Universität Stuttgart

Institut für Formale Methoden der Informatik
Abteilung Algorithmik

Universitätsstraße 38
70569 Stuttgart

Masterarbeit

Barrier Resilience Problems

David Kruse

Studiengang: M.Sc. Informatik

1. Prüfer: Prof. Dr. Stefan Funke

2. Prüfer:

Betreuer: Prof. Dr. Stefan Funke

begonnen am: 08.07.2023

beendet am: 08.01.2024

Abstract

The barrier resilience problem can be stated as follows: Given two points s and t and a set of unit disks D in 2D-space, what is the minimum number of disks k that a curve connecting s and t must cross. An example of a problem instance together with a curve c solving the problem optimally is given in fig. 1. Observe how a disk must be crossed twice in order to cross the least disks possible. It is currently unknown if there exists a polynomial time algorithm that can find the optimal solution k for any problem instance.

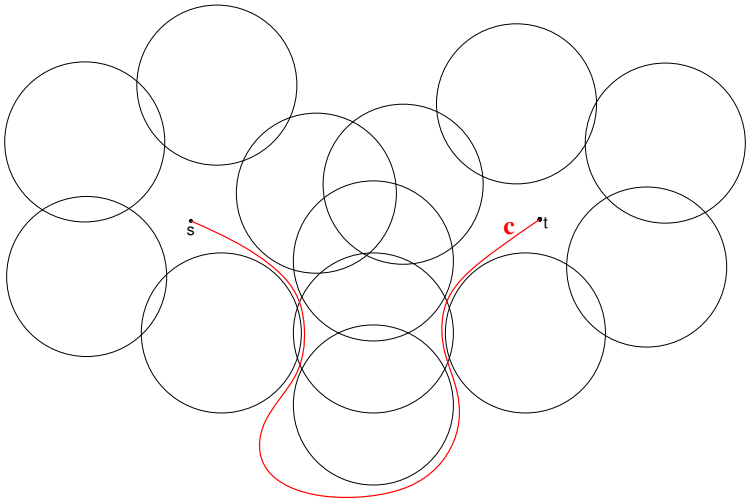


Figure 1: An example where the barrier resilience is 1

This work aims to give an intuition on why it is difficult to solve and at the same time show some restrictions on the complexity, which prevent an easy reduction of NP-hard problems. As a result, neither a polynomial time algorithm nor a reduction of an NP-hard problem is currently known or discovered in this work. Existing ideas and approximations are explained and expanded upon, whilst also introducing novel algorithms. One technique used is linear programming, which was not previously considered for solving the barrier resilience problem. In contrast to previous works, the algorithms offer both upper and lower bounds for k, which are very useful in practice in restricting k to a small range of values, even restricting this range to only the optimal value k in most randomly generated instances. Existing and novel algorithms are implemented in C++. The different approximations are compared both theoretically and practically, with their performance in terms of runtime and accuracy being measured on random instances.

Abstract Deutsch

Das 'Barrier Resilience'-Problem lautet wie folgt: Gegeben sind zwei Punkte s und t und eine Menge von Einheitskreisen D im zweidimensionalen Raum. Die Frage ist nun, was die geringste Anzahl an Kreisen k ist, die eine Kurve von s nach t schneiden muss. Ein Beispiel einer Probleminstanz mit einer Kurve c, die das Problem optimal löst, ist in fig. 2 dargestellt. Man beachte, wie ein Kreis zweimal betreten werden muss, um die optimale Lösung zu erhalten. Es ist zurzeit unbekannt, ob es einen Algorithmus gibt, der die optimale Lösung k für jede Probleminstanz in Polynomialzeit findet.

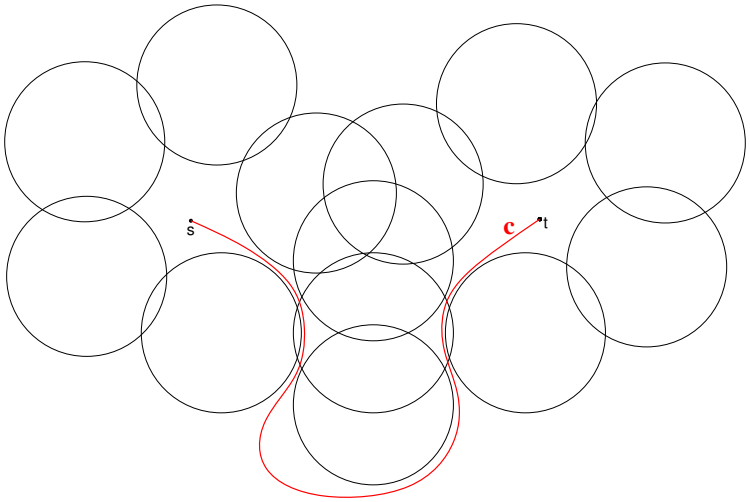


Figure 2: Ein Beispiel in dem die barrier resilience 1 ist

In dieser Arbeit wird eine Intuition vermittelt, weshalb das Problem schwer zu lösen ist. Gleichzeitig werden Restriktionen aufgezeigt, die eine einfache Reduktion eines NP-Schweren Problems verhindern. Deshalb wird auch in dieser Arbeit weder ein Algorithmus gefunden, der das Problem in Polynomialzeit löst, noch gelingt eine solche Reduktion. Vorherige Ideen und Algorithmen werden erklärt und erweitert, aber auch neue Algorithmen werden eingeführt. Eine Technik hierbei ist die lineare Programmierung, welche bisher noch nicht zur Lösung des 'barrier resilience' Problems herangezogen wurde. Im Gegensatz zu vorherigen Arbeiten liefern die vorgestellten Algorithmen Ober- und Untergrenzen, die k in der Praxis sehr gut einschränken und in den meisten zufällig generierten Instanzen sogar nur den optimalen Wert k zulassen. Existierende und neue Algorithmen werden in C++ implementiert. Sie werden theoretisch als auch praktisch verglichen, wobei ihr Ergebnis im Bezug auf Laufzeit und Approximationsfaktor an zufällig generierten Instanzen gemessen wird.

Contents

1 Introduction **6**

2 Fundamentals **8**
2.1 Barrier resilience definition . . . 8
2.2 Resolving degeneracy . . . 8
2.3 Decision problem . . . 9
2.4 Approaching a geometric optimization problem . . . 10
2.5 Coverage graph . . . 10
2.6 Arrangement graph . . . 11
2.7 Notation . . . 12
2.8 Linear Programming . . . 12

3 Intuition **14**
3.1 How often must a curve enter the same disk . . . 14
3.2 Why the problem is hard . . . 16

4 Previous work **19**
4.1 Open belt region . . . 19
4.2 Closed belt region . . . 20
4.3 Beyond unit disks . . . 21

5 Searching a path through the arrangement **22**
5.1 Barrier Thickness . . . 23
5.2 ILP formulation . . . 24
5.2.1 LP Relaxation . . . 25
5.2.2 Practical analysis . . . 29

6 Working with the coverage graph **31**
6.1 Intuition . . . 32
6.2 Barrier resilience in the coverage graph . . . 33

7 Ring problem **35**
7.1 Rings . . . 35
7.1.1 Extended graph representation . . . 35
7.1.2 Deciding whether a ring separates s and t . . . 37
7.1.3 Non-intersecting ring . . . 38
7.1.4 Minimal ring . . . 38
7.2 Finding a ring . . . 40
7.2.1 Disk view . . . 42
7.2.2 Graph view . . . 44
7.2.3 Disregarding the 2D plane . . . 45
7.3 Problem definition . . . 47
7.4 Naive algorithm . . . 48
7.5 Ring problem as an ILP . . . 48
7.5.1 LP relaxation . . . 49
7.5.2 Seperation oracle . . . 49
7.5.3 Using the LP result . . . 51

7.6 Node-disjoint rings 52
7.6.1 Bounding the number of node-disjoint rings 53
7.6.2 Greedy algorithm 55

8 Benchmark of algorithms **57**

9 Conclusion **61**

10 Appendix **64**

1 Introduction

Digital security systems are widespread, using sensors to detect potential intruders in buildings, estates, or even national borders. For example, there are hundreds of surveillance cameras along the US-Mexico border, that have a 360° view and autonomously detect migrants miles away, leading to the term 'virtual wall' [1]. To measure the penetrability of such an arrangement of sensors, Kumar et al. introduced the concept of barrier resilience in 2005 [14] under the name 'barrier coverage'. It measures the minimum number of sensors whose sensing area must be crossed in order to traverse such a barrier. In contrast to the meaning of coverage in mobile networks, it is not important how many sensors cover every point of a region, instead the number of distinct sensors crossed when going through the barrier is summed. This is probably why the term 'barrier resilience' prevailed in subsequent papers on the topic.

There are different types of barrier resilience. In the version of the problem considered in this work, sensors are modeled as unit disks that detect someone entering their sensing disk, regardless of the duration of the visit or the proximity to the sensor. Other versions of the problem use different shapes of sensing areas or use probabilistic models for when the sensors actually detect an intruder. When the sensors are arranged in a long strip that an intruder wants to cross, as with the US-Mexico border, the problem can be solved easily. Another version considers a donut-shaped area where the sensors lie, with the intruder trying to get from the outside of this area to the inside, as with a protected estate. In this work, the more general version is considered, where the intruder wants to get from some point s to another point t, with sensors being placed anywhere in the 2D plane. The donut version and the general version seem to be equally hard, with it being unknown so far whether there exists a polynomial time algorithm that computes the barrier resilience or if the problem is NP-hard [4].

An example of a problem instance is shown in the abstract (fig. 1). Observe how the optimal solution is forced to enter one sensor twice. When an optimal curve is not forced to enter a sensor twice, the problem becomes trivial to solve, which will become obvious later.

The practical relevance of this version of the problem is limited because the sensor model is very unrealistic and restrictive. Real-world examples can also usually be modeled as a strip of sensors, for which it is easy to compute the resilience for. Even for the general version of the problem, real instances will not have the constructed arrangements that make the problem hard to solve. In these cases, approximation algorithms manage to solve the problem perfectly. What makes this problem so fascinating, however, is how simple it can be stated and understood even by people without a background in computer science or mathematics. Despite this, the existence of a polynomial-time algorithm has not been proven or disproven since the problem was first proposed in 2005 [15].

The first chapter of this thesis presents the fundamentals of the problem and introduces the graph representations and tools that are used to solve it. The next chapter aims to convey an intuition for the problem by exploring the properties that make it difficult to solve. After that, previous works are presented, which use different approaches to approximate barrier resilience and also consider some different versions of the problem. The main part of this work is divided into two parts. First, the arrangement is looked at and a path through it is searched. Linear programming is utilized for this, which has not been considered for this purpose before. The results of this are poor and do not advance the state of the art. The second part looks at the problem more abstractly, and the so-called 'ring problem' is introduced. This is a novel problem formulation equivalent to the barrier resilience problem. It can be approximated very well and efficiently. A solution based on linear programming is presented, for which no example could be found where it does not return the correct solution. Therefore, it could be the sought-after polynomial-time algorithm that solves the barrier resilience problem optimally. However, this could not be proven. In the end, the different

algorithms will be analyzed theoretically and benchmarked on random instances. In the conclusion, the contributions of this work are summarized and open questions that might help to finally solve the barrier resilience problem are raised.

2 Fundamentals

This chapter will start by defining the barrier resilience problem. It will then explain what the central question regarding this problem, whether there is an efficient algorithm or the problem is NP-hard, means. The general approach for dealing with such problems is explained. Two graphs representing the barrier resilience problem are introduced, which are used in previous works as well as this one. Finally, the notation when working with the graphs is clarified, and the technique of linear programming, which will be used extensively, is briefly introduced.

2.1 Barrier resilience definition

The problem instance of the barrier resilience problem is defined in the 2D plane, which contains a set of unit disks D and the points s and t. A unit disk is a disk with a radius of 1. The problem instance contains the centers of all disks $D = \{(x_1, y_1), (x_2, y_2), ..., (x_{|D|}, y_{|D|})\}$. Additionally, the start point $s = (x_s, y_s)$ and target point $t = (x_t, y_t)$ are given. So a problem instance is defined by the tuple (D, s, t).

There are two equivalent ways of stating the problem of barrier resilience, one focusing on the curve c, while the other focuses on the set of disks I:

Definition 1. *Barrier resilience problem:*

- *The barrier resilience problem asks what the minimum number of disks k is that a curve c must cross in order to go from point s to point t.*
- *Find the minimum subset of disks $I_{min} \subseteq D$, $|I_{min}| = k$, so that a curve c going from s to t that does not cross any disk $d \in D \setminus I_{min}$ may exist.*

The variable k will consistently represent the optimal solution to the barrier resilience problem for a given instance throughout this work. The variable I will always refer to the subset of disks that are removed from the 2D plane, in order to allow a curve from s to t to exist. The disks that are part of this set I are sometimes called 'deactivated', while the disks that are not part of this set are called 'active'. This corresponds to the motivation of the problem, where the goal is to find the minimum number of sensors that need to fail in order to allow an intruder to enter an area undetected. s and t are called separated if there is no curve between the points that does not intersect any disks.

The first part of this work focuses on finding a curve through the arrangement that traverses as few disks as possible, as outlined in the first definition. In the second part, the problem is looked at in a more abstract way, meaning that there is no equivalent to a curve through the 2D plane anymore, with algorithms focused on finding the set I_{min}, so that s and t are no longer separated by any active disks.

2.2 Resolving degeneracy

Degeneracy can occur in three ways that are handled easily:

- A disk covers s or t
  $\rightarrow$ Disks that cover s or t can be disregarded, as they have to be crossed in any case
- Two or more disks might have the same center
  $\rightarrow$ Move the disks by an infinitesimal amount randomly

- Two or more disks are tangent, meaning their centers are a distance of exactly 2 apart
  → Scale all disks to be infinitesimally smaller or larger, depending on whether tangential disks should intersect or not. The intended behavior has not been defined.

Therefore, these degenerate cases will not be considered further, and it is always assumed that no disk covers s or t, the distance between the center of disks is never exactly 2, and disks never have exactly the same center. In a lot of figures, disks are colored gray, in order to represent multiple disks that are stacked on top of each other. Their centers are also infinitesimally close together without being the same.

2.3 Decision problem

The barrier resilience problem can also be stated as a decision problem:

Definition 2. *Barrier resilience as a decision problem:*
Given an integer l, is there a curve c from s to t that crosses at most l disks.

The decision problem is equivalent to the optimization problem in the sense that an algorithm to solve one of the problems can be trivially used to solve the other one: When the optimal solution k of the optimization problem is known, the decision problem can be solved by checking if $l \leq k$. In the reverse direction, the optimal solution k can be found with a linear number of calls to the decision problem by trying every $l \in \{0, 1, ..., |D|\}$.

The decision problem is in a class called NP, which is the set of all decision problems, for which a proof, also called certificate, can be given if the answer to the decision problem is yes, and this proof can be verified in polynomial time [11]. For the barrier resilience problem, this proof is the set of disks $I \subseteq D$, that has a size of $|I| \leq l$, and allows for a curve from s to t to exist when the disks in I are deactivated. How to verify such a set I will become obvious in the following chapters. There is a naive brute-force algorithm for every problem in NP, which simply runs the verifier for every possible proof $I \in I^*$. This will always find the right answer to the problem, although the number of tries $|I^*|$ is exponential in the size of the input.

Now the question is raised if the problems can be solved efficiently. "A minimal requirement for an algorithm to be considered "efficient" is that its running time is bounded by a polynomial function of the input size: $O(n^c)$ for some constant c, where n is the size of the input." [11] The size of the input $n = |D|$. The set of decision problems solvable in polynomial time is called $P \subseteq NP$. To prove that the problem is solvable in polynomial time, it suffices to develop such an algorithm and prove its correctness and runtime. There are several *NP-Hard* problems that are believed to not be solvable in polynomial time, but even for those problems, there is no proof that they cannot be solved in polynomial time [11]. So it is unlikely to prove that the barrier resilience problem cannot be solved in polynomial time directly. Instead, in order to prove that there is probably no polynomial time algorithm for a problem, one of these *NP-Hard* problems is reduced to it. This reduction must present an algorithm that takes any instance of an *NP-Hard* problem and turns it into an instance of the barrier resilience problem, so that the answer to the barrier resilience decision problem is *YES*, if and only if the answer to the *NP-Hard* problem is *YES*. If this algorithm transforms the *NP-Hard* problem instance into a barrier resilience instance of polynomial size in polynomial time, the barrier resilience problem is *NP-Hard*. As *NP-Hard* problems are generally assumed not to be solvable in polynomial time [11], the barrier resilience problem would also be assumed not to be solvable in polynomial time by extension.

In conclusion, the barrier resilience problem is in NP, but whether it is *NP-Hard* or it is in P remains open.

2.4 Approaching a geometric optimization problem

There are many geometric optimization problems that are more or less similar to the barrier resilience problem. M. Bern and D. Eppstein summarize the approach when solving various geometric problems as follows: "First, geometric problems can often be reduced to combinatorial optimization problems involving graphs or set systems, only the resulting problem instances are usually quite special. [...] The constrained nature of geometric problems often helps in approximation." [3]. This describes exactly what is done in this work, with graphs being introduced in the following two sections that represent the problem instance. The *ring problem*, which will be introduced later, is essentially a set system with which the barrier resilience can be computed. During all of this, the geometric constraints of the problem are used in order to bound the result of the approximations in terms of k. These constraints can also mean that coming up with an exact solution to the problems that barrier resilience is reduced to can be done efficiently, even if the underlying problem is *NP-Hard* in general. Because of the geometric constraints of geometric problems, proving them to be *NP-Hard* is very difficult [3].

Going beyond approximations, even for NP-Hard problems, there is still interest in coming up with exponential time algorithms for solving them exactly [20]. Even if different algorithms all take exponential time, there can be a huge difference between their runtimes, and they can be significantly faster than using brute force, which is not captured in traditional complexity theory. Efficient algorithms might be sufficient for solving moderately sized instances. Therefore also exact algorithms that might take exponential time are analyzed in this work in terms of their runtime.

2.5 Coverage graph

The coverage graph $CG = (N, E)$ was introduced in the initial paper proposing the barrier resilience problem by Kumar et al. [15]. Disks are represented by nodes $n \in N$ and undirected edges $e \in E$ are inserted between any nodes whose respective disks intersect:

Definition 3. *Coverage graph* $CG = (N, E)$ *with*
N *for every disk* D
$E = \{\{n_1, n_2\} \in N^2 | n_1 \neq n_2 \wedge |\overline{n_1 n_2}| \leq 2\}$.

An example is shown in fig. 4 (b). Note that the node variables n_1 and n_2 represent the center of the corresponding disks in the edge definition. This notation will often be used depending on the context. Similarly, an edge can represent the segment between the centers of the two adjacent nodes $e = \overline{n_1 n_2}$ in the plane. The number of nodes $|N| = |D|$ and the number of edges is bounded by $|E| < |D|^2$. Constructing the graph can be trivially done in $O(|D|^2)$ time, by calculating the distance between every pair of nodes.

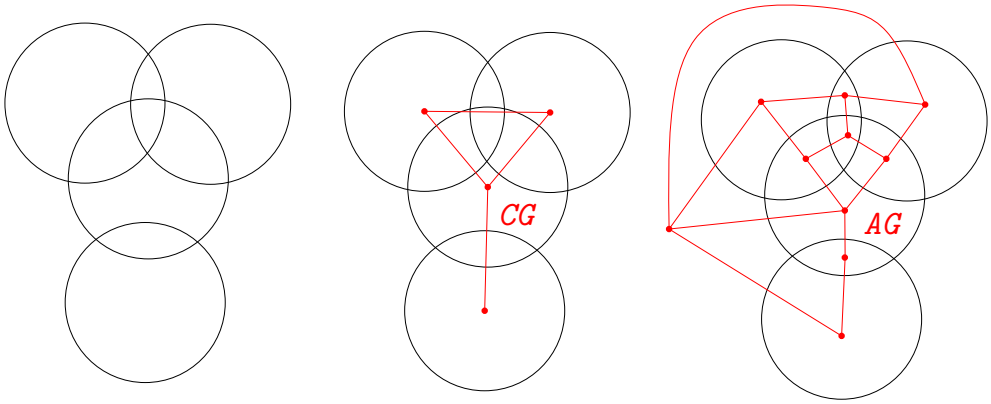
(a) s and t are not separated (b) s and t are separated

Figure 3: The coverage graph CG does not represent the problem sufficiently

In general, this graph does not represent the problem sufficiently, as it is not clear if s and t are separated in a graph CG, as shown in fig. 3, where the same graph separates s and t on the left, but does not on the right. Therefore, some geometric information needs to be included in the problem instance. In most algorithms and visualizations, the coordinates of nodes N, as well as s and t are preserved. An extension of CG that replaces the need to keep the coordinates of nodes is presented later.

When a set of disks I is deactivated, the set of associated nodes $I \subseteq N$ is deactivated in CG, resulting in the graph CG_{-I}, where nodes I together with their adjacent edges are deleted:

Definition 4. $CG_{-I} = (N \setminus I, \{e \in E | e \cap I = \emptyset\}$



(a) arrangement of 4 disks (b) coverage graph CG (c) arrangement graph AG

Figure 4: The different graph representations of a problem instance

2.6 Arrangement graph

Another way of looking at the problem is to consider the arrangement of the disks. The disks divide the 2D plane into cells that contain a continuous area. The dual graph of the arrangement contains a node $n \in N$ for every cell and has an edge $e \in E$ between any two cells that are neighbors in the arrangement, meaning that they share a border. The dual graph is used in [13, 6, 2] to approximate k. As the term dual graph might be misleading in the larger context of barrier resilience, this graph is referred to as the arrangement graph AG in this work. In addition to the graph, the cells that contain s and t are saved in the variables n_s and $n_t \in N$ respectively.

Definition 5. *Arrangement graph* $AG = (N, E)$ *with*
N *for every cell in the 2D plane*
$E = \{\{n_1, n_2\} \in N^2 | n_1 \neq n_2 \wedge n_1, n_2$ *share a border*$\}$

An example is shown in fig. 4 (c). The number of edges and nodes is limited by $|N| = O(|D|^2)$ and $|E| = O(|D|^2)$. An edge between two cells requires at least one continuous part of the rim of a disk. The number of parts that a disk rim is partitioned into is equal to 1 if there are no intersections, otherwise it is equal to twice the number of disks that intersect this disk. Therefore, there are $|E| \leq 2 * |D|^2$ edges in AG. The number of edges limits the number of nodes to $O(|D|^2)$ as well. How to construct this arrangement graph is outlined in appendix B.1, with the algorithm taking $O(|D|^2 log(|D|))$ time.

As with the coverage graph, the arrangement graph does also not contain enough information by itself in order to solve the problem optimally. The arrangement graph will be used as stated for an approximation of k, with additional information later being added in order to solve the problem optimally.

2.7 Notation

Before working with the graphs, the notation is settled, based on the definitions given by Williamson in [19]. Sequences, also called lists, contain elements in an ordered collection, which may contain the same element multiple times at different positions. They will be defined using round brackets:
$l = (1, 2, 3, 4, ...)$
The element of a sequence l at index i is denoted by $l[i]$, where zero-based indexing is used, so the first element of a sequence is $l[0]$. The size of the sequence is denoted by $|l|$ and the set of elements contained in the sequence is denoted by $set(l)$. This set contains duplicate elements only once and knows no ordering.

A lot of literature, such as [19], refers to nodes in a graph as vertices, however this thesis will consistently call them nodes. A curve always refers to a curve in the 2D plane, which is not necessarily attached to any graph. In a graph, a walk is a finite sequence of edges that join a sequence of nodes. They are named ω and represented by the sequence of nodes that are walked along:
$\omega = (n_0, n_1, ..., n_{length(r)})$
The associated sequence of edges is called $E(\omega)$:
$E(\omega) = (\{\omega[0], \omega[1]\}, \{\omega[1], \omega[2]\}, ..., \{\omega[length(r) - 1], \omega[length(r)\})$
Obviously, these edges have to exist in the graph for the walk to be valid. The length always refers to the number of edges that a walk 'travels along', which is one less than the length of the sequence of nodes:
$length(\omega) = |E(\omega)| = |\omega| - 1$
A walk is closed if its last node is the same as its first node:
$\omega[0] = \omega[length(\omega)]$

A walk may traverse the same nodes multiple times and go along the same edge multiple times. In contrast, a trail is a walk in which all edges are distinct. In a path, not only are the edges distinct, but the nodes are all distinct as well. A path will be represented by the symbol π. A closed path is also called a cycle.

2.8 Linear Programming

Linear programming is a technique that has not previously been considered for solving the barrier resilience problem, but will be extensively used in this work. This section defines linear programming based on the book "Linear programming" by V. Chvátal [8]. Firstly, the term 'linear function' is defined. Given variables $x_1, x_2, ... \in \mathbb{R}$ and constants $c_1, c_2, ... \in \mathbb{R}$, a linear function is defined as follows:

$$f(x_1, x_2, ...) = c_1 * x_1 + c_2 * x_2 + ...$$

A linear equation assigns a linear function a real value $b \in \mathbb{R}$, while a linear inequality gives an upper or lower bound for its value:

$$\begin{aligned} f(x_1, x_2, ...) &= b \quad \text{linear equation} \\ f(x_1, x_2, ...) &\geq \text{or} \leq b \quad \text{linear inequality} \end{aligned}$$

Linear equations and linear inequalities are referred to as linear constraints. Now, a linear programming (LP) problem is the problem of maximizing or minimizing a linear function, which will be called the objective function, while being subject to a finite number of linear constraints. A solution of this linear program assigns a real value to every variable $x_1, x_2, ...$ If all constraints (equalities and inequalities) hold for the assignment, the solution is *feasible*. The feasible solution that maximizes / minimizes the objective function is called the *optimal* solution. The value of the objective function of the optimal solution is called the *optimal value* of the linear program, which will sometimes be denoted as the variable lp. The following is an example of a linear program written in standard form, which is how all linear programs will be stated in this work.

$$\begin{aligned} \text{maximize} \quad & 3x_1 - x_2 \\ \text{subject to} \quad & x_1 + x_2 \leq 2 \\ & -2x_1 - 2x_2 \leq -10 \\ & x_1, x_2 \geq 0 \end{aligned} \tag{1}$$

Finding the optimal solution is done in practice by the simplex method, which might take exponential time in worst-case scenarios. In theory however, the *ellipsoid method* provides an algorithm that can solve any LP problem in polynomial runtime. The great thing about the ellipsoid method is that it will find the optimal solution in polynomial time even if there is an exponential number of constraints. It can do this by starting with an empty set of constraints. First, a feasible solution given the current constraint set is calculated. Now a separation oracle needs to return a constraint that is violated by this assignment of the variables, which is then added to the linear program. This is repeated until no constraint is violated by the solution found. Even if there is an exponential number of constraints, the optimal solution is bounded by as many constraints as there are variables in the problem. Therefore, the ellipsoid method can work, even though it only adds a polynomial number of constraints to the problem [7].

Linear programming is very useful for solving various optimization problems. Integer linear programs (ILPs) add an integer constraint to the variables, which forces them to take on integer values. This allows integer linear programs to represent a wide range of problems in computer science and beyond. For example, a lot of *NP-Hard* problems can be modeled as ILPs, which already proves that solving ILPs is generally *NP-Hard*. Despite this, some ILPs are solvable efficiently enough to be useful. For this, the formulation of the ILP can make a big difference, even if the different formulations define the same problem [17]. The LP relaxation drops the integer constraints of the ILP, resulting in a linear program in which the ILP solution is obviously still feasible, but solutions that deliver a more optimal objective function value that were not feasible in the ILP are allowed [17]. This relaxation of the problem might still be useful though: "Relaxation strategies based on linear programming have played a unifying role in the construction of approximation algorithms for a wide variety of combinatorial optimization problems" [7].

This work will use different integer linear programs to solve the barrier resilience exactly and their LP relaxation to approximate it. The GNU Linear Programming Kit (GLPK) is used to solve them, which supports both ILPs and LPs.

3 Intuition

This chapter aims to give an intuition for why the barrier resilience problem is difficult to solve optimally. At the same time, the geometric restrictions of working only with unit disks limit this perceived difficulty. The introduction already gave an example where a curve might have to cross a disk twice for an optimal solution. This raises the question of what the maximum number of times a disk has to be entered in order to achieve an optimal solution is.

3.1 How often must a curve enter the same disk

Theorem 1. *For an optimal solution, disks need not be entered more than twice, except when start and target points are close, in which case disks may have to be entered thrice.*

This theorem was already proven in [2]. The main idea of the proof is repeated here, with the details being left out. The example where a disk actually needs to be entered three times has not been presented in the paper, so it is shown here.

If disks must not be entered more than twice / thrice, this means that for a solution I, there exists a curve from s to t that does not enter any active disks and enters deactivated disks at most twice / thrice. Therefore, in order to prove the theorem, it suffices to construct such a curve. For this, the so-called Dubins path will be considered as the curve. The Dubins path from s to t is simply the shortest possible curve between the two points, which does not intersect any active disk. As a curve between s and t that does not intersect any active disk must exist for I to be valid, this shortest curve must also exist. It consists of straight lines that hit disks tangentially and then go along the rim of these disks until they leave the rim again on a tangential trajectory.

To prove that this curve enters no disk more than twice / thrice, every disabled disk d and its surrounding active disks are considered separately. This can be done, as the curve does not enter any active disk at all, and the Dubins path is entirely defined by the active disks.

For this purpose, the deactivated disk d is partitioned into cells c_1, c_2, ... Cells represent coherent areas within the disk, so that any two points within a cell can be connected without leaving d, while no two points of different cells c_1, c_2 can be connected without leaving d. The area of a cell is defined by the rim of d and some active surrounding disks. Only cells that have at least some of the rim of d exposed are considered, as the cell cannot be reached by a curve otherwise. An example of a disk d that is partitioned into three exposed cells is shown in fig. 5

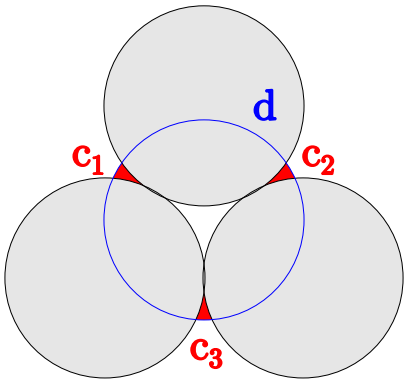
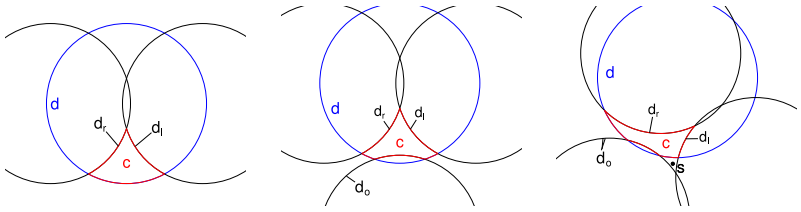


Figure 5: The blue disk d is partitioned into three cells c_1, c_2, c_3 (red) by the active disks (gray)

Lemma 1. *The Dubins path never enters a cell twice.*

Proof. It is proven in [2] that when two points of a disk can be connected by going through the disk, the way through the disk is always shorter than leaving the disk and entering it again. As two points within a cell can be connected without leaving the disk by definition, the Dubins path will never leave a cell and then enter it again. □

Now, the goal is to limit the number of cells that a disk can have. Obviously, the path between a cell and all other cells of d needs to be blocked by at least two disks, which are called d_l and d_r in fig. 6. Look at the following type of cells: In the 'open cell' (a), the Dubins path will not enter d at all, as there is no disk forcing the shortest path to traverse the cell. In a proper cell (b), the curve is forced to enter d by an active disk called d_o, which covers part of the rim of the cell. d_o is not allowed to intersect d_l or d_r, as the curve would not be able to enter the cell on one side and leave it on the other side otherwise. This constraint is lifted in the 'start cell' (c), as the curve has its start or target point right next to the cell. This way, the outer disk forcing the curve to enter d may intersect either d_l or d_r.



(a) Open cell (b) Proper cell (c) Start cell

Figure 6: The 3 relevant cell categories

Based on the geometry of the problem, the maximum number of cells can be limited to 3. When there are 2 proper cells, there is no space for a start cell anymore, so this configuration is impossible. Therefore, a disk must only be crossed three times if it has 1 proper cell and 2 start cells, which requires both s and t to be close to d. This construction, where the blue disk d has two start cells and one proper cell, can be seen in fig. 7. Note how precise the instance needs to be constructed in order for d to have these three cells, as the margins are very tight.

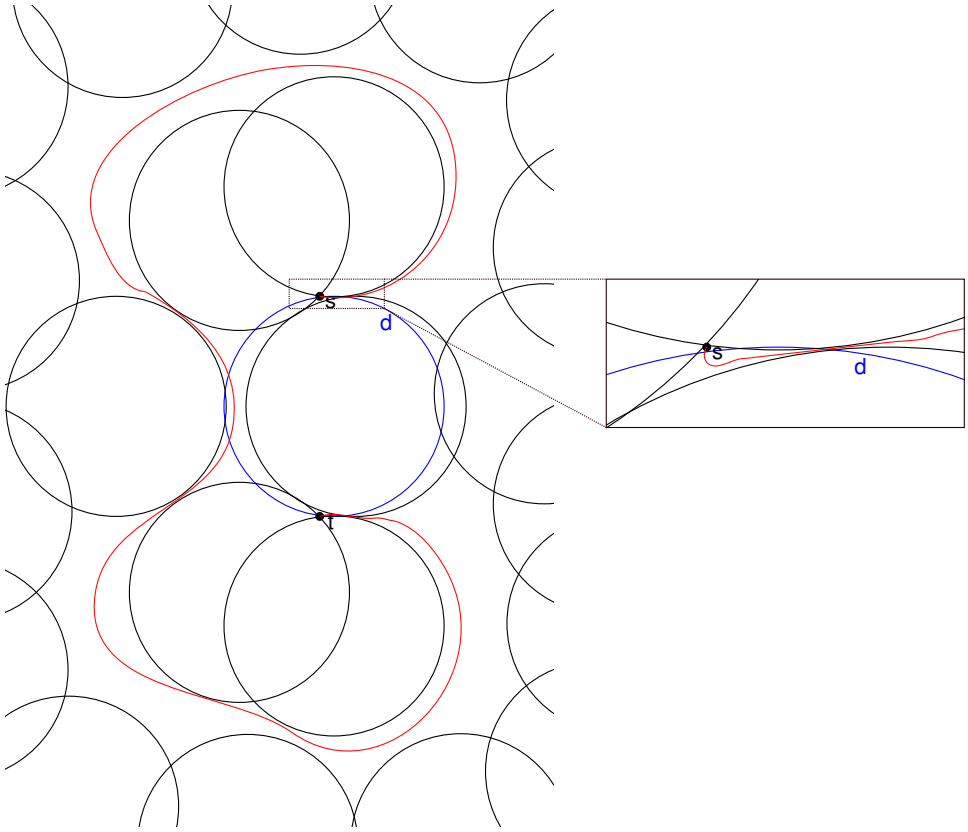


Figure 7: The blue disk d has to be entered three times by the optimal curve (red) with $k = 1$.

3.2 Why the problem is hard

The fact that a disk must be entered multiple times might not seem complicated at first. When constructing a curve through the arrangement, it can simply be checked if a disk has been entered previously. But constructing a curve through the arrangement 'step-by-step' does not work for the following reason: Consider an optimal curve between s and t called c_1. Now take an arbitrary point p on this curve. Consider the part of the curve between s and this point p and call it $c_{sp,1}$. Now there might be a curve $c_{sp,2}$ from s to p that crosses fewer disks than $c_{sp,1}$, making it more optimal for this subpart of the curve c_1 (fig. 8).

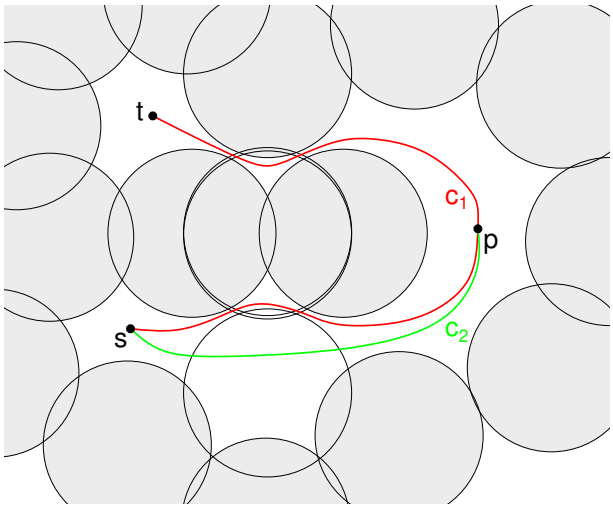


Figure 8: While c_1 is optimal from s to t, c_2 is better for the way between s and p. Gray disks represent 3 stacked disks

There can actually be an exponential number of distinct curves between s and some point p, with only one of them actually leading to an optimal curve going from s through p to t. The construction in fig. 9 shows an arrangement where there are 2^3 curves from s to p. As the optimal curve continues until it reaches t, it is forced to cross the disks that were traversed between s and p again. Due to this, only one curve between s and p actually leads to an optimal curve when continuing the curve to t. It is impossible to know which one of the $s-p$ curves this is without looking 'across the barrier'.

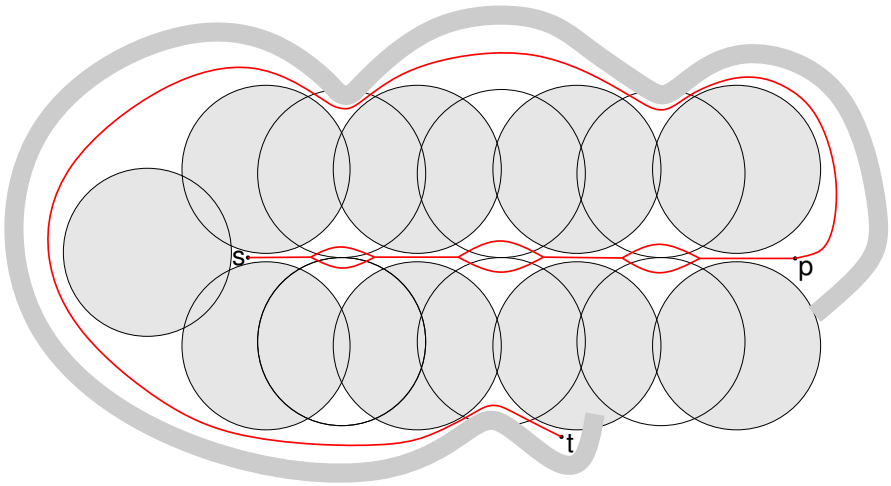


Figure 9: There are 2^3 different curves from s to p, with only one of them leading to the optimal curve $s-p-t$ with $k=3$. Gray disks represent 4 stacked disks, the gray band also represents a barrier consisting of many disks

This leads to an idea for reducing another problem that might be NP-Hard to the barrier resilience problem: The curve between s and p can be seen as a bit string, where the curve going through the upper disk represents a 1 and going through the lower disk represents a 0. This bit string can be viewed as the certificate of a problem in NP, with there being some verifier logic that enforces certain conditions on the bit string. These constraints are constructed by forcing the curve to traverse certain disks again, either sequentially, which represents an AND on the associated bits, or the curve can be forced to go through either one of two disks and be reconnected after that, representing an OR on the associated bits. The barrier resilience solver would then have to 'guess' the bit string, so that the curve can pass through the verifier logic without entering any additional disks. The problem with this idea is that the verifier logic is very limited, so it probably cannot represent the verifier of any NP-Hard problem. The well-known 3-SAT problem, for example, requires the logical disjunction of three bits, but it is impossible to give the curve the option to traverse any one of three disks in a construction as the one shown in fig. 9.

4 Previous work

There have been a lot of papers dealing with this problem. The best fixed approximation factor that any algorithm delivers gives a result $res : k \leq res \leq 1.5k$ [6]. This algorithm searches a path through the arrangement graph AG and is explained in the following chapter. An existing approximation algorithm that directly gives a lower bound for k is unknown to me. The bound $res \leq 1.5k$ can obviously be used as a lower bound for k, but the algorithms introduced in this work directly calculate lower bounds for k that are significantly closer to k in practical examples.

A $1 + \epsilon$ approximation was developed for the case where the maximum number of overlapping disks is bounded in the paper [13]. This maximum number of disks that overlap at any point in the plane is called Δ. If $\Delta = O(1)$, a brute force algorithm calculates the optimal path between cells in the arrangement up to a given distance between the cells. Now, the shortest path between the start and target node in AG can be found using these shortcuts. Although restricting Δ might make sense for practical applications, the introduction already explained how real world uses of barrier resilience might not be that complicated to begin with. Worst-case examples where approximation algorithms perform poorly will often raise the degree of overlap Δ linearly when scaling the resilience k of the arrangement. This work does not consider such restrictions on the distribution of the disks.

In the following, two constrained versions of the barrier resilience problem are shown, which are discussed in the literature.

4.1 Open belt region

A constrained version of the problem is considered in the initial paper [14], where the domain is bounded by a rectangle called an open belt, so that the centers of all disks lie within this rectangle. The goal now is to find a path from one side of the rectangle to the opposite side, crossing as few disks as possible. For this problem, the coverage graph CG is extended with two nodes n_l, n_r, which are connected to all disks intersecting the left and right boundary of the belt respectively, see fig. 10. Kumar et al. prove that the resilience is equal to the minimum vertex cut, that cuts every possible path from n_l to n_r [14]. This is rather intuitive, as every path from n_l to n_r represents a barrier when crossing the rectangle. Therefore, one node of every path needs to be deactivated. Finding this minimum vertex cut can be done in polynomial time, with a recent paper decreasing the expected time needed for finding it to $O(|D|^{3/2} * polylog(|D|))$ [5].

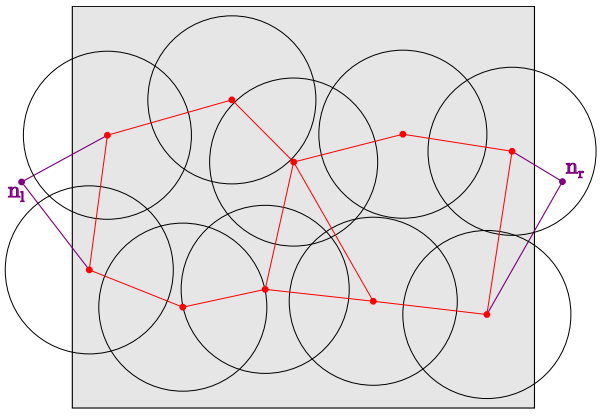


Figure 10: The gray area represents the open belt region. The coverage graph (red) is extended by the nodes n_l and n_r (purple)

A fixed parameter tractable (FPT) algorithm has been developed that optimally solves the general version of the barrier resilience problem by transforming it into open belt instances [13]. Fixed parameter tractability means that its runtime is not exponential in the size of the instance $|D|$, but it is exponential in some other parameter, in this case k, and k can be linear in $|D|$ depending on the instance. The details of this transformation are very complicated and it is not clear whether the authors have implemented it, so it cannot be compared to the algorithms presented in this work for solving barrier resilience optimally.

4.2 Closed belt region

A harder to solve problem considers closed belts, where the belt that previously had two sides that were 'impenetrable' is closed by connecting these sides. The result is a donut shape, with the question being how many disks need to be crossed in order to cross this donut, meaning that a curve c must go from the inside of the donut to the outside. Kumar et al. also introduced this problem in their publication of barrier resilience. In Theorem 4.2 of the conference version of their paper [14], the resilience is equated to the maximum number of cycles in CG that go around this donut. Although the concept behind this theorem will be used to approximate k later on, the theorem itself is incorrect. In the final publication of the paper [15], the theorem was retracted, without actually proving it to be false. Figure 11 shows a counterexample, where $k = 2$, but there are no two node-disjoint cycles in CG.

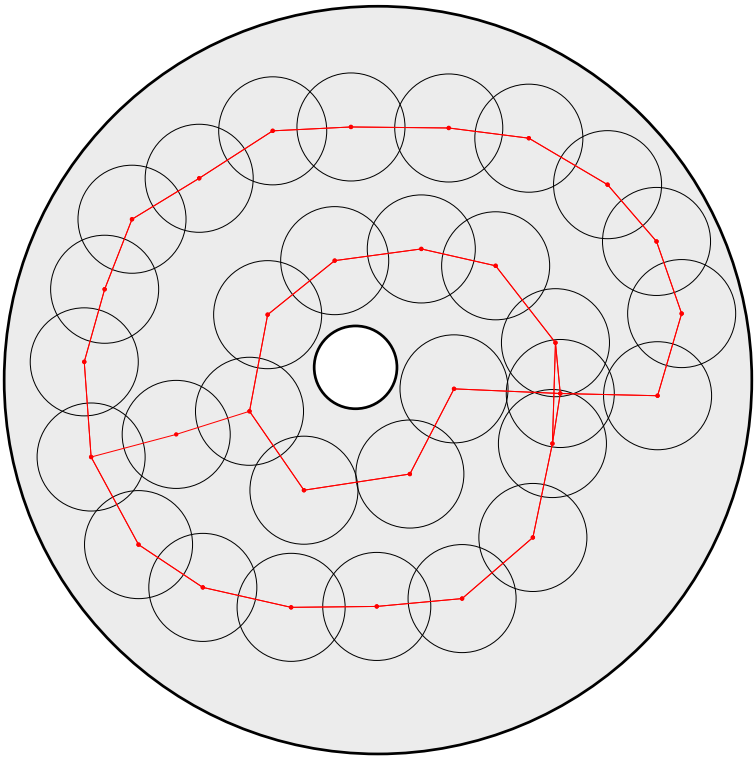


Figure 11: The gray area represents the closed belt region. There are no two node-disjoint cycles in CG (red), although $k = 2$

Without being able to prove this, the closed belt version of barrier resilience seems equally difficult to solve as the general problem formulation. Constructions that make the barrier resilience problem difficult to solve can typically be constrained to the closed belt. Therefore, this work does not consider this restricted version and tries to solve the barrier resilience problem in the general domain.

4.3 Beyond unit disks

Different versions of barrier resilience were already hinted at in the introduction. Probabilistic models for when a sensor detects an intruder might model them more realistically. For this work, it is especially interesting to look at the case where the sensing regions are allowed to be areas other than unit disks, as this problem is closely related. Critically, Theorem 1 no longer applies when other shapes of sensing regions or even just disks of different sizes are considered, so a sensor might have to be traversed more than twice by an optimal path. It is proven for a lot of sensing regions, that the problem of barrier resilience is *NP-Hard*. First, this was proven for skinny regions, like line segments [18]. But it has also been proven for fatter regions, such as axis-aligned rectangles with an aspect ratio close to 1 [13].

Interestingly, the arrangement graph AG and coverage graph CG defined previously can be constructed for any sensor shape, so the presented approximation algorithms work on such shapes as well. The proofs for approximation bounds, however, use the geometry of unit disks extensively and therefore no longer apply when other shapes are considered.

5 Searching a path through the arrangement

Considering the arrangement of the disks looks at the problem as a human would, with the disks arranged on the 2D plane, together with the start and target points. The goal of an algorithm is to find a path through the arrangement that crosses as few disks as possible. The previously defined arrangement graph AG (Definition 5) represents coherent areas called cells as nodes and connects them via edges if the cells share a border. A curve c in the 2D plane starting in s and ending in t can now be easily mapped to a walk ω in the graph by walking along the nodes associated with the cells that c traverses. The same is true in the opposite direction, as a walk ω in AG can be converted to a curve c in the 2D plane, since any point in a cell can be connected to any other point in the same cell without leaving it, and an edge in the graph represents a shared border between cells that can be crossed by c. Because it does not make sense for a curve to leave a cell and enter it again, only paths π that do not go through the same node multiple times are considered in AG. The previously defined Dubins path will always correspond to a path in AG, as it never enters a cell twice (Lemma 1).

Additional information is needed in order to state the barrier resilience problem using just the graph AG. The definition of the undirected graph $AG = (N, E)$ together with the start and target node n_s,n_t is used to calculate the barrier thickness and thereby approximate k. However, it is not sufficient to fully represent the problem in order to find the optimal solution when a disk needs to be entered twice. Just looking at the arrangement graph, it is not clear whether a path enters a new disk, or if the disk has been previously entered. In order to calculate the optimal solution, more information is therefore needed. There are two obvious properties of the arrangement:

- A coherent area (cell) n is covered in its entirety by a fixed set of disks. This set of disks covering a node is saved in the mapping $Disks : N \mapsto Subsets(D)$. (Remark: There may be multiple cells covered by the exact same set of disks, so the mapping is not bijective)
- Two nodes connected by an edge share a border made up of the rim of exactly one disk d. This disk therefore covers one node and not the other, while the other disks covering the nodes n_1 and n_2 are equal: $(Disks(n_1) \cap Disks(n_2)) \cup \{d\} = Disks(n_1) \cup Disks(n_2)$.

Therefore, there is a disk d that is entered when going along one direction of any edge e while d is left going in the opposite direction. The disk that a given edge enters is saved in the mapping $disk\_entered : E \mapsto D$, which maps every edge to a disk. Now the tuple $(AG, n_s, n_t, disk\_entered)$ contains enough information to calculate barrier resilience using it.

The barrier resilience problem can now be formulated as follows:

Definition 6. *Barrier resilience in AG: Find a path π in AG going from n_s to n_t, so that the set of disks entered by the path $\{disk\_entered(e) : e \in E(\pi)\}$ is minimal.*

This problem definition is exactly equivalent to the minimum-color single-path problem, where disks D are called colors C (note that edges are called links by the authors of the following quote): "Given network $G = (N, L)$, where N is the set of nodes and L is the set of links, and given the set of colors $C = \{c_1, c_2, ..., c_K\}$ where K is the maximum number of colors in G, and given the color $c_l \in C$ for every link $l \in L$, find one path from source node s to destination node d such that it uses the minimum number of colors." [21]. This problem is proven to be *NP-Hard* in [21], so there is no efficient algorithm to solve it generally and exactly. But the authors have proposed several algorithms to approximate the optimal solution, which work well for their use cases. In the context of barrier resilience, an approximation of the minimum-color single-path problem has been developed in order to approximate k to a factor of 1.5 [6] using structural properties of the graph such as an optimal path not having to enter a disk more than twice.

5.1 Barrier Thickness

The term barrier thickness was introduced in [2] to describe the minimum number of times that a curve needs to enter a disk to get from s to t. Unlike barrier resilience, every time a disk is entered by c is counted, while barrier resilience only counts the number of distinct disks traversed. In the paper, a weighted and directed graph $AG_{weighted}$ is constructed, which is equivalent to the undirected arrangement graph AG except for the undirected edges being replaced by two weighted edges in both directions. In the direction where a disk is entered, the edge has $weight(e) = 1$, while the edge leaving the disk has $weight(e) = 0$. Now, the barrier thickness is simply the shortest path in this weighted graph $AG_{weighted}$ between the nodes n_s and n_t. As weights are either 0 or 1, the shortest path can be calculated using breadth-first search in $O(|N| + |E|)$ time.

Calculating the barrier thickness does not actually require this directed and weighted version of the graph though: As every disk that is entered has to be left again (s and t are not covered by any disks), the number of edges traversed by a path π from n_s to n_t is twice the number of disks entered. Therefore, the number of disks entered by a path π from n_s to n_t in AG is equal to $\frac{1}{2}length(\pi)$, so the $barrier\_thickness = \frac{1}{2}length(\pi_{min})$ for the shortest path π_{min} from n_s to n_t. This shortest path is also found using breadth-first search in $O(|N| + |E|)$ time, so this simpler formulation does not improve the runtime.

Barrier thickness is a very good approximation of barrier resilience in practice, as will be shown on random instances in the conclusion. Obviously, it provides an upper bound on barrier resilience, with the approximation ratio being the following:

Theorem 2. *Barrier thickness is at most twice the barrier resilience of an instance, except when $|\overline{st}| \approx 2$, where it might be thrice the barrier resilience.*

Proof. Theorem 1 states that an optimal path does not need to cross any disk more than twice, except when $|\overline{st}| \approx 2$, where disks may have to be entered three times by an optimal path. This optimal path that enters at most $2k$ or $3k$ disks when counting multiple entries of the same disk multiple times means that there always exists a path in AG that enters at most $2k$ or $3k$ disks. As barrier thickness searches for the path with the fewest number of disks entered, $k \leq barrier\_thickness \leq 2k$ or $3k$. □

The example in fig. 12 shows an arrangement where $barrier\_thickness = 2 * k - 1$, which can be expanded for any k. The found path c_2 does not enter any disk twice, so barrier resilience cannot be computed by simply taking the path that barrier thickness comes up with and only counting unique disks.

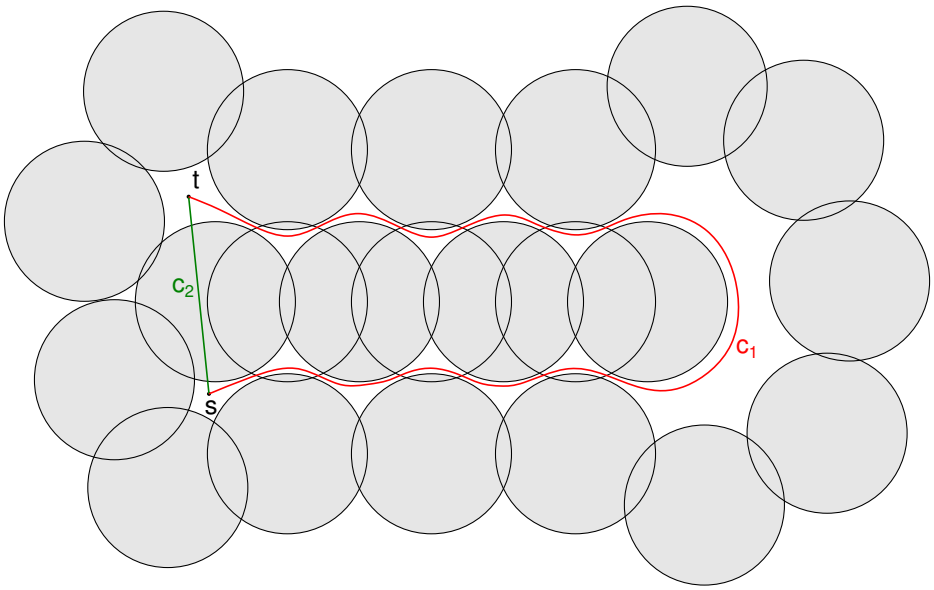


Figure 12: Gray disks represent 5 stacked disks. Barrier thickness finds the curve c_2 (green), which enters 5 disks, but the optimal curve is c_1 (red) with $k = 3$

5.2 ILP formulation

The reason that barrier thickness is not always the same as barrier resilience is that it counts the cost of entering a disk each time a disk is entered, even if it has already been entered before. It was demonstrated in the intuition chapter that this limitation cannot be easily overcome when constructing paths through the arrangement step-by-step. Nevertheless, barrier resilience can be represented nicely in an integer linear program (ILP). In the context of the ILP formulations, edges E are always directed, going in the direction where they enter a disk. This means that every edge $e \in E$ is defined by a tuple $e = (n_1, n_2)$, so that the disk $d = \mathit{disk\_entered}(e)$ is entered going from n_1 to n_2: $\mathit{Disks}(n_2) = \mathit{Disks}(n_1) \cup \{d\}$. In the ILP formulation, a flow with a value of 1 from the start node to the target node is searched, representing the optimal path π in the graph. There is an activation variable a_d for every disk d, where $a_d = 1$ if the disk is entered by the flow and $a_d = 0$ if it is not. There is also a flow variable f_e for every edge $e \in E$, which is 1 if the path π takes the edge in the forward direction, -1 if π takes the edge in reverse direction and 0 otherwise. The objective function minimizes the number of disks crossed, which is the sum of disk activations:

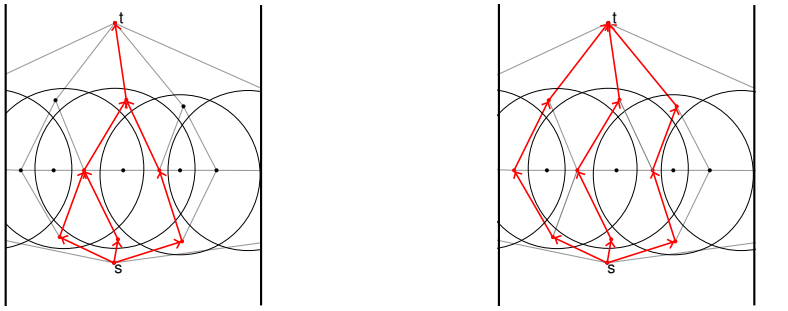
$$\begin{aligned}
&\text{minimize} && \sum_{d \in D} a_d \\
&\text{subject to} && \text{Flow conservation constraint } (\mathit{inflow} - \mathit{outflow} = 0): \\
& && \forall n \in N : \sum_{e \in E | n_2 = n} f_e - \sum_{e \in E | n_1 = n} f_e = \begin{cases} 1 & \text{if } n = n_s \\ -1 & \text{if } n = n_t \\ 0 & \text{otherwise} \end{cases} \qquad (2) \\
& && \text{Disk activation constraint (if flow enters a disk, it must be activated):} \\
& && \forall e \in E : a_{\mathit{disk_entered}(e)} \geq f_e
\end{aligned}$$

This ILP results in an optimal solution for the problem, as the flow through the arrangement is synonymous with a curve c through the plane, and the number of disks crossed by the flow is minimized. As ILPs are generally *NP-Hard* to solve, it is, however, not necessarily an efficient solution to the problem. Even in practice, the performance is very poor, which will be shown in the benchmark later. The reason becomes apparent in the following section, where the LP relaxation of the ILP is considered, whose results are far off from the ILP solution.

5.2.1 LP Relaxation

A linear program that does not require variable values to be integers is solvable in polynomial time. The LP relaxation uses this by dropping the integer constraints of the original problem formulation, while hoping that the result will still be useful in solving the problem or at least approximating it. As the integer constraints are dropped, while no additional constraint is added, the result of the optimization function can only become 'better' and the LP relaxation of eq. (2) will therefore be a lower bound for k. Concretely, this means that a flow value is no longer restricted to be 1, 0 or -1. The result is that the flow from n_s to n_t is split, and the flow variables f_e have real-valued solutions between $-1 \leq f_e \leq 1$. Because the flow from n_s to n_t must still sum to one, and no flow can be 'lost' or 'generated' at any node, the flow can be represented as the sum of flows going along different paths from n_s to n_t. This set of paths is called $Paths$, and the flow going along each one of these paths is saved in the mapping $flow : Paths \mapsto \{f \in \mathbb{R} | 0 < f \leq 1\}$. The sum of flows must be equal to 1: $\sum_{\pi \in Paths} flow(\pi) = 1$. The paths that the flow goes along, and their associated flow values, are not unique given an LP solution, but it is still a useful way of thinking about a solution to the LP problem.

Unfortunately, the LP relaxation of the problem does not provide useful approximations for barrier resilience. The flow taking many paths that are close to each other is actually rewarded, as entering a disk through different edges only increases the activation variable to that of the edge with the maximum flow going into the disk. Even when entering the same cell through multiple edges, the sum of the flow going through a cell does not impose a lower bound on the activation variable of the associated disks. An example of this is shown in fig. 13 (a), where the flow takes multiple paths through the barrier, which only force the activation variable of 3 disks to be $\frac{1}{3}$, so the optimal value of the LP is 1, even though $k = 2$.



(a) Basic LP formulation (b) With the cell flow constraint

Figure 13: The flow is split along 3 different paths (red), each with a flow of $\frac{1}{3}$. The activation of three disks is therefore $\frac{1}{3}$ and the LP solution is 1, despite $k = 2$

This example can be extended and when increasing the overlap of disks, an arbitrarily bad

approximation factor can be achieved. This means that the LP solution will be close to 1, while the actual barrier resilience k can be arbitrarily raised. Figure 14 shows such a construction where the optimal LP value is kept close to 1, while $k = 3$. Note that the nodes of the cells are drawn at the leftmost point of the cell, so the paths of the flow shown in red are on the left side of the cells that they cross. Observe how the optimal curve that enters $k = 3$ disks simply goes toward the right, but because there are fewer intersections on this curve, this way is 'less interesting' for the LP solution, as it benefits from crossing the same disks through a lot of different paths. This leads to some of the edges of the optimal path having a flow of 0: $\exists e \in E(\pi_{opt}) : f_e = 0$. Therefore, there is no obvious way of using the LP solution to come up with the optimal path.

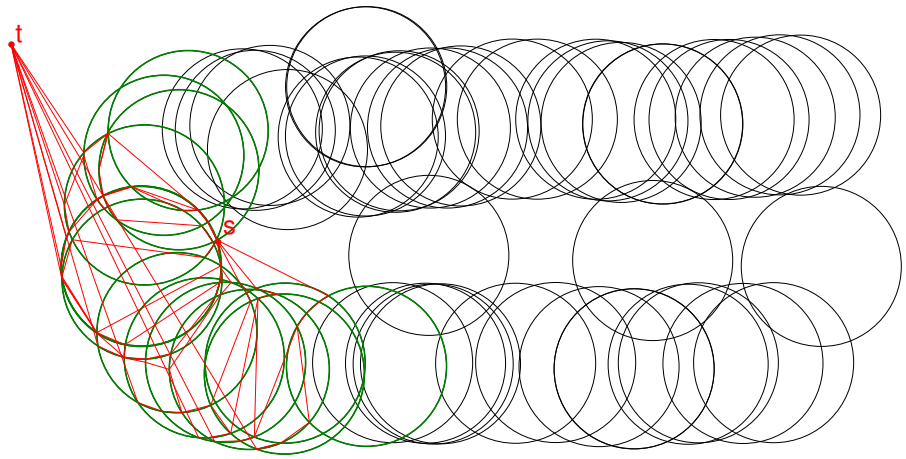


Figure 14: The LP has an optimal value ≈ 1, even though $k = 3$. The disks with $a_d > 0$ are painted green, while the paths that the flow travels along are drawn in red.

Now the idea is that the LP can be formulated in a stricter way, at the expense of the formulation being less concise. These constraints were not necessary in the ILP, but when variables are not forced to be integers, additional constraints might limit this partial solution, making it a better approximation for k and potentially improving the runtime of solving the ILP. Before adding additional constraints, an equivalent LP formulation is introduced, which is slightly more verbose than the original one. The flow variables f will now only be positive. Therefore, two flow variables are needed for every edge so that the flow can still go in both directions through an edge. The flow going forward through an edge e entering the associated disk $d = disk\_entered(e)$ is saved in the variable $f_{e,f}$, while the flow in the backward direction leaving the disk d is captured in the variable $f_{e,b}$.

$$\begin{aligned}
\text{minimize} \quad & \sum_{d \in D} a_d \\
\text{subject to} \quad & \text{Flow conservation constraint } (inflow - outflow = 0): \\
& \forall n \in N : \sum_{e \in E | n_2 = n} (f_{e,f} - f_{e,b}) + \sum_{e \in E | n_1 = n} (f_{e,b} - f_{e,f}) = \begin{cases} 1 & \text{if } n = n_s \\ -1 & \text{if } n = n_t \\ 0 & \text{otherwise} \end{cases} \\
& \text{Disk activation constraint (if flow enters a disk, it must be activated):} \\
& \forall e \in E : a_{disk_entered(e)} \geq f_{e,f}
\end{aligned} \tag{3}$$

This formulation is equivalent to the previous one, but it allows for adding additional constraints. The main problem of the LP relaxation is that the path is split into many different paths, so that many paths traverse the same disk, while entering it through different edges. One way to restrict this behavior slightly is to sum the flow entering any cell. There might be multiple edges entering a cell, so the flow through a cell can be higher than that of its adjacent edges. The flow going through a cell n is now measured in the variable cf_n. It is calculated by summing the flow going along edges that point into the node, plus the sum of flow going backwards along edges that leave the node. If a path traverses a cell, all disks covering this cell must be deactivated. Therefore, the flow cf_n can be used as a lower bound for the activation variable of every disk covering this cell. This additional constraint will be called the 'cell flow constraint':

$$\begin{aligned}
& \forall n \in N : cf_n = \sum_{e \in E | n_2 = n} f_{e,f} + \sum_{e \in E | n_1 = n} f_{e,b} \\
& \forall n \in N : \forall d \in Disks(n) : a_d \geq cf_n
\end{aligned} \tag{4}$$

Note that cf_n is just an auxiliary variable, meaning that it can be discarded by inserting the definition of cf_n directly into the second constraint.

This addition does constrain the problem usually, meaning that the LP result does get closer to the ILP solution for random examples. It still has the same problem, however, because paths traversing a disk by going through different cells still only force the activation of a disk to be that of the cell with the highest flow. How this works in theory is shown in fig. 13 (b), and the example in fig. 14 actually includes this constraint. But flow going through a disk using node-disjoint paths cannot be forbidden, as the optimal solution might have to enter a disk multiple times. Because an optimal solution only has to enter any disk twice, summing the flow that enters a disk and then dividing it by 2 can be used as a lower bound for the disk activation:

$$\forall d \in D : a_d \geq \frac{1}{2} \sum_{e \in E | disk\_entered(e) = d} f_{e,f} \tag{5}$$

Note how if the factor of $\frac{1}{2}$ were absent from the constraint, the ILP would represent barrier thickness, as when a disk is entered twice by a flow, it is counted twice towards the activation variable. This is why the constraint will be called the 'thickness constraint'. Importantly, also the LP relaxation that is currently being looked at would come up with a result for barrier thickness. Consider the result of the optimization function:

$lp = \sum_{d \in D} a_d$

If the newly added constraint did not have the $\frac{1}{2}$ factor, it would limit a_d like so:

$a_d \geq \sum_{e \in E | disk\_entered(e)=d} f_{e,f}$
$\rightarrow lp \geq \sum_{d \in D} \sum_{e \in E | disk\_entered(e)=d} f_{e,f}$
$\rightarrow lp \geq \sum_{e \in E} f_{e,f}$

Remember that $f_{e,f}$ is the flow that enters a disk going along the edge e. Now remember that the flow is partitioned into paths from n_s to n_t that have a positive flow going along them. Every path needs to enter at least *barrier thickness* many disks, as this is the definition of barrier thickness. This means that the flow of every path contributes to the value of at least *barrier thickness* many flow variables $f_{e,f}$ and as the sum of flow over all paths is 1, $lp \geq$ *barrier thickness*. With the factor $\frac{1}{2}$ added again to the constraint in eq. (5), this still limits the best possible LP solution to be $lp \geq \frac{1}{2}$*barrier thickness*, so $lp \geq \frac{1}{2}k$.

Even with this fixed approximation bound for the LP relaxation achieved by adding this constraint, the solution might not traverse the same edges and disks that the optimal solution does. The same example as before is shown in fig. 15, where the optimal value of the LP is just above 1.5. It does get closer to k, but the paths that the flow goes along still do not include the optimal path that goes toward the right.

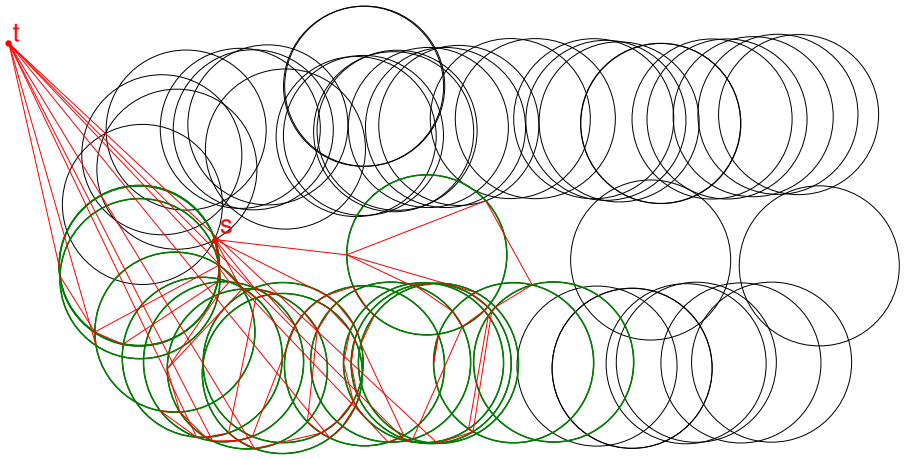
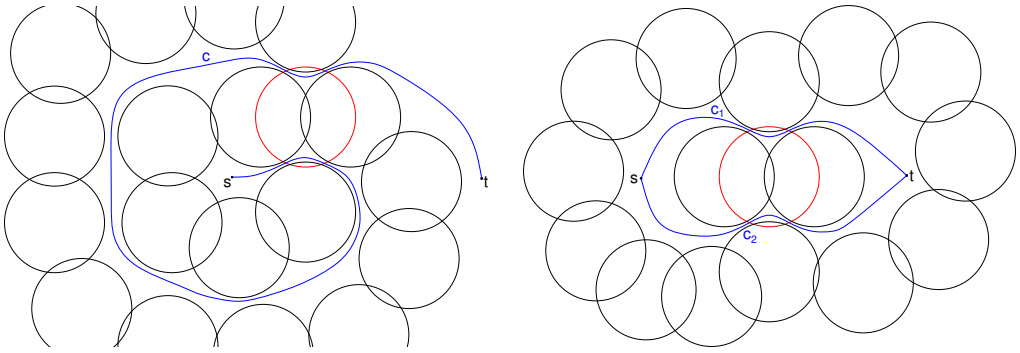


Figure 15: The LP has an optimal value ≈ 1.5, even though $k = 3$. he disks with $a_d > 0$ are painted green, while the paths that the flow travels along are drawn in red.

It is actually impossible to achieve a better lower bound for the approximation ratio of the LP when only the immediate surroundings of cells/disks are considered: Look at the two instances in fig. 16. The arrangement surrounding the red disk is exactly the same in both examples. The LP must allow the curve c in (a), which enters the disk twice. Allowing this curve means only forcing the activation variable of the red disk to be 1 and not higher. In (b), however, the flow might split, going along the two curves c_1 and c_2. Just looking at the problem from the perspective of the red disk, these two curves look the same as the one curve in (a). Because of the linearity of constraints, the activation variable can only be forced to be $\frac{1}{2}$ when the flow is split equally between c_1 and c_2.



(a) The red disk is traversed twice by an optimal curve c (blue)

(b) The two curves c_1 and c_2 (blue) traverse the red disk as c does in (a)

Figure 16: Local constraints cannot differentiate the two cases

5.2.2 Practical analysis

The results and runtimes of the different LP formulations are given in table 1. The "LP formulation simple" uses the initially introduced LP (eq. (2)), the "LP formulation flow positive" uses the formulation where flow variables were split into two, so that the flow is always positive (eq. (3)). This formulation is used together with the additional cell flow constraint (eq. (4)) in "Constraint 1", the thickness constraint (eq. (5)) in "Constraint 2", and both constraints combined in "Constraint 1+2". The algorithms were run for 20 randomly generated instances, containing 100 disks, with their runtime and result recorded. The approximation factor was calculated for every run and then averaged over the different runs. The runtime of solving the LP and the runtime with the additional integer constraints were averaged over all runs.

	LP formulation simple	LP formulation flow positive	Constraint 1	Constraint 2	Constraint 1+2
LP Runtime	5.0s	2.6s	11.2s	1.7s	10.8s
ILP Runtime	>1000s	>1000s	157s	58s	49s
Approximation factor	0.33	0.33	0.65	0.60	0.75

Table 1: Comparison of the different LP formulations on random instances, $|D| = 100$

The approximation factor of the basic LP is quite poor, its result is on average lower than $\frac{1}{2}k$. Adding the extra constraints helps immensely, particularly the cell flow constraint. This is interesting, because the thickness constraint guarantees an approximation factor of $\frac{1}{2}k$, which the cell flow constraint does not. Nevertheless, there are instances where the thickness constraint achieves a better approximation than the cell flow constraint. Adding constraints obviously always improves the LP result, with the combination of both constraints clearly offering the best approximation out of all LP formulations.

Interestingly, the more verbose formulation of the basic LP actually achieves a better runtime than the more compact one. The cell constraint increases the runtime drastically, probably because

there is a constraint added for every disk covering every cell, so the number of constraints added is quite large. Conversely, introducing the barrier thickness constraint actually decreases the runtime of the LP. For the runtime of the ILP, the extra constraints help immensely, as the basic ILP formulation takes up to one hour to solve for the tested instances. The thickness constraint is significantly better than the cell flow constraint for helping the ILP to resolve faster, with both constraints combined once again being best. This proves the theorized benefit of having the LP solution more closely match the ILP solution when solving the ILP.

Even the best LP formulation with both constraints added is not that useful in practice, though. In the randomly generated instances, barrier thickness always returned an optimal solution with a runtime $< 100ms$. If the goal is to find a lower bound for k, the ring problem LP introduced later delivers a lower bound that also returns the optimal solution k for the analyzed instances, with its runtime also being $< 100ms$. The LP relaxation also does not offer a good basis for finding the optimal solution, as the paths taken by the flow for all program formulations do not necessarily include the optimal path.

This concludes the first part of this thesis, which worked on the arrangement graph AG. The second part of this thesis will work with the coverage graph CG and will achieve much better results.

6 Working with the coverage graph

Recall the definition of the coverage graph $CG = (N, E)$: Disks are represented by nodes N, which are connected by an edge $e \in E$, if the disks intersect (Definition 3). This graph definition was used for the solution of the open belt problem shown previously. In the general version of the barrier resilience problem, this primal graph does not sufficiently represent the problem, as it is not clear if s and t are separated by only looking at the graph CG without any geometric information, as shown in fig. 3, where the exact same graph separates s and t on the left, but not on the right. Therefore, most algorithms and visualizations preserve the coordinates of nodes N, as well as s and t. Another representation of the problem instance where this information is omitted will be introduced later.

When solving the barrier resilience problem using the coverage graph, the goal is to find a set of nodes $I \subseteq N$ that are deactivated, so that s and t are not separated in the graph CG_{-I}, where nodes I together with their adjacent edges are deleted:

Definition 7. *Barrier resilience in CG: Find a minimum set of nodes $I_{min} \subseteq N$ such that there can be a curve c from s to t that does not intersect any edge in the graph $CG_{-I_{min}}$.*

To prove the equivalence of this problem definition to the barrier resilience problem, it must be shown that any solution I to this problem is also a solution to the original problem, and vice versa. Thereby, a minimum solution to this problem I_{min} is also a minimum solution of the original problem, and the size of the minimum solution $|I_{min}|$ is therefore equal to k.

Theorem 3. *A set of disks I is a solution to the barrier resilience problem $\iff$ the associated set of nodes I is a solution to the barrier resilience problem in CG.*

Proof. In both problems, a set of disks/nodes I is a valid solution, if by deactivating the elements in I, there exists a curve c that does not intersect any disks/edges. So by proving that for a given set of deactivated disks I, there exists a curve c that does not intersect any active disk, if and only if there exists a curve that does not intersect any active edges in the coverage graph CG_{-I}, the theorem is proven. The following two lemmas will prove this statement in one and the other direction. □

Lemma 2. *A curve c that goes from s to t without intersecting any active disk $d \in D \setminus I$ does not intersect any edge $e \in E$ in the coverage graph CG_{-I}*

Proof. Proof by contradiction: Assume there is a point p in c that lies on an edge e in CG_{-I}. As the length $|e| < 2$, there is one endpoint $n \in e$ that has a distance $|\overline{np}| < 1$. As both nodes of an active edge have to remain active, c would therefore also intersect the active disk associated with n. □

In the opposite direction, a curve c_{CG} that does not intersect any active edge in CG_{-I} may intersect a disk $d \in D \setminus I$. Therefore, the following lemma is different from the previous one:

Lemma 3. *If there is a curve c_{CG} from s to t that does not intersect any active edge in the coverage graph CG_{-I}, then there exists a curve $c_{original}$ from s to t that does not intersect any active disk $d \in D \setminus I$.*

Proof. The following algorithm converts c_{CG} into a curve $c_{original}$ that does not cross any disks, proving the lemma. Consider the point a where c_{CG} enters a disk $d \in D$ and the first point b going along c_{CG} from a that is not covered by any disk. The set of all disks that the path c_{CG} intersects between a and b is called D_c. Every point of c_{CG} between a and b must be covered by a disk $d \in D_c$.

Therefore, when a new disk $d_{new} \in D_c$ is entered at the point $p \in c_{CG}$, there must be at least one previously entered disk $d_{prev} \in D_c$ that still covers p. The newly entered disk must lie on the same side of c_{CG} as d_{prev}, as there would otherwise be an edge between d_{new} and d_{prev} crossing c_{CG}. Therefore, all disks $d \in D_c$ lie on the same side of c_{CG}. There can also not be a disk on the other side of c_{CG} that intersects any disk $d \in D_c$ as this would also create an edge crossing c_{CG}. Therefore, the curve $c_{original}$ can simply go around all disks $d \in D_c$ by going along the rim of disks D_c between a and b. This can be done for any part of c_{CG} covered by disks, resulting in the path $c_{original}$, that does not cross any disk $d \in D \setminus I$.

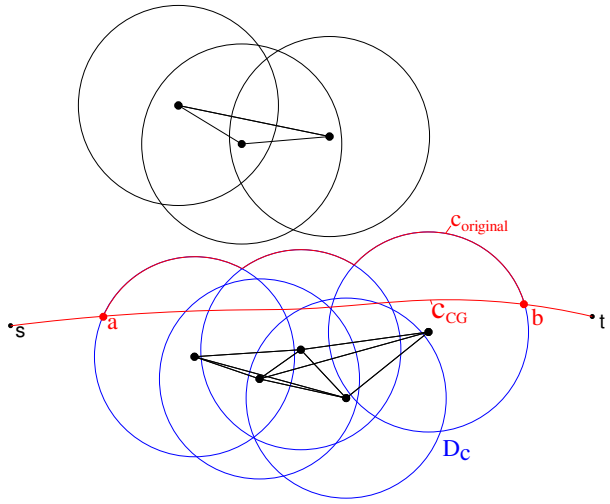


Figure 17: The curve c_{CG} can be transformed into the curve $c_{original}$, as all disks D_c lie on the same side of c_{CG} and can be circumvented

□

6.1 Intuition

In this section, the properties of the coverage graph CG are explored, which originate from the geometry of the arrangement.

Remember that the barrier resilience problem is difficult to solve because in certain arrangements, a disk might have to be crossed multiple times by an optimal curve. If this were not the case, barrier thickness would provide an optimal solution to the problem. Such an arrangement is shown in fig. 18, where the central disk might need to be crossed twice. The coverage graph of this arrangement is overlaid in the figure. Notice how there are two intersecting edges, whose endpoints are not all connected. If this were not the case, meaning that all intersecting edges had all their endpoints connected, the approximations presented later would be optimal. This kind of arrangement will repeatedly show up in constructions which prove that the presented algorithms do not solve the problem optimally.

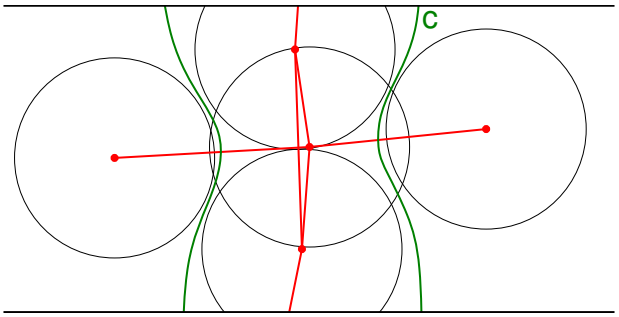


Figure 18: The central disk might have to be crossed twice by the optimal curve c (green). The coverage graph (red) contains two intersecting edges whose endpoints are not all connected.

The arrangement can actually be simplified for the coverage graph to contain this problem, by removing the node on the right. Even though the arrangement no longer represents a disk that may be crossed twice, there are still two intersecting edges whose endpoints are not all connected. This is sufficiently 'hard' for some algorithms to have problems and will be sufficient for some worst-case constructions. Just as in the dual problem, where a path only needs to cross a disk twice, this problem in the coverage graph can also be constrained due to geometric limits. The following lemma will show up repeatedly in the following chapter:

Lemma 4. *If two edges $a, b \in E$ intersect, there is at least one endpoint $n \in a \bigcup b$ that is connected to all other endpoints of a and b.*

Proof.

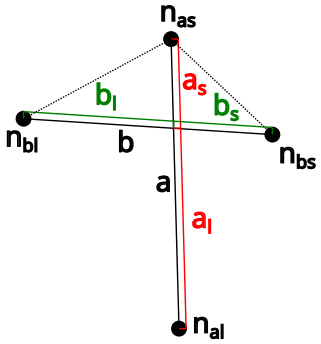


Figure 19: Proof for Lemma 4

The edge a is divided into the two parts on either side of the intersection point: $a = a_s + a_l$, so that $|a_s| \leq |a_l|$, same for b. Assume WLOG $|a_s| \leq |b_s|$. For the nodes to be connected, $|a| < 2$ and $|b| = |b_s| + |b_l| < 2$. Since a_s is the smallest part, $|a_s| \leq |b_s| \leq |b_l|$. Now a_s can be inserted for b_s and b_l in the equation $|b_s| + |b_l| < 2$: $|a_s| + |b_l| < 2$ and $|b_s| + |a_s| < 2$. Because of triangle inequality, this means that the edge between $\overline{n_{as}n_{bs}}$ and the edge $\overline{n_{as}n_{bl}}$ have a length < 2, so these edges must exist in CG, connecting n_{as} with every other node in the graph.

□

6.2 Barrier resilience in the coverage graph

Although this graph representation is very useful for solving the problem constrained to an open belt, it is not obvious how to approach this problem in the general domain. An idea to transform the plane into an open belt was proposed in [13], where an infinite curve $c(s, t, \inf)$ that starts in s, passes through t, and goes off to a point infinitely far away is considered. The 2D plane is then 'cut' along this edge, resulting in an open belt instance that can be easily solved. For the solution of this open belt instance to be accurate, $c(s, t, \inf)$ is not allowed to cross the optimal curve c_{opt}, and dealing with a curve that crosses the same disk twice is not trivial.

In the previously discussed closed belt version of the problem, Theorem 4.2 in [14] states that the barrier resilience is equal to the maximum number of node-disjoint cycles in CG that go around the belt. It has been shown previously that there may be fewer than k cycles in CG. There is also the problem that the property of a cycle to 'go around the belt' has to be defined more clearly, so that an algorithm can easily decide whether this is the case, especially when considering the general domain not constrained to a closed belt.

In this work, a similar idea is developed, which also uses the idea of cycles to solve the barrier resilience problem. Unlike earlier attempts, the properties of rings are developed more thoroughly, which allows for the concept to be applied outside of closed belts and enables a more precise and easily implementable mechanism to determine whether a ring separates s and t. A new problem definition that is equivalent to barrier resilience called the ring problem is introduced, which manages to hold true even for the example where there are fewer than k node-disjoint cycles.

7 Ring problem

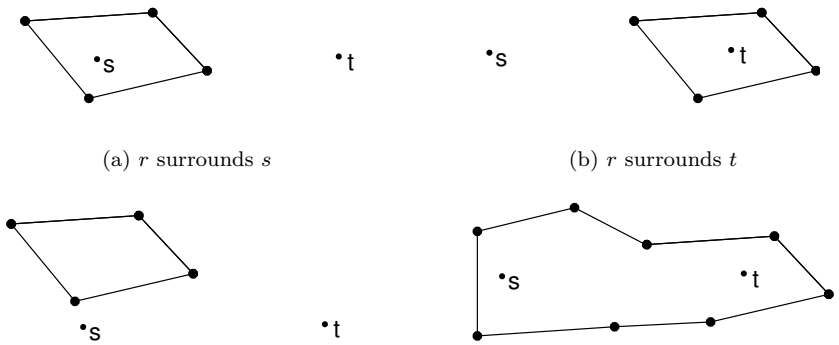
As the coverage graph by itself does not offer an intuitive way of solving the barrier resilience problem, a new way of thinking about CG is introduced in this chapter. The focus is on finding cycles in CG, which separate s and t. These cycles will be called rings. First, the definition and properties of rings are developed, which are then used to define the so-called *ring problem*. This will be shown to be equivalent to the barrier resilience problem. Algorithms for solving the ring problem are developed, which return lower bounds for k, and are therefore very useful in combination with the previously discussed barrier thickness approximation.

7.1 Rings

A ring is a closed walk in the coverage graph CG, but it is assigned a distinct name in this work as a lot of special properties are discussed. An example showing 4 different rings is shown in fig. 20. A ring that traverses a node multiple times is called a *non-simple ring*, otherwise the ring is called *simple*. A simple ring hence corresponds to a closed path in CG, which is generally referred to as a *cycle*. A ring r is represented as a sequence of nodes, where the last node equals the first one. The length of the ring corresponds to the number of edges that the walk travels along, which is one less than the length of the sequence, as the first/last element is contained twice in the sequence. The sequence of nodes induces a sequence of edges that is travelled along, which is called $E(r)$.

Definition 8. *Ring* $r = (n_0, n_1, ..., n_{length(r)-1}, n_0)$
$E(r) = (\{r[0], r[1]\}, \{r[1], r[2]\}, ..., \{r[length(r) - 1], r[length(r)]\})$
$length(r) = |r| - 1 = |E(r)|$
$set(E(r)) \subseteq E$
r *is simple* $\iff |set(r)| = length(r)$

The set of all possible rings is called R^*.



(a) r surrounds s (b) r surrounds t

(c) r surrounds neither s nor t (d) r surrounds s and t

Figure 20: Different rings embedded in the 2D plane

7.1.1 Extended graph representation

At this point, the definition of a ring is equivalent to that of a closed walk in a graph. It was previously shown that it is not clear whether such a walk separates s and t when just looking at the graph. Figure 20 shows four cases, where s and t are separated in the first two cases. In cases

(a) and (b), the rings surround the points s and t respectively, thereby separating the points. In case (c), it does not surround any point, while in case (d), the cycle surrounds both points s and t, which both results in s and t not being separated. This observation can be used to decide if a ring separates s and t. Whether a ring surrounds a point p can be checked by summing the angle between all adjacent nodes to that point: $\sum_{i\in\{0,1,\dots,length(r)-1\}} \angle(r[i]\ p\ r[i+1])$. If this sum equals 2π, p is surrounded by r. This can be calculated for both s and t, so that a ring separates s and t if and only if it surrounds exactly one of the two points. While this approach works and was used and implemented initially, another way of determining whether a ring separates s and t that requires less information is introduced, which will be used instead. The main advantage is that it unifies the decision process, so it does not have to be checked if a ring surrounds s and if it surrounds t separately.

For this, an arbitrary curve $c(s,t)$ from s to t is chosen, for example the straight line between the two points. In the following, whether a ring r separates s and t will be decided based on a ring r intersecting $c(s,t)$ an odd or an even number of times. Some degeneracies might arise if $c(s,t)$ is poorly chosen: $c(s,t)$ is not allowed to cross the center of any node exactly, as it would not be clear which edges were crossed.

The idea of this curve is that determining whether a path / ring intersects $c(s,t)$ an even / odd number of times can be decided without knowing the trajectory of the curve and the location of the nodes. In order to capture whether some walk ω in the graph crosses $c(s,t)$ an even or odd number of times as a boolean value, the term polarity is introduced:

Definition 9.

$$polarity(\omega) = \begin{cases} true & \text{if } \omega \text{ crosses } c(s,t) \text{ an odd number of times} \\ false & \text{if } \omega \text{ crosses } c(s,t) \text{ an even number of times} \end{cases}$$

This definition of polarity can also be applied to a curve c that does not represent a walk in the graph, so $polarity(c) = true$ if c crosses c(s,t) an odd number of times, $false$ otherwise.

When a path is added to another path, it suffices to XOR the polarities of the paths in order to calculate the polarity of the combined path:

$polarity(\omega_1 + \omega_2) = polarity(\omega_1) \oplus polarity(\omega_2)$

An edge can be seen as a path of length 1, giving it a polarity as well. This is saved in the following mapping:

$polarity : E \mapsto \{true, false\}$

$polarity(e) = true$ if e crosses $c(s,t)$ an odd number of times, $false$ otherwise

With this, the polarity of any walk ω can be calculated by XORing the polarity of its edges:

$polarity(\omega) = \bigoplus_{e\in E(\omega)} polarity(e)$

Now, the positions of nodes and the curve in the 2D plane are no longer necessary for determining whether a walk ω crosses $c(s,t)$ an even or odd number of times after calculating the polarities of the edges. Representing a problem instance using only the coverage graph $CG = (N, E)$ together with this mapping $polarity : E \mapsto \{true, false\}$ will later be shown to contain enough information to verify whether a certificate $I \subseteq N$ solves the barrier resilience problem. Therefore, it can also be used to find the optimal solution to the problem (at least in exponential time). The advantage of this representation is that the binary nature of the polarity makes it easy to implement efficiently. In practical implementations, the straight line between s and t is chosen as the curve: $c(s,t) = \overline{st}$.

A useful lemma for some later proofs is the following, which allows a transformation of a curve c while keeping its polarity constant.

Lemma 5. *A curve c going from point a to point b has the same polarity as a curve c' from a to b, if the area contained by the two curves includes neither s nor t. It is required that c and c' do not intersect themselves.*

Proof. When $c(s,t)$ crosses either of the curves, it lands in the area contained by c and c'. To leave the area again, which it must, it needs to cross either c or c' again, equalizing their polarities. □

7.1.2 Deciding whether a ring separates s and t

For an arbitrary ring, the following lemma holds:

Lemma 6. *A ring r separates s from t if the ring crosses $c(s,t)$ an odd number of times.*

This lemma is proven by looking at only a part of the ring and proving that this part of the ring separates s and t. As the curve of this subring is entirely present in the original ring, if the subring separates s and t, the original ring will separate the points as well.

Proof. First of all, the ring is transformed into a simple ring, which uses a subset of the edges of the original ring: Consider any node that is visited twice at index i and j in r. Now split the ring into two parts, one going from $i-j$ and the other going from $j-i$. One of them must cross $c(s,t)$ an odd number of times, as the original ring crosses $c(s,t)$ an odd number of times. Continue with this part of the ring. Repeat this procedure until the ring is simple.

Now, there still might be two edges $a, b \in E(r)$ in the ring that intersect in the 2D plane, see fig. 21 left. On the right side it is shown how the ring can be split in two, with the curves of both rings making up exactly the original ring, the edges are simply reordered. Therefore, one of the two rings must cross $c(s,t)$ an odd number of times. Continue with this ring and repeat this procedure until every intersection in r has been removed. Note that the resulting ring is not made up of entire edges contained in the graph CG anymore, but this is irrelevant for the proof. In the end, there is a ring using a subset of the edges of the original ring, which is simple, contains no intersections, and crosses $c(s,t)$ an odd number of times. It will be proven in the following Lemma 7 that such a ring separates s and t, proving that the original ring separates s and t as well.

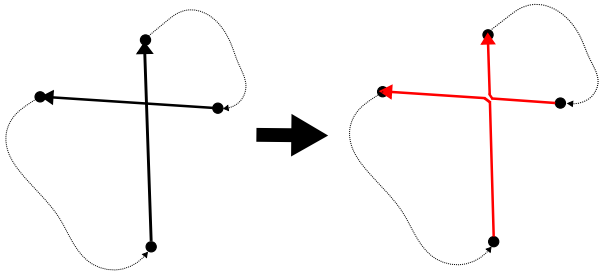


Figure 21: The original ring (black) is split in two (red), one of them will cross $c(s,t)$ an odd number of times

□

This does not mean that a ring which crosses $c(s,t)$ an even number of times does not separate s and t, as demonstrated in fig. 22. Additional properties of the ring have to hold for that to be the case, which are described next.

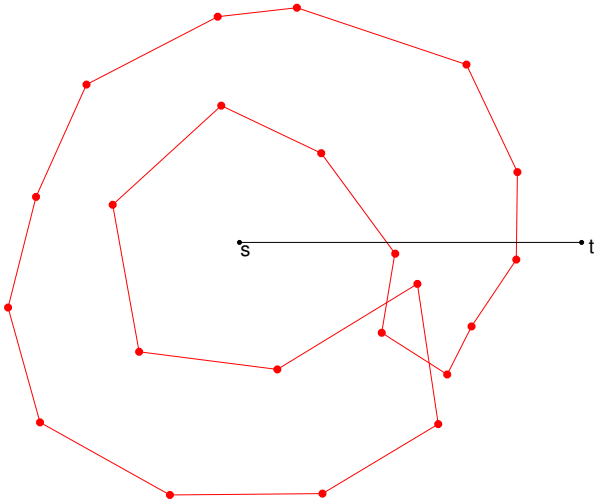


Figure 22: The ring in red separates s and t while crossing $c(s,t)$ an even number of times

7.1.3 Non-intersecting ring

Definition 10. *A non-intersecting ring is a simple ring r that contains no two edges $\nexists e_1, e_2 \in E(r)$ that intersect in the 2D plane.*

A non-intersecting ring has the nice property that it separates the plane into exactly two areas, one on one side of the edges and one on the other. This allows for a stronger variant of Lemma 6.

Lemma 7. *A non-intersecting ring separates s from t if and only if the ring crosses $c(s,t)$ an odd number of times.*

Proof. A non-intersecting ring separates the plane into exactly two areas, which lie on either side of every edge. Whenever the ring crosses $c(s,t)$, the area below and above the crossing are part of the different areas. When $c(s,t)$ is crossed again, the areas outside of the two crossings must be the same, as they are separate from that in between the crossings. As s and t are on opposite ends of $c(s,t)$, if the ring crosses $c(s,t)$ an odd number of times, the points are in the two separated areas, while they are in the same if the number of crossings is even. Note that the number of crossings of $c(s,t)$ is odd if the $polarity(r) = true$ and even otherwise via the definition of polarity.

□

The set of all possible non-intersecting rings is called $R^*_{ni} \subseteq R^*$.

7.1.4 Minimal ring

Definition 11. *A minimal ring r is a simple ring that does not have any two nodes $n_1, n_2 \in r$ that are non-adjacent in the ring and are connected by an edge in CG:*
$\nexists n_1, n_2 \in r : \{n_1, n_2\} \notin E(r) \wedge \{n_1, n_2\} \in E$

In graph theory, an edge between two nodes of a cycle, which are non-adjacent in that cycle, is called a *chord*. A cycle without a chord, equivalent to the definition of a minimal ring, is called an *induced* or *chordless* cycle [10].

A minimal ring has the following nice properties, which make it very useful:

- A minimal ring is always non-intersecting, as two intersecting edges always have one endpoint connected to all other endpoints (Lemma 4). An intersecting ring therefore has a node that has three neighbors that are also part of the ring, which is impossible in a minimal ring.
- The set of edges of the ring $set(E(r))$ can be deduced from the set of nodes in the ring without knowing the order of the nodes in the ring, as every node in the ring has exactly two neighbors that are also part of the ring: $set(E(r)) = \{e \in set(r)^2 | e \in E\}$
- Similarly, the order of the nodes can be deduced after choosing a starting node and a direction of travel using just the set of nodes $set(r)$.

Because the set of nodes of the ring $set(r)$ is sufficient to describe the ring fully, minimal rings are usually represented by just this set of nodes. This also means that two minimal rings are equivalent if $set(r_1) = set(r_2)$.

The following will present an algorithm for transforming any ring r (even a non-simple ring) that crosses $c(s,t)$ an odd number of times into a minimal ring r_{min}, which also crosses $c(s,t)$ an odd number of times, using a subset of nodes of the original ring.

Algorithm 1 Ring minimization

Take two indices i, j in r where the nodes are connected through an edge e ($\{r[i], r[j]\} \in E$), but are not adjacent in the ring ($|i - j| \neq 1$). Now split the ring r into the two walks ω_1 and ω_2 by splitting the ring at the indices i and j (see figure). Since these two walks together contain all the edges of the original ring $E(\omega_1) + E(\omega_2) = E(r)$, one of the walks crosses $c(s,t)$ an even number of times, while the other crosses $c(s,t)$ an odd number of times. The edge e can either cross $c(s,t)$ or not. Combine the edge e with one of the walks ω_1 or ω_2 to form a new ring r', so that the resulting ring crosses $c(s,t)$ an odd number of times. As i and j are non-adjacent, there is at least one node deleted from the ring. Repeat this procedure until there are no longer any neighboring nodes that are non-adjacent in the ring, meaning that the ring is minimal by definition.

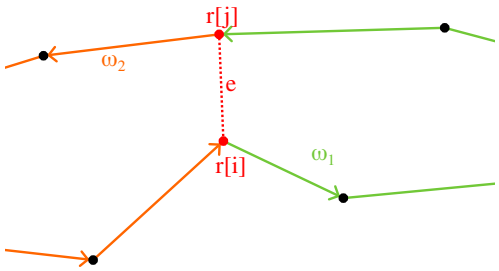


Figure 23: Ring r is split into the two walks ω_1 (green) and ω_2 (orange) at indices i and j

Note that during the transformation, adding the edge e might introduce intersections into the ring r'. This is not a problem however, as the ring will lose all intersections once the ring is fully minimized. The implementation is shown in appendix B.2. Because non-adjacent nodes n_i, n_j are discovered cleverly, and deleting part of the ring is implemented in constant time, the runtime is $O(|r| + |m(r)|)$, where $m(r)$ denotes the edges connected to nodes of the ring. It is also important to note that minimization is not deterministic, meaning that there could be multiple possible minimal rings that can be returned based on an initial non-minimal ring. As a ring being minimal is a local property, the minimization can even result in minimal rings of different lengths given the same input.

This algorithm makes minimal rings very useful, as any ring that has a *polarity* of *true* can be transformed into a minimal ring that uses a subset of the nodes of the original ring. In the context of this work, a minimal ring 'dominates' its non-minimal counterpart, in the sense that non-minimal rings can be discarded in favor of their minimized counterpart.

The set of all possible minimal rings is called $R^*_m \subseteq R^*_{ni} \subseteq R^*$. There may be an exponential number of minimal rings in a graph. Such an instance is shown in fig. 24, where a ring might take the outer or inner node in every diamond. As the construction of every diamond requires 3 nodes, there are $2^{\frac{|N|}{3}}$ minimal rings. Note that these minimal rings all have the same length.

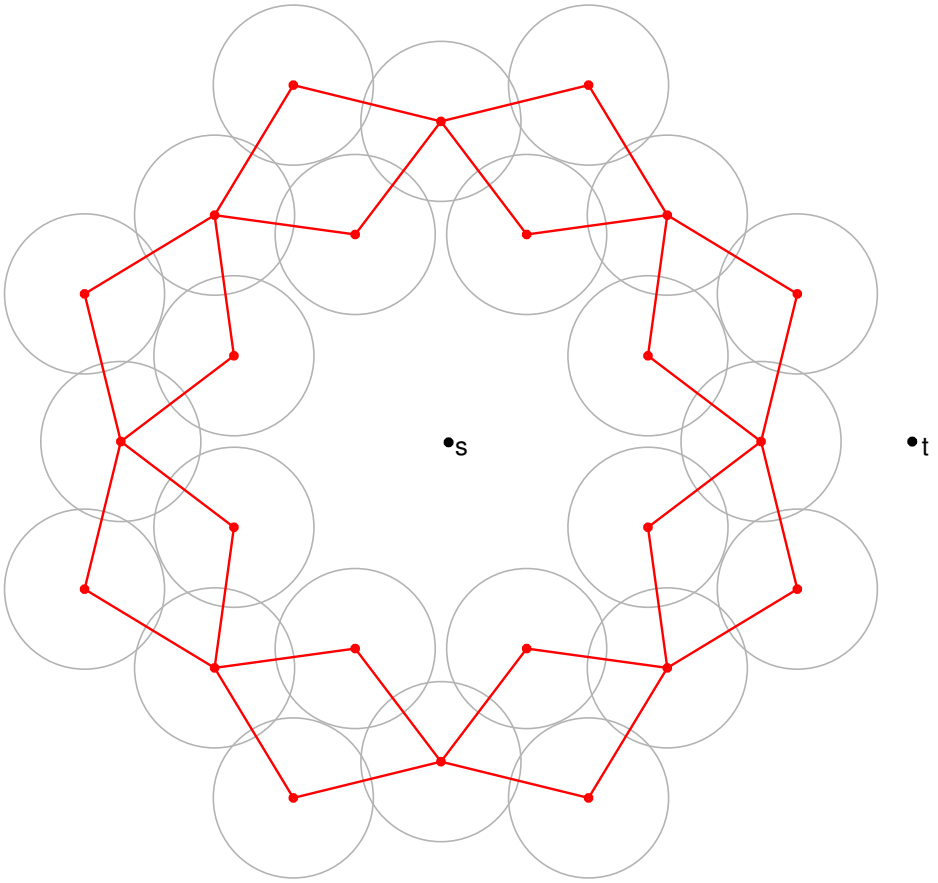


Figure 24: There are 2^8 minimal rings in the coverage graph (red), because they can take either the outer or inner node in every diamond.

7.2 Finding a ring

In this section, three algorithms are presented, which always find a minimal ring when s and t are separated. The first method looks at the 2D plane with the actual disks, taking their positions into account. The second one does the same but looks only at the coverage graph embedded in the 2D plane. The rings found by these algorithms will be transformed into a minimal ring, which is important to prove that there is always a minimal ring if s and t are separated by disks or CG respectively.

The third algorithm only uses the coverage graph CG with the mapping $polarity : E \mapsto \{true, false\}$ to always find a minimal ring, if such a ring exists. As the first two algorithms prove that there is always such a minimal ring if s and t are separated, this third algorithm will also always find a ring if s and t are separated. Thereby, it will show that this representation of the problem contains enough information to find a ring, which will later be shown to suffice for solving the barrier resilience problem optimally. The first and third algorithm are also practically implemented and will be useful in some algorithms for solving or approximating barrier resilience presented later.

While it might seem obvious that a ring exists in CG if s and t are separated, that is not true for any graph in the 2D plane, see fig. 25. Therefore, some geometric properties of the problem have to be taken into account. Even when looking at real examples of the barrier resilience problem, it is not trivial to find rings.

A naive algorithm might start going along the curve $c(s,t)$ until it hits an edge in CG. It would then follow this edge going left and continue taking the leftmost edge whenever it reaches the end of the previous edge. The idea is that it will eventually reach the first edge again, thereby finding a ring. But always going left in the graph might miss out on large parts of the graph, as shown in fig. 26, where the algorithm only explores two separate parts of the graph, depending on which edge is hit first.

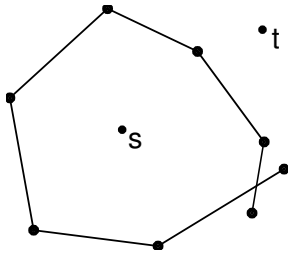


Figure 25: There is no ring in the graph even though s and t are separated

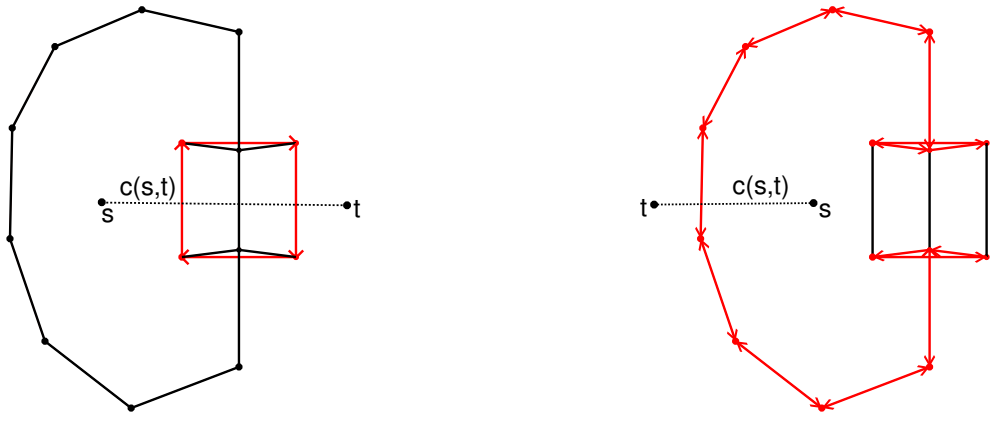


Figure 26: Traversing CG by starting at an edge on $c(s,t)$ and always going to the leftmost edge misses the ring separating s and t, independent of $c(s,t)$

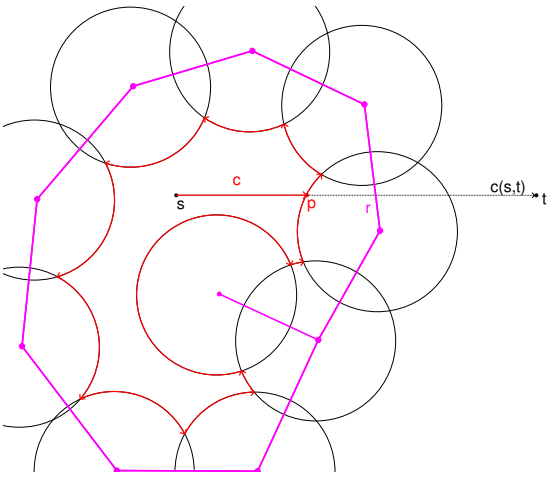
7.2.1 Disk view

The first real algorithm for finding rings in CG will look at the disks drawn in the 2D plane.

Algorithm 2 Finding a ring: Disk view

The algorithm constructs a curve c by traversing the 2D plane. Start by going along $c(s,t)$ from s to t until a disk is hit in point p. Now go clockwise along the rim of this disk. If another disk is hit, continue by going clockwise along that disk, and so on. Note that when going along disks this way, the starting point p is eventually reached again. If, however, a point p', that is further along $c(s,t)$ is reached while going around the disks, so that t lies to the left, stop going along the disks and continue along $c(s,t)$. Repeat this procedure when a new disk is hit again. If s and t are separated, a loop is eventually entered that reaches p again. If s and t are not separated, t is eventually reached and c represents a path connecting s and t without crossing any disks.

(a) A loop is found as s and t are separated



(b) s and t are not separated, c connects them without crossing any disks

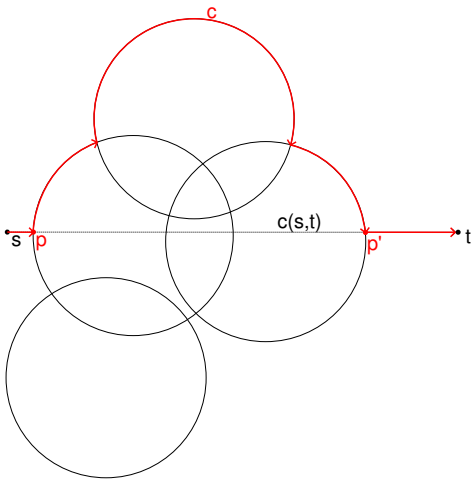


Figure 27: The algorithm 'walking along' the disks

If a loop is reached, this closed curve that originates and ends in p is called c_{loop}. The sequence of disks that were walked along forms a closed walk and therefore a ring r, which will be minimized and returned.

It needs to be proven that the algorithm always returns a minimal ring r with $polarity(r) = true$ if s and t are separated by the arrangement of disks.

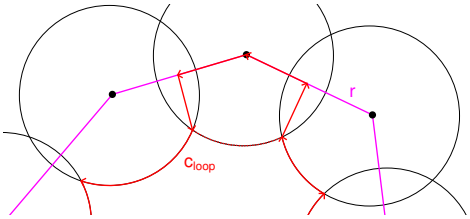
Proof. If s and t are separated, the curve c eventually reaches a loop c_{loop}. This closed curve has the following property:

There is a coherent area to the left side of c_{loop}, in which s lies, and a coherent area to the right side of c_{loop}, in which t lies.

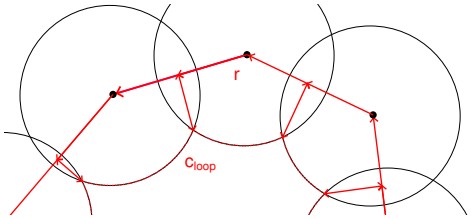
$\rightarrow$ c_{loop} does not contain any intersections

$\rightarrow$ c_{loop} crosses $c(s,t)$ an odd number of times (compare Lemma 7)

What is not obvious is that the associated walk in the graph r also crosses $c(s,t)$ an odd number of times. Tracing this ring gives a contracted version of c_{loop}. This contraction can be performed as shown in fig. 28, where c_{loop} is iteratively replaced by the curve of r, disk by disk. This does not change the polarity of the curve via Lemma 5, as the area between the new and old curve is entirely covered by the disk, so it cannot contain s or t. Therefore, the ring r still crosses $c(s,t)$ an odd number of times and can therefore be minimized using algorithm 1, resulting in a minimal ring separating s and t.



(a) c_{loop} (red) is replaced by r (purple) one disk at a time



(b) The remaining stubs of c_{loop} (red) can be removed, as they contain no area

Figure 28: c_{loop} is transformed into r, while keeping its polarity

□

There is a really nice property of the ring r even before minimization, which was not used and is not proven here: There are no intersections in r. It might utilize some nodes twice, but there is always a coherent area to the left side of r that contains s.

Although this algorithm sounds rather theoretical, as the disks are 'walked along', implementing it is not that abstract. Going along the rim of a disk and choosing the neighbor that is hit first simply imposes an order on the neighbors, according to the angle of the intersection point that is first hit going clockwise along the rim:

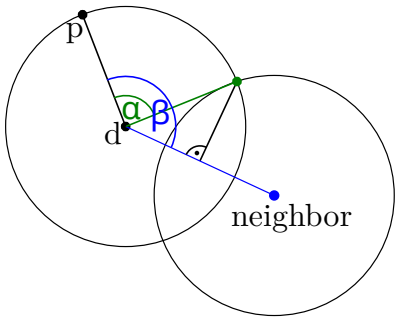


Figure 29: The angle α to the intersection point depends on β and the distance to the neighbor

Assume the disk d is entered from a point p. Now going to the 'leftmost' neighbor represents an ordering according to the angle $\beta = \angle(p\ d\ neighbor)$. The angle to the first hit intersection point is $\alpha = \beta - acos(\frac{|\overline{d\ neighbor}|}{2})$. This subtraction that depends on the distance between d and its neighbor is the difference between going to the leftmost edge in the graph and going along the disks D and is the reason why no edges in the ring r intersect.

An interesting aspect of this algorithm will later become relevant: While the curve c_{loop} along the disks is unambiguous and therefore the closed walk r will always be the same, no matter the choice of $c(s,t)$, the following minimization is not deterministic, and therefore it is not certain which minimized ring r_{min} the algorithm will actually return given a fixed problem instance.

The implementation is shown in appendix B.3.1. It has a runtime of $O(|N|\log(|N|) + |E|)$ or $O(|D|^2)$ when only considering the number of disks.

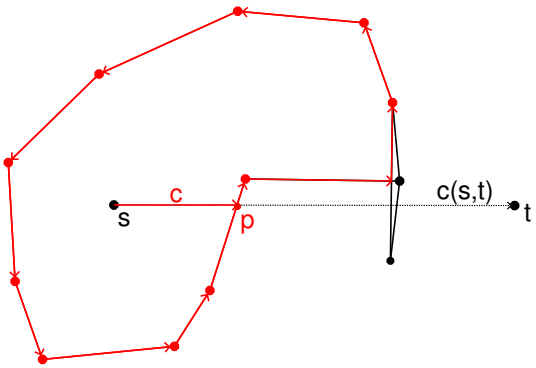
7.2.2 Graph view

This algorithm will consider the coverage graph CG embedded in the 2D plane. The idea is similar to the previous algorithm, but the curve c goes along the graph CG instead of going along the disks.

Algorithm 3 Finding a ring: Graph view

Walk along $c(s,t)$ until an edge is hit at the point p, then go left to follow this edge. If another edge is hit, turn left to follow this new edge. If the end of an edge is reached, continue going along the leftmost edge. The algorithm thereby follows a border bounding the area containing s. If a point further along $c(s,t)$ is hit, so that t lies to the left, continue going along $c(s,t)$. If no such point is reached, p will eventually be reached again, resulting in the closed curve c_{loop} starting and ending at p (fig. 30 (a)). Now consider the nodes that c_{loop} has walked along. Notice that sometimes the curve goes from one edge e_1 to another edge e_2 in an intersection, so it does not traverse any node at this point. In this case, add the endpoint of e_1 that would have been reached if the curve had not turned left and the node where e_2 originates, considering the direction in which the curve travels along the edge (fig. 30 (b)). The resulting sequence of nodes is a closed walk in the graph and therefore a ring r, which has $polarity(r) = true$. This ring can therefore be minimized using algorithm 1, resulting in a minimal ring separating s and t.

(a) c follows the inner edge of the graph by always going left



(b) when c_{loop} goes along an intersection, the ring r goes through the endpoint of the edges

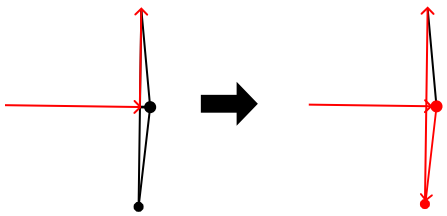


Figure 30: The algorithm 'walking along' the coverage graph

If s and t are separated in CG, the algorithm always reaches a loop with $polarity(c_{loop}) = true$. For this closed curve a sequence of nodes $r = (n_0, n_1, n_2, ..., n_0)$ is returned, but it needs to be proven that the edges between the nodes even exist and that the $polarity(r) = true$.

Proof.

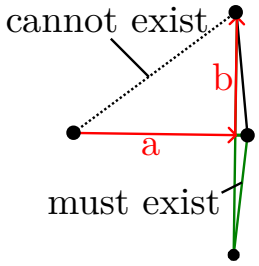
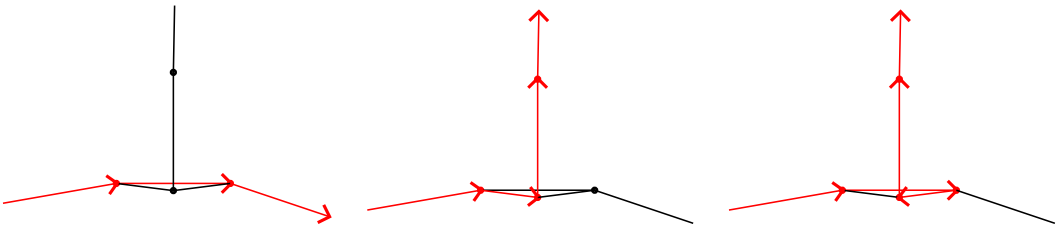


Figure 31: Transforming c_{loop} (red) to r adds the green triangle

When c_{loop} (red) travels along an intersection of two edges a, b, their endpoints are added to the sequence r. The endpoints on the other side of the edges may not be connected in CG, because then the algorithm would not travel along this inner edge. According to Lemma 4, there is thus an edge between the endpoints that are connected in the walk. As the transformation of the curve c_{loop} to this walk always adds a small triangle (green), which cannot contain s or t, its polarity stays the same (Lemma 5).

□

This algorithm is similar to the disk view. However, the disk view does not always take the first intersecting edge and might therefore leave some nodes out, resulting in the ring containing fewer nodes before minimization. The disk view is also easier to implement and more efficient because there are usually more intersecting edges than there are edges themselves, which is why this algorithm is not implemented. Figure 32 shows the different approaches: (a) always go left in the primal graph, which causes it to completely miss the branch to its left. This was previously shown in fig. 26, where it caused the algorithm to completely miss a ring separating s and t. (b) is the walk of the disk view, which contains no intersections, as it does not always take the leftmost edge. (c) shows the graph view which always takes the leftmost edge, but then goes back to take the branch going upwards, resulting in a walk which contains an intersection and one more node than that of the disk view.



(a) Always 'go left' in the graph (b) Walk of the Disk view (c) Walk of the Graph view

Figure 32: The 3 walks of the proposed algorithms

7.2.3 Disregarding the 2D plane

The following algorithm will utilize the fact proven by the previous algorithms, that if s and t are separated, there exists a minimal ring r_{min} that separates the points. This allows for an algorithm, that only uses the graph $G = (N, E)$ and the mapping $polarity : E \mapsto \{true, false\}$. It will look for the shortest possible ring $r_{shortest}$ that has $polarity(r_{shortest}) = true$. As any ring r with $polarity(r) = true$ can be transformed into a minimal ring r_{min} with $polarity(r_{min}) = true$ using a subset of the nodes of r, the shortest ring will always be minimal. Note that there typically are many equally long rings, with the algorithm finding one of them.

Algorithm 4 Finding a ring: Without geometric information

Consider all nodes as the root node $\forall n_r \in N$:

1. Calculate the shortest path tree T with the root n_r
2. Consider every pair of nodes $n_1, n_2 \in N$, which are connected in CG, but not in T:
   (a) Connect n_1 and n_2 to form a cycle in T called r.
   (b) If $polarity(r) = true$, save r if it is the shortest ring found so far.

Return the shortest found ring.

As the algorithm only finds rings with $polarity(r) = true$, if s and t are not separated, no such ring can be found, and the algorithm will return none. However, it still needs to be proven that the algorithm always finds a shortest minimal ring if it exists.

Proof. Consider a shortest minimal ring r_{min}. Take any node of this ring as the root node n_r. Now the proof is split for the case where $length(r_{min})$ is *odd* (a) or *even* (b).

(a) $length(r_{min})$ is odd (b) $length(r_{min})$ is even

Figure 33: The shortest ring r_{min} is split into two paths π_1 and π_2 originating in n_r of equal length

(a) Consider the two nodes n_1 and n_2 that are farthest away from n_r and the paths from n_r towards them π_1, π_2. Now, there cannot be a path π' from n_r to n_1 shorter than π_1 with the same polarity, because replacing π_1 with π' would result in a ring shorter than r_{min}. Similarly, there cannot be a shorter or equally long path π' with the opposite polarity as π_1, as combining π_1 and π' would form a ring shorter than r_{min} with a *polarity* of *true*. The same goes for n_2 and π_2. Therefore, the paths π_1 and π_2 or equally long paths with the same polarity are present in the shortest path tree T. Since n_1 and n_2 have the same distance from n_r, the edge between n_1 and n_2 is not present in T, and the ring created by connecting the shortest paths towards n_1, n_2 with this edge is considered. Thus, this ring or an equally long ring is returned.

(b) Now there is a single node n_3 that is farthest away from n_r. It has two paths π_1 and π_2 with opposite polarities connecting it to n_r. There obviously cannot be a shorter path from n_r to n_3, so a path with the same length needs to be present in the shortest path tree. Assume WLOG that the polarity of the path from n_r to n_3 in the shortest path tree is the same as the polarity of π_1. In the shortest path tree, the path to the second last node of π_2, called n_2, has the same length and polarity as π_2 for the same reason as in (a). Therefore, the ring formed by connecting n_2 and n_3 in the shortest path tree is considered by the algorithm, and it, or an equally long ring, is returned.

□

The implementation of the algorithm is shown and analyzed in appendix B.3.2. It uses very simple operations in each step and can be optimized for average cases. The runtime is significantly worse than that of the disk view however, as it is $O(|N|(|N|+|E|))$, or $O(|D|^3)$ when only considering the number of disks in the instance.

7.3 Problem definition

Now that the properties of rings have been explored and ways of finding rings have been shown, the notion of rings will be used to actually solve the barrier resilience problem. Therefore, the ring problem is introduced, which uses the concept of rings to describe the barrier resilience problem without trying to find a curve c between s and t, which the previous algorithms in the arrangement focused on.

Definition 12. *Ring problem:*
Find the minimum set of nodes $I \subseteq N$, so that I contains a node of every minimal ring with a polarity of true:
$\forall r \in R^*_m : polarity(r) = true \implies I \cap r \neq \emptyset$.

Now the equivalence between the ring problem and the barrier resilience problem is proven. Note that it does not matter if only minimal rings or all rings are included in the definition above. But only accounting for minimal rings reduces the number of rings and therefore makes the problem easier to solve.

Theorem 4. *A set of nodes I is a solution to the ring problem $\iff$ I is a solution to the barrier resilience problem.*

Proof. A solution to the barrier resilience problem has to contain a node of every possible ring separating s and t, as the disks of a ring that remains active obviously block any curve from s to t. Any ring with a *polarity* of *true* separates s and t (Lemma 6).

To prove the inverse direction of the implication, assume that there is a solution I for the ring problem where s and t are still separated by disks $D \setminus I$. The previously presented algorithm 2 can find a minimal ring in CG_{-I} when s and t are separated by the set of disks $D \setminus I$, so I does not cover every possible minimal ring.

□

Note that the same theorem and proof can be applied to the barrier resilience problem in the coverage graph:

Theorem 5. *A set of nodes I is a solution to the ring problem $\iff$ there is a curve from s to t that does not intersect any edges in the graph CG_{-I}.*

Proof. I must contain a node of every ring, as a ring with no disabled node cannot be crossed by any curve. In the other direction, algorithm 3 has shown how to find a minimal ring r with $polarity(r) = true$ when s and t are still separated in the coverage graph CG_{-I}. □

With these two equivalences, the previous proof that there exists a curve between s and t that does not intersect any edge in CG_{-I} if and only if there exists a curve that does not intersect any disk in $D \setminus I$ (Theorem 3) becomes redundant:
ring problem $\overset{Theorem\ 4}{\iff}$ barrier resilience $\overset{Theorem\ 3}{\iff}$ barrier resilience in CG $\overset{Theorem\ 5}{\iff}$ ring problem
Because of the equivalence, the minimum set I_{min} that covers every possible ring separating s and t is the optimal solution to the barrier resilience problem, so $k = |I_{min}|$. As rings can be

found using only $CG = (N, E)$ and $polarity : E \mapsto \{true, false\}$ (algorithm 4), this shows that this representation of the problem instance contains enough information to solve the barrier resilience problem.

7.4 Naive algorithm

The ring problem has an obvious algorithm that always finds the optimal solution I_{min}:

Algorithm 5 Ring problem: Naive algorithm

Start with an empty set of rings $R = \emptyset$. Repeat:

1. Find a minimum hitting-set I that covers every ring in R.
2. Construct the graph CG_{-I} by removing every node $n \in I$ and its associated edges from CG.
3. Find a minimal ring in the graph CG_{-I} and add it to the set of Rings R. Terminate if no ring can be found.

This algorithm will always terminate as there is a limited number of possible rings, and no ring is found twice, as every previously found ring has a node $n \in I$. When the algorithm terminates, there does not exist a ring that is not covered by I while I is minimal, thereby finding an optimal solution $|I| = k$ according to Theorem 4. The algorithm may however take exponential time, as there may be an exponential number of possible rings. Without proving this here, before the size of the hitting-set increases by one, an exponential number of rings might have to be discovered, if the rings are discovered in an unfortunate order and an unfortunate hitting-set is chosen. This can also be the case if the shortest possible ring is always added to the constraints. Regardless of whether the number of rings is exponential, the hitting-set problem is generally NP-hard, meaning that it cannot be solved generally in polynomial time even if the number of sets is not exponential. This applies to geometric hitting-set problems as well [16].

7.5 Ring problem as an ILP

The ring problem can also be formulated as an ILP. There is an integer variable a_n for every node $n \in N$ that denotes whether n is in I: $a_n = 1 \; if \; n \in I, \; else \; 0$. The objective is to minimize the size $|I|$ under the constraint that for every possible ring $r \in R^*_m$, the sum of the activation of all nodes in r must be ≥ 1. This means that at least one node of every ring must be contained in I:

$$\begin{aligned} \text{minimize} \quad & \sum_{n \in N} a_n \\ \text{subject to} \quad & \forall r \in R^*_m : \sum_{n \in r} a_n \geq 1 \end{aligned} \tag{6}$$

This ILP simply states the minimum hitting-set problem, with the added difficulty that there may be an exponential number of rings and therefore an exponential number of constraints. The naive algorithm can be implemented using this ILP, starting with an empty set of constraints ($R^*_m = \emptyset$) and iteratively adding constraints that the current ILP solution violates and then resolving the ILP. Constraints in the context of this ILP are synonymous with rings that have to be covered by the hitting-set I.

7.5.1 LP relaxation

The ILP can be relaxed by allowing non-integer values for the variables: $0 \leq a_n \leq 1$. There is still the problem that there might be an exponential number of rings and therefore an exponential but finite number of constraints. The ellipsoid method can circumvent this problem. Instead of defining the constraints prior to solving the LP, the ellipsoid method gives a possible solution to the current set of constraints, starting with an empty set of constraints. Then it requires a separation oracle that returns a violated constraint for the given solution. The algorithm for this oracle is described in the next section and works in polynomial time. In total, the ellipsoid method guarantees that the number of calls to this oracle is polynomial until the optimal solution to the LP is found [9]. Because of the fixed dimensionality of the problem, the number of constraints that define the optimal solution is limited, even if there is an exponential number of constraints.

Now that it has been shown that the LP relaxation of the ring problem can be computed in polynomial time, it is still not clear how good the result of it is. The LP relaxation of the hitting-set problem is generally not a very good approximation of the integer solution, and is far away from having a constant approximation factor [12]. But this does not prove that the approximation bound cannot be better in the ring problem, with the ring problem being strongly constrained by its geometry. Some lower bounds on the result of the LP relaxation will be theorized later on.

As the ellipsoid method is not easy to implement, the actually implemented method uses a standard LP solver to find the optimal solution for the current set of constraints and asks the oracle to come up with a violated constraint for this solution. After adding the constraint, the LP solves the problem again. This is repeated until no possible constraint violates the LP solution, meaning that the optimal solution to the LP has been found. This method does not guarantee that the LP solver takes polynomial time and does not guarantee that the number of constraints required is sub-exponential, but works well enough in practice.

7.5.2 Seperation oracle

It still needs to be shown how the separation oracle for the ellipsoid method works. The goal of the separation oracle is to find a constraint, which a given solution to the LP violates. The current LP solution gives activations a_n for every node in CG. A constraint that is violates is just a minimal ring r, with $polarity(r) = true$, whose nodes have an activation < 1: $\sum_{n \in r} a_n < 1$. To make the terminology of the algorithm consistent with similar algorithms, the activation of a node a_n is renamed to the cost of a node $cost(n)$ and the sum of the activation of the nodes in a path / ring will be called the $cost(\pi)/cost(r)$. Instead of searching for any ring with $cost(r) < 1$, a ring with minimum cost r_{min} and $polarity(r_{min}) = true$ will be searched. If the lowest-cost ring has a cost ≥ 1, there obviously is no ring with a cost < 1. If the cost of every node is positive, which can be achieved by adding an infinitesimally small cost to every node that has $cost(n) = 0$, the lowest-cost ring will automatically be a minimal ring, which makes for a stricter constraint than a non-minimal ring and thus helps the algorithm to converge faster.

The problem of finding a ring with minimum cost seems similar to finding the lowest cost cycle in the cycle-canceling algorithm for finding the minimum cost flow in a graph. However, there is the unique requirement that a cycle/ring needs to have a *polarity* of *true*. On the other hand, the problem is considerably simpler since all costs are positive. As far as I know, this problem is novel and therefore requires a custom algorithm to solve it efficiently. The idea is equivalent to the previously shown algorithm for finding the shortest ring (algorithm 4) when replacing the shortness of a ring with its cost.

Algorithm 6 Finding the minimum cost ring

For every node as the root node $\forall n_r \in N$:

1. Build the minimum cost tree T with root n_r
2. For all nodes $n_1, n_2 \in N$ that are connected in CG, but not in T:
   (a) Connect n_1 and n_2 to form a cycle in T called r.
   (b) If $polarity(r) = true$, save r if it is the lowest cost ring found so far.

Return the lowest cost ring that has been found.

The algorithm obviously returns the lowest-cost ring with a *polarity* of *true* that it finds. However, it is not immediately obvious that the lowest cost ring is always found at all by the algorithm, as there are more possible rings in CG than there are rings formed by connecting nodes in T (there might be an exponential number of rings, but only $O(n^3)$ rings can ever be considered). The proof is also very similar to the previous proof for finding a shortest ring.

Proof. For the proof, look at the lowest cost ring r and take any node of the ring as the root node n_r. Find the two connecting nodes n_1, n_2 that are adjacent in the ring, so that the path π_1 to n_1 has lower cost than the path going around the other side of the ring ($cost(\pi_1) < cost(\pi_2) + cost(n_1)$) and the same is true for n_2. A ring split like this is visualized in fig. 34.

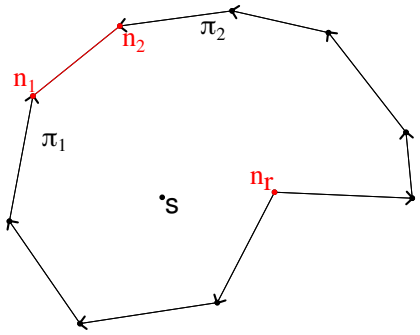


Figure 34: The lowest cost ring is split into paths π_1 and π_2, so that reaching n_2 through π_1 costs more than $cost(\pi_2)$ and reaching n_1 through π_2 costs more than $cost(\pi_1)$.

This optimal ring is found by the algorithm if the two paths π_1, π_2 are also present in the minimum cost tree T with root n_r, as the algorithm would then consider this ring by connecting n_1 and n_2. As the ring is optimal, there cannot be a lower cost path π_1' with the same polarity from n_r to n_1, as this would form a lower cost ring together with π_2. If there were a lower or equal cost path π_1'' with the opposite polarity, taking this path together with π_1 would result in a ring r_{better} with $polarity(r_{better}) = true$ and $cost(r_{better}) \leq 2 * cost(\pi_1) - cost(n_1)$ (the cost of n_1 is subtracted as it is counted in both $cost(\pi_1)$ and $cost(\pi_1'')$). By the definition of the split between n_1 and n_2, $cost(\pi_1) < cost(\pi_2) + cost(n_1) \rightarrow cost(r_{better}) < cost(\pi_1) + cost(\pi_2)$. This would contradict the initially considered ring being the lowest cost ring, so there cannot be such a path π_1'' either. Therefore, π_1 or a path of equal cost with the same polarity is present in T with the same argument applying to π_2 as well, meaning that this optimal ring or one with equal cost is discovered by the algorithm. □

The implementation of the algorithm is shown in appendix B.3.3. It is almost equivalent to the previous algorithm for finding the shortest ring, but has a slightly increased runtime of $O(|N|(|N|\log(|N|)+|E|))$. When only considering the number of disks of the instance, the runtime complexity is $O(|D|^3)$, which is the same as for finding the shortest ring.

7.5.3 Using the LP result

In practice, all variables typically have integer values, even for the relaxed LP. In that case, the LP solution is obviously the optimal ILP solution, so an optimal solution to the barrier resilience problem has been found. The fact that the values are integers also proves that this optimum k has been found.

If some variables have non-integer values, their values are fractions like $\frac{1}{2}$ or even fractions with a seemingly arbitrary denominator like multiples of $\frac{1}{7}$, so that the result may be $\frac{1}{7}$ smaller than the whole number ILP solution. Fractional optimal values lp deliver a lower bound for k: $k \geq lp$, but as k is always integer, the optimal LP value can even be rounded up: $k \geq \lceil lp \rceil$. When the assignment of every variable of a solution is rounded up, the resulting objective function value gives an upper bound on k, as there cannot be a ring with activation < 1 when there was no ring with activation < 1 for the lower variable values. However, there is a smarter way to do this, which gives a lower (thus better) upper bound, which will be referred to as 'smart rounding':

A constraint can be added for the variable with the highest activation variable a_d, which forces it to be 1: $a_d = 1$. Then the LP is solved again with the ellipsoid method, which might require additional ring constraints being added. This is repeated for every activation variable that does not already have this constraint until all variables have integer values. Because at no point did an ILP have to be solved, the algorithm still works in polynomial time. The result is an upper bound on k, as there is no ring with activation < 1 when the algorithm is finished and all activation variables have integer values. However, there may be a better optimal solution to the ILP, which has now become infeasible because of the additional constraints $a_d = 1$, that are not part of the ring problem itself. I could not come up with an arrangement of disks where this rounding scheme did not return the optimal solution k, so it remains to be proven whether this 'ring problem LP + smart rounding' algorithm actually solves barrier resilience optimally.

For an algorithm that is guaranteed to return the optimal solution k, the result of the LP relaxation can be taken as the basis for solving the ILP. Therefore, the optimal solution of the LP with the rings that have been added to the LP is used and solved as an ILP. However, the constraints that define the LP optimum may allow a solution to the ILP problem, which is not allowed due to some other constraint that is not tight for the LP solution. This means that the polynomial number of rings added during the ellipsoid method may not be sufficient to define the ILP solution, meaning that they may allow an integer solution ilp where $ilp < k$. If there is a ring that is not covered by the ILP solution, it can be found using one of the algorithms presented and added to the ILP, which is then resolved. So this algorithm can be understood as using the LP relaxation for generating an initial set of rings, which are then used by the naive algorithm to solve the problem optimally. This way, eventually k will be found. As the LP solution is always very good in practice, the ILP solution can be calculated very fast. During tests, it was never the case that the constraints for the LP allowed an infeasible ILP solution, so the naive algorithm never actually added any rings. The opposite was usually the case, where running just the naive algorithm comes up with fewer constraints than the LP relaxation, and the constraints of the naive algorithm did allow an infeasible solution to the LP relaxation. This is a kind of Las Vegas algorithm, that always returns an optimal solution, it is just the runtime that might be very long. Unlike a true Las Vegas algorithm, the algorithm is not randomized, and its runtime depends entirely on the problem instance.

7.6 Node-disjoint rings

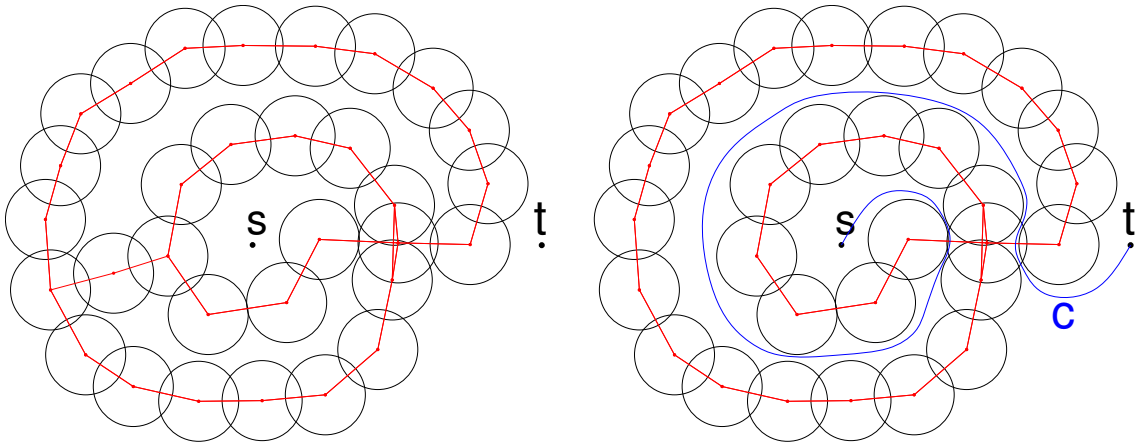
An intuitive way of solving the ring problem is to look for the maximum possible number of node-disjoint rings, leading to the following hypothesis:

Hypothesis 1. *The maximum number of node-disjoint rings separating s and t equals k.*

Obviously it only makes sense to look for minimal rings, as they use a subset of nodes of non-minimal rings. In the previously mentioned conference version of [14] this is stated somewhat similarly, for the problem constraining disks to a closed belt region. This statement was retracted in the final version of the paper, and previously shown to be false in this work.

Any set of node-disjoint rings $|R_{node-disjoint}|$ is obviously a lower bound for k, as every ring $r \in R_{node-disjoint}$ separates s and t. Therefore, at least one node of every ring needs to be deactivated. As the rings are node-disjoint, at least $|R_{node-disjoint}|$ nodes need to be deactivated.

It was already shown in the closed belt version of the problem that there are problem instances where there exist less than k node-disjoint rings. The counterexample is shown again in fig. 35 (a). The construction has two intersecting edges whose endpoints are not all connected, which has previously been described as the source of complexity when working with CG. Interestingly, the construction looks like a proof that an optimal curve c may have to cross a disk twice (b), except for an additional disk that was inserted to block exactly this curve c. This hints at a kind of duality, where barrier thickness is 'pessimistic' and counts a disk each time it is entered, even if the curve crosses the same disk twice, while looking for node-disjoint rings is 'optimistic' as it always assumes the disk can be crossed in one step, even when the 'shortcut' is blocked by another disk. The intuition behind this needs to be studied further, however, in order to come up with a real understanding when the maximum number of node-disjoint rings does not match k.



(a) There are no two node-disjoint rings, despite $k = 2$

(b) *barrier thickness*$= 2$, despite $k = 1$

Figure 35: Barrier thickness and the number of node-disjoint rings seem related

Interestingly, the previously shown LP relaxation of the ring problem actually solves the problem of finding the maximum number of node-disjoint rings. While the process of finding the maximum number of possible rings was theorized to take exponential time in [14], using the ellipsoid method on the LP could actually solve the barrier resilience problem perfectly in polynomial time if the

hypothesis was true: Call the optimal solution of the LP-relaxation of the ring problem lp. As in the LP, for every ring, the sum of the activation of the nodes needs to be ≥ 1, this also applies to all $r \in R_{node-disjoint}$. As the rings are node-disjoint, the activation of a node cannot count towards two rings, meaning that $|R_{node-disjoint}| \leq lp \leq k$. If $|R_{node-disjoint}|$ and k are not equal, the LP solution lp might actually be closer to k than $|R_{node-disjoint}|$. In the example in fig. 35 $lp = 1.5$. The activations of the disks and the rings that are added to the LP as constraints are visualized in fig. 44 in the appendix. Maybe this example is as bad as the approximation of lp gets, leading to the following hypothesis:

Hypothesis 2. *The optimal value of the ring problem LP lp is always larger than the middle between k and the maximum number of node-disjoint rings $|R_{node-disjoint}|$:*
$lp \geq \frac{|R_{node-disjoint}|+k}{2}$.

Independent of whether this hypothesis is true or not, bounding the number of node-disjoint rings in terms of k would be very advantageous for proving a lower bound of lp and also for understanding the ring problem better.

7.6.1 Bounding the number of node-disjoint rings

Despite Hypothesis 1 being false, the maximum number of node-disjoint rings is still often very close to k in practice. Figure 35 has shown the construction for an upper bound on $|R_{node-disjoint}|$ of $\lceil \frac{k}{2} \rceil$, which can be scaled to any k by building the same construction around the previous ones, like layers of an onion. If this is the worst case, then the following hypothesis holds:

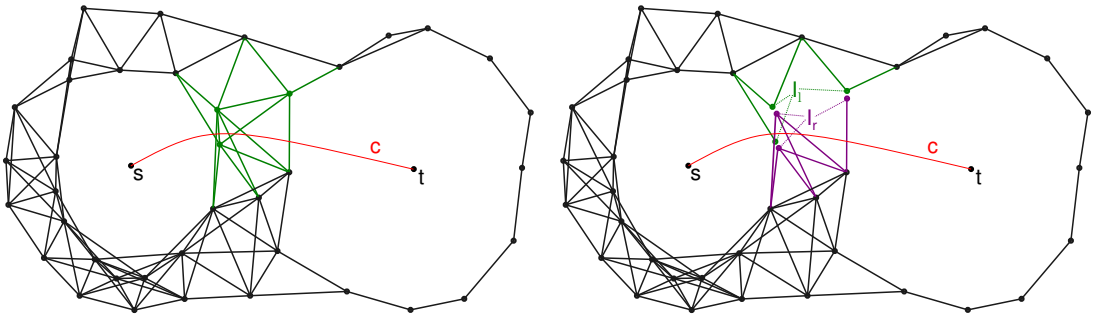
Hypothesis 3. *For an instance with barrier resilience k, there are at least $\lceil \frac{k}{2} \rceil$ node-disjoint rings in the coverage graph CG that separate s and t.*

Although not even this hypothesis can be proven here, another even stricter hypothesis is also proposed:

Hypothesis 4. *For any problem instance, there are at least $\lceil \frac{barrier\_thickness}{2} \rceil$ many node-disjoint rings in the coverage graph CG that separate s and t.*

As $barrier\_thickness > k$, this hypothesis implies the previous one.

The following will present the idea for a proof, with the part left out that has not been proven yet. The minimum solution I_{min} consists of k nodes. By removing every node in this set from CG, a curve c between s and t may exist (fig. 36 (a)). Now every node $n \in I_{min}$ is duplicated, so that there is a set of nodes I_l and I_r. Connect all edges that were connected to a node in I_{min} that come from a node to the left side of c to I_l and all others to I_r (fig. 36 (b)). Now add a node l and connect it to every node $n_l \in I_l$ and a node r that is connected to every node $n_r \in I_r$. This new graph is called CG'



(a) The optimal solution I_{min} and its adjacent edges are colored green

(b) The nodes in I_{min} are duplicated and the edges adjacent to I are reassigned to these sets I_l (green) and I_r (red)

Figure 36: CG is transformed into CG' by duplicating nodes and reassigning edges

Now the idea is that there are k node-disjoint paths from the nodes in I_l to the nodes in I_r.

Lemma 8. *There are k internally node-disjoint paths between l and r in CG', which will be called false rings.*

Proof. According to the vertex-connectivity version of Menger's theorem, the number of internally node-disjoint paths between l and r is equal to the size of the minimum vertex cut for l and r, if l and r are nonadjacent. l and r are obviously nonadjacent, as I_l and I_r are disjoint sets. Assume that there is a vertex cut smaller than k. This vertex cut would cut every possible ring with a *polarity* of *true*, as every ring goes through I_{min} at least once. Therefore, every possible ring has at least a subpath between a node in I_l and I_r in CG'. Therefore, this vertex cut would represent a solution to the ring problem that would be smaller than k, contradicting the definition of I_{min}. □

There are k paths and $|I_l| = |I_r| = k$, so every node in I_l is connected to one node in I_r. However, these paths are called false rings, as they aren't necessarily rings in CG, because the path does not necessarily start and end in the same node $n \in I_{min}$. This becomes apparent in the example in fig. 35 (a), where for any optimal solution I_{min} with $I_{min} = 2$ two false rings can be constructed, but there cannot be two real rings. These false rings intersect, but as not all endpoints of the two intersecting edges are connected, the intersection cannot be removed. If this were not the case, the intersections between false rings could simply be removed, resulting in k real rings, which would prove the incorrect Hypothesis 1. But it might still be true that the false rings can be transformed into a certain number of real rings, which the following lemma states:

Lemma 9. *The k false rings in CG' can be converted into $\lceil \frac{k}{2} \rceil$ real rings in CG*

Despite much thought, this lemma has not yet been proven, although there is also no known example where this was not the case.

7.6.2 Greedy algorithm

As the maximum number of node-disjoint rings proves to be quite a good approximation for k in practice, it is still worthwhile trying to find this number. As previously mentioned, the LP of the ring problem manages to find it in polynomial time, but it is not easy to implement, and the runtime might still be bad in practice. Therefore, the following presents an algorithm that greedily searches for node-disjoint rings, thereby approximating the maximum number of node-disjoint rings:

Algorithm 7 Greedy algorithm for finding node-disjoint rings

Start with an empty set of Rings $R = \emptyset$ and the graph $CG' = CG$. Repeat:

1. Find a minimal ring r in CG' and add it to R, terminate if none can be found.
2. Remove all nodes of the ring r and their adjacent edges from CG'.

The idea is that by finding minimal rings, the rings use as few nodes as that they can and the approximation should therefore be good. How to find a minimal ring is not specified in the algorithm, and any algorithm can be used for that purpose. Obviously the previously proposed algorithms are used for the task: One looks for a ring by walking around the inner edge of the disks in the 2D plane, while the other searches for the globally shortest ring that separates s and t. The choice of algorithm can make a difference, as even a minimal ring might use nodes that could have allowed two node-disjoint rings to exist instead. In fig. 37, an example of the disk-walk algorithm is given, with the three rings that were found drawn in purple. Note that there are two highlighted edges in green, which would have allowed for an additional ring around s, if they were not both 'blocked' by the one purple ring around s. This flaw of the greedy algorithm is used in the construction in fig. 38, where the one minimal ring (purple) blocks three possible rings. If the rings use some of the nodes highlighted in green, three node-disjoint rings can be found. Both proposed algorithms might find this blocking ring, depending on their starting point and some randomness. This construction can be scaled endlessly, proving that the greedy algorithm may lead to an arbitrarily bad approximation for any k with both algorithms.

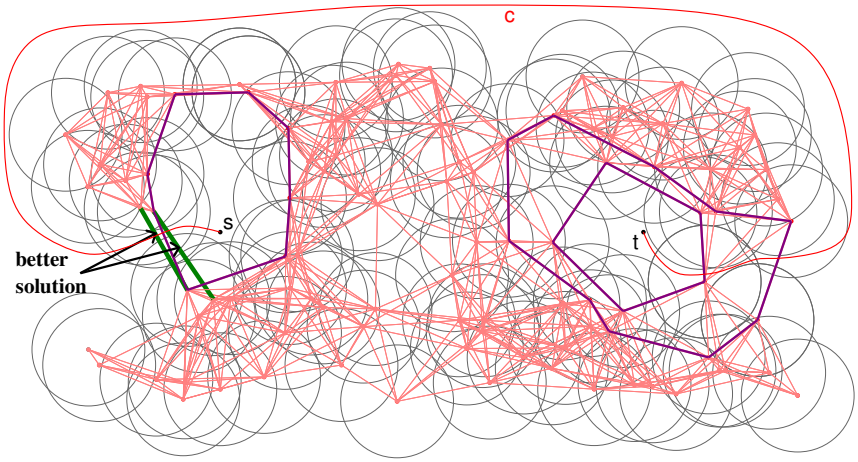


Figure 37: The greedy algorithm finds 3 rings (purple), although the optimal curve c (red) has to cross 4 disks. Using the two edges highlighted in green, 4 node-disjoint rings can exist.

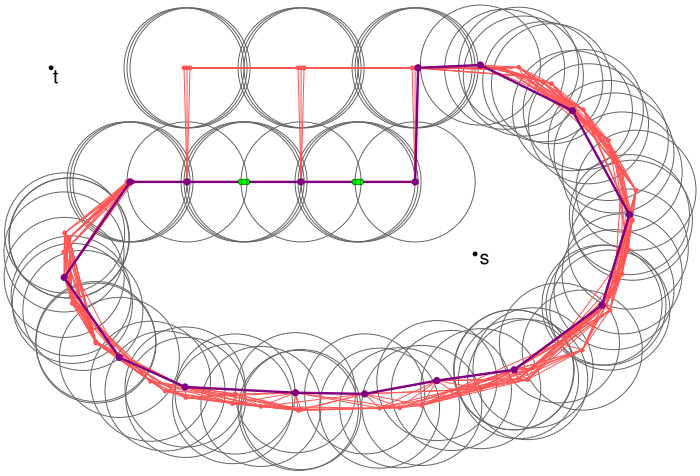


Figure 38: The greedy algorithm finds 1 ring (purple), although there are 3 node-disjoint rings when the green nodes are used. This example can be scaled arbitrarily.

In most cases, however, the greedy algorithm provides a good lower bound for k efficiently. Adding randomness to decide which ring the greedy algorithm should add first can prevent worst cases, such as the one shown in fig. 38, from happening across multiple runs of the algorithm. In the algorithm that always searches for the shortest possible ring, there are usually multiple such shortest rings. Therefore, a random one of these could be picked. The algorithm walking along the disks is deterministic, but the following minimization of the found ring is not, and randomness could therefore be introduced into this minimization. However, during testing, when multiple randomized runs of the greedy algorithm were performed on a given problem instance, the number of rings returned always remained the same, even if this was not the maximum possible number of rings.

There were, however, significant differences found between the results of the two algorithms. The algorithm searching for the globally shortest ring performed slightly better than the disk view on average. This was not always the case, and there were instances where the latter algorithm returned more rings than the former.

Any variant of this greedy algorithm can be used in order to initialize the LP or ILP with some constraints, as adding the node-disjoint rings as constraints to the ring problem LP makes it start with an optimal value $lp = |R_{node-disjoint}|$ which should be better than starting with an empty set of constraints. During testing, it was shown that the runtime of the LP was sped up by a factor between 1 and 5, even for large instances with a large k. For the naive algorithm, it is split: When the naive algorithm always takes the globally shortest ring, adding the set of node-disjoint rings at the start similarly helps a little bit. When using the disk view algorithm to find rings, the runtime could be improved by orders of magnitude, especially as the instances get larger.

8 Benchmark of algorithms

In this work, several algorithms for solving the barrier resilience problem have been developed. In total, 28 different algorithms / variations of the proposed algorithms have been implemented. In order to keep this comparison concise, only a selection of these algorithms are presented, particularly algorithm variants that dominate their counterparts in terms of approximation factor and runtime. Table 2 provides an overview of the algorithms that are benchmarked, with a brief recap of how they work.

		Short Description
Arrangement	Barrier Thickness	Barrier thickness counts every time a disk is entered, so it might count some disks twice.
	Path ILP	The Path ILP searches a path through the arrangement that enters as few disks as possible using an ILP formulation. The additional 'cell flow' and 'thickness' constraints are added, as they improve the ILP runtime and the result of the LP relaxation
	Path LP	The LP relaxation of the Path ILP, which allows the path to split into many paths.
Ring problem	Naive Algorithm Disk view / Shortest ring	Starts with an initial set of node-disjoint rings found by the "Rings greedy - Shortest ring" algorithm. Repeatedly computes the minimum hitting-set that covers every ring, then adds a ring that is not covered by this hitting-set, until no such ring can be found anymore. The minimum hitting-set is computed using an ILP. To find a ring that is not covered by the hitting-set, either the disk view (algorithm 2) or the shortest ring (algorithm 4) is used
	As LP	The LP relaxation of the ILP formulation of the ring problem, which allows nodes to be partially covered by the minimum hitting-set. It is solved by starting with a set of node-disjoint rings and iteratively adding rings to the constraint set until there are no more rings not covered by the solution.
	LP+rounding	Uses the LP relaxation and then iteratively forces the highest variable to be 1, until all variables have integer values.
	LP+ILP finish	Uses the constraint set of the LP relaxation as the basis for solving the ILP with the naive algorithm.
Rings greedy	Disk view / Shortest ring	Searches greedily for node-disjoint rings, using either the disk view algorithm for finding rings or the shortest ring algorithm.

Table 2: Overview of the algorithms that are benchmarked

The theoretical properties of the algorithms are presented and they are benchmarked on randomly generated problem instances in table 3. The instances are generated by randomly placing 500 disks in a rectangular region, in which s and t are also placed. The only requirement is that the disks do not cover s and t. An example of such a generated instance together with the optimal solution I is shown in fig. 39. The instances have an optimal solution of $6 \leq k \leq 15$.

		Theoretical			Benchmark	
		Lower bound	Upper bound	Runtime	Approximation factor	Runtime
Arrangement	Barrier Thickness	k	$2k$ ($\lvert\overline{st}\rvert \not\approx 2$) else $3k$	$O(n^2 log(n))$	1.0	103ms
	Path LP	$\frac{1}{2}k$	k ($\lvert\overline{st}\rvert \not\approx 2$)	polynomial	0.80	273s
	Path ILP	exactly k ($\lvert\overline{st}\rvert \not\approx 2$)		exponential	1.0	831s
Ring Problem	Naive - Disk view	exactly k		exponential	1.0	$>$ 1h
	Naive - Shortest ring	exactly k		exponential	1.0	10.9s
	As LP	$\frac{3}{4}k$ (Hypotheses 2 AND 3) $\frac{1}{2}k$ (Hypotheses 2 OR 3)	k	polynomial	1.0	824ms
	LP+rounding	k	maybe k	polynomial	1.0	825ms
	LP+ILP finish	exactly k		exponential?	1.0	837ms
Rings greedy	Disk view	1	k	$O(kn^2)$	0.94	16ms
	Shortest ring	1	k	$O(kn^3)$	0.96	66ms

Table 3: Overview over the most interesting algorithms

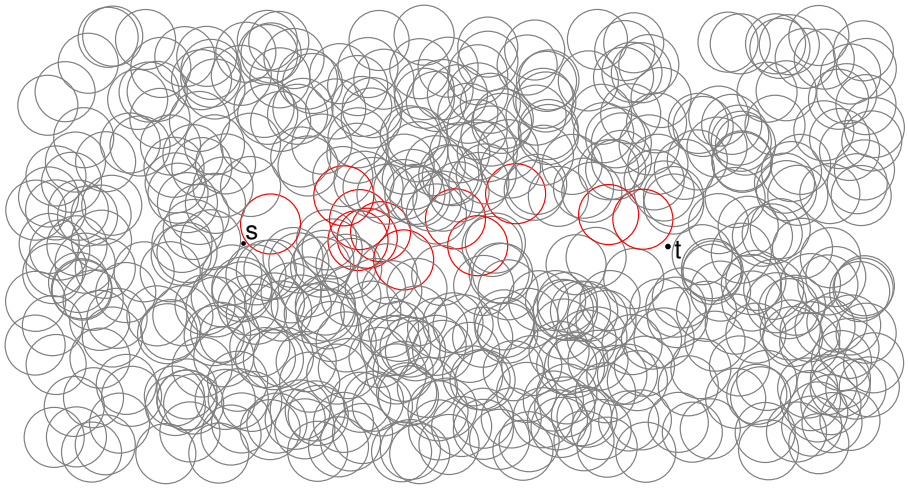


Figure 39: A problem instance as they are generated for the benchmark. The disks crossed by an optimal solution are painted red, $k = 12$

The asymptotic behavior of the algorithms has also been recorded on random instances and will be considered in the following analysis. The plots showing the runtime and approximation factor across different instance sizes are shown in appendix A.

Barrier thickness approximates k perfectly for all randomly generated instances and has a very good runtime as well, with the runtime being dominated by the construction of the arrangement graph, while finding the shortest path in the arrangement graph is very efficient. The LP that searches for a path in the arrangement graph is suboptimal even for random instances, making it significantly worse than many other approximation algorithms. The runtime is also quite high, since the LP has as many variables as there are cells in the arrangement, which can be quadratic in the number of disks. The ILP exhibits an exponential increase in runtime as the number of disks increases, making it unsuitable for large instances.

In the ring problem, the naive algorithm performs poorly, as expected. When searching rings using the disk view algorithm, it takes an unrealistically long time, even if the algorithm is initialized with node-disjoint rings. However, when using the shortest ring algorithm to find rings, the runtime is significantly better, albeit very inconsistent across runs. The LP representation of the ring problem returns the optimal solution for all randomly generated instances. Its runtime is also quite good, likely because there is exactly one variable for each disk, so the dimensionality of the LP is proportional to the number of disks. The drawback of this LP is that the constraints are not defined at the beginning, as there might be an exponential number of them, but are discovered incrementally. This forces the algorithm to solve the LP repeatedly and then search for a violated constraint. In practice, solving the LP is comparatively fast, while most of the runtime is spent on the algorithm for finding a violated constraint. Nevertheless, the runtime is much better than that of the Path LP and it is quite suitable even for large instances. Since the LP already gave an optimal result in all tests, improving the result with the proposed rounding scheme or by solving the LP as an ILP afterward did not add any runtime to the problem and could not improve the already optimal solution. There was neither a randomly generated nor a constructed instance, where the smart rounding of the LP did not yield the optimal result k, and there is no example where the ILP finish exhibits an exponential runtime.

The algorithms that greedily search for node-disjoint rings have low runtimes, being the fastest and second-fastest algorithms in the comparison. As the density of the disks increases, they both significantly outperform barrier thickness in terms of runtime. But as the density increases, the approximation ratios also become worse. The greedy algorithm using the disk view algorithm to find rings is by far the fastest. If the minimization step performed after a ring is found only shortens the ring by a constant factor, the runtime for the greedy algorithm using the disk view is actually limited to $O(n \log(n) + m)$, where n is the number of nodes and m is the number of edges in the coverage graph CG. However, its approximation is usually worse than that of the greedy algorithm that always searches for the globally shortest ring.

9 Conclusion

This work has given an overview of the barrier resilience problem along with existing approaches for solving the problem. In the arrangement graph AG, the approximation using barrier thickness introduced in [2] has been explained and implemented. As a novel approach in the arrangement graph, barrier resilience has been represented using an ILP formulation based on this graph. This ILP formulation and its LP relaxation have been implemented and studied, with the LP and ILP showing a poor runtime in practice and the result of the LP relaxation approximating the optimal solution worse than other approximation algorithms.

In the coverage graph CG, a new problem formulation has been introduced, called the ring problem. The ring problem is essentially the minimum hitting-set problem, where the sets to be 'hit' represent rings in the coverage graph. The ring problem is an improvement on the previously proposed idea of solving the barrier resilience problem by searching for node-disjoint cycles in CG. This idea was shown to be incorrect by constructing a counterexample. The novel ring problem was shown to be equivalent to the barrier resilience problem, so a solution to the ring-problem is always a solution to the barrier resilience problem and vice versa. The ring problem was formulated as an ILP, whose LP relaxation has a good runtime and provides very good approximations in practice. A rounding scheme for the LP relaxation has been proposed and implemented, which may always yield an optimal solution to the ring problem. No counterexample could be found where this is not the case. However, the optimality of the solution could also not be proven. A greedy algorithm that approximates a solution to the ring problem was also explored, which has a very short runtime, but can theoretically deliver arbitrarily bad results. The different algorithms were implemented and benchmarked on random instances.

A major advantage of the LP relaxation of the ring problem is that, unlike previous approximation algorithms, it provides a good lower bound for the optimal solution k. Combined with the existing approximations based on barrier thickness, which provide an upper bound for k, the solution for any problem instance can be bounded in both directions, with the bounds being very close in practice.

Open Questions

One aspect that is unexplored so far is how to randomly generate interesting instances. For the benchmark, disks were scattered randomly on the 2D plane, which leads some approximation algorithms to always return the optimal result over hundreds of instances generated this way. It is not trivial to generate random instances where this is not the case.

It is also important to develop a deeper understanding of the ring problem in order to prove approximation bounds for its LP relaxation. One possible approach is to try to relate the solution of the ring problem to the number of node-disjoint rings that can exist in the coverage graph. Hypotheses for this have been made, but have not yet been proven. An interesting result would be either a proof that the LP relaxation of the ring problem together with the proposed rounding scheme actually leads to an optimal solution, or a counterexample showing that it is suboptimal.

References

[1] Hilary Beaumont. Virtual wall: how the us plans to boost surveillance at the southern border. *The Guardian*, 2023.

[2] Sergey Bereg and David Kirkpatrick. Approximating barrier resilience in wireless sensor networks. In Shlomi Dolev, editor, *Algorithmic Aspects of Wireless Sensor Networks*, pages 29–40, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] Marshall Bern and David Eppstein. Approximation algorithms for geometric problems. *Approximation algorithms for NP-hard problems*, pages 296–345, 1997.

[4] Sergio Cabello. Some open problems in computational geometry (invited talk). In *Some Open Problems in Computational Geometry (Invited Talk)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[5] Sergio Cabello and Wolfgang Mulzer. Minimum cuts in geometric intersection graphs. *Computational Geometry*, 94:101720, March 2021.

[6] David Yu Cheng Chan and David Kirkpatrick. Multi-path algorithms for minimum-colour path problems with applications to approximating barrier resilience. *Theoretical Computer Science*, 553:74–90, October 2014.

[7] Vijay Chandru and M. Rammohan Rao. Linear programming. *SSRN Electronic Journal*, 1998.

[8] Vašek Chvátal. *Linear programming*. Macmillan, 1983.

[9] Daniel Dadush, László A. Végh, and Giacomo Zambelli. *On finding exact solutions of linear programs in the oracle model*, page 2700–2722. Society for Industrial and Applied Mathematics, January 2022.

[10] Reinhard Diestel. *The Basics*, pages 1–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2017.

[11] Jeff Erickson. *Algorithms*. University of Illinois, June 2019.

[12] Guy Even, Dror Rawitz, and Shimon (Moni) Shahar. Hitting sets when the vc-dimension is small. *Information Processing Letters*, 95(2):358–362, 2005.

[13] Matias Korman, Maarten Löffler, Rodrigo I. Silveira, and Darren Strash. On the complexity of barrier resilience for fat regions and bounded ply. *Computational Geometry*, 72:34–51, 2018.

[14] Santosh Kumar, Ten H. Lai, and Anish Arora. Barrier coverage with wireless sensors. In *Proceedings of the 11th annual international conference on Mobile computing and networking*, MobiCom05. ACM, August 2005.

[15] Santosh Kumar, Ten H. Lai, and Anish Arora. Barrier coverage with wireless sensors. *Wireless Networks*, 13(6):817–834, October 2006.

[16] Nabil H. Mustafa and Saurabh Ray. Improved results on geometric hitting set problems. *Discrete Computational Geometry*, 44(4):883–895, September 2010.

[17] J Cole Smith and Z Caner Taskin. A tutorial guide to mixed-integer programming models and solution techniques. *Optimization in Medicine and Biology*, pages 521–548, 2008.

[18] Kuan-Chieh Robert Tseng. *Resilience of wireless sensor networks.* PhD thesis, University of British Columbia, 2011.

[19] E.A.B.S.G. Williamson. *Lists, Decisions and Graphs.* S. Gill Williamson, 2010.

[20] Gerhard J. Woeginger. *Exact Algorithms for NP-Hard Problems: A Survey*, pages 185–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[21] S. Yuan, S. Varma, and J.P. Jue. Minimum-color path problems for reliability in mesh networks. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 4, pages 2658–2669 vol. 4, 2005.

10 Appendix

In the appendix, a more detailed benchmark is presented that visualizes the asymptotic behavior of the algorithms using plots. The actual C++ implementation of the algorithms that were proposed in this work is presented and discussed. Finally, the graphical interface, which was developed to create instances and run the algorithms on them, is shown.

Contents

A Extended Benchmark **65**

B Algorithm implementations **67**

B.1 Calculating the arrangement 67

B.2 Ring minimization 68

B.3 Finding a ring 69

B.3.1 Disk view 69

B.3.2 Globally shortest ring 71

B.3.3 Minimum cost ring 74

C Graphical interface **76**

A Extended Benchmark

In addition to the benchmark for a fixed number of disks that was presented in table 3, the asymptotic behavior is analyzed using plots. The number of disks is varied from 100 to 500 and the area where the disks are randomly distributed in was chosen so that k rises proportional to the number of disks:

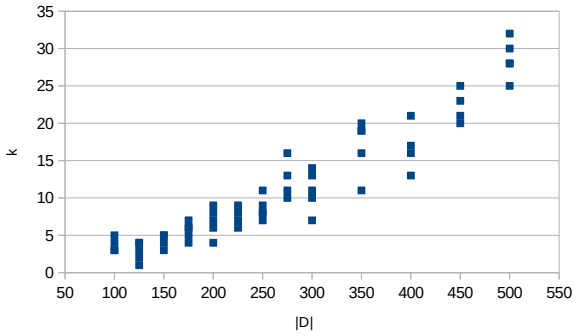
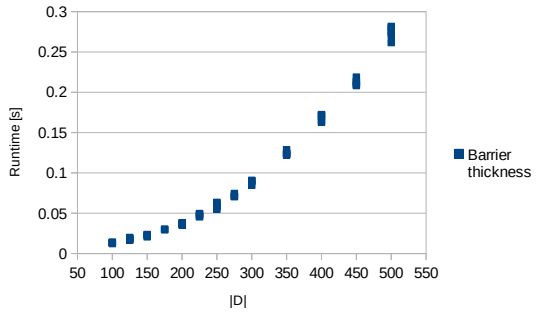
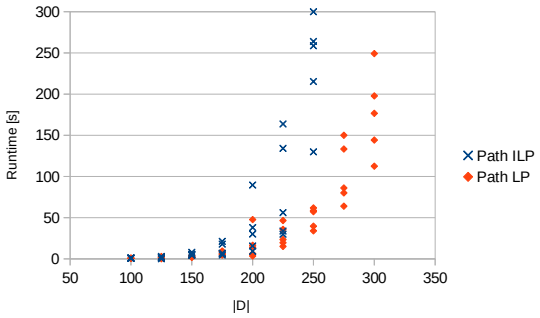


Figure 40: The barrier resilience k of the benchmarked instances. The instances were created so that k roughly rises proportional to $|D|$, except for $|D| = 100$

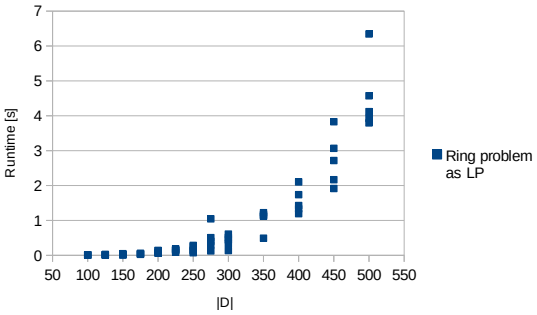
The following plots show the runtime of the algorithms that were described in table 2. Note that the runtime was capped at 300s, thus some algorithms were not run on larger instances.



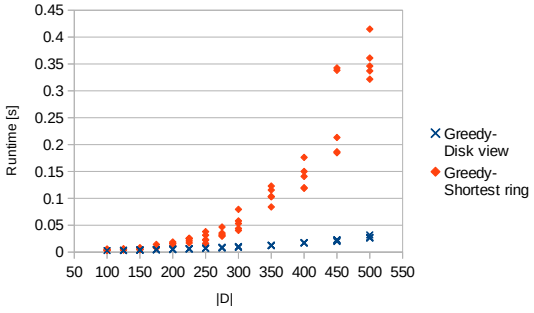
(a) Barrier thickness



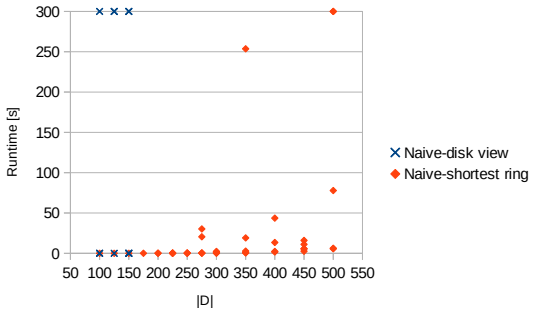
(b) Path ILP and its LP relaxation



(c) Ring problem as an LP



(d) Greedy algorithms for finding node-disjoint rings



(e) Ring problem naive algorithms

Figure 41: The runtimes of several algorithms across various instance sizes

The naive algorithms as well as the Path ILP are guaranteed to return the optimal solution, while the barrier thickness and the LP formulation of the ring problem returned the optimal solution in all generated instances without being guaranteed to do so. The Path LP and the greedy algorithms are the only algorithms that did not always return the optimal solution, so their approximation factor is shown in the following plot.

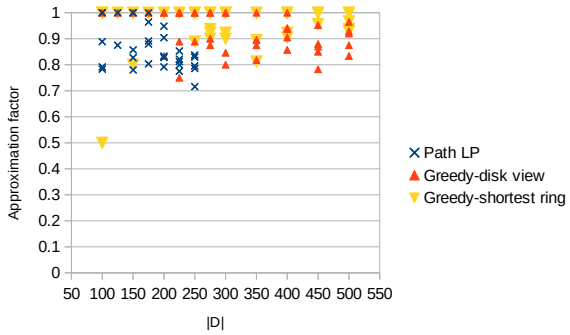


Figure 42: The approximation factors of the algorithms that did not return optimal results

B Algorithm implementations

First of all some general information about the structure of the code is needed:

- The code uses C++17 features such as std::tuple and std::optional
- In the implementations, disks are all referred to by their index, which is just an integer from $0, 1, ..., |D| - 1$.
- To get the coordinates of a disk, there is a vector containing the x- and y-coordinates of the center of every disk that can simply be indexed with this disk index, like so:

  ```
  disk_coords[disk_index].x
  ```

- The coverage graph is represented by a variable of type 'Neighbors'. This variable is a vector which for every disk contains an inner list. This inner list has an element for every edge connected to the disk. The inner list saves the index of the adjacent disk that is on the other side of the edge together with the polarity of the edge as a tuple. This representation is chosen to quickly access the neighbors of some disk as a list, in order to iterate through them:

  ```
  for(auto [neighbor_disk, edge_polarity]: neighbors[disk_index])
  ```

  The type definition looks as follows:

  ```
  typedef std::vector<std::list<std::tuple<int, bool>>> Neighbors;
  ```

- When polarities are XORed, the '!=' operator is used
- Sometimes, it is important to know the polarity of the edges of a ring. Therefore, the type 'RingWithPolarity' is introduced, which saves for every node of the ring the polarity of the path of it to the first node of the ring.

  ```
  typedef std::vector<std::tuple<int, bool>> RingWithPolarity;
  ```

B.1 Calculating the arrangement

The code for calculating the arrangement graph AG is too long and complex to show here, but the idea will be presented. Calculating the arrangement is done in a sweep-line manner, going from the left ($x = -\inf$) to the right ($x = \inf$). The sweep line stops when a new disk begins, ends, or when the rim of two disks intersect. At these events the cells are created, connected or merged. The start and stop steps might take $O(|D|)$ time, while the intersection steps take constant time. Therefore, the sweep line algorithm takes $O(i + |D|^2)$ time when traversing the plane, where i is the number of intersections. As the events have to be ordered according to their x coordinate prior to starting the sweep line, this takes $O((i + |D|)log(i + |D|))$ time. As the maximum number of intersections $i < |D|^2$, the algorithm as a whole takes $O(|D|^2 log(|D|))$ time. The sweep line also stops at the points s and t and saves the cell in which the points lie in. The implementation for calculating the arrangement might not be optimal, especially as all events are sorted before the sweep line starts, but as the resulting graph may have $O(|D|^2)$ many nodes and edges, the only factor that might be optimized is the $O(log(|D|))$, which is small enough for the context of this work, as the algorithm has no problem dealing with the benchmarked instances in under a second.

B.2 Ring minimization

```
std::list<int> minimize_ring(const RingWithPolarity& ring,
                             const Neighbors& neighbors, const int num_disks)
{
    std::vector<int> minimized_ring;

    std::vector<int> node_positions(num_disks, -1);
    std::vector<bool> node_polarities(num_disks);

    for(auto [node, current_polarity] : ring)
    {
        while(true)
        {
            int neighbor_index = -1;
            bool skipped_part_polarity;
            bool shortcut_polarity;
            for(auto [neighbor, edge_polarity] : neighbors[node])
            {
                if(!minimized_ring.empty() && neighbor != minimized_ring.back())
                {
                    if(node_positions[neighbor] > neighbor_index)
                    {
                        neighbor_index = node_positions[neighbor];
                        shortcut_polarity = edge_polarity;
                        skipped_part_polarity = current_polarity != node_polarities[neighbor];
                    }
                }
            }
            if(neighbor_index < 0)
                break;
            if(shortcut_polarity == skipped_part_polarity)
            {
                while(minimized_ring.size() > neighbor_index+1)
                {
                    node_positions[minimized_ring.back()] = -1;
                    minimized_ring.pop_back();
                }
            }
            else
            {
                minimized_ring.emplace_back(node);
                return {minimized_ring.begin() + neighbor_index, minimized_ring.end()};
            }
        }
        minimized_ring.emplace_back(node);
        node_positions[node] = (int)minimized_ring.size() - 1;
        node_polarities[node] = current_polarity;
    }
    assert(false); //unreachable
}
```

The ring minimization works by constructing a new minimized ring in the variable 'minimized_ring',

that starts off empty, so that the path in this variable is never not minimal. Iteratively, the nodes of the original ring are added to this ring (lines 44-46). Before a new node is added to the minimized ring, it is checked if one of its neighbors is already in the minimized ring (lines 11-43). This is done via the lookup table 'node_positions', which saves the position where a node occurs in the 'minimized_ring' variable using the disk number as the index. The position is -1 if the disk is not (yet) in the minimized ring. If a neighbor of the new node is already in the ring, the edge between the new node and this neighbor that is already in the ring is considered as a shortcut, as it can be used to skip the part of the minimized ring between this neighbor node and the new node. If the shortcut has the same polarity as the part of the ring that is skipped, this part of the ring is actually removed (lines 32-37). If the shortcut has the opposite polarity however, the shortcut together with the part that was to be removed actually forms a new ring, which has a *polarity* of *true*. Therefore, this ring is returned (lines 38-42). There may actually be the case that two neighbors of a node that is to be added to the ring are already part of the minimized ring. To deal with this case, this shortcut routine is wrapped in a while loop (line 11) in which always the shortcut to the node that comes first when going backwards through the minimized ring is considered. This order is required so that if a ring consisting of the shortcut and the part contained in the minimized ring is returned (line 41), this returned ring is actually minimal. The function will always return at some point through line 41: As the last node is considered to be added to the minimized ring, it is always a neighbor of the first node and the resulting ring must have a *polarity* of *true*, as the input ring has a *polarity* of *true*. Therefore, line 48 is never reached.

For the runtime, it is important to note that the $while(true)$ loop at line 11 is only run a fixed number of times, as there can only be a limited number of neighbors of a node that are part of the minimized ring. Therefore, it does not impact the big-O notation. The outer for loop is run for every node of the initial ring. The inner loop at line 16 loops through every neighbor of every node in the ring. The while loop at line 32 always removes an element from the minimal ring, of which there can only be $|r|$ many, so it can only run $|r|$ many times. This limits the runtime to $O(|r| + |m(r)|)$, where $m(r)$ is the set of edges connected to nodes in r: $m(r) = \{e \in E | e \cap r \neq \emptyset\}$. Technically the initialization of the list 'node_positions' (line 6) takes $O(|D|)$ time, but this vector could be reset in $O(|r|)$ time and thereby utilized across different runs of the minimization algorithm, so the initialization of this variable is ignored here.

B.3 Finding a ring

B.3.1 Disk view

In the theoretical version of this algorithm, the path $c(s,t)$ is walked along until a disk is hit in point p. Then the algorithm goes along the outside of the disks until either $c(s,t)$ is reached again or the initially hit point p is reached again. The implementation works a little differently: It always goes along the outside of the disks until p is reached again. The closed walk r in CG that represents the disks that were walked along is now considered. If it has $polarity(r) = true$, this ring separates s and t and is returned. If its polarity is $false$, the disks of r can simply be walked around until $c(s,t)$ is reached again. In the implementation, this is accomplished by simply 'deleting' the nodes from the graph and restart going around disks that are hit first when going from s to t.

The following subroutine goes around disks, starting at the disk 'start_node', which is entered at an angle 'start_angle'. It returns the nodes that were walked along, together with the polarity of the path from the 'start_node' towards them.

```
RingWithPolarity go_around_disks(const std::vector<Point>& disks, const Neighbors& neighbors,
                                 const std::vector<bool>& disk_removed,
                                 const int start_disk, const double start_angle)
{
    RingWithPolarity ring{{start_disk, false}};
    int current_disk = start_disk;
    double current_angle = start_angle + 0.0001;
    while(true)
    {
        int closest_disk;
        bool closest_disk_polarity;
        double angle_to_closest_disk = 100;
        for(const auto[neighbor, edge_polarity]: neighbors[current_disk])
        {
            if(!disk_removed[neighbor])
            {
                double angle_to_neighbor = limit_angle_0_2pi(
                calc_intersection_angle(disks[current_disk], disks[neighbor]) - current_angle);
                if (angle_to_neighbor < angle_to_closest_disk)
                {
                    closest_disk = neighbor;
                    closest_disk_polarity = edge_polarity;
                    angle_to_closest_disk = angle_to_neighbor;
                }
            }
         }
        if(angle_to_closest_disk == 100)
            break;
        if(current_disk == start_disk)
        {
            if(limit_angle_0_2pi(start_angle - current_angle) <= angle_to_closest_disk)
                break;
        }
        current_angle = calc_intersection_angle2(disks[closest_disk], disks[current_disk]);
        current_disk = closest_disk;
        ring.emplace_back(closest_disk, std::get<1>(ring.back()) != closest_disk_polarity);
    }
    return ring;
}
```

This function is used within the following function, which takes as its input the vector 'disks_on_st'. This vector contains all disks that are on the curve $c(s,t)$, ordered according to where the disks intersect $c(s,t)$, so that the last element is hit first when traveling along $c(s,t)$. Just as in the theoretical description, the disk that is hit first is taken, and then the algorithm goes around the disks until the initial point is reached again. If the returned walk in the graph has a *polarity* of *true*, it is minimized and returned. If its *polarity* is $false$, the disks that were walked along are part of a cluster of disks that is separate from any disks outside of this cluster, so no disk inside this cluster is part of a ring separating s and t. Therefore, the disks of r can simply be removed and the procedure repeated.

```
std::optional<std::list<int>> find_closest_ring(const std::vector<Point>& disks,
                            const Neighbors& neighbors,
                            std::vector<std::tuple<int,double>>& disks_on_st,
                            std::vector<bool>& disk_removed)
{
    while(!disks_on_st.empty())
    {
        auto [closest_disk, start_angle] = disks_on_st.back();
        if(disk_removed[closest_disk])
        {
            disks_on_st.pop_back();
        }
        else
        {
            RingWithPolarity ring = go_around_disks(
                disks, neighbors, disk_removed, closest_disk, start_angle);
            if (std::get<1>(ring.back()))
            {
                return minimize_ring(ring, neighbors);
            }
            else
            {
                for (auto [disk, _polarity]: ring)
                {
                    disk_removed[disk] = true;
                }
            }
        }
    }
    return std::nullopt;
}
```

For the runtime, it is important to know that disks are visited at most twice when walking around them. As a disk that is walked around is either returned when a ring with a *polarity* of *true* is found, or it is removed from the coverage graph, every disk and its neighbors are only considered at most twice, so walking around the disks takes at most $O(|E|)$ time. Sorting the disks that intersect the curve $c(s,t)$ takes $O(|N|\log(|N|)$ time. Therefore, the total runtime is $O(|N|\log(|N|) + |E|)$ or $O(|D|^2)$ when only considering the number of disks.

B.3.2 Globally shortest ring

Searching for the shortest ring with a *polarity* of true is done by constructing the shortest path tree in the coverage graph CG for any root node. The following subroutine searches for the shortest ring given such a root node and an integer 'shortest_length', for which it should only return the shortest ring if it is shorter than it.

```cpp
std::optional<std::list<int>> find_shortest_ring_with_root(
     const Neighbors& neighbors, const int root_node,
     int& shortest_length, const int num_disks)
{
    auto new_nodes = std::make_unique<std::vector<int>>(1,root_node);
    std::vector<PathType> Paths(num_disks, {-1, -1, false});
    Paths[root_node] = {0, -1, false};
    std::array<int, 2> connecting_nodes = {-1, -1};
    int t = 0;
    while(!new_nodes->empty())
    {
        t++;
        if(2*t > shortest_length)
            break;
        std::unique_ptr<std::vector<int>> current_nodes = std::move(new_nodes);
        new_nodes = std::make_unique<std::vector<int>>();
        for(const int current_node : *current_nodes)
        {
            const bool current_polarity = Paths[current_node].polarity;
            for (auto [next_node, crosses_st]: neighbors[current_node])
            {
                const bool next_polarity = current_polarity != crosses_st;
                if (Paths[next_node].length == -1)
                {
                    Paths[next_node] = {t, current_node, next_polarity};
                    new_nodes->push_back(next_node);
                } else if (next_polarity != Paths[next_node].polarity)
                {
                    const int ring_length = t + Paths[next_node].length;
                    if (ring_length < shortest_length)
                    {
                        shortest_length = ring_length;
                        connecting_nodes = {current_node, next_node};
                    }
                }
            }
        }
    }
    if(connecting_nodes[0] == -1)
        return std::nullopt;
    std::list<int> ring;
    for(int node = connecting_nodes[1]; node != -1; node = Paths[node].previousNode)
    {
        ring.emplace_back(node);
    }
    for(int node = connecting_nodes[0]; node != -1; node = Paths[node].previousNode)
    {
        ring.emplace_front(node);
    }
    return std::make_optional(ring);
}
```

The tree is saved in the 'Paths' variable, where for every node its ancestor in the shortest path tree is saved, along with the distance to the root node and the polarity of the path from the root node. Constructing the tree is done in a breadth first manner, where iteratively all neighbors of nodes added to the tree in the last iteration are added to the tree if they are not yet present in it. If the neighbor already has a path in the tree, then the ring created by connecting this edge is considered. If the ring has polarity true and is shorter than 'shortest_length', it is taken as the currently shortest ring. This way the tree is build and the edges not present in the tree are checked at the same time. In each iteration, the current depth of the tree is saved in the variable t. Any ring that will be found has a length of $2 * t$ or $2 * t - 1$. Therefore, if $2 * t > shortest\_length$, no shorter ring will be found anymore and the algorithm breaks out of the construction of the tree. When a shortest ring is found, the two nodes that need to be connected in the tree to form this ring are saved in the array 'connecting_nodes'. At the end of the function, the ring is actually built in the variable 'ring' from these two connecting nodes. The 'Paths' variable is read like a linked list in order to get the two paths and create the ring.

This function takes $O(n+m)$ time, as for every node every neighbor is considered once, so every edge is considered twice.

This algorithm that calculates the shortest ring given a root node is called in the following function for several root nodes. As an optimization, not every node has to be considered as the root node. As every ring with a *polarity* of *true* must contain at least one edge that crosses $c(s,t)$, if a subset of nodes is picked, so that the subset contains at least one endpoint of every edge crossing $c(s,t)$, every possible ring contains at least one node of this subset. In the implementation, this subset simply consists of every endpoint of the edges crossing $c(s,t)$ that is to the left side of $c(s,t)$.

```
std::optional<std::list<int>> find_shortest_ring(
    const Neighbors& neighbors, const std::vector<int>& root_nodes)
{
    std::optional<std::list<int>> shortest_ring = std::nullopt;
    int shortest_ring_length = std::numeric_limits<int>::max();
    for(const int root_node: root_nodes)
    {
        auto ring = find_shortest_ring_with_root(
                neighbors, root_node, shortest_ring_length);
        if (ring.has_value())
        {
            shortest_ring = ring;
        }
    }
    return shortest_ring;
}
```

As the size of the set of the root nodes might be $\frac{|D|}{2}$, the previous algorithm might be called $O(n)$ times, making the total runtime for finding the shortest ring $O(n(n+m))$ or $O(|D|^3)$ when only considering the number of disks.

B.3.3 Minimum cost ring

Searching for the minimum cost ring is done very similar to searching for the shortest ring, just that the vector 'costs' assigns a cost to every node.

```
std::optional<std::list<int>> find_lowest_cost_ring_with_root
        (const Neighbors& neighbors, const std::vector<double>& costs,
         const int root_node, double& lowest_cost)
{
    std::priority_queue<EventType, std::vector<EventType>, EventTypeComparator> priorityQueue;
    priorityQueue.push({
        costs[root_node] / 2,
        root_node
    });
    std::vector<CostPathType> Paths(neighbors.size(), {-1, -1, false});
    Paths[root_node] = {0.0, -1, false};
    std::array<int, 2> connecting_nodes = {-1, -1};
    while(!priorityQueue.empty())
    {
        double cost = priorityQueue.top().cost;
        if(2*cost >= lowest_cost)
            break;
        const int current_node = priorityQueue.top().node;
        const bool current_polarity = Paths[current_node].polarity;
        priorityQueue.pop();
        for(auto [next_node, edge_polarity]: neighbors[current_node])
        {
            const bool next_polarity = current_polarity != edge_polarity;

            if(Paths[next_node].cost == -1)
            {
                Paths[next_node] = {cost, current_node, next_polarity};
                priorityQueue.push({cost + costs[next_node], next_node});
            }
            else if(next_polarity != Paths[next_node].polarity)
            {
                double ring_cost = cost + Paths[next_node].cost + costs[next_node];
                if (ring_cost < lowest_cost)
                {
                    lowest_cost = ring_cost;
                    connecting_nodes = {current_node, next_node};
                }
            }
        }
    }
    if(connecting_nodes[0] == -1)
    {
        return std::nullopt;
    }
    //Construct the ring based on the connecting_nodes as before
    ...
}
```

The main difference to the previous algorithm is that it cannot iteratively consider all neighbors of the nodes previously added to the tree, as each node has an individual cost. Therefore, in every step, only the node that is reached with the lowest cost is added to the tree. All of its neighbors that are not yet connected in the tree are reached through this node and these paths are therefore added to the tree. In order to know which node is reached with the minimum cost next, they are saved in a priority queue.

Every edge is still considered twice, but as the insertion and extraction of nodes into the priority queue takes logarithmic time, the runtime is $O(n \log(n) + m)$

Considering a subset of nodes as the root nodes and returning the best ring works just as in the previous algorithm, resulting in a total runtime of $O(n(n \log(n) + m))$, which is the same runtime as the previous algorithm when only considering the number of disks: $O(|D|^3)$.

C Graphical interface

A graphical interface which visualizes barrier resilience instances has been implemented in python using the Tkinter library. It calls the C++ program that implements the various algorithms for solving barrier resilience. In total, 28 different algorithms / combinations of algorithms exist and can be used. Problem instances can be created by clicking on the position where a disk should appear, generating them randomly, and they can also be read from and written to disk. The user interface is shown in the following screenshot, which shows a randomly generated instance as they were created for the benchmark.

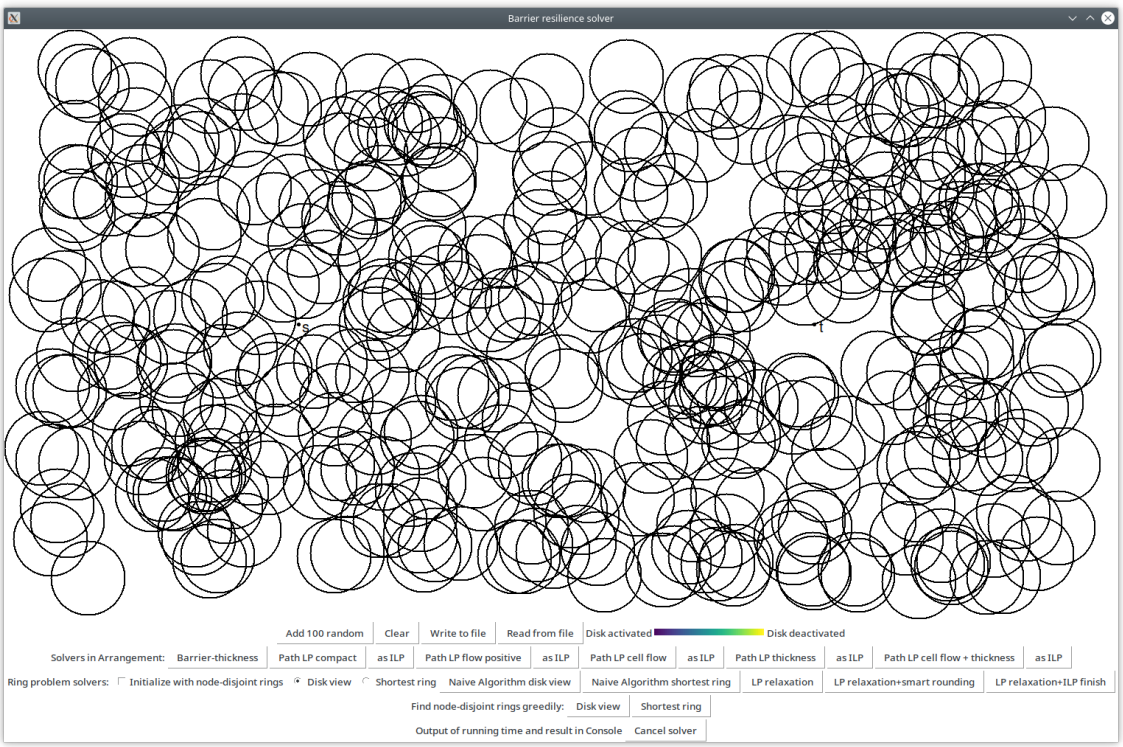


Figure 43: Screenshot of the program displaying a randomly generated instance

After a solving algorithm has been called, the result is visualized. For ring-based algorithms, all the rings that were found are visualized using different colors. If an activation for the disks is calculated, it is represented by dyeing the disk according to a color scale from dark blue for active disks to yellow for deactivated disks. Disks that are kept completely active are grayed out. The result of the LP relaxation of the ring problem is shown in the following screenshot, where the activation of three disks is 0.5 and 4 rings were added to the constraint set in order to find this optimal solution.

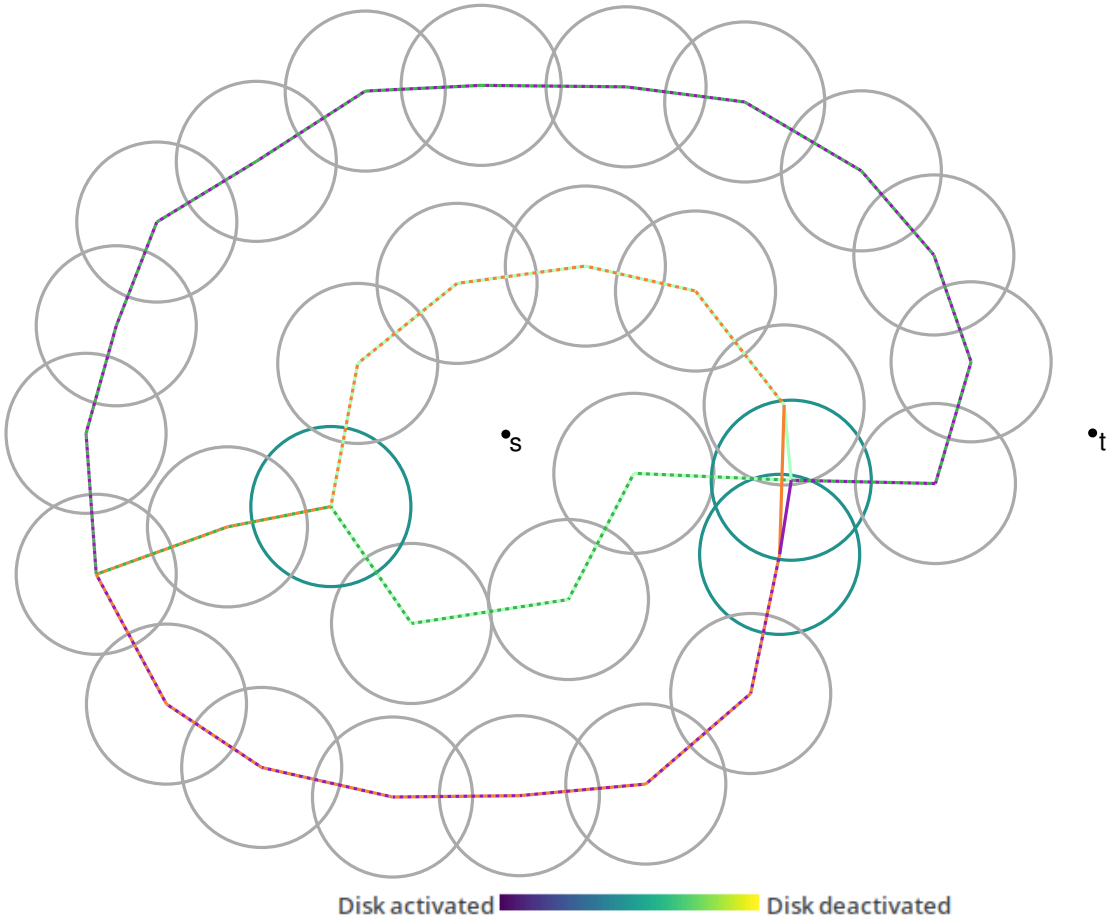


Figure 44: Screenshot of the program after solving the ring-problem as an LP

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Druck-Exemplaren überein.

Datum und Unterschrift:

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted hard copies.

Date and Signature: