Implementing and Evaluating Tarski Fixed Point Algorithms to Decide the ARRIVAL Graph Game

Pavan Chhalani

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

The ARRIVAL problem, a zero-player game on a switch graph model, has recently attracted interest due to its complexity status. Despite its apparent simple nature, no polynomial-time algorithm has been found for the problem yet, providing motivation for study in this field. In this thesis, we focus on the implementation and conducting experiments of various algorithms for the ARRIVAL problem, with a particular focus on the application of Tarski fixed point algorithms in solving the ARRIVAL problem.

We provide some background on switch graphs and their applications, formally define the ARRIVAL problem and give some background on it, discuss its complexity status, and review recent algorithmic advancements on this problem. Our main contribution lies in the implementation and experimental analysis of different algorithms for the ARRIVAL problem, including the subexponential algorithm by [Gärtner et al., 2021], the algorithm by [Fearnley et al., 2021b], and the iterative fixed point algorithm from [Etessami et al., 2019]. We develop an instance-based approach for generating various types of ARRIVAL graphs with different properties, like clustering and branched structures. Our practical runtimes and experimental results align with the runtime complexities mentioned in the literature.

Finally, we give a short proof for an interesting theorem about ARRIVAL graphs with two branches, viewing the problem as a comparison of two integer numbers.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Pavan Chhalani*)

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Prof. Kousha Etessami, for his invaluable guidance and support during my research and for helping me understand the complexities of ARRIVAL graphs and algorithms. I am also deeply grateful to my family for their patience and encouragement.

Table of Contents

Chapter 1

Introduction

1.1 Motivation to solve the ARRIVAL graph problem

Switch graphs also known as switching graphs have been studied previously in various contexts, but mostly to study combinatorial problems. In the literature, switching graphs are often defined as directed graphs with additional structures called switches, which are placed on some or all of the vertices [Jurdziński, 1998, Ploeger, 2009]. These switches introduce a dynamic element to the graph, allowing for the selection of specific outgoing edges based on the switch settings.

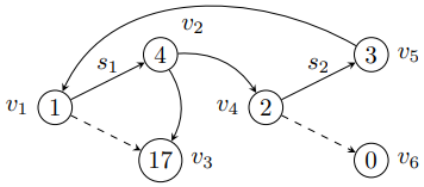


Figure 1.1: A directed labelled switching graph [Ploeger, 2009]

For instance, Groote and Ploeger [2009] employed labelled directed switching graphs (Figure 1.1) as a tool for tackling Boolean equation systems (BES) for model checking and parity game decision problems. By leveraging the switch settings, they develop techniques to solve these problems efficiently. Katz et al. [2012] did an algorithmic study on switch graphs, they derive various results on switch graph with different graph properties. We encourage our readers to review Katz et al. [2012], Ploeger [2009] for a comprehensive study of the existing research done on switch graphs.

A *zero-player* game on the switch graph model was introduced by Dohrau et al. [2017], whereby *zero-player* we mean that the initial state of vertices and switches uniquely determines the result. The question of the ARRIVAL problem is closely related to cellular automata, and Dohrau et al. [2017] mention that it was motivated by the online game *Looping Piggy* [Scr] that Bernd *Gärtner* made for a Swiss-based educational initiative *Kinderlabor* [kin, 2022].

We define the switch graph of the ARRIVAL problem as a directed graph, with all the nodes having an out-degree of 2, i.e., all the nodes in the graphs have an ”even” and an ”odd” successor. The train on this switch graph can be thought as a deterministic simulation of a random walk. Pseudorandom walks like this have been studied under various names rotor-router walks [Holroyd and Propp, 2010], Propp Machines [Cooper et al., 2007], and Eulerian walks [Priezzhev et al., 1996]. In the later parts of this thesis, we formally define our problem and its variants.

Talking about its complexity status, ARRIVAL was shown to be in $NP \cup coNP$ by Dohrau et al. [2017]. It is arguably the simplest problem in $NP \cup coNP$, which is not in P. In contrast, other well-known graph games such as simple stochastic games , mean-payoff games Zwick and Paterson [1996], and parity games Jurdziński [1998], which are also in NP ∩ coNP but not known to be in P, involve two players (or controllers). Considering this, one might anticipate that a zero-player game like Arrival would be significantly simpler to solve. However, despite efforts, no polynomial-time algorithm for Arrival has been discovered yet.

Fearnley et al. [2021a] shows that the lower bound complexity of ARRIVAL is in NL-hard. Any problem in this class is at least as hard as a problem which can be solved by a nondeterministic Turing machine using logarithmic space (i.e., the amount of memory or space the machine uses is logarithmic in the size of the input). Complexity classes of other variants of ARRIVAL have been studied, and a search version of Arrival by Karthik [2017] was shown to be in UniqueEOPL by Fearnley et al. [2021a].

To decide the problem, an obvious approach is to simulate the run. For a graph with n vertices, simulating the train (algorithm 1) can take at most $O(n2^n)$ steps. Recently, Gärtner et al. [2021] made significant progress on the upper bound complexity and provided an algorithm which can decide the ARRIVAL problem in $2^{O(\sqrt{n}\log n)}$ time.

In this thesis, we try to further explore the ARRIVAL problem, we investigate the application of Tarski fixed point algorithms to decide the outcome of Arrival graphs. Tarski’s theorem guarantees the existence of a fixed point of a montone function on a complete lattice. By Reducing the ARRIVAL problem into a monotone function, we can apply Tarski fixed point algorithms on it. This can lead to efficient algorithms for deciding where the train would end up in the network.

The use of Tarski fixed point algorithms has certain potential benefits. First of all, these algorithms provide a systematic approach for deciding the ARRIVAL problem. The algorithms can be designed to efficiently explore the state space of the Arrival graph, avoiding unnecessary computations and reducing the overall runtime.

1.2 Summary of our Contribution

In this thesis, we have made the following contributions to the study of the ARRIVAL problem and its related algorithms:

- We have implemented the generation of different types of ARRIVAL graphs with certain properties:

1. We adopted an instance-based approach in generating a standard version of ARRIVAL graphs, allowing for a flexible generation and manipulation of the graphs. To avoid end-less loops, we implement grouping of unreachable nodes.
2. Each instance is reduced to a monotone function, which encodes the number of visits to each vertex.
3. We increase the difficulty of the problem by increasing the length of the direct path from origin to destination, e.g., clustering together different nodes.
4. We experiment with Bit-counter like graphs, which can take exponential time.

- Regarding the Implementation and experimentations of the algorithms:
  1. we implement and analyse the performance of various algorithms for the ARRIVAL problem.
  2. we measure different algorithms against the iterative algorithm to find fixed points, starting from the least element in the lattice and iteratively applying the function.
  3. We implement and analyse algorithms from [Gärtner et al., 2021], and Fearnley et al. [2021b]
  4. We have provided detailed pseudocode and explanations for each algorithm, enabling readers to understand and reproduce our results.
  5. We investigate the impact of different graph properties, such as size and diameter, on the algorithms' behaviour and performance.
- We give and prove an interesting theorem about the Arrival graph with two branches. It views the problem as a comparison of two integer numbers.

1.3 Road map of the thesis

In Chapter 2, we discuss the fundamental concepts for understanding the thesis. We take a glimpse over the vast landscape of Tarski fixed points and Monotone functions and their connections to our investigations. We examine the existing algorithms that we intend to implement and experiment with.

Moving forward, in Chapter 3, we reduce the ARRIVAL problem to a fixed-point search problem. We present formal definitions, introduce the key theorems that have inspired our approach, and discuss the algorithmic architecture that navigates the problem space. The reader will gain insight into the workings of key algorithms.

We discuss our implementation choices in chapter 4, starting with generating different graphs for the ARRIVAL problem. We give the choices made for each algorithm and the python class/functions associated with sub-parts of the algorithms.

In Chapter 5, we focus primarily on the experimental evaluation of our algorithms on different types of Arrival graphs. We try to observe the behaviour of these algorithms in practice and to determine whether the use of fixed point algorithms provides any benefits in deciding the ARRIVAL problem.

Finally, in Chapter 6, we conclude and summarise the lessons we learned. We mentioned the difficulties faced during the research and implementation of our work. At Last, we mention the limitations of our work and mention some future works.

Chapter 2

Background

Our Project aims to implement and test algorithms for deciding on a specific problem called the ARRIVAL graph problem using Tarski fixed point algorithms. The zero-player ARRIVAL problem has a starting vertex and two destinations, with a train(token) running on a switch network. We discuss the problem and its graph structure later in this chapter. To help the readers grasp our objective and locate how and where the discussed algorithms fit in the vast landscape, we use this chapter to characterise and explain various parts of our approach. We also present the state-of-the-art research in this field. By understanding the simple nature of the problem, readers can understand our interest in finding a polynomial time algorithm for this problem. We will finish this chapter by pointing at some related works done around similar problems.

2.1 Preliminaries

In this section, we want to help the readers grasp the notation and definitions of certain terms in the thesis. So, unless mentioned explicitly, the following are standards for this thesis.

Lattices: In this thesis, we work with finite complete lattices, where each point in the grid has n-dimensions, so the lattice contains every point $x \in \mathbb{N}^n$ such that $1 \leq x_i \leq N$, N is the width of all the dimensions.

Ordering: Throughout we use $\leq$ to compare two points on a lattice, it gives the natural partial ordering over this lattice, and $x \leq y$ if and only if $\forall i \leq n\ x_i \leq y_i$.

Inflows and Outflows of a vertex in ARRIVAL: for some vertex $v \in V$, we follow notations used by previous works, we denote $E^-(v)$ as the set of incoming edges of v and $E^+(v)$ as the set of outgoing edges of v, where E is Edges of the graph. Furthermore, a function $x : E \to N_0$, the outflow of x at v is $x^+(v) := \sum_{e \in E^+(v)} x(e)$ and $x^-(v) := \sum_{e \in E^-(v)} x(e)$ to denote the inflow of x at v. Let Y be an artificial vertex used to avoid special treatment of the origin vertex, this vertex essentially points to the origin to maintain the flow conservation at each vertex $\neq Y$.

2.2 Defining the TARSKI Problem

Tarski's fixed point theorem states that given a complete lattice $U = (L, \leq)$, where $\leq$ is the partial order used to compare lattice elements. A monotone function $f : L \to L$ has at least one fixed point, i.e., $\exists x \in L. f(x) = x$. Recently, there has been interest in finding efficient fixed point algorithms due to its application in Nash equilibrium of supermodular games, stochastic games, and its use in answering the ARRIVAL problem. TARSKI total search problem, as mentioned in Fearnley et al. [2021b], is defined -

Definition 1 ([Fearnley et al., 2021b]) ***Input****: Given a Lattice L, and a function f : $L \to L$, then find one of the following,*

- *(T1) a point $x \in L$, such that $f(x) = x$*
- *(T2) Two points $x, y \in L$, such that $x \leq y$ and $f(x) \not\leq f(y)$*

Theorem 1 ([Tarski, 1955]) *Every order-preserving function on a complete lattice has a fixed point.*

The solutions in Definition 1 are of two types, T1 are the fixed point of function f, whereas T2 type is an witness that function f is not order-preserving. In our work, we assume that our function is order-preserving and that no type T2 solution exists in the lattice L. This simplifies some of the algorithms we use in chapter 4.

2.2.1 Prior work on finding Tarski Fixed Points

The use of monotonicity and Tarski's theorem in existence proofs of equilibria is widespread in economics, while Tarski's theorem is also often used for similar purposes in the context of verification [Etessami et al. 2021]. However, there has been relatively little analysis of the complexity of finding the fixed points and equilibria guaranteed by this result.

Dang et al. considered the k-dimensional lattice defined by the set of integer points in a k-dimensional box and studied the Oracle query complexity of finding a Tarski fixed point in this setting, i.e., the monotone function is given as a black box. They provided the algorithm that finds a fixed point using $O(\log^k n)$ Oracle queries to the Montone function, where n is the maximum side length of the box, and k is the dimension. Etessami et al. [2019] showed that finding some fixed point requires at least $\log^2 N$ function evaluations already on the 2-dimensional grid, even for randomized algorithms. They also proved a matching $\Omega(\log^2 n)$ lower bound on the query complexity for the k=2 case, hence showing the algorithm of [Dang et al.] is optimal for k=2.

In a surprising result, Fearnley et al. [2021b] refuted the conjecture, providing an algorithm with query complexity $O(\log^2 n)$ for three-dimensional case. They also proved a decomposition theorem that reduces the k-dimensional Tarski problem and gives an improved $O(\log^{\lceil 2k/3 \rceil} n)$ upper bound for all dimensions $k \geq 3$. Most recently, Chen and Li [2022] introduced a new variant of the Tarski problem called "Tarski*" and developed a novel decomposition theorem for this variant. By utilizing Tarski* as a recursive subroutine, they obtained an improved upper bound of $O(\log^{\lceil (k+1)/2 \rceil} n)$ for k-dimensional case.

Etessami et al. [2019] showed that Tarski problem of finding a fixed point, given a monotone function in a boolean circuit, is in class PLS of problems which are solvable by local search and, surprisingly, also in class PPAD. However, finding the greatest or least fixed point guaranteed by Tarski's theorem requires d * N steps, and is NP-hard in the white box model.

2.3 Defining the ARRIVAL Problem

Imagine a train starts from an origin station and traverses through a network of stations, the main goal being to reach a preset target station. The network is designed in a way that each time the train traverses through a station, its switch changes direction immediately after the train passes through. Hence, when the next time the train reaches that station, it will take the other direction, so the directions keep alternating at a switch. Now, the question we are trying to answer is, given the network and origin, will the train ever reach the target destination?

Definition 2 (ARRIVAL) *Problem ARRIVAL is to decide which destination is reached by Run Procedure (1) when it terminates, for a given instance A = (V, o, d, $\bar{d}$, s_{even}, s_{odd}).*

This zero-player problem ARRIVAL, as introduced in Dohrau et al. [2017], runs on a switch graph: directed graph G with vertices having two outgoing edges; the number of visits to a vertex determines the successor of a vertex. In our context, if the train visits a vertex for the first time, the train will be pointed to its even successor; for each subsequent visit, the pointer switches between the odd and even outgoing edges.

In our work, we consider the two-destination version of the problem mentioned in Gärtner et al. [2021] in which we assume that the train will reach one of the two destinations (target destination or sink destination vertex), and we decide on which one. We call such an instance *terminating* as it is guaranteed that either d or $\bar{d}$ is reached.

Formally, two destination instance of ARRIVAL is the 6-tuple $A = \left(V, o, d, \bar{d}, s_{even}, s_{odd}\right)$; where V is the set of vertices, $o \in V$ is the origin vertex, $d, \bar{d} \notin V$ are the destinations, and successor functions $s_{even}, s_{odd} : V \to V \cup \{d, \bar{d}\}$. s_{even} gives the even successor and s_{odd} gives the odd successor. The directed graph of instance A, which connects each vertex $v \in V$ to its even and odd successors, is denoted by G(A).

The most intuitive approach to solve this problem would be to try and simulate the run of the train through the network (Algorithm 1); starting at the origin, we move the train along the switch graph until it reaches d or $\bar{d}$. In the worst case, 1 traverses through an exponential number of edges before reaching any destination. Dohrau et al. [2017], Manuell [2021] constructed a bit-counter version of the ARRIVAL graph in which Algorithm 1 (Run Procedure) traverses through 2^n edges before terminating.

Algorithm 1 Run Procedure [Gärtner et al., 2021]

Input: ARRIVAL instance $A = (V, o, d, \bar{d}, s_{even}, s_{odd})$
Output: final destination reached, d or $\bar{d}$
Let s_0 and s_1 be arrays indexed by vertices containing **even** and **odd** successors
$v \leftarrow o$
$s\_curr \leftarrow$ copy of s_0 ▷ current switches for each vertex
$s\_next \leftarrow$ copy of s_1 ▷ next switch for each vertex
while $v \neq d$ and $v \neq \bar{d}$ **do**
    $w \leftarrow s\_curr[v]$
    $s\_curr[v] \leftarrow s\_next[v]$
    $s\_next[v] \leftarrow w$
    $v \leftarrow w$
end while
return v

The above algorithm essentially translates the problem from one state to another state, whereby state, we mean the current condition of the switches and the location of the train in the graph.

Dohrau et al. [2017] worked on a one-destination version of the Arrival graph, they mentioned that since only $n2^n$ different states can occur, we can be sure that the Algorithm would terminate by either reaching the destinations or repeating any previously visited state, in which case we terminate by stating that it will never reach the destination (we run in an infinite loop). An interesting question here would be: Can we produce a one-destination instance that needs an exponential number of steps to reach the destination? Figure 2.1 (Dohrau et al. [2017]) shows an example of one such instance which reaches the destination but takes an exponential number of steps.

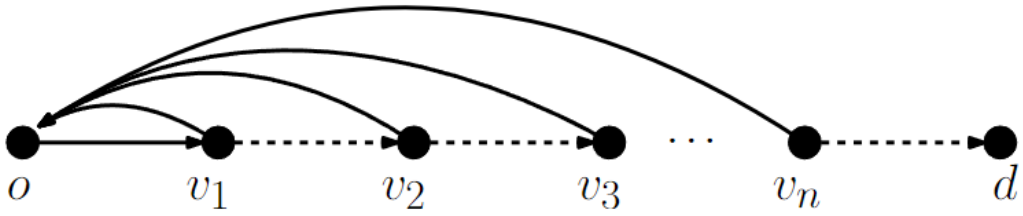


Figure 2.1: Let dashed lines be odd edges and solid lines be even edges. If we encode the current positions of the switches at $v_n, ..., v_1$ with an n-bit binary number (0: even successor is next; 1: odd successor is next), then the run counts from 0 to 2n 1, resets the counter to 0, and terminates. Dohrau et al. [2017]

Gärtner et al. [2021] builds on these results and presents a subexponential time algorithm ($2^{O(\sqrt{n}\log n)}$ runtime) for a slightly different ARRIVAL instance with two destinations (we use the same version) and they decide which destination is reached. In chapter 3, we give the reduction of two-destination terminating instances of the problem to a

monotone function on finite lattices. The question of the ARRIVAL problem is closely related to cellular automata, and Dohrau et al. [2017] mention that it was motivated by the online game *Looping Piggy* (https://scratch.mit.edu/projects/1200078/) that Bernd *Gärtner* made for a Swiss-based educational initiative *Kinderlabor* (http://kinderlabor.ch/).

2.4 Stochastic Games

Before diving into the Arrival problem and related algorithms, we take readers on a small excursion into the topic of SSGs. A simple stochastic game (SSG) involves two players, MAX and MIN, strategizing to move a token through a directed graph from start vextex towards the 'sink' vertices, aiming to reach their designated winning sink. A directed graph $G = (V, V_o, V_1, V_2, \delta)$ whose vertices V include the sinks, and other than that V_o(random),V_1(max),V_2(min). Each player chooses and adheres to a strategy; for a MAX player, the strategy is the mapping from V_1 to V, and for a MIN player it is a mapping from V_2 to V. At start a token is placed on the start vertex; each player's move depends on the vertex type—MAX or MIN vertices require player action, while moves from V_o vertices are random.

There has been a lot of research on algorithms to find value of such games, from randomised algorithms, linear programming or quadratic programming algorithms. Recently, [Etessami et al., 2019] presented the reduction of Condon's and Shapley's SSG to Tarski problem. We can try using algorithms for Tarksi to find the value of SSG instances in polynomial time, given the representation of the game in terms of a monotone function. Although we don't cover this implementation in our thesis we wanted to mention our future interests in comparing the runtimes of [Fearnley et al., 2021b, Chen and Li, 2022] algorithm implementations with the current mathematical programming-based approaches for simple stochastic games.

Chapter 3

Understanding Algorithms Used to decide the ARRIVAL problem

We hope that our readers are now familiar with switch graphs and especially the ARRIVAL problem and our motivation behind trying to decide the ARRIVAL problem. We now want to dive deeper into the algorithms and graph generations we implemented. We begin by exploring the reduction of the ARRIVAL problem to a monotone function, which forms the basis for various algorithmic approaches. Next, we discuss various algorithms to find the fixed points of our monotone function.

3.1 Reduction of ARRIVAL Problem to a monotone function

In this section, we reduce a terminating ARRIVAL instance to a monotone function [Etessami, 2024]. Rendering the deciding of the ARRIVAL problem into a search problem of finding fixed points of the monotone function. Gärtner et al. [2021] proves how such a fixed point can be used to decide the problem.

Lemma 1 *Let A = (V, o, d, $\bar{d}$,s_{even}, s_{odd}) be a terminating ARRIVAL instance, $\mid V \mid = n$. Let v V and suppose that the shortest path from v to a destination in G(A) has length m. Then v is visited (the train is at v) at most 2^m times by Algorithm 1 (Run Procedure).*

We define X_i as the number of times vertex i (station in the train network) was visited. It is easy to understand that in a switching movement, X_i depends on all the X_j where node j has an edge pointing to i. Formally, we define $X_i \; \forall \; i \neq o$ as:

$$X_i = \min\Big(N, \sum_{j \in even_i} \lceil X_j/2 \rceil + \sum_{j \in odd_i} \lfloor X_j/2 \rfloor\Big) \tag{3.1}$$

where, $even_i$ is the set of all the nodes $n \in V$ which have their s_0^{th} edge pointing at node i, similarly odd_i is the set of nodes whose s_1^{th} edge points to node i; and $N = 2^n$ where $n = \mid V \mid$. The origin o is visited at the start of the graph traversal, so we change the above equation for X_o,

$$X_o = \min\Big(N, \sum_{j \in even_o} \lceil X_j/2 \rceil + \sum_{j \in odd_o} \lfloor X_j/2 \rfloor + 1\Big) \tag{3.2}$$

We argue that we can restrict X_i to a finite number N because of the assumption that the ARRIVAL instance is terminating and the following lemma:

Definition 3 *Let A = (V, o, d, $\bar{d}$,s_{even}, s_{odd}) be a terminating ARRIVAL instance, | V | = n. Let $V = \{-1, 0, 1, ..., n-2\}$ we the vertices, where 0 is the origin vertex, -1 is sink node and n-2 is the target node. Let $N = 2^n$ and consider the following function $D : \{0, 1, ..., N\}^n \to \{0, 1, ..., N\}^n$ defined by:*

$$D(w) = \begin{pmatrix} \min\Big(N, \sum_{j \in even_0} \lceil X_j/2 \rceil + \sum_{j \in odd_0} \lfloor X_j/2 \rfloor + 1\Big) \\ \min\Big(N, \sum_{j \in even_1} \lceil X_j/2 \rceil + \sum_{j \in odd_1} \lfloor X_j/2 \rfloor\Big) \\ \vdots \\ \min\Big(N, \sum_{j \in even_n} \lceil X_j/2 \rceil + \sum_{j \in odd_n} \lfloor X_j/2 \rfloor\Big) \end{pmatrix}, \quad w \in \{0, 1, \ldots, N\}^n.$$

then D is an order-preserving function on a finite lattice and thus has a fixed point.

An input to the function D(w) would be an array indexed by vertices V, denoting the number of times each vertex $v \in V$ was visited.

Therefore, we reduce the problem of computation of switching flows to Tarski fixed point calculation. This system of linear equations can be solved using general purpose ILP solvers. Gärtner et al. [2021] provides an efficient method to solve these equations which in the worst case takes $O(p(n)n^{1.633\sqrt{n}})$ for some polynomial p and $|V| = n$. This is still a subexponential time

3.2 Iterative Algorithm

Tarski's theorem on the lattice of fixed points forms the conceptual basis for various algorithms that compute fixed points of functions. When applied to our grid framework, the theorem leads to an explicit algorithm for finding a fixed point of a monotone function f over an d-dimensional grid space. where we consider the grid points as vectors in a discrete space, and f is assumed to be monotone, meaning that if $x \leq y$ (component-wise ordering), then $f(x) \leq f(y)$.

Given this setup, Etessami et al. [2019] mentioned an easy proof of Tarski's theorem on grid framework: the process to find a fixed point can be translated into an algorithm, which in the worst case takes dN iterations to find a fixed point, N is the size of the grid in each dimension, and d is the dimensionality of the space.

3.2.1 Algorithm Overview for Arrival Instance

This section describes how the algorithm in solving the Arrival instances. Definition 3 gives the monotone function $D : \{0, 1, ..., N\}^n \to \{0, 1, ..., N\}^n$ corresponding to an ARRIVAL instance with $|V| = n$. We start with the all-1 vector $\bar{1} = (1, ..., 1)$, which has

an n-dimensional vector where all the coordinates are 1. We then repeatedly apply D to the current vector: $D(\bar{1}), D(D(\bar{1})), ..., D^i(\bar{1})$.

Due to monotonic nature of D, it's guaranteed that $D^i(\bar{1}) \leq D^{i+1}(\bar{1})$ for all $i \geq 0$. This means that at each iteration the components of the vector either increases or stays the same. The sum of components weakly increases at each step. But, the grid is finite i.e., there is a limit to how much the sum can increase.

Therefore, after at most $n2^n$ iterations, a fixed point must be reached. The algorithm stops when $D^i(\bar{1}) = D^{i+1}(\bar{1})$, indicating $D^i(\bar{1})$ is the fixed point.

3.2.2 Using Iterative Algorithm as Benchmark

With the worst case of $O(n2^n)$, it is clear that this algorithm is a benchmark to measure the performance of more sophisticated or domain-specific algorithms, as we will mention later in this chapter. Our main idea is that any algorithm that improves either the worst-case complexity for Arrival or gives superior average-case performance in practical Arrival graph generations is considered a better algorithm.

3.3 Subexponential Algorithm

This section aims to help readers understand the workings of the subexponential algorithm for solving the ARRIVAL problem [Gärtner et al., 2021]; we have already discussed the complexity of the ARRIVAL problem. The idea behind this algorithm is to reduce the problem of computing a switching flow (a certificate for the output of the ARRIVAL instance) to a fixed-point search problem. The algorithm exploits the monotonicity property of the underlying function and applies the theory of Tarski fixed points to efficiently find a fixed point, which corresponds to a switching flow. We will now explain the terminology of this algorithm.

3.3.1 Multi-Run Procedure

The Multi-Run Procedure is a key component of the subexponential algorithm. It is a generalisation of the original simulation of algorithm 1 to simulate the simultaneous movement of multiple trains starting from different vertices. The trains continue to move until they reach either a destination or a vertex in subset S. The size and choice of this subset is something we will discuss later in this thesis.

We start one train from the origin and multiple trains from each vertex in subset S. The movement of trains from vertices in $V \setminus S$ follows the same switching behavior as in the original Run Procedure. The choice of the vertex v and the number of trains τ to move in each iteration is non-deterministic. Different choices may lead to different outcomes, but we show later in this section that the resulting candidate switching flow is unique and minimal.

The efficiency of the Multi-Run Procedure depends on the choice of S and the switching behavior of the ARRIVAL instance. By carefully constructing S as a ϕ-set (Lemma

10), the algorithm ensures that the shortest path from any vertex to $S \cup \{d, \bar{d}\}$ is short, allowing for a subexponential time implementation of the Multi-Run Procedure.

Algorithm 2 Multi-Run Procedure [Gärtner et al., 2021]

Input: Terminating ARRIVAL instance $A = \left(V, o, d, \bar{d}, s_{even}, s_{odd}\right)$ with edges E; $S = \{v_1, v_2, \ldots, v_k\} \subseteq V, w = (w_1, w_2, \ldots, w_k) \in \mathbb{N}_0^k$ (one train starts from Y, and w_i trains start from v_i).
Output: number of trains arriving at $d, \bar{d}$, and in S, respectively
Let t be a zero-initialized array indexed by the vertices of $V \cup \{d, \bar{d}\}$
$t[o] \leftarrow 1 \quad /* \text{ traversal of } (Y, o) */$
for $i = 1, 2, \ldots, k$ **do**
  $t\left[s_{even}(v_i)\right] \leftarrow t\left[s_{even}(v_i)\right] + \lceil w_i/2 \rceil \quad /* \lceil w_i/2 \rceil \text{ traversals of } (v_i, s_{even}(v_i)) */$
  $t\left[s_{odd}(v_i)\right] \leftarrow t\left[s_{odd}(v_i)\right] + \lfloor w_i/2 \rfloor \quad /* \lfloor w_i/2 \rfloor \text{ traversals of } (v_i, s_{odd}(v_i)) */$
Let s_{curr} and s_{next} be arrays indexed by the vertices of $V \setminus S$
for $v \in V \setminus S$ **do**
  $s_{curr}[v] \leftarrow s_{even}(v)$
  $s_{next}[v] \leftarrow s_{odd}(v)$
while $\exists v \in V \setminus S : t[v] > 0$ **do**
  pick $v \in V \setminus S$ such that $t[v] > 0$ and choose $\tau \in \{1, \ldots, t[v]\}$
  $t[v] \leftarrow t[v] - \tau$
  $t\left[s_{curr}(v)\right] \leftarrow t\left[s_{curr}(v)\right] + \lceil \tau/2 \rceil \quad /* \lceil \tau/2 \rceil \text{ traversals of } (v, s_{curr}(v)) */$
  $t\left[s_{next}(v)\right] \leftarrow t\left[s_{next}(v)\right] + \lfloor \tau/2 \rfloor \quad /* \lfloor \tau/2 \rfloor \text{ traversals of } (v, s_{next}(v)) */$
  if τ is odd then
    $swap\left(s_{curr}[v], s_{next}[v]\right)$
return $\left(t[d], t[\bar{d}], t\left[v_1\right], t\left[v_2\right], \ldots, t\left[v_k\right]\right)$

3.3.2 Switching Flows

Definition 4 *(Switching Flow [Dohrau et al., 2017]). Let A = (V, o, d, $\bar{d}$,s_{even}, s_{odd}) be a terminating ARRIVAL instance, $|V| = n$. A function $x : E \to N_0$ is a switching flow for A if,*

$$x^+(Y) = 1,$$
$$x^+(v) - x^-(v) = 0, \quad \textit{for } v \in V \textit{ (flow conservation)}$$
$$x(v, s_{even}(v)) - x(v, s_{odd}(v)) \in \{0, 1\}, \quad \textit{for } v \in V \textit{ (switching behaviour)}$$

moreover, x is called a switching flow to $t \in \{d, \bar{d}\}$ if $x^-(t) = 1$.

A switching flow models the movement of a single train in the ARRIVAL instance. a function is a switching flow if it satisfies all three conditions. The train starts at Y (condition 1), and at each vertex, its inflow equals its outflow (condition 2). The third condition ensures the switching behaviour: the train alternately chooses between the even and odd successors at each vertex.

Theorem 2 (Switching flows are certificates) *[Dohrau et al., 2017]. Let $A = \left(V, o, d, \bar{d}, s_{even}, s_{odd}\right)$ be a terminating ARRIVAL instance, $t \in \{d, \bar{d}\}$. Algorithm 1 (Run Procedure) outputs t if and only if there exists a switching flow to t.*

Theorem 3 (run profile is the minimal switching flow) *[Dohrau et al., 2017]. Let $A = \left(V, o, d, \bar{d}, s_{even}, s_{odd}\right)$ be a terminating ARRIVAL instance with edges E. Let $\hat{x}$ be the run profile of A, meaning that $\hat{x}_e$ counts the number of times edge e is traversed during Algorithm 1 (Run Procedure). Then $\hat{x} \leq x$ for all switching flows x. In particular, $\hat{x}$ is the unique minimizer of the total flow $\sum_{e \in E} x_e$ over all switching flows.*

3.3.3 Candidate Switching Flows

Candidate switching flows relax the concept of switching flows; some of the conditions of the switching flows are relaxed for our version of the ARRIVAL problem.

Definition 5 (Candidate Switching Flow) *[Gärtner et al., 2021] Let $A = (V, o, d, \bar{d}, s_{even}, s_{odd})$ be a terminating ARRIVAL instance with edges E, $S = \{v_1, v_2, \ldots, v_k\} \subseteq V$, $w = (w_1, w_2, \ldots, w_k) \in \mathbb{N}_0^k$. A function $x : E \to \mathbb{N}_0$ is a candidate switching flow for A (w.r.t. S and w) if*

$$x^+(Y) = 1,$$
$$x^+(v) - x^-(v) = 0, v \in V \setminus S \text{ (flow conservation at } V \setminus S)$$
$$x^+(v_i) = w_i, i = 1, 2, \ldots, k, \text{ (outflow w at S)}$$
$$x(v, seven(v)) - x_{(v, s_{odd}(v))} \in 0, 1, v \in V(switchingbehavior).$$

The key difference between switching flows and candidate switching flows is that, candidate switching flows relax the flow conservation constraint at the vertices in S. Instead, the outflows from these vertices are fixed to the values specified by the vector w.

The motivation behind candidate switching flows is to provide a way to reason about the behaviour of trains starting from multiple vertices simultaneously. In the Multi-Run Procedure (Algorithm 2), the resulting flow values $\hat{x}$ from the 2 form a candidate switching flow w.r.t. S and w. Theorem 4 gives two important properties about candidate switching flows.

Theorem 4 (Each Multi-Run profile is the minimal candidate switching flow) *[Gärtner et al., 2021] Let A, E, S, w be as in Definition 5 and let $\hat{x}$ be a Multi-Run profile of A, meaning that $\hat{x}(e)$ is the number of times edge $e \in E$ was traversed during some run of Algorithm 2 (Multi-Run Procedure). Then the following statements hold.*

1. *$\hat{x} \leq x$ for all candidate switching flows x (w.r.t. S and w). In particular, $\hat{x}$ is the unique minimizer of the total flow $\sum_{e \in E} x(e)$ over all candidate switching flows.*
2. *For fixed A, E, S, define $F(w) = (\hat{x}^-(v_1), \ldots, \hat{x}^-(v_k)) \in \mathbb{N}_0^k$. Then the function $F : \mathbb{N}_0^k \to \mathbb{N}_0^k$ is monotone, meaning that $w \leq w'$ implies that $F(w) \leq F(w')$.*

3.3.4 Constructing the ϕ-set

Through Theorem 4(i), we can say that the size of the set S and choice of vertex v in each iteration of the Multi-Run procedure does not affect the final candidate flow obtained. This property is crucial for the correctness of the overall algorithm. But the size of the subset S is crucial to the efficiency of the algorithm.

Therefore, Gärtner et al. [2021] designed the ϕ-set to achieve two main goals: Firstly, to provide a small subset of vertices S that can be used to define the function F and search for a fixed point. Secondly, to ensure that the shortest path from any vertex to $S \cup \{d, \bar{d}\}$ is short, which bounds the number of iterations needed in the Multi-Run Procedure. The optimal choice of $\phi = \sqrt{3}/\sqrt{2n}$ balances these two aspects and leads to the overall subexponential runtime of the algorithm.

Lemma 2 *Let $A = \left(V, o, d, \bar{d}, s_{even}, s_{odd}\right)$ be a terminating ARRIVAL instance with $|V| = n$. Let $\phi \in (0,1)$ be a real number. In $O(n)$ time, we can construct a ϕ-set S, meaning a set $S \subseteq V$ such that*
(i) $|S| \leq \phi \cdot (n+2)$;
(ii) for all $v \in V$, the shortest path from v to $S \cup \{d, \bar{d}\}$ in $G(A)$ has length at most $\log_2(n+2)/\phi$.

We decompose the ARRIVAL switch graph into layers based on the distance of the vertices to a destination, using BFS in $O(n)$ time. $L_i := \{v \in V \cup \{d, \bar{d}\} : dist(v) = i\}$, where dist(v) is the smallest distance from v to any of the destinations. Let $\ell := \max\{dist(v) : v \in V\}$

Algorithm 3 Procedure to compute a ϕ-set [Gärtner et al., 2021]

Input: ARRIVAL instance with layer decomposition $(L_0, \ldots, L_\ell), \phi \in (0,1)$
Output: a ϕ-set S
$S \leftarrow \emptyset$
$U \leftarrow L_0$
for $i = 1, \ldots, \ell$ do
  if $|L_i| < \phi |U|$ then
    $S \leftarrow S \cup L_i$
    $U \leftarrow \emptyset$
  $U \leftarrow U \cup L_i$
return S

In summary, the above algorithm for the ARRIVAL problem efficiently decides a given instance by searching for a switching flow. The algorithm first constructs a ϕ-set S, a small subset of vertices that covers the graph, using a careful choice of the parameter ϕ to balance the size of S and the lengths of shortest paths. It then uses the Multi-Run Procedure 2 to evaluate the Montone function, which maps outflows from S to inflows into S by simulating the movement of trains according to the switching behaviour. The resulting candidate switching flow $\hat{x}$ and its inflows are used in a search for a fixed point using Tarski's theorem. The search requires a subexponential number of evaluations of the monotone function, each taking a subexponential number of iterations of the Multi-Run Procedure. Once a fixed point w is found, the corresponding candidate switching flow $\hat{x}$ is an actual switching flow, and its destination (d or $\bar{d}$) determines the output of the ARRIVAL instance. This is done in subexponential time (in terms of the size of graph).

3.4 Finding Tarski Fixed Points by Decomposing k-Dimensional Lattice into Smaller Instances

We hope that our readers are now familiar with the subexponential algorithm from above, which was domain-specific for ARRIVAL graphs. In this section, we discuss a general purpose algorithm to find Tarski fixed points of a monotone function on a finite lattice. Dang et al. previously proposed an algorithm that solves the k-dimensional Tarski problem using $O(\log^k N)$ queries, where N is the width of the lattice. They conjectured that this query complexity is optimal, and this conjecture was supported by others in the field.

Fearnley et al. [2021b] recently disproved the optimality conjecture for dimensions higher than two by introducing a new algorithm. Their algorithm solves the 3-dimensional Tarski problem using $O(\log^2 N)$ queries, improving upon the $O(\log^3 N)$ queries required by the Dang et al. algorithm. The key ideas of the algorithm are:

- The concept of the "inner algorithm" that, given a 2D sub-instance, returns any point in the up or down set (formally defined later) of the 3D instance, rather than necessarily finding a fixed point.
- The "outer algorithm" that uses the inner algorithm to find a Tarski fixed point in 3D using $O(\log N)$ iterations.

3.4.1 Finding Fixed points for 3 Dimensional Instance

The algorithm that solves a 3-dimensional Tarski problem using $O(\log^2 N)$ queries, consists of two main components: an outer algorithm and an inner algorithm.

We borrow the definitions of these terms from the original paper. Given a function $f : L \to L$ on a finite Lattice L, the up set ($Up(f)$) is the set of all points $x \in L$ in which $x \leq f(x)$ (Component-wise ordering), and similarly the down set ($Down(f)$) is the set of all points $y \in L$ in which $f(y) \leq y$.

Slice: we use slices to fix some dimensions of the lattice while finding for fixed points, it is defined by a tuple $s = (s_1, s_2, \cdots, s_k)$, where $s_i \in \mathbb{N} \cup \{*\}$. We essentially fix all the dimensions with $s_i \neq *$ to be s_i. we will say that a slice is a principle slice when fixing only one dimension and letting other be free. **Sub-instance** : for some points $x, y \in L$ and $x \leq y$, the sub-instance $L_{x,y}$ is all the points in between x and y in component-wise ordering.

3.4.1.1 Outer Algorithm

The task of the outer algorithm is to find a solution to the Tarski instance by making $O(k \cdot \log N)$ calls to the inner algorithm, where n is the width of each dimension. We only apply the outer algorithm with $k = 3$. It is important to understand that here we are looking for points that lie in the up or down set of the three-dimensional instance.

The outer algorithm relies on the concept of an inner algorithm, which is defined as:

Definition 6 (Inner algorithm) *[Fearnley et al., 2021b] An inner algorithm for a Tarski instance takes as input a sub-instance $L_{a,b}$ with $a \in Up(f)$ and $b \in Down(f)$, and a principle slice s of that sub-instance. It outputs one of the following:*

- *A point $x \in L_{a,b} \cap L_s$ such that $x \in Up(f)$ or $x \in Down(f)$.*
- *Two points $x, y \in L_{a,b}$ that witness a violation of the order preservation of f.*

The outer algorithm maintains two points $x, y \in L$ throughout its execution, with the invariant that $x \leq y$, $x \in Up(f)$, and $y \in Down(f)$. The readers might wonder how we can ensure whether a sub-instance of L contains the Tarski fixed point or not. We provide Lemma 3 from Fearnley et al. [2021b], which proves that if certain conditions are satisfied by the sub-instance, then it will contain the fixed point.

Lemma 3 *[Fearnley et al., 2021b] Let L be a lattice and $f : L \to L$ be a Tarski instance. If there exist points $a, b \in L$ satisfying $a \leq b$, $a \in Up(f)$, and $b \in Down(f)$, then $L_{a,b}$ contains either:*

- *A fixed point x such that $f(x) = x$*
- *Two points x, y such that $x \leq y$ and $f(x) \not\leq f(y)$*

We start by initialising x as the least element of the Lattice L, and y as the greatest element. It is easy to see that the conditions in 3 are met for this assignment.

At each iteration of the outer algorithm, we can reduce the number of points in $L_{x,y}$ by a factor of two. To do this, the algorithm selects a dimension i of that sub-instance with greatest width. It then makes a call to the inner algorithm for the principle slice s defined so that $s_i = \lfloor (y_i - x_i)/2 \rfloor$ and $s_j = *$ for all $j \neq i$.

The inner algorithm would either return a violation of order preservation in L, or a point p in the slice such that $p \in Up(f)$ so we can set $x := p$, or a point p such that $p \in Down(f)$ so we can set $y := p$. The algorithm then moves to the iteration with updated sub-instance $L_{x,y}$. The algorithm can continue as long as there exists a dimension i such that $y_i - x_i \geq 2$, since this ensures that there will exist a principle slice strictly between x and y in dimension i that cuts the sub-instance in half. At the final sub-instance, we can exhaustively search for the fixed points.

3.4.1.2 Inner Algorithm

The ”inner algorithm” is a crucial component of the overall algorithm for finding Tarski fixed points in three dimensions. It efficiently finds a point in the up or down set of the three-dimensional instance, given a two-dimensional sub-instance and a principle slice. At a high level, the inner algorithm maintains a sub-instance $L_{a,b}$ that satisfies a specific invariant, ensuring that the sub-instance contains a solution to the Tarski problem.

In each iteration, the inner algorithm examines the midpoint of the current sub-instance and makes a case distinction based on the behavior of the Tarski function f at this point. Depending on the case, the algorithm either finds a solution directly or reduces the search space by focusing on a smaller sub-instance. Due to the number of cases handled by the inner algorithm, we find it difficult to provide a complete and accurate working

of the algorithm in our thesis. We strongly encourage readers who are interested in following the algorithm case-by-case to refer to Fearnley et al. [2021b], which includes the invariant, lemmas, and cases which we do not mention in this work.

3.4.2 Decomposition Theorem for k-Dimensional Instances

In order to solve Tarski instances in k dimensions using algorithms, Fearnley et al. [2021b] gives a decomposition theorem 5 that allows them to solve Tarski instances using algorithm mentioned in section 3.4.1.

Theorem 5 (Decomposition Theorem) *[Fearnley et al., 2021b] If an a-dimensional Tarski problem can be solved in q_a queries and a b-dimensional Tarski problem can be solved in q_b queries, then an $(a \cdot b)$-dimensional Tarski problem can be solved in $q_a \cdot (q_b + 2)$ queries.*

The algorithm for solving the Tarski problem on L proceeds by running the a-dimensional algorithm on A and using the b-dimensional algorithm to solve subproblems corresponding to slices of L.

We start by running the a-dimensional algorithm on the lattice A. Whenever the a-dimensional algorithm makes a query to a point $x \in A$. Now the algorithm constructs a slice s_x of the lattice L by fixing the coordinates corresponding to x in A and allowing the remaining coordinates to vary freely. The slice s_x will be a b-dimensional lattice and given to the b-dimensional algorithm. If the b-dimensional algorithm returns a violation of order preservation, terminate the entire algorithm and return the violation. Otherwise, the b-dimensional algorithm returns a fixed point y of the slice s_x. We combine x and y to get output from the monotone function. Since, the monotone function returns output of $a + b$ dimensions, we use the first a components to construct the response to the query x made by the a-dimensional algorithm. If the a-dimensional algorithm returns a fixed point $x \in A$, combine it with the fixed point y of the corresponding slice s_x to obtain a fixed point (x, y) of the lattice L.

By applying the decomposition theorem with the $O(\log^2 N)$-query algorithm for three-dimensional Tarski instances, Fearnley et al. [2021b]. We can decompose the original lattice into a cartesian product of the smaller lattices with dimensions of at most three and recursively find a fixed point in these smaller lattices. Thus, we obtain an algorithm for solving k-dimensional Tarski instances using $O(\log^{2\lceil k/3 \rceil} N)$ queries.

Theorem 6 *[Fearnley et al., 2021b] k-dimensional Tarski can be solved using $O(\log^{2\lceil k/3 \rceil} N)$ queries.*

Chapter 4

Implementing Algorithms to Decide and Generate ARRIVAL Instances

In this chapter, we will explore the practical aspects of working with ARRIVAL graphs and implementing the algorithms and graph generation discussed in the previous chapters. We will present pseudocode for key algorithms, discuss data structures and representations suitable for ARRIVAL graphs, and explain the implementation choices. Python will be our language of choice due to its flexibility, readability, and extensive ecosystem of libraries and tools. By providing clear explanations and concrete examples, this chapter aims to help readers implement and experiment with ARRIVAL algorithms on their own, gaining a deeper understanding of the concepts through practical application. The knowledge and skills acquired from this chapter will prepare readers for further exploration and research in the field of ARRIVAL graphs and their algorithms.

4.1 Arrival Graphs

Recall, Definition 2 : an arrival instance has functions s_{even} and s_{odd} which take a vertex and return their even and odd successors, respectively. Instead of dealing with functions which are relatively slower, we represent even and odd successors as an array indexed by vertices. All the vertexes are represented by integer numbers, so for example, the start node is always set to 0; Arrival graphs had two destinations d and $\bar{d}$ and in our implementation we assume that vertex d is the largest numbered vertex in the graph and $\bar{d}$ is always set to -1, for the rest of the chapter we refer to $\bar{d}$ as the sink node, and d as the target node.

The random generation process can be customized by modifying the constructor of the Arrival class. For example, setting the `do_reachability` tag while generating a new instance will trigger the reachability analysis algorithm, through which we can make sure that each vertex has a path to at least one of the destinations, i.e., catering to our assumption that the Arrival instances are terminating. Additional constraints or specific generation patterns can be incorporated to create ARRIVAL instances with desired characteristics.

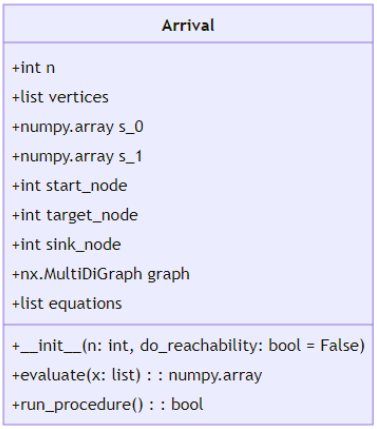


Figure 4.1: Arrival Python Class

4.1.1 Adopting an Instance based approach

In the provided code, the Arrival class represents an instance of the ARRIVAL problem. The constructor takes the number of nodes n as input and initializes the graph with n vertices, where vertex 0 is the origin and vertex $n-1$ is the destination. For the standard version of the graph generations, the even and odd successors for each vertex are normally and randomly chosen from the set of vertices **excluding the vertex itself**. This instance-based approach allows for flexible generation and manipulation of ARRIVAL graphs.

Once the successors lists are generated, we choose to convert them into NetworkX MultiDigraphs to achieve an object-oriented approach; the Arrival class also provides methods to access and modify the graph structure.

4.1.2 Construction of Equations for Monotone Functions

The `get_equations()` method in the Arrival class constructs a system of equations representing the monotone functions for each vertex in the graph. These equations capture the relationship between the number of visits to each vertex based on the switching behavior of the ARRIVAL problem.

The equations are constructed using SymPy, a Python library for symbolic mathematics. For each vertex v, the equation is given by:

$$X_v = \min\left(\sum_{p \in odd_v} \left\lfloor \frac{X_p}{2} \right\rfloor + \sum_{p \in even_v} \left\lceil \frac{X_p}{2} \right\rceil + \delta_v, n \cdot 2^n\right) \tag{4.1}$$

where X_v represents the number of visits to vertex v, odd_v and $even_v$ are the sets of vertices that have v as their odd and even successor, respectively, and δ_v is 1 if v is the origin and 0 otherwise. The equations are stored in the `equations` attribute of the Arrival instance, along with the corresponding SymPy symbols in the `X` attribute. These equations can be evaluated using the `evaluate()` method, which takes a list of values for each vertex and returns the evaluated results based on the equations. This method

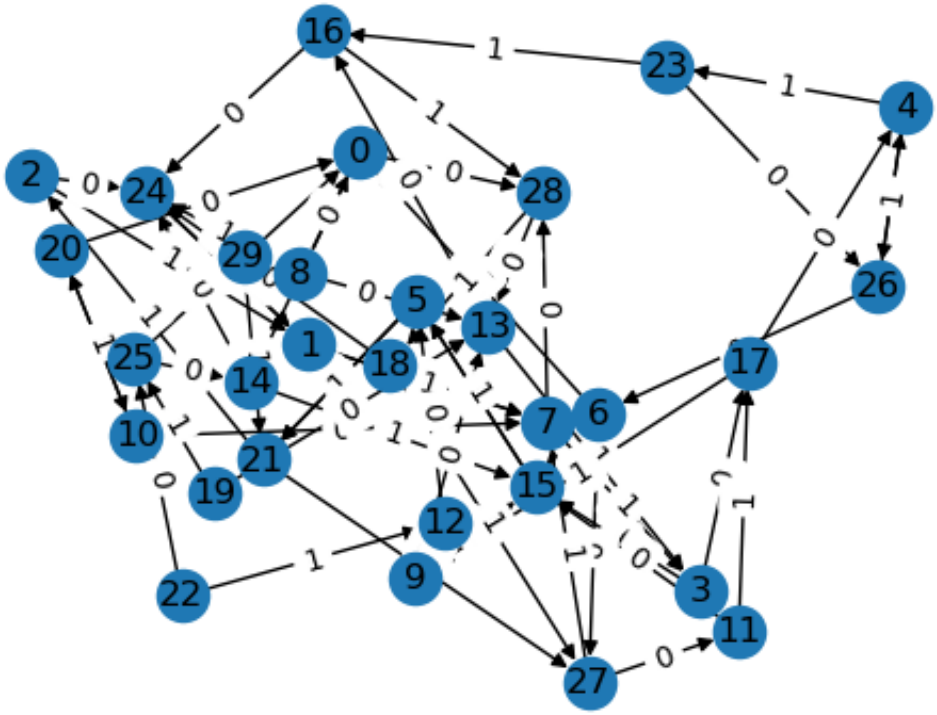


Figure 4.2: Arrival Graph of n = 30, edge label 0 and 1 represent the even and odd successors respectively. Note: This graph was generated without combining of unreachable nodes.

serves as the monotone function that we use to find Tarski fixed points in the ARRIVAL problem. The monotonicity of the `evaluate()` function (Definition 3 is crucial for applying Tarski's fixed point theorem and finding a solution to the problem.

4.1.3 Reachability of the destinations

To prevent the run procedure from getting stuck in loops, we perform some reachability analysis in the randomly generated graph. The provided code includes functionality to determine the reachability of the destination vertex in the ARRIVAL graph. The `combine_unreachable_nodes()` method identifies the vertices that do not have a direct path connecting them to the target vertex and combines them into a single sink vertex labelled as -1. By "direct path", we mean ignoring the even & odd labels and finding a directed path from a vertex to the target vertex. This is done primarily by doing a BFS search from the target node, which takes $O(n)$ time.

Reachability analysis and the combination of unreachable nodes are important preprocessing steps in solving the ARRIVAL problem, as they help us maintain the assumptions made about the graph structure.

4.2 Generating Branch Arrival Graphs

4.2.1 Generating Branch Arrival Graphs

In this section, we mention a specific type of ARRIVAL graph that we call a Branch Arrival Graph, inspired by the bit counter version of the ARRIVAL graph mentioned in the paper by Manuell [2021]. These graphs are designed to have a branching structure, where the origin vertex splits into two branches: one leading to the target vertex and the other leading to the sink vertex. The input parameters of the generating function `get_branch_instance_without_random` determine the lengths of these branches. This function takes two parameters, `size_of_instance`: An integer specifying the total number of vertices in the graph, and `split_ratio`: A float value between 0 and 1 indicating the ratio at which the graph splits into the two branches. If not provided, a random value is chosen.

By generating Branch Arrival Graphs with different `size_of_instance` and `split_ratio` values, we can create instances with varying lengths of the two branches. This allows for the analysis of the behaviour and performance of ARRIVAL algorithms on graphs with specific branching structures. The Branch Arrival Graphs provide a controlled and parameterized way to generate ARRIVAL instances, enabling the exploration of different scenarios and the testing of algorithms on graphs with known properties. We generate this graph with different randomisation and parameter settings, a detailed overview of our testing and results will be covered in Chapter 5

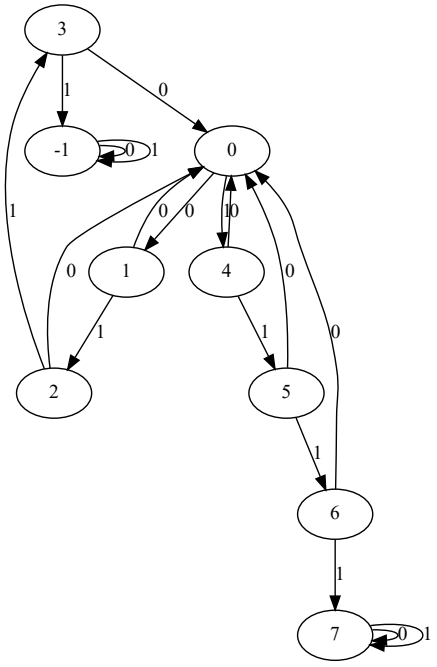


Figure 4.3: Graph Generated using `get_branch_instance_without_random` function, size of instance = 8, and split ratio = 0.5

4.2.2 Branch Arrival with Even-Odd Integer pairs

In this section, we introduce a variant of Branch Arrival Graphs with Even-Odd Integer Pairs. These graphs are constructed using an input even-odd integer pair, where the binary representations of these numbers determine the structure of the graph. The resulting Branch Arrival Graph has a structure that represents the binary representations of the input numbers `a` and `b` along the direct path from the origin to the end of the branch. They also have branches ending with destination vertex, the first branch ending at the target node and the other ending at the sink node. Each vertex in the graph has exactly two outgoing edges. All the vertices in the graph (excluding the origin and two destination vertices) have one outgoing edge to the origin vertex.

It is important to note that the construction of a valid ARRIVAL graph restricts us to use even-odd integer pairs, rather than any arbitrary pair of integers. This constraint arises from the requirement that the graph must have a well-defined switching behaviour. As the last bit of an even number in binary form is 0, and 1 for odd numbers, we ensure that the resulting graph is consistent with switching behaviour. Figure 4.4 shows an example of a graph made using (4,3) as the even-odd integer pair.

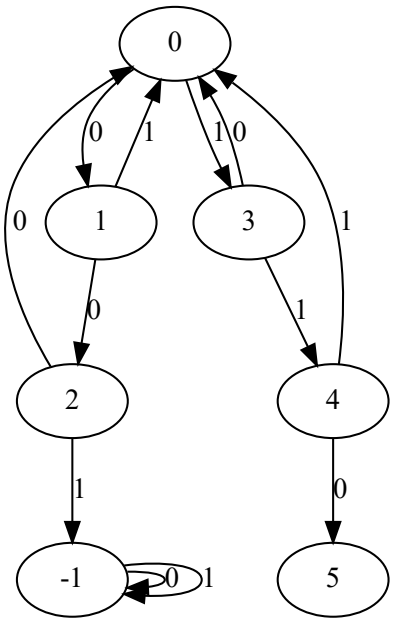


Figure 4.4: Branch Arrival with (4,3) integer pair.

4.3 Implementation of Iterative Algorithm

In this section, we will discuss our implementation for the iterative algorithm from section 3.2. The `iteration` function implements the iterative algorithm. It takes the monotone function `f` and an initial vector `x0` as parameters. The algorithm starts with

the all-1 vector $\bar{0} = (0,...,0)$ and repeatedly applies the monotone function `f` to the current vector until a fixed point is reached.

The choice of starting with the all-0 vector is based on our choice of lattice structure. `f`. As mentioned in the definition, for an Arrival instance with $|V| = n$, the corresponding monotone function $D : \{0,1,...,N\}^n \to \{0,1,...,N\}^n$ satisfies the property that if $x \leq y$ (component-wise ordering), then $f(x) \leq f(y)$. By starting with the all-0 vector, which is the smallest possible vector in the lattice, we ensure that the algorithm converges to the fixed point.

The convergence check is performed using the `is_converged` function, which calculates the maximum absolute difference between the corresponding components of consecutive iterations. If the maximum difference is 0, then we have found a fixed point and terminate the algorithm. This component-wise convergence check aligns with the monotonic nature of the function, where the components of the vector either increase or stay the same at each iteration. We now want to use this to compare different domain-specific algorithms like the subexponential algorithm and more general Tarski fixed point algorithms.

Algorithm 4 Iterative Algorithm

```
function IS_CONVERGED(x_new, x_old)
    return max(|x_new − x_old|) = 0
end function

function ITERATION(f, x_0)
    max_iterations ← len(x_0) · 2^len(x_0)
    x_old ← x_0
    for i ← 0 to max_iterations − 1 do
        x_new ← f(x_old)
        if is_converged(x_new, x_old) then
            return x_new
        end if
        x_old ← x_new
    end for
end function
```

4.4 Implementation of Subexponential Algorithm

In this section, we discuss the implementation details of the subexponential algorithm for solving the ARRIVAL problem, as described in [Gärtner et al., 2021].

4.4.1 Multi-Run Procedure & Candidate Switching Flows

The implementation of the Multi-Run Procedure follows the steps outlined in Algorithm 2. It initializes Python dictionary to keep track of the number of trains at each vertex and the current and next successors for vertices in $V \setminus S$, which have constant lookup

times. Recall that we defined the waiting set as all the vertices $V \setminus S$, which have some trains left. We use Python lists for the waiting set, taking $O(n)$ space. The random choice of the vertex v and the number of trains τ in our implementations was done using 'random.choice()' function from Python's 'random' module, which selects an element **uniformly** at random.

The implementation of candidate switching flows involves running the Multi-Run Procedure with a given subset of vertices S and a vector w specifying the outflows from each vertex in S. The resulting $\mathtt{t}$ values $\hat{x}$ from the Multi-Run Procedure form a candidate switching flow with respect to S and w.

4.4.2 Constructing the ϕ-set

The implementation of the ϕ-set construction follows the procedure described in Algorithm 3. It starts by decomposing the ARRIVAL switch graph into layers based on the distance of vertices to the destinations using breadth-first search (BFS) in $O(n)$ time and $O(\log n)$ space. The algorithm then iteratively adds vertices to S based on the size of the current layer L_i and the set of uncovered vertices U. We used $\phi = \sqrt{3}/\sqrt{2n}$, balancing the size of S and the lengths of shortest paths

The subexponential algorithm for the ARRIVAL problem combines the construction of the ϕ-set, the Multi-Run Procedure, candidate switching flows, to find a fixed point that is used to decide the problem.

4.5 Implementing Algorithm that Decomposes k-Dimensional Lattice into Smaller Instances

4.5.1 Outer Algorithm

We implement the outer algorithm 5 closely following the description mentioned in the previous chapter. The main 'while' loop of the outer algorithm iteratively reduces the search space $L_{x,y}$ until a solution is found or the search space cannot be further reduced. In each iteration, the algorithm finds the dimension i with the largest gap in x and y using 'np.argmax(y - x)'. Now this dimension i is used to create a principle slice s by fixing the dimension i to the centre of that dimension using 'np.full()' and 'np.nan'. The inner algorithm is invoked on the current sub-instance $L_{a,b}$ and principle slice s with 'inner_algorithm(La_b, s, f)'.

Each time the inner algorithm is invoked, it can return one of two things: 'None' meaning there was a violation in order preservation, so we terminate the whole algorithm; or inner algorithm returns updated x and y bounds reducing the size of the lattice. The loop continues until $y_i - x_i < 2$ for all dimensions i.

4.5.2 Inner Algorithm

The inner algorithm is a key component in finding Tarski fixed points in a 3-dimensional lattice. It takes a sub-instance, a principle slice, and the function being analyzed as

Algorithm 5 Outer Algorithm [Fearnley et al., 2021b]

```
procedure OUTERALGORITHM(f, x, y)
    k ← len(x)
    while any(y − x ≥ 2) do
        i ← argmax(y − x)
        s ← full(k, nan)
        s[i] ← x[i] + (y[i] − x[i])//2
        La_b ← (x.copy(), y.copy())
        result ← InnerAlgorithm(La_b, s, f)
        if result = None then
            print("An order preservation violation was found.")
            return None
        end if
        x, y ← result
    end while
    return x, y
end procedure
```

input. The algorithm identifies the fixed and free indices based on the principle slice and evaluates the function at the midpoint and four boundary points. The algorithm makes case distinctions to either detect an order-preserving violation or refine the sub-instance that potentially contains a Tarski fixed point. The pseudocode for the inner algorithm can be found in Appendix A.1.

4.5.3 Decomposition Algorithm

To implement the decomposition algorithm for finding Tarski fixed points in higher dimensions, we decided to create separate Python classes for algorithms $\mathcal{L}$, A, and B. This modular approach allows us to separate focus and work amongst various components and also maintains some universal variables like set P in algorithm $\mathcal{L}$, which store (x, y) pairs.

Algorithm $\mathcal{L}$ (Algorithm 6) is the main algorithm that takes inputs of the function f and the initial Lattice given by the lower and upper bound points. Algorithm $\mathcal{L}$ creates instances of algorithms A and B as needed and combines their results to find the fixed point.

Algorithm A & B (Algorithm 7,8) takes a sub-instance of the lattice of at most three dimensions. It then applies the outer algorithm for 3-dimensional instances with a lambda function returning only first three dimensions from the output of function f. So each time the outer algorithm queries a point (which is of three dimensions) an instance of algorithm B is invoked on the rest of the lattice. Algoirthm B finds and returns a fixed point for the slice of b-dimensions. We use lambda functions to create the sub-function f_s for the slice s. Algorithm B in turn calls algorithm $\mathcal{L}$ on the b-dimensional sub-lattice.

The lattice is represented using upper and lower bound points, which define the boundaries of the search space. All the points within these boundaries are considered part of the lattice. This representation allows for efficient manipulation and updating of the search space throughout our algorithms.

Algorithm 6 Class $\mathcal{L}$ $(f, lower, upper)$

$f \leftarrow f$
$P \leftarrow \emptyset$ ▷ Initialize the set of past points
procedure EVALUATIONFUNCTIONA(x)
  $l, u \leftarrow ConstructLU(x)$
  $algoB \leftarrow AlgorithmB(x, f, (l, u)).Solve()$
  $P \leftarrow P \cup \{Concatenate(x, y)\}$
  return $f(Concatenate(x, y))[: len(x)]$ ▷ Lambda sub-function
end procedure

procedure CONSTRUCTLU(x) ▷ least upper bound and greatest lower bound
  $D \leftarrow \{p[len(x) :] \mid p \in P \wedge p[: len(x)] \leq x\}$
  if $D = \emptyset$ **then**
    $l \leftarrow lower[len(x) :]$
  else
    $l \leftarrow Max(D)$ ▷ Element-wise maximum
  end if
  $U \leftarrow \{p[len(x) :] \mid p \in P \wedge p[: len(x)] \geq x\}$
  if $U = \emptyset$ **then**
    $u \leftarrow upper[len(x) :]$
  else
    $u \leftarrow Min(U)$ ▷ Element-wise minimum
  end if
  return l, u
end procedure

procedure SOLVE
  $algoA \leftarrow AlgorithmA(EvaluationFunctionA, (lower[: 3], upper[: 3]))$
  return $algoA.SolveTarski3D()$
end procedure

Algorithm 7 Class A ($sub\_function$, $lower, upper$)

```
1:  sub_function ← sub_function
2:  procedure SolveTarski3D
3:      final_sub_instance ← OuterAlgorithm(sub_function, lower, upper)
4:      srt, end ← final_sub_instance
5:      for i ← srt[0] to end[0] do
6:          for j ← srt[1] to end[1] do
7:              for k ← srt[2] to end[2] do
8:                  if sub_function([i, j, k]) = [i, j, k] then
9:                      return [i, j, k]
10:                     raise Exception("No fixed point found")
11:
```

Algorithm 8 Class B (x, f, $Lattice_B$)

```
1:  lower, upper ← Lattice_B
2:  procedure Solve
3:      if len(lower) > 3 then
4:          algoL ← AlgorithmL(SubFunction, (lower, upper))
5:          return algoL.Solve()
6:      else if len(lower) = 3 then
7:          algoA ← AlgorithmA(SubFunction, (lower, upper))
8:          return algoA.SolveTarski3D()
9:      end if
10: end procedure
11:
12: procedure SubFunction(y)
13:     return f(Concatenate(x, y))[−len(y) :]
14: end procedure=0
```

Chapter 5

Results of Experimentation on Arrival Graphs and its Algorithms

In this Chapter, we focus primarily on the experimental evaluation of our algorithms on different types of Arrival graphs. By generating diverse Arrival graphs and comparing the performance of fixed point algorithms, we try to observe the behaviour of these algorithms in practice and to determine whether the use of fixed point algorithms provides any benefits in deciding the ARRIVAL problem.

5.1 Time Complexity

Due to the time limit, our implementations as restricted majorly to only three algorithms: the decomposition algorithm [Fearnley et al., 2021b], the subexponential time algorithm [Gärtner et al., 2021] and the iterative algorithm. We were not able to fully implement the improved algorithm by Chen and Li [2022] for Tarski fixed point calculations. Furthermore, our experimentations were limited due to the explosion in runtimes of ARRIVAL instances of $n \geq 50$.

We know that the runtimes of our algorithms depend on the machine's computing power. We would like to declare that all the experiments were run on the same machine, with the following specifications: **Processor**: AMD Ryzen 9 4900HS with Radeon Graphics @3.00 GHz,**Installed RAM**: 16.0 GB. No graphics card was used for our experimentations.

5.2 Run Procedure

Figure 5.1 shows the results of Algorithm 1 on randomly generated Arrival graphs with sizes ranging from 10^1 to 10^4 vertices. The average number of vertices visited increases exponentially as the graph size grows, reaching around 2000 vertices for graphs with 10^4 nodes. which is not that high compared to the worst-case complexity of Algorithm 1. On the other hand, Figure 5.2 presents the results for branched Arrival instances generated using the `get_branch_instance_without_random` method with

sizes ranging from 10 to 40 vertices and split ratio = 0.5. The average number of vertices visited grows exponentially with the graph size, reaching approximately $2^2 1$ vertices for graphs with 40 nodes.

The readers can note that the average number of vertices visited in Figure 5.2 is much higher compared to Figure 5.1, despite the smaller graph sizes. This suggests that the branched Arrival instances generated by `get_branch_instance_without_random` indeed depict the worst-case of the obvious Run Procedure algorithms.

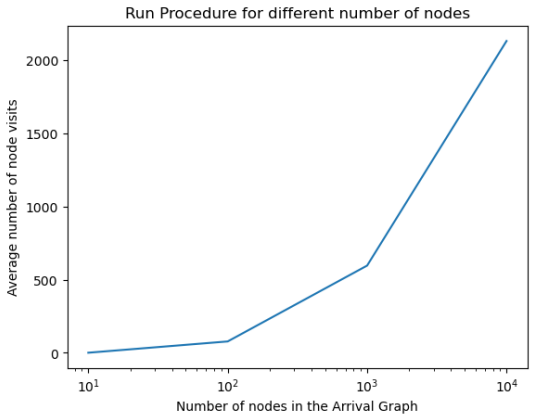


Figure 5.1: Average number of vertices visited by algorithm 1 before termination. The average was calculated over ten iterations, with graphs of 10, 100, 1000, 10000 vertices each.

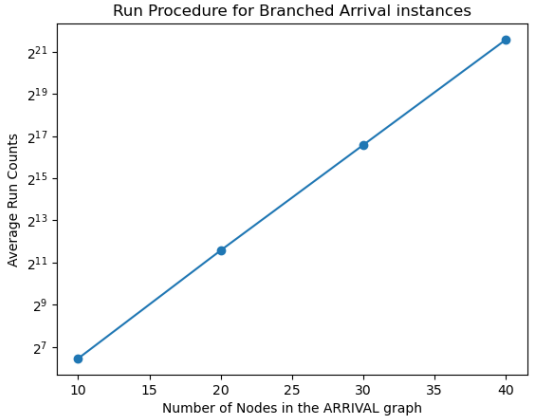


Figure 5.2: Average number of vertices visited by algorithm 1 before termination. The average was calculated over ten iterations, with Branch ARRIVAL graphs of 10, 20, 30, 40 vertices each (split ratio = 0.5).

5.3 Increasing the Graph Diameter by Dividing the vertices into Connected Clusters

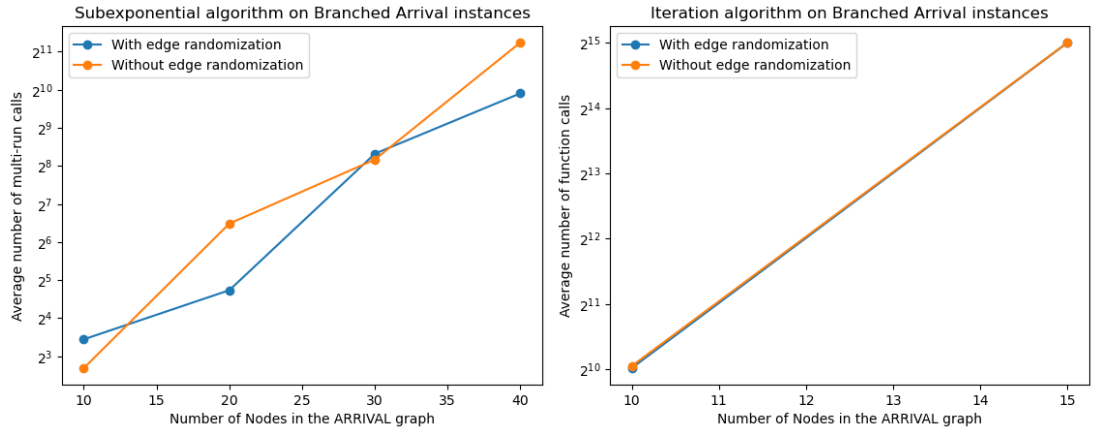
We generated graphs with clustered structures to try and increase the distance between the origin and destinations of the generated ARRIVAL graph. The approach we used is not novel, but it serves the purpose of experimentation and obtaining random hard instances for testing the performance of our algorithms.

Given the total number of vertices(n) and number of clusters(k), our code assigns unequal cluster sizes (ensuring each cluster has at least one vertex). We start by creating switch graph structure in each cluster and then do some inter-cluster linking. For each cluster, we randomly select another cluster (excluding itself) and choose a random source node from the current cluster and a random target node from the selected cluster (both nodes are excluding destination nodes). We randomly decide whether to update the even (s_0) or odd (s_1) successor of the source node to point to the target node. All the randomness is done using `random.random()`, which generates normally distributed random values. We also ensure that all the nodes in a cluster have a directed path to one of the destinations.

5.4 Experimenting with Branch Arrival instances

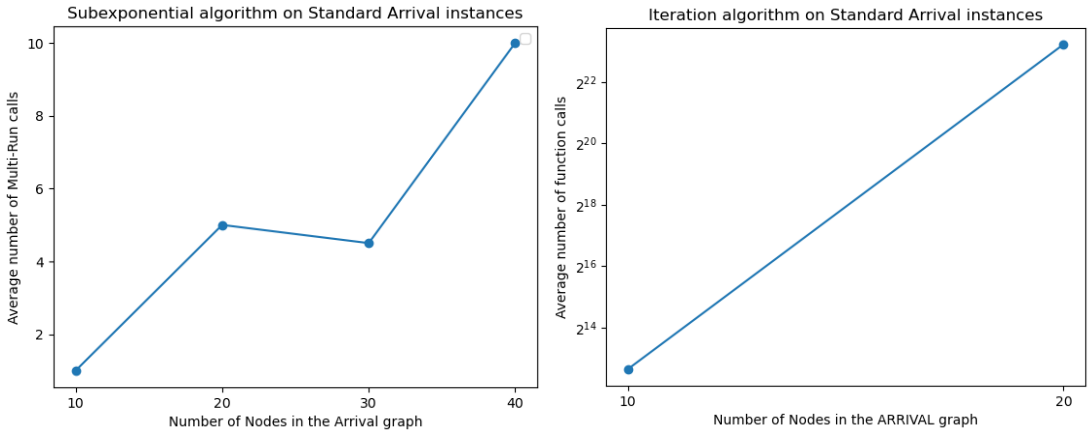
In this section, we compare the runtimes, behaviour of our implemented algorithms. The two plots in Figure 5.3 performance of the subexponential algorithm and the iteration algorithm on branched Arrival instances. First, the graph on the left (5.3(a)) shows the subexponential algorithm's number of times the Algorithm called the Multi-Run Procedure(2) on larger graphs (up to 40 nodes). Despite the exponential growth in the number of multi-run calls, the algorithm manages to handle these larger instances, showcasing practical runtimes aligned with the theoretical complexity, that the problem can be decided in at most $O(n^{2^{\lceil \phi(n+2)/3}})$, where n is the size of the ARRIVAL graph. On the other hand, we were only able to test the iteration algorithm's performance on smaller graphs (up to 15 nodes) because of its complexity in worst cases like these branch instances. In figure 5.3(b), we measure the number of times the monotone function was queried before a fixed point was found for different-sized graphs. Comparing the two algorithms based on these plots, it becomes clear that the subexponential algorithm demonstrates better scalability and potential for handling larger Arrival instances.

Similarly, we ran experiments on the standard version of the graph generations. We found out interesting that the number of Multi-Run procedure calls significantly decreased for the same-sized instances but iteration algorithm still increased exponentially with size of the problem. This indicates that we should do use Branched graph instances to experiment on other algorithms we implement.



(a) (b)

Figure 5.3: Comparison between the subexponential algorithm (left) and the iterative algorithm (right) for Branch ARRIVAL problems.



(a) (b)

Figure 5.4: Comparison between the subexponential algorithm (left) and the iterative algorithm (right) for randomly (normal distribution) generated Arrival problems.

5.4.1 Results on Branch Arrival with Even-Odd Integer pairs

In section 4.2, we defined how we can construct an Arrival instance using two integer numbers. In this section we show the results of experimentation on those instances. In our experimentation with Branch Arrival Graphs constructed using even-odd integer pairs, we observed a consistent result: the branch corresponding to the smaller integer in the pair is always chosen by the Algorithm 1. This means that the train will always reach the destination which is on the path represented by the binary expansion of the smaller integer.

We can prove this result by examining the structure of the Branch Arrival Graph and the properties of even-odd integer pairs. Let's consider an even-odd integer pair (a, b), where a is even, and b is odd, which are used to construct a Branch Arrival Graph.

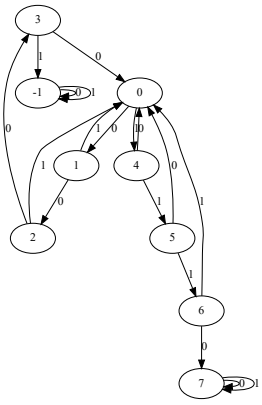
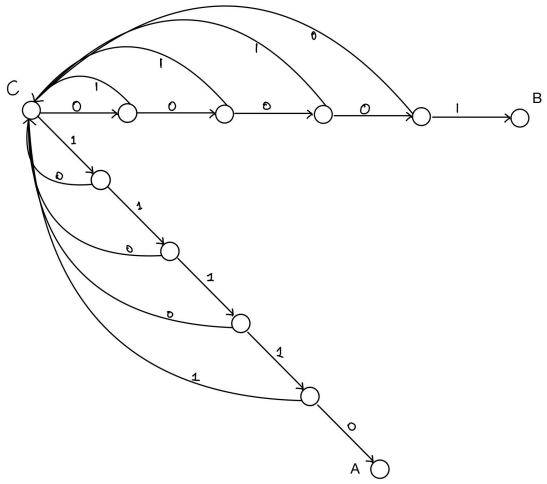


Figure 5.5: Branch Arrival with (8,7) integer pair.

Theorem 7 *In a Branch Arrival Graph constructed using an even-odd integer pair* (a,b)*, where* $a < b$*, the destination on the branch corresponding to the integer a is always reached by the Algorithm 1.*

We prove this theorem by using the following graph generated using two integers a and b.



The graph shows the origin vertex C and two destinations, A & B, corresponding to the branch made using integers a & b, respectively. We can observe from the graph that the path corresponding to A would be taken at most b times before the algorithm reaches B, and similarly, the path corresponding to B will be taken at most a times before the algorithm reaches destination A; By algorithm, we mean the Algorithm 1.

Therefore, if $a < b$, then the path corresponding to destination A will be traversed more than the other path and thus destination A will be reached. Similarly, for $a > b$, destination B will be reached.

This result provides insight into the behaviour of the ARRIVAL algorithm on Branch Arrival Graphs constructed using even-odd integer pairs. It shows that the algorithm will always favour the branch corresponding to the smaller integer, leading to a predictable traversal pattern. One might wonder whether there is a way to represent general ARRIVAL graphs as integers and compare them.

5.5 Installation and Experiment Recreation

To facilitate the reproducibility of the experiments and results presented in this thesis, all the code used throughout the research is provided in a GitHub repository (https://github.com/PC53/Honors-Project-ARRIVAL-Graphs.git). This allows readers and researchers to clone the repository easily, set up the necessary environment, and recreate the experiments on their own machines.

Chapter 6

Conclusions

Finally, we have implemented the algorithms and did start a comprehensive evaluation about the use of Tarski Fixed Point algorithms to decide ARRIVAL. Is this a good approach to solve the problem? Our answer is ”It depends”. We note that in testing larger graph structures, due to limited computational power, the algorithms took a lot of time to terminate. Whereas, just simulating the train run (Algorithm 1) was significantly faster due to its simple nature. Fixed point algorithms can be better in cases of one-destination Arrival problem where reaching the destination is not guaranteed.

6.1 Originality and difficulty

Throughout the duration of this thesis, we learned and understood the intricacies of Tarski Fixed Point algorithms. To implement algorithms, it was crucial that we make implementation choices carefully; we finalised the details of the code independently to increase the scalability of the problem. We adopted a variety of methods to analyse the runtimes and memory usage for our implementations.

Since there is no similar theoretical discussion and/or experimental implementations of ARRIVAL graphs known to us, we found the development of the right experiments and implementations of the right graph structure challenging. In our experimental evaluations, we faced challenges in our search for real patterns. When experimenting, we found that the algorithms were performing well because the graphs we generated were easier to solve, even for 1000 nodes in the graph. We focused a significant amount of time on generating graphs that can become much harder to solve and thus can come close to the upper bounds of these algorithms. We developed a systematic approach to experiments, focusing on both average-case and worst-case analysis.

6.2 Limitations and Future Work

Throughout our work on implementing and evaluating Tarski fixed point algorithms for the ARRIVAL problem, we have made significant progress in understanding the problem and the performance of various algorithms. Due to time restrictions, we were

not able to experiment thoroughly on the decomposition algorithm given by Fearnley et al. [2021b], which we have implemented in the previous chapters. This algorithm has shown promising theoretical results, and further experimentation might provide insights into its practical performance.Moreover, there is potential for implementing and testing other advanced fixed point algorithms, such as the one proposed by Chen and Li [2022], which further improves the upper bound for finding fixed points of a monotone function. These are currently state-of-the-art algorithms. Additionally, it would be interesting to see if implementations of algorithms from different areas for solving the system of linear equations associated with the ARRIVAL problem, could improve the practical runtimes.

We acknowledge that our algorithms' implementations can be improved. We encourage readers to explore and contribute to our GitHub repository for this project, as collaborative efforts can help optimize the performance. The open-source nature of our work allows anyone to build or improve our findings. In Section 5.4.1, we discussed a representation of special ARRIVAL graphs as integer numbers and proved a theorem related to this representation. Future work focusing on generalizing this representation for the ARRIVAL graph can find patterns which help in deciding faster. Looking beyond the ARRIVAL problem, we encourage researchers to explore implementations of other switch graph problems, such as Propp Machines Cooper et al. [2007] and Eulerian walks Priezzhev et al. [1996]. These problems share similarities with the ARRIVAL problem and have important applications in various domains. We hope that our work serves as a foundation and inspiration for further investigations in this area.

Bibliography

URL https://scratch.mit.edu/projects/1200078/.

Oct 2022. URL http://kinderlabor.ch/.

Xi Chen and Yuhao Li. Improved upper bounds for finding tarski fixed points, 2022.

Joshua Cooper, Benjamin Doerr, Joel Spencer, and Gábor Tardos. Deterministic random walks on the integers. *European Journal of Combinatorics*, 28(8):2072–2090, 2007.

Chuangyin Dang, Qi Qi, and Yinyu Ye. Computations and complexities of tarski's fixed points and supermodular games. URL http://arxiv.org/abs/2005.09836.

Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jiří Matoušek, and Emo Welzl. Arrival: A zero-player graph game in np ∩ conp, 2017.

Kousha Etessami. Personal communication, 2024. Personal communication via [medium] on [specific date].

Kousha Etessami, Christos H. Papadimitriou, Aviad Rubinstein, and Mihalis Yannakakis. Tarski's theorem, supermodular games, and the complexity of equilibria. *CoRR*, abs/1909.03210, 2019. URL http://arxiv.org/abs/1909.03210.

John Fearnley, Martin Gairing, Matthias Mnich, and Rahul Savani. Reachability switching games. *Logical methods in computer science*, 17, 2021a.

John Fearnley, Dömötör Pálvölgyi, and Rahul Savani. A faster algorithm for finding tarski fixed points, 2021b.

Bernd Gärtner, Sebastian Haslebacher, and Hung P Hoang. A subexponential algorithm for arrival. *arXiv preprint arXiv:2102.06427*, 2021.

Alexander E Holroyd and James Propp. Rotor walks and markov chains. *Algorithmic probability and combinatorics*, 520:105–126, 2010.

Marcin Jurdziński. Deciding the winner in parity games is in up co-up. *Information Processing Letters*, 68(3):119–124, 1998.

CS Karthik. Did the train reach its destination: The complexity of finding a witness. *Information processing letters*, 121:17–21, 2017.

Bastian Katz, Ignaz Rutter, and Gerhard Woeginger. An algorithmic study of switch graphs. *Acta informatica*, 49:295–312, 2012.

Graham Manuell. A simple lower bound for arrival. *arXiv preprint arXiv:2108.06273*, 2021.

Sebastiaan Cornelis Willem Ploeger. Improved verification methods for concurrent systems. 2009.

Vyatcheslav B Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Physical Review Letters*, 77 (25):5079, 1996.

Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. 1955.

Uri Zwick and Mike Paterson. The complexity of mean payoff games on graphs. *Theoretical Computer Science*, 158(1-2):343–359, 1996.

Appendix A

Some Pseudocode Used In Implementation

A.1 Inner Algorithm used in section 4.5.2

Algorithm 9 Inner Algorithm

procedure INNERALGORITHM(L, s, f)
  $a, b \leftarrow L$
  $fixed\_index \leftarrow None$
  for $i \leftarrow 0$ **to** $len(s) - 1$ **do**
    if $notisnan(s[i])$ **then**
      $fixed\_index \leftarrow i$
      $a[i] \leftarrow s[i]$
      $b[i] \leftarrow s[i]$
      break
    end if
  end for
  $free\_idx_1 \leftarrow (fixed\_index + 1) \bmod 3$
  $free\_idx_2 \leftarrow (fixed\_index + 2) \bmod 3$
  $mid \leftarrow (a + b)//2$
  $bot \leftarrow full(len(mid), nan)$
  $bot[fixed\_index] \leftarrow a[fixed\_index]$
  $bot[free\_idx\_1] \leftarrow floor((a[free\_idx\_1] + b[free\_idx\_1])/2)$
  $bot[free\_idx\_2] \leftarrow a[free\_idx\_2]$
  $top \leftarrow full(len(mid), nan)$
  $top[fixed\_index] \leftarrow b[fixed\_index]$
  $top[free\_idx\_1] \leftarrow floor((a[free\_idx\_1] + b[free\_idx\_1])/2)$
  $top[free\_idx\_2] \leftarrow b[free\_idx\_2]$
  $left \leftarrow full(len(mid), nan)$
  $left[fixed\_index] \leftarrow a[fixed\_index]$
  $left[free\_idx\_1] \leftarrow a[free\_idx\_1]$
  $left[free\_idx\_2] \leftarrow floor((a[free\_idx\_2] + b[free\_idx\_2])/2)$
  $right \leftarrow full(len(mid), nan)$
  $right[fixed\_index] \leftarrow b[fixed\_index]$
  $right[free\_idx\_1] \leftarrow b[free\_idx\_1]$
  $right[free\_idx\_2] \leftarrow floor((a[free\_idx\_2] + b[free\_idx\_2])/2)$

```
31:     f_mid ← f(mid)
32:     if mid[free_idx_1] ≤ f_mid[free_idx_1] and mid[free_idx_2] ≤ f_mid[free_idx_2] then
33:         return mid, b
34:     else if mid[free_idx_1] ≥ f_mid[free_idx_1] and mid[free_idx_2] ≥ f_mid[free_idx_2] then
35:         return a, mid
36:     else if mid[free_idx_1] ≤ f_mid[free_idx_1] and mid[free_idx_2] > f_mid[free_idx_2] then
37:         if mid[fixed_index] ≤ f_mid[fixed_index] then
38:             f_right ← f(right)
39:             if right[fixed_index] > f_right[fixed_index] then
40:                 return None
41:             else if right[free_idx_1] < f_right[free_idx_1] then
42:                 p ← right
43:             else
44:                 return a, right
45:             end if
46:         else
47:             f_bot ← f(bot)
48:             if bot[fixed_index] > f_bot[fixed_index] then
49:                 return None
50:             else if bot[free_idx_2] > f_bot[free_idx_2] then
51:                 p ← bot
52:             else
53:                 return bot, b
54:             end if
55:         end if
56:     else if mid[free_idx_1] > f_mid[free_idx_1] and mid[free_idx_2] ≤ f_mid[free_idx_2] then
57:         if mid[fixed_index] ≤ f_mid[fixed_index] then
58:             f_top ← f(top)
59:             if top[fixed_index] > f_top[fixed_index] then
60:                 return None
61:             else if top[free_idx_2] < f_top[free_idx_2] then
62:                 p ← top
63:             else
64:                 return a, top
65:             end if
66:         else
67:             f_left ← f(left)
68:             if left[fixed_index] > f_left[fixed_index] then
69:                 return None
70:             else if left[free_idx_1] > f_left[free_idx_1] then
71:                 p ← left
72:             else
73:                 return left, b
74:             end if
75:         end if
76:     end if
```

```
77:     if all(b − a ≤ 1) then
78:         print("Terminal phase reached.")
79:     end if
80:     return a, b
81: end procedure
```