CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Model-based deadlock prevention for traffic planning of autonomous vehicles

Master's thesis in Computer science and engineering

David Möller
Alexander Ohlin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Model based deadlock prevention for traffic planning of autonomous vehicles

David Möller
Alexander Ohlin

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Model-based deadlock prevention for
traffic planning of autonomous vehicles
David Möller, Alexander Ohlin

© David Möller, Alexander Ohlin, 2023.

Supervisor: Andru Gheorghiu, Department of Computer Science and Engineering
Mentor: Jonas Hellgren, Volvo Autonomous Solutions
Examiner: Peter Damaschke, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2023

Model-based deadlock prevention for
traffic planning of autonomous vehicles
David Möller, Alexander Ohlin

Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Volvo Autonomous Solutions are developing a system for planning the routes of fleets of autonomous vehicles. Autonomous control creates several problems that must be solved; among these is the possibility for the policy of said vehicles to end up in deadlock. This thesis proposes new concepts to describe the problem and methods for preventing a vehicle fleet from deadlocking. As the action that led to deadlock might not be recent, the term implicit deadlock was introduced, which is a configuration of vehicle positions from which deadlock is inevitable. The methods developed successfully prevent deadlocks at several pilot and test sites. However, results indicate that time for computing implicit deadlocks grows exponentially in the size of the site and the number of vehicles in the fleet. A neural network model was also trained using data generated from preprocessing of deadlocks to approximate the process and enable deadlock predictions not discovered before.

Keywords: Computer science, neural networks, graph theory, deadlock, autonomous vehicles.

Acknowledgements

We want to thank our advisor, Jonas Hellgren, for all his help in this project while motivating us to explore different ideas and providing continuous valuable feedback. We also want to thank our supervisor, Andru Gheorghiu, for the interesting discussions we had regarding the methodology and results of the project, as well as his helpful comments and feedback for writing a qualitative report. Lastly, we would like to thank Volvo Autonomous Solutions for allowing us to do our thesis work with them and providing some of the software that we have used.

David Möller, Gothenburg, 2023-06-12

Alexander Ohlin, Gothenburg, 2023-06-12

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Background . . . 2
1.2 Previous work . . . 3
1.2.1 Deadlock avoidance via machine learning . . . 4
1.2.2 Deadlock avoidance in multi-AGV systems . . . 4
1.3 Problem definition and goals . . . 5
1.4 Limitations . . . 6
1.4.1 States . . . 6
1.4.2 Data . . . 6

2 Theory **7**
2.1 Graph theory . . . 7
2.1.1 Graph . . . 7
2.2 Site & site graph . . . 8
2.3 State . . . 8
2.3.1 Vehicle claiming . . . 9
2.3.2 Example of *Claim* . . . 10
2.3.3 Validity . . . 10
2.3.4 Example of invalid state . . . 11
2.4 State graph . . . 11
2.5 Deadlock . . . 13
2.5.1 Coffman deadlock . . . 13
2.5.2 Safe state . . . 14
2.5.3 Explicit deadlock state . . . 14
2.5.4 Implicit deadlock state . . . 15
2.5.5 Example of true implicit deadlock . . . 17
2.5.6 Classifying a deadlock . . . 17
2.6 Trie data structure . . . 18
2.7 Neural networks . . . 19
2.7.1 Perceptrons and dense neural networks . . . 19
2.7.2 Activation functions . . . 20

3 Methods **23**
3.1 Explicit deadlock finder . . . 24
3.1.1 Finding blocking vehicles . . . 24
3.1.2 Finding chains of blocking vehicles . . . 25
3.1.3 Combining returned chains of vehicles . . . 26
3.1.4 Trimming deadlocks . . . 26
3.2 Implicit deadlock finder . . . 27
3.2.1 Example of one implicit finding step . . . 28
3.3 Memory-based deadlock classifier . . . 29
3.3.1 Lookup using masking . . . 29
3.3.2 Lookup using trie data structure . . . 29
3.3.3 Implementing the deadlock constraint . . . 31
3.4 Neural networks . . . 31
3.4.1 Dataset . . . 31
3.4.2 Dense neural network . . . 31
3.4.3 Classifying unknown deadlocks . . . 32

4 Results **33**
4.1 Description of sites . . . 33
4.2 Explicit finder results . . . 34
4.3 Comparing memory-based classifiers . . . 35
4.4 Implicit deadlock finder . . . 36
4.4.1 Test sites . . . 36
4.4.2 Sharp sites . . . 38
4.5 Neural network classifiers . . . 40
4.5.1 Classifying unknown deadlocks . . . 41
4.6 Finding all possible deadlocks . . . 42
4.7 Summary of results . . . 43

5 Conclusion **45**
5.1 Discussion . . . 45
5.2 Ethics . . . 46
5.3 Future work . . . 46
5.3.1 More sites of different scales . . . 46
5.3.2 Benchmark against existing solution . . . 47
5.3.3 Improving explicit deadlock finder . . . 47
5.3.4 Implicit deadlock computational limits . . . 47
5.3.5 Possible unknown implicit deadlocks . . . 48
5.3.6 N-step TD implicit finder . . . 49
5.3.7 Neural networks . . . 49

Bibliography **51**

A Appendix 1 **I**
A.1 Correctness proof for implicit deadlock finder . . . I
A.2 Explicit deadlock classifier . . . II
A.2.1 Finding blocking vehicles . . . II

A.2.2 Using blocked vehicles III
A.3 Graph convolutional network IV
A.4 GCN - results V
A.5 Explicit deadlock finder psuedocode VI
A.6 Implicit deadlock finder VIII
A.7 Trie data structure IX
A.8 Deadlocked states X
A.9 Site graphs XIII

List of Figures

1.1 Example of a site with left) Explicit deadlock right) Implicit Deadlock 2
1.2 Search tree of the traffic planner, nodes $S0, S1, ..., S9$ are fleets states, the red nodes $S3, S4, S6$ are explicit deadlocks and node $S1$ is an implicit deadlock. 3

2.1 Example of a directed graph 8
2.2 Part of a site graph where the physical locations of vehicle v_A, v_B, and v_C are marked in blue while claimed nodes, C, are marked in yellow 10
2.3 Part of a site graph where the physical locations of vehicle v_A, v_B, v_C, and v_D are marked in blue while claimed nodes, C, are marked in yellow. Node 5, marked in red, is claimed by both vehicle v_B and v_D 11
2.4 Example of how a state graph is constructed from states in a site graph 12
2.5 Example of a state graph. 12
2.6 Example of True Explicit Deadlock 15
2.7 To the left, a true implicit deadlock $S5$ is visualised. To the right is an example of a state graph where state $S7$ is an explicit deadlock and $S5$ and $S6$ are true implicit deadlock. 17
2.8 Example of trie data structure. Null nodes are excluded. 18
2.9 Dense Neural network example. 20
2.10 The Softmax and ReLU activation functions 21

3.1 Flowchart describing the different components developed and their relation. 23
3.2 Flowchart describing the explicit deadlock finder. 24
3.3 Scenario 1 and 2 of vehicles blocking each other, the orange nodes are the locations of the vehicles and the blue squares are the area which the vehicle claims 25
3.4 Example of finding a explicit deadlock 25
3.5 Example of finder encountering split node 26
3.6 Example of composite deadlock with three vehicles A, B and C 26
3.7 Flowchart of the implicit deadlock finder method 27
3.8 Example of finding implicit deadlocks one transition from explicit deadlock, explicit deadlocks marked with ED and confirmed implicit deadlocks marked with ID 28
3.9 Trie example with two inserted deadlocks 110 and 101 30

4.1 Computational time of explicit deadlock for the four different sites, the right plot is the same as the left but includes Sharp Site 2 which requires adjusting the y-axis. 34
4.2 Comparing deadlock lookup time between trie and masking. X-axis shows number of deadlocks in data structure and y-axis shows lookup time. 35
4.3 Graph over Vehicle/Depth/Time (S) of Screwdriver site. Generated for max 13 vehicles and 100 depth. 37
4.4 Graph over Vehicle/Depth/Time (S) of Oval site. Generated for max 14 vehicles and 50 depth. 38
4.5 Graph over Vehicle/Depth/Time (S) of Sharp site 1. Generated for max 14 vehicles and 500 depth. 39
4.6 Graph over Vehicle/Depth/Time (S) of Sharp site 2. Generated for max 17 vehicles and 25 depth. 40
4.7 Confusion matrices comparing the results of the neural network classifier vs the Memory-Based classifier, both using data generated from 7 vehicles 41

A.1 Scenario 1 and 2 of vehicles blocking each other. The orange nodes are the locations of the vehicles and the blue squares are the area which the vehicle claims. II
A.2 Confusion matrices comparing the results of the graph neural network classifier vs the Memory-Based classifier, both using data generated from 7 vehicles V
A.3 Explicit deadlock of 2 vehicles on Screwdriver site X
A.4 Explicit deadlock of 2 vehicles on Oval site X
A.5 Explicit deadlock of 4 vehicles on Sharp site 1 XI
A.6 Implicit deadlock of 4 vehicles on Sharp site 1 XI
A.7 Implicit deadlock of 6 vehicles on Sharp site 1 XII
A.8 Site graph of screwdriver site XIII
A.9 Site graph of Oval site XIII
A.10 Site graph of Sharp site 1 XIII
A.11 Site graph of Sharp site 2 XIV

List of Tables

3.1 Examples of states which are looked up in the Trie in Figure 3.9 . . . 30

4.1 Table of nodes and deadlocks per site. * denotes that not all deadlocks have been found, and that this is the highest amount that have been computed to completion. 33

4.2 Results from Dense Neural Network Classifier 40

A.1 Results from Graph Neural Network Classifier using one GCN layer . V

1

Introduction

Every day, throughout the world, millions of goods and resources get transported to where they need to be. Transportation could be considered one of the most essential systems in keeping the modern world running. Due to the immense interdependence of society, delivery and routing form an immensely complex web of pick-ups and drop-offs. Factories need materials, supermarkets need essentials, people need to get to work, and so on. Luckily, not all these systems are connected and can be divided into subsystems. However, when vehicles need to coordinate, we generally want them to arrive quickly and efficiently, which is not always easy.

To solve this problem, intelligent systems and algorithms have been developed to approach optimal planning of tasks. For example, one could simulate several steps ahead, look at the possible outcomes, and then pick the best one. However, when not simulating everything, the system might be tricked into planning a way that seems good but leads to terrible results. One such bad outcome is that some vehicles manage to deadlock themselves. This means that, through some earlier choice, none of these vehicles can continue driving as they are all blocking each other. These situations are ones we would like to avoid.

This thesis project has investigated how to predict a future deadlocked state in a traffic planner. This has been done by developing a constraint for the system that avoids such situations. We were given access to an already existing traffic planner, developed by Volvo Autonomous Solutions (VAS), which lacked the functionality to detect deadlock. This planner manages fleets of autonomous vehicles, such as trucks and load carriers, and coordinates their movement to maximize productivity. In this thesis project, we aimed to extend the planner such that deadlocks could be found and prevented.

VAS has deployed fleets of autonomous trucks at several sites worldwide, such as mines and quarries. Some sites have critical sections where deadlocks can occur, and factors such as limited charging stations and single-lane areas exist. Detecting these deadlocks manually can be done, as one can check all vehicles and see if there is a subset of them where they are all blocked. However, this thesis has expanded the definition of deadlock to include detecting implicit deadlocks, i.e. states that will eventually lead to a deadlock.

1.1 Background

The traffic planning system being developed by VAS is a planner for the vehicle fleet. This fleet, in turn, comprises several trucks or other autonomous vehicles that handle delivery tasks at a specific site. A site is a location of one of Volvo's customers that currently use their autonomous trucks for delivery and where the planner could potentially be used. These sites are represented abstractly as directed graphs (for details, see Definition 2.1.2), which simplifies roads, charging stations, etc., into a set of nodes and edges. A state refers to a specific assignment of vehicles to some set of nodes. States are further explained in Definition 2.3.1. Lastly, a deadlock comes in two variants, implicit and explicit. An explicit deadlock is the traditional type of deadlock, where vehicles are each blocking some other vehicle and themselves being blocked such that no vehicle can move. An implicit deadlock is a type of deadlock where all vehicles can potentially drive. However, any sequence of actions from an implicit deadlock will lead to an explicit deadlock in a finite number of actions. Implicit deadlocks are further explained in Definition 2.5.4.

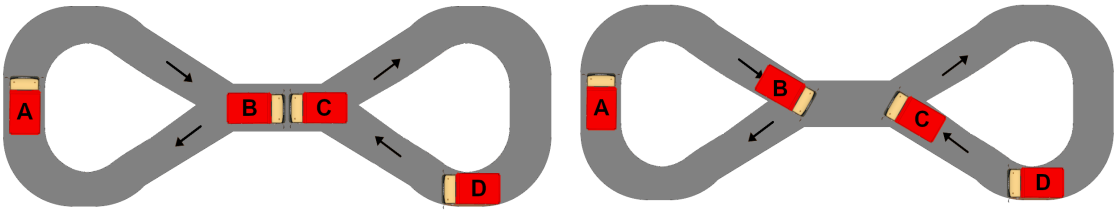


Figure 1.1: Example of a site with left) Explicit deadlock right) Implicit Deadlock

The entire site and the current positions of all vehicles are known to the planner at all times. An example of a fleet state at an example site can be seen in Figure 1.1. Note that, in this example, vehicles can only drive forward or stand still. The left image shows an example of explicit deadlock. In this case, vehicles B and C block each other from driving forward, thus being stuck. As such, vehicles B and C are in explicit deadlock. This is a clear-cut case that one wants to avoid to minimize vehicle downtime. On the other hand, the example to the right shows a case of implicit deadlock. All vehicles can drive in this example, but vehicles B and C are still going to deadlock. There is not enough space on either side for B or C to get past each other, meaning regardless of what they do, they will end up in an explicit deadlock. As such, this is an example of B and C being in implicit deadlock. Finally, as one might have noted, vehicles A and D never mattered regarding if the site was in deadlock or not. This is normal, as it is usually only a subset of vehicles that become deadlocked. These concepts are further expanded upon in Theory Sections 2.5.2 and 2.5.4.

The planner turns the above example state into a directed graph. It uses the state to plan which edges the vehicles should follow. To do this, it creates a search tree of possible future states and uses state estimations and a set of constraints to decide which path does not validate any constraints. A constraint is a Boolean rule specifying which states the vehicles should not be allowed to travel to. Because the planner operates in real-time, it is infeasible to compute the entire search tree, and

the value of a state must therefore be approximated. An example of how the search tree might look with such a deadlock constraint is shown in Figure 1.2.

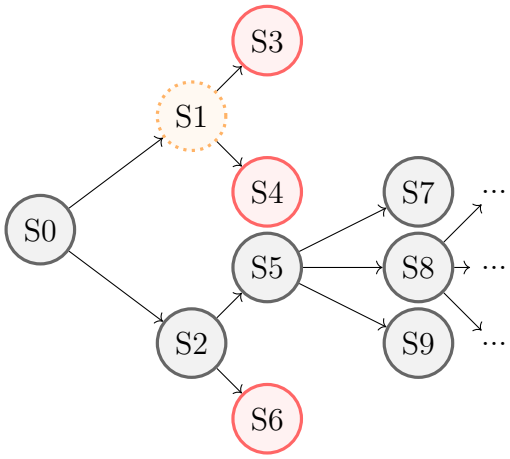


Figure 1.2: Search tree of the traffic planner, nodes $S0, S1, ..., S9$ are fleets states, the red nodes $S3, S4, S6$ are explicit deadlocks and node $S1$ is an implicit deadlock.

State $S1$ in Figure 1.2 is in implicit deadlock because all possible states after $S1$ will eventually lead to a deadlock. To determine whether a state S is an implicit deadlock, one must simulate all future states after S and see if they all lead to an explicit deadlock. This could either be preprocessed or done during run-time. However, this approach is very time-consuming and impractical. As such, while designing a planner that searches far into the future in run-time is possible, it is more practical to preprocess some of the work. Developing a method to predict that a state is deadlocked would be very useful for the traffic planner.

Predicting deadlocks is very important for several reasons. Firstly, some actions will result in a state where there is no way to escape a deadlocked situation. Some deadlock recovery, either manual or automatic, must then be used to get the fleet back to a safe state. Doing all of this is very inefficient and wastes time. It would therefore be much better if the situation could be predicted and avoided altogether. In specific cases, being unable to deal with deadlocks means that the planner cannot be used at specific sites, as resetting is hard. Additionally, the ability to accurately predict implicit deadlock allows all states beyond the state in question to be ignored. This can remarkably improve performance by reducing the number of states the planner must consider and check for other constraints.

1.2 Previous work

Various methods have been developed to avoid deadlock. Many algorithms make deadlock unlikely or impossible by only doing specific actions in specific states (following a set policy). For instance, deadlock prevention has been studied in networks of agents [1], train systems [2][3], and intelligent warehouses [4]. However, it is sometimes difficult to avoid deadlock altogether, especially when the number of states in a system is large and long simulations are needed.

1.2.1 Deadlock avoidance via machine learning

In the situations described above, researchers have explored deep learning to assist in detecting and preventing deadlock. For example, Kaid *et al.* [5] used a neural network to detect faults in manufacturing machines that could lead to deadlock. This shows that neural networks are potential candidates to identify undesirable states, such as deadlock.

Reinforcement learning (RL) has also been applied to deal with deadlock. Goto *et al.* [6] used RL in an automotive setting, examining the use of RL with multiple asynchronous agents to learn a policy for avoiding deadlock. Using an asynchronous multi-agent actor-critic method, they could train agents to consider the perspectives of others to solve deadlock situations. This further shows that machine learning methods can effectively address deadlock problems. However, we aimed to predict deadlock as a constraint for a policy rather than develop the policy itself.

1.2.2 Deadlock avoidance in multi-AGV systems

In a previous master's thesis from Chalmers in 2020, the authors aimed to create a deadlock-free system of multiple Automated Guided Vehicles (AGVs) [7]. Guiding AGVs is similar to what we aimed to do, as it manages the deadlock problem for autonomous vehicles. Although, the authors had a different approach and scope. Their approach is to procedurally generate rules that make the AGVs avoid previously detected deadlocks.

This thesis, on the other hand, is interested in classifying a state of vehicles as a deadlock or not. A rules-based approach would be able to prevent explicit deadlock due to the relatively small amount of them, by designing a rule for each deadlock found. However, designing general and straightforward rules that prevent all implicit deadlocks, especially those that do not create more deadlocks by themselves, is not simple. While it is possible to stochastically find every state in deadlock and design an explicit rule to avoid it when encountered, this becomes an unfeasible large problem when the sites grow. Lastly, searching through and verifying that any rule has not been broken is, regarding computation time, a costly process and would not work for a system that considers many future states every time it makes a decision. As such, these are the reasons why their approach was not chosen to solve the problems outlined in this thesis.

Furthermore, deadlock avoidance in AGV has been explored in depth. For example, Zhou *et al.* [8] developed a method for avoiding explicit deadlocks. They defined a term known as a "deadlock cycle", which is a state where some subset of vehicles creates a cycle of vehicles blocking each other such that none of them can drive. Then, they built a system so their AGVs could avoid such situations. To do as such, they let their vehicles be aware of their possible moves two steps later and use that information to avoid stepping into a deadlock cycle.

However, there are some differences compared to the work done for this thesis. Firstly, their the AGVs have different paths depending on different types of AGVs. Furthermore, their AGVs are only aware of their own path and are independant

agents. In comparison, our methods are built for a single planner that decides the movement of all vehicles and can know if a future state is a collision. All vehicles move on the same directed graph. Secondly, Zhou lets each AGV act in real-time, while the methods developed for this thesis finds a substantial amount of deadlocks beforehand and then use real-time checking of said data to ensure deadlock is avoided. Finally, and most critically, Zhou does not appear to include scenarios where a sequence of states leads to a deadlock, i.e., implicit deadlocked states. For them, a state is either a deadlock or is completely safe. We have defined the term implicit deadlock, which are states that will reach a deadlock regardless of actions taken but is not deadlocked in the present. Based on the paths their AGVs follow, it appears that implicit deadlocks are not possible as two paths at most intersect at a single point. This is a large difference from the work done in this thesis.

1.3 Problem definition and goals

This thesis has attempted to develop a solution that can accurately predict deadlock from the current fleet state, subject to some requirements. Specifically, the goal was to develop and train a mathematical model to classify whether a given state is in a deadlock. This approach has two essential requirements: (1) it must make predictions without conducting an exhaustive search, using at most a limited number of future states, and (2) it must be able to make predictions within a limited time frame. The research question that drives us can therefore be stated as follows:

How can a mathematical model predict deadlock in a vehicle planning system?

The planner works in real-time, meaning it does not precompute a plan but rather considers the current state and calculates the approximate optimal decision. As such, decisions related to deadlock must be made very quickly. This means it is not feasible to exhaustively calculate all possible future states and mark every deadlock found.

To evaluate the model's performance mentioned earlier, two key metrics will be considered: accuracy and computation time. The model's accuracy must be high enough to distinguish between deadlock and safe states. While the model does not need to reach 100% accuracy, it must be sufficiently high so that the planner can identify and avoid the path to a deadlock. Computation time is also a critical factor, as the model must be able to make predictions within the limited time frame available. Due to the vast number of states that needs to be examined per second, the status of a single state must be concluded in less than a millisecond.

1.4 Limitations

Due to time constraints and the scope of the thesis, it was impossible to examine every approach. This section outlines the limitations and related problems that this thesis will not deal with. Limitations constrain the work so that focus is put on relevant parts and that actual results can be reached. Furthermore, this section also outlines limitations on preliminary resources and tools available.

1.4.1 States

For this thesis, we have only examined and designed a deadlock constraint for the planner at VAS. Therefore, we have not investigated any other problems that have to do with the optimal planning of vehicles, physical collisions or specific requirements of trucks, such as the need to recharge. We can assume that such problems either already are solved or will be solved outside the purview of this thesis. While some of these factors might lead to trucks doing specific actions, our only focus was to ensure that whatever the goals of the truck were, they did not reach a state of deadlock.

Additionally, we only examined one fixed site state at a time. This means we assume that the trucks are at specific digital coordinates which might not exactly be where they are in real life. From these coordinates, we then determine if the site is in a deadlock. In real life, the trucks might not precisely be where we assume they are, but this abstraction is sufficient for deadlock detection. We also consider all possible actions a truck can do, not just the ones most valuable to the planner.

1.4.2 Data

There was no readily available training data when we started working. We were required to design models that could work without data or, as we chose, gather this data ourselves. As such, we had a limitation on how much data we could create and use. The tools that were available to us were the traffic planner and several sites that could be simulated. From these simulated sites, data could be generated to train a model, but this amount of data was limited. We could not simply train a model based on preexisting data to solve the problem.

2

Theory

This chapter explains several concepts relevant to this thesis. It will introduce some common machine learning and graph theory concepts while presenting the authors' new definitions, such as explicit and implicit deadlock. Readers experienced in graph theory and machine learning may ignore those parts in this chapter.

2.1 Graph theory

There are a few concepts of graph theory that one needs to understand to follow the explanations and descriptions of problems and solutions in this thesis. This section introduces these concepts with a short description of each.

2.1.1 Graph

A *graph* is a set of items and a set of connections between those items.

Definition 2.1.1 (Graph)**.** A *graph* G is defined as a pair (N, E), where N is a set of nodes (also called vertices), and E is a set of edges between the nodes $E \subseteq \{ (u, v) \mid u, v \in N\}$ [9].

Nodes could, for example, be specific locations on a map and edges being the roads used to traverse between them. They could also be a set of states in a system where the edges denote that the system can go from one state to another [10]. Graphs come in two types, which are undirected and directed. See Figure 2.1 for an example of a directed graph which is defined in Definition 2.1.2.

Definition 2.1.2 (Directed graph)**.** A graph $G = (V, E)$ is a directed graph if all edges $(u, v) \in E$ are ordered pairs.

The edges of a graph are written as (u, v) for an edge between the nodes u and v. Additionally, u is called the tail node, and v is called the head node, as the edge leads from node u to node v.

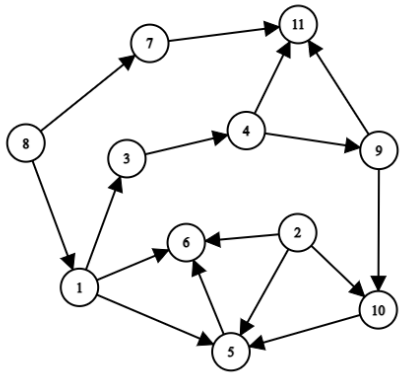


Figure 2.1: Example of a directed graph

2.2 Site & site graph

A site is an enclosed area with a road network, such as a quarry or mine, where trucks collect, transport and deliver materials. A site graph is the digital graph representation of this site, comprised of nodes and edges. Integers denote nodes. For example, the first node is written as node 1. Edges denote possible actions a vehicle can take from a given node and will be written as (u, v) for some nodes u and v.

The edges from each node n represent the possible vehicle movement at a given node. This movement does not necessarily mean driving forward and could be any type of movement, including reversing, turning and standing still. Note also that all nodes n have the edge $(n, n) \in E(n)$, which defines the action of not moving. Finally, it is worth noting that it is assumed that no site has a node with only a self-edge, a.k.a. a node that, once entered, can never be left. With these explanations, a site graph is defined in the following way:

Definition 2.2.1 (Site graph)**.** A Site Graph is a directed graph *SiteGraph* $= (N, E)$, where N denotes the set of nodes of the site and E denotes the set of edges or vehicle actions. Edges are written as pairs of nodes $\{(u, v) \mid u, v \in N\}$. Each node has a unique numerical identifier, $N = \{n_1, n_2, n_3, ..., n_i\}$.

2.3 State

Each site has many different states that it can be in, depending on how many vehicles are on it and where they are. The position of a vehicle, where it is on the site, is defined as an edge of the site graph and not a node. This was done because the vehicle is assumed to be physically situated on the edge's tail node and is driving towards the head node. It is currently taking the action represented by the edge it is on. For clarity, the terms edge (of the site graph), action and position of a vehicle

all refer to the same thing: a pair of nodes in the site graph. Formally, a state is defined as:

Definition 2.3.1 (State)**.** A state is a specific assignment of vehicles to edges on a site. For a $SiteGraph = (N, E)$, V is a subset of the edges of the site, s.t. $V \subseteq E$.

Each edge in V has a unique vehicle present. Additionally, each unique choice of V defines one unique state that the site can be in.

2.3.1 Vehicle claiming

Each vehicle occupies a physical space, simplified as a rectangle around it. This rectangle can contain multiple nodes besides the two nodes part of the position of a vehicle. The same node being in the rectangles of two different vehicles implies that they both need to occupy the same physical space, which is impossible. As such, the set C of claimed nodes and the Claim function define what nodes are already occupied by some vehicle. Claiming is done to keep track of where vehicles are and what space they occupy and prevent vehicles from colliding. The claim function is defined on edges because, as a reminder, the set of vehicles is a subset of edges on a site graph. It is also needed to define a vehicle's possible actions, which is further explained in Definition 2.4.1.

Definition 2.3.2 (Claim)**.** For a given state s with vehicles V, the set of all claimed nodes C is a subset of the nodes in N. To define C, the *Claim* function, $E \rightarrow \mathcal{P}(N) \setminus \emptyset$, must first be defined. Claim, for some vehicle v, is defined as a non-empty subset of nodes N, such that $Claim(v) = \{n_1, n_2, n_3, ..., n_m \mid 1 \leq m \leq |N|\}$. The set of claimed nodes C is then defined as the union of all subsets of nodes for each vehicle, or formally:

$$C = \bigcup_v^V Claim(v)$$

In the above definition, each vehicle claims a set of nodes on a site, and the set C is the union of these sets. The vehicle's position and physical size define the space it claims. The position determines the vehicle's rotation by the direction from the tail node to the head node. The physical space claimed by a vehicle is calculated as a rectangle around the edge (i.e. containing the two nodes of the edge) where the vehicle is present. The width and height of said rectangle are defined by the real-life width and length of the vehicle and some marginal padding for safety reasons. As such, for a vehicle v, all nodes contained in the area of the rectangle computed from v is in $Claim(v)$.

2.3.2 Example of *Claim*

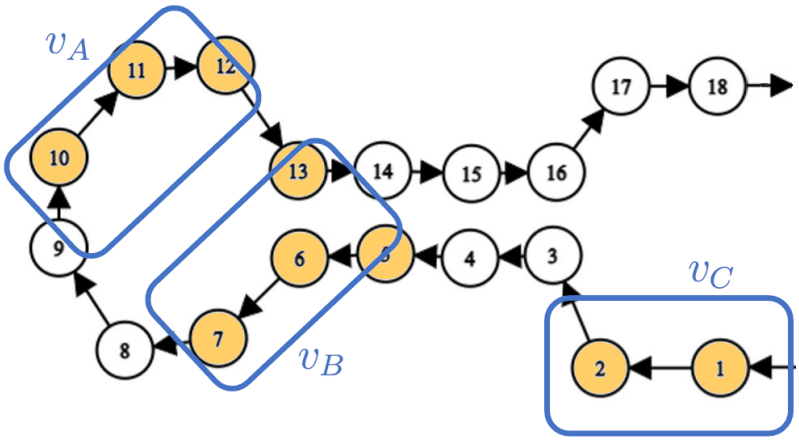


Figure 2.2: Part of a site graph where the physical locations of vehicle v_A, v_B, and v_C are marked in blue while claimed nodes, C, are marked in yellow

Figure 2.2 shows an example of a site graph. Note that the self-edges have been removed from the figure for clarity. This example has 3 vehicles, named A, B and C. Each vehicle claims some nodes, shown by its blue rectangle. Vehicle A stands on position (10, 11), B on position (6, 7) and C on position (1, 2). As can be seen, the Claim of vehicle A is $Claim((10,\ 11)) = \{10,\ 11,\ 12\}$, for vehicle B it is $Claim((6,\ 7)) = \{6,\ 7,\ 5,\ 13\}$ and lastly for vehicle C the Claim is $Claim((1,\ 2)) = \{1,\ 2\}$. A does not claim node 9, as it does not overlap with the center of the node. In reality, nodes have no radius and are simply coordinates.

2.3.3 Validity

A state is valid when the claimed nodes of each vehicle do not overlap with anyone else. Otherwise, some vehicles require the same physical space, which is impossible and means the state is invalid. It can be **assumed** that all states given as examples or considered in practice are valid. As such, validity is defined as:

Definition 2.3.3 (Validity)**.** A valid state s with the set of vehicles V is valid if it holds that $\forall a, b \in V : Claim(a) \cap Claim(b) = \emptyset$. Otherwise, the state is invalid.

2.3.4 Example of invalid state

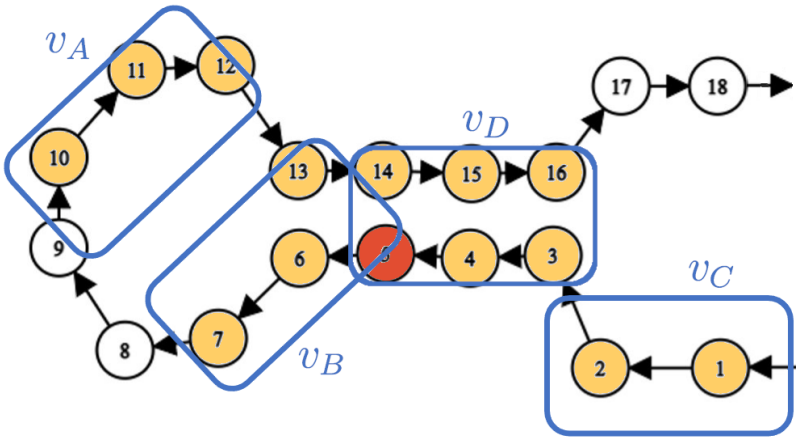


Figure 2.3: Part of a site graph where the physical locations of vehicle v_A, v_B, v_C, and v_D are marked in blue while claimed nodes, C, are marked in yellow. Node 5, marked in red, is claimed by both vehicle v_B and v_D

Figure 2.3 shows an example of an invalid state. In comparison to the state in Figure 2.2, vehicle v_D has been inserted. This results in two vehicles claiming the same node, that is $Claim(v_B) \cap Claim(v_D) = \{n_5\}$. This state is, therefore, invalid since no two vehicles may claim the same node.

2.4 State graph

The state graph, which could also be referred to as a finite-state machine, describes all possible valid states of a specific site and the transitions between states. See the following definition:

Definition 2.4.1 (State graph)**.** For a given $SiteGraph = (N, E)$, the state graph is defined as the directed graph $G = (S, T)$. The set S contains all possible valid states, and the set T contains the transitions between states based on the *SiteGraph*. For states a, $b \in S$, there exists a transition $(a, b) \in T$ if and only if and there exists some action $(u, v) \in E$, and $u \in a$ and $v \in b$.

As per the definition, every state in the state graph maps to one unique, valid assignment of vehicles to the site. The depth from one state to another describes the shortest path length between the two states. Informally, transitions are defined as one vehicle taking a single action, leading from one *valid* state to another. The remaining vehicles do not move.

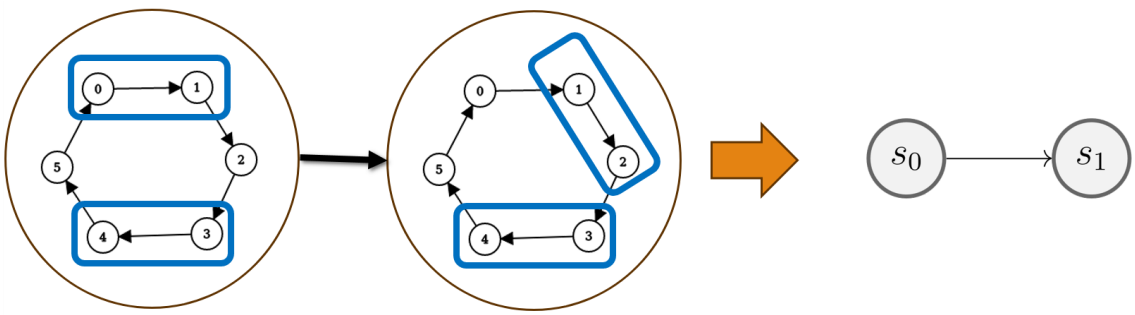


Figure 2.4: Example of how a state graph is constructed from states in a site graph

For an example of a transition, see Figure 2.4. As can be seen, there is an underlying site graph with two vehicles. On the left, two states are visualised, the only difference being that one vehicle has moved. As such, each state is represented in the state graph to the right as a node, and the transition as an edge between these nodes. The complete state graph can be seen in Figure 2.5.

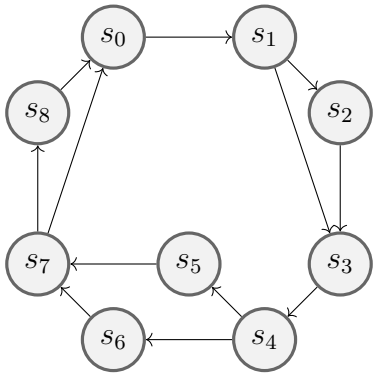


Figure 2.5: Example of a state graph.

The example of a state graph in Figure 2.5 shows that the underlying site graph would only have 9 valid states. Note that the number of states grows exponentially in relation to the number of nodes in the corresponding site graph, making it impractical to store the whole state graph in memory. Furthermore, as an example of depth, if one searches from state s_3 with a depth of 2, one would find states s_5 and s_6.

However, one action could also be to stand still, which corresponds to following a self-edge in the state graph, as no vehicle moving does not change the state. Although, it can be assumed that this never happens unless no alternative action is available (i.e. the state is an explicit deadlock). As such, self-edges are excluded from any figure displaying a state graph.

2.5 Deadlock

This section describes the concept of deadlock. First, the commonly used definition of deadlock, referred to as a Coffman Deadlock, will be described along with a description of why site graphs can deadlock. Based on this definition, all the terms used in the rest of the thesis concerning deadlock will be defined and explained. Lastly, why only keeping track of a small number of deadlocks allow full deadlock coverage will be explained.

2.5.1 Coffman deadlock

Deadlock is a common problem in many systems, particularly those involving parallel action, limited resources, and multiple agents. It first appeared as a problem in computer systems, but many real-life systems have, over the years, also been examined from the perspective of deadlock prevention. Formally, based on the definition of deadlock specified by Coffman [11], deadlocks can occur in a system if and only if **all** the following 4 conditions hold at the same time:

1. **Mutual exclusion:** At least one resource must be non-shareable. Only one agent can use the resource at any given time.

2. **Resource holding:** An agent is currently holding at least one resource and requesting additional resources which are being held by other agents.

3. **No Preemption:** The system can/must not de-allocate resources once they have been allocated; they must be released by the agent holding them voluntarily.

4. **Circular Wait:** An agent is waiting for a resource that is held by another agent, who in turn is waiting for the first agent to release their resource. In general, there exists a set of waiting agents $A = A_1, A_2, A_3, ...A_N$, s.t A_1 is waiting for A_2 who is waiting for A_3, and so on until A_N is waiting for A_1.

In this work's context, the agents are the vehicles situated on different sites. Since the vehicles drive on a road network, one can naturally model the problem as a directed graph. Resources are made up of physical positions that each vehicle can be at. It can be easily shown that the first three conditions hold trivially. For mutual exclusion, there can not be two vehicles in the same space simultaneously as they would have collided. For resource holding, a vehicle will always have a physical position, a resource, while simultaneously wanting to reach other positions to progress its mission. These other positions might, in turn, be held by other vehicles, meaning that the second condition also holds. Additionally, a resource in this context can never be de-allocated involuntarily, as that is impossible for the physical vehicle. Therefore, it can be seen that the existence and, likewise, the avoidance of deadlock hinges on the presence of a circular wait condition.

2.5.2 Safe state

A safe state, or a non-deadlock, is simply any state that is not in deadlock. As long as one chooses actions to remain in safe states, all vehicles will eventually be able to move. There must also always exist a path in the state graph that leads back to the current state, i.e. each safe state is part of at least one cycle of safe states. Safe states are defined as such:

Definition 2.5.1 (Safe state)**.** Let s be a state in the state graph G. s is a *Safe State* if there exists some cycle in G that starts and ends in s, and in which all vehicles have moved at least once.

It can also be assumed that any site should have some safe states, as otherwise, it would be impossible to operate on said site in a way that does not lead to deadlock, which means that the site is faulty by design. Discovering that a site has no safe states is also an efficient way to signal that there is some problem with said site.

2.5.3 Explicit deadlock state

The definition of explicit deadlock is identical to that of a Coffman deadlock. An explicit deadlock is a state where vehicles wait for other vehicles to move, forming a circular wait condition. This definition was also used by Zhou *et al.* [8], although they called it a deadlock cycle. However, this thesis introduces two types of explicit deadlocks: true and composite. These are defined as follows:

Definition 2.5.2 (True explicit deadlock)**.** True explicit deadlocks (TED), for a given state graph $G = (S, T)$, are a subset of states $TED \subseteq S$. A state e is in TED if it has no outgoing edges in the state graph, excluding its self-edge. Furthermore, it must also hold that no vehicle can be *removed* from the state such that the new state with the vehicle removed also has no outgoing edges.

Informally, this definition means that there is no action a for any vehicle in V of this given state e that, if followed, leads to another valid state. As such, the definition implies that all the vehicles of the state are in a circular wait. For there to be no valid actions means that the claims of the vehicles are all preventing each other from moving forward. Since all vehicles are prevented from moving, the state must contain a circular wait condition, where some vehicle v_1 is blocked by vehicle v_2 and so on until vehicle v_n is blocked by v_1. The key difference that separates a true explicit deadlock from a composite one is that all vehicles are in a circular wait condition.

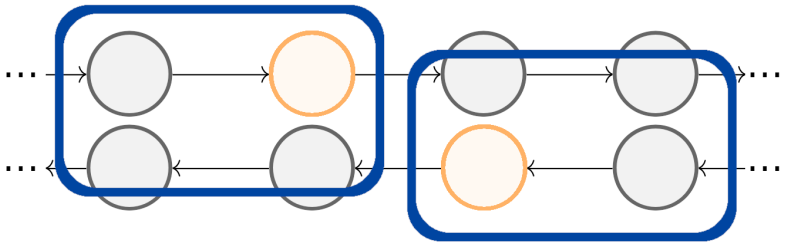


Figure 2.6: Example of True Explicit Deadlock

An example of this deadlock can be seen in Figure 2.6. Both vehicles cannot move as they are blocked by each other. This fits the definition of a circular wait condition, as they both are waiting for each other to give up their current position while reserving their own. As all vehicles cannot move, removing one would remove the deadlock, meaning this state is a true explicit deadlock.

Definition 2.5.3 (Composite explicit deadlock)**.** A composite explicit deadlock is a state where a subset of vehicles are in true explicit deadlock.

A composite explicit deadlock only has some vehicles form a circular wait. As such, some sequence of *reductions* exists, i.e., a set of vehicles that can be removed, from every composite explicit deadlock that results in a true explicit deadlock. Note that a composite explicit deadlock in the state graph might have outgoing edges due to other non-deadlocked vehicles' presence. While this state might have some vehicles that can move, the state is still in deadlock. This is because the subset of vehicles in true deadlock can never move, regardless of the other vehicles' actions. As such, it is a state that should be avoided. As a consequence, any actions taken from a composite explicit deadlock can only ever lead to another composite explicit deadlock.

There are no constraints on the vehicles which are not in a true explicit deadlock, which means that the remaining vehicle might or might *not* eventually become stuck. There are two scenarios from a composite deadlock in the state graph; either these vehicles are part of a cycle or a finitely long path. Being in a cycle means that some vehicles are in a true explicit deadlock while others could choose a sequence of actions such that they never get stuck. The simplest example is that a non-deadlocked vehicle is in a roundabout and drives around it forever. On the other hand, they might also be on a path that leads to being blocked by vehicles in true deadlock. Then vehicles will then eventually get stuck behind the true deadlocked vehicles and can thus never move from that point.

2.5.4 Implicit deadlock state

Implicit deadlocks are states in the state graph that will *unavoidably* reach a deadlock. This means that, regardless of what sequence of actions one takes from an implicit deadlock, the resulting state (and any in-between) are **always** implicitly or explicitly deadlocked. However, the vehicles in an implicit deadlock are not required to be in a circular wait, and it is possible to have an implicit deadlock where all

vehicles have some available action. Like explicit deadlocks, there are both true and composite implicit deadlocks. These are defined as follows:

Definition 2.5.4 (True implicit deadlock)**.** A state s in the state graph G is a true implicit deadlock if and only if s has at least one outgoing edge, s is not part of any cycle in G and can not be reduced to a true explicit deadlock.

True implicit deadlocks are defined by the vehicles' inability to avoid reaching explicit deadlocks in the future. The only states in G that have an outgoing edge (i.e. the state is not a true explicit deadlock) and are not part of any cycle (i.e. the state is not a safe state) must be those on their way to an explicit deadlock. The remaining states are either true implicit deadlocks or composite explicit deadlocks. For each state to have a single label, the definition also includes that a true implicit deadlock can not be reduced to a true explicit deadlock, which removes composite explicit deadlocks.

Definition 2.5.5 (Composite implicit deadlock)**.** A composite implicit deadlock is a state where a subset of vehicles are in true implicit deadlock.

Like with composite explicit deadlocks, it is always possible to reduce a composite implicit deadlock, i.e. remove some vehicles, to get a true implicit deadlock. Note that composite implicit deadlocks also might be part of a cycle. For example, imagine two vehicles are driving towards each other in a one-way tunnel while a third one drives around a roundabout. Given that the vehicles in the tunnel never drive towards each other, the vehicle in the roundabout can drive in circles forever such that these states form a cycle on the state graph. Additionally, if the tunnel vehicles drive and get stuck, the states move from a cycle of composite implicit deadlocks to a cycle of composite explicit deadlocks. It is only true implicit deadlocks that are never part of a cycle.

2.5.5 Example of true implicit deadlock

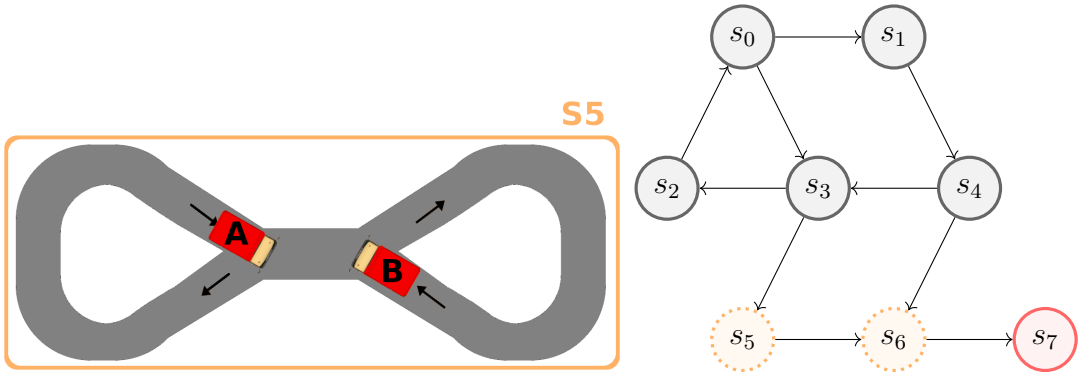


Figure 2.7: To the left, a true implicit deadlock $S5$ is visualised. To the right is an example of a state graph where state $S7$ is an explicit deadlock and $S5$ and $S6$ are true implicit deadlock.

Figure 2.7 shows a true implicit deadlock to the left and how it would look in the state graph to the right. As seen in the right image, nodes $S0$ to $S7$ represent different arrangements of vehicles in a site, i.e., different states of the site. States $S0$ to $S4$ are part of cycles, meaning they are not deadlocks. State $S7$, on the other hand, is in true explicit deadlock since there are no outgoing edges between vehicles in this state. States $S5$ (visualised in the left image) and $S6$ are not explicit deadlocks, but they will eventually end up in state $S7$ since there are no paths back to a safe state. States $S5$ and $S6$ are therefore *true implicit deadlocks*.

2.5.6 Classifying a deadlock

True and composite versions have been defined for both explicit and implicit deadlocks. The key difference between these two types is that a composite deadlock can always be reduced to a corresponding true deadlock. Why this fact is key will be explained in this section.

Imagine a site of 50 positions that fits a maximum of 10 vehicles, with the goal of finding all deadlocks. Suppose this site has some true explicit and implicit deadlocks which are known. Additionally, these deadlocks only use 2 vehicles. Since 10 vehicles might drive on this site, all composite deadlocks for up to 10 vehicles must be found to ensure the site never deadlocks. However, simply computing all composite deadlocks is computationally inefficient. Assume that the state d is a true deadlock of 2 vehicles (only occupying 2 positions) from the site. This leaves a choice of placing the remaining 8 vehicles over all 48 remaining positions. Excluding combinations with less than 10 vehicles, the number of composite deadlocks from d equals $\binom{48}{8}$. The number of composite deadlocks grows exponentially with the size of the site and the number of available vehicles.

Suppose there was some state s that should be classified. The system must keep track of a list of all composite deadlocks, and it must iterate entirely over it to confirm s is not a deadlocked state. Although, from the definition, all these composite

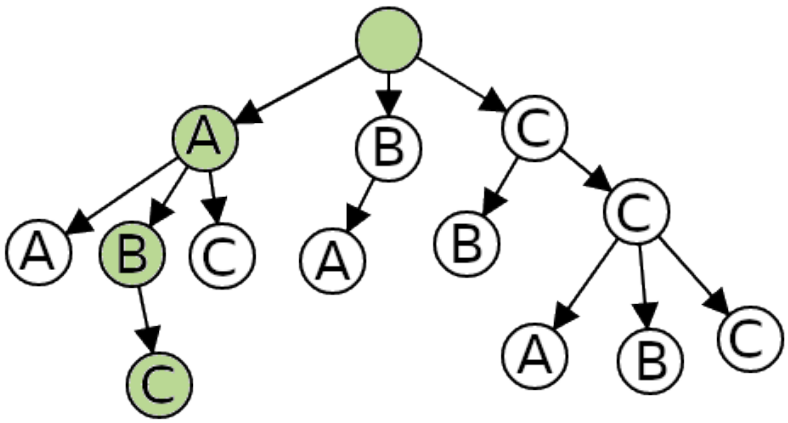
deadlocks can be reduced back to the state d. Instead of computing all composites and checking if s is one of them, one can check if s is either a true deadlock or if it can be reduced to one. This introduces the idea of a superstate.

Definition 2.5.6 (Superstate & substate)**.** Let V_s and V_d be the set of vehicles of state s and d, respectively. Then s is a superstate of d, and d is a substate of s, if $V_d \subseteq V_s$.

If s is a superstate of d, it is possible to remove vehicles from it such that it exactly becomes the state d. By only keeping track of all the substates, the true deadlocks, it becomes possible to know whether a state is a composite deadlock or a safe state, regardless of the number of vehicles involved.

2.6 Trie data structure

Trie, or *Trie Memory*, is a data structure used to store data consisting of sequences of symbols, such as words. In practice, a trie works like a set, allowing for fast insertion of new elements and quick lookup of inserted elements [12]. A trie has the form of a tree, very similar to a binary tree, but instead of each node having two children, each node of the trie has the same number of children as the number of possible symbols. All children are placed in an array which is indexed by the symbols. Children which has not been initialised are Null nodes.



Sequences Inserted
AA
ABC
AC
BA
CB
CCA
CCB
CCC

Figure 2.8: Example of trie data structure. Null nodes are excluded.

In Figure 2.8, a trie stores sequences of the symbols 'A', 'B', and 'C'. When looking up if a sequence is stored in the trie, one starts at the root node at the top and takes one step down the tree for each sequential element. Looking up the sequence "ABC", which is marked in green, can be done by going *Left*, *Middle*, *Right* in the tree, meaning that the sequence "ABC" has been inserted into the trie. However, if one looks up the sequence "ABB", one finds no 'B' inserted after "AB", and the sequence "ABB" is therefore not in the trie.

2.7 Neural networks

A neural network is a machine learning model that takes input data and generates an output. It does so by feeding the data through one or more layers of computation. One could describe a neural network as a replication of a mathematical function. However, instead of the parameters and constants of the function being known, they are learned by the network by looking at the input data and updating itself to produce the correct output.

As defined above, a neural network has input data and expected output from said data. The dimensionality of said data, and the desired output, vary greatly depending on what task the neural network is trying to solve. The data for a classification problem could be as simple as a vector of numbers for input, representing an image or other relevant data. The output is then defined as a vector where each element represents the probability of each class. Therefore, neural networks have an input layer, where data is fed, and an output layer, where the result can be read.

2.7.1 Perceptrons and dense neural networks

Neural network architectures usually have a hidden layer between the input and output layers. The hidden layer contains several perceptrons, or neurons as they are often called. The width of a layer describes the number of neurons that it has.

A neuron is a computational unit that takes every output from the previous layer and computes a new value. The computation for a dense neuron j and previous layers ith outputs can be described using equation 2.1.

$$z_j = \sum_i x_i w_{ij} + b_j \tag{2.1}$$

Each output x_i from the previous layer gets multiplied with a corresponding weight w_{ij} and summed together. Then, the bias scalar for neuron b_j gets added to the sum. Lastly, the results get fed through an activation function, which adds nonlinearity to the network and its possible outputs. Some common nonlinear functions used are ReLU, tanh or the sigmoid function.

$$y_j = activation(z_j) \tag{2.2}$$

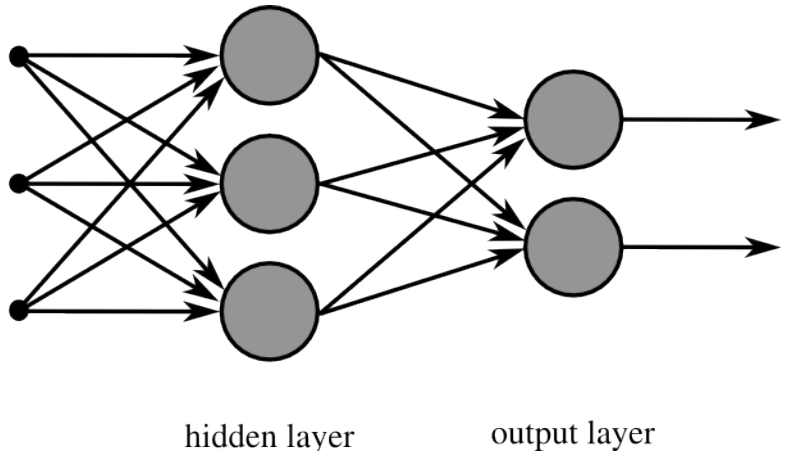


Figure 2.9: Dense Neural network example.

An example network with an input layer of 3 values, a hidden layer of 3 neurons and an output layer of 2 neurons can be seen in Figure 2.9. As can be seen, each neuron computes its input from every output in the preceding layer. This type of network is called a dense neural network. There are many other types of networks, such as convolutional neural networks, graphical neural networks or transformers.

2.7.2 Activation functions

Activation functions are used to normalise the output of a neural network model. They are simple mathematical functions which help the neural network to converge while training. It is called an activation function because it filters the output of each neuron. The ReLU function, for example, only outputs the given value if it is positive; otherwise, the value is 0 [13]. For this thesis, only the Softmax and ReLU functions will be used, but there are others such as Gudermannian, Leaky ReLU and many more. Sofmax and ReLU are defined as:

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & otherwise \end{cases}$$

$$Softmax(\overrightarrow{X}) = \frac{e^{\overrightarrow{X}}}{\sum_{j=1}^{K} e^{\overrightarrow{X}_j}} \text{ where } K = rank(\overrightarrow{X})$$

ReLU only returns x if x is positive, while Softmax normalises all values in a vector. The output of a single element from the Softmax function is between 0 and 1 and follows a Sigmoid curve, which can be seen in Figure 2.10.

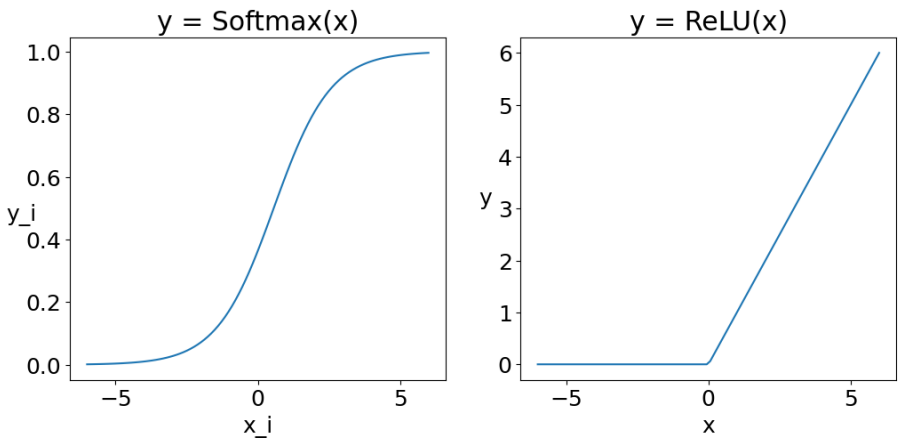


Figure 2.10: The Softmax and ReLU activation functions

3

Methods

This chapter presents the methods that can be used to prevent the traffic planning system from reaching a deadlocked state. This is done through three separate systems, finding all true explicit deadlocks possible for a site, finding all true implicit deadlocks that derive from the explicit ones, and constructing a data structure that allows quick verification if some state in question is a deadlock or a superstate of one. These steps entirely prevent deadlock, as described in Section 2.5.6. A flowchart displaying the methods how they interact can be seen in Figure 3.1.

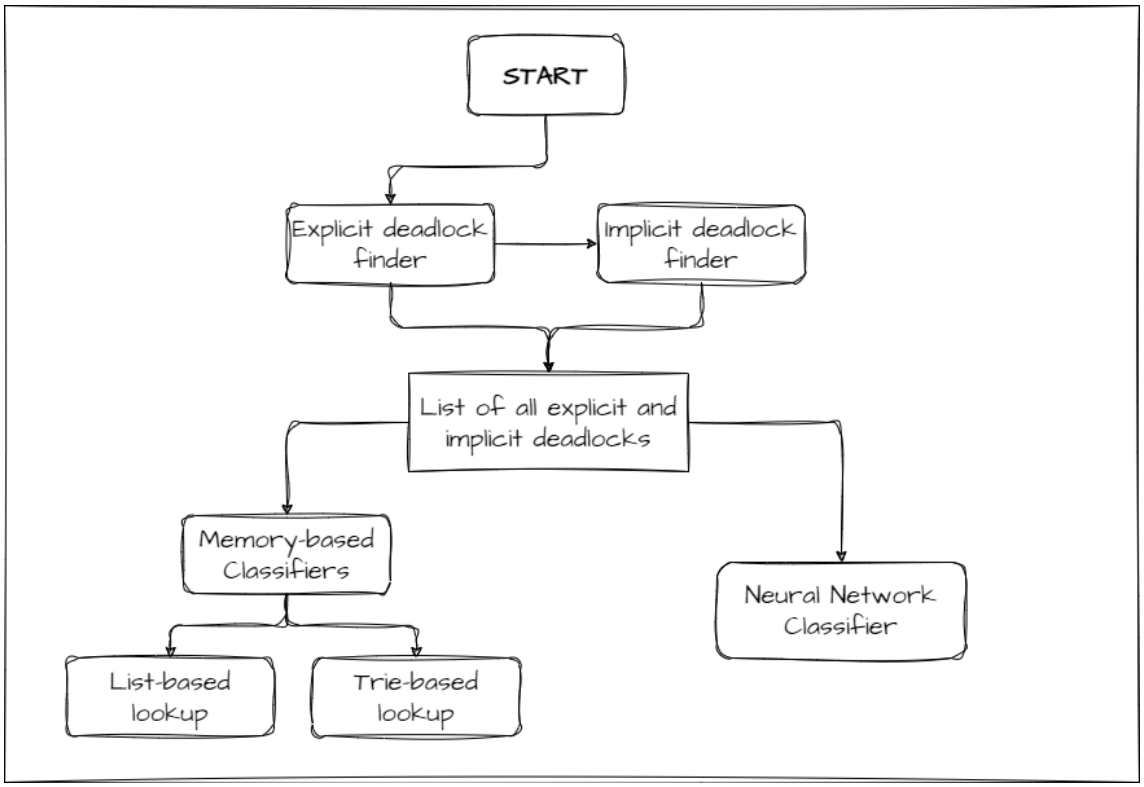


Figure 3.1: Flowchart describing the different components developed and their relation.

3.1 Explicit deadlock finder

The first component of solving the problem is to find all possible explicit deadlocks for a given site. To accomplish this, the explicit deadlock finder places some initial vehicle on the site, then tries to add more vehicles such that the most recently placed one gets blocked from moving. This method finds all true explicit deadlocks from which the composite ones can be derived. A flowchart of the explicit deadlock finder can be seen in Figure 3.2 and the complete pseudo code of it can be read in Appendix A.5.

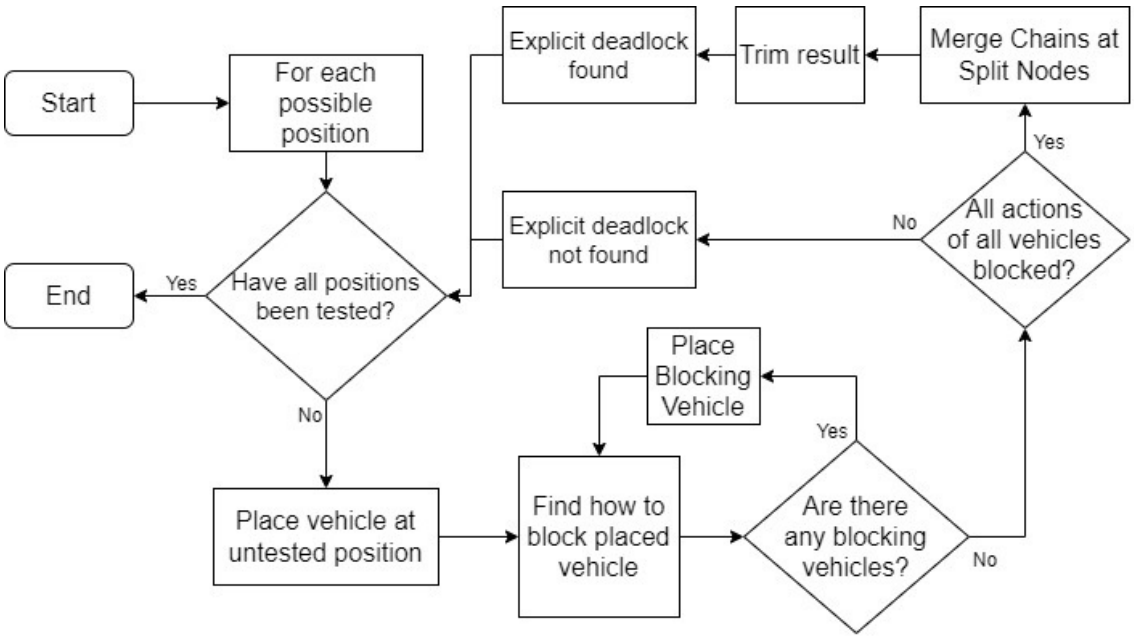


Figure 3.2: Flowchart describing the explicit deadlock finder.

The finder start by computing all possible positions of a site (see Definition 2.2.1) and the claims for each position (see Definition 2.3.1). Then, from each position, it tries to find chains of blocking vehicles which form an explicit deadlock when combined. The following subsections will explain all the algorithm's main parts.

3.1.1 Finding blocking vehicles

For a given vehicle v, a blocking vehicle u is another vehicle that prevents v from driving. The system finds possible positions of u by first checking what new nodes would have been claimed, for a given action from the tail node of v. A blocking vehicle u is then one that claims at least one of these new nodes.

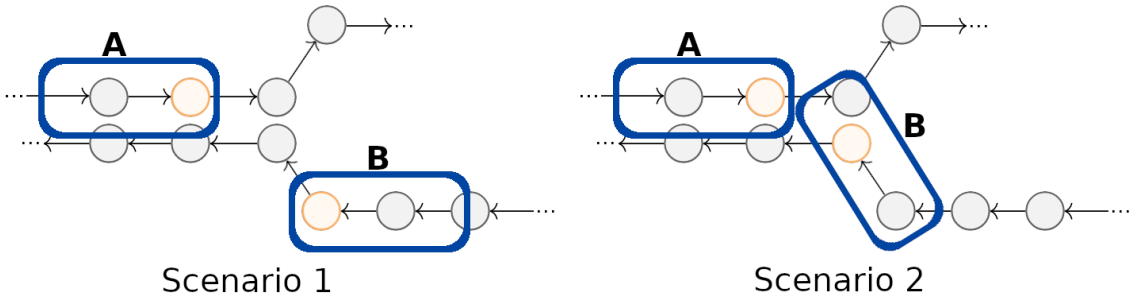


Figure 3.3: Scenario 1 and 2 of vehicles blocking each other, the orange nodes are the locations of the vehicles and the blue squares are the area which the vehicle claims

For an example of what a blocking vehicle looks like, see Figure 3.3. The figure shows scenarios 1 and 2, where vehicle A wants to move to the right and vehicle B wants to move to the left. The blue squares show the physical space that each vehicle claims. In scenario 1, vehicle A can continue driving up and beyond without colliding with another vehicle. In Scenario 2, on the other hand, neither vehicle A nor B can drive forward since vehicle A is *blocked* by vehicle B and vehicle B is *blocked* by vehicle A.

3.1.2 Finding chains of blocking vehicles

As described in Definition 2.5.2, a true explicit deadlock is a state where each vehicle has all its actions blocked by other vehicles. This means that there exists a chain of vehicles that all block each other sequentially. The finder's goal is to find all these possible chains.

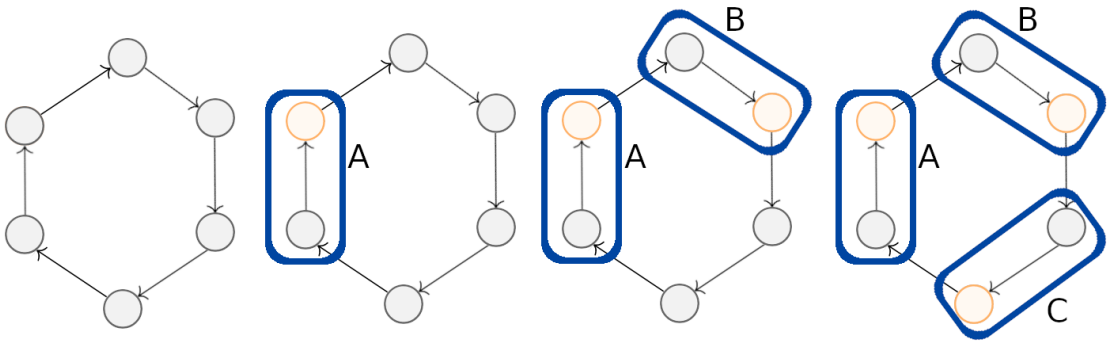


Figure 3.4: Example of finding a explicit deadlock

The finder picks one position on the site and places a vehicle A there. Then, it finds all vehicles that could block A from driving to one of its destinations and recursively places down another vehicle B at every possible blocking position. Then, repeats the process for B until no more vehicles can be placed down, as seen in Figure 3.4.

However, the finder can not place a vehicle that overlaps with the claims of other vehicles. When no more vehicles can be placed down, two alternatives exist. Either the actions of all vehicles have been blocked, meaning a cycle of blocking vehicles

has successfully been created. This can be seen in Figure 3.4, where vehicles A, B, and C block each other. Alternatively, no more vehicles can be placed down, but at least one can move. This could have happened in the example above if there had been one more node between the head of C and the tail of A. If so, there would not be enough space to place down another vehicle to block the path of vehicle C, meaning the state is not an explicit deadlock.

3.1.3 Combining returned chains of vehicles

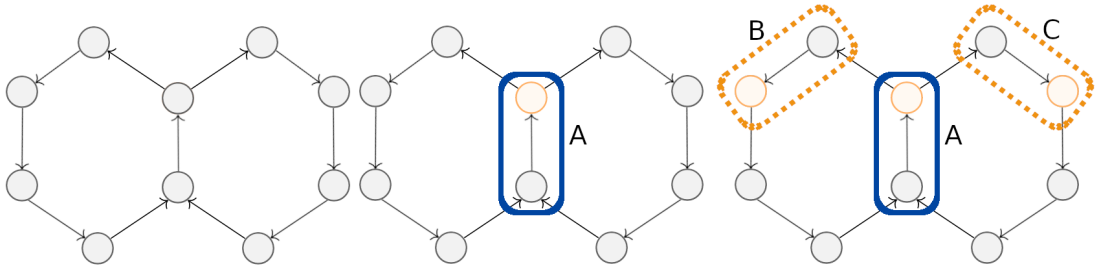


Figure 3.5: Example of finder encountering split node

When placing down a vehicle, the vehicle might have multiple ways of moving. As seen in Figure 3.5, vehicle A could go either left or right, both needing to be blocked for vehicle A to be blocked. In this case, the finder explores how to block each action individually and then merges the resulting chains of blocking vehicles.

3.1.4 Trimming deadlocks

In many cases, the result after 3.1.3 will result in composite deadlocks, i.e., deadlocks where vehicles can be removed and still be deadlocks.

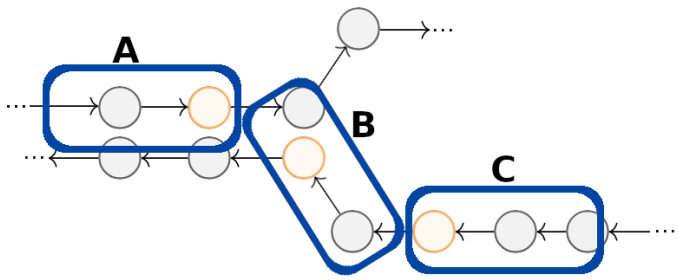


Figure 3.6: Example of composite deadlock with three vehicles A, B and C

To reduce these states to true deadlocks, the vehicles which can be removed must be removed. As seen in Figure 3.6, vehicles A, B and C are all stuck. But since vehicle C is not blocking any other vehicles, it can be removed, resulting in a state with only vehicles A and B which is a true deadlock. This action can be performed iteratively, removing each vehicle which does not block any other vehicle until no more vehicles can be removed.

3.2 Implicit deadlock finder

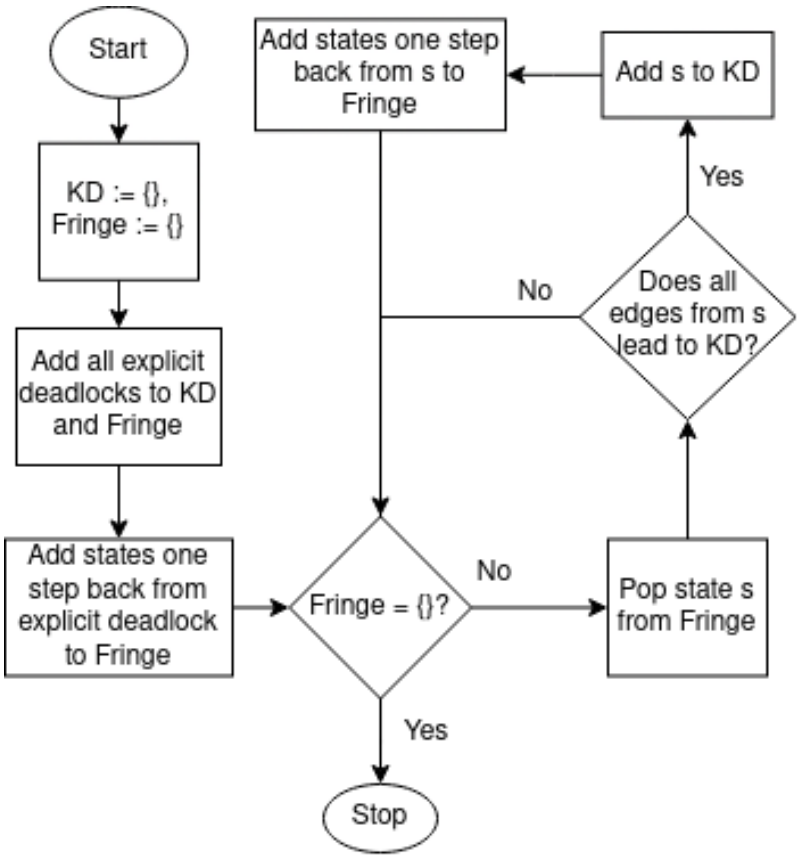


Figure 3.7: Flowchart of the implicit deadlock finder method

The second component to solving the problem is to find all true implicit deadlocks, which is done by the implicit deadlock finder, visualised with a flowchart in Figure 3.7. It starts by receiving all explicit deadlocks for a given site and then stores them in a set called KD (Known Deadlocks). Then, it follows edges backwards in the state graph (see Definition 2.4.1), starting from each explicit deadlock and adding the previous states to a set called Fringe. When all previous states have been added to the Fringe, it picks the top state of the Fringe and checks if all possible actions from said state lead to deadlocks already in KD. If this is true, then by definition, the current state is a true implicit deadlock. In that case, add it to KD and add all its previous states to the bottom of the Fringe. If all possible actions do not lead to known deadlocks, do nothing. Lastly, pick another state from the Fringe and repeat the process until the Fringe is empty. In essence, this algorithm explores the state graph of a site and finds states that are guaranteed to be implicit deadlocks. At least one state can always be added to KD, a proof of which is described in Appendix A.1. Lastly, for details on how the implicit finder algorithm works, see the pseudocode in Appendix A.6.

A key detail about this algorithm is that it must be done in order of the explicit deadlocks, sorted by the number of vehicles involved. For example, to find implicit deadlocks for 5 vehicles requires that all implicit deadlocks for 2, 3 and 4 vehicles

have already been found. As such, one has to run the implicit finder on all explicit deadlocks with 2, 3 and 4 vehicles before starting on ones with 5. This requirement comes from the fact that a state might be a superstate of a smaller deadlock (see Section 2.5.6). For example, say there exists two explicit deadlocks, one with 2 vehicles called A and one with 6 called B. When exploring the previous states of B, one of these states might lead to a composite implicit deadlock based on A. Since only true deadlocks are stored, the only way to know if this state is a deadlock would be to find it when exploring behind A. All implicit deadlocks that originate from A must be explored and added to KD before the exploration of B begins. This is done in the finder by separating explicit deadlocks by their size and finishing exploring some of a certain size before moving on to the next one.

3.2.1 Example of one implicit finding step

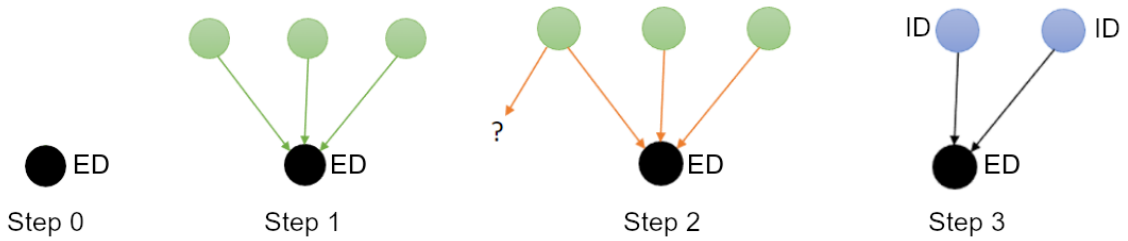


Figure 3.8: Example of finding implicit deadlocks one transition from explicit deadlock, explicit deadlocks marked with ED and confirmed implicit deadlocks marked with ID

An example of the three steps of implicit finding can be seen in Figure 3.8. As previously described, the search works by looking backwards at prior states, looking forwards at the next states, and adding any states which can be verified as implicit deadlocks to the set of known deadlocks.

In this example, there is only one explicit deadlock at step 0, marked by the black circle. Step 1 shows that this state has 3 prior states, marked in green. As such, they get added to the fringe. Then in Step 2, every next state from the green states gets examined. All states lead back to the explicit deadlock, but the left state also leads elsewhere. Since this state is not in the known deadlocks, one can not be sure that the left state is an implicit deadlock. Therefore, as seen in Step 3, only the two states marked in blue get added to the list of known deadlocks. To find more implicit deadlocks, the finder goes back to step 0. But, instead of examining states one step back from the explicit deadlock, it examines states one step back from the newly found implicit deadlocks. This is repeated until no more new implicit deadlocks are found.

3.3 Memory-based deadlock classifier

Using both finders, finding all deadlocks for a site is now possible. However, the traffic planner still needs an efficient way to verify whether a specific state s is a deadlock. Therefore, a memory-based method of verification was developed. This section describes two variations of such a method. For both methods, it is assumed that *all* true implicit and explicit deadlocks have been found in the given site. The task is then to determine whether the current state matches a deadlock. As described in Section 2.5.6, this can be done by checking if the current state is a superstate of all known true explicit or implicit deadlocks.

3.3.1 Lookup using masking

A simple lookup can be done in linear time, $O(n)$, where n is the number of deadlocks in memory. Let s be the site state represented by a binary number where a 1 or 0 at bit i means there is or there is no vehicle at position i in the site graph. Let $D = \{d_1, d_2, ..., d_n\}$ be our memory set of deadlocks where each d_j is a binary number, each representing a site state where all vehicles are in a deadlock. The "bitwise and" operation (&), executed in constant time, can be used to compare each deadlock with the site state. If all bits in d_j are also set in s, it holds that $d_j \& s = d_j$, that is, some group of vehicles in the site state forms a deadlock. So, if there is a deadlock, then $\exists d \in D : d \& s = d$, which our method validates by simply looping over each deadlock in memory. See pseudocode below:

Require: s is a binary representation of site state
Require: D is the set of all deadlocks
  for d in D **do**
    if $d \wedge s == d$ **then**
      return that s is a deadlock
    end if
  end for
  return that s is not a deadlock

3.3.2 Lookup using trie data structure

A *trie memory*, as described in Section 2.6, is suitable for finding if a state matches or is a superstate of any deadlock. The trie can check multiple deadlocks simultaneously, lessening computation time per lookup. Normally, a trie is used to find an exact match in a set of sequences. As such, it would only be able to find true explicit deadlocks. However, by modifying the lookup process, one could find if the given sequence is a superset of any inserted sequence.

As in Section 3.3.1, let s be the binary representation of a site state. s can then be inserted into the trie memory as a sequence of 1s and 0s.

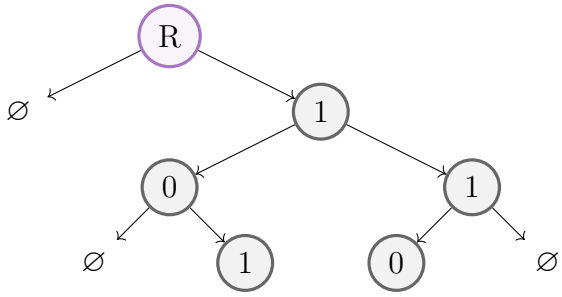


Figure 3.9: Trie example with two inserted deadlocks 110 and 101

As seen in Figure 3.9, two states have been inserted: 110 and 101. These are both true deadlocks. 0s are inserted to the left, and 1s are inserted to the right. When looking up if a sequence has been inserted into the tire, one traverses the trie from the root R and goes left or right if the given element in the sequence is 1 or 0. The given sequence has been inserted if and only if a trie has been inserted at each step. For example, looking up sequence 111 in Figure 3.9, one first goes right twice before finding no more tries inserted to the right. 111 has not been inserted into the trie and is, therefore, not a true deadlocked state.

Table 3.1: Examples of states which are looked up in the Trie in Figure 3.9

Sequence	Is a superset	Comment
111	Yes	Is superset of 110 and 101
110	Yes	Matches 110
101	Yes	Matches 101
100	No	
011	No	

When looking up if sequence s is a superset of any inserted sequence, one traverses the trie in a similar manner, but when encountering a 1, one explores both left and right sides. If one wants to check if 111 is a superset of any inserted sequence, after two steps right, one more step right is impossible, but one could explore left instead, resulting in a sequence match. The matched sequence is 110, which is a subset of 111. As such, 111 is a composite deadlock of the true deadlock 110. Table 3.1 shows a number of examples of whether a sequence is a subset or not.

Compared to using masking, as in Section 3.3.1, a trie can check multiple positions at the same time. At the root node, the trie looks at the first position for *all* deadlocks and sees if there is a deadlock with a vehicle there or not. This is much better than a list, which needs to individually check the first position for each deadlock. While the performance lessens when searching deeper into the trie, it is as good as the list method at worst. Pseudo code of inserting and checking if the trie contains a state (if the state is a superstate of any inserted state) can be seen in Appendix A.7.

3.3.3 Implementing the deadlock constraint

With a deadlock classifier constructed, constraints for preventing deadlock could now be implemented and connected to the existing planning system at VAS. When the planner wants to know if a state is a deadlock, the classifiers use lookup and subset checking to return the correct answer. Both classifying methods were implemented, and their results can be seen in Chapter 4.

3.4 Neural networks

Using the methods described above, one could determine whether a given site state is a deadlock. This information is then used to train a neural network to validate if it can comprehend what a deadlock is and if such a network can be used to find deadlocks that have not been found before.

3.4.1 Dataset

A dataset was first generated by randomly sampling a large set of valid site states. These states were then labelled as ”Not deadlock”, ”Explicit deadlock”, or ”Implicit deadlock”. It was found that most states were classified as ”Not deadlock’,’ which required the dataset to be rebalanced. If not, any classifier could label all explicit and implicit deadlocks as ”Not deadlock” and receive a high accuracy rating. However, the explicit and implicit deadlock accuracy would be very bad.

To balance the three classes, all explicit and implicit states in the training dataset were duplicated, so each class had the same number of states. When testing, the ratios of deadlock classes were left untouched.

3.4.2 Dense neural network

Using the dataset from Section 3.4.1, a dense neural network was constructed for classifying the different states. The input of this network was a vector with n input nodes. Each node corresponded to a position in the site graph and had a value of 1 or 0 if a vehicle occupied the given position or not. The output vector had a dimension of 3 corresponding to the probability of ”Not deadlock”, ”Explicit deadlock” and ”Implicit deadlock”.

$$f_1(X) = Softmax(XW^{(0)}) \tag{3.1}$$

$$f_2(X) = Softmax(ReLU(XW^{(1)})W^{(2)}) \tag{3.2}$$

The model of each site, except for Sharp Site 2, resembles Equation 3.1, where the weight matrix $W^{(0)}$ has the dimensions of $n \times 3$. Although this is a simple model, it was sufficient for high-accuracy results. For Sharp Site 2, the model described in Equation 3.2 was used. It has one hidden layer of size 256, i.e. $W^{(1)}$ has the dimensions 538×256 and $W^{(2)}$ has the dimensions 256×3.

3.4.3 Classifying unknown deadlocks

One potential advantage of a neural network classifier is that it might develop a general intuition of what positions are involved in a deadlock. If so, one could generate training data representing a subset of all explicit and implicit deadlocks, train the model using this data and then find previously unknown deadlocks. Such a method would be especially useful if smaller deadlocks with few vehicles could be used to train a classifier that then learns to find much larger ones. The computational effort needed to find all deadlocks would greatly decrease, as this classifier could disregard potential exponential behaviour in generating all possible deadlocks of many vehicles.

A dataset, as described in Section 3.4.1, was generated to test if the neural network could generalise classifying deadlock. Each sampled state in this dataset had at most 7 vehicles, and was then used to train a neural network model. However, a dataset where each state had 11 vehicles was used when validating the model. If the model can generalise, it should be able to correctly classify new states that are true deadlocks and not found in the training dataset.

4

Results

This chapter describes the results found by applying the methods described in the previous chapter on various sites. Additionally, the problem with finding *all* deadlocks will be expanded upon.

Data was created for a site using the explicit and implicit finders (see Sections 3.1 and 3.2), and the limits of such a method were explored. This data is directly used in the memory classifying methods, which means they could be evaluated. Then, the data was used to train and test a dense and graph neural network to see if they were any better than a strictly memory-based method.

4.1 Description of sites

There were a total of four sites used. These were split into test sites and Sharp sites. The test sites, which are purely conceptual, were the Screwdriver and Oval sites. The Sharp sites, which are based on sites from Volvo customers, are called Sharp Sites 1 and 2. Table 4.1 shows a table containing statistics for each site. These numbers show the largest number of deadlocks that have been found, for the largest number of vehicles it is possible to generate for. Sites where all deadlocks could not be found in a feasible amount of time have a star next to the number of deadlocks. As to why all deadlocks can not be found, see Section 4.6. Additionally, different examples of deadlocked states can be seen in Appendix A.8.

Table 4.1: Table of nodes and deadlocks per site. * denotes that not all deadlocks have been found, and that this is the highest amount that have been computed to completion.

Name	Nodes	Explicit Deadlocks	Implicit Deadlocks
Screwdriver	44	5	76637
Oval	47	4	2*
Sharp 1	42	7	80812
Sharp 2	538	164*	24825*

The Screwdriver site is the simplest test site and the first site where the developed methods were tested. The site graph can be seen in Appendix A.8. It is a simple site with no node nor deadlock amount to be considered difficult to solve. However,

it proved a simple start to verify that the methods worked. On this site, there are only 29 deadlocks for reasonable amounts of vehicles. However, for over 10 vehicles, most states become implicit deadlocks which is why the number of deadlocks in very high in Table 4.1.

Oval is also a test site and similarly has very few nodes. The site graph can be seen in Appendix A.9. However, it introduces nodes where the path splits or merges, can be more difficult for the system than the deadlocks in Screwdriver. Furthermore, it introduces deadlocks at its path merging nodes where turning the vehicle could result in a collision, another type of deadlock that had to be found. Although, there appears to be a very large number of deadlocks for 12 vehicles or more, as computation time grows very fast (for more, see Section 4.4.1).

The first real site is called Sharp Site 1, which a Volvo client owns. It can be seen in Appendix A.10. It is about the same size as the test sites but has nodes very close to each other. It is similar to Screwdriver, where the amount of deadlocks only become significant for 12 vehicles. Additionally, it is possible to compute all deadlocks in a reasonable amount of time for this site.

Finally, there is Sharp site 2, which a Volvo client also owns. This site is much larger than all the other sites and can be seen as a difficult real-world example at which the fleet planning system could be deployed. Appendix A.11 shows a map of it. It has a total of 534 nodes, so computing the number of deadlocks on this site was very difficult for a large set of vehicles. Furthermore, it has many one-vehicle tunnels, which create many true deadlocks with only a few vehicles. Thereby, there are an enormous amount of true implicit deadlocks as the number of vehicles increases. As such, while the number of implicit deadlocks in Table 4.1 seems low, it is because the computation grows exponentially and does not finish, not because of a lack of deadlocks.

4.2 Explicit finder results

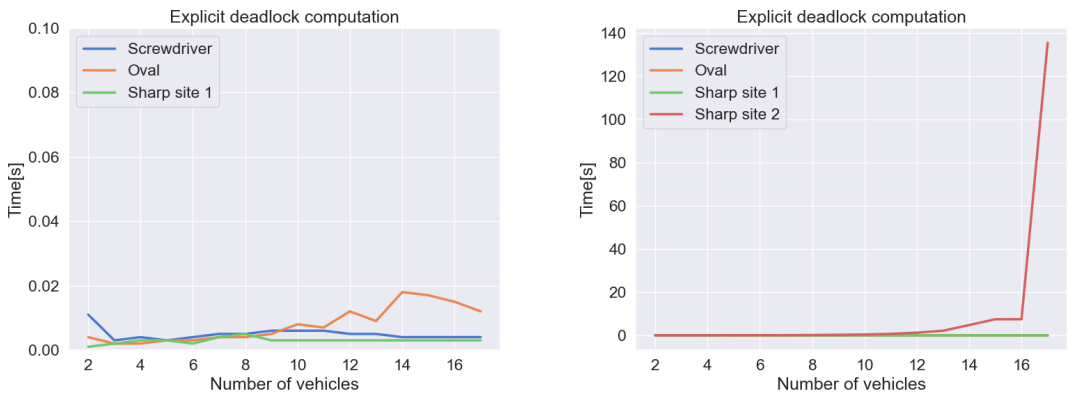


Figure 4.1: Computational time of explicit deadlock for the four different sites, the right plot is the same as the left but includes Sharp Site 2 which requires adjusting the y-axis.

This section describes the time it takes to compute all explicit deadlocks up to 17 vehicles involved. All results can be seen in Figure 4.1. As can be seen, due to the size of the sites, computing deadlocks for all sites except Sharp site 2 takes a small amount of time. As such, when sites grow in nodes, so does the time to compute explicit deadlocks. The results for Sharp site 2 show that the time it takes to compute explicit deadlocks can be exponential regarding the number of vehicles. Computing explicit deadlocks on Sharp site 2 for 18 vehicles fails to finish as the computation takes too long. As such, it is obvious that computation time depends highly on the site. Finding explicit deadlocks will eventually become a problem if larger or more complex sites are attempted.

4.3 Comparing memory-based classifiers

Two methods were used for the memory-based classification of states: a list or a trie of known deadlocks. Compared to the set of deadlocks, these methods check if a state is a superstate (see Definition 2.5.6). The lookup time of both methods was compared by inserting an increasing amount of deadlocks into the data structure and then looking up 10 000 random states. Note that the same random states were used in each iteration, i.e. list and trie lookup was tested on the same data. As such, this tests shows how lookup time increases as the size of the data structure increases.

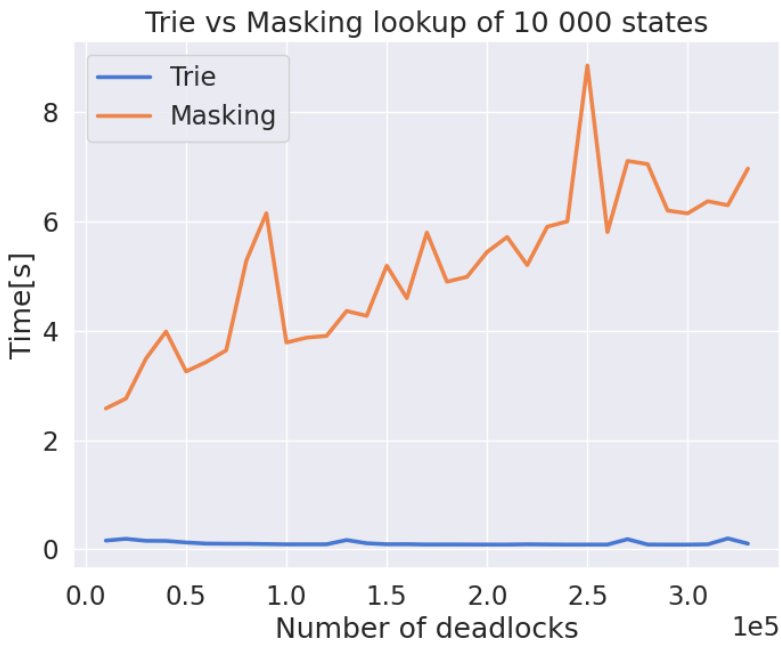


Figure 4.2: Comparing deadlock lookup time between trie and masking. X-axis shows number of deadlocks in data structure and y-axis shows lookup time.

The results can be seen in Figure 4.2, which displays graph over the number of deadlocks in the data structure on the x-axis and the lookup time for 10 000 random states on the y-axis. As seen, the masking method roughly follows a linear curve

related to the number of deadlocks. On the other hand, the trie data structure's lookup time is almost constant, with an average lookup time of 85 milliseconds. Tries are, therefore, an immediate improvement over using a list. As such, trie-based memory classification was the only memory classifier used from this point on.

4.4 Implicit deadlock finder

Since the lookup time of the trie classifier is very low, the factor that controls the classifier's performance is the time it takes to find all implicit deadlocks. As such, the following section will detail computation times for the test and Sharp sites. A search has been run for each combination of depth and max number of vehicles. As a reminder, depth is the maximum distance from an explicit deadlock that the implicit finder considers. It decides how many iterations of finding previous states that are allowed. Depth was used to constrain the search such that results could be generated beyond the normal limits of the implicit finder (i.e. searching where the time starts becoming exponential).

4.4.1 Test sites

For Screwdriver, Figure 4.3 shows the time it takes to find all implicit deadlocks. The site was tested for up to 13 vehicles and a generous depth of 100. This site fits a maximum of around 14 vehicles, and finding deadlocks for most choices of vehicles on site is simple, as there are only 29 implicit deadlocks for 10 or fewer vehicles. However, the computational time grows when searching for deadlocks with more than 10 vehicles. That is because many sites states on Screwdriver with more than 10 vehicles are implicit or explicit deadlocks, i.e., there is no way for 11 vehicles to move in a loop on Screwdriver. Therefore, the algorithm will list all valid states with 11 vehicles and mark them as deadlocks. Although, as seen in the figure, computation time stops growing when all deadlocks have been found.

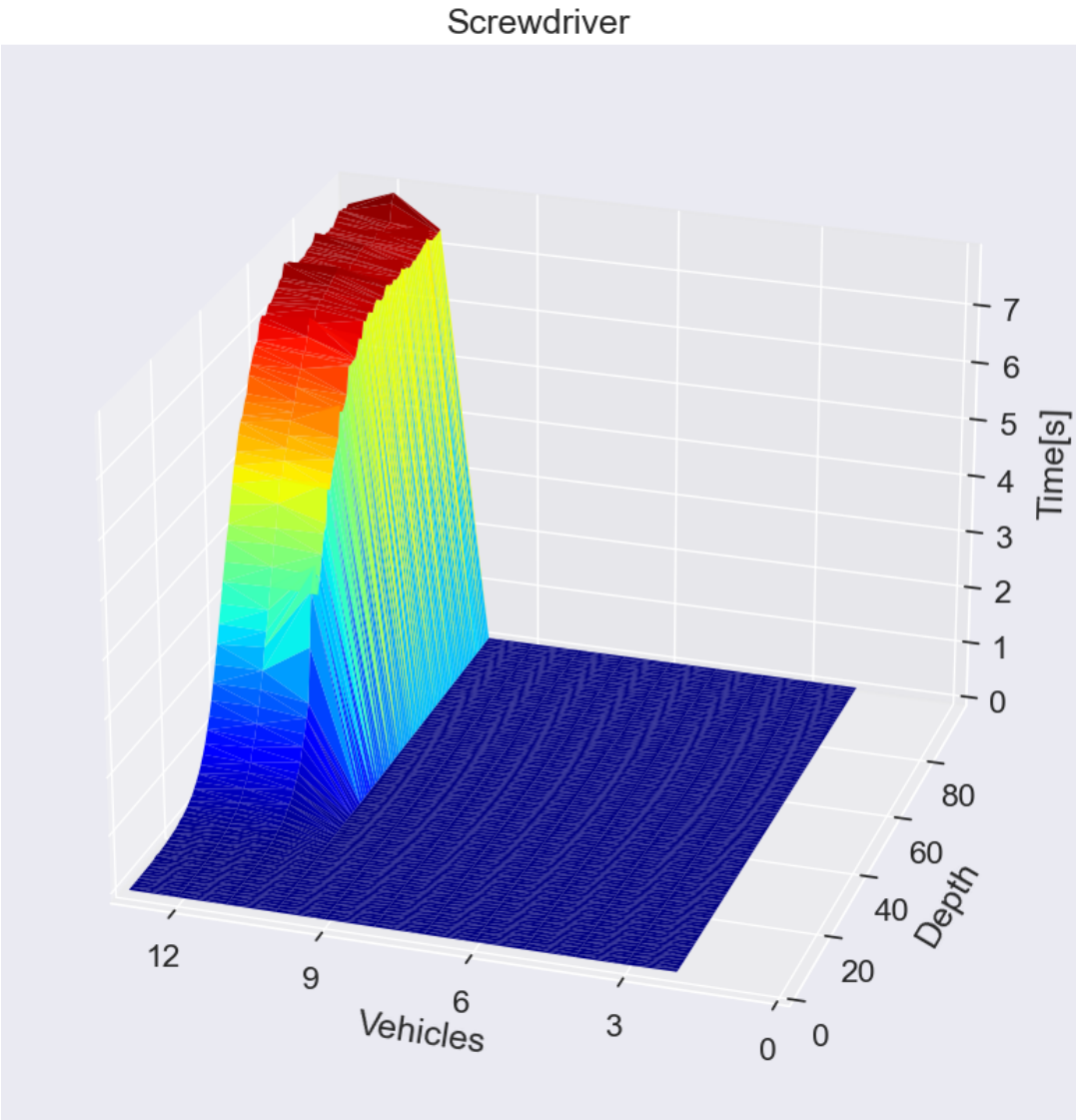


Figure 4.3: Graph over Vehicle/Depth/Time (S) of Screwdriver site. Generated for max 13 vehicles and 100 depth.

Figure 4.4 displays the time it takes to find all implicit deadlocks for the Oval site. Like Screwdriver, this site only starts becoming unfeasible to test around the maximum number of vehicles, which is 14. As can be seen, most combinations take less than a second. The rise in time starts around 12 vehicles, and the slope of the curve can be seen to be exponential. Computation time quickly goes from less than a second to over 100 seconds. This means continuing testing for deeper depths at 12 vehicles or higher was impossible. Most states of 12 vehicles are believed to be deadlocks, explaining why the computation grows so quickly.

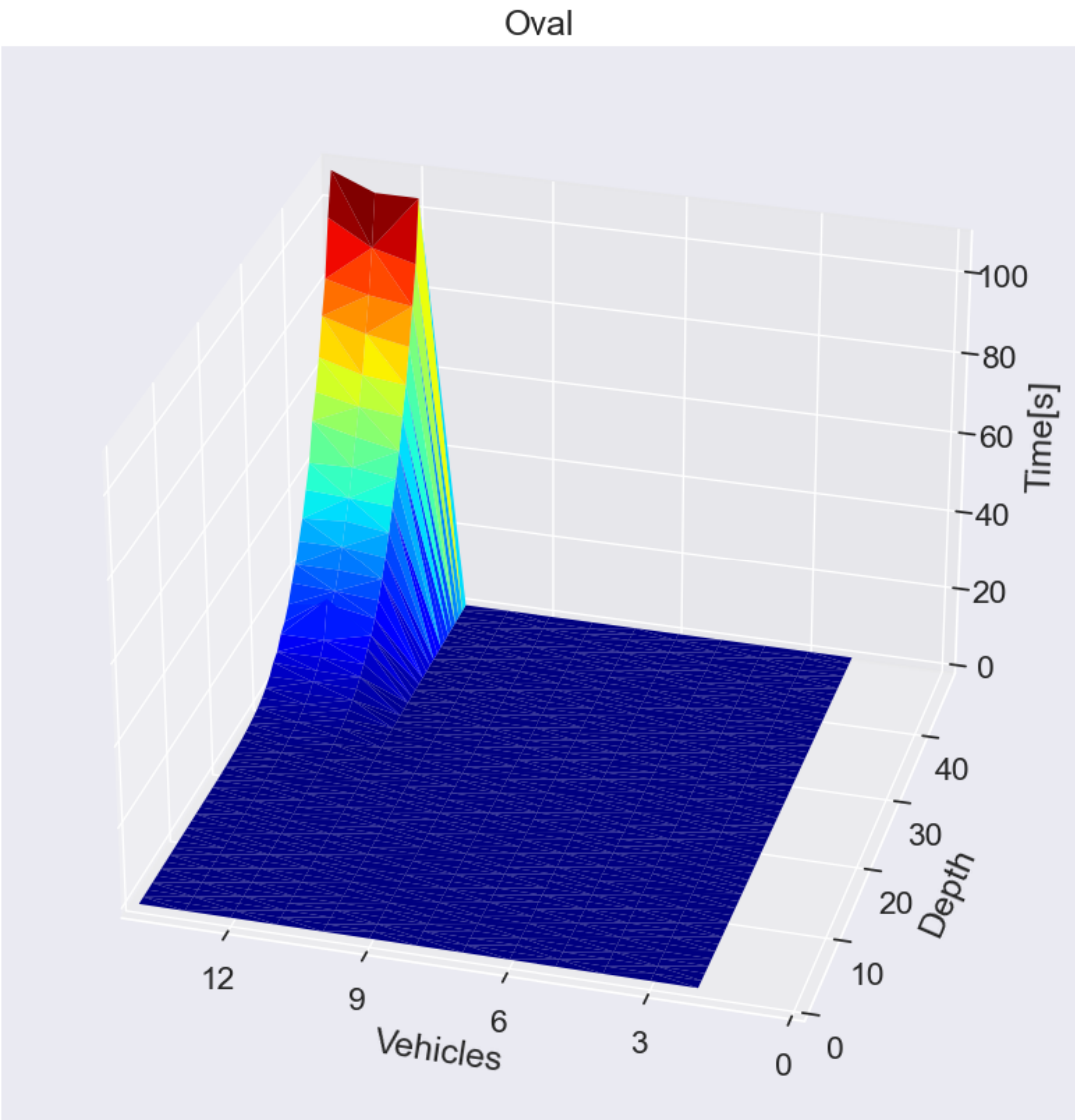


Figure 4.4: Graph over Vehicle/Depth/Time (S) of Oval site. Generated for max 14 vehicles and 50 depth.

4.4.2 Sharp sites

Results for Sharp site 1 can be seen in Figure 4.5. As mentioned, the figure looks similar to Screwdriver but has linear growth instead of an "S" shaped curve. It is worth noting that all deadlocks have been found at 12 vehicles and about 450 depth, where the slope in the figure stops growing. As such, the trie classifier can prevent deadlock for any valid amount of vehicles on this site (as long as they start in a safe state).

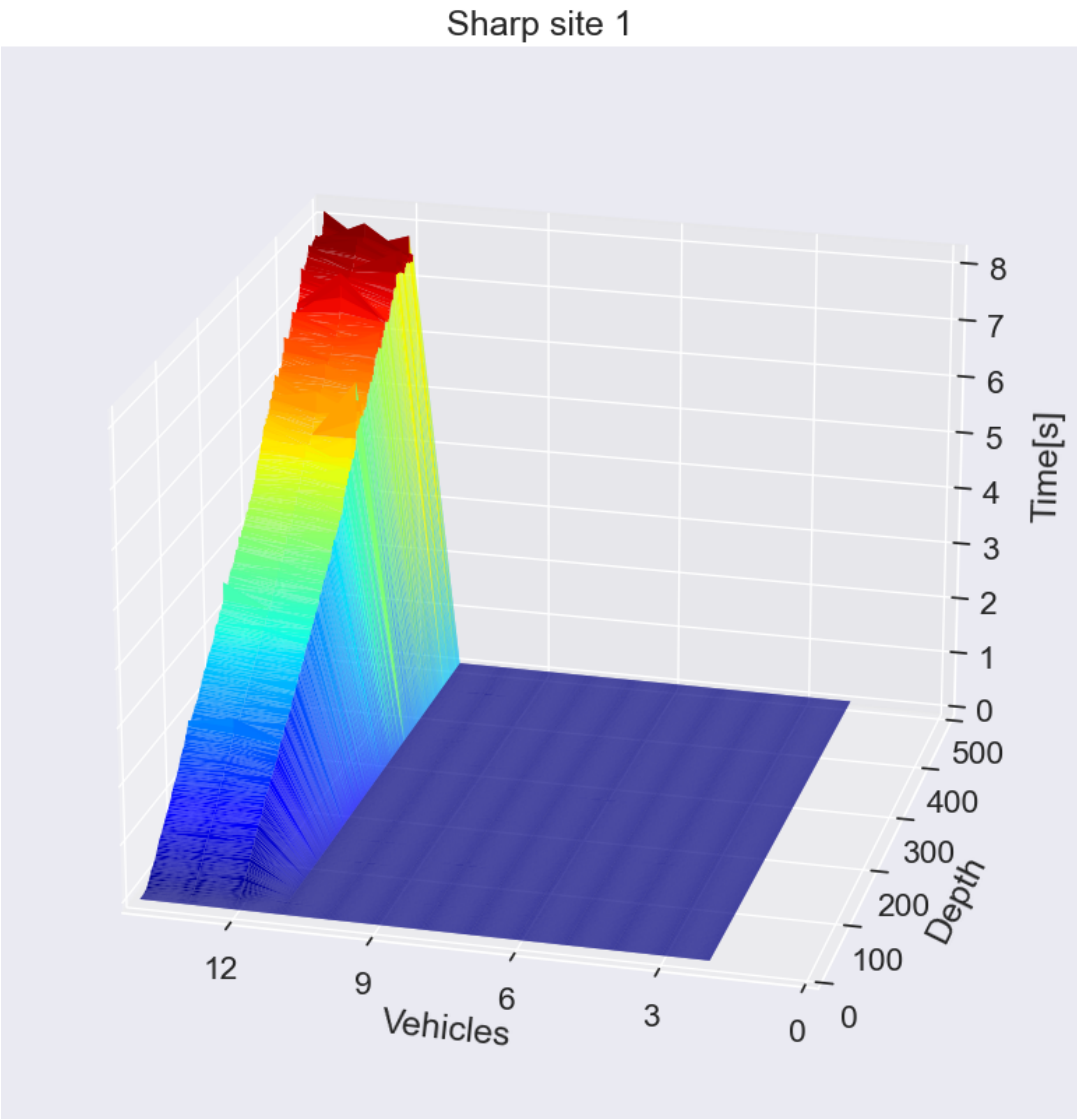


Figure 4.5: Graph over Vehicle/Depth/Time (S) of Sharp site 1. Generated for max 14 vehicles and 500 depth.

Lastly, the results for Sharp site 2 can be seen in Figure 4.5. This is a very large site where the exponential behaviour at higher vehicle counts is at its worst. As can be seen, for 16 vehicles, the time to test goes over 5 minutes. No depth higher than 25 is possible at this point, as the computation has not managed to finish using available resources. It is worth noting that around 12 vehicles currently drive on this site, meaning it is possible to prevent deadlock as it currently stands. However, this is the maximum that the implicit finder can handle. Why this problem occurs has been further explained in Section 4.6, and ideas for solving it can be found in Section 5.3.

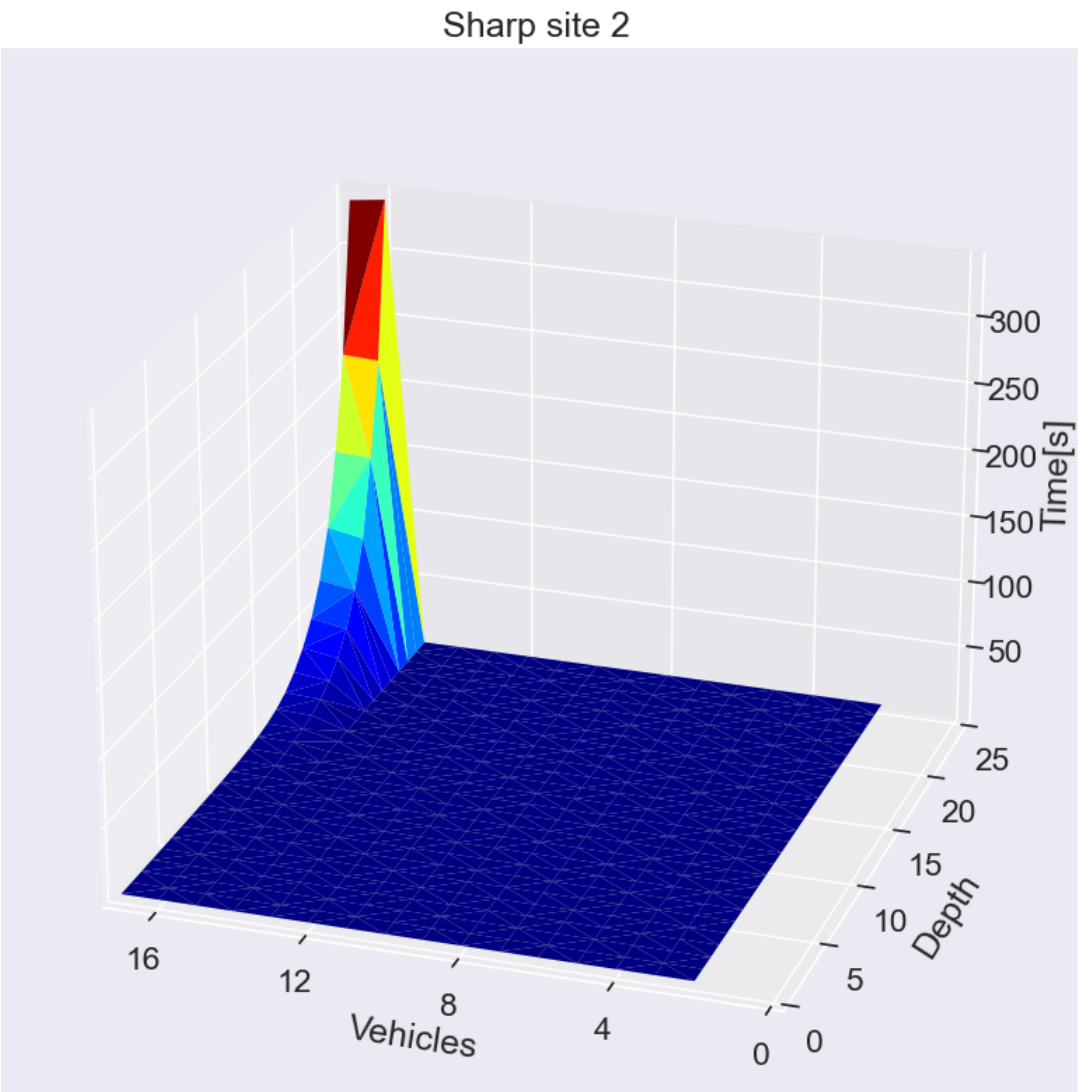


Figure 4.6: Graph over Vehicle/Depth/Time (S) of Sharp site 2. Generated for max 17 vehicles and 25 depth.

4.5 Neural network classifiers

The neural network classifiers are trained on the same data the memory classifiers use. These classifiers aim to generalize the process of classifying deadlocks for any number of vehicles. One model was built and trained for each site. Each dataset had at most 7 vehicles in each sample state.

Table 4.2: Results from Dense Neural Network Classifier

Site	Train Samples	Test Samples	Class Dist. (ND, ED, ID)	Accuracy
Screwdriver	160 000	40 000	(75%, 4.2%, 21%)	97.0%
Oval	160 000	40 000	(96%, 2.3%, 1.9%)	98.0%
Sharp Site 1	400 000	100 000	(96%, 2.4%, 1.9%)	96.6%
Sharp Site 2	1 800 000	200 000	(85%, 1.1%, 14%)	97.0%

As can be seen in Table 4.2, classifier accuracy at each site was very high at above 96%. The main reason why the classifiers did not get a better accuracy was that there where some confusion between explicit and implicit deadlocks. Therefore, if only ”Not Deadlock” and ”Deadlock” were used, the classifiers had an accuracy of 99.9%.

Due to the uneven class distribution between ”Not Deadlock”, ”Explicit Deadlock” and ”Implicit Deadlock”, as can be seen in the second column from the right in Table 4.2, the training samples for explicit and implicit deadlocks where duplicated such there was an equal 33% split between the three classes in the training dataset. No states were duplicated for the test dataset, but the samples were weighted. As such, if the classifier had marked all test samples as ”Not Deadlock”, it would only have a 33% accuracy.

4.5.1 Classifying unknown deadlocks

A dataset of 10 000 unique random states at Sharp Site 1 was generated, containing 11 vehicles each. With at most 7 vehicles, there exist 5 true explicit deadlocks and 137 true implicit deadlocks, while with at most 11 vehicles, there exist 6 true explicit deadlocks and 172 true implicit deadlocks. So, among the 10 000 generated samples, there are several composite explicit and implicit deadlocks the model had not seen before.

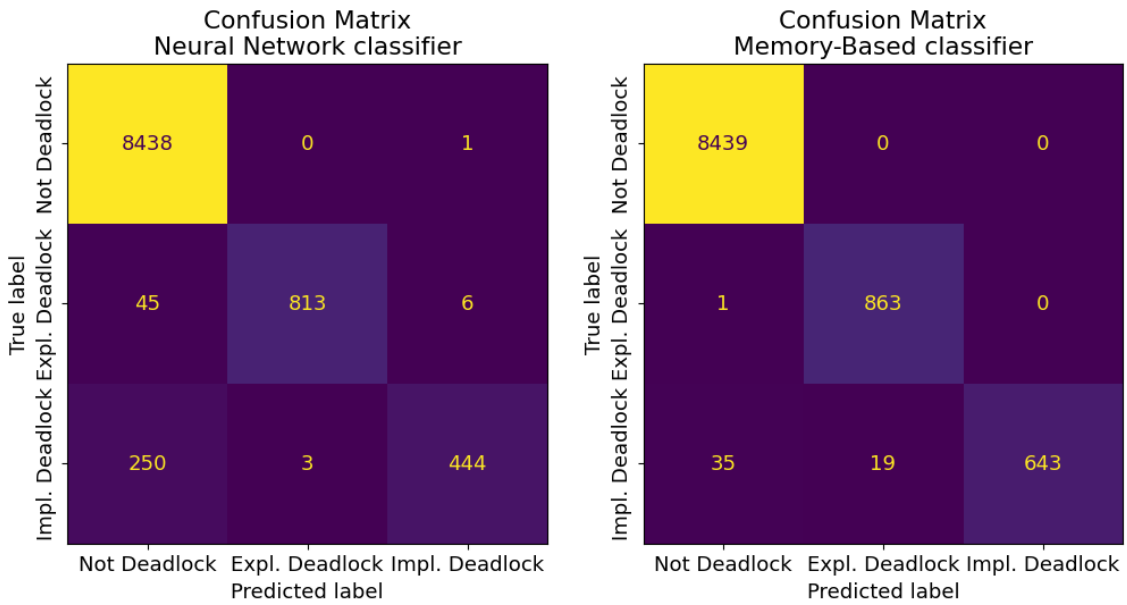


Figure 4.7: Confusion matrices comparing the results of the neural network classifier vs the Memory-Based classifier, both using data generated from 7 vehicles

The result from training the neural network classifier using the 7-vehicle dataset and evaluating the model using the 11-vehicle dataset can be seen in the left confusion matrix of Figure 4.7. It could classify several states correctly, but there were still 295 states that were incorrectly classified as ”Not Deadlock,” resulting in a weighted accuracy of 86%.

In contrast, one can observe the result of using the memory-based classifier in the right confusion matrix of Figure 4.7. The memory contained all true implicit and explicit deadlocks of states with 7 vehicles or less, meaning the only incorrect classifications are deadlocks with more than 7 vehicles in true or composite deadlock. This resulted in only 36 deadlocks being classified as "Not Deadlock", resulting in a weighted accuracy of 97%.

4.6 Finding all possible deadlocks

All classifying methods rely on finding as many deadlocks as possible, either for lookup or as training data. However, one can quickly encounter computational problems finding all possible deadlocks for a given site. The machine's memory runs out as the number of states to consider becomes too large. The problem could be partially alleviated by getting a better computer, but this section will explain why this is not a solution to the problem.

To begin with, the sites decide the maximum number of positions and, consequently, the maximum number of vehicles. Additionally, how the nodes are organised on the site can increase or decrease the amount of deadlocks substantially. All these factors, in turn, affect how long it takes to compute all explicit deadlocks and then how long it takes to explore these to find implicit deadlocks. As the number of positions increases, so do the potential starting points of deadlock chains (see Section 3.1). When explicit deadlocks increase, so does the number of implicit deadlocks, which increases the computation time. However, the key factor (for both explicit and implicit finders) that controls computation time is the number of actions the finders have when placing vehicles in chains or finding previous states.

An action for the explicit finder is a position where a vehicle can be placed to block another vehicle. Each valid action is a chain that must be explored. As such, when the number of actions becomes large (i.e. the branching factor becomes large), the computation time increases exponentially. When nodes are close together, there are many ways to place vehicles, creating many possible actions. However, the time is constrained by the maximum number of vehicles allowed, creating a fixed stopping point where the finder will not take any more actions. As such, it is only for many vehicles on large sites where the explicit finder fails to finish.

However, the same does not hold for the implicit finder. Actions are finding previous states in the state graph that needs to be explored. As such, the number of actions is controlled by the number of vehicles and the number of ways to follow an edge backwards. The number of actions already becomes large when there are many vehicles, as there can be many ways to reach the same state. Additionally, all previous states are always explored. Unfortunately, there is no maximum distance that the implicit finder can explore. There can not be one, as the goal of the implicit finder is to find the implicit deadlocks furthest back from the explicit deadlocks, as those are the states that one wants to avoid stepping into. But, with no constraints on the search backwards, the branching factor combined with the long search distance makes the number of computations too large for the finder to handle.

Having too many previous states to search for is what happens on Sharp site 2, making the site too large for the implicit finder to handle. The deadlocks for a maximum of 15 vehicles are computable, but any more has the computation time outpace the heap space required to compute them. Furthermore, one could choose an even harder site or even more vehicles to solve, which becomes an even harder problem. Unconstrained searches back in time are the core problem in finding implicit deadlocks.

4.7 Summary of results

In summary, while it is possible to find many deadlocked states, it was not possible to find all implicit deadlocks for all sites. The explicit and implicit finders can suffer from exponential growth based on the number of positions and vehicles. While they can manage the Sharp sites with the amount of vehicles in use currently, it has also been clearly shown that the computation time of the finders does not scale. More work must be done to find explicit and implicit deadlocks of many vehicles. Additionally, the site's design affects whether it has exponential behaviour or not. Screwdriver, Oval and Sharp site 1 all have about the same number of nodes, yet only Oval's computation time grew exponentially. As for the choice of classifier, the memory classifiers have a 100% accuracy, given that they rely on lookup. Specifically, the trie-based classifier has a very short lookup time, making it the best one for the traffic planner.

Furthermore, it was found that simple neural network models could be used to classify states at each site tested which high accuracy. However, when the model was only trained on samples of states containing at most 7 vehicles, it could not generalise and maintain high classification accuracy for states with more vehicles.

5

Conclusion

This chapter describes the conclusions that can be drawn from this report, discussions for how the work went, what processes could have be improved, and what avenues of future work that the authors' find interesting.

5.1 Discussion

As a reminder, the research question supporting this thesis was "How can a mathematical model predict deadlock in a vehicle planning system?". Requirements were also that this model should be fast enough to be practically applicable in the traffic planner. As such, different models were developed that can predict deadlock. These models were then used to guide the traffic planner. To begin with, all models rely on the data found by the explicit and implicit finders. Then, the variation between models was if the memory was directly queried or if an approximative model was trained on the data.

The best performing model was the tire classifier. Due to being a trie, lookup speed was not a constraining factor to the model's abilities. Additionally, since the model does deterministic lookup, it has 100% accuracy and will never falsely classify a state that it knows of. Instead, its greatest problem was gathering the required data, as the implicit finder is unable to compute all previous states when sites grow large and vehicles become many. As explained in Section 2.5.6, the branching factor and depth needed to find all implicit deadlocks make the computation time required infeasible. As such, more work needs to be done to either improve implicit deadlock finding or forgo it entirely with functioning approximative methods.

However, it is worth noting that the memory-based trie classifier can handle the current requirements of VAS. For the two Sharp sites analysed, 12 vehicles are around the max of what is currently in use. The implicit finder can compute all deadlocks for these requirements. As such, if the planner was implemented, it could completely avoid deadlocks in a very time-efficient way. The preventive abilities of the classifier have been verified by simulating the sites for an extended period of time and concluding that the system does not deadlock. These simulations use the trie classifier and checks in real-time whether a prospective state is a deadlock. Compared to the current method of preventing deadlocks, this system reduces manual work required for new sites and is a possible improvement if the current human decisions for how vehicles should drive are not optimal.

5.2 Ethics

Deadlocks are, inherently, a subset of states preceding collisions. While they do not cover all states before collisions, as queues are not in deadlock, deadlocked states can be dangerous as if any vehicle decides to drive, the result will be a collision. For this thesis, it has been assumed that some other system prevents real-life collision by using sensors on vehicles or other preventative measures. However, this does not need to be the case, nor is it reasonable to assume that such a system would be faultless. In this case, deadlock *avoidance* acts as a safety measure. If a site can be deterministically guaranteed never to deadlock, the risk of collision from a deadlocked state lessens.

Furthermore, current control of traffic planning relies in part on human intuition to design how vehicles should move. Human intuition can lead to human error, leading to dangerous situations. It is difficult for humans to consider every scenario and every possible way their rules can play out. Autonomous deadlock prevention, such as that designed for this thesis, can help mitigate the danger. In this case, the finders have tried *every* combination of how vehicles can be positioned and thus knows all states that must be avoided. However, the finders are inherently based on a model or simulation of real life. As such, there might be abstractions that cause unpredictable behaviour. While modelling should never be the sole safety insurance, a collaboration between well-designed deadlock prevention and human intuition should prove safer than either method in isolation.

However, deadlock prevention is not always deterministic, as some of the models designed for this thesis are neural networks. They can hallucinate, which could lead the model to fail at a task it is assumed to be good at. In this case, safe states could be classified as deadlocks and vice versa. Regardless of how this affects performance, it might also lead to unpredictable behaviour. Such behaviour can be dangerous if the vehicles interact with human drivers, who assume that autonomous vehicles will drive like humans. If a neural networks is to be used, extra safety measures need to be implemented such that eventual hallucinations are detected and disregarded accordingly.

5.3 Future work

This section will describe the current limitations of the system and the possible angles of future work that the authors' believe are able to solve said limitations.

5.3.1 More sites of different scales

Currently, only 4 sites have been examined for deadlocks. It would be good to test the finders' capabilities on additional sites. For example, Oval introduced deadlocks that relied on the vehicle's tail node, a previously undetected bug until the system was tested on that site. Other unknown deadlock situations may exist, and the current system having complete coverage of all types of deadlocks is very unlikely. As such, the only solution is to perform more tests on different types of sites.

Furthermore, from an evaluation perspective, creating a site between the size of Sharp site 1 and 2 would be useful. As has been found, the leap from Sharp site 1 to Sharp site 2 is very large, and perhaps a mid-size site could have been used to bridge the gap and clearer relation between small and large sites.

5.3.2 Benchmark against existing solution

The method of finding all explicit and implicit deadlocks has only been tested in simulations to verify that the traffic planner does not get stuck in deadlock. However, no benchmarks have been run with the new traffic planner to compare with the existing rule-based solution. If the new solution improves the utilisation of the vehicle fleet, then Volvo Autonomous Solution would be strongly motivated to continue the development of the traffic planner. Additionally, the new solution could be tested at a real-world site such as Sharp Site 1 to verify that the system is functional in practice.

5.3.3 Improving explicit deadlock finder

The method presented for finding all explicit deadlocks did not receive much attention during the development of the thesis. That is because even if the method had been improved, the bottleneck for the entire system was the computation of implicit deadlocks, which the authors gave more attention to.

Due to the nature of the algorithm, where for a given vehicle v, one computes what vehicles can be placed such that v is blocked, this lends itself to be solved using a dynamic programming approach. But since one vehicle can have multiple potential actions that need to be blocked, and where multiple different vehicle positions can block each action, the problem becomes difficult. One would need to define a set of properties which are always true whenever the position of a vehicle is visited, such that redundant computations can be reduced.

Although the method might not be optimal, the authors are satisfied with the resulting method since it can still compute all explicit deadlocks at a large site within minutes. However, improvements could be made, which could be a source of future work.

5.3.4 Implicit deadlock computational limits

As explained multiple times, increasing site size and the number of vehicles on the site leads to the implicit finder being unable to find all deadlocks. This inability comes from the number of states that must be searched growing exponentially. While no immediate solution to this problem has been formulated, these are the authors' ideas.

For the implicit deadlock finder, each previous state in the fringe can be explored in any order without affecting correctness. As such, its operation can be easily parallelised. This could lead to a large reduction in computation time but does not solve the inherent problem of time complexity.

Since the number of vehicles controls the branching factor (as more vehicles have more previous states), methods for searching multiple actions at once could greatly improve the finder. Currently, it only considers one step backwards at a time, which is inefficient when there are many vehicles. A method that combines actions in some way and concludes if *all* of them are implicit deadlocks would be more efficient.

Furthermore, taking more than one step backwards per vehicle is also a good improvement angle. Some attempts at such a method were tried: to take steps backwards until before a vehicle reaches an interesting choice of action. If a step backwards only leaves going another step backwards or standing still as available actions for a vehicle, more than one step could have been taken. In fact, the finder only needs to find the outermost implicit deadlocks, as they are the entrances to the deadlock that the system wishes to avoid. As such, keeping track of all deadlocks is unnecessary, rather than just saving the ones the system can reach from safe states (which could be another improvement). Implementing a method that takes multiple steps would be very efficient, but the authors are unsure how to always prove the correctness of such a method.

5.3.5 Possible unknown implicit deadlocks

The implicit finder computes all deadlocks for each number of vehicles at a time. As described before, it does this since implicit deadlocks can lead to smaller ones. As such, the smaller ones must already have been found such that the deadlocks with more vehicles are not missed. However, this means there can exist implicit deadlocks that the finder cannot find. If the only actions from a true implicit deadlock lead to composite deadlocks, implicit or explicit, the finder cannot find the true implicit deadlock. For example, imagine a state s where 4 vehicles are in true implicit deadlocks. From s, there is only one future state: an explicit deadlock of 3 vehicles. That means there is no direct path to s from an explicit deadlock of 4 vehicles. As such, the *only* way to find s would be to search backwards from the true deadlock of 3 vehicles and add the last required vehicle.

In real life, this is quite the odd scenario. There is some state of 3 vehicles that are in deadlock. But, if one of these vehicles takes a step backwards, it becomes a safe state. However, if this vehicle takes a step backwards and there is another, previously unrelated vehicle at some specific position, the state becomes an implicit deadlock.

There is no such deadlock for any site used in this thesis (to the author's knowledge), but it is a theoretical possibility. Despite this, the implicit finder is, by choice, unable to find this deadlock. This was done because, otherwise, the finder must try and place extra vehicles around implicit (and explicit) deadlocks up to the maximum number of vehicles allowed, which is extremely computationally inefficient. As such, a correct and computationally efficient method for finding these "true to composite" deadlocks is another direction for future work.

5.3.6 N-step TD implicit finder

There were attempts at creating another method for finding implicit deadlocks. Instead of deterministically working backwards, one could pick a random state and randomly traverse forward until a deadlock is reached. Then, each state visited would have its ”chance” of being a deadlock updated. The goal of this method was to find implicit deadlocks faster than the current finder does, such that the branching issue becomes alleviated. Instead of searching all paths, this method would only search one at a time.

As such, an n-step Temporal Difference method was built, as described in *Reinforcement Learning: An Introduction* [14]. As described above, it would randomly search in the state graph and map states to a probability of them being an implicit deadlock. When it reached an explicit deadlock, it would update its previous states via the TD-learning method.

However, this method did not produce relevant results. It did not frequently visit implicit deadlocks such that it could separate them from other states. This is, as the authors believe, due to the considerable number of possible states. Another very relevant factor was that implementation of this method started late. As such, this method could produce good results (perhaps even better than the current finder) if more time was spent on development. In general, exploring the state graph forward (instead of backward from explicit deadlocks) has a lot of potential and is an area the authors would recommend future work to be done in.

5.3.7 Neural networks

Neural network models were successfully created to classify states with high accuracy. However, since the models could not generalise, their usefulness is limited. As described in Section 3.4.3, if a model had understood what an explicit or implicit deadlock is, it could have been used to approximate the implicit deadlock finding method.

One could generate a better training dataset to make the model classify deadlocks with a greater number of vehicles correctly. Since it is possible to generate a subset of implicit deadlocks with a given number of vehicles, the neural network might be able to find the rest. However, such an improved dataset would need states from all classes, including ”Not deadlock” states. These are difficult to generate unless all explicit and implicit deadlocks have been found. In our current method, a state is ”Not deadlock” if and only if it is not in our list of known explicit nor implicit deadlocks. However, this method does not work if all deadlocks can not be found. For a given state to be classified as ”Not deadlock”, one would need to find a sequence of steps leading back to the same state and ensure all vehicles moved at least once in said sequence. One source of future work is how this could be done in a computationally efficient way.

Another approach would be to use some other type of neural network model. In this thesis, a Graph Convolution Network (GCN), as described in Appendix A.3, was explored as a possible solution for improving the model’s generalisation. The

idea was that the network would be able to use the graph structure of a site as context and that many vehicles at locations such as tunnels or intersections would indicate deadlock. However, Using a GCN did not yield better results, as seen in Appendix A.4, compared to the network already described in the thesis. On the other hand, GCN has only briefly been explored by the authors and would need further investigation before being dismissed.

Bibliography

[1] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983, ISSN: 0734-2071. DOI: 10.1145/357360.357365.

[2] V. D. Sasso, L. Lamorgese, C. Mannino, A. Onofri, and P. Ventura, "The tick formulation for deadlock detection and avoidance in railways traffic control," *Journal of Rail Transport Planning & Management*, vol. 17, p. 100 239, 2021, ISSN: 2210-9706. DOI: 10.1016/j.jrtpm.2021.100239.

[3] V. Dal Sasso, L. Lamorgese, C. Mannino, A. Tancredi, and P. Ventura, "Easy cases of deadlock detection in train scheduling," *Oper. Res.*, vol. 70, no. 4, pp. 2101–2118, Jul. 2022, ISSN: 0030-364X. DOI: 10.1287/opre.2022.2283.

[4] M. Chen, Y. Lu, and C. Zhang, "Deadlock-solving traffic control methods for automated guided vehicle systems," in *2021 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 2021, pp. 51–57. DOI: 10.1109/IEEM50564.2021.9673048.

[5] H. Kaid, A. Al-Ahmari, E. A. Nasr, *et al.*, "Petri net model based on neural network for deadlock control and fault detection and treatment in automated manufacturing systems," *IEEE Access*, vol. 8, pp. 103 219–103 235, 2020. DOI: 10.1109/ACCESS.2020.2999054.

[6] T. Goto, H. Itaya, T. Hirakawa, T. Yamashita, and H. Fujiyoshi, "Solving the deadlock problem with deep reinforcement learning using information from multiple vehicles," in *2022 IEEE Intelligent Vehicles Symposium (IV)*, Aachen, Germany: IEEE Press, 2022, pp. 1026–1032. DOI: 10.1109/IV51971.2022.9827367.

[7] W. Qin and Z. Ren, *Improving deadlock avoidance in multi-agv systems*, https://hdl.handle.net/20.500.12380/302306, Gothenburg, Sweden, 2020.

[8] Y. Zhou, H. Hu, Y. Liu, and Z. Ding, "Collision and deadlock avoidance in multirobot systems: A distributed approach," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 7, pp. 1712–1726, 2017. DOI: 10.1109/TSMC.2017.2670643.

[9] P. J. T. Paul E. Black, *Dictionary of algorithms and data structures*, Accessed: 2023-05-17, 2022. [Online]. Available: https://www.nist.gov/dads/HTML/graph.html.

[10] N. L. Biggs, *Graph theory 1736-1936*, 1st. Clarendon Press, 1976, ISBN: 0-19-853901-0.

[11] E. G. Coffman, M. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971, ISSN: 0360-0300. DOI: 10.1145/356586.356588.

[12] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960, ISSN: 0001-0782. DOI: 10.1145/367390.367400.

[13] N. N. 101, *Neural network activation function*, https://neuralnetwork101.com/activation-function, Accessed: 2023-05-17, 2023.

[14] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 2nd. MIT press, 2018, ISBN: 9780262039246.

[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. DOI: 10.48550/arXiv.1609.02907. arXiv: 1609.02907.

A

Appendix 1

A.1 Correctness proof for implicit deadlock finder

This proof will prove why there always is at least one possible state in the Fringe set that can be verified as a deadlock and added to KD (set of Known Deadlocks). To begin with, if there are no true implicit deadlocks that are not in KD then the proposition always holds. As such, we can ignore this scenario.

To prove the other case, we use a proof by contradiction. Assume that there are some arbitrary amount of true implicit deadlock states which are not in KD but that none of them lead exclusively to KD. Call this set of unknown states F. Also, we assume that the states in F all have the m number of vehicles and that all true deadlocks of fewer than m vehicles already are in KD.

Choose an arbitrary state in F and call it S_1. S_1 must have 1 or more outgoing edges, since only true explicit deadlock have no outgoing edges. Furthermore, 1 or more of these edges do not lead to KD, per the assumption above. We can call these the unknown edges of S_1 or UE_{S_1}. Since S_1 is a true implicit deadlock, none of the edges in UE_{S_1} lead to a safe state. Nor can they lead to an explicit deadlock, since they are all assumed to be in KD. The edges can also not lead to a deadlock with fewer than m vehicles, as they are also already in KD. As such, each edge can only lead to some other state in F. Following one of the edges in UE_{S_1}, leads to another state $S_2 \in F$. This process can then be repeated, such that we have path P of states in F.

However, since P is a path of true implicit deadlocks, P can not be a cycle, as per the definition of true implicit deadlocks (see Section 2.5.4). As such, it must be some finite path. We denote the final state of P as S_n. S_n can not lead back to any previous state, as that would form a cycle, nor can it lead to a safe state as previously shown. It must also have at least one outgoing edge, as it is not in the set of true explicit deadlocks. As such, the only possibility is that S_n has edges that lead to states in KD, which is a contradiction to the initial assumption.

A.2 Explicit deadlock classifier

This section describes a system that can classify if a state is in explicit deadlock without using a preprocessed list of deadlocks. It works by finding and removing vehicles which are note blocked until no more vehicles can be removed. When no more vehicles can be removed, but some remain, the state is in an explicit deadlock. But, if there are no vehicles left at all, the state is a safe state.

A.2.1 Finding blocking vehicles

This section defines what a blocked vehicle is. A vehicle is *fully blocked* if the vehicle is stationary and that, for any possible movement by the vehicle, it will collide with some other stationary vehicle. Additionally, a vehicle is *partially blocked* if some, but not all, of its actions are blocked by another vehicle.

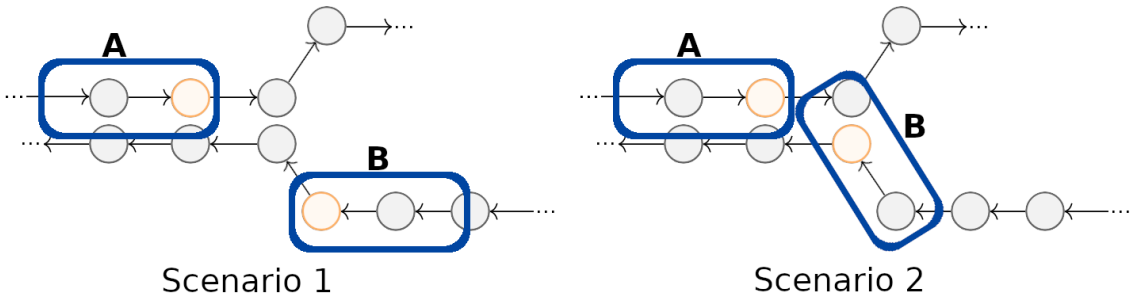


Figure A.1: Scenario 1 and 2 of vehicles blocking each other. The orange nodes are the locations of the vehicles and the blue squares are the area which the vehicle claims.

In Figure A.1, the reader can see scenario A and B where vehicle 1 wants to move to the right and vehicle 2 want to move to the left. The blue squares show the physical space (with some margin) that each vehicle claims. In scenario A, vehicle 1 can continue driving up to node 4 without colliding with another vehicle. Vehicle 2, on the other hand, is *blocked* by vehicle 1 resulting in a set of all blockings being $\{(2, \{1\})\}$. In scenario B, vehicle 1 can not continue as it is being blocked by vehicle 2 and vice versa. As such, the resulting set of blockings is $\{(1, \{2\}), (2, \{1\})\}$.

In order to find these blockings, one can simply iterate over each vehicle, checking for each "move action" if there is a collision with another vehicle. If all "move actions" for a given vehicle result in a collision, the vehicle which is collided with is recorded as a blocking.

A.2.2 Using blocked vehicles

From the method in Section A.2.1, the set of all blockings is created:

$$B = \{(v_a, \{v_b, v_c, ...\}), (v_d, \{v_e, v_f, ...\}), ...\} \tag{A.1}$$

where vehicle $v_i \in V$, i.e., the set of all vehicles, and for each ordered pair $(u, \{a, b, ...\}) \in B$, vehicles a, b, and so on is blocking vehicle u for one action. The next step is to validate if there is a circular wait, i.e., explicit deadlock. Given the definition of explicit deadlock, there must be a set of vehicles U which block each other for there to be a deadlock.

If vehicle v does not have all its actions *blocked*, then it cannot be part of a deadlock since it can take a step forward. Subsequently, all other vehicles blocked by v can also not be part of a deadlock. As such, blockings can iteratively be removed when the blocking vehicle becomes unblocked. The blocked vehicle will then be marked as "not blocked" which will unblock even more vehicles in the next iteration. If there are no more blockings in B that can be removed, but B is not empty, then there is a subset of vehicles that block each other, i.e., there is an explicit deadlock. See the complete algorithm description in the following pseudocode:

```
Require: B is the set of all blocked actions
  isBlocked ⇐ ordered list of blocked state, each isBlocked_u is true if a vehicle u
  is blocked, otherwise false
  found ⇐ true
  while found = true do
      found ⇐ false
      for (u, blockerList) in B do
          remove all vehicles from blockerList which are not blocked
          if blockerList is empty then
              isBlocked_u ⇐ false
          end if
          if isBlocked_u = false then
              remove (u, blockerList) from B
              found ⇐ true
          end if
      end for
  end while

  if B is empty then
      return false
  else
      return true
  end if
```

A.3 Graph convolutional network

A graph convolutional network (GCN), or graph neural network (GNN), is a special type of network which utilises the information given by the graph structure in its input data. Such data could be relations between users in a social network, compute network, road network or something else with a network structure. Similarly to the use of a convolutional neural network (CNN) for image classification, which for each pixel considers surrounding pixels, the GCN considers the adjacent nodes in the graph when evaluating each node [15].

As an example, for a dense neural network with two layers, one can define the model as $f(X)$ such that:

$$f(X) = Softmax(ReLU(XW^{(0)})W^{(1)})$$

In this model, X is the input vector where $W^{(0)}$ and $W^{(1)}$ are the weights of the two layers, and Softmax and ReLU are used as activation functions. If two GCN layers are used instead, as in the paper by Kipf et al. [15], the resulting model $f(X, A)$ where A is an adjacency matrix is defined as

$$\begin{aligned}
f(X, A) &= Softmax(\hat{A}\ ReLU(\hat{A}XW^{(0)})W^{(1)}) \\
&\text{where} \\
\tilde{A} &= A + I_N \\
D_{ii} &= \sum_j \tilde{A}_{ij} \\
\hat{A} &= \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \\
W^{(0)} &\text{ and } W^{(1)} \text{ are weight vectors}
\end{aligned}$$

A.4 GCN - results

Table A.1: Results from Graph Neural Network Classifier using one GCN layer

Site	Train Samples	Test Samples	Class Dist. (ND, ED, ID)	Accuracy
Screwdriver	160 000	40 000	(75%, 4.2%, 21%)	91.2%
Oval	160 000	40 000	(96%, 2.3%, 1.9%)	97.8%
Sharp Site 1	400 000	100 000	(96%, 2.4%, 1.9%)	96.9%
Sharp Site 2	1 800 000	200 000	(85%, 1.1%, 14%)	92.3%

Table A.1 is the same as Table 4.2, but Table A.1 uses one GCN layer in the beginning of the network, while the models in 4.2 only use linear layers.

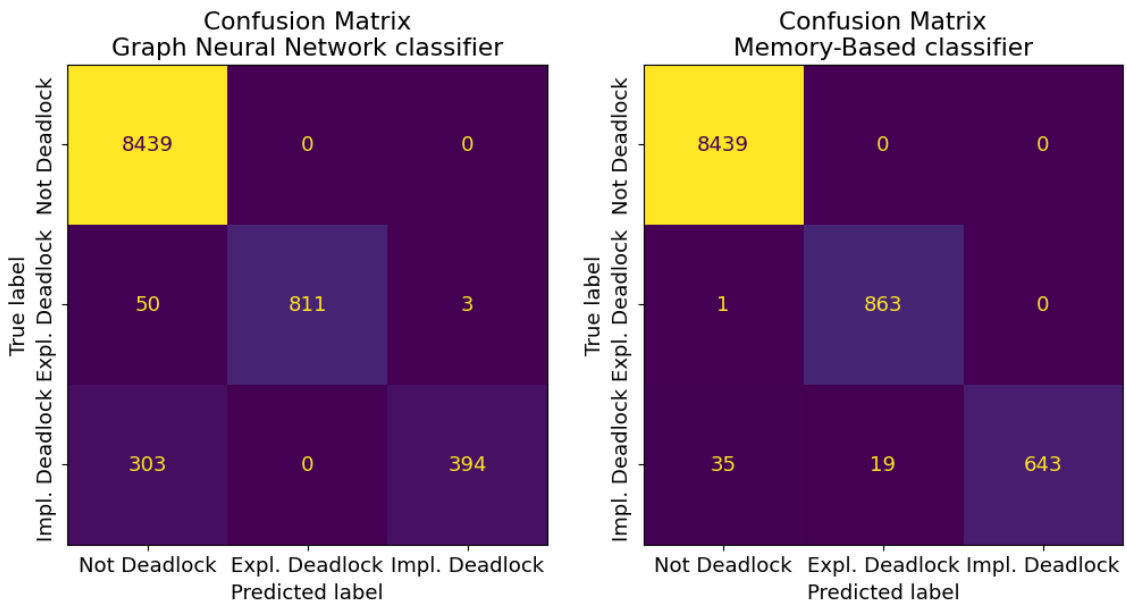


Figure A.2: Confusion matrices comparing the results of the graph neural network classifier vs the Memory-Based classifier, both using data generated from 7 vehicles

Figure A.2 illustrates the same thing as Figure 4.7, but Figure A.2 uses one GCN layer in the beginning of the network, while the model in 4.7 only uses linear layers.

A.5 Explicit deadlock finder psuedocode

Note that the procedure $findExplicitDeadlocks$, which finds all explicit deadlocks, is the last procedure defined in this block of pseudo code.

```
Require: SiteGraph = (N, E) in Definition 2.2.1
Require: Claim(e) in Definition 2.3.2
  procedure getBlockingVehicles(a, C)
      return {v|Claim(a) ∩ Claim(v) ≠ ∅ ∧ Claim(v) ∩ C = ∅}
  end procedure
  procedure getValidActions((u, v), C)
      return {(v, w)|(v, w) ∈ E ∧ (Claim((v, w)) ∩ (C \ Claim((u, v))) = ∅}
  end procedure
  procedure categoricalProduct(setOfCategories)
      return X_1×X_2×...×X_m where X_i ∈ setOfCategories and m = |setOfCategories|
  end procedure
  procedure getBlockChains((u, v), C)
      validActions ⇐ getValidActions((u, v), C)
      if validActions = ∅ then
          return {{(u, v)}}
      end if
      actionBlockChains ⇐ ∅
      C_new ⇐ C ∪ Claim((u, v))
      for a in validActions do
          blockingVehicles ⇐ getBlockingVehicles(a, C_new)
          if blockingVehicles = ∅ then
              return ∅
          end if
          actionBlockChain ⇐ ∅
          for b in blockingVehicles do
              actionBlockChain ⇐ actionBlockChain ∪ getBlockChains(b, C_new)
          end for
          actionBlockChains ⇐ actionBlockChain ∪ {actionBlockChain}
      end for
      blockChains ⇐ categoricalProduct(actionBlockChains)
      for chain in blockChains do
          chain ⇐ chain ∪ {(u, v)}
          if ∃a, b ∈ chain : Claim(a) ∩ Claim(b) ≠ ∅ then
              remove chain from blockChains
          end if
      end for
      return blockChains
  end procedure
  procedure trimDeadlock(deadlock)
      found ⇐ 1
```

```
    while found = 1 do
        found ⇐ 0
        for v in deadlock do
            if v is not blocking a vehicle in deadlock then
                found ⇐ 1
                remove v from deadlock
            end if
        end for
    end while
    return deadlock
end procedure
procedure findExplicitDeadlocks
    deadlocks ⇐ ∅
    for e in E do
        blockChains ⇐ getBlockChains(e, ∅)
        for chain in blockChains do
            deadlocks ⇐ deadlocks ∪ {trimDeadlock(chain)}
        end for
    end for
    return deadlocks
end procedure
```

A.6 Implicit deadlock finder

Require: $fringe$ is a stack of States
Require: KD the set of known deadlocked States
Require: E is the set of explicit deadlock states

```
for eD in E do:
    fringe ⇐ getPreviousStates of eD
    KD ⇐ eD
end for

while fringe ≠ ∅ do
    current ⇐ last element from fringe
    remove last element from fringe

    nextStates ⇐ getNextStates of current
∀ s in nextStates
    if s in KD then
        KD ⇐ current
        fringe ⇐ getPreviousStates of current
    end if
end while

return KD
```

A.7 Trie data structure

```
procedure INSERT(trie, stateBits)
    current ⇐ trie
    n ⇐ number of bits in stateBits
    for i in 0, 1, .., n − 1 do
        if current[stateBits[i]] = empty then
            current[stateBits[i]] ⇐ [empty, empty]
        end if
        current ⇐ current[stateBits[i]]
    end for
end procedure
procedure CONTAINS(trie, stateBins, startIndex)
    current ⇐ trie
    n ⇐ number of bits in stateBits
    for i in startIndex, startIndex + 1, .., n − 1 do
        if stateBins[i] = 0 ∧ current[0] = empty then
            return false
        else
            if stateBins[i] = 1 then
                if contains(current[1], stateBins, i + 1) = false then
                    return Contains(current[0], stateBins, i + 1)
                else
                    return true
                end if
            end if
        end if
    end for
    return true
end procedure
```

A.8 Deadlocked states

This section contains some examples of explicit or implicit deadlock. Note that the slight overlap of vehicles is because the rectangle includes a padding of air around each vehicle. The only requirement for collision is that two vehicles claim the same node, which none of these example do.

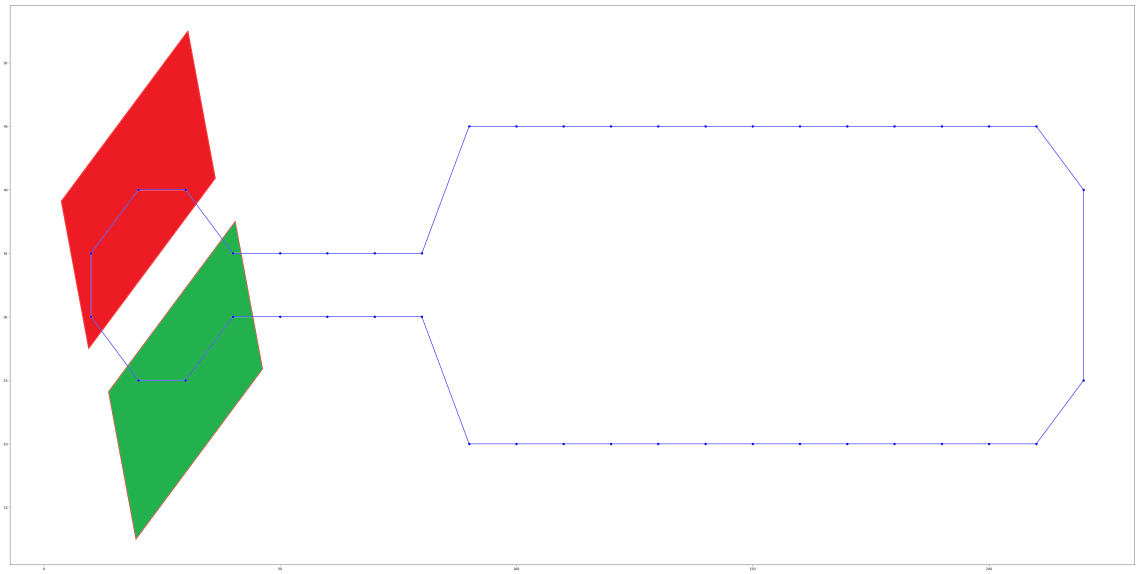


Figure A.3: Explicit deadlock of 2 vehicles on Screwdriver site

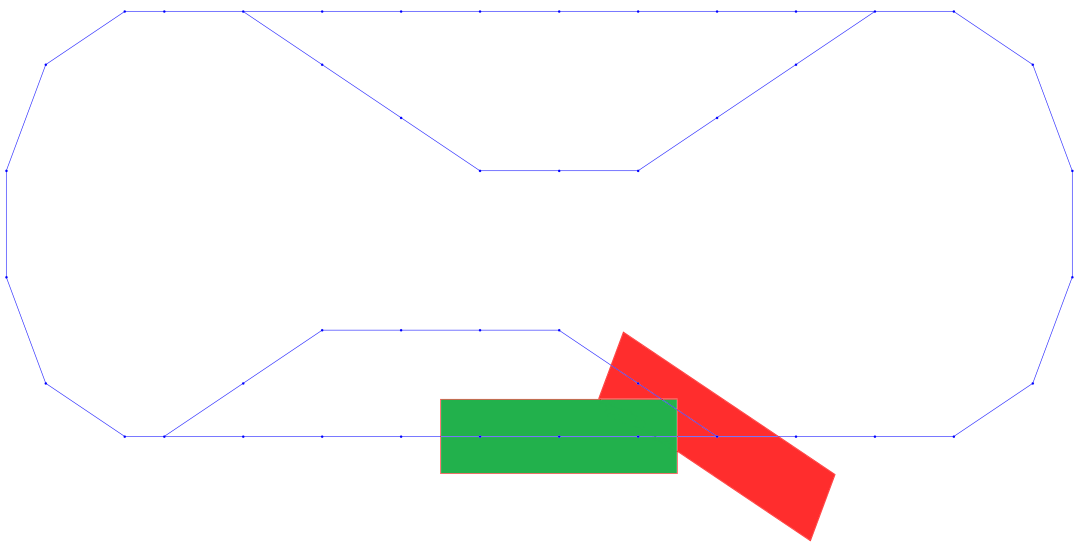


Figure A.4: Explicit deadlock of 2 vehicles on Oval site

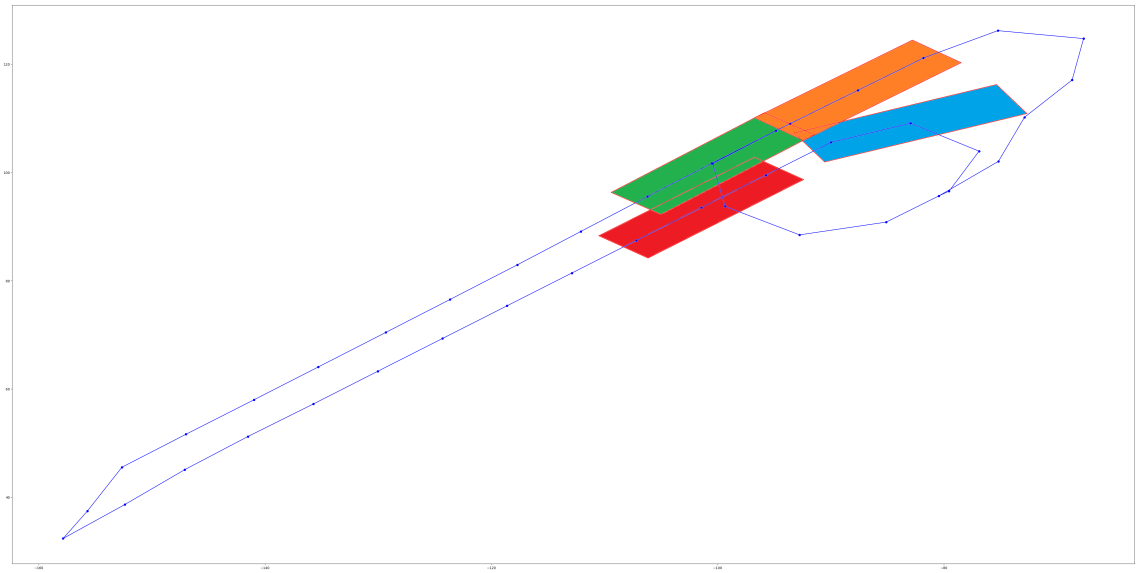


Figure A.5: Explicit deadlock of 4 vehicles on Sharp site 1

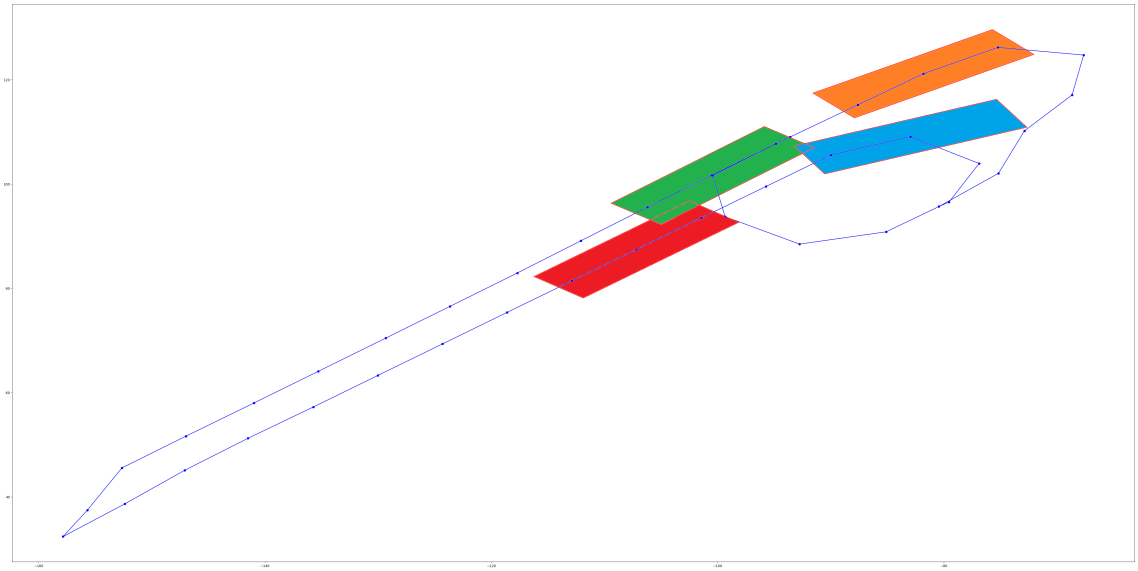


Figure A.6: Implicit deadlock of 4 vehicles on Sharp site 1

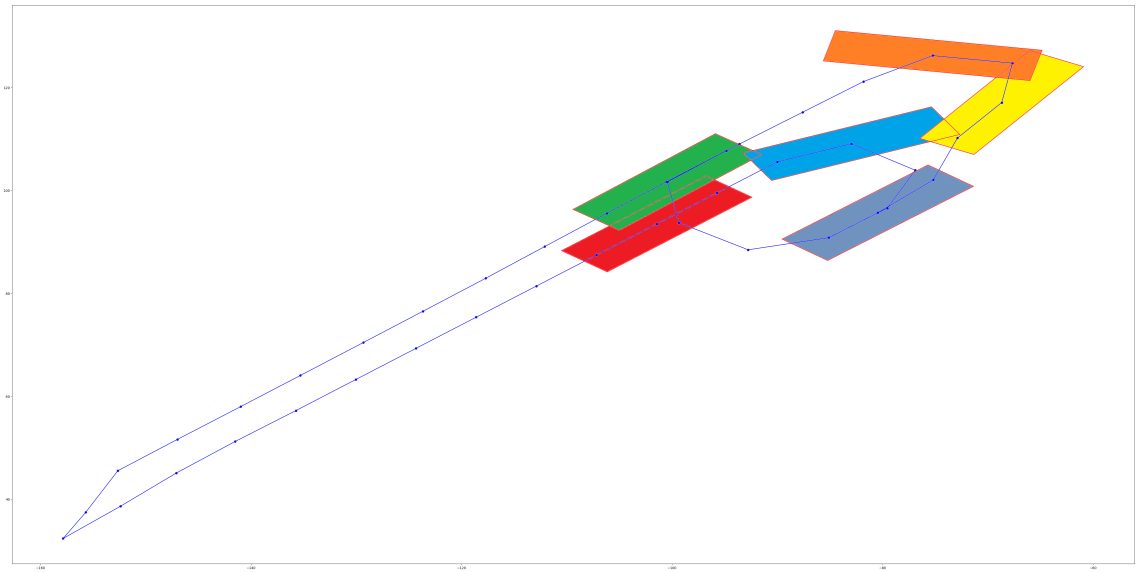


Figure A.7: Implicit deadlock of 6 vehicles on Sharp site 1

A.9 Site graphs

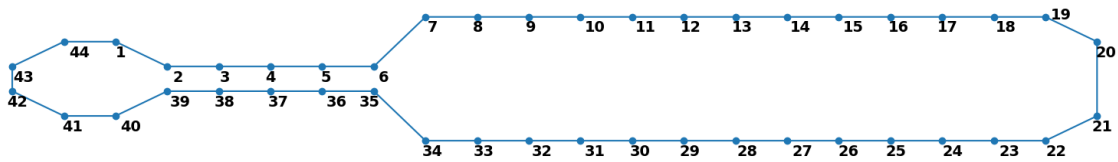


Figure A.8: Site graph of screwdriver site

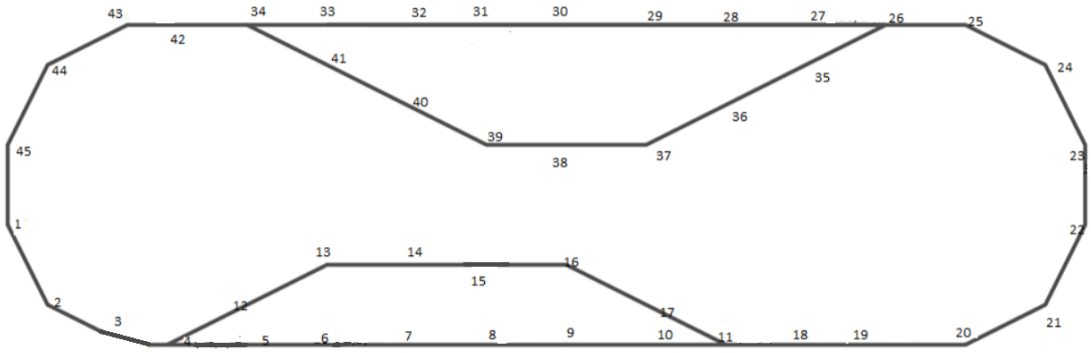


Figure A.9: Site graph of Oval site

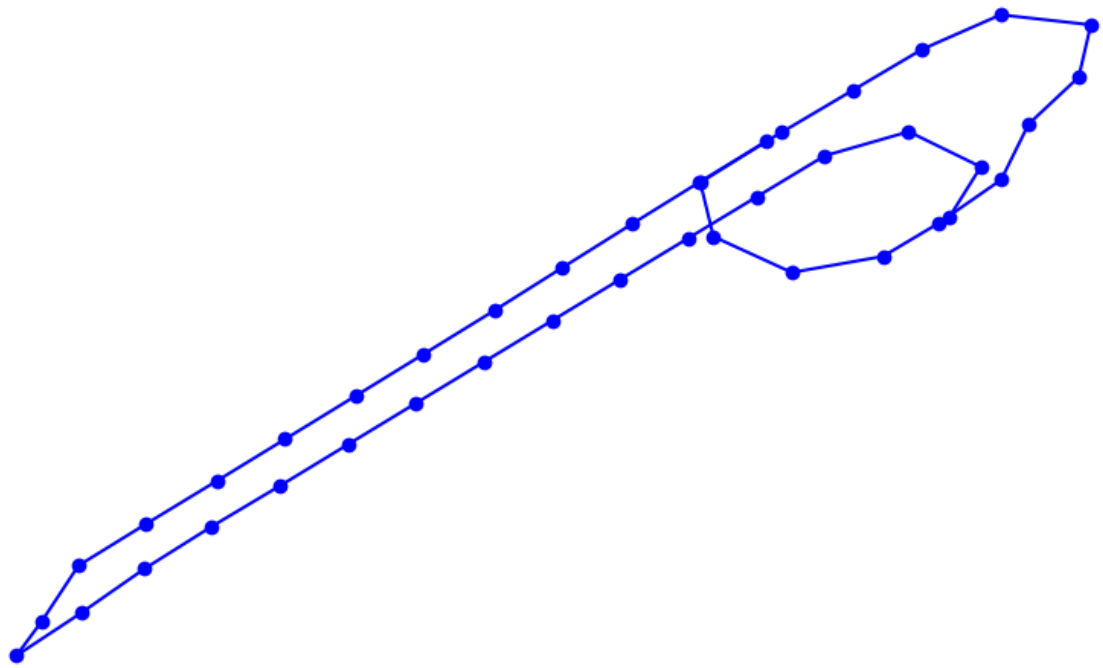


Figure A.10: Site graph of Sharp site 1

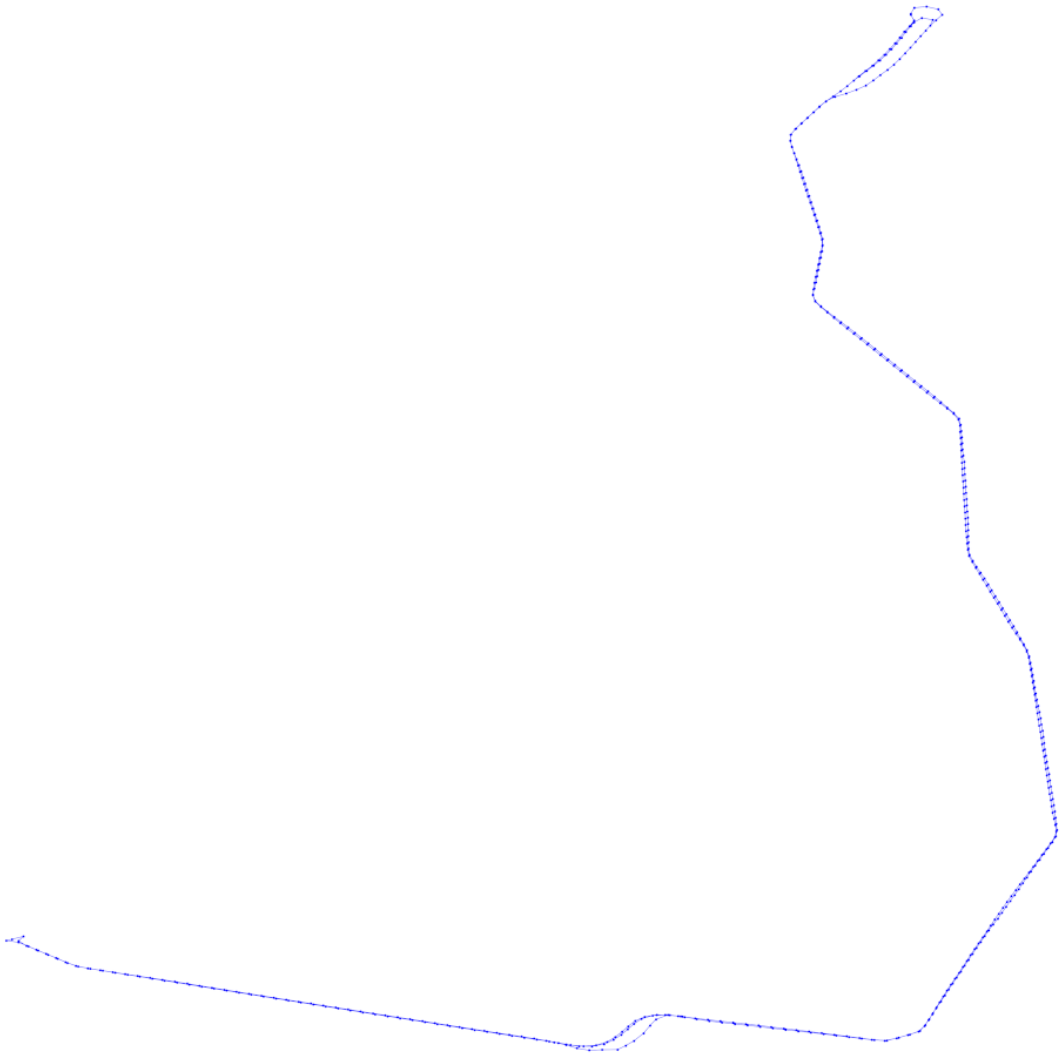


Figure A.11: Site graph of Sharp site 2