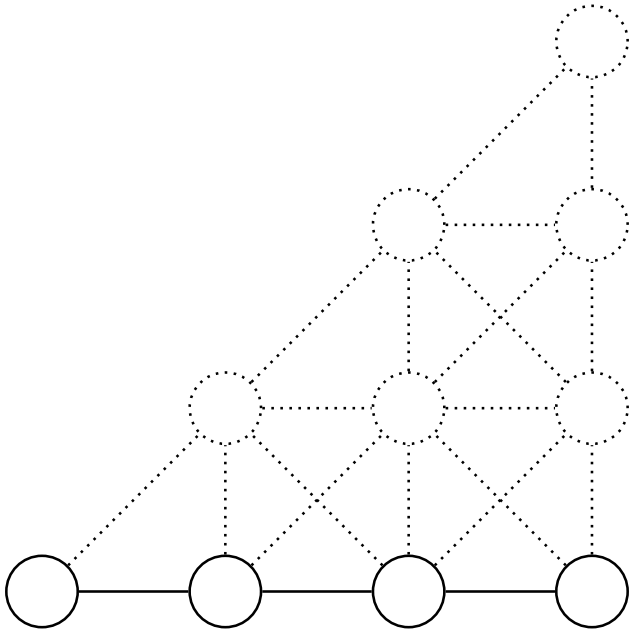


Critical Patrolling Schedules for Two Robots on a Line

Master's Thesis in Computer Science and Engineering

ANTON GUSTAFSSON
IMAN RADJAVI

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Critical Patrolling Schedules for Two Robots on a Line

ANTON GUSTAFSSON

IMAN RADJAVI

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Critical Patrolling Schedules for Two Robots on a Line

ANTON GUSTAFSSON
IMAN RADJAVI

Supervisor: Peter Damaschke, Department of Computer Science and Engineering
Examiner: Nir Piterman, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Four stations on a line, which may also be represented as a triangle graph when patrolling with two robots.

Typeset in LaTeX
Gothenburg, Sweden 2021

Critical Patrolling Schedules for Two Robots on a Line

ANTON GUSTAFSSON
IMAN RADJAVI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Patrolling with unbalanced frequencies (PUF) involves scheduling mobile robots that continuously visit a finite number of fixed stations, with known individual maximal waiting times, placed on a line. Recent work demonstrates that PUF is a remarkably challenging problem. To advance current research in the best-known solution, we searched for *critical instances* of the integer patrolling problem (IntPUF). A solvable instance is critical if it becomes impossible to schedule the visits when any station's waiting time is decremented.

Formulating IntPUF as a specific graph problem enabled several algorithms and heuristics to be developed, mainly for solving any instance of IntPUF and searching for critical instances. The algorithms were proved to be correct and implemented in a Java program to collect the critical instances. Benchmarking the implementation shows that solving instances work well when the number of stations is relatively small while searching for critical instances turned out to be extremely difficult. However, numerous critical instances were found, which reveal interesting patterns that may be further analyzed and utilized to improve the search for additional critical instances.

Keywords: Patrolling with unbalanced frequencies, scheduling, robots, graph, algorithms, heuristics, Java.

Acknowledgements

First and foremost, we want to express our endless gratitude to our supervisor Peter Damaschke. He has contributed with enormous support and guidance, shown great interest in our progress and given us valuable feedback with minimal waiting times. We would also like to thank Nir Piterman for taking the time to be our examiner and for showing an interest in the project.

Anton Gustafsson & Iman Radjavi, Gothenburg, June 2021

Contents

List of Figures

List of Tables

1

Introduction

Patrolling problems involve scheduling mobile robots such that they visit points repeatedly with known minimal frequencies [1]. These problems are interesting for a variety of applications, such as maintenance, monitoring and fetching resources. Previously, various cases and aspects of patrolling problems have been studied. Some examples include patrolling when points have to be visited with the same frequency, robots having different speeds and where all points have equal pairwise distances [2, 3]. More recently, these problems gained new attention because of an observation that different individual frequencies, i.e., when certain points have to be visited more often than others, makes the problems difficult [4].

In this project we deal with basic research on combinatorial and algorithmic aspects of patrolling problems with different individual frequencies. Immediate applications of the problem is out of scope for this project.

1.1 Background

In this section the project's background and relevant work is introduced. In Section 1.1.1 we cover the specific patrolling problem studied in this project. Finally, in Section 1.1.2, a similar problem is presented, which has been studied previously, and compared with the problem studied in this project.

1.1.1 Patrolling with Unbalanced Frequencies

The problem called Patrolling with Unbalanced Frequencies (PUF) considers $m+1$ stations deployed at fixed points s_i, for $i \in \{0, ..., m\}$, placed on the real line L [1]. Every station i allows a maximal waiting time t_i and the problem is to construct a patrolling schedule, such that the time between two consecutive visits of a station never exceeds its maximal waiting time. Figure 1.1 illustrates an instance of the problem with four stations.

Patrolling with different time constraints on every station seems natural and observable in practice [4]. For instance, [4] has observed this in continuous testing of virtual machines in cloud systems. In such systems, the frequency of testing vir-

Figure 1.1: Example of a PUF instance with four stations at fixed points s_i, for $i \in \{0, 1, 2, 3\}$, on the line. Each station i has a maximal waiting time t_i.

tual machines for undesirable symptoms may vary depending on the importance of dedicated cloud operational mechanisms. Another example is that network administrators can apply patrolling to detect network failures or to discover web pages that need to be indexed by search engines [2]. Networks and web pages may also have different importance and must therefore be patrolled with different frequencies.

The problem of PUF was recently studied; for the case of one robot patrolling on the line, PUF is easily solvable in linear time [4]. However, the study shows that PUF is surprisingly hard to solve when increasing the number of robots. Already for two robots, the complexity status is open. To solve the problem with two robots, several approximation algorithms are provided in [4]. Until recently, one of them was the best known solution - a $\sqrt{3}$-approximation algorithm. In this context, a c-approximation means that for every station i and $c \geq 1$, the time between two consecutive visits never exceed ct_i, where t_i is the maximal waiting time of station i. Moreover, this means that PUF can be defined as an optimization problem with the goal of finding a schedule with minimum c.

Damaschke [1] presents a state-of-the-art solution to PUF with two robots. The solution is a polynomial-time approximation scheme (PTAS). A PTAS is a type of approximation algorithm that, given an optimization problem and a parameter $\epsilon > 0$, produces, in polynomial time, a solution that is within a factor $1 + \epsilon$ of being optimal. This means that the PTAS enables a $(1 + \epsilon)$-approximation algorithm to solve PUF with two robots. The solution is further discussed in section 1.3.

1.1.2 Boundary Patrolling / Fence Patrolling

Czyzowicz et al. [2] has studied a problem similar to PUF. With a slightly reformulated description, the problem known as boundary patrolling or fence patrolling considers k mobile robots, each with a predefined maximal speed. The robots are supposed to endlessly protect points in a given environment from an adversary which attempts to intrude a point. The adversary needs a given time interval of length τ to accomplish the intrusion. This means that, to avoid a point from being intruded, no point can remain unprotected by a robot for a time period greater than or equal to τ.

The problem is to schedule the robots such that an intrusion is prevented, which may be solved by designing an algorithm that minimizes the idle time I, i.e., the longest time interval that a point remains unprotected by some robot. This means that there is an obvious goal of achieving $I < \tau$. Czyzowicz et al. [2] presents two

algorithms that are optimal in some cases.

The key difference, compared to PUF, is that no station remains unprotected for longer than a given time period τ. This means that the stations in this problem have equal waiting times, as opposed to PUF where stations have individual time constraints. As shown in [4], finding schedules to satisfy the requirements for PUF, or even deciding on their existence for two robots, turns out to be a highly intricate problem.

1.2 Notations

The following notations and their definitions are used consistently throughout the report, unless stated otherwise:

- The variable m refers to the last index of points indexed $0, 1, ..., m$.
- The variable i is defined as an index ranging over $\{0, 1, ..., m\}$.
- The variable r refers to the roof value for `Search`, see Section 3.3.
- The variables a and b refer to the intersection of ranges $[a, b] = \bigcap_i [a_i, b_i]$, see Section 3.2.1.
- $s \to ... \to t$ denotes a path from a position $s = (s_x, s_y)$ to a position $t = (t_x, t_y)$, that may include visiting other intermediate positions. We may also denote the path as a sequence of positions: $(s_x, s_y)...(t_x, t_y)$.
- For two sets S and T, $S \to ... \to T$ denotes any $s \to ... \to t$ path where $s \in S$ and $t \in T$.
- `Solve` refers to the approach for solving an instance of the patrolling problem, first described in section 3.2.
- `Search` refers to the approach for searching for critical instances (see section 1.3.2), first described in section 3.3.

1.3 Problem Statement

Damaschke [1] proposes an approach for solving PUF with two patrolling robots on a line. By using insights from the integer version of PUF (section 1.3.1) a PTAS can be constructed. However, the PTAS is not yet practical. In the following sections we present the integer version of PUF and important insights that contribute to the practicality of the PTAS.

1.3.1 The Integer Version of PUF

The integer version of PUF, called *IntPUF*, is a variant of PUF [1]. It is defined precisely as PUF (see section 1.1) but with the following additional demands:

- All maximal waiting times t_i are integer and $t_i > 0$.
- All s_i, i.e., the position of stations, are integers.
- Every robot either stays at some point with integer coordinate or moves at unit speed.
- Every robot can change its speed or its moving direction only at integer times and at integer points.

1.3.2 Critical Instances

It is important to note that the PTAS in [1] is not yet practical. To achieve practicality, we must efficiently solve IntPUF instances up to some number of points. An important insight is that we can efficiently solve IntPUF instances once we know all *critical instances* of IntPUF for a fixed m [1, Lemma 2]. Critical instances are instances that are solvable but become infeasible when some waiting time is further reduced. For instance, using one robot, figure 1.1 illustrates the only critical instance of IntPUF with four stations [1, Theorem 2]. The optimal schedule for the robot is to "zigzag", i.e., to perpetually move from the left-most to the right-most station and back. Any instance, where all waiting times are greater than or equal to the waiting times of a critical instance, must also be solvable. This insight allows for an efficient method of solving an instance of IntPUF, see section 1.3.3.

1.3.3 Usage of Critical Instances

Suppose we know all critical instances of IntPUF and their solutions up to a fixed m. We can then solve an instance with $n+1$ stations and waiting times $t_0, ..., t_n$ using one of the following methods:

1. If $n \leq m$: For all critical instances with $n+1$ stations and for $i = 0, ..., n$, compare each waiting time t_i with the waiting time u_i of the critical instance. If there exists a critical instance, where $t_i \geq u_i$ for all i, then the instance is solvable, otherwise not.

2. If $n > m$: The instance must first be approximated and then solved using method 1. The approximation error will be smaller the larger m is.

1.4 Contribution

As discussed in section 1.3.2 and 1.3.3, one may use critical instances to efficiently solve an instance of IntPUF. However, this relies on the knowledge of all critical instances for a given m. Identifying critical instances appears to be a highly non-trivial task. Manual work suggests a combinatorial explosion as m increases. More specifically, for $m > 6$. In this project we aim at identifying and collecting the critical instances of IntPUF with two patrolling robots, together with the solutions to them, up to a fixed m, as large as possible. This will be achieved by developing search heuristics and implementing them in a computer program with the help of ideas from [1, 5]. By collecting the critical instances with our computer program, we will enable further progress with the best known solution of PUF, i.e., the PTAS from [1]. The critical instances are a necessity for solving IntPUF instances efficiently, and thus making the PTAS practical. Furthermore, our hope is that the critical instances may also improve the understanding of the general structure of patrolling schedules. Some surprising patterns may emerge here.

1.5 Research Questions

In this project, we seek to answer the following questions:

1. Can we collect all critical instances and their solutions for $m = 7$, $m = 8$, etc (m as large as possible)?

2. Can we display solutions to critical instances in a comprehensible way for further research?

3. How does the search time scale with m?

4. For every given m, can we search for all critical instances without limiting the search using an upper bound on the waiting times of an instance, i.e., a roof value?

1.6 Limitations

This project deals with basic research on combinatorial and algorithmic aspects of PUF. For this reason, immediate applications of the problem is out of scope for this project. Furthermore, The project will not consider the usage of critical instances to solve arbitrary instances. The reason is twofold: method 1 in section 1.3.3 is a trivial comparison of waiting times and method 2 has already been treated by Damaschke [1].

Since identifying critical instances manually suggests a combinatorial explosion as m increases, we expect success only for rather small m during the limited project time. Currently, it is unknown how many critical instances exist for a given m.

2

The Patrolling Problem

This chapter introduces a generic formulation of the patrolling problem with individual waiting times. This formulation will serve as an important foundation for the rest of this report, since our contributions are heavily based on it. Everything that is introduced in this chapter has been contributed by Damaschke [5].

2.1 The Position Graph

An instance of the patrolling problem consists of a *position graph*. The position graph is an undirected finite graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. The vertices of V are called *positions*. Furthermore, the instance also consists of *properties*, which are $m+1$ subsets $P_i \subset V$. Every property P_i has a *waiting time* t_i. If a position $v \in P_i$, we may say that the position *has the property* P_i.

A *path* in the position graph is defined as a sequence

$$v_1 \to ... \to v_k$$

of positions $v_j \in V$, such that there exists an edge between v_j and v_{j+1} for

$$j \in \{1, ..., k-1\}.$$

A cycle C in the position graph G is defined precisely as a path but with an additional criteria; a cycle has an edge between v_k and v_1 in order to form the cycle. Note that positions may appear several times in a cycle. Moreover, the starting position is immaterial, i.e., any cyclic shift

$$v_{j+1} \to ... \to v_k \to v_1 \to ... \to v_j$$

is considered to be the same cycle.

Given an instance as described above, the patrolling problem is to find a *solution cycle* C in the position graph that satisfies the following for every property P_i: When we walk C, we always encounter the next position with property P_i after traversing at most t_i edges. Intuitively, one can imagine a robot patrolling the position graph.

However, note that this formulation is also suitable for the case of $r > 1$ robots in a position graph H that must perpetually visit every property P_i with waiting time at most t_i. This multi-robot version can simply be reduced to the generic formulation presented above: We define a position graph G whose positions are the r-tuples of positions of H, indicating all robots' positions. There is an edge between two positions $(u_1, ..., u_r) \neq (v_1, ..., v_r)$ in G if and only if, for all j, positions u_j and v_j are identical or there is an edge between them in H. A position in G has exactly all properties of its r entries. Now, one robot in G represents r robots in H.

2.1.1 Distances

Let $d(x, y)$ denote the distance from some position x to some position y in the position graph. The distance is the length of a shortest $x \to ... \to y$ path. Note that $d(x, y) = d(y, x)$ since the position graph is undirected. For two subsets $X, Y \subset V$, the distance of X and Y is naturally defined as the length of a shortest $X \to ... \to Y$ path:

$$d(X, Y) = min\{d(x, y) \mid x \in X, y \in Y\}.$$

We may also write $d(x, Y)$ for $d(\{x\}, Y)$.

2.2 Vectors of Waiting Times

Given a position graph with a set of $m + 1$ properties we may represent the waiting times as an integer vector

$$(t_0, ..., t_m).$$

An instance of the patrolling problem may simply be identified using this integer vector of waiting times. For any two vectors V and W of the same length we write $V \leq W$ if V is component-wise smaller than or equal to W. We write $V < W$ if $V \leq W$ and at least one component of V is strictly smaller than its counterpart in W. We then say that V is smaller than W. For example, $(1, 1, ..., 1)$ is smaller than $(2, 1, 1, ..., 1)$. Note that some instances are incomparable, e.g., $(2, 1, 1, .., 1)$ and $(1, 2, 1, 1, .., 1)$.

We call an instance I with the integer vector T of waiting times *feasible* if the patrolling problem has a solution cycle with waiting time at most t_i for every property P_i. If a vector T is feasible and no instance with vector $T' < T$ is feasible, we call T and I critical.

2.3 The Valid Position Graph

Let R denote any set of vertices of which we know that vertices outside R cannot appear in a solution cycle. We may then restrict the patrolling problem to the *valid position graph* $G[R]$, which is an induced subgraph of G [6]. $G[R]$ is the subgraph whose vertex set is R and whose edge set is the set of those edges in G that connect two vertices in R.

We define the *range* of a property P_i to be the set of nodes

$$R_i := \{v \in V \mid \exists u \in P_i \ : \ d(u, v) \leq \lfloor t_i/2 \rfloor\}.$$

Informally, R_i is the set of nodes such that the distance from a node u having property P_i to any node v is at most $\lfloor t_i/2 \rfloor$. Another way to say it is that the range R_i of a property P_i will be the subset of vertices in the position graph that the robots can visit in order to fulfill the waiting time of property P_i. Visiting a vertex outside of R_i would violate the waiting time of P_i. An important observation is that any solution cycle C cannot contain vertices that are not in R_i. The reason is that sometimes C visits P_i, and it will visit P_j before and after the visit of P_i. Any subpath $P_j \to ... \to P_i \to ... \to P_j$ of C, where P_i does not appear elsewhere in the path, must be of length at most t_i, otherwise C is not a solution cycle by definition. Thus, if $d(P_j, P_i) = d(P_i, P_j) > \lfloor t_i/2 \rfloor$, the total length of the subpath would be greater than t_i. Since this holds for every property, it is safe to initially define

$$R := \bigcap_i R_i.$$

Similarly, we define R' to be the set of edges of which we know that only these edges can be traversed by a solution cycle. It is safe to initially define R' to be the edge set of the valid position graph $G[R]$.

Lemma 2.1. *Consider an integer $t > 0$. For every property P_i and P_j where $i \neq j$, every solution cycle C such that $t_i \leq 2t + 1$ must contain either a $P_i \to ... \to P_j$ path of length at most t, or a $P_j \to ... \to P_i$ path of length at most t. Furthermore, C cannot visit any vertex v with $d(v, P_i) \geq t + 1$.*

Proof. When traversing C, both P_i and P_j must be visited at some point, and P_i will be visited before and after the visit of P_j. The total length of the subpath $P_i \to ... \to P_j \to ... \to P_i$, where P_i does not appear elsewhere in the path, is at most $2t + 1$, otherwise t_i is violated. Hence, one of the paths, $P_i \to ... \to P_j$ or $P_j \to ... \to P_i$, has a length at most t. The last assertion is shown similarly: when traversing C, P_i is visited before and after some assumed visit of v. Since $d(v, P_i) \geq t + 1$, the length of the path from $P_i \to ... \to v \to ... \to P_i$, where P_i does not appear elsewhere in the path, would be at least $2t + 2$, hence, violating $t_i \leq 2t + 1$. □

Some immediate consequences of Lemma 2.1 for any properties P_i and P_j are: $t_i \geq 2d(P_i, P_j)$. Hence, $t_i \geq 2\text{max}_j d(P_i, P_j)$ and any solution cycle for $t_i \leq 2d(P_i, P_j) + 1$

must traverse some shortest $P_i \rightarrow ... \rightarrow P_j$ path. Rephrasing the first consequence: every feasible vector T satisfies $T \geq D$, where D has the components $t_i := 2\text{max}_j d(P_i, P_j)$ for $i = 0, ..., m$. In the case that D itself is feasible, it follows that D is the only critical vector.

2.4 Paths in the Valid Position Graph

Consider a path p in the valid position graph. Let s and f denote the first and last vertex, respectively, of p. Let s_i and f_i denote the first and last vertex, respectively, of p that has property P_i. We define the path p to be *valid* if, for every P_i, it satisfies one of the two following conditions:

- If no vertex of p has property P_i, then the sum of the following three lengths must be at most t_i:
  - The length of the shortest $P_i \rightarrow ... \rightarrow s$ path.
  - The length of path p.
  - The length of the shortest $f \rightarrow ... \rightarrow P_i$ path.
- If some vertices of p have property P_i, then each of the following lengths must be at most t_i:
  - The length of the shortest $P_i \rightarrow ... \rightarrow s$ path + the length of the subpath $s \rightarrow ... \rightarrow s_i$ of p.
  - The length of every path between two consecutive P_i-vertices on p.
  - The length of the subpath $f_i \rightarrow ... \rightarrow f$ of p + the length of the shortest $f \rightarrow ... \rightarrow P_i$ path.

3

Basic Approach

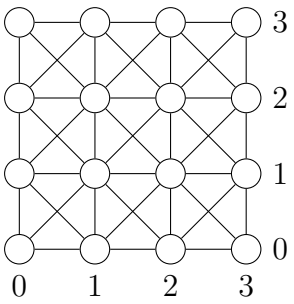
This chapter presents a basic approach for collecting all critical instances of IntPUF for a given m. Section 3.1 introduces how to represent an instance of IntPUF as an instance of the patrolling problem. This is necessary in order to understand the following sections. Section 3.2 presents an approach for solving an instance of the problem. Finally, Section 3.3 presents a search algorithm for finding all critical instances of IntPUF for a given m by solving instances systematically.

3.1 Representation of IntPUF Instances

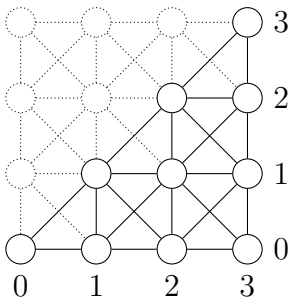
This section adopts contributions by Damaschke [5]. The problem IntPUF, as defined in Section 1.3.1, can be reduced to an instance of the patrolling problem (Chapter 2). We consider the problem with two robots r_x and r_y. The vertex set of the position graph may then consist of all points with integer coordinates (x, y), where $0 \leq x \leq m$ and $0 \leq y \leq m$. The vertices can be thought of as points in a Cartesian coordinate system, where the x-axis represents the position of robot r_x and the y-axis the position of robot r_y. Since we consider the position (x, y) to be equivalent to position (y, x) we restrict the vertex set to consist only of positions where $x \geq y$. This forms a triangle graph instead of a square grid graph, see figure 3.1. An edge is created from a position (x_1, y_1) to a position (x_2, y_2) if robot r_x can reach x_2 from x_1 in one time unit and robot r_y can reach y_2 from y_1 in one time unit. It follows that a position (a, b) can have up to eight neighbours. These are exactly the positions (x, y) such that $a - 1 \leq x \leq a + 1$, $b - 1 \leq y \leq b + 1$ and $(x, y) \neq (a, b)$.

Each property P_i is a subset of positions that has property P_i. A position (x, y) has the property P_i if $x = i$ or $y = i$, i.e., if some robot visits station i. Figure 3.2 shows all properties for $m = 3$. We define $a_i := i - \lfloor t_i/2 \rfloor$ and $b_i := i + \lfloor t_i/2 \rfloor$. Remembering the definition of the range R_i from section 2.3 we may also define R_i of property P_i to be the set of all vertices in the triangle graph satisfying $a_i \leq x \leq b_i \vee a_i \leq y \leq b_i$.

A solution cycle C in the position graph corresponds to a schedule for the two robots patrolling the line. Thus, if a solution cycle for a given instance of the patrolling problem is found, we have also found a solution for the IntPUF problem.

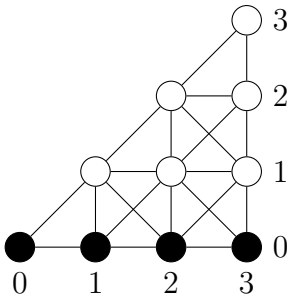


(a) Square grid graph

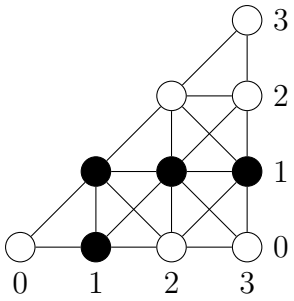


(b) Triangle graph

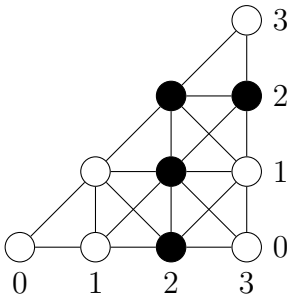
Figure 3.1: Two position graphs for $m = 3$. The x-axis represents the position of robot r_x and the y-axis the position of robot r_y.



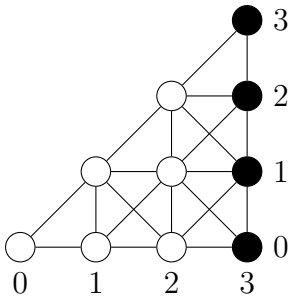
(a) Property P_0.



(b) Property P_1.



(c) Property P_2.



(d) Property P_3.

Figure 3.2: Properties of the position graph for $m = 3$. The filled vertices indicate the positions that has a property.

3.2 Solving an Instance of the Patrolling Problem

Instances of the patrolling problem can be solved by constructing solution cycles in the position graph, or showing infeasibility. This section present `Solve`, a basic approach for constructing solution cycles to any instance.

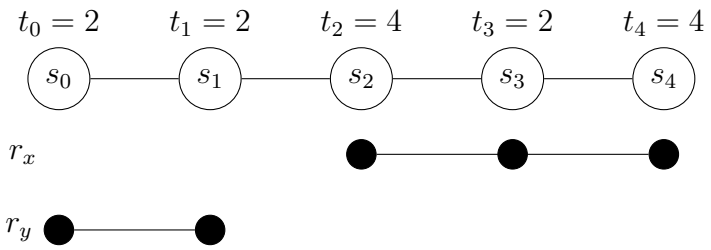
3.2.1 Scenarios

This section is based on contributions by Damaschke [5]. For two robots patrolling on the line, we consider two scenarios for the solution cycles:

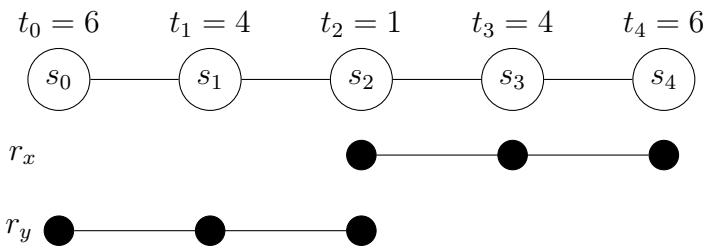
Scenario 1: The empty intersection $\bigcap_i [a_i, b_i] = \emptyset$.

Scenario 2: The non-empty intersection $\bigcap_i [a_i, b_i] \neq \emptyset$.

Intuitively, Scenario 1 means that the two robots need to visit separate properties independently of each other. Therefore, this scenario is equivalent to solving the patrolling problem with one robot on two separate parts of the line. Scenario 2 means that both robots can visit some properties in order to fulfill the waiting times. Figure 3.3a illustrates an example of scenario 1 where the filled dots highlight the stations that each robot is scheduled to visit. The two robots r_x and r_y must independently patrol separate parts of the line to fulfill the given waiting times. Figure 3.3b illustrates an example of Scenario 2. The two robots have to share station s_2 to fulfill the given waiting times.



(a) Scenario 1: $\bigcap_i [a_i, b_i] = \emptyset$



(b) Scenario 2: $\bigcap_i [a_i, b_i] = \{2\}$

Figure 3.3: Instances that illustrate the two scenarios. The filled dots highlight the stations that each robot is scheduled to visit. The corresponding solution cycles are available in Appendix A.2.

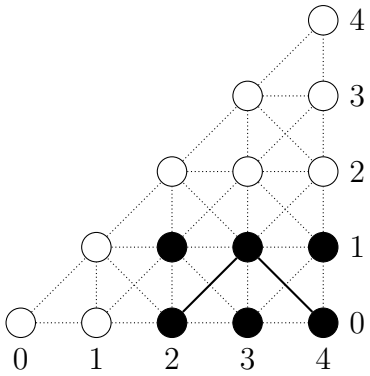


Figure 3.4: The solid line illustrates the BB path in $[2,4] \times [0,1]$ (filled vertices) for the critical instance $(2,2,4,2,4)$ with demarcation point $d = 1$.

The simple case of Scenario 1 has already been solved in [1, 4]: The two robots zigzag independently in the intervals $[0,d]$ and $[d+1,m]$ for some demarcation point d. For example, in Figure 3.3a the two robots zigzag independently using the demarcation point $d = 1$. An equivalent notion for this type of solution is defined as a *Billiard Ball Path*, or *BB Path* for short, which always follows a straight line with slope $+1$ or -1 until it reaches a side or corner of R where it is reflected. Figure 3.4 illustrates an example of a BB Path.

All critical instances of Scenario 1 is given by Proposition 3.1, which is a reformulation of results from [1] and [4].

Proposition 3.1. *For $m \geq 4$, all critical vectors with $\bigcap_i [a_i, b_i] = \emptyset$ are given by $t_i = \max\{1, 2 \cdot \max\{i, d-i\}\}$ for all $i \leq d$, and $t_j = \max\{1, 2 \cdot \max\{j-d-1, m-j\}\}$ for all $j \geq d+1$, where d is any fixed integer with $0 \leq d \leq m-1$. Moreover, the corresponding solution cycles are exactly the BB Paths in $[d+1, m] \times [0, d]$.*

For Scenario 2 we have $\bigcap_i [a_i, b_i] \neq \emptyset$, which we now denote as $[a, b]$ unless stated otherwise. Let $a' := \max_i a_i$ and $b' := \min_i b_i$. Then, $a = \max\{a', 0\}$ and $b = \min\{b', m\}$. For example, in Figure 3.3b we have $[a, b] = [2, 2]$.

3.2.2 Constructing Solution Cycles

Now that the preliminaries have been described, we can introduce our contributions. We have developed an algorithm for constructing solution cycles for any given instance vector. The algorithm distinguishes between the two scenarios in 3.2.1 by using the values of a and b.

An instance v that corresponds to Scenario 1 in Section 3.2.1 has $a > b$. Such instances can be solved using the critical instances from Proposition 3.1. Let C be the set of critical instances from Proposition 3.1. We only need to look for the critical instance $c \in C$ such that $c \leq v$ and return the BB path of c. However, it may be that $v < c$ for every $c \in C$. In that case, v is infeasible.

For Scenario 2 we proceed as follows: intuitively, the algorithm begins with an initial set of valid paths of length 1. This set may simply be all edges in the valid position graph $G[R]$, see Section 2.1.1. These edges are equivalent to all possible paths of length 1 in $G[R]$. Next, the algorithm will extend these initial paths to create new paths of length 2. Extending a path means to add a neighbour of the last position at the end of the path. These new paths must be checked to be valid, see section 2.4. From now on, we refer to this process as *validating* a path. If a path of length 2 is valid, it will be extended to paths of length 3. Once all paths of length 2 have been validated and extended, the process is repeated with paths of length 3 and so on, up to a length x. The length x depends on when a solution cycle is found for a feasible instance or when all paths become invalid for an infeasible instance. Algorithm 1 describes how to construct solution cycles in more detail.

Algorithm 1 Algorithm for solving an instance vector $v = (t_0, ..., t_m)$

```
1:  P := set of valid paths to extend

2:  function SOLVEINSTANCE(v):
3:      if a > b then
4:          return BB path or infeasible
5:      initialize P with valid paths of length 1
6:      while P not empty do
7:          p ← remove a path from P with shortest length
8:          for each vertex q adjacent to last vertex in path p do
9:              if path p → q is valid then
10:                 if path p → q is a solution cycle then
11:                     return path p → q
12:                 else
13:                     append path p → q to P
14:     return v infeasible
```

Proposition 3.2. *Algorithm 1 will return a solution cycle and terminate if an instance $v = (t_0, ..., t_m)$ is feasible. If v is infeasible, the algorithm will eventually terminate.*

Proof. The algorithm terminates if and only if a solution cycle is found or P is empty. Let us consider the case when v is feasible. If an infinitely long valid path p exists in the valid position graph of v, we can divide p in epochs [5]. An epoch is a subpath of p with length $t := \max_i t_i$. Note that an epoch visits all properties since it is a valid path and has the length t, see Section 2.4. The valid position graph $G[R]$ has $|R|$ vertices so there are at most $|R|^t$ unique epochs since each of the t places in an epoch can have one out of $|R|$ vertices. This means that among $|R|^t + 1$ consecutive epochs in the infinite path p, there must appear at least two identical epochs. Hence, p has a subpath c of length at most $t|R|^t$, and the epoch that follows c on p must be identical to some epoch of c, i.e., p has the subpath $c \to c_1$, where c_1 is an epoch of c. Now the last position of c can be connected to the first position

of c, in order to form a cycle. Walking this cycle is a valid solution: let c' be the subpath of c after c_1. We know that $c \to c_1$ is a valid path, which is equivalent to $c_1 \to c' \to c_1$. Hence, we know that c_1 can be followed by c' and therefore the path $c_1 \to c' \to c_1 \to c'$ is also a valid path, which is equivalent to walking the cycle c. Conversely, any cycle can be winded up to an infinite valid path. This shows that if v is feasible, there exists a solution cycle with length at most $t|R|^t$. Since the algorithm validates paths of length x before paths of length $x+1$, it will identify and return a solution cycle c of length at most $t|R|^t$ before extending and generating a valid path d of length $t|R|^t + 1$. Thus, the algorithm will identify and return a solution cycle c if v is feasible. If v is infeasible, it will not have an infinitely long valid path p. Therefore, the algorithm will eventually invalidate all paths of some finite length $x \leq t|R|^t$ where none of them can be further extended to a valid path of length $x+1$. Thus, P will eventually become empty and the algorithm terminates. □

3.3 Searching for Critical Instances

This section presents `Search`, which is an approach for finding all critical instances for a given m. The approach is based on solving instances using Algorithm 1. If an instance is infeasible, we know that at least one of the waiting times need to be incremented in order to become feasible. Furthermore, we know that a critical instance is defined to be an instance such that decreasing any of the waiting times makes the instance infeasible. By using these facts, we may search for critical instances by searching "upwards": begin with an infeasible lower bound instance and work upwards by incrementing each of the waiting times by 1, creating $m+1$ new instances. A trivial infeasible lower bound instance for $m > 1$ is the vector $(1, ..., 1)$. The upwards search is done one *level* at a time. An instance is at level x if it has been incremented x times from the lower bound instance $(1, 1, ..., 1)$. The level x of an instance is exactly $\sum_i (t_i - 1)$. Figure 3.5 shows an example of the search tree up to level 2 for $m = 3$. To prevent incrementing to infinity, the waiting times are not incremented beyond a *roof* value r. It follows that $(r, ..., r)$ is the last instance that can be generated in the search algorithm.

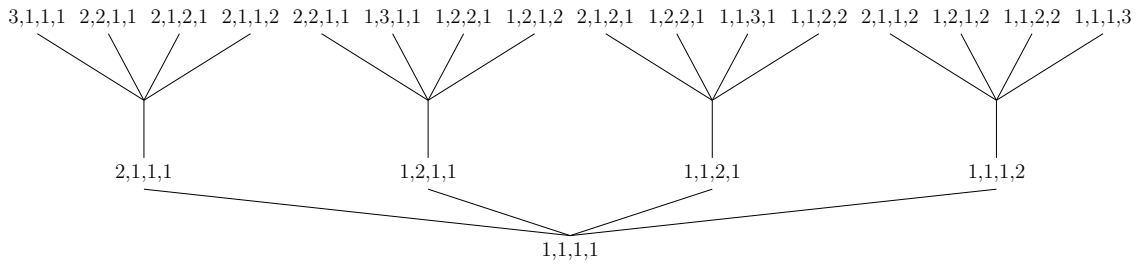


Figure 3.5: Generated instances up to level 2 of the search algorithm for $m = 3$.

The search algorithm continuously keeps track of two sets C and U. The set C contains all currently known critical instances. The set U contains the current maximal instances that are known to be infeasible. Initially, C contains all the

critical instances from Proposition 3.1. These correspond to critical instances of Scenario 1. U is initialized to contain the lower bound instance $(1, 1, ..., 1)$. After initializing the two sets, we need to search for all critical instances of Scenario 2. Algorithm 2 is a full description of our search algorithm.

Algorithm 2 Search algorithm for finding all critical instances for a given m

```
1: U ← Set of infeasible instances
2: C ← Set of known critical instances

3: function searchForCriticalInstances(m):
4:     add critical instances from Proposition 3.1 to C
5:     add lower bound instance [1, ..., 1] to U
6:     while U not empty do
7:         u ← remove an instance from U with lowest level
8:         V ← empty set of incremented instances
9:         for i = 0, ..., m do
10:            v ← copy of u
11:            if v_i ≥ r then
12:                continue
13:            v_i ← v_i + 1
14:            if ∄c ∈ C(c ≤ v) then
15:                add v to V
16:        for each instance v in V do
17:            if v feasible then
18:                add v to C
19:            else
20:                add v to U
21:    return C
```

Proposition 3.3. *When algorithm 2 terminates, the set C will only contain all critical instances for m bounded by r.*

Proof. The only time an instance is inserted in C is when some $v \in V$ is feasible. Let f be a feasible, but not critical, instance. Assume $f \in V$. Since f is not critical, there exists a critical instance $e < f$. It follows that $e \notin C$ since $f \in V$ and V has the property that for any instance $v \in V$, there does not exist a critical instance $c \in C$ such that $c \leq v$. The search algorithm starts with an infeasible lower bound vector and works upward one level at a time. Since $e < f$, e must have a lower level than f. Thus, e must have been visited before f and inserted in C, i.e., $e \in C$ which is a contradiction. Hence, $f \notin V$. This shows that feasible instances in V must be critical and thus only critical instances are inserted in C. Furthermore, every possible instance $w \leq (r, ..., r)$ is generated and inserted in V unless there exists a $c \in C$ such that $c \leq w$. Also, from the definition of the roof value r, we can conclude that there can not exist a critical instance $s > (r, ..., r)$. Hence, all critical instances will be present in C when the algorithm terminates. □

Note that, in order to find all critical instances for a given m, the roof value r needs to be an upper bound on the waiting times such that no critical instance has a waiting time greater than r. Unfortunately, a theoretical upper bound is currently unknown.

4

Improving `Solve`

This chapter introduces improvements made to the basic approach of constructing solution cycles, see Section 3.2.

4.1 Reducing the Number of Paths

When solving an instance using Algorithm 1, every iteration will check if a path p of length $|p|$ is valid. If p is valid, up to eight new paths of length $|p| + 1$ may be generated, one for each neighbour of the last position in p, see figure 4.1. The

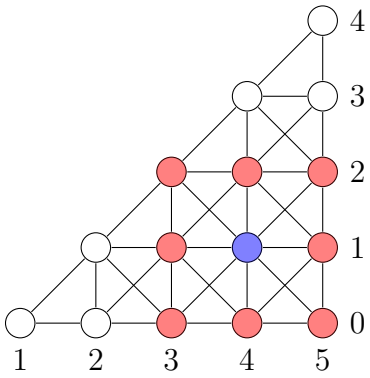


Figure 4.1: Position $(4, 1)$ (blue) in the position graph with all its eight neighbours (red).

maximum number of paths of length x is $n^{x-1}v$ when the initial set of valid paths contain v paths and a path can generate up to n new paths. Algorithm 1 will first generate paths of length 1, then of length 2, and so on, up to a length t. The length t depends on when the algorithm terminates, i.e., when it finds a solution cycle or can not produce more valid paths. The number of paths that Algorithm 1 will generate in the worst case is shown in Equation 4.1.

$$\sum_{x=1}^{t} 8^{x-1}v = \frac{v}{8}\sum_{x=1}^{t} 8^x = \frac{v}{8}\frac{8}{7}(8^t - 1) = \frac{v}{7}(8^t - 1) = \mathcal{O}(v8^t) \tag{4.1}$$

In Section 3.2.2, the initial set of valid paths for algorithm 1 was set to contain all edges of the valid position graph when solving an instance.

Lemma 4.1. *The valid position graph of an instance has $\mathcal{O}(m^2)$ edges.*

The proof of Lemma 4.1 has been omitted since it is trivial. Using Lemma 4.1 and Equation 4.1, we can conclude that, in the worst case, the number of generated paths with our basic approach is $\mathcal{O}(m^2 8^t)$.

4.1.1 The Loneliest Starting Property

Every property must be visited at least once in a solution cycle and the starting position of a solution cycle is immaterial, see Section 2.1. Thus, instead of having all edges of the valid position graph in our initial set of valid paths, we may restrict the set significantly in order to decrease v in equation 4.1. We only need to consider one *starting property* P_i, and the paths of length 1 in the valid position graph such that the starting position of the path has P_i. Figure 4.2 illustrates all possible paths of length 1 when P_0 is the starting property.

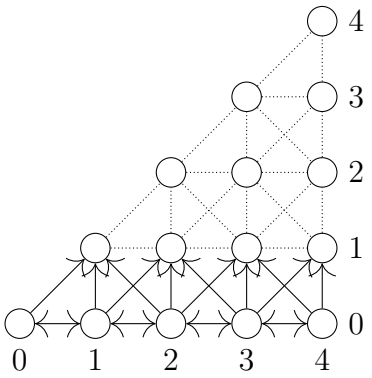


Figure 4.2: All possible paths of length 1 (solid edges) when P_0 is the starting property.

The choice of a starting property also affects the initial number of valid paths. The reason is that, for each property P_i, there are a different number of paths that start in property P_i. We want to choose the property with fewest such paths in order to minimize the initial set of valid paths. We call such a property *the loneliest starting property.*

Lemma 4.2. *The loneliest starting property is always P_0 or P_m. Furthermore, the number of paths of length* 1 *that start in the loneliest starting property is $\mathcal{O}(m)$.*

The proof of Lemma 4.2 has been omitted since it is trivial. Using Lemma 4.2, the number of generated paths for Algorithm 1 in the worst case can be reduced from $\mathcal{O}(m^2 8^t)$ to $\mathcal{O}(m 8^t)$. However, note that the loneliest starting property does not

guarantee the minimum number of paths generated. Starting at another property with a larger initial set of valid paths may in the end generate fewer number of paths.

4.1.2 Reversed Initial Paths

The set of initial valid paths may include a path and its reversal. For example, using the loneliest starting property P_0, both paths $(0,0) \to (1,0)$ and $(1,0) \to (0,0)$ will be considered since they start in P_0, see Figure 4.2. However, only one of these paths need to be considered in the initial set of valid paths.

Proposition 4.1. *Consider a path and its reversal. Only one of them needs to be considered in the initial set of valid paths.*

Proof. Let P be the initial set of valid paths. Consider that $s \to t \in P$ and $t \to s \notin P$. This means that the path $s \to t$ would be extended as a new path $s \to t \to ...$, while the path $t \to s$ would not be extended as a new path $t \to s \to ....$ However, if $t \to s \to ... \to t$ is a solution cycle it will be generated from the initial path $s \to t$, since a cycle is immaterial and can be traversed in either direction, see Section 2.1. □

4.1.3 Hypotenuse

We may reduce the vertex set R of the valid position graph $G[R]$ by removing vertices on the hypotenuse H of the position graph, i.e, the set of vertices of the form (i,i). Proposition 4.2 and its proof is based on contributions by Damaschke [5].

Proposition 4.2. *Vertices in H are never needed in a solution cycle C.*

Proof. Consider the case when C contains a position (i,i) and its two neighbours in C are not in H. These two neighbours are either identical or adjacent since they must be one of $(i,i-1)$, $(i+1,i-1)$ or $(i+1,i)$. If the two neighbours are $(i,i-1)$ or $(i+1,i)$, we can simply remove (i,i) from C since both neighbours visit P_i. If the two neighbours are $(i,i-1)$ and $(i+1,i-1)$, the position (i,i) can be replaced with $(i+1,i)$. One can proceed similarly in the symmetric case or if both neighbours are $(i+1,i-1)$. Now we consider the case when C contains a path of two or more consecutive positions in H. Let $(i-1,i-1) \to (i,i)$ be the end of this path. Hence, the next position in C is $(i+1,i)$, $(i+1,i-1)$ or $(i,i-1)$. In either case, (i,i) may simply be removed or replaced with $(i,i-1)$ in C. Thus, all vertices in H can be successively removed and the triangle graph is reduced to the positions (x,y) with $x > y$. □

By using Proposition 4.2, the number of generated paths will be reduced. The reason is simply that the vertex and edge set of the valid position graph will be reduced,

effectively reducing the number of possible paths that can be generated. Figure 4.3 illustrates a triangle graph with the hypotenuse excluded.

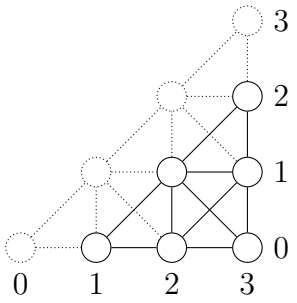


Figure 4.3: A triangle position graph for $m = 3$ with the hypotenuse excluded.

4.1.4 Redundant Paths

Consider a valid path p in the valid position graph and a subpath q_1 of p that starts at position s and ends at position t. If q_1 in p can be replaced by another path q_2 that starts at s and ends at t, and p is still valid, then one of q_1 or q_2 can be considered a *redundant* path. If the valid position graph contains k redundant paths of length x, and these are further extended up to some length $y > x$, then using Equation 4.1 we can conclude that there exists at most $k8^{y-x}$ paths that are extended from the k redundant paths. These must also be redundant and we may therefore reduce the total number of generated paths of length y in Algorithm 1 with $k8^{y-x}$ paths.

Fingerprints A general case when two valid paths are redundant is when they have the same *fingerprint*. The fingerprint of a path p is defined as the vector

$$(s, t, l(s, s_0), ..., l(s, s_m), l(t_0, t), ..., l(t_m, t))$$

where s and t is the first and last position of p, respectively, s_i and t_i is the first and last position of p that has property i, respectively, and $l(u, v)$ is the length of the subpath from position u to position v in p. If two valid paths p_1 and p_2 have the same fingerprint, then one of p_1 and p_2 can be considered a redundant path. Note that if a path is valid, its fingerprint only depends on the first and last w_{max} positions of the path, respectively, where w_{max} is the maximum waiting time of the instance. The reason is that if any of the lengths $l(s, s_i)$ or $l(t_i, t)$ is greater than w_{max}, then the waiting time of property i is violated and the path can not be valid. It follows that many valid paths can have the same fingerprint since the inner positions of a path that are not considered in the fingerprint can differ in many ways, hence two paths can have equal fingerprints regardless of their lengths.

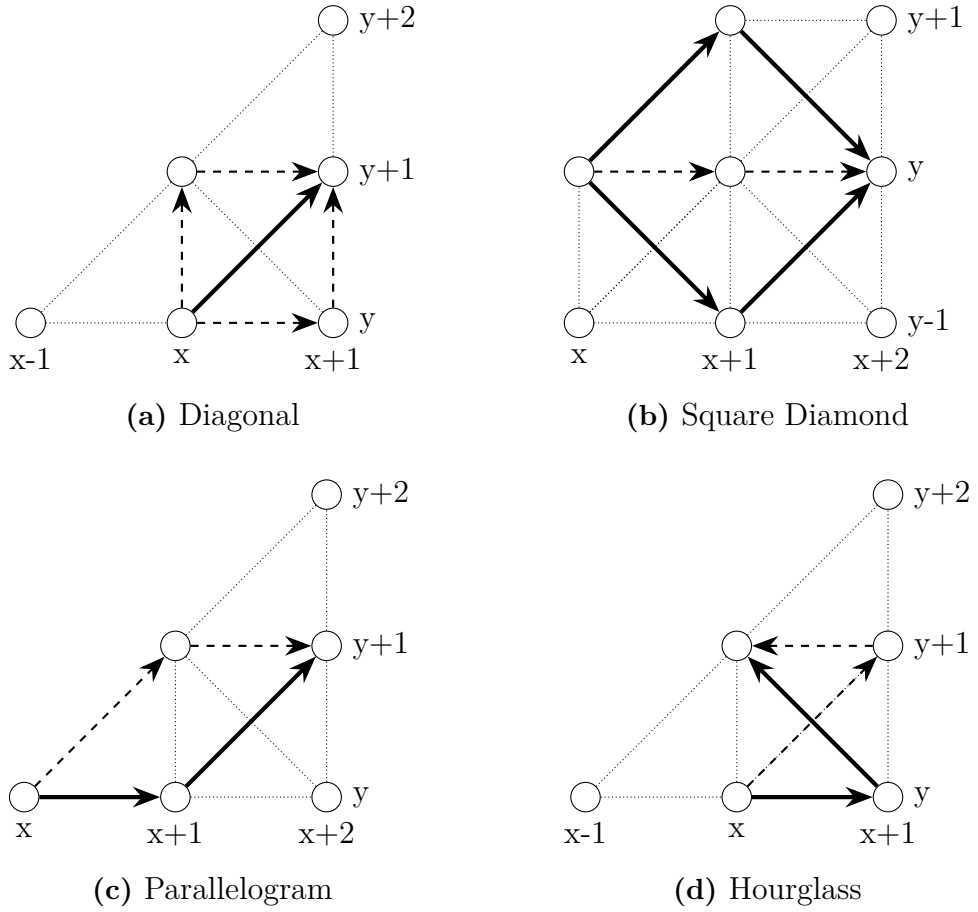
There are special cases where two paths with unequal fingerprints may still be redundant paths. For the following cases, we consider p, q_1 and q_2 as defined in the beginning of this section. The arguments for all of the following paths also hold similarly for any symmetric case.

Diagonal Consider the subpath $q_1 = (x, y)(x+1, y)(x+1, y+1)$. The intermediate position $(x + 1, y)$ can simply be removed since the first and last position in q_1 visit the same properties in fewer steps. Thus, q_1 may be replaced by the path $q_2 = (x, y)(x+1, y+1)$. Figure 4.4a illustrates an example where the dashed arrows correspond to q_1 and the solid line corresponds to q_2. Generally, we may say that a path $a \to b \to c$ is redundant if $|a_x - c_x| = 1 \wedge |a_y - c_y| = 1$ since the path may be reduced to $a \to c$ given the same argument as previously.

Square Diamond Consider the subpath $q_1 = (x, y)(x + 1, y)(x + 2, y)$. This path may be replaced by $q_2 = (x, y)(x+1, y+1)(x+2, y)$ if $(x+1, y+1)$ is a valid position, or by $q_2' = (x, y)(x + 1, y - 1)(x + 2, y)$ if $(x + 1, y - 1)$ is a valid position. This holds since q_2 and q_2' still fulfill the properties x, $x + 1$ and $x + 2$ at the same time units as q_1 but with the addition of also visiting property $y + 1$ or $y - 1$ in between the visits of property y. The property y would only be violated if its waiting time is 1, but then the positions $(x + 1, y + 1)$ and $(x + 1, y - 1)$ would not be valid because of the range of property y. Figure 4.4b illustrates an example where the dashed arrows correspond to q_1 and the solid arrows illustrate q_2 and q_2'.

Parallelogram Consider the subpath $q_1 = (x, y)(x + 1, y + 1)(x + 2, y + 1)$. This path may be replaced by the path $q_2 = (x, y)(x+1, y)(x+2, y+1)$ if the intermediate position $(x + 1, y + 1)$ of q_1 was not the last chance to visit property $y + 1$. The two paths visit the same properties at the same time unit, except that q_2 delays the visit of property $y+1$ by one time unit by staying on property y an extra time unit. Staying at property y an extra time unit is not harmful, but delaying the visit of property $y + 1$ may be harmful; p may become invalid, or an extension of p that contains q_2 and is a cycle may not be valid when traversing the cycle, while the same extension of p containing q_1 instead of q_2 is valid because property $y+1$ can be fulfilled one time unit earlier. Hence, if q_1 is replaced by q_2 and one knows that the intermediate position $(x + 1, y + 1)$ of q_1 was not the last chance to visit property $y + 1$, then q_1 is a redundant path. Figure 4.4c illustrates an example where the dashed arrows correspond to q_1 and the solid line corresponds to q_2. Note that the previous arguments only hold when replacing q_1 by q_2 and not vice versa. This is because leaving property y one time unit earlier may be harmful; it might be that p is still valid when replacing q_2 by q_1, but an extension of p containing q_1 is not valid while the same extension of p containing q_2 instead of q_1 is valid. Hence, if q_2 is replaced by q_1 and one knows that it is not harmful to leave property y one time unit earlier, then q_2 is a redundant path.

Hourglass Consider the subpath $q_1 = (x, y)(x+1, y+1)(x, y+1)$. This path may be replaced by the path $q_2 = (x, y)(x + 1, y)(x, y + 1)$ if the intermediate position $(x + 1, y + 1)$ of q_1 was not the last chance to visit property $y + 1$. The arguments are similar to the case of Parallelogram. Figure 4.4d illustrates an example where the dashed arrows correspond to q_1 and the solid arrows correspond to q_2.



(a) Diagonal **(b)** Square Diamond

(c) Parallelogram **(d)** Hourglass

Figure 4.4: Paths with dashed arrows may be replaced by paths with solid arrows.

Wings Consider the subpath $q_1 = (x, y)(x+1, y)(x+2, y+1)(x+3, y+1)$. This path may be replaced by the path $q_2 = (x, y)(x+1, y+1)(x+2, y)(x+3, y+1)$ if $(x+1, y+1)$ and $(x+2, y)$ are valid positions. This holds since the properties x, $x+1$, $x+2$, $x+3$ are still visited at the same time unit but with the difference that property $y+1$ is visited earlier while delaying the second visit of property y by one time unit. Delaying the second visit of property y is only harmful if y has a waiting time of 1. This is because property y is visited at the first position in both q_1 and q_2. However, if y has a waiting time of 1 then q_1 is not a valid path. Figure 4.5 illustrates an example where the dashed arrows correspond to q_1 and the solid arrows correspond to q_2.

4.1.5 Babysitting

A feasible instance must have a solution cycle that visits the outermost stations at some point. If there does not exist a valid path that visits one of the outermost stations of an instance, the instance must be infeasible. Consider an instance I of Scenario 2 with stations placed at $s_0, ..., s_m$. Recall that the position graph of I may contain only positions where $x > y$, see Section 3.1 and 4.2. This means that robot r_x is always to the right of robot r_y on the line. Hence, when scheduling the robots to visit each station of I, robot r_x must at some time visit s_m and robot r_y must at

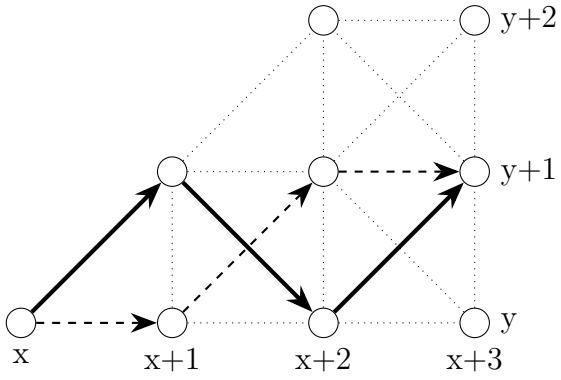


Figure 4.5: The path with dashed arrows may be replaced by the path with solid arrows.

some time visit s_0. Recall that the range $[a, b]$ of I can be patrolled by both robots, see Section 3.2.1. It follows that robot r_x can patrol the range $[a, m]$ and robot r_y can patrol the range $[0, b]$. If robot r_x leaves the range $[a, b]$ to visit s_m, robot r_y must *babysit* its entire range $[0, b]$ since r_x can not assist in the range $[a, b]$ during this event. Similarly, r_x must babysit the range $[a, m]$ if r_y traverses to s_0. The shortest paths that traverses to the outermost stations and back to the range $[a, b]$ are the two *wild card* paths

$$W_x = (b, *)(b + 1, *)...(m - 1, *)(m, *)(m - 1, *)...(b + 1, *)(b, *)$$
$$W_y = (*, a)(*, a - 1)...(*, 1)(*, 0)(*, 1)...(*, a - 1)(*, a)$$

where $(i, *)$ is any position in the valid position graph $G[R]$ with $x = i$ and similarly for $(*, i)$.

Proposition 4.3. *Consider an instance I of Scenario 2. If I is feasible, then there exists a solution cycle containing W_x or W_y. Regarding their validity, if one of W_x or W_y is invalid in the valid position graph $G[R]$ of I, then I is infeasible.*

Proof. In every solution cycle, every station is visited by at least one robot since we do not consider infinite waiting times. Assume that I is feasible, which means that there exists some solution cycle C. Consider any $i \in [a, b]$. If r_x visits s_i in C, then C contains some subpath $(b, *)...(m, *)...(b, *)$ since r_x must also visit s_m at some point. W_x is the shortest such path, thus, there exists a solution cycle C_x that contains W_x, otherwise no solution cycle can exist where r_x visits s_i. Similarly, if r_y visits s_i in C, then C contains some subpath $(*, a)...(*, 0)...(*, a)$. W_y is the shortest such path, thus, there exists a solution cycle C_y that contains W_y. Since at least one of r_x or r_y visits s_i in a solution cycle, then there exists a solution cycle containing W_x or W_y if I is feasible. The last assertion is shown as follows: if W_x is invalid in $G[R]$, it simply means that the waiting time of some station in the range $[0, m - 1]$ can not be fulfilled by W_x. It can not be waiting time t_m, otherwise $t_m < 2(m - b)$ and $a \geq a_m = m - \lfloor t_m/2 \rfloor > m - (m - b) = b$, which means that I would have been of Scenario 1. If the waiting time t_j such that $j \in [b + 1, m - 1]$ can not be

fulfilled by W_x, then either $t_j < 2(j-b)$ or $t_j < 2(m-j)$; in the first case we have $a \geq a_j = j - \lfloor t_j/2 \rfloor > j - (j-b) = b$, hence I would have been of Scenario 1. In the second case, there can not exist a solution cycle for I since r_x can not visit both s_j and s_m without violating t_j. Finally, if the waiting time t_j such that $j \in [0, b]$ can not be fulfilled by W_x, then r_y can not babysit the range while r_x visits s_m and thus, no solution cycle can exist. Hence, if I is of Scenario 2 and W_x is invalid in $G[R]$, no solution cycle can exist and I is infeasible. Similar arguments hold for W_y. □

Using Proposition 4.3, we can deem an instance as infeasible by finding the two wild card paths instead of generating exponentially many paths up to an unknown length that depends on how many valid paths exist in $G[R]$. Furthermore, we can also utilize the wild card paths for `Solve`. For example, one may initialize the set of valid paths to extend with the valid wild card paths.

4.2 Distances in the Valid Position Graph

Every time a path is validated, distances to every property are used, see Section 2.4. Distances can be computed by finding shortest paths in the valid position graph $G[R]$. There are several approaches available for finding shortest paths in a graph but their time complexity varies. In `Solve`, exponentially many paths are validated. Thus, it is crucial to choose the most efficient approach for our problem. Since the position graph consists of Cartesian coordinates, simple distance heuristics can be used to guide the search for the shortest path. Furthermore, the valid position graph has a specific shape that makes it possible to use an efficient greedy algorithm to find the shortest path.

4.2.1 Shape of the Valid Position Graph

This section is based on contributions by Damaschke [5]. For Scenario 2 in Section 3.2.1, the two numbers a and b are cornerstones in characterizing R, i.e., the vertex set of the valid position graph $G[R]$ (see Section 2.3). For $x \geq y$, R contains all positions where $a \leq x \leq b$ and all positions where $a \leq y \leq b$. Equivalently, we say that R contains the *stripes* $a \leq x \leq b$ and $a \leq y \leq b$ of the position graph. Furthermore, R cannot contain positions with $x < a$ or $y > b$ since they are out of range for some property. However, R may intersect the rectangle $Q := [b+1, m] \times [0, a-1]$ of the position graph. See Figure 4.6 for an illustration.

For any position $(x, y) \in Q$, the following conditions are equivalent:

$$(x,y) \in R \iff \forall i : (x,y) \in R_i \iff \forall i : y \geq a_i \vee x \leq b_i \iff \nexists i : y < a_i \wedge x > b_i$$

Geometrically, the conditions mean that the positions in $Q \cap R$ can be obtained from Q. This is done by cutting out all positions (x, y) where $y < a_i \wedge x > b_i$, which is

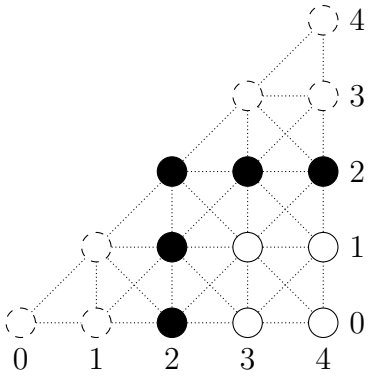


Figure 4.6: The position graph of an instance with $a = b = 2$. The rectangle Q contains the unfilled positions. R must contain the filled positions but may also intersect Q. R cannot contain the dashed positions.

equivalent to cutting out all quadrants in the position graph with upper left corner of the form (b_i+1, a_i-1). Hence, $Q \cap R$ is the region above some increasing staircase curve. The quadrants that intersect Q are characterized in Lemma 4.3.

Lemma 4.3. *$Q \cap R$ is obtained from Q by cutting out all quadrants with upper left corner of the form $(b_k + 1, a_k - 1)$, for all $k \in [a, b]$. Furthermore, we have $a_i \leq 0$ for $i < a$, and $b_j \geq m$ for $j > b$.*

Proof. Consider any $i \notin [a, b]$ in a feasible instance, say $i < a$. From the definition of a_i and b_i (Section 3.1) and from Lemma 2.1 we can conclude $b_i - a_i + 1 \geq t_i \geq 2d(P_0, P_i) = 2i$. This equation holds since P_0 and P_i are two lines that cross the same stripe in R. Furthermore, we know $b_i - a_i$ is even by definition. Thus, it follows $b_i - a_i \geq 2i$. Hence, $a_i \leq 0$ and the quadrant with upper left corner $(b_i + 1, a_i - 1)$ does not intersect Q. For $j > b$, we proceed similarly. □

Figure 4.7 illustrates examples of the valid position graph for several instances.

4.2.2 Greedy Shortest Path

The Euclidean distance, which is the smallest possible distance between two points in the Cartesian coordinate system, may be used to estimate the shortest distance between two positions in the valid position graph and guide the search for the shortest path. Using this estimation, a greedy shortest path algorithm can be used to find the shortest path. The shortest path from a position p_1 to a position p_2 is simply calculated as follows:

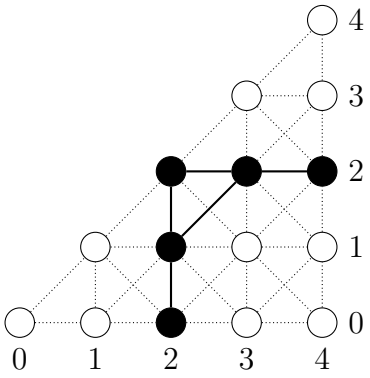
1. Let the current position be p_1.

2. For each neighbour of the current position, calculate its euclidean distance to p_2.

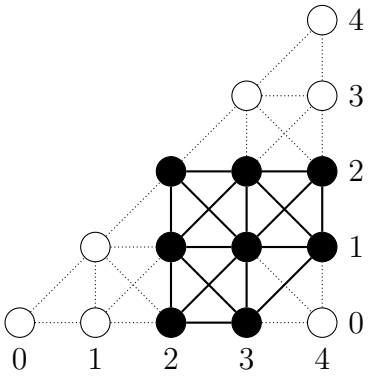
3. Traverse to the neighbour with smallest euclidean distance and update the current position.

4. Repeat task 2 until we reach p_2.

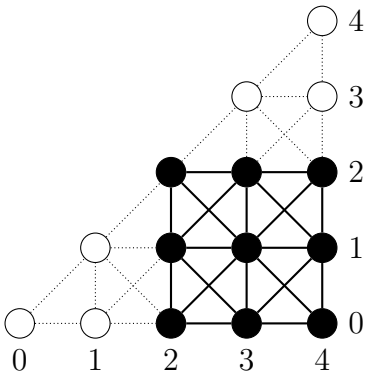
5. The shortest path is then the traversed path.



(a) Instance: (6, 4, 1, 4, 6)



(b) Instance: (4, 2, 2, 2, 4)



(c) Instance: (5, 8, 8, 2, 8)

Figure 4.7: The valid position graph $G[R]$ of different feasible instances with $a = b = 2$. R contains the filled vertices. The edge set of $G[R]$ contains the solid edges.

Proposition 4.4. *The greedy shortest path algorithm from a position $s \in R$ to a position $t \in R$ will eventually return a path from s to t. Furthermore, the returned path is the shortest path from s to t in $G[R]$.*

Proof. Let $e(s,t)$ denote the Euclidean distance from s to t. From Section 4.2.1, we know that R can only contain the positions in the rectangle $[a,m] \times [0,b]$ but may cut out some positions in $Q = [b+1,m] \times [0,a-1]$. We define a position $(x,y) \notin R$ to be a hole if $(x-c_1,y) \in R \wedge (x+c_2,y) \in R$ for some $c_1, c_2 > 0$, or $(x,y-d_1) \in R \wedge (x,y+d_2) \in R$ for some $d_1, d_2 > 0$. From Lemma 4.3, we know that if a position (x_1,y_1) is cut out from R, so is any position (x_2,y_2) where $x_2 \geq x_1 \wedge y_2 \leq y_1$. Thus, the valid position graph $G[R]$ can not contain any holes.

We can conclude from the definition of R that there must exist a path between every pair of positions in $G[R]$. The greedy algorithm from s to t will always choose the neighbour $n = (n_x, n_y)$ of the current position $c = (c_x, c_y)$ with smallest Euclidean distance to t. Since there are no holes in $G[R]$, it means that c always has a neighbour n with $c_x < n_x < t_x$ or $c_y < n_y < t_y$, hence $e(n,t) < e(c,t)$. Similarly, this holds for the case $t_x < n_x < c_x$ or $t_y < n_y < c_y$. Thus, the greedy algorithm will always find a path from s to t. Furthermore, this path is of the form $s \to s_1 \to ... \to s_k \to t$ where $e(s,t) > e(s_1,t)$, $e(s_i,t) > e(s_j,t)$ for all $1 \leq i < j \leq k$ and $e(s_k,t) > e(t,t)$. Due to this form, the shape of $G[R]$ and the choice of n in every iteration of the algorithm, the path is the shortest path between s and t in $G[R]$. □

We will always traverse towards the target horizontally, vertically or diagonally. In the worst case we have to traverse the whole graph horizontally and then vertically, never using the diagonal. The resulting path will then be of length $m + m = \mathcal{O}(m)$. For every position in the traversed path we compare its 8 neighbours Euclidean distance to the target. This is done in constant time. Thus, the time complexity for this greedy shortest path algorithm is $\mathcal{O}(m)$.

4.3 Validating Paths in Parallel

In Algorithm 1, the inner part of the while-loop (row 7-13) operates on a path p from the set of paths P. The operations done on a path p is independent of the operations on other paths. Thus, one may simply process multiple paths p in parallel. Aside from the initialization of the algorithm, the while-loop covers the entire execution time. Consider that P contains x paths at some point. The operations of every path in P can then be run in parallel instead of sequentially. At this specific time, a theoretical speed-up of x can be achieved assuming that we have infinite parallelism. Since the size of P changes dynamically depending on how many paths are further extended, it is difficult to determine the theoretical speed-up for the entire algorithm.

4.4 Look-ahead

Some instances have a solution of the form $P \to q \to P^{-1}$, where P is a sub-path, P^{-1} is the reversal of P and q is a position. For example, the instance $v = (6,4,3,3,4,6,8)$ has the following solution cycle where $P = (6,3) \to (5,2) \to (4,1) \to (3,0)$ and $q = (2,0)$:

$$(6,3) \to (5,2) \to (4,1) \to (3,0) \to (2,0) \to (3,0) \to (4,1) \to (5,2) \to (6,3)$$

In the worst case, the basic approach would generate all valid paths up to length 8 before finding this solution. If we instead, when a path $P \to q$ is valid, check if the path $P \to q \to P^{-1}$ is a solution cycle, we can eliminate exponentially many paths. We call this operation *look-ahead*. When solving v we would identify the solution cycle already after validating the path $(6,3) \to (5,2) \to (4,1) \to (3,0) \to (2,0)$ since look-ahead will find the solution cycle by extending the path with $(3,0) \to$

$(4, 1) \to (5, 2) \to (6, 3)$. Thus, in the worst case, we only need to generate all valid paths up to length 4 when solving v. When applicable, look-ahead will reduce the maximum length of the generated paths by half. However, there is an additional cost: every valid path will validate an additional path. Thus, for Equation 4.1, look-ahead will reduce t by half but add a factor of 2 to the number of paths generated for every t. For example, if we consider the instance v, look-ahead will generate at most $\sum_{x=1}^{4} 2n^{x-1}v$ paths instead of $\sum_{x=1}^{8} n^{x-1}v$ paths.

The problem with look-ahead is that, when not applicable, at most twice as many paths will be generated. Hence, if look-ahead is not applicable to a significant number of instances in `Search`, it will instead stall our search for critical instances.

5

Improving Search

This chapter introduces improvements made to the basic approach when searching for critical instances, see Section 3.3.

5.1 Lower Bound Instances

The search algorithm presented in Section 3.3 starts with the trivial lower-bound instance $(1, 1, ..., 1)$. Instead of starting with this instance, we may restrict the search space by starting with stricter lower-bounds. Remembering the definition of $[a, b]$ from section 3.2.1, we can rephrase the last statement of Lemma 4.3 to immediately get the useful lower bounds in Lemma 5.1 [5].

Lemma 5.1. *In every feasible instance we have $t_i \geq 2i$ and $t_i \geq 2(b - i)$ for all $i < a$, and similarly, $t_j \geq 2(m - j)$ and $t_j \geq 2(j - a)$ for all $j > b$.*

For a given a and b, Lemma 5.1 yields lower bounds on the waiting times of stations $i < a$ and $i > b$. The waiting times of stations $a \leq i \leq b$ can be derived from the definition of a_i and b_i, see Section 3.1. Thus, a lower bound instance can be generated for a given a and b. Let L be the set containing all these instances for each a and b.

Since the instances in L are lower bounds for a given a and b, there may exist an instance in L that is greater than or equal to another instance in L. Let $L' \subseteq L$ be the subset of the smallest lower bound instances.

In Algorithm 2, the set C is initialized to the critical instances of Scenario 1, see Section 3.3. There may exist an instance in L' that is greater than or equal to a critical instance in C. Let $L'' \subseteq L'$ be the subset excluding such instances. What remains in L'' are infeasible or critical instances. The critical instances in L'' can be added to C before proceeding with the search. The set U in Algorithm 2 may then be initialized with the infeasible instances in L''.

5.2 Visited Instances

In Algorithm 2, instances are generated and solved to check for feasibility. We say that an instance has been *visited* after this. Due to the inherent symmetry of the problem and the algorithm incrementing each waiting time of an instance, the same instance may be visited repeatedly. For example, the instance $(1, 1, t_2, ..., t_m)$ will generate the instances $(2, 1, t_2, ..., t_m)$ and $(1, 2, t_2, ..., t_m)$. Next, if these two instances are infeasible, both will generate $(2, 2, t_2, ..., t_m)$ that will be visited twice. This would create two equivalent subtrees in the search tree when it is sufficient with one. Note that `Solve` is, in the worst case, an expensive operation that will be executed on every visit. Instead, on the first visit of an instance, we may simply mark it as visited and ignore it the next time it is generated. Furthermore, consider any instance $v = (t_0, ..., t_m)$ and its reversal $v_r = (t_m, ..., t_0)$. Both v and v_r will be visited in the algorithm. However, due to symmetry, anything that holds for v also holds for v_r. Thus, we only need to visit one of v or v_r; when visiting one of them, we may mark its reversal as visited. Figure 5.1 shows all the instances that are visited when marking instances as visited. For example, note that the instance $(2, 2, 1, 1)$, including its reversal, would be visited four times at the second level if it was not marked as visited.

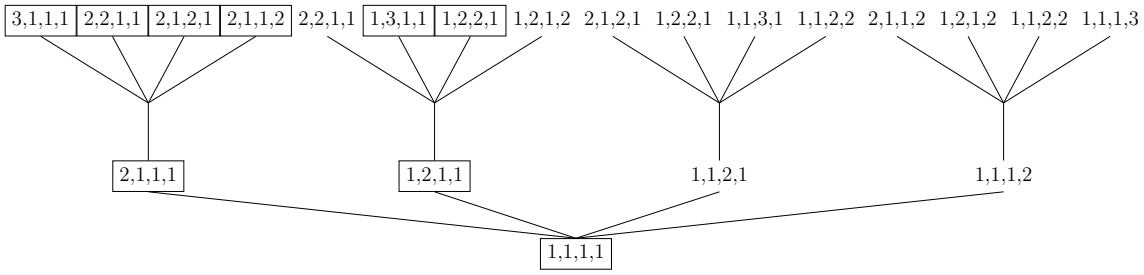


Figure 5.1: Three levels of the search tree for $m = 3$. Instances marked with a surrounding box illustrate the first visit to an instance when visiting instances level by level, left to right. Instances that are not marked has already been visited. Note that marked instances include its reversal as visited.

5.3 Maximal Infeasible Instances

In the basic approach, the upwards search has to check the feasibility of all incremented instances, see Section 3.3. For example, consider the infeasible instances $I = [1, .., 1]$ and $J = [r, ..., r, 1, 1]$, where r denotes the roof value, as discussed in section 3.3. Starting at I, the upwards search will generate and solve exponentially many instances K, such that $K < J$. However, we know that J is infeasible. Thus, all K must also be infeasible.

We may systematically generate a set of instances that contains *maximal infeasible instances*. These are instances that become feasible if any of the non-roof waiting times is further incremented. A significant amount of infeasible instances can be safely discarded by comparing an instance to instances in the set. An instance

smaller than any instance in the set is infeasible.

An algorithm has been developed in order to create a set of maximal infeasible instances. The algorithm continuously keeps track of two sets G and U. The set G contains the currently known maximal infeasible instances. The set U contains instances with unknown feasibility status. Initially, U may contain the vector $(r, ..., r)$ since this is the largest vector generated in Algorithm 2. Then, we iteratively choose an instance u from U and proceed with one of the following phases based on the feasibility status of u:

- **Decrement Phase:** If u is feasible, some waiting time has to be decremented to become infeasible. In order to quickly make u infeasible, we set each of the waiting times that is r to 1, creating one new instance for every r and inserting them in U.

- **Increment Phase:** If u is infeasible it is a currently known maximal infeasible instance unless there exists an instance $g \in G$ such that $u < g$. If it is a currently known maximal infeasible instance, u is inserted in G and any smaller instance in G is removed. However, there might exist an infeasible instance $h > g$. Therefore, we increment one of the waiting times in u that is not r by 1, creating one new instance for every waiting time that was not r and insert them in U.

Furthermore, we apply the idea of visited instances, as discussed in Section 5.2. Algorithm 3 is a full description of how to generate maximal infeasible instances. Figure 5.2 shows four levels of the search tree when searching for maximal infeasible instances for $m = 2$.

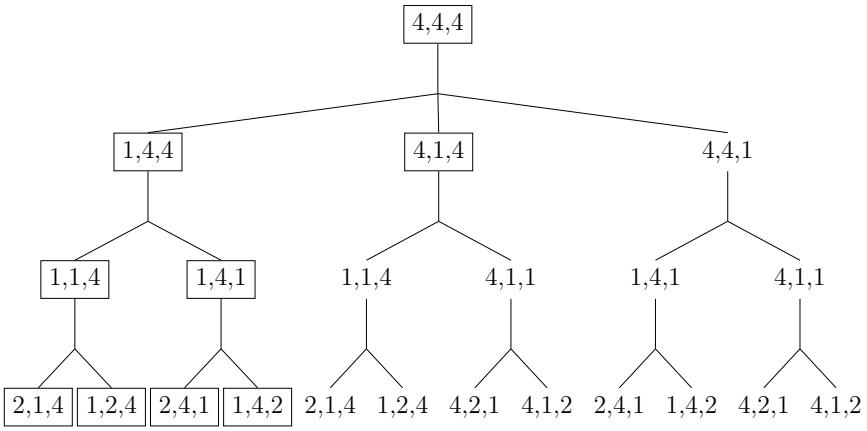


Figure 5.2: Four levels of the search tree when searching for maximal infeasible instances for $m = 2$. Instances marked with a surrounding box illustrate the first visit to an instance when visiting instances level by level, left to right. Instances that are not marked has already been visited. Note that marked instances include its reversal as visited.

Algorithm 3 Algorithm for generating maximal infeasible instances for m

```
1: U ← Set of instances with unknown feasibility status.
2: G ← Set of known maximal infeasible instances
3: V ← Set of visited instances

4: function GenerateMaximalInfeasibleInstances(m):
5:     add (r, ..., r) to U
6:     while U not empty do
7:         u ← remove an instance from U
8:         if u feasible then
9:             for i = 0, ..., m do
10:                if u_i = r then
11:                    v ← copy of u
12:                    v_i = 1
13:                    if v ∉ V then
14:                        add v to U
15:        else
16:            if ∃g ∈ G(u ≤ g) then
17:                continue
18:            else
19:                remove all g ∈ G where u > g
20:                add u to G
21:                for i = 0, ..., m do
22:                    if u_i ≠ r then
23:                        v ← copy of u
24:                        v_i = v_i + 1
25:                        if v ∉ V then
26:                            add v to U
27:    return G
```

Proposition 5.1. *Algorithm 3 eventually returns a set G containing maximal infeasible instances.*

Proof. An instance g is only inserted in G if g is infeasible. Hence, G will not contain any feasible instances. Furthermore, if g is infeasible and there exists an instance $h \in G$ such that $g \leq h$, the instance g is not inserted in G. For the case when g is inserted in G, there might exist an instance $f \in G$ such that $f < g$. Any such instance f will be removed when inserting g in G. Thus, if the algorithm has terminated, there does not exist a pair of instances $g_1, g_2 \in G$ such that $g_1 < g_2$. Assume that there exists a $g \in G$ that is not a maximal infeasible instance, i.e., there exists an infeasible instance $h \notin G$ such that one non-roof waiting time $h_i = g_i + 1$ and all other waiting times of g and h are equal. After g was inserted in G, the algorithm generated h and inserted it in U. Since U is non-empty, the algorithm proceeds and will eventually check the feasibility of h and insert it in G, thus removing g from G. The algorithm proceeds similarly with h unless it is a maximal infeasible

instance. If h is a maximal infeasible instance, it will never be removed from G. Hence, if the algorithm terminates, any $g \in G$ is a maximal infeasible instance. The algorithm terminates if and only if U is empty. The only possible instances that can be generated by the algorithm are instances between $(1, ..., 1)$ and $(r, ..., r)$. Thus, there are only finitely many instances that can be generated and inserted in U. Since we only insert instances in U if they have not been visited already, the algorithm will visit finitely many instances and eventually terminate. □

5.4 Solving Instances in Parallel

Recall that `Search` works upward one level at a time. Furthermore, from the proof of Proposition 2 we know that if a feasible instance is found at a level it is critical. This means that there is no dependency between any two instances on the same level. Therefore, for each level, we can process each instance in parallel. Assuming that we have infinite parallelism and every instance takes equal amount of time to process, we would get, at each level, a theoretical speedup of the number of instances to process. However, processing an instance includes using `Solve` which has an unpredictable running time given an instance.

6

Implementation

This section introduces PATSCH, which is a computer program that implements `Solve` and `Search`, including the best improvements, as a Java 11 library. The name PATSCH is a portmanteau of the words *Patrolling* and *Schedules*. The components of PATSCH are presented in the following subsections, which are divided in accordance with the implementation's repository[1]. In these sections, the use of the word instance may refer to an instance of the patrolling problem or a Java instance. Therefore, unless stated otherwise, the word instance refers to an instance of the patrolling problem. Furthermore, PATSCH uses hash maps and hash sets extensively and these are mentioned throughout this section. Unless stated otherwise, they have been implemented using the `HashMap` and `HashSet` implementation from `java.util`. Assuming the hash function of the elements is collision-resistant, `HashMap` offers constant time performance for the basic operations get and put and `HashSet` offers constant time performance for the basic operations add, remove, contains and size [7, 8].

6.1 Models

The classes in this section model the fundamental components of the patrolling problem.

Some of the classes are heavily used as elements in hash maps and hash sets. Such classes must override two functions, `equals` and `hashCode`, to function properly with the data structures [9, p. 37]. Furthermore, when overriding these functions, their general contracts must be obeyed. For example, the `equals` function must be reflexive, symmetric, transitive and consistent [10]. A typical method for producing a good hash code is to compute an integer value based on the significant fields of a class, which are further mentioned in each class that overrides the `hashCode` function [9, p. 50-54].

[1] Available at: `https://github.com/radjavi/patsch`

6.1.1 The `Position` class

The `Position` class models a position (x, y) in the position graph using two integer variables; one for x and one for y. This class is heavily used with hash sets and hash maps; the `equals` and `hashCode` function has been implemented using typical methods for significant fields of primitive types, which in this class are the integers x and y [9, p. 37-54].

6.1.2 The `Range` class

The `Range` class models the range of a property, see Section 2.3. The constructor of the class takes as argument a `Property` instance. In the `Range` class, we represent the range of a property i as two integers a and b, which correspond to the range $[a_i, b_i]$ explained in Section 3.1. Furthermore, a hash set of `Position` instances are created that contains all positions of the position graph that are in the range $[a, b]$, i.e., the positions where $a \leq x \leq b$ or $a \leq y \leq b$.

6.1.3 The `Property` class

The `Property` class models one property of an `Instance` class, which is presented in Section 6.1.5. The constructor of the class takes as arguments an instance of `Instance` and the property's index i. Using these arguments, the constructor creates a hash set containing `Position` instances, i.e., positions that has the property. As discussed in section 3.1, the positions of a property i are all positions in the position graph where $x = i$ or $y = i$. Finally, the constructor creates a `Range` instance that corresponds to the range of the property.

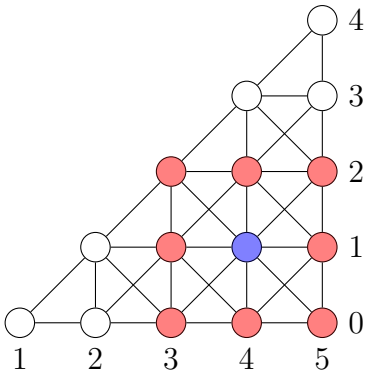
6.1.4 The `PositionGraph` class

The constructor of the class takes a set of `Position` instances and connects them to form a position graph. The position graph has been implemented as a hash map that maps a `Position` to its adjacency set. This approach offers constant time performance for collecting the neighbours of a `Position`, which is beneficial since `Solve` extensively operates on these. Figure 6.1a illustrates a position and its eight neighbours, while figure 6.1b shows the corresponding mapping of this position to its adjacency set.

6.1.5 The `Instance` class

The `Instance` class models an instance of the patrolling problem. The constructor of the class takes as argument a list of $m+1$ integers `int[] waitingTimes` representing an instance $(t_0, ..., t_m)$ and creates a minimal version of an instance that can be used for comparing instances. Creating a minimal version, that only contains the integer waiting times, is important for reducing memory usage. This is explained further after introducing the extended version of the class.

The extended version of the `Instance` class contains a list of `Property` instances



(a) Position $(4, 1)$ (blue) in the position graph with all its eight neighbours (red).

```
(4,1) -> {(3,0), (4,0), (5,0),
   (3,1), (5,1), (3,2), (4,2),
   (5,2)}
```

(b) The corresponding hash map entry for the position $(4, 1)$.

Figure 6.1: The mapping of a position to its neighbours in the position graph for $m = 5$.

that correspond to the $m+1$ properties. Furthermore, the instance contains the valid position graph, which is an instance of the `PositionGraph` class. This valid graph is simply computed by intersecting the set of positions from the `Range` instance of all $m + 1$ `Property` instances, which is an $\mathcal{O}(m^3)$ operation. Although this operation has an expensive time complexity, it will not be a significant fraction of the overall runtime of `Search` since the valid graph is only computed once for every instance and m is rather small in our successful runs. Using all $m + 1$ `Range` instances, two integers a and b are computed. These integers correspond to the definition of $[a, b]$ of Scenario 2 as explained in Section 3.2.1. Note that if $a > b$, an instance is of Scenario 1. An instance of the `Instance` class also contains an instance of `DistanceStorage`, which is used for caching distances in the `PositionGraph` and is further explained in Section 6.2.2. An `Instance` is solved using the function `solve`, which in turn uses the `InstanceSolver` class explained in Section 6.2.1.

Constructing a minimal version of `Instance` is important since `Search` creates exponentially many `Instance` objects and only the minimal version is needed to, e.g., check if an `Instance` has already been visited before solving it (see Section 5.2). The minimal version of an instance has a $\mathcal{O}(m)$ space complexity, while the extended version has a worst-case space complexity of $\mathcal{O}(m^3)$; the extended version contains $m+1$ properties, each having a range containing $\mathcal{O}(m^2)$ positions in the worst case, and the `DistanceStorage` instance may store at most $\mathcal{O}(m^3)$ distances between properties and positions.

The `Instance` class contains a generic function `distance` that takes two arguments, `from` and `to`. Because of the function being generic, each argument can be an instance of any class. However, the function has been restricted to instances of the `Position` or `Property` class since only these are considered when computing distances. To be able to compute the distance between the two types of classes, the arguments are converted to two sets of `Position` instances. Then, the distance between two sets of positions is computed in accordance with the definition of $d(X, Y)$

explained in Section 2.1.1. To choose the pair of positions, one from each set, that are closest to each other in the position graph, one can heuristically choose the pair of positions with minimum Euclidean distance. Note that this is not the actual distance in the position graph. However, the pair of positions with minimum Euclidean distance must also have the minimum actual distance. The actual distance is computed using a function `shortestPath`, which implements the greedy approach explained in Section 4.2.2. The function `shortestPath` takes as arguments two `Position` instances.

This class is extensively used in hash maps and hash sets; the implementation of the `equals` and `hashCode` functions adopt typical methods for significant fields of primitive types, which in this class are the $m + 1$ integer waiting times that solely identifies an instance [9, p. 37-54]. Additionally, these functions have been slightly modified to support that an instance and its reversal are considered equal. For the `equals` function, this simply means an additional comparison with the reversal of one of the operand's waiting times. The general contract of the `hashCode` function states that two equal `Instance` objects must produce the same hash code, which is not the case when applying the typical method since an `Instance` and its reversal would produce different hash codes. This can be fixed as follows. Consider the instance $I = (t_0, ..., t_m)$ and its reversal $I_r = (t_m, ..., t_0)$. If I_r is lexicographically larger than I, use I_r to produce the hash code of both I and I_r. Otherwise, use I to produce the hash code of both I and I_r.

6.1.6 The `Path` class

The constructor of the `Path` class takes an instance of `Instance` since a path is related to the position graph and properties of an instance. The path itself is represented by a linked list of `Position` instances and two integer arrays s_i and f_i that contains the path index of the first and last position, respectively, that has property i. A path can be extended by adding positions to the tail of the linked list. Furthermore, when extending a path, all s_i and f_i are computed for each property i by iterating over the linked list.

The class contains a function `valid` that is used to validate a `Path`. The function implements the approach exactly as explained in Section 2.4. To check if a `Path` is a solution cycle, the class contains a convenient function `isSolutionCycle`. This function checks that the path is a cycle, is valid and visits all properties. Moreover, the length of the subpath $f_i \to ... \to f$ + the length of the subpath $s \to ... \to s_i$ must be at most t_i for each property i. This corresponds to the path being valid when traversing the cycle repeatedly. Finally, the class also contains the function `redundant` that uses the wrapper `RedundantPaths` (Section 6.2.4) and a function `fingerprint` that returns the fingerprint as a list, which corresponds to the vector defined in Section 4.1.4.

6.2 Wrappers

This section describes classes that implement functions and data structures used in `Patsch`. These classes wrap their implementation to increase the abstraction level when using them.

6.2.1 The `InstanceSolver` class

The wrapper `InstanceSolver` contains necessary functions needed when solving an instance. The entry-point of `InstanceSolver` is a static function `solve` that takes an instance of `Instance` as argument and solves it.

To solve an instance, the function first checks if the instance is of Scenario 1 or one of the following holds for an instance of Scenario 2: $[a, b] = [0, 0]$, $[a, b] = [m, m]$ or $[a, b] = [0, m]$. If this is the case, the robots can independently patrol the line and the instance is trivially solved by comparing it to all critical instances from Proposition 3.1.

For all other cases, the function proceeds by looking for the two types of babysitting wildcard paths (Section 4.1.5). This is done by using the function `findBabysitting-Paths`, which for each type of wildcard path, extends paths in the valid position graph to the form of the wildcard path. Futhermore, a path is not extended if it is redundant or a path with an equivalent fingerprint has already been extended. If there are no valid paths of any type, the instance is deemed infeasible. However, if there are valid paths of both types, we may initialize the paths to extend in `Solve` with the valid wildcard paths in order to heuristically reduce the total number of generated paths when further solving the instance in the next step. Note that we may always initialize the paths to extend with babysitting paths, meaning that we do not need to consider the improvements dealing with loneliest starting property and reversed initial paths.

Using the function `extendPath`, `solve` proceeds by extending a path in the list of valid paths to extend, until either the list becomes empty or a solution is found. The list of valid paths to extend is implemented using a linked list. During the algorithm, paths will be inserted in correct order, from shortest to longest. A linked list has a time complexity of $\mathcal{O}(1)$ for insertion and removal of the first and last elements, which means that removing the path with shortest length is done efficiently. The function `extendPath` corresponds to the lines $8 - 13$ in Algorithm 1. Additionally, a path is not extended if it is redundant or a path with an equivalent fingerprint has already been extended. The fingerprint of paths that have already been extended is stored in a hash set.

6.2.2 The `DistanceStorage` class

When validating a path, the distance between a property and a position is used several times, see section 2.4. For example, the start node of a path is always the same when extending the path. This means that, for every extension, the distance $P_i \to s$ would be computed for every property i. Therefore, previously computed distances are cached using the `DistanceStorage` class. The underlying storage is implemented using a hash map, which maps a pair of property and position to the distance between them. By storing previously computed distances for an instance, the number of distance computations may be significantly reduced and an $\mathcal{O}(1)$ lookup may be utilized for distances in the storage. The trade-off is an increased space complexity; since only distances between $m+1$ properties and $\mathcal{O}(m^2)$ positions are needed, the space complexity is $\mathcal{O}(m^3)$ in the worst case.

6.2.3 The `SingleExecutor` class

To utilize parallelism in Patsch, the `ExecutorService` class from the package `java.util.concurrent` has been used. Executors are objects that encapsulate thread management and creation, and the `ExecutorService` class is an interface with features that help manage the life cycle, both of the threads' individual tasks and of the executor itself [11, 12].

The `SingletonExecutor` class is a singleton, which ensures that only one instance of the `ExecutorService` class is created and provides a global point of access to it via the static function `getInstance` [13]. A static function `init` must be used in order to specify the level of parallelism when creating the single instance of `SingletonExecutor`. The underlying `ExecutorService` uses a work stealing thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce thread contention [14]. The single instance of `SingletonExecutor` can then be used to submit tasks to the `ExecutorService`, which will schedule the tasks to be run in parallel.

6.2.4 The `RedundantPaths` class

In the implementation of `Solve`, valid paths are extended if they are not redundant. To check if an extended path p is redundant, only the last three or four positions need to be considered depending on the type of the redundant path. This is because the other positions in p will have already been checked to be redundant before extension.

The wrapper class `RedundantPaths` uses two functions `length2` and `length3`. The function `length2` checks if the last three positions of p is of the form Diagonal or Square Diamond, see figure 4.4. Note that we did not implement Parallelogram nor Hourglass. This was due to the time limit of this project, which restricted us from implementing these correctly. The function `length3` checks if the last four positions of p is of the form Wings, see figure 4.5. Since it is difficult to generalize these forms, a function `direction` is used, which takes two adjacent positions of p as arguments, p_i and p_j, and returns the cardinal or ordinal direction from p_i to p_j.

Using `direction`, the form of the path's tail can be identified and, using a switch-case, we check if the path has the form of one of the redundant paths. For example, if the subpath containing the last three positions of p moves in the directions "East $\rightarrow$ East", then p is redundant and of the type Square Diamond.

6.2.5 The `InstanceLevelBuckets` class

Since it is important that `Search` removes instances with lowest level from U in Algorithm 2, an efficient data structure has been implemented that stores instances sorted by their level. The underlying implementation is a hash map that maps a level to the set of instances on this level, which are called buckets. The class `ConcurrentHashMap` from `java.util.concurrent` has been used in order to allow concurrent modification and synchronization of the hash map when searching in parallel. When `Search` adds an instance to U, the wrapper class `InstanceLevelBuckets` adds the instance to the correct bucket given the level of the instance. Insertion in buckets, given the level, has a time complexity of $\mathcal{O}(1)$ since insertion in a `ConcurrentHashMap` is $\mathcal{O}(1)$. When `Search` proceeds to a higher level, the set of instances on this level is collected from the class `InstanceLevelBuckets` with a time complexity of $\mathcal{O}(1)$.

6.3 The `Search` class

The `Search` class contains functions that are needed in order to search for critical instances. The entry-point of `Search` is a function `searchForCriticalInstances`, which takes as arguments the two integers m and r. If the `SingletonExecutor` has been instantiated, a parallel version of `Search` will be used.

The first step when searching for critical instances is to initialize a hash map C of currently known critical instances with the ones from Proposition 3.1. The hash map maps a critical `Instance` to its solution cycle, which is an instance of `Path`. The function `criticalsWithEmptyIntersection` collects all these critical instances, which are then inserted in C.

The second step is to initialize a set U of current maximal instances that are known to be infeasible with the infeasible lower bound instances in L, see Section 5.1. The set U is implemented using the wrapper class `InstanceLevelBuckets`, see Section 6.2.5.

As a third step, a hash set M of maximal infeasible instances is initialized. The set is initialized using the function `generateMaximalInfeasible` which implements Algorithm 3 in order to generate all the maximal infeasible instances.

After initialization, the search proceeds upwards, level by level. A set W keeps track of instances that have already been visited. The set W contains minimal versions of the `Instance` class and is implemented as a `Set` backed by a `ConcurrentHashMap` to allow concurrent modification and synchronization when searching in parallel. On every new level, W may be cleared since instances on one level may only have

been previously visited on the same level. Next, the set of instances U_{min} with the currently smallest level is popped from U and the instances are then processed in accordance with rows $8 - 20$ in Algorithm 2, except for some modifications:

1. The set M is iterated over and if there exists a maximal infeasible instance $m \in M$ such that for $u \in U_{min}$ it holds $u \leq m$, then instead of incrementing all waiting times of u, only the waiting times u_i where $m_i < r$ are incremented.

2. After an incremented instance v is created, it is discarded if $v \in W$. Otherwise, w is inserted in W and the algorithm proceeds.

If the parallel version of `Search` is used, the instances in U_{min} are processed in parallel by submitting them as tasks to the `ExecutorService` contained in the `SingletonExecutor` wrapper.

Finally, since a critical instance and its reversal are considered equal, the search only finds one of them. Therefore, the function `searchForCriticalInstances` ends with manually inserting reversed critical instances to C.

7

Results

This chapter contains results of running `Search` and `Solve` using Patsch. Section 7.1 presents the impact of improvements to `Solve` and statistics about the execution time and number of validated paths when solving instances. In Section 7.2, the impact of improvements to `Search` is presented. Furthermore, the execution time and number of instances solved when searching for critical instances for a given m and r is also presented. Finally, Section 7.3 shows the number of critical instances found for a given m and r.

7.1 Benchmarking `Solve`

The benchmarks in this section have been executed on a on a shared resource with a Intel Xeon Gold 6126 (2.6 GHz) processor and 768 GB RAM.

7.1.1 Improvements

Here we present the results of improvements made to the basic approach of `Solve`. Since there are many improvements to `Solve`, we have decided to combine them into a few implementations that we compare:

- `ImplSolveBasic`: Implements the basic approach. Shortest paths in the valid position graph are computed using the most widely known best-first search called A* search, which is optimal using the Euclidean distance as a heuristic function [15].

- `ImplSolve2`: Extension of `ImplSolveBasic` with the following improvement:
  - Greedy shortest path (Section 4.2.2).

- `ImplSolve3`: Extension of `ImplSolve2` with the following improvements:
  - Loneliest Starting Property (Section 4.1.1).
  - Reversed Initial Paths (Section 4.1.2).
  - Hypotenuse (Section 4.1.3).

- `ImplSolve4`: Extension of `ImplSolve3` with the following improvement:
  - Redundant paths excluding Parallelogram and Hourglass (Section 4.1.4).
- `ImplSolveBest`: Extension of `ImplSolve4` with the following improvement:
  - Babysitting (Section 4.1.5).

Since `ImplSolveBasic` is slow and each of the above improvements has an effect on all types of instances, the implementations are only compared by solving two instances of Scenario 2:

1. The infeasible instance $I = (16, 16, 7, 16, 2, 16, 16, 16, 16)$.
2. The feasible instance $F = (16, 16, 7, 16, 3, 16, 16, 16, 16)$.

For each of the two instances I and F, table 7.1 shows the execution time and the number of validated paths for different implementations.

Table 7.1: The execution time and number of validated paths when solving I and F using different implementations of `Solve`.

	I		F	
	Ex. Time (s)	**Validated Paths**	**Ex. Time (s)**	**Validated Paths**
`ImplSolveBasic`	369	362,778,895	105	68,842,476
`ImplSolve2`	379	362,778,895	102	68,842,476
`ImplSolve3`	24.9	21,424,011	0.18	253,267
`ImplSolve4`	0.388	304,195	0.096	56,447
`ImplSolveBest`	0.00067	66	0.01	5,239

7.1.2 Solving Instances in `Search` using `ImplSolveBest`

Table 7.2 shows statistics about the execution time and number of validated paths for instances of Scenario 2 that are solved in the best implementation of `Search` using `ImplSolveBest` and a roof value of $r = 2m$.

7.1.3 Solving Random Instances using `ImplSolveBest`

Table 7.2 shows statistics that are representative for instances solved when searching for critical instances in `Search` with a roof value of $r = 2m$. However, these instances do not highlight statistics about instances with larger waiting times, i.e., larger than $2m$. Instead of raising the roof value r, which leads to an explosion in the number of instances, 1000 infeasible instances and 1000 feasible instances with larger waiting times have been randomly generated, see Table 7.3 and Table 7.4. The waiting times are randomly chosen in the range $[1, 5m]$. Every instance has been solved 11 times and the median execution time was used for the statistics.

Table 7.2: Statistics about the execution time and number of validated paths for instances of Scenario 2 that are solved in the best implementation of `Search` using `ImplSolveBest` and a roof value of $r = 2m$.

		Execution Time					Number of Validated Paths				
	No. Instances	Mean (ms)	Median (ms)	Min (ms)	Max (ms)	STD (ms)	Mean	Median	Min	Max	STD
m=4											
Infeasible	37	0.11	0.07	0.02	0.56	0.11	1.9	1	1	7	1.7
Feasible	30	0.68	0.42	0.14	4.3	0.79	36	25	6	124	34
m=5											
Infeasible	108	0.11	0.09	0.02	0.56	0.08	5.9	2	1	23	6.7
Feasible	61	0.83	0.69	0.12	6.2	0.86	112	72	8	343	102
m=6											
Infeasible	2,251	0.26	0.17	0.01	10	0.34	178	40	1	1,174	227
Feasible	117	1.1	0.87	0.07	7	1.0	346	274	10	2,114	384
m=7											
Infeasible	144,620	0.96	0.78	0.01	29	0.92	1,126	857	1	9,742	1,235
Feasible	197	1.8	1.2	0.08	20	2.1	975	696	12	5,450	1,117
m=8											
Infeasible	6,299,777	1.6	0.36	0.01	86	3.7	1,966	206	1	71,381	5,091
Feasible	298	5.6	2.3	0.11	138	14.6	3,363	1,395	14	73,626	8,618

Table 7.3: Statistics about the execution times and number of validated paths when solving 1000 random infeasible instances of Scenario 2 using `ImplSolveBest`.

1000 Infeasible

	Execution Time					Number of Validated Paths				
m	**Mean (ms)**	**Median (ms)**	**Min (ms)**	**Max (ms)**	**Std (ms)**	**Mean**	**Median**	**Min**	**Max**	**Std**
7	0.07	0.01	0.001	2.2	0.18	68	20	1	1,830	159
8	0.12	0.02	0.001	2.9	0.32	118	32	1	2,149	271
9	0.28	0.03	0.001	6.9	0.83	243	44	1	5,243	640
10	0.89	0.08	0.001	33	3.5	664	95	1	19,634	2,377

Table 7.4: Statistics about the execution times and number of validated paths when solving 1000 random feasible instances of Scenario 2 using `ImplSolveBest`.

1000 Feasible

	Execution Time					Number of Validated Paths				
m	**Mean (ms)**	**Median (ms)**	**Min (ms)**	**Max (ms)**	**Std (ms)**	**Mean**	**Median**	**Min**	**Max**	**Std**
7	4.8	2.0	0.01	49	7.8	3,804	1,543	12	37,755	6,244
8	47	7.0	0.02	889	112	33,595	5,337	14	581,070	77,292
9	126	17	0.02	3,618	358	80,286	12,302	16	1,786,972	204,609
10	1,826	65	0.03	56,148	5,738	849,659	41,503	18	23,009,282	2,510,683

7.2 Benchmarking `Search`

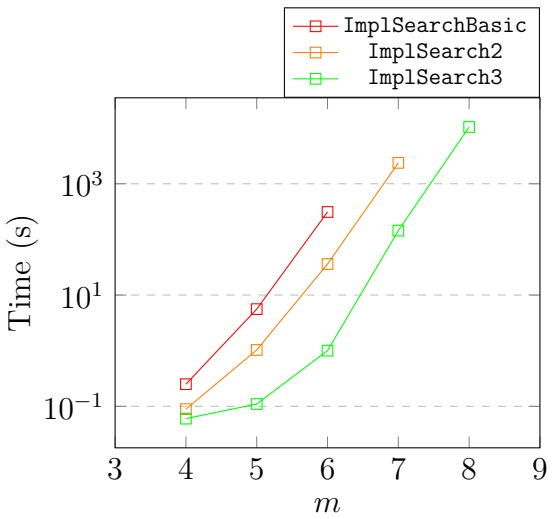
In this section, the implementation of `Search` and its improvements are benchmarked.

7.2.1 Improvements

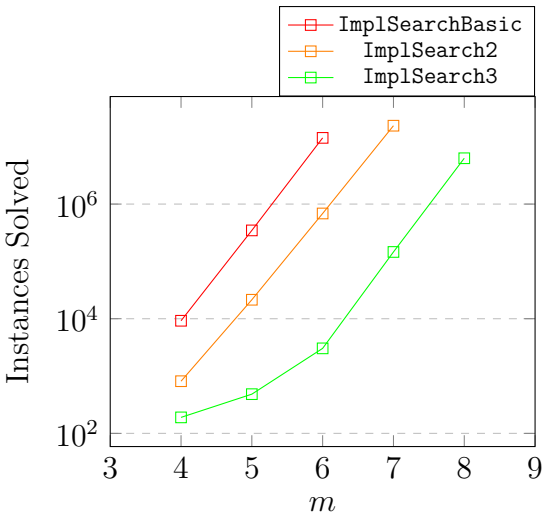
Here we present the results of the improvements to the basic approach of `Search`. The implementation of the basic approach has the addition of visited instances (Section 5.2) because otherwise the search runs out of memory already for $m = 4$. Furthermore, all implementations use the `ImplSolveBest` implementation for solving instances. The following implementations of `Search` are compared:

- `ImplSearchBasic`: Implements the basic approach with the addition of:
  - Visited instances (Section 5.2).
- `ImplSearch2`: Extension of `ImplSearchBasic` with the addition of:
  - Lower bound instances (Section 5.1).
- `ImplSearch3`: Extension of `ImplSearch2` with the addition of:
  - Maximal infeasible instances (Section 5.3).

The benchmarks have been executed on a shared resource with a Intel Xeon Gold 6126 (2.6 GHz) processor and 768 GB RAM. Figure 7.1 presents the execution time and number of solved instances for the three implementations.



(a) Execution time of the search.

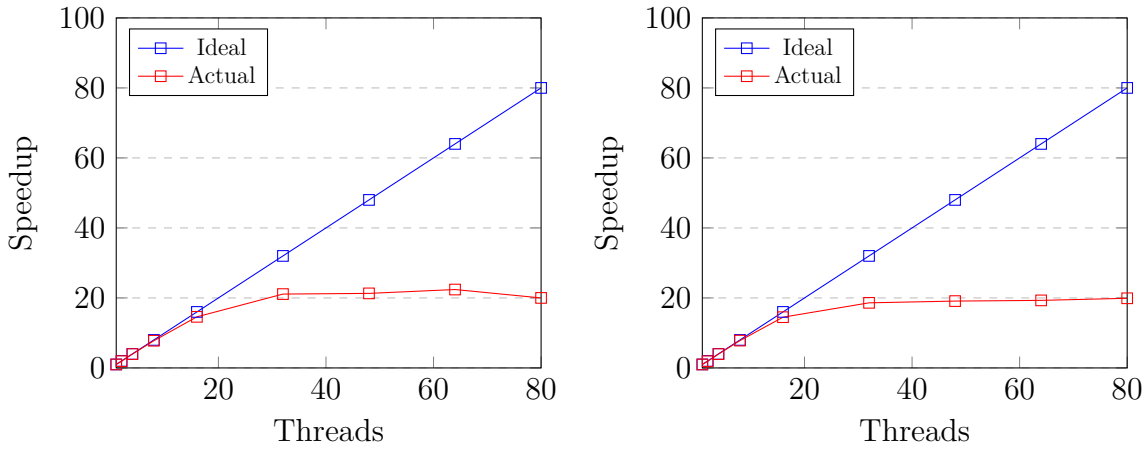


(b) Number of instances solved during the search.

Figure 7.1: Benchmark of different `Search` implementations with a roof value of $r = 2m$.

7.2.2 Parallel Search

`ImplSearchBest` is a parallel version of `ImplSearch3`. The speedup of `ImplSearchBest` has been benchmarked on a shared resource containing two compute nodes, each with a AMD EPYC 7451 (2.9 GHz) processor with 24 cores and 48 threads, and 512 GB RAM. Figure 7.2 shows the results for $m = 7$ and $m = 8$. Moreover, the ideal speedup is also shown.

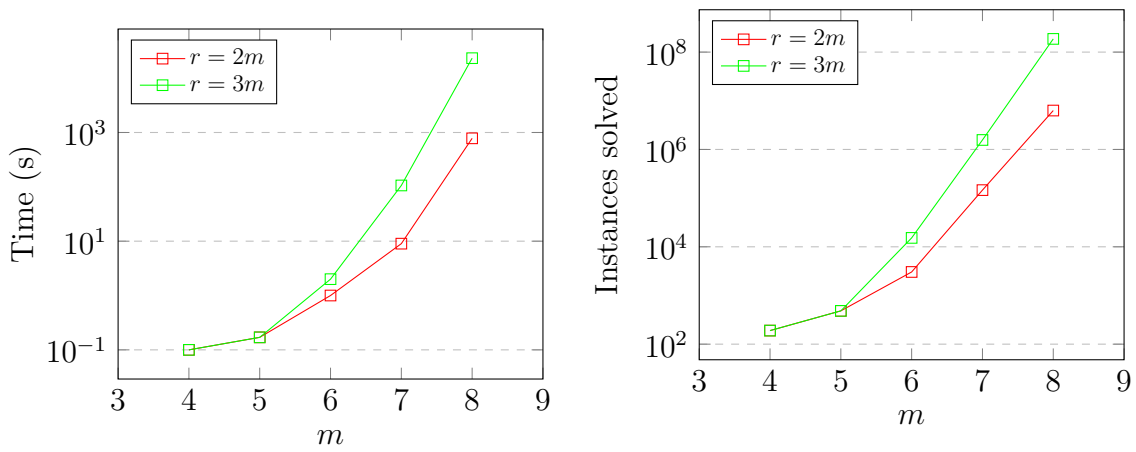


(a) $m = 7$ **(b)** $m = 8$

Figure 7.2: The speedup of `ImplSearchBest` using different number of threads and $r = 2m$.

7.2.3 Increasing the roof value in `ImplSearchBest`

Figure 7.3 shows how the execution time and the number of instances of `ImplSearchBest` scales with increasing m for varying roof values r.



(a) Execution time of the search.

(b) Number of instances solved during the search.

Figure 7.3: Benchmarking roof values using `ImplSearchBest` with 64 threads.

7.3 Critical Instances

Critical instances up to $m = 9$ have been successfully identified using `ImplSearchBest` and varying roof values that are executable within a reasonable time limit. The critical instances are available in Appendix A. Figure 7.4 shows the number of critical instances found for different roof values.

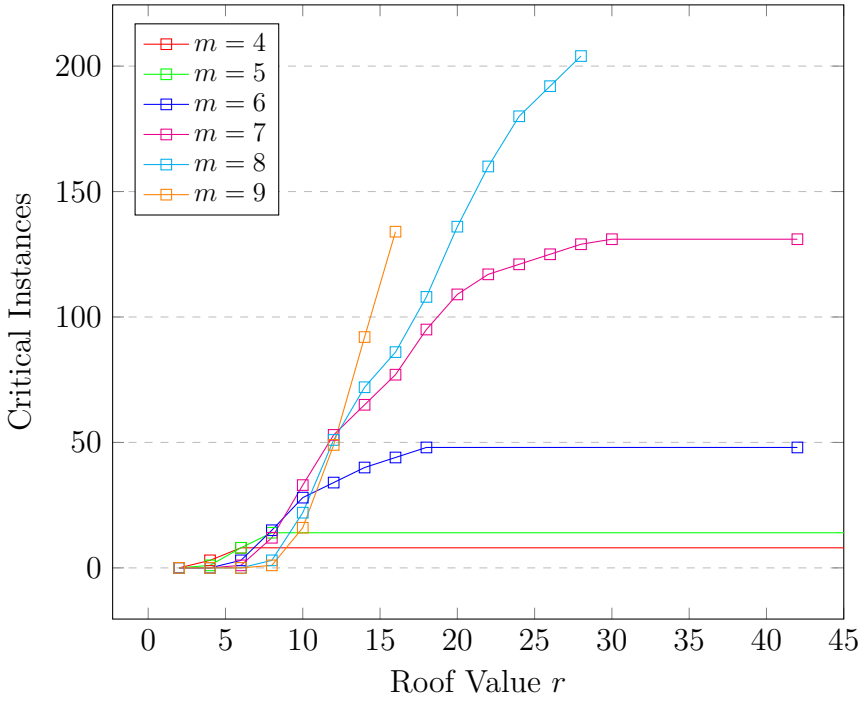


Figure 7.4: The number of critical instances found for different roof values. For $m \geq 6$, larger roof values are omitted because of the execution time exploding.

8

Discussion

The results presented in Chapter 7 are discussed and interpreted in this chapter while highlighting the current strengths and weaknesses of Patsch.

8.1 Results for `Solve`

This section discusses and interprets the results from benchmarking `Solve`. First, the improvements made to `Solve` are discussed in Section 8.1.1. Finally, the sections 8.1.2 and 8.1.3 discusses the difficulty of solving instances of different character.

8.1.1 Improvements

As can be seen in Table 7.1, our best implementation of `Solve` has been significantly improved compared to the basic approach.

Comparing `ImplSolveBasic` with `ImplSolve2`, the implementations differ in the shortest path algorithm used. Compared to the greedy shortest path algorithm, the A* search algorithm additionally needs to store a cost of explored positions and needs a priority queue to maintain positions to expand in sorted order based on their cost [15]. Note that insertion and removal of elements in a priority queue, containing n elements, is a $\mathcal{O}(\log n)$ time operation [16]. Looking at the execution time of both implementations, there is no significant difference, except for some small variations that may occur from, e.g., garbage collection or the fact that the benchmarking was done on a shared resource. We believe the reason that the difference is insignificant is that the position graph is rather small for these instances. Because of this, both algorithms find shortest paths just as quickly. Furthermore, distances are cached and there are at most $\mathcal{O}(m^3)$ distances to cache. The instances solved has $m = 8$, which means that the number of shortest path computations is small and is an insignificant proportion of the running time considering that millions of paths are validated. Because of the size of the graph and cached distances, the difference in time and space complexity of the two algorithms does not show. Nonetheless, we chose to keep using the greedy algorithm as it is simpler, theoretically better in terms of time and space complexity and has been proved to be optimal, see Section 4.2.2.

The improvements added in `ImplSolve3` mainly reduce the number of initial paths to extend in `Solve`, which in turn reduces the number of validated paths. Moreover, `ImplSolve3` has a reduced set of positions in the valid position graph, which also reduces the number of validated paths. As can be seen in Table 7.1, the execution time is correlated to the number of validated paths. With these improvements, the number of validated paths can at most be reduced by a factor linear in m.

The improvement of redundant paths added in `ImplSolve4` reduces the number of validated paths by an exponential factor since each discarded redundant path discards exponentially many extended paths. Hence, the execution time is vastly improved as expected. Using the two redundant paths, Hourglass and Parallelogram, one can reduce the execution time even further.

Finally, `ImplSolveBest` uses babysitting, which turns out to be extremely powerful. The main reason is that exponentially many paths are ignored when finding the two valid babysitting wildcard paths, see Section 4.1.5. If any of them is not found, the instance is quickly deemed infeasible and further valid paths do not need to be extended. Hence, the number of validated paths is extremely small for I in Table 7.1. Furthermore, if both wildcard paths are found, `Solve` uses them as initial valid paths to extend, i.e., ignoring the exponentially many paths of length up to the length of the wildcard paths used. Note that this improvement makes the previous improvements affecting the number of initial paths obsolete. It is also worth noting that infeasible instances may have valid wildcard paths, meaning that valid paths need to be further extended to deem the instances infeasible. For example, the infeasible instance

$$(6, 14, 5, 4, 7, 12, 6, 12)$$

has valid wildcard paths and can not be deemed infeasible quickly using babysitting. Interestingly, a large majority of such infeasible instances seem to have equal $[a, b]$, which tend to be a short interval in the middle of the line, e.g., $[3, 3]$ for $m = 7$ or $[3, 4]$ for $m = 8$. This could be worth investigating further.

The improvement of look-ahead (Section 4.4) is not used in any of the implementations that we compare. The reason is that look-ahead only has an effect on specific types of feasible instances while also adding an additional cost of validating the look-ahead path when solving any instance. As can be seen in Table 7.2, most of the instances solved in `Search` are infeasible, which look-ahead is not applicable to, and the fraction between number of infeasible and feasible instances increases with m. Therefore, we have chosen not to use look-ahead in our search for critical instances.

The improvement of validating paths in parallel (Section 4.3) is also not used in any of the implementations. The reason is that the sequential `ImplSolveBest` is generally fast and `Search` has instead been parallelised, which means that parallelising `Solve` would not yield any further improvements since it is used within `Search`.

8.1.2 Solving Instances in Search

Table 7.2 shows that, generally, solving instances generated in `Search` for $m \leq 8$ is fast. The average execution time is up to a few milliseconds and the standard deviations indicate that the variation in execution time is low, which means that the majority of instances are easy to solve. The maximum execution time is only 138 milliseconds, which is fast. Furthermore, the average execution time does not seem to increase by much for increasing m but the average number of validated paths is increasing at a faster pace. The maximum number of validated paths for an instance is close to the order of 10^5, which is a large number of paths. For larger m, this will most likely increase and the execution time become slower so decreasing the number of paths even further is an important and non-trivial task. However, the average is significantly less than the maximum and the standard deviation is rather low, meaning that for the majority of instances, the number of validated paths is currently not a problem.

8.1.3 Solving Random Instances

Looking at the random infeasible instances in Table 7.3, one can clearly note that these are solved extremely fast. Compared to the infeasible instances solved in `Search` (Table 7.2), the 1000 random infeasible instances are solved faster on average. This shows that, for an instance, the character of the waiting times affects the difficulty of solving the instance. The infeasible instances solved in `Search` generates instances upwards with waiting times in the range $[1, 2m]$, while the random instances have waiting times that are randomly chosen in the range $[1, 5m]$, which most likely gives the instances different character.

Solving random feasible instances with rather large waiting times is notably slower than solving random infeasible instances, see Table 7.3 and 7.4. This is simply because feasible instances need to extend paths up to the length of a solution cycle, starting from babysitting wildcard paths, and when waiting times are large, many valid paths may exist. Furthermore, extending paths is done breadth-first which leads to exponentially many paths being validated before finding a long solution cycle, even though the solution cycle might have a simple form. Also, some infeasible instances may be quickly deemed infeasible using babysitting, while we currently do not know of a similar approach for quickly deeming feasible instances as feasible.

8.2 Results for `Search`

This section discusses and interprets the results from benchmarking `Search`. First, the improvements made to `Search` are discussed in Section 8.2.1. Second, in Section 8.2.2 the parallel version of `Search` and its level of parallelism is discussed. Finally, Section 8.2.3 discusses the impact of increasing the roof value in `Search`.

8.2.1 Improvements

The execution time of the search implementations is strictly correlated to the number of instances solved, as can be seen in Figure 7.1. The improvements made to the basic approach of `Search` has made a big impact on reducing the number of instances solved. The improvement of lower bound instances in `ImplSearch2` has resulted in an expected, considerable reduction in the number of instances solved compared to `ImplSearchBasic` when increasing m. Compared to `ImplSearch2`, the implementation `ImplSearch3` has the addition of a pre-processing step before searching, i.e., generating maximal infeasible instances. This improvement is compelling, as it solves some instances before-hand to extensively reduce the total number of instances solved in the search. This can be seen in Figure 7.1b, which for `ImplSearch3` includes instances solved when generating maximal infeasible instances. A concluding remark is that, while the improvements have made a significant impact, `ImplSearch3` still has a rapid growth in execution time and number of instances solved when increasing m.

8.2.2 Parallel Search

Analysing Figure 7.2, it is apparent that the maximum speedup seems to be around 20 for both $m = 7$ and $m = 8$. From empirical profiling of `ImplSearch3`, the proportion of the program that may be parallelized is roughly 98.6% and 99.9% for $m = 7$ and $m = 8$, respectively. Using these proportions, Amdahl's law gives an attainable speedup of roughly 72 and 1000, respectively [17]. For 64 threads, which on our hardware gave the best speedup, Amdahl's law yields a theoretical speedup of 34 and 60, respectively. We believe the difference between the actual speedup of 20 and the theoretical speedups depends on a few factors. Amdahl's law assumes that the parallel part of the program can be perfectly parallelized [17]. However, for the implementation of parallel programs, many design decisions affect the execution time. For example, synchronization, load balancing and the utilization of hardware resources [17]. In our implementation, we believe one of the main issues is that the execution time of solving instances is diverse, which can be seen in the results of `Solve` in Section 7.1. This may lead to poor load balancing among the threads. Moreover, the parallel search shares a set of visited instances between the threads and it is synchronized to prevent multiple threads from solving the same instance. This will lead to thread contention that may impact the execution time. Note that the set of visited instances does not necessarily need to be synchronized; it does not affect the results of `Search` if multiple threads solve the same instance because of race conditions. However, it is unnecessary to solve the same instance multiple

times and could be more harmful for the execution time than synchronization.

8.2.3 Increasing the roof value in `ImplSearchBest`

As expected, increasing the roof value leads to a rapid growth in the number of instances solved and thus the execution time, see Figure 7.3. Once again we see how the number of instances solved is fairly correlated to the execution time. The curve of the execution time increases slightly faster, which may indicate that the execution time of `Solve` slightly increases with m or external factors, such as garbage collection and executing on a shared resource, may have an impact. However, the main factor that impacts the execution time of `Search` seems to be the number of instances solved. It is interesting to note that for $m = 4$ and $m = 5$, the results are equivalent regardless of r. This is due to the improvement of maximal infeasible instances; infeasible instances generated in the search are all comparable to some maximal infeasible instance, meaning that infeasible instances are not incremented all the way up to r. This is not the case for $m > 5$ since there are not enough maximal infeasible instances generated. An interesting observation is that, for $m = 4$ and $m = 5$, the results are equivalent also for $r = \infty$, or in Java, the maximum integer value $2^{31} - 1$ [18]. Thus, a conjecture that the execution time of generating maximal infeasible instances seems to only depend on m and not r. However, this requires further investigation.

8.3 Critical Instances

As mentioned in Section 7.3, critical instances have been found up to $m = 9$. It is worth noting that PATSCH may be used to find critical instances for greater m as well. However, due to a few factors we have decided not to try searching with larger m. One reason is the accelerated growth in the execution time of `Search` when increasing m and r. Secondly, we have been using a shared resource with a time limit on executing tasks. Finally, due to the time limit of this project, we have not been able to implement checkpoints in PATSCH so that `Search` can be executed at non-consecutive intervals.

For $m \leq 5$, all critical instances have been found; increasing r further does not lead to an increase in execution time. The reason is that the improvement of maximal infeasible instances (Section 5.3) makes the search terminate at some level x, which does not change when increasing r since no infeasible instances are generated with a level greater than x, and thus, the search terminates. For $m > 5$, increasing r leads to more infeasible instances being generated and the search continues to higher levels. Analyzing Figure 7.4, the number of critical instances found for $m \leq 7$ seems to have converged for increasing r, while $m = 8$ seems to be converging. However, it is important to note that it is currently unknown if there exists more critical instances beyond the roof values that were tested for $m \geq 6$. This requires further investigation. Our conjecture is that it is unlikely for further critical instances to appear for $m \leq 7$ and higher roof values. For $m = 8$ it seems that the number of critical instances will converge around 250.

Looking at the waiting times of the critical instances found in more detail, an interesting observation can be noted. A majority of critical instances with a large maximum waiting time has the form

$$(..., 2, 3, ...)$$

or its reversal. For example, the instance

$$(30, 14, 4, 2, 3, 6, 8, 20)$$

which has a long and complicated solution cycle

$$\begin{aligned}&(5,3)(4,2)(3,1)(4,0)(3,1)(4,2)(5,3)(6,4)(6,3)(5,2)(4,3)(5,2)-\\&(6,3)(7,4)(6,3)(5,2)(4,3)(4,2)(3,1)(3,2)(4,3)(5,2)(6,3)(7,4)-\\&(6,3)(5,2)(4,3)(4,2)(5,3)(6,4)(5,3).\end{aligned}$$

It is surprising that this instance is critical when it has such a large waiting time of 30. At first glance, one might think that the instance should still be feasible when decreasing t_0. This is also the case for the larger instance

$$(28, 8, 2, 3, 8, 6, 8, 10, 28)$$

also with a long and complicated solution cycle

$$\begin{aligned}&(3,2)(4,1)(5,2)(6,3)(7,2)(8,3)(7,2)(6,3)(5,2)(4,1)(3,2)(4,1)-\\&(5,2)(6,3)(7,2)(6,2)(5,3)(4,2)(3,1)(2,0)(3,1)(4,2)(5,2)(6,3)-\\&(7,2)(6,3)(5,2)(4,1)(3,2).\end{aligned}$$

Another interesting observation is that some critical instances with x waiting times are extensions of critical instances with $y < x$ waiting times, for example:

$$\begin{aligned}m = 6 &: (8, 6, 4, 3, 3, 4, 6)\\m = 7 &: (8, 6, 4, 3, 3, 4, 6, 8)\end{aligned}$$

or:

$$\begin{aligned}m = 8 &: (10, 8, 6, 6, 4, 4, 6, 6, 8)\\m = 9 &: (10, 8, 6, 6, 4, 4, 6, 6, 8, 10)\end{aligned}$$

or:

$$\begin{aligned}m = 8 &: (12, 10, 8, 6, 4, 4, 4, 6, 8)\\m = 10 &: (12, 10, 8, 6, 4, 4, 4, 6, 8, 10, 12)\end{aligned}$$

This hints on the ability of using previously found critical instances to find new ones for larger m.

Further analysis of the critical instances found has not been pursued in this project. The reason is that the time constraint of this project has put our focus on the search of critical instances, which we have shown to be a highly difficult task.

As a last remark, it was previously believed that all critical instances were found for $m \leq 6$ [5]. However, using PATSCH, further critical instances were found.

9

Conclusion

In this project, two algorithms `Solve` and `Search`, both with a proof of correctness, have been developed, improved and implemented in a Java program that we call PATSCH in order to find critical instances of the integer version of the patrolling problem (IntPUF). We may conclude that our best implementation of `Solve` is generally fast at solving instances up to $m = 10$, but suffer from many paths being validated for exceptional instances. Moreover, our best implementation of `Search` works well for rather small m and r. It is highly parallel and benefits from the use of multiple threads. However, the biggest issue for `Search` is the enormous amount of instances solved.

This project has demonstrated that the search for critical instances is not an easy task but given the correctness and results of PATSCH, it may act as a foundation for further research. It is worth noting that solving instances of IntPUF using PATSCH may currently be the best alternative, compared to using found critical instances, since `Solve` has been easier to realize than `Search`. To conclude this project, the research questions are answered in Section 9.1. Finally, Section 9.2 discusses opportunities for improvement for both `Solve` and `Search`.

9.1 Research Questions

This section is aimed for answering the research questions in Section 1.5.

1. **Can we collect all critical instances and their solutions for $m = 7$, $m = 8$, etc (m as large as possible)?**

   First of all, using PATSCH, the discovery was made that more critical instances exist for $m \leq 6$ than was previously stated [5]. For $m \leq 5$, all critical instances have been found. For $6 \leq m \leq 9$, it is currently unclear if all critical instances have been found due to `Search` being limited by the roof value. However, our conjecture is that all critical instances for $m \leq 7$ have been found.

2. **Can we display solutions to critical instances in a comprehensible way for further research?**

   As the task of searching for critical instances turned out to be highly intricate and due to time limits, the focus has not been on visualizing solution cycles. One suggestion contributed by Damaschke [5] is to visualize solution cycles in the position graph by highlighting the edges that are traversed, as done in Figure 3.4 for the solution cycle $(2,0)(3,1)(4,0)(3,1)(2,0)$.

3. **How does the search time scale with m?**

   Currently, the search time of the best implementation `ImplSearchBest` clearly scales rapidly for increasing m and r, see Section 7.2.3. However, the sample size of the benchmarks is too small to accurately claim a time complexity.

4. **For any given m, can we search for all critical instances without limiting the search using an upper bound on the waiting times of an instance, i.e., a roof value?**

   No. This is a non-trivial task that currently has not been solved. The difficulty lies in that one must safely be able to stop the search without missing critical instances. The roof value fulfills this condition if a safe roof value is known, which currently is not the case. Future work regarding this task is further discussed in Section 9.2.2.

9.2 Future Work

This section presents opportunities for improving `Solve` and `Search`, which have not been further investigated due to the time constraint of this project. Finally, the section ends with discussing how the critical instances found in this project may be utilized.

9.2.1 `Solve`

As mentioned in Section 8.1.1 there is an opportunity to investigate the infeasible instances that can not be deemed infeasible quickly using babysitting. There seems to be a pattern that may be utilized to understand why instances are infeasible.

Extending valid paths in `Solve` is done breadth-first. This leads to exponentially many paths being validated, especially for instances with long solution cycles. There could be opportunities for adding some depth-first elements here. One example is look-ahead (Section 4.4), which works well for some feasible instances.

The benchmarking results of `Solve` indicate that infeasible instances are easier than feasible instances to solve on average. Therefore, the main focus for improving `Solve` could lie on reducing the number of validated paths, especially for feasible

instances. A straightforward idea, that builds on ideas presented in this project, is to implement further redundant paths, see Section 4.1.4. The two redundant paths Parallelogram and Hourglass were not implemented; it could be worth investigating if these are easy to implement. Another is to use the idea of wildcard paths that must exist in solution cycles, similar to the idea of babysitting (Section 4.1.5). Also, an idea is to use a stricter but correct validation of paths that may reduce the number of paths that are valid. A final idea, not as obvious, is to be able to use informal decisions when solving instances, such as with good probability guessing whether an instance is feasible or not before extending paths. This would enable different approaches to be used depending on the guess.

9.2.2 `Search`

The main issues with `Search` are that a safe roof value is currently unknown and the total number of instances solved is enormous already for rather small m. A safe roof value needs to be further investigated to ensure that all critical instances have been found using PATSCH. To reduce the total number of instances solved, there are a few ways to go about.

An observation is that, if an instance $I = (t_0, t_1, ..., t_m)$ is infeasible, then the extended instance $I' = (t_0, t_1, ..., t_m, t_x)$ must also be infeasible. This holds no matter the position of t_x in I' since adding a station to the line can never help to make an infeasible instance feasible. Furthermore, some critical instances may be pre-computed as extensions of previously found critical instances, as discussed in Section 8.3. One could therefore use the search results from instances of length $y < x$ when searching for critical instances of length x.

Another approach for reducing the number of instances solved is to make informed increments of waiting times. This means that, if knowledge exists about which waiting times do not need to be incremented, then an extensive amount of instances may be discarded. This is similar to the idea of maximal infeasible instances (Section 5.3) but instead of pre-processing, the knowledge is used during the search. For example, when a path is deemed invalid it is invalid because it failed some condition. This condition may inform on which waiting times do not need to be incremented.

Finally, an idea could be to change the way of searching for critical instances. Generating instances in order to find critical instances might not be the best approach. Instead, there may be a possibility of generating cycles in the valid position graph and from these, generate a minimal instance such that the cycle is valid.

9.2.3 Critical Instances

The critical instances and corresponding solution cycles found in this project should be analyzed in order to find interesting patterns and understand their structure. For example, as discussed in Section 8.3, a majority of critical instances containing a large waiting time has a special form. From knowing patterns about the critical instances, it may be possible to develop an approach for finding critical instances

more effectively. The same goes for their solution cycles; there may be typical paths that appear frequently and these could be utilized to effectively create solution cycles.

Bibliography

[1] P. Damaschke, "Two robots patrolling on a line: Integer version and approximability," in *IWOCA 2020* (L. Gąsieniec, R. Klasing, and T. Radzik, eds.), vol. 12126 of *LNCS*, (Cham), pp. 211–223, Springer International Publishing, 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-48966-3_16.

[2] J. Czyzowicz, L. Gąsieniec, A. Kosowski, and E. Kranakis, "Boundary patrolling by mobile agents with distinct maximal speeds," in *ESA 2011* (C. Demetrescu and M. M. Halldórsson, eds.), vol. 6942 of *LNCS*, (Berlin, Heidelberg), pp. 701–712, Springer Berlin Heidelberg, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-23719-5_59.

[3] R. Holte, L. Rosier, I. Tulchinsky, and D. Varvel, "Pinwheel scheduling with two distinct numbers," *Theoretical Computer Science*, vol. 100, no. 1, pp. 105–135, 1992. [Online]. Available: https://doi.org/10.1016/0304-3975(92)90365-M.

[4] H. Chuangpishit, J. Czyzowicz, L. Gąsieniec, K. Georgiou, T. Jurdziński, and E. Kranakis, "Patrolling a path connecting a set of points with unbalanced frequencies of visits," in *SOFSEM 2018* (A. M. Tjoa, L. Bellatreche, S. Biffl, J. van Leeuwen, and J. Wiedermann, eds.), vol. 10706 of *LNCS*, (Cham), pp. 367–380, Springer International Publishing, 2018. [Online]. Available: https://doi.org/10.1007/978-3-319-73117-9_26.

[5] P. Damaschke private communication, November 2020.

[6] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*. United States of America: Elsevier Science Publishing Co., Inc., 5th ed., 1982.

[7] Oracle, "HashMap (Java SE 11 & JDK 11)." https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html. Accessed: 2021-04-29.

[8] Oracle, "HashSet (Java SE 11 & JDK 11)." https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashSet.html. Accessed:

2021-04-29.

[9] J. Bloch, *Effective Java: best practices for the Java Platform*. United States of America: Pearson Education, 2018.

[10] Oracle, "Object (Java SE 11 & JDK 11)." `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)`. Accessed: 2021-05-07.

[11] Oracle, "Executors." `https://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html`. Accessed: 2021-05-21.

[12] Oracle, "Executor Interfaces." `https://docs.oracle.com/javase/tutorial/essential/concurrency/exinter.html`. Accessed: 2021-05-21.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Singleton," in *Design Patterns: Elements of Reusable Object-Oriented Software*, pp. 127–134, Addison-Wesley, 1995.

[14] Oracle, "Executors (Java SE 11 & JDK 11)." `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executors.html#newWorkStealingPool(int)`. Accessed: 2021-05-21.

[15] S. Russel and P. Norvig, "Informed (heuristic) search strategies," in *Artificial Intelligence: A Modern Approach*, ch. 3.5, Harlow, England: Pearson Education, 3rd ed., 2016.

[16] J. Kleinberg and E. Tardos, "A more complex data structure: Priority queues," in *Algorithm Design*, ch. 2.5, Harlow, England: Pearson Education, 1st ed., 2013.

[17] T. Rauber and G. Rünger, *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2nd ed., 2013.

[18] Oracle, "Integer (Java SE 11 & JDK 11)." `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Integer.html#MAX_VALUE`. Accessed: 2021-05-25.

A

Critical Instances

This appendix contains all critical instances and their solution cycles for successful m and r.

A.1 $m = 4$

For $m = 4$ and $r = \infty$, there are 8 critical instances. See table A.1 for all critical instances and their solution cycles.

Table A.1: All 8 critical instances and their solution cycles for $m = 4$.

Instance	**Solution Cycle**
[6,4,4,1,6]	(4,3)(3,2)(3,1)(3,0)(3,1)(3,2)(4,3)
[6,1,4,4,6]	(4,1)(3,1)(2,1)(1,0)(2,1)(3,1)(4,1)
[2,2,4,2,4]	(2,0)(3,1)(4,0)(3,1)(2,0)
[1,6,4,4,6]	(1,0)(2,0)(3,0)(4,0)(3,0)(2,0)(1,0)
[4,2,2,2,4]	(4,2)(3,1)(2,0)(3,1)(4,2)
[4,2,4,2,2]	(3,0)(4,1)(3,2)(4,1)(3,0)
[6,4,4,6,1]	(4,0)(4,1)(4,2)(4,3)(4,2)(4,1)(4,0)
[6,4,1,4,6]	(4,2)(3,2)(2,1)(2,0)(2,1)(3,2)(4,2)

A.2 $m = 5$

For $m = 5$ and $r = \infty$, there are 14 critical instances. See table A.2 for all critical instances and their solution cycles.

Table A.2: All 14 critical instances and their solution cycles for $m = 5$.

Instance	**Solution Cycle**
[6,2,2,4,4,6]	(5,1)(4,2)(3,1)(2,0)(3,1)(4,2)(5,1)
[1,8,6,4,6,8]	(1,0)(2,0)(3,0)(4,0)(5,0)(4,0)(3,0)(2,0)(1,0)
[8,6,1,4,6,8]	(5,2)(4,2)(3,2)(2,1)(2,0)(2,1)(3,2)(4,2)(5,2)
[8,1,6,4,6,8]	(5,1)(4,1)(3,1)(2,1)(1,0)(2,1)(3,1)(4,1)(5,1)
[8,6,4,6,1,8]	(5,4)(4,3)(4,2)(4,1)(4,0)(4,1)(4,2)(4,3)(5,4)
[2,2,6,4,4,6]	(2,0)(3,1)(4,0)(5,1)(4,0)(3,1)(2,0)
[4,2,3,4,4,6]	(5,2)(4,1)(3,0)(2,1)(3,0)(4,1)(5,2)
[4,2,4,4,2,4]	(3,0)(4,1)(5,2)(4,1)(3,0)
[8,6,4,6,8,1]	(5,0)(5,1)(5,2)(5,3)(5,4)(5,3)(5,2)(5,1)(5,0)
[6,4,4,2,2,6]	(5,3)(4,2)(3,1)(4,0)(3,1)(4,2)(5,3)
[6,4,4,6,2,2]	(4,0)(5,1)(4,2)(5,3)(4,2)(5,1)(4,0)
[6,4,2,2,4,6]	(5,3)(4,2)(3,1)(2,0)(3,1)(4,2)(5,3)
[6,4,4,3,2,4]	(5,2)(4,1)(3,0)(4,1)(5,2)(4,3)(5,2)
[8,6,4,1,6,8]	(5,3)(4,3)(3,2)(3,1)(3,0)(3,1)(3,2)(4,3)(5,3)

A.3 $m = 6$

For $m = 6$ and $r = 7m$, there are 48 critical instances[1].

A.4 $m = 7$

For $m = 7$ and $r = 6m$, there are 131 critical instances[1].

A.5 $m = 8$

For $m = 8$ and $r = 28$, there are 204 critical instances[1].

A.6 $m = 9$

For $m = 9$ and $r = 16$, there are 134 critical instances[1].

[1] Available at: `https://github.com/radjavi/patsch/tree/main/critical-instances`