, and may even create our own map as we go along. This problem is known as the map construction problem, or as we will refer to it the *online graph exploration problem*. It is online in the sense that the information is revealed gradually. This is in contrast to the travelling salesman problem which can be referred to as the offline graph exploration problem. Formally we will define the online graph exploration problem in terms of an undirected graph with weighted edges. The searcher starts in an origin vertex, and only know the labels of adjacent vertices and the weights of incident edges. Each time the searcher traverses an edge to a new vertex a cost is incurred equal to the weight of the edge. Upon arrival, the adjacent vertices and incident edges are revealed. The goal is to visit all vertices and return to the origin with the minimum cost.

Our focus will be to investigate how much longer the online tour is compared to the offline tour. In technical terms, we want to find the competitive ratio between the optimal offline solution and the different online algorithms. We prove both upper and lower bounds as well as conduct experiments to investigate this.

Motivation

The online graph exploration problem is interesting because its competitive ratio is not very well understood [7]. There is a large gap between the lower bound of $3.3\overline{3}$ and the upper bound of 16 for planar graphs. In the words of Miyazaki [15] "Narrowing this gap is a challenging problem". Moreover, for general graphs it is an open question whether or not an algorithm with constant competitive ratio even exists. The problem is an active research area with several articles published very recently, for example [3, 9]. It is stated in [3] that it is a fundamental problem in robotics, where it is used for map construction and exploring unknown environments [9]. There are also applications in network security and for exploring data on the internet [15].

Organisation

The report consists of four parts. In Part I we provide basic definitions, introduce the online and offline graph exploration problem, and give an overview of known results in the literature. In Part II we describe some of these results. As a starting point, we use the nearest neighbour algorithm to establish the difficulty of the problem. Then we go into more detail showing that DFS has a competitive ratio of 2 on unweighted graphs. And further showing that this bound is tight, by proving a lower bound on unweighted graphs for all algorithms. The $\texttt{Blocking}_\delta$ algorithm is treated in depth in Section 7. We present a proof that the competitive ratio of $\texttt{Blocking}_\delta$ is 16 on planar graphs. In writing this proof, we encountered an error in claim 1 in [15]. This error was confirmed and based on a correspondence with one of the authors, our Lemma 7.3 present a revised proof. We also present a graph of which our $\texttt{Blocking}_\delta$ implementation obtains a competitive ratio of 5.25. In Part III we present a series of experiments made using our implementation of $\texttt{Blocking}_\delta$. These study the competitive ratio of $\texttt{Blocking}_\delta$ on small random planar graphs showing that $\texttt{Blocking}_\delta$ often obtains a competitive ratio of around 2. Finally, we conclude on our studies in Part IV.

1 Basic definitions

Let us first define a *graph* $G = (V, E)$ as a set of vertices and a set of edges. Each edge $e = (u, v) \in E$ connects two vertices u and v. We will consider settings with unweighted and weighted graphs. In weighted graphs we associate a non-negative weight w with all edges. We will only consider *undirected graphs*, in which all edges are undirected. In an undirected graph, a *cycle* is a path $(v_1, v_2, \ldots, v_k)$ with $k > 1$, in which, $v_1 = v_k$ and where all edges are distinct. A *simple cycle* is a cycle where all vertices are distinct except for

$v_1 = v_k$ [5, Appendix B]. A *Hamiltonian cycle* of an undirected graph is a simple cycle containing all vertices of the graph. We adopt the non-standard use of the term *tour* from [15], this definition of a tour is similar to that of a Hamiltonian cycle, but without the restriction that edges and vertices must be distinct.

Definition 1.1. *Tour*
In an undirected graph $G = (V, E)$ a *tour* is a path $(v_1, v_2, \ldots, v_k)$ with $v_1 = v_k$ which contains all vertices in V.

If a vertex is not reachable from the origin vertex the tour will be undefined. We define the *cost* of a tour to be the total length of all edges in the tour including duplicate edges. The problem of finding a *tour* which minimises the cost as much as possible, is the formal definition of the *graph exploration problem* that we will use. Our project will be investigating solutions to this problem. For this, we will consider two classes of algorithms *offline graph algorithms* and *online graph algorithms*. Given a graph $G = (V, E)$ an *offline graph algorithm* has total knowledge of V and E throughout its execution. On the contrary, given a graph $G = (V, E)$ an *online graph algorithm* has only partial knowledge of V and E when it is initiated. Instead the remaining vertices and edges are explored throughout the execution of the algorithm.

When working with online problems, such as the online graph exploration problem, it is useful to compare online solutions to the optimal offline solution. For this we provide the following three definitions.

Definition 1.2. *Competitive ratio*
Given an algorithm and a concrete instance of a minimisation problem, we define the competitive ratio as [15]

$$\text{competitive ratio} = \frac{\text{cost of the online algorithm}}{\text{cost of the optimal offline algorithm}}$$

The competitive ratio gives a measure for how much worse the solution found by the online algorithm is compared to the optimal offline. We will sometimes use the notation $\frac{Algorithm}{Optimal}$ to denote the competitive ratio of an online algorithm. *Optimal* refers to the cost of the optimal offline and *Algorithm* to the cost of the online algorithm [16].

Definition 1.3. *c-competitive*
For a positive real number c, a minimising online algorithm *Algorithm* is c-competitive if it is true for all inputs that competitive ratio $\leq c$.

A 4-competitive online algorithm solving the exploration problem will therefore produce a tour that is at most 4 times the length of the optimal offline tour.

The approach to calculating c is to find a lower bound for the optimal tour and an upper bound for the online algorithm, as this will imply the needed inequality $\leq$. We will later show such constructions in great detail. In fact, a large part of this project is to show that the online graph exploration algorithm $\texttt{Blocking}_\delta$ is 16-competitive on planar graphs.

Definition 1.4. *Competitive ratio on a graph class and c-competitive on a graph graph class*
For a given graph class the competitive ratio of an algorithm is the least upper bound on the competitive ratio for all instances in the graph class [3]. For a positive real number c, an online algorithm *Algorithm* is c-competitive on a graph class if it is true that the competitive ratio on the graph class is less than or equal to c.

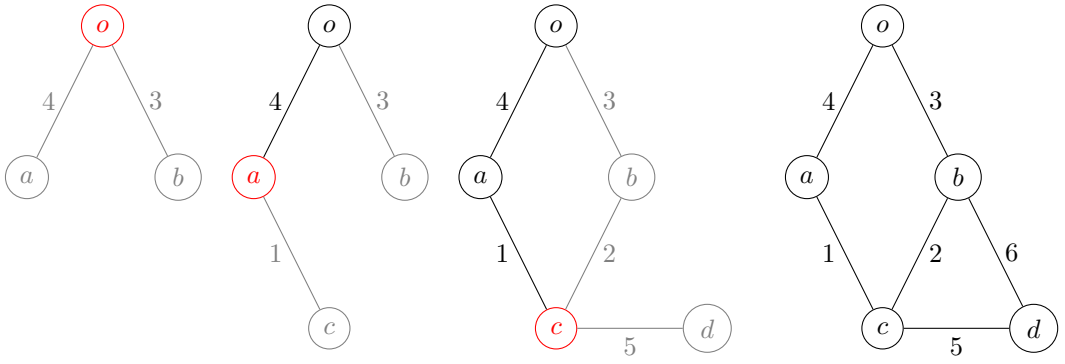
2 Online graph exploration

This section is meant to give a better understanding of the online graph exploration problem.

An online graph exploration algorithm uses the notion of a searcher. This is meant as a reference to the vertex from which the algorithm is currently basing its decision. The order of which the searcher visits the vertices defines the tour. The goal is to discover the entire graph by visiting all vertices and return to the origin vertex. Online graph exploration algorithms also use the notion that some vertices are explored and some are not. Throughout the execution of the algorithm, more and more vertices, are explored.

Definition 2.1. *Explored*
We say that a vertex is explored if it has been visited by the searcher, and an edge $e = (u, v)$ is explored if both u and v is explored.

Exploring new vertices and edges requires that the searcher visits adjacent vertices. That is when the searcher explores a vertex v, all the adjacent vertices and their labels and incident edges with their weights are revealed. Figure 1 illustrates this concept. Initially, only the origin vertex o is explored. This is where the exploration starts. The adjacent vertices and incident edges of o are revealed to the searcher. Based on the algorithm, the searcher may choose to traverse the edge (o, a) or (o, b) as in Figure 1. Here the algorithm chooses (o, a) and a new vertex c and the edge (a, c) is revealed. The searcher must then again decide how to continue the exploration. It can either traverse (a, c) or backtrack and traverse (o, b). On Figure 1 it chooses to traverse (a, c). This continues until the entire graph has been explored, at which point the searcher must return to the origin vertex o before the algorithm terminates. Note that the example on Figure 1 did not finish exploration, as there are still unexplored vertices b and d.



(a) **Online algorithm**: The first 3 iterations of an online algorithm exploring a graph. Vertices and edges are revealed as the searcher explores the graph. The searcher's location is marked with red. Explored vertices and edges are black and unexplored are grey.

(b) **Offline algorithm**: The algorithm setting has total knowledge of the graph

Figure 1: The two different perspectives of the same graph, using online exploration algorithms on the left (a) and offline exploration algorithms on the right (b)

We will in the next section consider the offline problem and the solution to it. This will provide some background knowledge and a better understanding of the problem before diving deeper into the online setting in later sections.

3 Offline graph exploration

The offline graph exploration problem is closely related to the famous travelling salesman problem (*TSP*). It is the version called the *travelling salesman problem with multiple visits* (*TSPM*) that exactly match our definition of the exploration problem.

We will first describe the standard *TSP* in the following section. Then in Section 3.2 we describe *TSPM*. Lastly, we show how to reduce *TSPM* to *TSP* in order to use standard *TSP* solvers in Section 3.3.

3.1 *TSP*

TSP is often formulated as follows: Given a *fully connected* graph, what is the shortest path that visits all vertices *exactly* once and returns to the origin. This is the same as the minimum weight Hamiltonian tour.

TSP is proved to be NP-Hard. Thus there cannot exist a polynomial-time exact algorithm, assuming P $\neq$ NP. This essentially implies that the best solution is a brute-force approach with some heuristics. These heuristics do however drastically improve the complexity in practice. The currently best solutions are

variants of the branch-and-bound optimisation algorithms [10]. We will not discuss these solutions, as this project is not meant to be about the *TSP*, but rather the online counterpart and its competitive ratio. We do, however, use a *TSP*-solver [11] in our experimentation section that makes use of these ideas.

When working with graph exploration we only require that the graphs are connected, but not fully connected. A connected graph may not contain a Hamiltonian tour and the *TSP* is therefore not always feasible in our setting. Instead of *TSP*, the offline version of the graph exploration problem is the *TSP* with multiple visits, which we will discuss next.

3.2 *TSP* with multiple visits

A variant of the *TSP* is the *TSP* with multiple visits (*TSPM*). This problem is defined as follows: Given a *connected* graph, what is the shortest path that visits all vertices *at least* once and returns to the origin.

This problem matches our definition of the exploration problem. It allows visiting the same vertex or traversing an edge multiple times. This makes backtracking possible in graphs where backtracking is necessary to complete a tour.

When finding the competitive ratio of an online exploration algorithm we need to compare it to the optimal offline solution. Therefore we need to be able to compute the optimal solution to the *TSPM* as efficiently as possible. It is however an NP-hard problem just like *TSP*. There are implementations available that solve *TSP* in a reasonable time, such as [11]. In order to use these, we will in the next section show a reduction from *TSPM* to *TSP*.

3.3 Reduction from *TSPM* to *TSP*

As discussed above, we cannot immediately use the *TSP*-solvers, as not every connected graph contains a Hamiltonian cycle. We, therefore, define a reduction $TSPM \leq_P TSP$ such that the cost of the optimal tour on the fully connected *TSP* instance is the same as the cost of the optimal tour on the connected *TSPM* instance. This reduction is needed to make use of the efficient *TSP*-solvers on not fully connected graphs. We will in the following refer to the original *connected* graph as the *TSPM* graph and the newly constructed *fully connected* as the *TSP* graph.

The Reduction

The idea is to create edges between every pair of vertices, corresponding to the shortest path between these. More precisely, the reduction can be described as follows. Let $G_{TSPM} = (V_{TSPM}, E_{TSPM})$ be the original *TSPM* graph. Copy the vertices V_{TSPM} and define the *TSP* graph as $G_{TSP} = (V_{TSP} = V_{TSPM}, E_{TSP} = \emptyset)$, which at this point contains no edges. Now run a all-pairs-shortest-path algorithm on G_{TSPM}. This obtains the shortest distance between every pair of vertices. We can now add edges to G_{TSP}. For every unordered pair $v, u \in V_{TSP}$ add an edge $e_{TSP} = (v, u)$. Let the weight of e be the length the shortest path from v to u. This concludes the reduction [3, 20].

Consider as an example Figure 2. Here the shortest path from a to c is the path through edges $((a, b), (b, d), (d, c))$ with length 8. The reduction creates, among others, an edge (a, c) with length 8 in the *TSP* graph G_{TSP}.

Figure 2: Reduction from *TSPM* graph G_{TSPM} (left) to *TSP* graph G_{TSP} (right)

It is important to note that the original *TSPM* graph did not necessarily obey the triangle inequality, but the fully connected graph obtained by the reduction does. Consider an edge $e_{TSPM} = (u, v)$ in the

original *TSPM* graph. If a path $u \rightsquigarrow v$ with length less than $|e_{TSPM}|$ is found by the all-pairs-shortest-path algorithm, the new edge $e_{TSP} = (u, v)$ will have the weight equal to the shortest path and not the weight of e_{TSPM}. Therefore, the edge $e_{TSP} = (u, v)$ will always be a shortest path between the two vertices u, v. We will use the triangle inequality to show the below Lemma 3.1. The *TSP*-solver assumes the classical *TSP* definition and will therefore find the minimum weight Hamiltonian cycle. Lemma 3.1 states that the optimal tour of G_{TSP} will also be the minimum weight Hamiltonian cycle of G_{TSP}. This is important, as it allows us to use the *TSP*-solver to find the optimal tour of G_{TSP}.

Lemma 3.1. *There exists an optimal tour of G_{TSP} that visits every vertex exactly once. This implies that the optimal tour of G_{TSP} will also be the minimum weight Hamiltonian cycle of G_{TSP}.*

Proof. Assume for sake of contradiction that the optimal tour visits a vertex more than once. Call this vertex b. Then at some point in the tour, we will visit b for the second time. At this point, the tour will have been at a vertex a right before b and move to some other vertex c right after b. But the triangle inequality then tells us that we can construct a shorter or equal length tour, namely, the one which only traverses the edge (a, c) instead of $((a, b), (b, c))$. This is a contradiction to the tour being optimal. Note that the length of (a, c) and $((a, b), (b, c))$ may be equal, but this is okay, as there still exists an optimal tour that only visits b once.

□

It is also important that the cost of the optimal tour of G_{TSP} is the same as the cost of the optimal tour on G_{TSPM} if we are to use the *TSP*-solver to find the optimal tour of G_{TSPM}.

Lemma 3.2. *The cost of the optimal tour of G_{TSP}, the TSP-tour, is the same as the cost of the optimal tour of G_{TSPM}, the TSPM-tour.*

Proof. We can convert an arbitrary tour of G_{TSP} into a tour of G_{TSPM} with the same length and vice versa.

First, the conversion from the *TSP*-tour to the *TSPM*-tour: When computing an edge $e_{TSP} = (u, v)$ for G_{TSP} in the reduction, the all-pairs-shortest-path algorithm can save the shortest path used from u to v in G_{TSPM}. Then an edge (u, v) in the *TSP*-tour can be converted into the shortest path between u and v in the *TSPM*-tour.

Second, the other way: We note that the tour in G_{TSPM} will potentially contain the same vertices multiple times. The conversion from the *TSPM*-tour to the *TSP*-tour will require that we only included edges to vertices if we visit the vertex for the first time. That is, every time the *TSPM*-tour visits a vertex u for the first time, we add the edge $e_{TSP} = (u, v)$ to the *TSP*-tour. Here v is the vertex which at this point has most recently been visited for the first time.

We conclude that there is a bijective mapping from tours of G_{TSP} to tours of G_{TSPM} with equal length. This implies that the cost of the optimal tour in either graph is the same. □

With these two lemmas, we conclude that we can use a *TSP*-solver on G_{TSP} to find the optimal tour of G_{TSPM}.

4 Known results

We now return to the online graph exploration problem and give an short overview of known results in the literature. The results are upper bounds on the competitive ratio on different algorithms, and lower bounds on the competitive ratio for any algorithm.

Year	Reference	Algorithm	Competitive Ratio	Graph type
1977	*Rosenkrantz et al. [17]*	*Nearest Neighbor*	$\Theta(\log n)$	*General*
2012	Megow et al. [15]	Hierarchical DFS modification	$\Theta(\log n)$	General
2016	Foerster et al. [8]	A greedy algorithm	$n-1$	Strongly connected directed graphs
2012	Megow et al. [15]	Hierarchical DFS	$2k$	Graphs with k different edge weights
2012	Megow et al. [15]	Blocking algorithm	$16(2g+1)$	General graphs of genus at most g
1994	*Megow et al. [15]*	*Blocking algorithm*	16	*Planar*
1994	Kalyanasundaram and Pruhs [13]	ShortCut (predecessor to blocking)	16	Planar
2020	Fritsch [9]	Blocking parameter modification	$5/2+\sqrt{2} \approx 3.91$	Cactus graphs
2020	Fritsch [9]	Blocking parameter modification	3	Unicyclic graphs
2009	*Miyazaki et al.* [16]	*Standard Depth First Search (DFS)*	2	*Unweighted graphs*
2020	Sebastian Brandt et al. [3]	A greedy algorithm	2	Tadpole graph with non-negative weight
2010	Asahiro et al. [1]	Nearest Neighbor	$3/2 = 1.5$	Simple cycles
2009	Miyazaki et al. [16]	Weighted Distance Computations	$\frac{1+\sqrt{3}}{2} \approx 1.37$	Simple cycles

Table 1: Upper bound on competitive ratio for different algorithms sorted by competitive ratio. For each algorithm, the type of graph and type of edge weights on which the competitive ratio applies is listed.

Year	Reference	Competitive ratio	Graph type
2020	Birx et al. [2]	$\frac{10}{3} = 3.3\overline{3}$	General
2012	Dobrev et al. [6]	2.5	General
2009	*Miyazaki et al. [16]*	2	*Unweighted graphs*
2010	Asahiro et al. [1]	$\frac{5}{4} = 1.25$	Simple cycles
2009	Miyazaki et al. [16]	$\frac{1+\sqrt{3}}{2} - \varepsilon \approx 1.37 - \varepsilon$	Simple cycles

Table 2: Lower bounds sorted by competitive ratio. For each lower bound the type of graph and type of edge weights on which the competitive ratio applies is listed.

The competitive ratio of online graph exploration is in the words of [7] not very well understood. As we can see in the two tables above, there is a large gap between the lower bound of $3.3\overline{3}$ on general graphs and the upper bound of 16 on planar graphs obtained by $\texttt{Blocking}_\delta$. Furthermore, it is still an open question if a constant competitive ratio can be achieved on general graphs, that is non-planar graphs with an arbitrary number of weights [2, 15].

Part II
Algorithms and lower bounds

The rest of the report will investigate the results marked with *italic* in Table 1 and Table 2. The first and also the oldest result is regarding the nearest neighbour algorithm which we cover in Section 5. This is one of the simplest algorithms for online graph exploration and has a competitive ratio of $\Theta(\log n)$, where n is the number of vertices. We will however only show that $\Omega(\log n)$ is a lower bound. Next, we cover standard depth-first search which yields a competitive ratio of 2 on all unweighted graphs. Looking at Table 2 we see that the lower bound on the competitive ratio for unweighted graphs is also 2. So DFS achieve the best possible competitive ratio on these graphs. The DFS algorithm will be the focus of Section 6 along with this lower bound. The algorithm $\texttt{Blocking}_\delta$ is a "sophisticated generalisation of a Depth First Search"[15]. It has a competitive ratio of $16(2g + 1)$ on graphs with genus g and with arbitrary edge weights. For planar graphs, this is simply 16. We go in-depth with the $\texttt{Blocking}_\delta$ algorithm in Section 7.

5 Nearest neighbour

The first online graph exploration algorithm we will introduce is the nearest neighbour (NN) algorithm. NN is perhaps the obvious first approach to the problem and therefore works great as an introduction. Simple as it is the NN heuristic also has some problems. The example in Figure 3 shows one of these problems. We will use this to prove Lemma 5.2 which motivates the need for more sophisticated algorithms. The NN algorithm starts in the origin vertex o and chooses to explore the nearest unexplored vertex. It does this recursively until no unexplored vertices exist. We consider connected graphs, but not necessarily fully connected. Therefore NN might backtrack already visited vertices to explore new vertices. Let G be a partially explored graph and y the vertex of which the searcher is currently located. Then the algorithm is as follows [17].

Algorithm 1: NN(G, y)

```
if there exists an unexplored vertex in G then
    Let u be the unexplored vertex with the shortest known path from y
    Walk the shortest known path from y to u
    NN(G, u)
else
    Walk the shortest path from y to the origin vertex o
end
```

The algorithm is initialised on the origin vertex o and the partially explored graph G. At initialisation, G contains o and the adjacent vertices and incident edges of o. As the algorithm explores the graph, the newly explored vertices and edges are added to G. That is, we add y and its adjacent vertices and incident edges to G. The tour found by the NN algorithm is called the *NN tour*. Note that there can be multiple NN tours on a given graph, as NN resolve tiebreakers arbitrarily. Figure 3 showcases a complete run of the NN algorithm on a small graph.

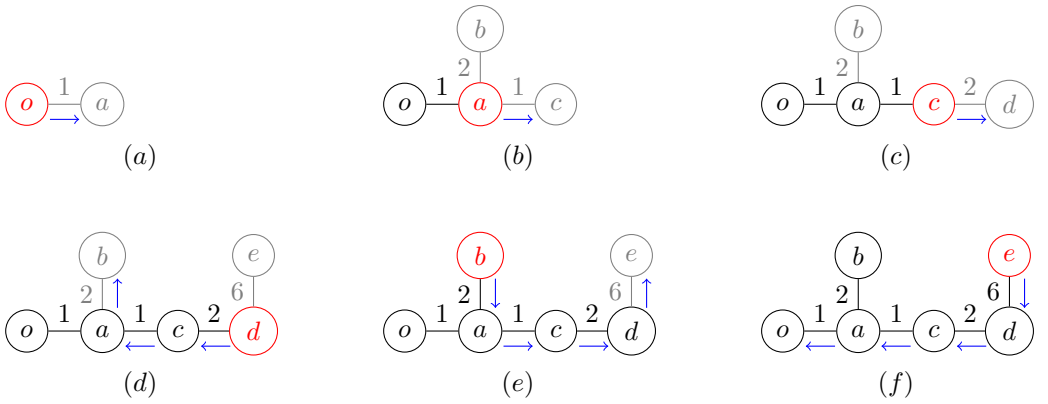


Figure 3:
A Complete run of the NN algorithm. Black vertices and edges are explored. Gray are unexplored. The red vertex marks where the searcher is currently located. The blue arrows indicate the path of the algorithm in the given iteration

An explanation of the steps (a)-(f) on Figure 3.

(a) NN is initialised on o. The only revealed unexplored vertex is a, making this a trivial choice for the algorithm. NN is recursively called $NN(G, a)$.

(b) Here NN has two choices to explore a new vertex. Either traverse (a, b) or (a, c). NN chooses (a, c) as c is the closest vertex to a.

(c) Again NN has two choices. Traverse (c, d) or walk the shortest path from c to b. This path has length 3 and $|(c, d)| = 2$, so NN traverses (c, d).

(d) The closest unexplored vertex is b, which NN, therefore, choose to explore. Note that the algorithm backtracks already explored vertices to reach b. This is where the NN algorithm makes a mistake compared to the optimal offline algorithm.

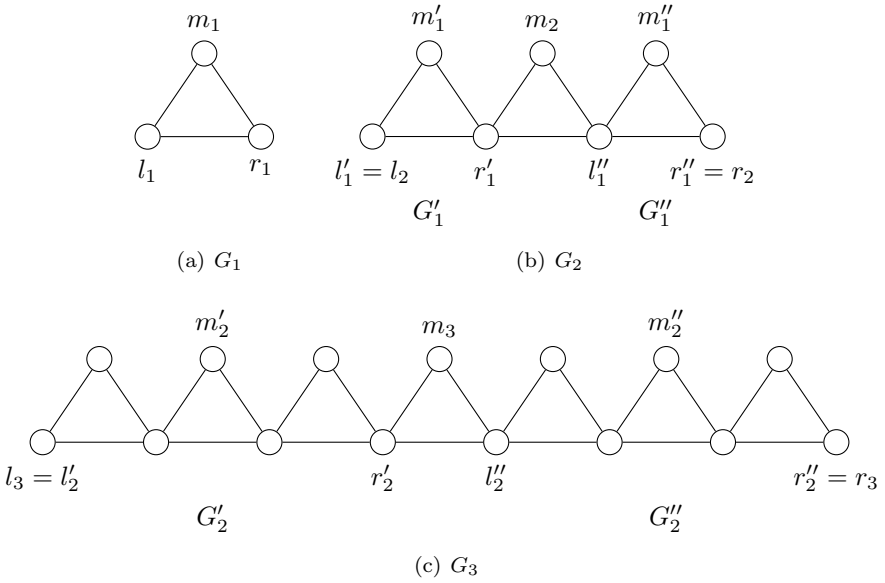
(e) Here the only unexplored vertex is e, making this a trivial choice.

(f) There are now no unexplored vertices in G so we consider G explored. At this point, NN returns to o using the shortest path and terminates.

The total cost of the NN tour is 30. The optimal offline tour is $o \to a \to b \to a \to c \to d \to e \to d \to c \to a \to o$ which has a cost of 24. Thus NN yields a competitive ratio of $\frac{NN}{Optimal} = \frac{30}{24} = 1.25$ on the graph on Figure 3. A competitive ratio of 1.25 is relatively low compared to the other algorithms we will discuss. However, note that NN was forced to backtrack the edges (a, c) and (c, d) multiple times. On this small example, this is not a problem, but we can use this observation to construct larger graphs that forces NN to backtrack much more. We will use one such construction to show the lower bound of $\Omega(\log n)$ on the competitive ratio of NN in Theorem 5.3.

The following lower bound construction and lemmas is based on Hurkens et al. [12]. The graph will consist of several triangles joined together as seen in Figure 4. The graph will only contain unweighted edges. We recursively define the graph to enable an induction proof. To get started let G_1 be the graph consisting of a single triangle, as seen in Figure 4(a). We name the leftmost node l_1, the rightmost node r_1 and the middle node m_1. Now for $k \geq 2$ we let G_k be the graph obtained by placing two copies of G_{k-1} named G'_{k-1} and G''_{k-1} besides each other with a new vertex m_k between them. A new triangle is formed between the two copies. More precisely, three new edges are created, as seen on Figure 4. These edges

connects r'_{k-1}, the rightmost vertex of G'_{k-1}, and l''_{k-1}, the leftmost vertex of G''_{k-1}, and the newly added m_k. Again we name the leftmost node of G_k for l_k, the rightmost node r_k and the middle node m_k. In Figure 4 the first three recursively defined graphs G_1, G_2 and G_3 can be seen.



(a) G_1 (b) G_2

(c) G_3

Figure 4: The first three recursively defined graphs G_1, G_2 and G_3.

Lemma 5.1. *For $k \geq 1$, the distance from m_k to l_k and r_k is 2^{k-1}.*

Proof. For $k = 1$, this is trivial, as $2^{1-1} = 1$. For $k \geq 2$, we use the recursive nature of G_k. There is 2^k vertices in the lower level of G_k, as the number of vertices double every time. Here, the lower level refers to the vertices in the lower layer in the embedding shown on Figure 4. In G_1, the lower level is vertices l_1 and r_1. Let G'_{k-1} and G''_{k-1} be the two copies of which G_k is constructed and m_k the connecting vertex. The shortest path from $l'_{k-1} = l_k$ to r'_{k-1} is the path consisting of all the vertices in the lower level of G'_{k-1}. This path contains $2^{k-1} = \frac{2^k}{2}$ vertices. Similarly the path l''_{k-1} to $r''_{k-1} = r_k$ contains 2^{k-1} vertices. Both of these paths contain $2^{k-1} - 1$ edges. One additional edge is needed to complete the path from $l'_{k-1} = l_k$ to m_k, namely the edge (r'_{k-1}, m_k). Similarly the path from m_k to $r''_{k-1} = r_k$, need the edge (l''_{k-1}, m_k). This yields a total length of $2^{k-1} - 1 + 1 = 2^{k-1}$ for the paths from l_k to m_k and m_k to r_k.

□

Lemma 5.2. *For $k \geq 1$, consider a graph G with G_k as an induced subgraph, where only the endpoints l_k and r_k have edges incident to $G - G_k$. There exists a NN tour in G that includes a path starting in l_k and ending in m_k, where this path includes all vertices in G_k and has length $(k+3)2^{k-1} - 2$.*

Proof. This proof is by induction in k.

Base case: Let $k = 1$. The tour $l_1 \to r_1 \to m_1$ is a NN tour visiting all vertices in G_1. It has length $2 = (1+3)2^{1-1} - 2 = (k+3)2^{k-1} - 2$. Thus the statement holds for the base case.

Induction hypothesis: The statement holds for $k-1$. That is, there exists a NN tour in G that includes a path starting in l_{k-1} and ending in m_{k-1}, where this path includes all vertices in G_{k-1} and has length $(k+2)2^{k-2} - 2$.

Induction case: Let $k \geq 2$. By Lemma 5.1 the distance from m'_{k-1} to r'_{k-1} and l'_{k-1} is 2^{k-2}. By the induction hypothesis NN visits all vertices in G'_{k-1} ending in m'_{k-1}. The distance from m'_{k-1} to an unvisited vertex is therefore at least $2^{k-2} + 1$. There is one additional edge from r'_{k-1} to l''_{k-1}. Thus the distance from m'_{k-1} to l''_{k-1} is $2^{k-2} + 1$. This means that NN may walk from m'_{k-1} to l''_{k-1}. From here NN may by the inductive hypothesis visit all vertices in G''_{k-1} ending in m''_{k-1}. Now the last unvisited vertex is m_k. The distance from m''_{k-1} to m_k is $2^{k-2} + 1$.

We conclude that there exists a NN tour that includes a path starting in l_k and ending in m_k with length:

$$\underbrace{(k+2)2^{k-2}-2}_{G'_{k-1}}+\underbrace{(k+2)2^{k-2}-2}_{G''_{k-1}}+\underbrace{2^{k-2}+1}_{m'_{k-1}\text{ to }l''_{k-1}}+\underbrace{2^{k-2}+1}_{m''_{k-1}\text{ to }m_k}=(k+3)2^{k-1}-2$$

□

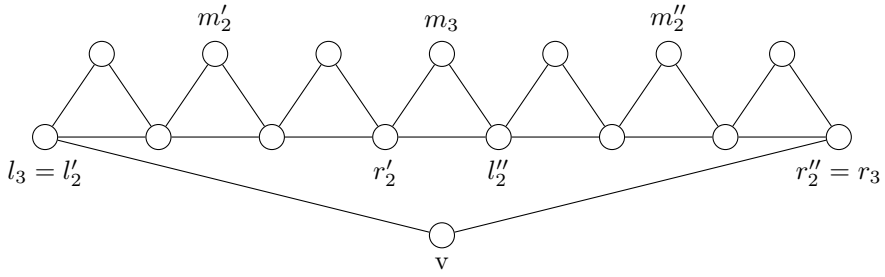


Figure 5: G_3 with vertex v added. G_3 now contains a Hamiltonian cycle.

Theorem 5.3. *For $k \geq 1$, there exists a graph G with $n = 2^{k+1}$ vertices where the optimal offline tour has length 2^{k+1} but the NN tour has length $(k+4)2^{k-1}$. Resulting in a competitive ratio of $\frac{1}{4}(3+\log n)$.*

Proof. Consider a graph G_k as previously described. Add a new vertex v to G_k, with v adjacent only to l_k and r_k. See as an example Figure 5 where v is added to G_3. This new graph G now contains a Hamiltonian cycle. We argued in Lemma 5.1 that the lower level contains 2^k vertices. Then clearly the upper level contains $2^k - 1$ vertices. With the additional vertex v, this yields a total of $2^k + 2^k - 1 + 1 = 2^{k+1}$ vertices. It is then easy to see that the optimal tour of G has length $Optimal = 2^{k+1}$.

Consider the NN tour where l_k is the origin vertex. Lemma 5.2 then states that there exist a NN tour containing a path that starts in l_k and ends in m_k with length $(k+3)2^{k-1} - 2$ that visits every vertex in G except v. To complete the NN tour, NN must now visit v and return to l_k. By similar reasoning to that in Lemma 5.2, the distance from m_k to v is $2^{k-1}+1$ and the distance from v to l_k is 1. In total the NN tour on G has length

$$NN = (k+3)2^{k-1} - 2 + 2^{k-1} + 1 + 1 = (k+4)2^{k-1}$$

As $n = 2^{k+1}$, this yields a competitive ratio of:

$$\begin{aligned}
\frac{NN}{Optimal} &= \frac{(k+4)2^{k-1}}{2^{k+1}} \\
&= \frac{k \cdot 2^{k-1}}{2^{k+1}} + \frac{2^{k+1}}{2^{k+1}} \\
&= \frac{k \cdot 2^{k-1}}{4 \cdot 2^{k-1}} + 1 \\
&= \frac{k}{4} + 1 \\
&= \frac{1}{4}(4+k) \\
&= \frac{1}{4}\left(3 + \log_2\left(2^{k+1}\right)\right) \\
&= \frac{1}{4}(3 + \log_2 n)
\end{aligned}$$

□

With Theorem 5.3 we can now capture the desired lower bound for NN in the below corollary.

Corollary 5.3.1. *The competitive ratio of NN is lower bounded by* $\Omega(\log n)$.

Overall we conclude that the NN algorithm is an intuitive first attempt at solving the online graph exploration problem. However, the lower bound of $\Omega(\log n)$ motivates the study of better algorithms. We will therefore move on and look at the DFS algorithm next.

6 Depth-first search

The depth-first search (DFS) algorithm can be seen as an online graph exploration algorithm. Starting in the origin vertex, DFS will traverse an arbitrary edge. It then makes a recursive call on the newly explored vertex. It then explores vertices recursively in this manner until it reaches a vertex with no unexplored adjacent vertices. At this point, it backtracks its path to the most recent vertex with unexplored edges and continues until the whole graph has been explored. The pseudocode is as follows. This is an adoption from [5, Section 22.3].

Algorithm 2: DFS(G, y)

```
1 while there exists an unexplored vertex u that is adjacent to y do
2 |   traverse the edge (y, u)
3 |   DFS(G, u)
4 |   backtrack the edge (y, u)
5 end
```

The initial call is DFS(G, o), where G is a partially explored graph in which only o is explored. Upon the call $DFS(G, y)$, we consider y to be explored. This is similar to the nearest neighbour from Section 5.

As for any online algorithm we are interested in the competitive ratio of DFS. To find this we first need an upper bound on the tour found by DFS. On unweighted graphs, however, there is no upper bound. This is because DFS may choose to include an edge of arbitrarily large weight, that is not included in the optimal tour as Figure 6 shows. In Figure 6(a) the searcher is located at vertex a. Here the DFS algorithm has two choices for the next step of the exploration. Importantly DFS does not base this decision on the weight of the edges. We can assume without lose of generality that edge (a, b) is traversed as in (b) of Figure 6, where a new edge (b, c) is now revealed. Now if the weight $|(a, b)|$ is much much higher than both $|(a, c)|$ and $|(b, c)|$, DFS has a problem. The optimal tour would then be $a \to c \to b \to c \to a$, but DFS choose $a \to b \to c \to b \to a$. The choice of $|(a, b)|$ would then essentially decide the competitive ratio, as the DFS tour contains it, but the optimal does not. Thus DFS yields an arbitrarily high competitive ratio for weighted graphs. This is an undesirable property of an online algorithm.

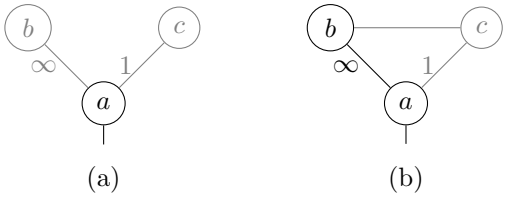


Figure 6: Subgraph of a potentially larger graph. Grey vertices and edges are unexplored.

However, given an unweighted graph, the exploration problem can be solved in a simple manner using DFS. In fact, we will now show that there cannot exist a better online algorithm than DFS on unweighted graphs.

6.1 DFS proofs

In Theorem 6.1 we argue that DFS has a competitive ratio of 2. This is complemented by Theorem 6.2 where we argue that there cannot exist an online graph exploration algorithm on unweighted graphs with a competitive ratio $2-\varepsilon$ for $\varepsilon > 0$. The section is based on Miyazaki et al. [16].

Theorem 6.1. *DFS is 2-competitive on unweighted graphs.*

Proof. Let $n > 0$ be the number of vertices in a graph G. DFS will traverse each of the $n-1$ edges in the DFS tree exactly twice. Note that the DFS forest will consist of one tree as we are only considering connected graphs in the online exploration problem. One time when exploring the new vertex and once more when backtracking. In total, the cost of DFS is exactly $2(n-1)$. The optimal offline solution must traverse at least one new edge to visit each new vertex, and at least one edge to return to the origin vertex. Therefore a lower bound on the cost of the optimal tour is n. The cost is exactly n if and only if the graph contains a Hamiltonian cycle. This yields a competitive ratio for DFS of

$$\frac{DFS}{Optimal} \leq \frac{2(n-1)}{n} = 2 - \frac{2}{n} < 2$$

Using Definition 1.3 this concludes that DFS is 2-competitive online algorithm on unweighted graphs. [16] □

The inequality $\frac{DFS}{Optimal} \leq 2 - \frac{2}{n}$ above might seem to contradict the next theorem stating that no online algorithm with competitive ratio $2-\varepsilon$ can exist for any positive ε. However this is not a contradiction, as for a given positive ε we can choose n large enough such that $2-\varepsilon < 2 - \frac{2}{n}$. The proof of this next theorem is adopted from Theorem 5 in Miyazaki et al. [16].

Theorem 6.2. *For unweighted graphs, there exist no online algorithm with a competitive ratio of* $2-\varepsilon$ *for any positive* ε*.*

Proof. We will consider an adversary who can build the graph as the searcher explores it. When the searcher enters a new vertex the adversary will have to decide what vertices are adjacent. This way the adversary can construct a graph based on the choices of the searcher, in a way that forces a worst case, no matter the choice of the algorithm. This line of reasoning holds because the searcher does not have any knowledge of the unexplored part of the graph. Because we want to consider the worst-case it makes sense to let an adversary choose it.

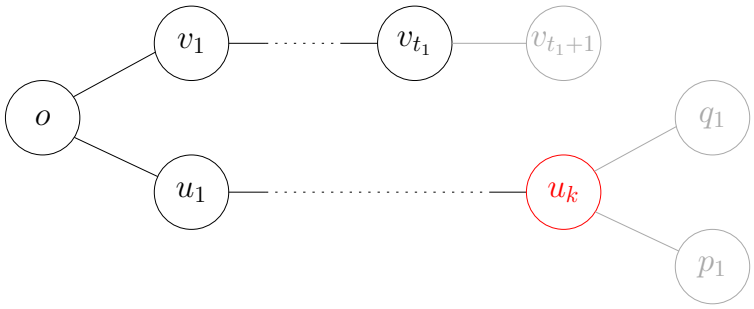


Figure 7: The graph constructed by the adversary. The red vertex denotes where the searcher is currently located and grey vertices and edges are unexplored

The overall strategy of the adversary is to force the searcher to go down the same path multiple times. The adversary can do this by first creating two paths u and v from the origin vertex o. The path u is defined as $u_1 \to u_2 \to ...$ and similar for v. At first only edges (o, v_1) and (o, u_1) are revealed. The algorithm will then explore the paths. When the searcher moves to a vertex, the adversary will reveal outgoing edges from the vertex and labels of adjacent vertices. The algorithm can always backtrack to explore the other path.

Let $k > \frac{3}{\varepsilon}$ be a fixed integer. At some point the searcher will reach the k'th vertex in a path, either u_k or v_k as on Figure 7. We can assume, without loss of generality that it is u_k and let v_{t_1} denote the last

explored vertex of v. The cost of edges travelled so far D_0 is:

$$D_0 \geq 2 \cdot |(o, v_1, \ldots, v_{t_1})| + |(o, u_1, \ldots, u_k)| = 2t_1 + k$$

Here, $|(o, v_1, \ldots, v_{t_1})|$ denotes the combined weight of edges in the path.

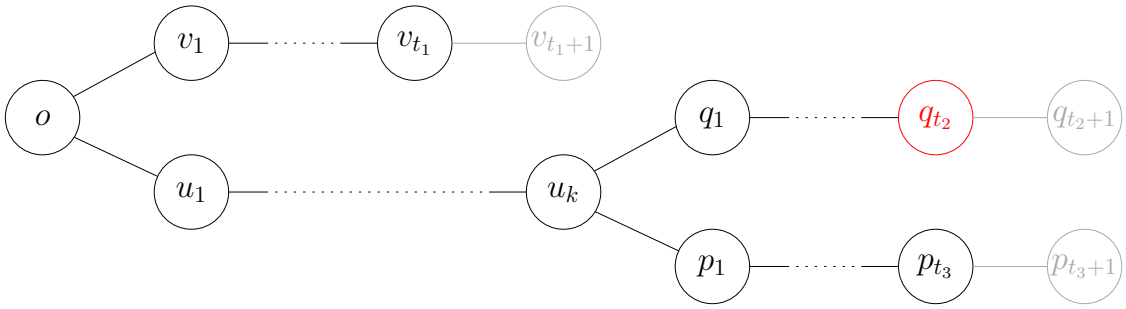


Figure 8: Two new paths p and q are revealed

The adversary will now reveal two new paths q and p starting from u_k one edge at a time as on Figure 8. The algorithm can explore thees paths in a similar way to before. We denote the last explored vertex in each path q_{t_2} and p_{t_3} respectively, the vertices q_{t_2+1} and p_{t_3+1} are thus unexplored. Note t_1 is fixed while t_2 and t_3 are still variables. At the point where the searcher has not yet explored q_{k+t_1} or p_{k+t_1}, the $(k+t_1)$'th vertex of either q or p, that is $t_2, t_3 < k + t_1$, we will consider two cases based on the choice of the searcher.

Case 1: The searcher reaches v_{t_1+1} before q_{k+t_1} or p_{k+t_1} as on Figure 9.

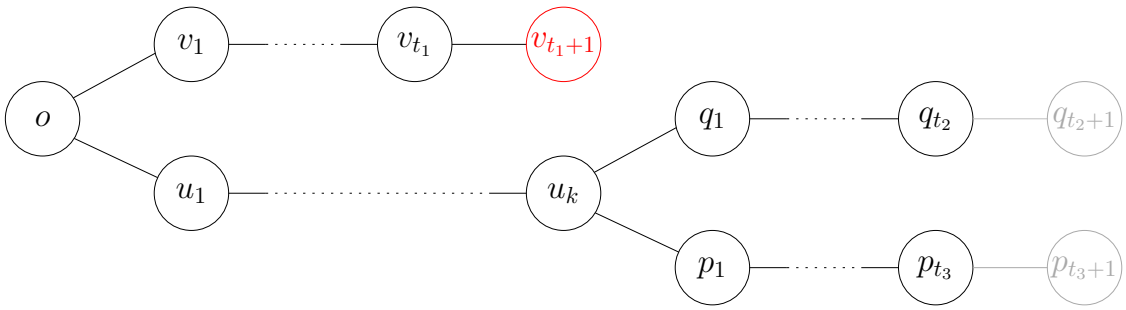


Figure 9: Case 1.

At this point the total travel cost D_1 is

$$\begin{aligned} D_1 &\geq D_0 + 2 \cdot |(u_k, p_1, ..., p_{t_3})| + 2 \cdot |(u_k, q_1, ..., q_{t_2})| + |(u_k, u_{k-1}, ..., o, v_1, ..., v_{t_1+1})| \\ &= D_0 + 2 \cdot t_3 + 2 \cdot t_2 + k + t_1 + 1 \,. \end{aligned}$$

The adversary then chooses not to reveal any further edges in the graph. The searcher then has to backtrack to reach both q_{t_2+1} and p_{t_3+1} before returning to the origin vertex o and terminating. This yields a total cost for the algorithm in case 1 of

$$\begin{aligned} \mathit{Algorithm} &\geq D_1 + |(v_{t_1+1}, v_{t_1}, ..., o, u_1, ..., u_k)| + 2 \cdot |(u_k, q_1, ..., q_{t_2+1})| + \\ &\quad 2 \cdot |(u_k, p_1, ..., p_{t_3+1})| + |(u_k, u_{k-1}, ..., o)| \\ &= D_1 + t_1 + 1 + k + 2 \cdot (t_2 + 1) + 2 \cdot (t_3 + 1) + k \\ &= 4 \cdot (k + t_1 + t_2 + t_3) + 6 \,. \end{aligned}$$

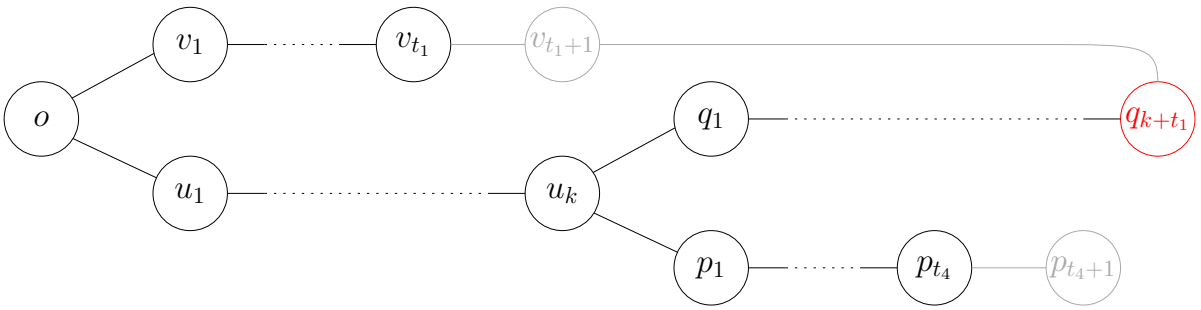


Figure 10: Case 2.

The optimal tour for the constructed graph is when the searcher only visits each branch once. One such optimal tour could be $(o, ..., v_{t_1+1}, ..., o, ..., u_k, ...q_{t_2+1}, ..., u_k, ...p_{t_3+1}, ..., u_k, ..., o)$ with the cost being $Optimal = 2 \cdot (k + t_1 + t_2 + t_3) + 6$. The competitive ratio then becomes

$$\begin{aligned}
\frac{Algorithm}{Optimal} &\geq \frac{4 \cdot (k + t_1 + t_2 + t_3) + 6}{2 \cdot (k + t_1 + t_2 + t_3) + 6} \\
&= \frac{4 \cdot (k + t_1 + t_2 + t_3) + 12}{2 \cdot (k + t_1 + t_2 + t_3) + 6} - \frac{6}{2 \cdot (k + t_1 + t_2 + t_3) + 6} \\
&= 2 - \frac{6}{2 \cdot (k + t_1 + t_2 + t_3) + 6} \\
&\geq 2 - \frac{6}{2 \cdot k} \\
&= 2 - \frac{3}{k} \\
&> 2 - \varepsilon
\end{aligned}$$

The first of the two inequalities hold because $t_1, t_2, t_3 \geq 0$, as they denote the number of vertices traversed by the searcher in the respective paths. The second because we choose $k > \frac{3}{\varepsilon}$, which implies that $\varepsilon > \frac{3}{k}$.

Case 2: The searcher reaches q_{k+t_1} or p_{k+t_1} before v_{t_1+1} as on Figure 10.

We can assume that the searcher reaches q_{k+t_1} before p_{k+t_1} without loss of generality. At this point the total travel cost D_2 is

$$D_2 \geq D_0 + 2 \cdot |(u_k, p_1, ..., p_{t_4})| + |(u_k, q_1, ..., q_{k+t_1})| = D_0 + 2 \cdot t_4 + k + t_1$$

Now the adversary reveals one last edge (q_{k+t_1}, v_{t_1+1}). The optimal strategy for the searcher is then to traverse (q_{k+t_1}, v_{t_1+1}). Now both paths to the last unexplored vertex p_{t_4+1}, namely $(v_{t_1+1}, v_{t_1}, ..., o, u_1, ..., u_k, ..., p_{t_4+1})$ and $(v_{t_1+1}, q_{k+t_1}, q_{k+t_1-1}, ..., u_k, ..., p_{t_4+1})$ are equally long, so the explores choice does not matter. Lastly the searcher must return the the origin vertex o. This yields a total cost for the algorithm in case 2 of

$$\begin{aligned}
Algorithm &\geq D_2 + |(q_{k+t_1}, v_{t_1+1})| + |(v_{t_1+1}, q_{k+t_1}, ..., u_k, p_1, ..., p_1, p_{t_4+1})| + |(p_{t_4+1}, p_{t_4}, ..., u_k, u_{k-1}, ..., o)| \\
&= D_2 + 1 + 1 + k + t_1 + t_4 + 1 + 1 + t_4 + k \\
&= 4 \cdot (k + t_1 + t_4 + 1)
\end{aligned}$$

An optimal tour for this constructed graph is $o, v_1, ..., v_{t_1+1}, q_{k+t_1}, q_{k+t_1-1}, ..., u_k, p_1, ..., p_{t_4+1}, p_{t_4}, ..., u_k, u_{k-1}, ..., o$

which results in a total length of $Optimal = 2 \cdot (k + t_1 + t_4 + 2)$. The competitive ratio then becomes:

$$\begin{aligned}
\frac{Algorithm}{Optimal} &\geq \frac{4 \cdot (k + t_1 + t_4) + 4}{2 \cdot (k + t_1 + t_4) + 4} \\
&= \frac{4 \cdot (k + t_1 + t_4) + 8}{2 \cdot (k + t_1 + t_4) + 4} - \frac{4}{2 \cdot (k + t_1 + t_4) + 4} \\
&= 2 - \frac{4}{2 \cdot (k + t_1 + t_4) + 4} \\
&\geq 2 - \frac{4}{2 \cdot k} \\
&= 2 - \frac{2}{k} \\
&> 2 - \varepsilon
\end{aligned}$$

which is true as we choose $k > \frac{3}{\varepsilon}$ implying $\frac{2}{k} < \frac{3}{k} < \varepsilon$.

In conclusion, $2 - \varepsilon$ is a lower bound for any algorithm, as we do not assume anything about the choices of the algorithm. [16] □

Overall we conclude using Theorem 6.1 and 6.2 that DFS is a best online algorithm on unweighted graphs. This result renders any further analysis of this setting rather redundant. We will therefore consider an online exploration algorithm for weighted graphs next.

7 Blocking$_\delta$

Let us now consider the online exploration algorithm called Blocking$_\delta$. The following section will be based on the article by Megow et al. [15]. Blocking$_\delta$ is a corrected version of the ShortCut algorithm proposed by Kalyanasundaram and Pruhs [13]. Blocking$_\delta$ obtains a constant competitive ratio of 16 on weighted planar graphs. This is currently the best-known result on this graph class. Blocking$_\delta$ is based on DFS in the sense that it explores the graph depth-first but introduces a *blocking* condition to edges. Intuitively an edge is blocked if it is either too heavy or too far away from the searcher compared to other edges. If an edge is blocked, then that edge will not be traversed before it is no longer blocked. Blocking$_\delta$ can be used on undirected graphs with arbitrary edge weights, however, on unweighted graphs, DFS and Blocking$_\delta$ behave the same, as no edge will ever be blocked.

The δ-parameter

Blocking$_\delta$ is a parameterised algorithm, with parameter δ which is a constant chosen before running the algorithm and remains fixed throughout. Intuitively, the δ-parameter regulates when an edge is too far away to be traversed. The effect of δ on the behaviour of Blocking$_\delta$ is described further in Section 7.3.

The article by Megow et al. [15] states that $\delta > 0$. The value of δ does however make sense for $\delta > -1$, but for $\delta \leq -1$ the Blocking$_\delta$ algorithm will behave just like DFS. This extended range of δ has been considered in newer research [9].

The δ parameter is important for the proof of Corollary 7.6.1 stating that Blocking$_\delta$ is 16-competitive on planar graphs. But as the experiments in Section 9 show, the δ-parameter does not seem to have a significant influence on the competitive ratio of Blocking$_\delta$ on the small random planar graphs tested.

Overview of section

The rest of this section is structured as follows. We will first introduce some definitions and the algorithm in Section 7.1. Thereafter, in Section 7.2 we will run it on two examples to see how it works. In Section 7.3 we discuss how the behaviour of Blocking$_\delta$ can be close to DFS for small δ, and more like Prims' *MST* algorithm for large δ. The main part of this section is an involved proof showing Theorem 7.6 stating

that $\texttt{Blocking}_\delta$ has a competitive ratio of $2(2+\delta)\left(\frac{2}{\delta}+1\right)$ for planar graphs. Lastly, we discuss ideas for improving $\texttt{Blocking}_\delta$ and a lower bound. For the lower bound we construct a graph of which $\texttt{Blocking}_\delta$ yields a competitive ratio of 5.25.

7.1 $\texttt{Blocking}_\delta$ definitions

Recall Definition 2.1 stating that a vertex u is explored when it has been visited by the searcher and that an edge $e = (u, v)$ is explored when both u and v have been explored.

Definition 7.1. *Boundary edge*
A boundary edge $e = (u, v)$ is an edge where u has been explored but v is unexplored.

Definition 7.2. *Close* and *far from*
We say that an edge $e = (u, v)$ is *close* to another edge $e' = (u', v')$ if the length of the shortest known path from u to v' is at most $(1 + \delta) \cdot |e|$, otherwise we say that e is *far from* e'.

Note that the *close* and *far from* relations on edges is not reflexive, as it depends on the length of the first edge.

Definition 7.3. *Blocked*
A boundary edge $e = (u, v)$ is *blocked* by e', if there exists another boundary edge $e' = (u', v')$ anywhere in the graph, which is shorter than e ($|e'| < |e|$) and e is *close* to e'. If no such e' exists we say e is *unblocked*.

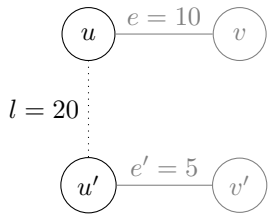


Figure 11: Boundary edge e is blocked by boundary edge e' as $\delta = 2$. The dotted line denotes that the arbitrarily long path from u to u' has length $l = 20$.

Intuitively, Definitions 7.3 and 7.2 states that light boundary edges should be preferred over heavy boundary edges, unless the light edge is too far away from the searcher. Note that it could potentially be more expensive to explore v' than v, but $\texttt{Blocking}_\delta$ prefers this.

On Figure 11, we see that boundary e is blocked by boundary e', as $|e'| < |e|$ and the shortest known path from u to v' is $l + e' = 25$ which is less than or equal to $(1 + \delta) \cdot |e| = (1 + 2) \cdot 10 = 30$, when $\delta = 2$.

Blocked edges are computed at every iteration and do not assume anything of where the searcher is currently located. This implies that if the searcher is located at u, it would still, perhaps surprisingly, choose to explore v' before v.

We are now ready to introduce the $\texttt{Blocking}_\delta$ algorithm. Besides δ, $\texttt{Blocking}_\delta$ also takes the partially explored graph G and the vertex y of which the searcher is located.

Algorithm 3: $\texttt{Blocking}_\delta(G, y)$ [15]

```
while there exists an unblocked boundary edge e = (u, v), such that either u = y or e had
  previously been blocked by an edge (u', y) do
    walk the shortest know path from y to u
    traverse boundary edge e = (u, v)
    Blocking_δ (G, v)
    walk the shortest know path from v to y
end
```

7.2 `Blocking`$_\delta$ examples

Example 7.1. Let us now run `Blocking`$_\delta$ on a concrete graph. For this example we choose $\delta = \frac{1}{10}$.

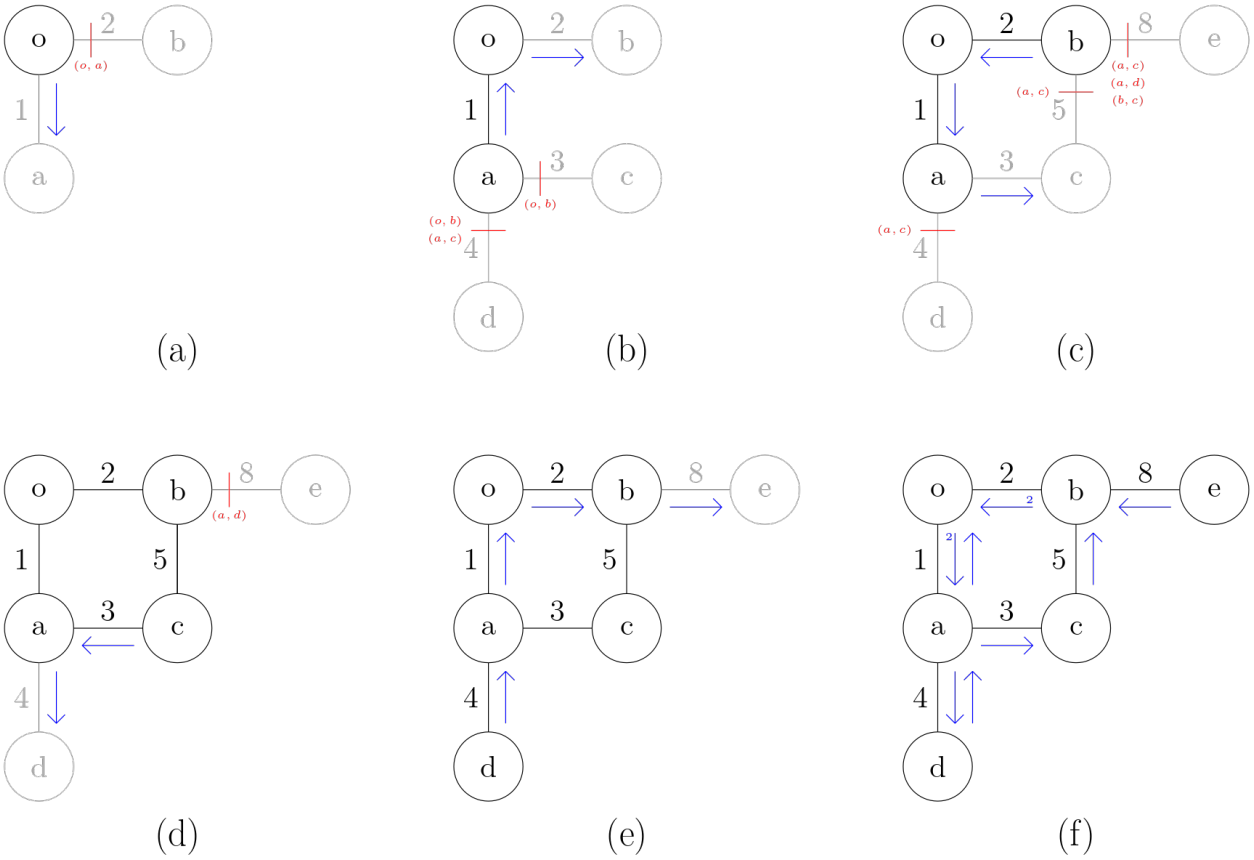


Figure 12: The `Blocking`$_\delta$ algorithm run with $\delta = \frac{1}{10}$. Grey vertices are unexplored but their labels are known to the searcher. Grey edges are boundary edges. The blue arrows marks the route taken by the searcher in the given iteration. The red lines indicates that an edge is blocked by an other edge.

We will describe each of the steps (a)-(f) of Figure 12 in detail to gain some intuition for how `Blocking`$_\delta$ works.

(a) When `Blocking`$_\delta$ is initiated on the origin vertex o, the edge (o, b) is blocked by the edge (o, a) since $|(o, b)| = 1 < 2 = |(o, a)|$ and since the shortest know path from o to b is 1 which is shorter than $(1+\delta) \cdot |(o, a)| = 2.2$. In line 3 of `Blocking`$_\delta$, (o, a) is traversed and `Blocking`$_\delta$ is called recursively on a.

(b) In this recursive call to `Blocking`$_\delta(G, a)$, the edge (o, b) is the only unblocked boundary edge. This edge was blocked by (o, a) in the previous call. The second condition of the while loop, *e had previously been blocked by an edge* (u', y), is therefore true, as $y = a$. The searcher will go back and explore the edge (o, b), and then make another recursive call `Blocking`$_\delta(G, b)$.

(c) At this point `Blocking`$_\delta$ has a perhaps surprising behaviour. The edge (b, c) is blocked by (a, c). Using Definition 7.2 and 7.3 we see that for (a, c) to block (b, c) the shortest known path from b to c is at most $(1+\delta) \cdot |(b, c)| = (1+\frac{1}{10}) \cdot 5 = 5.5$. One might think that the shortest know path is $b \to o \to a \to c$ with length 6, which is longer then 5.5. However the shortest know path is actually $b \to c$. This is somewhat counter-intuitive, since (b, c) is in some sense causing it self to be blocked. This behaviour only occurs when the two boundary edges ends in the same unexplored vertex. Additionally (a, c) was previously blocked by (o, b),

which again satisfies the second condition of the while loop. Thus yet another recursive call $\texttt{Blocking}_\delta(G, c)$ is made.

(d) The only unblocked boundary edge is (a, d). This edge was again previously blocked by an edge with an incident vertex equal to the searcher's current location, namely (a, c). Another recursive call $\texttt{Blocking}_\delta(G, d)$ is therefore made.

(e) The last unexplored vertex is e which will now be explored. The same reasoning as in (d) applies.

(f) Lastly, no more vertices are to be explored, so $\texttt{Blocking}_\delta$ returns from the recursive calls. Line 5 of $\texttt{Blocking}_\delta$ then states that we must revisit all the vertices corresponding to calls on the recursion stack. In doing so we use the shortest path, which may be shorter than the path taken at the time when the recursive call was made. An example of this is backtracking from b to c, here we have a path with length 5 which is shorter than the path described in part (c) of length 6. This backtracking is needed, as otherwise, $\texttt{Blocking}_\delta$ can get stuck. The next example covers this case.

At this point, $\texttt{Blocking}_\delta$ terminates. The final $\texttt{Blocking}_\delta$ tour is:

$$o \to a \to o \to b \to o \to a \to c \to a \to d \to a \to o \to b \to e \to b \to o \to a \to d \to a \to c \to b \to o \to a \to o$$

The total cost of the $\texttt{Blocking}_\delta$ tour is $1+1+2+2+1+3+3+4+4+1+2+8+8+2+1+4+4+3+5+2+1+1 = 63$. Note that at o we have explored the graph and returned to the start vertex, but $\texttt{Blocking}_\delta$ continues with backtracking the whole path. This essentially adds an unnecessary cost of 21. We will explore the effect of going directly back to the origin after exploring all vertices in Section 7.5.

The optimal offline tour is to go via the cycle in the graph

$$o \to a \to d \to a \to c \to b \to e \to b \to o$$

The total cost of the optimal offline tour is $1+4+4+3+5+8+8+2 = 35$. $\texttt{Blocking}_\delta$ yields a competitive ratio on this specific graph of $\frac{\texttt{Blocking}_\delta}{Optimal} = \frac{63}{35} = 1.8$. We will in Section 7.6 explore how to construct graphs that force higher competitive ratios than the on of Figure 12.

Example 7.2. We will use this second example to fully capture the behavioural aspects of $\texttt{Blocking}_\delta$. For this, the choice of δ will not matter, as every edge has the same weight. Definition 7.3 then states that no edges can ever be blocked, as no pair of edges e, e' satisfies that $|e| < |e'|$.

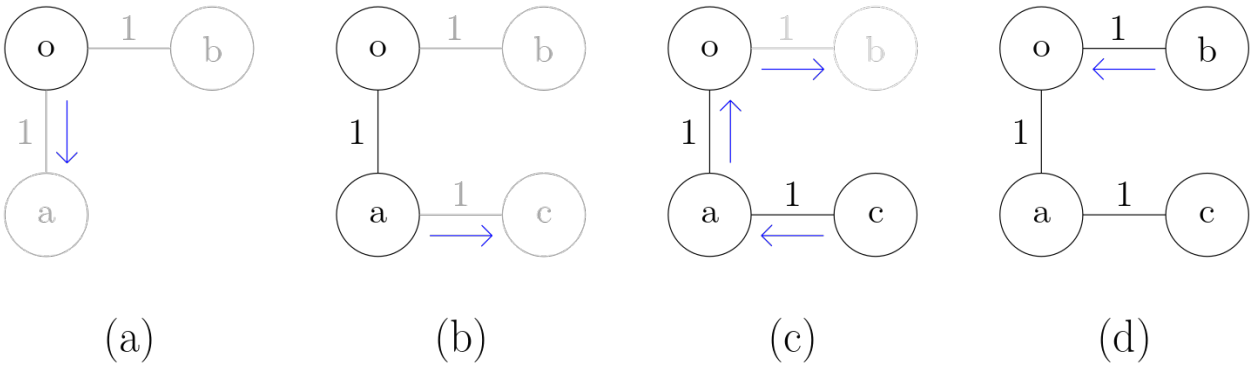


Figure 13: The $\texttt{Blocking}_\delta$ algorithm run with $\delta = 2$. Grey vertices are unexplored but their labels are known to the searcher. Grey edges are boundary edges. The blue arrows marks the route taken by the searcher in the given iteration. No edges are blocked in this example.

We will similarly to the previous example describe each of the steps (a)-(d) of Figure 13.

(a) The algorithm is initialized with $\texttt{Blocking}_\delta(G, o)$. At this point both edges (o, a) and (o, b) are candidates for exploration. Neither is blocked and both satisfy the first condition of the while loop, *there exists an unblocked boundary edge* $e = (u, v)$*, such that* $u = y$, because $y = o$. $\texttt{Blocking}_\delta$ makes the arbitrary choice of traversing (o, a) and recursively call $\texttt{Blocking}_\delta(G, a)$.

(b) Now both (a, c) and (o, b) are unblocked boundary edges. The edge (o, b) does not satisfy either of the while conditions when the searcher is located at vertex a. The edge (o, b) is not incident to a, making the first while condition false and (o, b) has not been blocked at any point, making the second condition false as well. However (a, c) satisfies the first condition and $\texttt{Blocking}_\delta$ traverses the edge and continues with $\texttt{Blocking}_\delta(G, c)$.

(c) At this point, only the boundary edge (o, b) remains. The searcher is currently located at vertex c, so using the argumentation from part (b), (o, b) does not satisfy the while condition. $\texttt{Blocking}_\delta$ will therefore return from the recursive call and backtrack the shortest know path from c to a. Now located at a, once again the same argument from part (b) applies for (o, b). $\texttt{Blocking}_\delta$ returns from the recursive call and backtracks the shortest known path from a to o. We are now at the initial call to $\texttt{Blocking}_\delta(G, o)$ with (o, b) satisfying the first condition of the while loop. Thus (o, b) is traversed.

(d) No more vertices are to be explored so $\texttt{Blocking}_\delta(G, b)$ returns from the recursive call and backtracks b to o. In example 7.1 $\texttt{Blocking}_\delta$ backtracked the whole path. This is not the case in this example, as $\texttt{Blocking}_\delta$ already backtracked from both vertices c and a in part (c). At this point $\texttt{Blocking}_\delta$ terminates.

Definition 7.4. *The set P*
We define the set P to be the set of all boundary edges e that are traversed in line 3 of the $\texttt{Blocking}_\delta$ algorithm.

Traversing a boundary edge will by definition reveal a new vertex. This is the only way of discovering a new vertex, thus the number of edges in P is the same as the number of explored vertices in G, excluding the origin vertex. In Figure 12 all edges except (a, c) are in P. Note that P forms a spanning tree.

7.3 The behaviour of $\texttt{Blocking}_\delta$

The behaviour of $\texttt{Blocking}_\delta$ depends on the value of δ. When δ is large enough the set P (the set of edges traversed in line 3 of $\texttt{Blocking}_\delta$) will be equal to a minimum spanning tree. Here large enough means $\delta \geq \frac{|E|-|e|}{|e|}$, where e is any (possibly the smallest) edge in the graph and $|E|$ is the total length of edges in the graph. On the other hand, when δ is small enough the edges in P will be the same as a depth-first tree of the graph. Here small enough means $\delta \leq -1$.

If $\delta \geq \frac{|E|-|e|}{|e|}$ all edges except the smallest unexplored boundary edge in the graph will be blocked. Recall Definition 7.2 which state that an edge $e = (u, v)$ is *close* to another edge $e' = (u', v')$ if the length of the shortest known path from u to v' is at most $(1 + \delta) \cdot |e|$, otherwise we say that e is *far from* e'. $|E|$ is an upper bound on the length of any shortest path in the graph. Thus we get from the definition of close that any edge e is close to all other edges if $|E| \leq (1 + \delta) \cdot |e|$. Isolating δ we get $\delta \geq \frac{|E|-|e|}{|e|}$

When $\delta < -1$ all edges will be unblocked. To see this, let $\delta < -1$ in Definition 7.2 then we have that an edge $e = (u, v)$ is *close* to another edge $e' = (u', v')$ if the length of the shortest known path from u to v' is at most $(1 + \delta) \cdot |e| < 0$. Thus no edge can be close to another edge for $\delta \leq -1$. This implies that no edges can be blocked.

When no edge can be blocked $\texttt{Blocking}_\delta$ behave exactly like DFS. Thus it is true that the "blocking condition depending on a fixed parameter $\delta > 0$, determines when to diverge from DFS."[15] though it cannot behave exactly like DFS in for the range $\delta > 0$ considered in [15].

There is an edge case for $\delta = -1$. Then an edge $e = (u, v)$ is *close* to another edge $e' = (u', v')$ if the length of the shortest known path from u to v' is at most $(1 - 1) \cdot |e| = 0$. This is however satisfied if the shortest path to e' has length 0. But then $|e'| \not< |e|$. So e is not blocked. Thus $\texttt{Blocking}_\delta$ also behave exactly like DFS for $\delta = -1$.

7.4 `Blocking`$_\delta$ proofs

Lemma 7.1. *`Blocking`$_\delta$ terminate with no unexplored vertices.*

Proof. Suppose for the sake of contradiction that `Blocking`$_\delta$ terminates with some vertex v unexplored. Then, as we only consider connected graphs, there must also be one or more boundary edges, let the shortest be $e = (u, v)$ with u explored at the point of termination. From Definition 7.3 it follows that e is unblocked, as e is the shortest boundary edge. e must have been blocked at some earlier point, otherwise, we would have traversed e when we explored u, as the first clause of the disjunction in `Blocking`$_\delta$ would be satisfied. Thus there is some point in time at which e becomes unblocked. Let $e' = (x, y)$ be the last edge to have blocked e. When y is explored, e will satisfy the seconds clause in the disjunction in `Blocking`$_\delta$ (G,y), as e is unblocked. Thus e will be traversed, which is a contradiction to e being the shortest boundary edge. In conclusion, every vertex is explored [15].

□

7.4.1 Competitive ratio of `Blocking`$_\delta$

We will now dive into the proof of the competitive ratio of the `Blocking`$_\delta$ algorithm. We will show that `Blocking`$_\delta$ has a competitive ratio of $2(2+\delta)(\frac{2}{\delta}+1)$ for planar graphs. For $\delta = 2$, which is the optimal choice of δ, `Blocking`$_\delta$ yields a competitive ratio of 16.

We will break the proof into multiple lemmas. First we argue in Lemma 7.2 that the cost of `Blocking`$_\delta$ is at most $2(2+\delta)|P|$. Then we argue in Lemma 7.3 and 7.4 that $|P|$, the combined length of all edges in P, is at most $(\frac{2}{\delta}+1)|MST|$, where $|MST|$ is the combined length of all edges in the minimum spanning tree which shares the most edges with P. Finally we argue in Lemma 7.5 that $|MST|$ is a lower bound on the length of the optimal tour. Combined this yields a competitive ratio of $2(2+\delta)(1+\frac{2}{\delta})$.

Definition 7.5. *Charge*
A edge e is *charged* when it is traversed in line 3 of `Blocking`$_\delta$. The amount e is *charged* is the combined cost from line 2, 3 and 5 in the same iteration of `Blocking`$_\delta$.

Lemma 7.2. *The cost of `Blocking`$_\delta$ is at most* $2(2+\delta)|P|$

Proof. An edge e must be a boundary edge to be charged since only boundary edges are traversed in line 3 of `Blocking`$_\delta$. Additionally, it will only be charged once, as it is then now no longer a boundary edge.

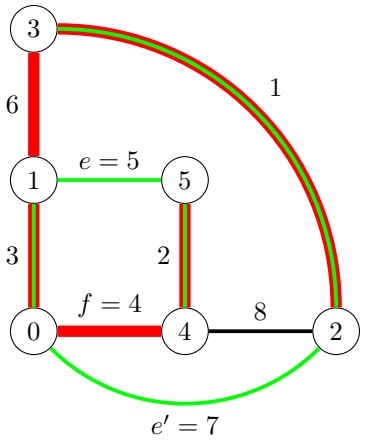
The edge e can at most be charged $2(2+\delta)|e|$. There are two cases for an unblocked edge $e = (u, v)$ being traversed in line 3. The first case is that the searcher is in $y = u$ in which case e will simply be charged at most $2|e|$. This charge is obtained, as line 2 of `Blocking`$_\delta$ contributes with 0, line 3 with $|e|$ and line 5 with at most $|e|$, as there could potentially be a shorter path than going back through e. The second case is that the searcher is in y and some edge (u', y) has previously blocked e. The searcher will then have to traverse at most $(1+\delta)|e|$ to get from y to u, otherwise (u', y) would have been blocked, and then traverse $|e|$. Together this is $(2+\delta)|e|$. The searcher will also have to go back, so in total at most $2(2+\delta)|e|$. Summing over the charge of all edges $e \in P$ we get that the cost of `Blocking`$_\delta$ is at most $\sum_{e\in P} 2(2+\delta)|e| = 2(2+\delta)|P|$. This is the total cost of `Blocking`$_\delta$ since no other cost occurs other than the charge of the edges in P. □

Lemma 7.3. *Let MST be a minimum spanning tree. If there are multiple minimum spanning trees consider the one sharing the most edges with P. If an edge $e \in P \setminus MST$ is contained in a cycle C in $P \cup MST$, then the length of the cycle C is at least $(2+\delta)|e|$ [15]*

Before proving the lemma, we will first look at an example. In Figure 7.4.1 we have a graph with edges are marked red if in *MST*, green if in *P* and black otherwise. In the proof two sets will be important. The first set $P \setminus MST$, from which e may be chosen, is colored purely green. Note that $(P \setminus MST) \cap C = C \setminus MST$. The second set $MST \setminus P$ is colored purely red. Note again that $(MST \setminus P) \cap C = C \setminus P$. In the graph on Figure 7.4.1 we could have $e = (1, 5)$, and a cycle $C \in MST \cup P$ containing e could be:

$$1 \to 3 \to 2 \to 0 \to 4 \to 5 \to 1$$

The lemma then states that the length of $|C| = 6+1+7+4+2+5 = 25 \geq 10.75 = (2+0.15)\cdot 5 = (2+\delta)\cdot|e|$ for $\delta = 0.15$



Proof. In order to prove the statement for e we will prove it for $e' = (u', v')$, where e' is the largest edge $\in C \setminus MST$, where C is the same cycle that contains e. In case there are multiple largest edges, we pick e' to be the edge among these that is charged the latest. Then clearly, $|C| \geq (2+\delta)|e'| \implies |C| \geq (2+\delta)|e|$ as $e \leq e'$ [15, 18].

First, we argue that e' is the largest edge in C. We already have that no edge in $C \cap P$ is larger than e'. Therefore we only need to argue that no edge in $C \setminus P$ is larger than e'. In Figure 7.4.1 $C \setminus P$ are the purely red edges in the cycle C

Assume for the sake of contradiction that there exist an edge f with $|f| \geq |e'|$ in $C \setminus P$. We can then remove f from MST which will split the MST into two sub-trees T_1 and T_2 rooted at the two vertices incident to f. Otherwise the MST would have contained a cycle. For the same reason the two vertices incident to f cannot be in the same tree. In Figure 7.4.1 we have $T_1 = \{(4,5)\}$ and $T_2 = \{(0,1),(1,3),(3,2)\}$. The edge f used to be the edge connecting T_1 and T_2, but going the other way around the cycle C, we can find another edge connecting the two trees. The path $C - f$ starts in T_1 and ends in T_2 and must therefore contain this edge $t \notin MST$ connecting T_1 and T_2. With the edge t we can obtain a new spanning tree $T_1 \cup T_2 \cup t$. We have that $|t| \leq |e'|$ since t is in $C \setminus MST$ and e' was chosen a largest edge in this set. By the assumption that $|e'| \leq |f|$ we then get that $|t| \leq |f|$. There are now two cases. In the first case $|t| < |f|$. This would yield a spanning tree $T_1 \cup T_2 \cup t$ smaller than MST, which is a contradiction. In the second case $|t| = |f|$. This would yield a spanning tree $T_1 \cup T_2 \cup t$ which shares more edges with P, since $t \in P$ and $f \notin P$ (more precisely $t \in C \setminus MST = C \cap (P \setminus MST)$ and $f \in C \setminus P$). Which is also a contradiction to MST being the minimum spanning tree sharing the most edges with P. Therefore such an edge f cannot exist, meaning no edge in $C \cap MST$ is larger than e'. We conclude that e' is a largest edge in C[15, 18].

Assume, again for the sake of contradiction, that $|C| < (2+\delta)|e'|$. Consider the point in time during the execution of the algorithm where e' is charged. As e' is a boundary edge and contained in the cycle C we can conclude that v' must be an unexplored vertex in the cycle C. There exists at least two paths from $u' \to v'$. One directly through e' and one through $C - e'$. Because v' is unexplored, C is not fully explored, so the path $C - e'$ must encounter another boundary edge $e'' = (u'', v'') \in C$ different from e'. Note that walking the path $C - e'$ could potentially encounter many boundary edges, but we define e'' to be the first of such boundary edges at the time e' is charged [15].

We know that e' is not blocked, as we consider the algorithm at the point in time where e' is charged. Recalling the Definition 7.3 of *blocked* and Definition 7.2 of *close* this implies that no other edge $a = (u, v)$ in the entire graph is both smaller than e' ($a < e'$) and simultaneously the distance from $u' \to v$ is less than or equal to $(1+\delta)|e'|$ (e' is close to a). From our assumption we get that $|C| - |e'| < (1+\delta)|e'|$ as seen on Figure 14. $C - e'$ is exactly the path on which we argued that e'' lies. We conclude that the shortest path from $u' \to v''$ is less than $(1+\delta)|e'|$. This implies that e' is *close* to e''. As we argued above then if $e'' < e'$, e' would be blocked, but e' is not blocked. We can therefore conclude, using the assumption, that $e' \leq e''$ [15, 18].

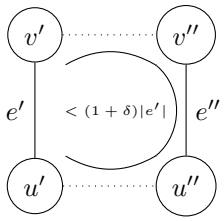


Figure 14:
The Cycle C: $u' \to v' \to ... \to v'' \to u'' \to ... \to u'$. C is a subset of a larger graph. The dotted lines are paths with arbitrary many vertices. Note that potentially $u' = u''$ or $v' = v''$, but not both at the same time.

We previously showed that e' is a largest edge in C, so it must then be that $|e'| = |e''|$. Recall that we consider the point in time where e' is charged. The unexplored edges of P in $C - e'$ must be smaller than e', as otherwise e' would not be the edge of maximal length that is charged the latest. e'' is the first unexplored edge of $C - e'$. Therefore, if $|e'| = |e''|$ then $e'' \notin P$. We already argued that there cannot exist an edge $f \in C \setminus P$ where $|f| \geq |e'|$. Thus, if $|e'| = |e''|$ then $e'' \notin MST$ [19].

We now get a contradiction on the fact that C consists of only edges $P \cup MST$, as $e'' \notin P$ and $e'' \notin MST$, which shows the claim [15, 18, 19]. □

We will now relate $|P|$ to $|MST|$. In doing so we will need some new terminology. For an embedding of a graph a *face* F is a region bounded by a set of edges, E_F, and a set of vertices, V_F. On Figure 15 the vertices and edges of F_2 are $\{2, 3, 4, 5\}$ and $\{(2, 3), (3, 4), (4, 5), (5, 2)\}$. Note that the unbounded outer face is also considered a face. An edge *borders* two faces F_i and F_j if it is in the edge set of both F_i and F_j [14].

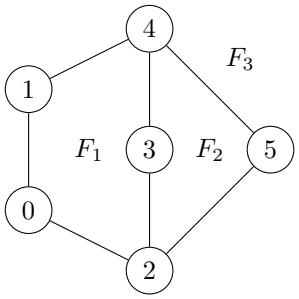


Figure 15:
The planar embedding of a graph. Faces are marked with F_i. Border edges between F_1 and F_2 are $\{(2, 3), (3, 4)\}$. Border vertex between F_1 and F_2 is $\{3\}$

Lemma 7.4. $|P| \leq (\frac{2}{\delta} + 1)|MST|$ *for planar graphs*

Proof. In proving the upper bound on P we break P into $P \setminus MST$ and $P \cap MST$, as $|P| = |P \setminus MST| + |P \cap MST|$. It is easy to see that $|P \cap MST| \leq |MST|$. Proving an upper bound for $P \setminus MST$ is however more involved, and is split into two parts. In part 1 we show that every edge $e \in P \setminus MST$ has a corresponding cycle C_e consisting of edges in E_{F_e} where F_e is one of the faces that e borders. In part 2 we show using Lemma 7.3 that the length of these cycles gives an upper bound on $P \setminus MST$.

Part 1
Consider the planar embedding of the subgraph consisting of edges in $P \cup MST$. Omitting the edges not in $P \cup MST$ from the embedding is valid, as these do not contribute to $|P|$. For this part of the proof, we first argue that there is a bijective mapping from edges in $P \setminus MST$ to inner faces of the graph. The proof for this mapping is to first show that every face must have at least one edge $\in P \setminus MST$. Then show for any pair of faces F_i and F_j that there is edges $e \in E_{F_i} : e \notin E_{F_j}$ and $e' \in E_{F_j} : e' \notin E_{F_i}$, where E_{F_i} and E_{F_j} is the edges $\in P \setminus MST$ of the respective faces.

The first claim is true, as the vertices of a face form a cycle. A cycle in a graph of $P \cup MST$ must contain at least one edge $\in P \setminus MST$, as otherwise the MST would have contained a cycle. To show the second claim consider the two possible cases. The first case is that the two faces do not share any edges $\in P \cup MST$. Then this is trivial, as we just showed that every face has at least one edge $\in P \setminus MST$ and the two faces do not share any edges. The second case is that the two faces share one or more edges, as in Figure 15. To show the claim, suppose, for the sake of contradiction that there is only one edge $\in P \setminus MST$ in the union of the two faces edges. Then that edge must be a border edge between the two faces. Otherwise one of the faces would be a cycle of only MST edges. Consider now the cycle consisting of the combined vertices and edges of the two faces, excluding the border vertices and edges. On Figure 15 the excluded border vertex is $\{3\}$ and the excludes border edges are $\{(2,3),(3,4)\}$. This cycle does not contain the sole edge $\in P \setminus MST$ and must therefore also be a cycle of only MST edges. This is a contradiction and shows that there are at least two different edges $\in P \setminus MST$ in the union of the edges of any pair of faces. This concludes that there is a bijective mapping from $P \setminus MST$ to inner faces in the graph. We will in the following use this mapping to refer to the cycles of the faces as C_e, where e refers to the corresponding edge $\in P \setminus MST$.

Part 2
We are now ready to look at the cost of the subset $P \setminus MST$ [15].

$$|P \setminus MST| = \sum_{e \in P \setminus MST} |e| = \sum_{e \in P \setminus MST} \frac{(1+\delta)|e|}{1+\delta} = \sum_{e \in P \setminus MST} \frac{(2+\delta)|e| - |e|}{1+\delta} \leq \frac{1}{1+\delta} \sum_{e \in P \setminus MST} |C_e - e|$$

The last $\leq$ is obtained from Lemma 7.3, which states that $C_e \geq (2+\delta)|e|$. We will now derive an upper-bound on $\sum_{e \in P \setminus MST} |C_e - e|$. As the graph is planar, any edge $\in P \cup MST$ can be contained in the cycle of at most two faces. Therefore, every edge $\in P \cup MST$ can contribute at most twice to $\sum_{e \in P \setminus MST} |C_e|$. Thus an upper-bound on this is $2|P \cup MST| = 2(|MST| + |P \setminus MST|)$. Additionally, we can subtract $|P \setminus MST|$, corresponding to e's in $|C_e - e|$, to obtain the below equation.

$$|P \setminus MST| \leq \frac{1}{1+\delta} \sum_{e \in P \setminus MST} |C_e - e| \leq \frac{1}{1+\delta} 2\left((|MST| + |P \setminus MST|) - |P \setminus MST|\right) = \frac{1}{1+\delta}(2|MST| + |P \setminus MST|)$$

Which we can rewrite in the following way:

$$\begin{aligned} |P \setminus MST| &\leq \frac{1}{1+\delta}(2|MST| + |P \setminus MST|) \\ (1 - \frac{1}{1+\delta})|P \setminus MST| &\leq \frac{1}{1+\delta} 2|MST| \\ |P \setminus MST| &\leq \frac{1}{1+\delta} \frac{1+\delta}{\delta} 2|MST| = \frac{2}{\delta}|MST| \end{aligned}$$

Since:

$$\frac{1}{1 - \frac{1}{1+\delta}} = \frac{1(1+\delta)}{(1 - \frac{1}{1+\delta})(1+\delta)} = \frac{1+\delta}{\delta}$$

In conclusion we get that the length of P is:

$$|P| = |P \setminus MST| + |P \cap MST| \leq |P \setminus MST| + |MST| \leq \frac{2}{\delta}|MST| + |MST| = \left(\frac{2}{\delta} + 1\right)|MST|$$

□

Lemma 7.5. *$|MST|$ is a lower-bound on the cost of the optimal offline tour*

Proof. This proof is an adaptation of the Held and Karp lower bound provided on page 253 in Cook et al. [4].

We argued in the *TSPM* section that the length of the optimal tour on an arbitrary graph is the same as on a corresponding fully connected graph. We used that for vertices u, v that are not adjacent, we can

create an edge $e = (u, v)$ between them, with the length of the shortest path from u to v. Additionally, we argued that the optimal tour in a fully connected graph do not visit any vertex more than once and do not traverse the same edge twice. With this we can, using the reduced fully connected graph, prove that $|MST|$ is a lower bound on the optimal tour for any graph.

Let V be all the vertices of the graph and T be the set of edges included in the optimal tour. The cost of the optimal tour is therefore $|T|$. Now let $T' = T - e_i$, where e_i is some arbitrary edge in T. The cost of T is then $|T| = |T'| + |e_i|$. Now T' is a spanning tree in V, as illustrated on Figure 16. From this we get that $|MST| \leq |T'|$, which implies that $|MST| \leq |T'| + |e_i|$. We conclude the lemma with $|MST| \leq |T|$, as $|T| = |T'| + |e_i|$.

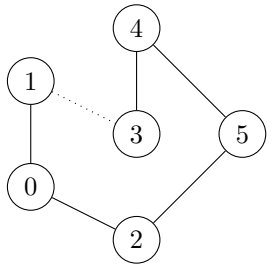


Figure 16:
The optimal tour on a graph. The edges not part of the optimal tour are omitted. The dotted edge is the edge e_i that is removed from T to obtain T'.

□

Theorem 7.6. *The algorithm* $\texttt{Blocking}_\delta$ *has a competitive ratio of* $2(2+\delta)\left(\frac{2}{\delta}+1\right)$ *for planar graphs.*

Proof. We showed in Lemma 7.2 that the cost of $\texttt{Blocking}_\delta$ is at most $2(2+\delta)|P|$. We then used the result from Lemma 7.3 to show that $|P| \leq (\frac{2}{\delta}+1)|MST|$ in Lemma 7.4. Combining this gets an upper-bound on the cost of $\texttt{Blocking}_\delta \leq 2(2+\delta)(\frac{2}{\delta}+1)|MST|$. Using Lemma 7.5 we conclude that the competitive ratio of $\texttt{Blocking}_\delta$ is

$$\frac{\texttt{Blocking}_\delta}{Optimal} \leq \frac{2(2+\delta)(\frac{2}{\delta}+1)|MST|}{|MST|} = 2(2+\delta)\left(\frac{2}{\delta}+1\right)$$

□

The parable $2(2+\delta)\left(\frac{2}{\delta}+1\right)$ is shown on Figure 17. Minimizing the function gives that the lowest competitive ratio is obtained when $\delta = 2$, namely 16. This important result is captured in the below corollary 7.6.1. Interestingly, the competitive ratio quickly approaches ∞ when $\delta \to 0$ in the interval $[0, 2]$ and likewise when $\delta \to \infty$ in the interval $[2, \infty]$.

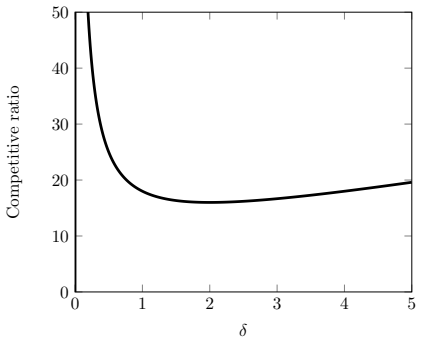


Figure 17: The theoretical competitive ratio of $\texttt{Blocking}_\delta$ as a function of δ.

Corollary 7.6.1. $\texttt{Blocking}_\delta$ *with* $\delta = 2$ *is 16-competitive on planar graphs*

7.4.2 General genus

The proof showing that $\texttt{Blocking}_\delta$ is 16-competitive on planar graphs can be expanded to show that $\texttt{Blocking}_\delta$ is $16(1+2g)$ competitive in the more general case of graphs with genus of at most g. Here the genus of a graph is the number of handles that must be added to the plane to obtain a surface onto which the graph can be embedded without any edges crossing each other [21].

7.4.3 Error in [15]

The proof of Theorem 7.6 is based on the proof of claim 1 in [15]. We found the proof of claim 1 to be incorrect, and will in this section describe why the original proof did not hold. The part of the proof that does not hold is the following:

> "Due to the choice of MST, the edge e' is strictly larger than any edge in $C \cap MST$. Otherwise, we could replace e' with an edge in MST to obtain a smaller minimum spanning tree or to obtain a minimum spanning tree that shares more edges with P." —[15] page 6 lines 5-7

The problem is that it is implicitly assumed that there were no other edges of $P \setminus MST$ on the cycle $(e = e')$, in which case the cycle property of MST [22] give that e' is strictly larger than any edge in $C \cap MST$. The cycle property of MST states that the largest edge in any cycle in the graph cannot be included in the MST. However we found a counterexample to that assumption where there are two edges of $P \setminus MST$ on the cycle. In fact, it is the example we have already been studying in Figure 7.4.1 and shown again here in Figure 18 for reference.

Having two edges of $P \setminus MST$ on the cycle mean that when replacing e' with an edge in MST we are not guaranteed to obtain a spanning tree. As seen in our proof of Lemma 7.3 replacing e' with f in the MST yield two separate trees $T_1 = \{(4,5)\}$ and $T_2 = \{(0,1),(1,3),(3,2)\}$.

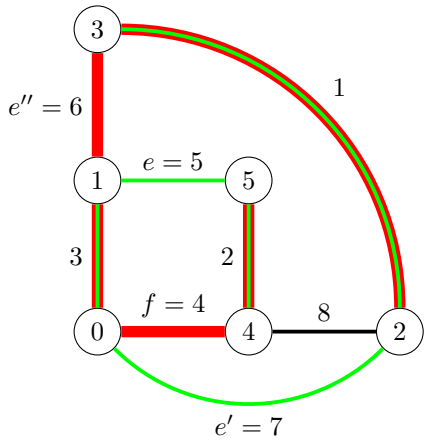


Figure 18:
A complete run of $\texttt{Blocking}_\delta$ with $\delta = 0.15$ on the above graph, starting at the vertex with lable 0.
The edges are marked red if in MST, green if in P and black otherwise.

Figure 18 showcase the problem the proof. It is however not an actual counterexample to the proof in [15]. The edge $e'' = (1,3)$ is the first boundary edge on $C - e'$, but swapping e'' and e' in would not split the MST into two separate trees.

The error has been confirmed by one of the authors of [15] in the correspondence [18, 19]. The clearest explanation of the alternative proof is found in [19].

7.5 Improvements on $\texttt{Blocking}_\delta$

The $\texttt{Blocking}_\delta$ algorithm might traverse unnecessarily many edges when it is done exploring the graph. During the execution of $\texttt{Blocking}_\delta(G, y)$, there is in line 4 a recursive call $\texttt{Blocking}_\delta(G, v)$, and thereafter

in line 5 the shortest know path from v to y is traversed. When the graph is fully explored this will cause $\texttt{Blocking}_\delta$ to revisit all the vertices corresponding to calls on the recursion stack. Here taking the shortest path to the origin vertex is the wanted behaviour. Recall that this is the behaviour of the nearest neighbour Algorithm 1. Given that it is stored whether each known edges is also explored, it can be checked if the graph is fully explored. We can thus use more memory to find a slightly shorter tour. If the path corresponding to vertices on the recursion stack is the shortest path to the origin vertex, there is no difference. This happened in Figure 13. If the recursion stack never grows to more than one, this will definitely be the case. An example of this is a star graphs with edges of equal weight.

7.5.1 Ideas for a tighter bound on competitive ratio

We can make an infinitesimal improvement on the lower bound on the optimal tour. A tour requires the searcher to return to the origin vertex. After having explored all vertices at least one more edge, call it e, must be traversed. The lower bound of $|MST|$ on the optimal tour does not take this into account. Therefore $|MST| + |e|$ is a lower bound on the optimal tour. This would give a competitive ratio strictly less than 16.

7.6 Lower bounds for $\texttt{Blocking}_\delta$

In this section, we will first show how to construct a planar graph for which $\texttt{Blocking}_\delta$ has a competitive ratio of 5.25. Then we will briefly mention that $\texttt{Blocking}_\delta$ does not have a constant competitive ratio on general graphs, as shown by the lower bound in [15].

7.6.1 Lower bound for $\texttt{Blocking}_\delta$ on planar graphs

We now show a graph on which $\texttt{Blocking}_\delta$ has a competitive ratio of 5.25. The graph will consist of a simple cycle. During the execution of $\texttt{Blocking}_\delta$ the known graph will be two paths starting in the origin vertex. The idea is to force $\texttt{Blocking}_\delta$ to alternate between exploring the boundary edge at the end of each of the two paths. Thus forcing it to backtrack all previously traversed edges each time. The weights are generated in such a way that only one boundary edge is unblocked at a time. This edge is always on the other path than where the searcher is currently located and thus forcing the alternation between the two paths. More precisely, the weight of the i'th edge is $|e_i| = \frac{\sum_{e \in E_i} |e|}{\delta+1} + 1$, where E_i is all the weights generated before e_i. In our experiment, we used $\texttt{Blocking}_\delta$ with $\delta = 2$ as this is the value where $\texttt{Blocking}_\delta$ has the lowest theoretical competitive ratio. The order of the edges is starting at the origin and then alternating between the two paths, as seen in Figure 19. The weights then ensure that when the searcher has just explored eg. vertex 3, the edge $(3, 5)$ will be blocked. However $(2, 4)$ will be unblocked, thus forcing $\texttt{Blocking}_\delta$ to walk on every explored edge again. Recall Definition 7.3 to see why this is true. Definition 7.3 states that $(3, 5)$ is blocked by $(2, 4)$, if $|(2, 4)| < |(3, 5)|$ and the shortest path from 3 to 4 is at most $(1 + \delta) \cdot |(3, 5)| = 3 \cdot |(3, 5)|$. The task is then to chose a correct value for $|(3, 5)|$. The path from 3 to 4 is all the previous edges $\sum_{e \in E'} |e| = |(0, 1)| + |(0, 2)| + |(1, 3)| + |(2, 4)|$. Therefore $3 \cdot |(3, 5)|$ must be larger than this sum for $(3, 5)$ to be blocked. This is achieved using the function from above: $|(3, 5)| = \frac{\sum_{e \in E'} |e|}{3} + 1 = \frac{|(0,1)|+|(0,2)|+|(1,3)|+|(2,4)|}{3} + 1 \approx 3.34$

On Figure 19 $\texttt{Blocking}_\delta$ would therefore explore the vertices in the order $1, 2, 3, 4, 5, 6, 7$. Additionally $\texttt{Blocking}_\delta$ must backtrack its path as we have described in Section 7.5. Thus $\texttt{Blocking}_\delta$ will find a tour of length ≈ 61.1. The optimal tour simply traverses the cycle once, thus the length of the optimal tour is the sum of the edge weights ≈ 28.7. Giving a competitive ratio of ≈ 2.1. By increasing the number of vertices, we can use this approach to force a competitive ratio of 5.25. The graph that obtains this competitive ratio can be seen in Appendix C.

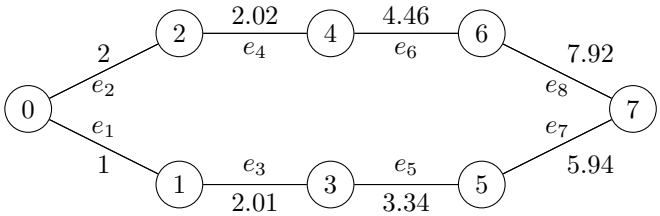


Figure 19: Graph type that forces a competitive ratio of 5.25 on $\texttt{Blocking}_\delta$. This concrete graph do not have a competitive ratio of 5.25, but in the interest of simplicity, this example is sufficient to show the approach. Note that on Figure 19 the weights of the first 4 edges has been manually chosen to get the function started.

Interestingly, 5.25 is the limit value for the competitive ratio using this approach. The ratio is achieved at approximately 35 vertices. Adding further vertices will not force the competitive ratio above 5.25.

One possible explanation for why the competitive ratio converges is that the small edges become so insignificant compared to the large edges that the number of times these small edges are traversed does not matter.

7.6.2 Lower bound for $\texttt{Blocking}_\delta$ on general graphs

In [15] the main contribution is actually showing that $\texttt{Blocking}_\delta$ does not have a constant competitive ratio on general graphs. They do this via a complicated lower bound construction showing that on general graphs the competitive ratio of $\texttt{Blocking}_\delta$ is $\Omega(n^{\frac{1}{\delta+4}})$. They use bipartite graphs with high outdegree, in which they swap vertices in the graph with gadgets. These gadgets are constructed with light edges of weight 1 and heavy edges of weight $w > 1$. The graph is constructed in a way that forces $\texttt{Blocking}_\delta$ to traverse all the heavy edges. We will not dive any deeper into this construction. We do however note that the constructed graph is not planar in general. Thus it does not state anything about the planar setting that we have mainly considered above. The lower bound of $\Omega(n^{\frac{1}{\delta+4}})$ is only on $\texttt{Blocking}_\delta$ and does not apply to any algorithm.

In fact, [15] also describes the hierarchical DFS, which is 4-competitive on the constructed graph. As the name hints at, it is inspired by DFS. The algorithm is hierarchical in the sense that it finds a part of the tour consisting of edges of weight w before finding the part of the tour consisting of edges of weight $w' > w$ and so on.

Part III
Implementation and Experiments

Our project includes an implementation of $\texttt{Blocking}_\delta$ and various components that are needed to experiment with $\texttt{Blocking}_\delta$. Section 8 will outline the content of the implementation and Section 9 will present the conducted experiments.

8 Implementation

The implementation is written in Python 3 and the source code can be found on: `GitHub`.
In addition to the implementation of $\texttt{Blocking}_\delta$, the code contains implementations of Dijkstra's algorithm, which is used by $\texttt{Blocking}_\delta$ to calculate the distances from its current location to unexplored vertices, and Prim's algorithm, which is needed to compute the *MST*. The *MST* was mainly used for theoretical purposes. It was especially useful in constructing the graph in 18, which was used to show the error in the proof of claim 1 in [15] as described in Section 7.4.3. The implementation in general was used to check our understanding of the behaviour of $\texttt{Blocking}_\delta$. For this purpose, the code supports the export of tikz figures that shows each iteration of $\texttt{Blocking}_\delta$ on a graph.

For experimenting with the competitive ratio, we needed a way to solve *TSP* instances. A fully connected graph is required to solve the *TSP* instance, but we do not want to limit our experiments to this. Recalling Section 3.2, we showed a reduction from *TSP* with multiple visits on connected graphs to *TSP* on fully connected fully. This reduction is implemented, such that we can conduct experiments on not fully connected graphs. The code contains a brute-force *TSP*-solver, but we quickly realised that this was not fast enough. Therefore, as this is not a *TSP* project, we used a third-party library for solving *TSP* instances [11]. Graphs of up to 20 vertices are feasible for this library.

8.1 Random graph generation

The code also contains a way of generating random planar graphs which we will now discuss in detail. Generating random graphs is important, as it allows us to test the behaviour and competitive ratio of $\texttt{Blocking}_\delta$ on a large number of graphs. We decided to only generate planar graphs, as these can be used to experiment with Theorem 7.6.

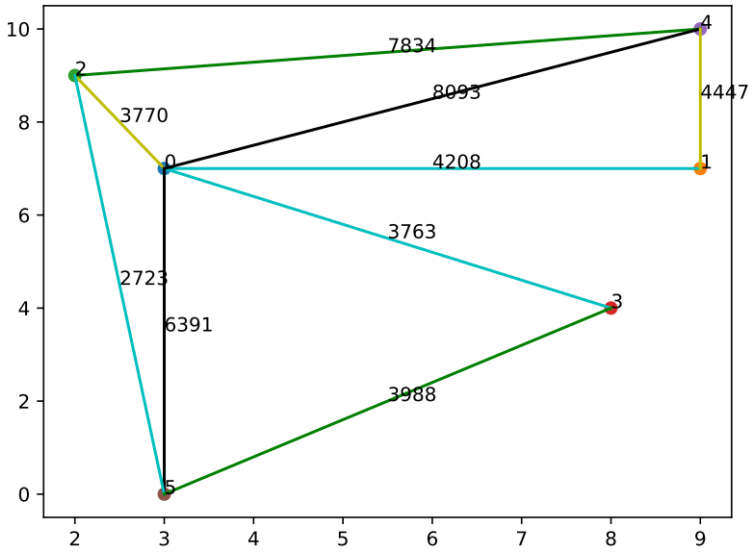


Figure 20: A randomly generated planar graph with uniformly distributed weights.

We generate planar graphs by generating n unique uniformly distributed points in the $\mathbb{R}^2$ plane. For ease of implementation, these points are restricted to $\{0, 1, \ldots, 2n\}^2$ where n is the number of desired vertices in the graph. Each point in the plane then represents a vertex in the graph. An edge between two vertices is represented as lines between the two corresponding points. It then follows that if no two lines intersect in the plane, the graph is planar. Figure 20 is an example of points and lines corresponding to vertices and edges of a planar graph. Lastly, these vertices and edges can be converted into Vertex and Edge objects in our code.

We want to generate connected graphs, as a tour is undefined if any vertex is not reachable from the origin vertex. We enforce this by creating a spanning tree T as the first step. Then adding additional edges as the second step. For the construction, we make use of the order of the vertices. This is simply the lexicographical order of their labels. Thus referring to the first vertex will be the vertex with the lowest lexicographical order. This order is not correlated with the positions of the vertices in the plane. It does however mean that we consider the vertices in the same order, every time they are iterated over.

Generating the spanning-tree T is done in the following way. Initially, T contains the first vertex. Then for every vertex $b \notin T$, choose a random vertex $a \in T$. The edge (a, b) is then created unless it intersects with an already created edges in the plane. In this case we keep b, but choose a new $a' \in T$ randomly from T. We then try to create the edge (a', b) instead. b is then added to T before the next iteration. We continue until all the vertices are in T. If we cannot find a valid edge between b and a vertex in T, we give up and try over with a new set of points. This approach is an attempt at avoiding unwanted patterns in the graphs.

For the next step, the generator uses a *connectivity* parameter. This parameter determines the likelihood of two vertices being adjacent. If *connectivity* $= 30$, then there is a 30% chance that a vertex will try to create an edge to another vertex. We iterate all the vertices again in the lexicographical order of their labels. More precisely, for every $v \in V$, we do a nested iteration $v' \in V$. This lets us iterate all the pairs of vertices. Add an edge (v, v') with likelihood *connectivity* and if the line between the two corresponding points does not intersect any other lines in the plane. We do not add duplicate edges, but even if (v, v') was not created due to the *connectivity* parameter, (v', v) might be added in a later iteration.

The weights are generated at the same time as the edges themselves. In other words, the iterative procedure described implies an ordering of the edges in the graph. This order is the order in which the weights are assigned.

The weights of the edges can be generated in a number of different ways. The *uniform* distribution is integers between $[1, 10000]$, and the *normal* distribution is floats with mean 5 and variance 3 restricted to be at least 0.00001. For the functions *linear*, *polynomial* and *exponential* the weights are i, i^2 and 1.5^i for $i = 1, 2, .., m$, where m is the number of edges. These are assigned in ascending order. There are two different methods for using the exponential weights. In the *small exponential* method, the weights are chosen assigned in their order, namely $1.5^1, 1.5^2, \ldots, 1.5^m$. In the *large exponential* method, the weights are chosen randomly from $1.5^1, 1.5^2, \ldots, 1.5^{m^2}$. This has the effect that the difference between weights in the graph will be much larger.

9 Experiments

We are now ready to test the implementation of $\texttt{Blocking}_\delta$, and investigate how the theoretical competitive ratio compares to the competitive ratios of $\texttt{Blocking}_\delta$ run on some concrete graphs. The primary result of our experiment is shown in Figure 21. We will discuss it in regards to δ in Section 9.1 and the weights in Section 9.2. The experiments are conducted on random connected graphs, generated as described in Section 8.1, where each graph have 15 vertices and approximately $25 - 30$ edges. This is the number of vertices where the graphs are sufficiently complex to force a high competitive ratio for $\texttt{Blocking}_\delta$ while remaining fast to solve for the *TSP*-solver. Note that we also present a run of $\texttt{Blocking}_\delta$ on a concrete graph 7.6.1 giving a competitive ratio of 5.25.

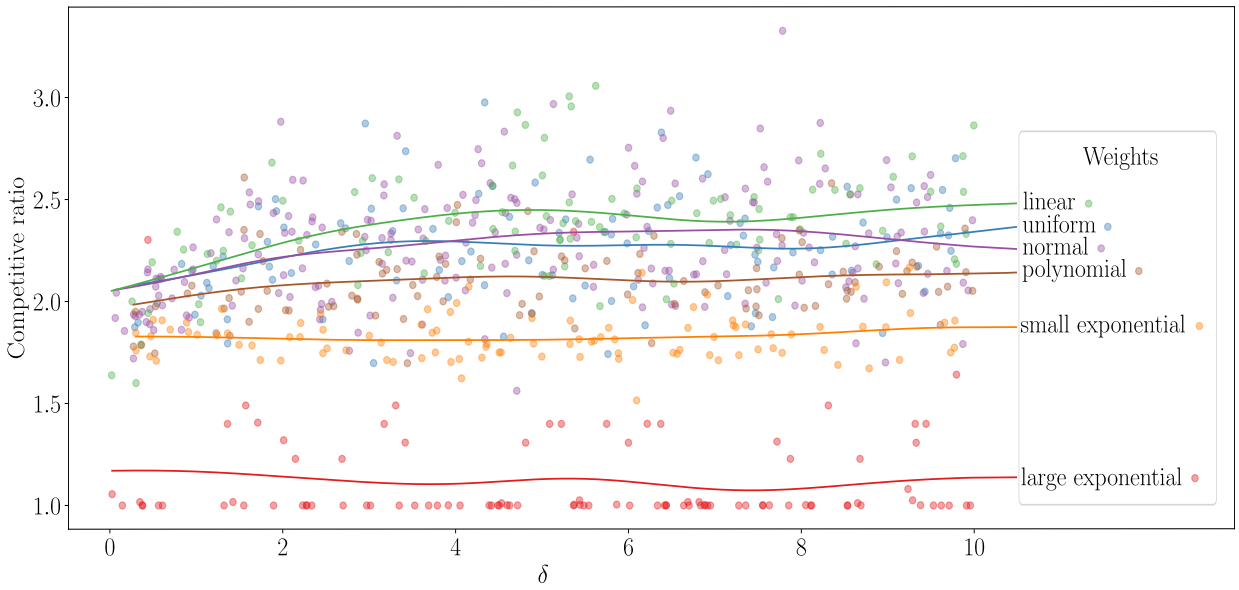


Figure 21: The $\texttt{Blocking}_\delta$ algorithm run on graphs with 15 vertices and different edge weight distributions. See Section 8.1 for a description of the edges weight distributions. Each dot mark a data point corresponding to a single run of $\texttt{Blocking}_\delta$. The lines are the moving average of the data points for each of the weights distributions separately.

9.1 Delta

The moving average lines show a slight trend towards that higher δ equals higher competitive ratio, but this seems insignificant. We have conducted further experiments with higher values of δ, but there is no difference in the behaviour of $\texttt{Blocking}_\delta$ when $\delta > 10$. This is because for $\delta \approx 10$, every boundary edge except the smallest will most likely be blocked in graphs of this size. Increasing δ will not change that. This is of course only tested on the relatively small graphs of maximum 20 vertices for which we were able to find the optimal offline solution.

The δ-parameter of $\texttt{Blocking}_\delta$ does not seem to have any significant effect on the competitive ratio for the graphs generated in this experiment. This is perhaps a surprising result. It would be interesting to investigate if there exist inputs, where the value of δ is important or maybe even follows the function from Theorem 7.6. Recall that Theorem 7.6 states that the competitive ratio of $\texttt{Blocking}_\delta$ is at most $2(2+\delta)\left(\frac{2}{\delta}+1\right)$ for planar graphs, as depicted on Figure 17.

Corollary 7.6.1 stated that $\texttt{Blocking}_\delta$ obtains a competitive ratio of 16 when $\delta = 2$. Figure 21 show that $\texttt{Blocking}_\delta$ performs much better in practice, at least on the graphs we were able to generate. Additionally the experiments do not show any spike in competitive ratio when δ approaches 0 as Figure 17 indicated. In fact, the opposite trend seems more likely.

The δ parameter of $\texttt{Blocking}_\delta$ could be studied further as it is done in [9].

9.2 Weights

We now look at how the weights of the graph influence the competitive ratio of $\texttt{Blocking}_\delta$. Most of the distributions give a similar competitive ratio but exponential weights give an extraordinarily low competitive ratio. In Figure 22 we have presented the data again in a way that makes the difference in competitive ratio between types of edges weights even clearer.

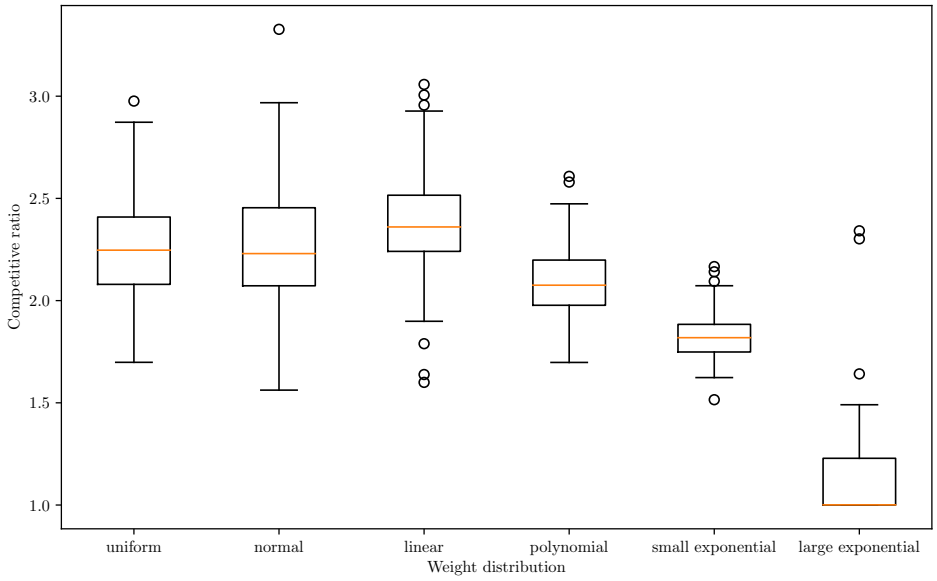
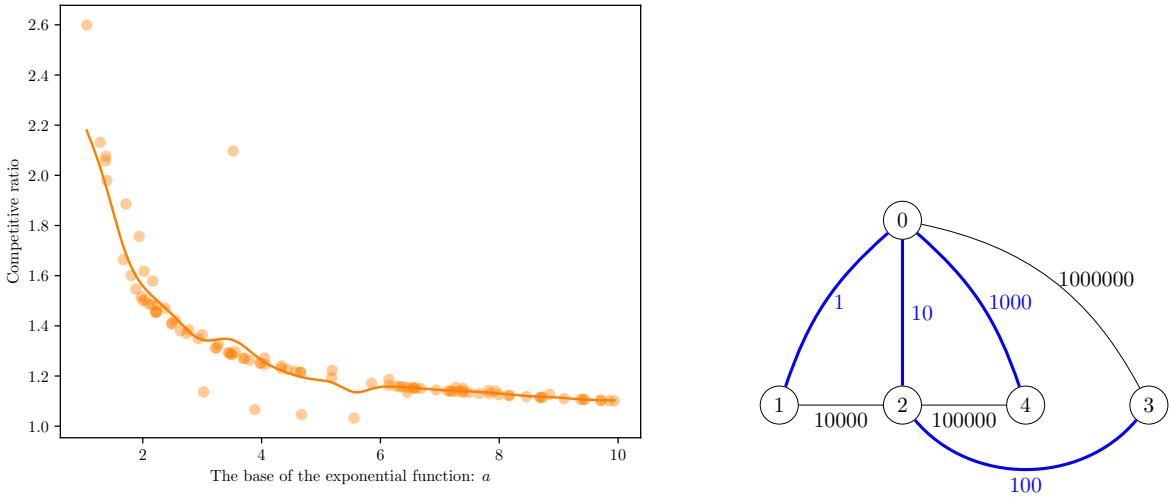


Figure 22: A box plot of the same data as Figure 21.

It was at first surprising that $\texttt{Blocking}_\delta$ has a competitive ratio near 1 on most graphs with *large exponential* weights, as defined in Section 8.1. Therefore we investigated this further by varying the base of the *small exponential* function as seen in Figure 23(a). We used the exact same method for creating the graph as described in Section 8.1, but with weights $a^0, a^1, \ldots, a^{m-1}$ for varying a instead of weights $1.5^0, 1.5^1, \ldots, 1.5^{m-1}$.



(a) (b)

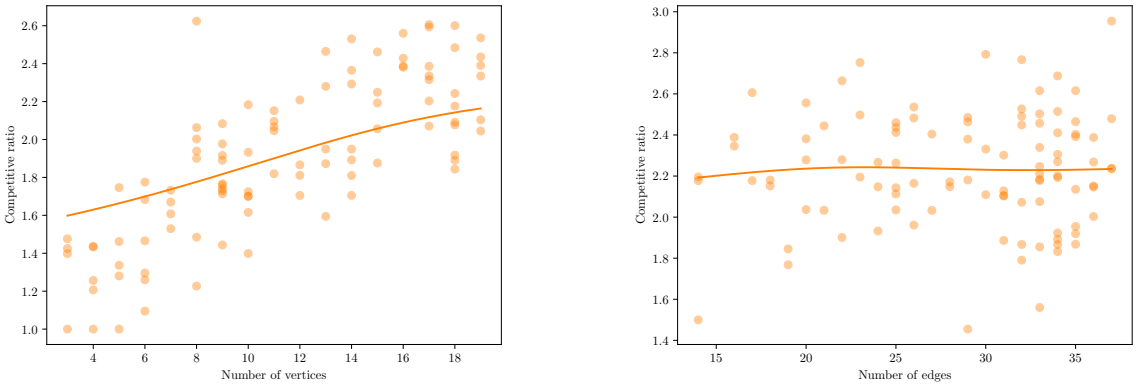
Figure 23: **(a)** Competitive ratio of $\texttt{Blocking}_\delta$ run on graphs with weights $a^0, a^1, \ldots, a^{m-1}$, where m is the number of edges in the graph and a is in the interval $[1, 10]$. The line is the moving average of the data points. **(b)** A graph with weights $10^0, 10^1, \ldots, 10^6$. The blue edges are the only edges of the tour found by $\texttt{Blocking}_\delta$ and the optimal tour. They are also the *MST* of the graph.

An example of a graph with weights $a^0, a^1, \ldots, a^{m-1}$ for $a = 10$ can be seen in Figure 23(b). For clarity, the graph has fewer vertices than the ones used in the experiment. The edges in the optimal tour, the tour found by $\texttt{Blocking}_\delta$ and *MST* are all the same in this example.

When the difference between edge weights is very large the optimal tour will have a large overlap with *MST*. Because including an edge not in *MST* will most likely increase the cost more than backtracking the edges of the *MST*. At the same time, the tour found by $\texttt{Blocking}_\delta$ will also have a large overlap with *MST*. This is because the large difference in edge weights will most likely make all but the smallest boundary edge blocked. Thus the behaviour of $\texttt{Blocking}_\delta$ will resemble that of Prims' algorithm for finding *MST*. This is similar to the case where δ is very large as described in Section 7.3. Furthermore, the large cost of the tour gives room for $\texttt{Blocking}_\delta$ to traverse the edges with weights which are very small in comparison many times without affecting the competitive ratio much. In conclusion, $\texttt{Blocking}_\delta$ will often find a tour with close to optimal cost on graphs with a very large difference between edge weights.

9.3 Size and connectivity

Our experiment shows that size *does* matter. Figure 24(a) shows a clear correlation between the size of the graph and the competitive ratio of $\texttt{Blocking}_\delta$. Larger graphs force a higher competitive ratio of $\texttt{Blocking}_\delta$. A plausible explanation for this is that $\texttt{Blocking}_\delta$ cannot deviate too much from the optimal tour on smaller graphs. That is, as on smaller graphs, a large percentage of the edges of the graph is part of the optimal tour, while on larger graphs, only a small percentage is. Therefore $\texttt{Blocking}_\delta$ is more likely to include edges of the optimal tour on smaller graphs.



(a) *Connectivity* = 30, $\delta = 2$, uniformly distributed weights

(b) Number of vertices = 15, $\delta = 2$, uniformly distributed weights

Figure 24: Competitive ratio of $\texttt{Blocking}_\delta$ as a function of **(a)** the number of vertices and **(b)** the number of edges. The lines are the moving average of the data points.

The *connectivity* parameter specifies how connected the graph is. It was used to generate graphs with a varying number of edges as seen in Figure 24(b). It appears that the number of edges does not affect the competitive ratio of $\texttt{Blocking}_\delta$.

When generating the data for both 24(a) and 24(b) the weights was chosen uniformly in the interval $[1, 10000]$ and $\delta = 2$. But for Figure 24(a) the connectivity parameter was set to 30, while for Figure 24(b) the size of the graphs was 15 vertices.

In conclusion, it appears that it is only the number of vertices that is important when testing $\texttt{Blocking}_\delta$ on the small graphs we were able to generate.

To experiment on larger graphs, with more than 20 vertices we cannot use *TSP*-solver from [11]. Therefore, we also conducted an experiment using *MST* as a lower bound on the length of the optimal solution. We can then compute the ratio between the length of the tour found by $\texttt{Blocking}_\delta$ and the length of the *MST*. This ratio will always be larger than the actual competitive ratio, thus giving an effective upper

bound on the competitive ratio. We can see in Figure 25 that this ratio levels out at around 2.7 when the graphs contain more than 100 vertices.

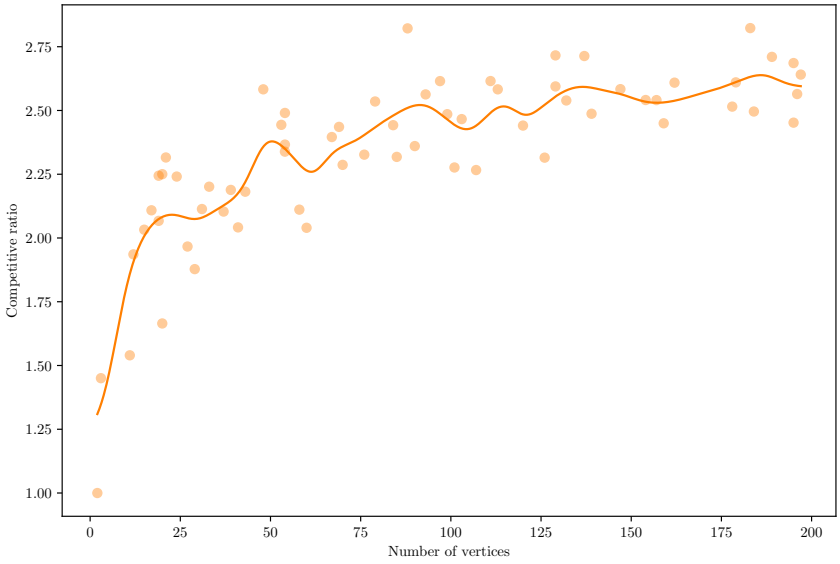


Figure 25: Competitive ratio of $\texttt{Blocking}_\delta$ as a function of the number of vertices. *Connectivity* $= 30$, $\delta = 2$, uniformly distributed weights. The competitive is calculated with the lower bound for the optimal tour of MST. The line is the moving average of the data points.

This concludes our experiments. On the small random graphs generated we found that the competitive ratio of $\texttt{Blocking}_\delta$ of around 2 is much lower than the theoretical upper bound of 16. We also found that the δ-parameter did not have a significant effect on these graphs. During the experimentation, we discovered a very low competitive ratio of $\texttt{Blocking}_\delta$ on graphs with a large difference in edge weights. We conducted further experiments to showcase this behaviour, which we then explained. Lastly, we saw that the competitive ratio of $\texttt{Blocking}_\delta$ increases with the number of vertices in the graph. We used the MST lower bound to show that this increase in competitive ratio stagnates at approximately 100 vertices.

Part IV
Conclusion

We studied the online graph exploration problem, presented three different algorithms and analysed them in different settings of the problem.

We used the NN algorithm as an introduction to the problem. This simple heuristic proved to have a competitive ratio of $\Theta(\log(n))$ on general graphs [12]. This motivated the need for more sophisticated algorithms.

We then presented the DFS algorithm which has an unbounded competitive ratio on general weighted graphs. However, we showed that DFS has a competitive ratio of 2 for unweighted graphs and that this bound is tight.

Lastly, we analysed the $\texttt{Blocking}_\delta$ algorithm in depth. We showed that $\texttt{Blocking}_\delta$ obtains a competitive ratio of 16 on planar graphs. This is the best known upper bound for planar graphs. In writing this proof, we encountered an error in claim 1 in [15]. This error was confirmed and based on a correspondence with one of the authors, our Lemma 7.3 presents a revised proof of claim 1 in [15]. It is not clear whether the upper bound of 16 on planar graphs can be improved. We tried experimentally to find a graph yielding the worst-case competitive ratio of 16. However, we only managed to find a graph for which $\texttt{Blocking}_\delta$ has a competitive ratio of 5.25. This graph is a simple cycle with exponentially increasing edge weights which force $\texttt{Blocking}_\delta$ to backtrack all previously explored edges every time it visits a new vertex. Thus an improved upper bound for $\texttt{Blocking}_\delta$ might still be possible. In parallel with our analysis of $\texttt{Blocking}_\delta$, we conducted a series of experiments. These studied the effect of the δ-parameter, the weight distribution and the number of edges and vertices on the competitive ratio.

There is a large gap between the lower bound of $3.3\overline{3}$ and the upper bound of 16 for planar graphs. In the words of Miyazaki et al. [16] "Narrowing this gap is a challenging problem". Moreover, for general graphs it is an open question whether or not an algorithm with constant competitive ratio even exists.

Acknowledgements

Thanks to Prof. Dr Pascal Schweitzer for the quick and precise response to our questions in regards to claim 1 in [15]. We also want to thank our supervisor Prof. Gerth S. Brodal for being a nice guy and providing helpful proofreading.

References

[1] Yuichi Asahiro, Eiji Miyano, Shuichi Miyazaki, and Takuro Yoshimuta. Weighted nearest neighbor algorithms for the graph exploration problem on cycles. *Inf. Process. Lett.*, 110(3):93–98, 2010.

[2] Alexander Birx, Yann Disser, Alexander V. Hopp, and Christina Karousatou. An improved lower bound for competitive graph exploration. *Theoretical Computer Science*, 868:65–86, 2021.

[3] Sebastian Brandt, Klaus-Tycho Foerster, Jonathan Maurer, and Roger Wattenhofer. Online graph exploration on a restricted graph class: Optimal solutions for tadpole graphs. *Theor. Comput. Sci.*, 839:176–185, 2020.

[4] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley and Sons, Inc, 1st edition, 1997. page 253, `http://math.mit.edu/~goemans/18453S17/TSP-CookCPS.pdf`.

[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] Stefan Dobrev, Rastislav Královič, and Euripides Markou. Online graph exploration with advice. In Guy Even and Magnús M. Halldórsson, editors, *Structural Information and Communication Complexity*, pages 267–278, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[7] Jorel Elmiger, Lukas Faber, Pankaj Khanchandani, Oliver Richter, and Roger Wattenhofer. Learning lower bounds for graph exploration with reinforcement learning. *1st Workshop on Learning Meets Combinatorial Algorithms, NeurIPS, Vancouver, Canada.*, pages 1–6, 2020.

[8] Klaus-Tycho Foerster and Roger Wattenhofer. Lower and upper competitive bounds for online directed graph exploration. *Theoretical Computer Science*, 655:15–29, 2016. Special Issue on Theory and Applications of Graph Searching Problems.

[9] Robin Fritsch. Online graph exploration on trees, unicyclic graphs and cactus graphs. *Information Processing Letters*, 168:106096, 2021.

[10] A.P. Punnen G. Gutin. *The Traveling Salesman Problem and Its Variations*, chapter 1.2 Simple variations of TSP. Springer, 2002.

[11] Fillipe Goulart. python-tsp (version 0.2.0). `https://pypi.org/project/python-tsp/`.

[12] Cor A.J. Hurkens and Gerhard J. Woeginger. On the nearest neighbor rule for the traveling salesman problem. *Operations Research Letters*, 32(1):1–4, 2004.

[13] Bala Kalyanasundaram and Kirk Pruhs. Constructing competitive tours from local information. *Theor. Comput. Sci.*, 130(1):125–138, 1994.

[14] Emanuel Lazar. Lecture notes from math 170: Ideas in mathematics "Planar Graphs and Euler's Formula". `https://www2.math.upenn.edu/~mlazar/math170/notes05-3.pdf`, Spring 2016. [Online; accessed 10-May-2021].

[15] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. *Theor. Comput. Sci.*, 463:62–72, 2012.

[16] Shuichi Miyazaki, Naoyuki Morimoto, and Yasuo Okabe. The online graph exploration problem on restricted graphs. *IEICE Trans. Inf. Syst.*, 92-D(9):1620–1627, 2009.

[17] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977.

[18] Pascal Schweitzer. Email correspondence April 22^{nd}, 2021. See appendix B.

[19] Pascal Schweitzer. Email correspondence May 3^{rd}, 2021. See appendix B.

[20] ADdV stackoverflow. TSP as you describe it is reducible to real TSP. https://stackoverflow.com/a/64196040, 2020. [Online; accessed 10-May-2021].

[21] Eric W. Weisstein. Graph genus. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/GraphGenus.html, 2020. [Online; Last updated: 18-05-2021].

[22] Wikipedia. Minimum spanning tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Minimum_spanni_tree&oldid=1025043917, 2021. [Online; accessed 6-June-2021].

Appendix

A Table of notation

Table 3: Table of Notation

(u, v)	$\stackrel{\text{def}}{=}$	An edge with incident vertices u and v.
$\lvert(u, v)\rvert$	$\stackrel{\text{def}}{=}$	The weight of the edge (u, v).
$(v_1, v_2, \ldots, v_k)$	$\stackrel{\text{def}}{=}$	The path through the vertices $(v_1, v_2, \ldots, v_k)$.
$\lvert(v_1, v_2, \ldots, v_k)\rvert$	$\stackrel{\text{def}}{=}$	The combined weight of edges in the path $(v_1, v_2, \ldots, v_k)$.
$u \to v$	$\stackrel{\text{def}}{=}$	The path u to v containing only the edge (u, v).
$u \rightsquigarrow v$	$\stackrel{\text{def}}{=}$	A path from u to v possibly containing many edges.
$\frac{Algorithm}{Optimal}$	$\stackrel{\text{def}}{=}$	The competitive ratio of the *Algorithm*.
$\lvert C\rvert$	$\stackrel{\text{def}}{=}$	The combined weight of all the edges in the cycle C.
$\lvert E\rvert$	$\stackrel{\text{def}}{=}$	The combined weight of all edges in the graph.

B Email correspondence with Pascal Schweitzer

Email sent 22.04.21 10:33

Dear Prof. Dr. Pascal Schweitzer

We are currently working on a BSc project under Gerth Brodal in computer science at Aarhus University.
We are writing to you in regards to the article "Online graph exploration: New results on old and new algorithms" 2012.
The article has come to be a large part of our bachelor project, but we are having problems understanding claim 1.
More specifically, we cannot get the following argument to apply in graph A in the attached pdf: "Due to the choice of MST, the edge e is strictly larger than any edge in $C \cap MST$. Otherwise we could replace e with an edge in MST to obtain a smaller minimum spanning tree or to obtain a minimum spanning tree that shares more edges with P."
We hope you have the time to look at the potential counter example to the statement in the pdf, where we have also elaborated on our problem with understanding claim 1.
Do let us know if you need more information, we can also meet online and explain our problems with understanding the claim.

Best regards
Jacob Monsrud & Asger Klinkby
Bachelor students at department of computer science Aarhus University, Denmark

Blocking, Claim 1

Asger Klinkby & Jacob Monsrud

April 2021

In "Online graph exploration: New results on old and new algorithms" (`http://dx.doi.org/10.1016/j.tcs.2012.06.034`) on page 6 lines 5-7 the part of claim 1 we do not understand is:

> "Due to the choice of MST, the edge e' is strictly larger than any edge in C $\cap$ MST. Otherwise we could replace e' with an edge in MST to obtain a smaller minimum spanning tree or to obtain a minimum spanning tree that shares more edges with P."

In the graph below we have $e = (1, 5)$ and $e' = (0, 2)$ fulfilling the criteria of claim 1, contained in a cycle C in $P \cup MST$. The cycle C is:

$$1 \to 3 \to 2 \to 0 \to 4 \to 5 \to 1$$

We do not understand how to argue that e' is larger than any edge in $C \cap MST$. Take the edge $a = (4, 5)$. Replacing e' with a in the MST does not obtain a spanning tree.

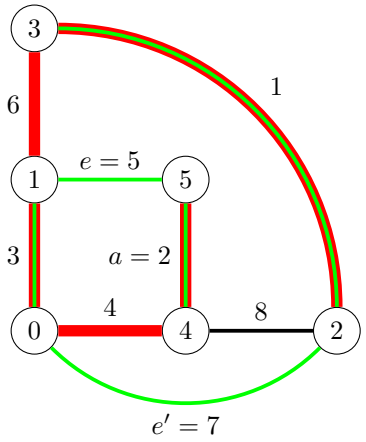


Figure 1: **Graph A**
A complete run of `Blocking`$_\delta$ with $\delta = 0.15$ on the above graph, starting at the node with lable 0. The edges are marked red if in MST, green if in P and black otherwise.

We understand the criteria of claim 1 to be that $e \in P \setminus MST$ is in a cycle, here C. And that $e' \in P \setminus MST$ is the edge in the cycle C which has $|e'| \geq |e|$ and is charged the latest. The minimum spanning tree, MST, must be the one with most edges overlapping with P. In graph **A** the MST is the unique minimum spanning tree. P is the set of edges traversed in line 3 of `Blocking`$_\delta$. The order in which the edges in P are traversed is:

$$(0, 1), (1, 5), (5, 4), (0, 2), (2, 3)$$

When `Blocking`$_\delta$ is called with $y = 4$ the unblocked boundary edges are $e' = (0, 2)$ and $(1, 3)$. Where e' is traversed since it was previously blocked by $(0, 4)$

Figure 26: The attached pdf from our email sent 22.04.21 10:33 containing a potential counter example

Email sent 22-04-2021 18:39

Dear Jacob and Asger,
I think the claim in the paper is wrong, and my recollection is that I assumed that there are no other edges of $P - MST$ on the cycle. Actually the claim was written for some older version of the lemma. In any case, I had at some point written a fix for this.
We choose e' with maximum weight in $P - MST$ and amongst the edges with maximal weight we choose it so it is charged last. Assume now that there is an edge f in (C intersection MST) without P (I think that $-P$ is missing as well in the write-up), that has larger weight than e'. Take out f. Then there is an edge in $C - MST$ that you can put back to get an MST (it does not have to be e'). But its weight is at most $|e'|$.
Does that make sense?
All the best
Pascal

Email sent 03.05.21 12:20

Dear Pascal
We understood your previous email as an alternative proof of claim 1. Based on your fix we have tried to write a proof of claim 1. Our problem is that we cannot see how the fix is sufficient to prove the claim. In the attached pdf you will find a short explanation of our understanding of the problem. We hope you have time to look at the questions in the pdf.
Best regards
Jacob & Asger

Blocking, Claim 1

Asger Klinkby & Jacob Monsrud

May 2021

Using the fix you kindly provided in your email we do not understand how to show that $e'' \notin MST$ which still seem to be needed in order to reach the contradiction in claim 1:

> "We conclude that e'' is not smaller than e' and therefore not in MST. This yields a contradiction to the fact that e' is the edge in $P \setminus MST$ with $|e'| \geq |e|$ (in the fix e' is the largest edge in $P \setminus MST$) that is charged the latest and shows the claim." — page 6 lines 11-13 [1]

In [1] it is stated that the edge e' is strictly larger than any edge in $C \cap MST$. Then $e' \leq e''$ imply $e'' \notin MST$. But in [2] it is only proved that e' is larger than or equal any edge in $C \cap MST \setminus P$. We can expand this proof to show that e' is strictly larger than any edge in $C \cap MST \setminus P$. Still, this does not seem to be enough, as it does not tell us that e' is strictly larger than edges in $C \cap MST \cap P$. Thus $e' \leq e''$ does not seem to imply $e'' \notin MST$.

Do you know a way to show that $e' > C \cap MST \cap P$ or another way to show that $e'' \notin MST$? Or is it a different claim we can prove?

References

[1] Nicole Megow, Kurt Mehlhorn, and Pascal Schweitzer. Online graph exploration: New results on old and new algorithms. *Theor. Comput. Sci.*, 463:62–72, 2012. `http://dx.doi.org/10.1016/j.tcs.2012.06.034`.

[2] Pascal Schweitzer. Email correspondence April 22^{nd}, 2021.

Figure 27: The attached pdf from our email sent 03.05.21 12:20 containing a description of our understanding of the new proof of claim 1

Email sent 03-05-2021 14:50

Dear Jacob and Asger,
here is a better write up that should prove the claim. You basically have to take the last edge of maximal weight.
Does that make sense?
Best
Pascal

A Fix for the claim in Theorem 1

May 3, 2021

Claim. If an edge $e \in P \setminus MST$ is contained in a cycle C in $P \cup MST$, then the cycle C has length at least $(2+\delta)|e|$.

Proof. Suppose otherwise. Without loss of generality we can assume that e has maximal weight among all edges in $P \setminus MST$. Note that no edge of $C \cap MST$ has larger weight than e: otherwise we could remove that edge from the MST and replace it with an edge in $P \setminus MST$ (not necessarily e) to obtain a smaller minimum spanning tree.

Overall e has maximal weight among all edges in C.

Let $e' = (u, v)$ be the edge in $P \cap C$ with $|e'| = |e|$ that is charged the latest.

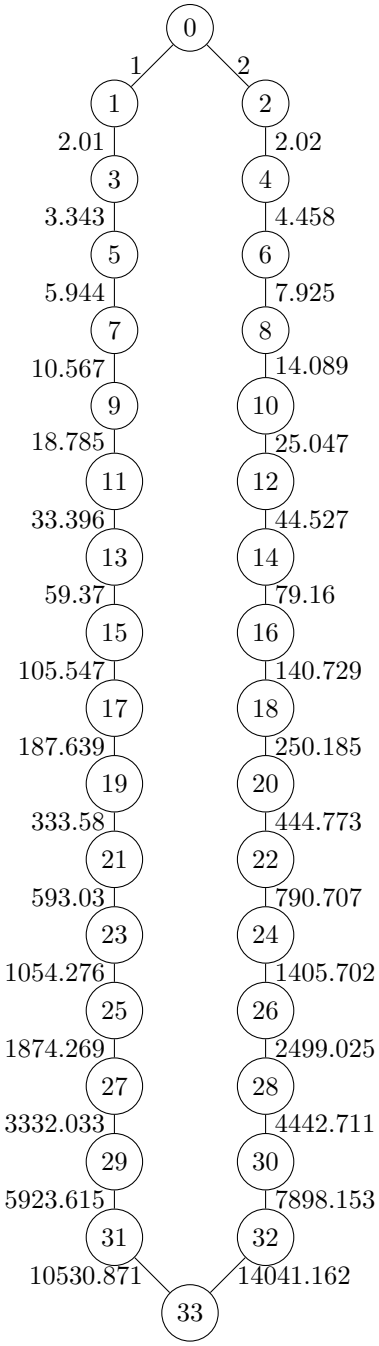
At the time e' is charged, e' is a boundary edge, and therefore not the whole cycle has been explored. Thus there is a boundary edge different from e on the cycle. Moreover at this point in time e' is not blocked. Let e'' be the first boundary edge encountered when traversing $C - e'$ starting from u towards v. Since we assume the cycle has length less than $(2+\delta)|e| \leq (2+\delta)|e'|$ and e' is not blocked, we conclude that $|e''|$ is not smaller than $|e|$. It cannot be that $|e''| = |e|$ and $e'' \in P$ due to the choice of e' being charged last among such edges. If $|e''| = |e|$ and $e'' \notin P$ then $e'' \in MST$ and then we can remove e'' and replace it with some edge from $C \setminus MST$ to obtain an MST of at most the same weight that shares more edges with P. Finally we already know that $|e''| > |e|$ is impossible since e is of maximal weight in C. □

Figure 28: The attached pdf from Pascals email sent 03.05.21 12:20 containing "a better write up that should prove the claim".

Email sent 04-05-2021 18:11

Dear Pascal
Thank you very much.
It was exactly the part "It cannot be that $|e''| = |e|$ and $e'' \in P$ due to the choice of e' being charged last among such edges." that we were missing. Understanding this was important for our project, so we really appreciate your help.
Best regards
Jacob & Asger

C Graph for which $\texttt{Blocking}_\delta$ obtains a competitive ratio of ≈ 5.25



0 is the origin vertex. Numbers are rounded to three decimal places.
Total cost of $\texttt{Blocking}_\delta$: 294651.24
Total optimal offline cost: 56161.65
Competitive ratio: 5.246