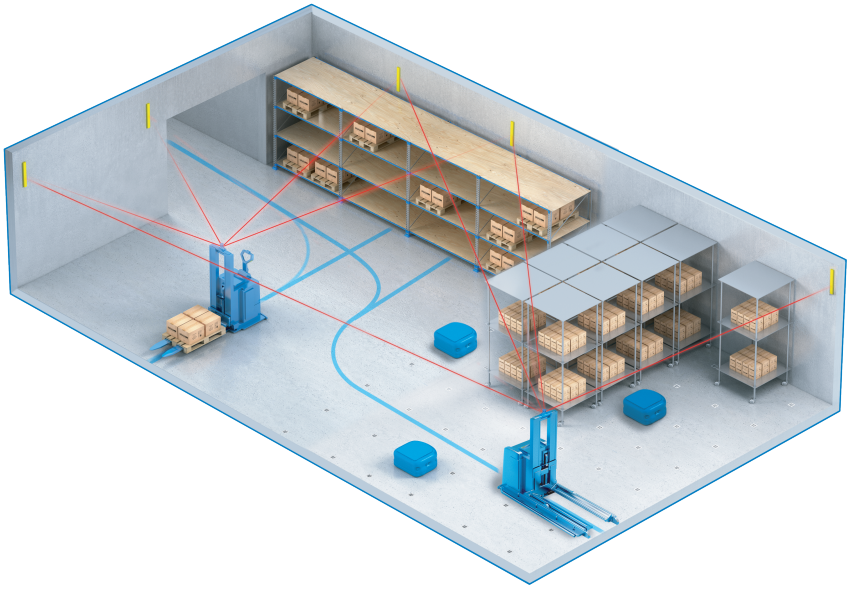
CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

MASTER'S THESIS



Solving Common Goal Conflicts in a Graph-based Multi-agent System

LIAM HÅKANSSON
DENNIS ZORKO

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

Solving Common Goal Conflicts in a Graph-based Multi-agent System

Finding and Assigning Queue Positions for AGVs dynamically in Kollmorgen's System Manager NG

LIAM HÅKANSSON
DENNIS ZORKO

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
Gothenburg, Sweden 2019

Solving Common Goal Conflicts in a Graph-based Multi-agent System
Finding and Assigning Queue Positions dynamically for AGVs in Kollmorgen's System Manager NG
LIAM HÅKANSSON
DENNIS ZORKO

Supervisor: Milica Bijelovic, Kollmorgen Automation AB
Examiner: Michael Patriksson, Department of Mathematical Sciences

Master's Thesis 2019
Department of Matemathical Sciences
Division of Applied Mathematics and Statistics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of two AGVs navigating in a warehouse environment. Property of Kollmorgen Automation AB.

Typeset in LaTeX
Printed by [Name of printing company]
Gothenburg, Sweden 2019

Solving Common Goal Conflicts in a Graph-based Multi-agent System
Finding and Assigning Queue Positions for AGVs dynamically in Kollmorgen's System Manager NG
LIAM HÅKANSSON
DENNIS ZORKO
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

We attempt to improve the throughput of graph-based multi-agent AGV system by solving the issue of common goal conflicts. The conflicts are solved by by assigning queue positions based on statistical properties of the graph. These properties include commonly used paths, as well as common fetch and drop locations for orders. These assignments allow agents, which would otherwise remain stationary due to the MAPP algorithm's inability to find a path to occupied goals, to instead be directed to a suitable intermediate position until the goal becomes available.

Keywords: AGV, MAPP, graph, path planning, common goal, conflict.

Acknowledgements

First of all, we would like to thank Kollmorgen for giving us the opportunity to work with this interesting subject for our thesis, providing us with everything we could possibly need as well as the fika every friday. We would like to thank our supervisor Milica for her work making this thesis possible and answering every question we've ever had. Lars and the rest of red team also deserves gratitude for all the feedback and guidance over the course of the project.

Lastly, we would like to thank our examiner Michael for his help with providing us with the theoretical background for the project, as well as lending a critical eye to this report.

Liam Håkansson & Dennis Zorko, Gothenburg, June 2019

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Finding a queue position . . . 2
1.3 Delimitations . . . 2
1.4 Specification of the problem . . . 3

2 Theory **5**
2.1 Previous work . . . 5
2.2 Defining a good queue position . . . 6
2.3 Graph theory . . . 6
2.3.1 Directed and undirected graphs . . . 7
2.3.2 Paths in graphs . . . 8
2.3.3 Betweenness Centrality . . . 9
2.4 Path planning . . . 9
2.4.1 Dijkstra's algorithm . . . 9
2.4.2 Multi-agent path planning . . . 10

3 Method **11**
3.1 Overview . . . 11
3.1.1 Defining a penalty function . . . 12
3.2 Static evaluation . . . 12
3.2.1 Statistical path usage . . . 12
3.2.2 Exit path . . . 13
3.2.3 Expected time loss . . . 14
3.3 Dynamic evaluation . . . 15
3.3.1 Search space . . . 15
3.3.1.1 Algorithm for finding the search space . . . 16
3.3.2 Paths of other vehicles . . . 16
3.3.3 Divergence from optimal path . . . 19
3.3.3.1 Algorithm . . . 19
3.3.4 Travel time . . . 19
3.4 Assigning the queue position . . . 20

4 Results **21**
4.1 Optimal queue position vs closest possible vs current solution 21
4.1.1 System A 21
4.1.2 System B 23
4.1.3 System C 23
4.2 Different variations on queue assignment 26
4.3 Time consumption 26

5 Conclusions **29**
5.1 Throughput performance 29
5.2 Time performance 30
5.3 Data 30
5.4 Challenges 31
5.4.1 Time constraint 31
5.4.2 Layout intentions 31
5.5 Ideas that were not implemented and suggested future work 31
5.5.1 Analyzing the flow in the system 32
5.5.2 Prioritizing the search space 32

Bibliography **33**

List of Figures

2.1 Different types of edges in a graph. 7

3.1 An example of a exit path in a layout. If a vehicle is at the marked station it always needs to traverse the exit path to leave the station. 13
3.2 A graph showing the probability of conflict depending on when the queuer is expected to arrive to the potential queue position. The closest planned paths (in time) are at the 15 and 50 step marks. The operation time is set to 15, which means that if the queuer arrives to the queue position before the 35 step mark, then it will likely leave the queue position before a conflict arises. 18

4.1 Histogram showing the average time in seconds to complete an order in system A running 2 vehicles. Each data point is an average of 50 orders. More details can be found in in Table 4.1 22
4.2 Histogram showing the average time in seconds to complete an order in system A running 3 vehicles. Each data point is an average of 50 orders. More details can be found in in Table 4.1 23
4.3 System B with five vehicles running. Queue assignment has a mean of 48.0 seconds, standard has a mean of 49.7 seconds, and MV has a mean of 48.8 seconds. This puts our solution at 3.5% higher throughput than the standard, and 1.6% higher throughput than MV. These numbers are summarized in Table 4.2 24
4.4 Average number of seconds to complete an order in system C. The system is running 5 AGVs, and performing 50 orders per test. The mean time to complete an order is 46.7 s for our queue assignment algorithm and 48.7 s for Kollmorgen's standard algorithm. This is 4.3% increase in throughput for the system using our queue assignment. 25
4.5 Average number of seconds to complete an order in system C. The system is running 8 AGVs, and performing 50 orders per test. The mean time to complete an order is 36.8 s for our queue assignment algorithm and 40.1 s for Kollmorgen's standard algorithm. This is 9.0% increase in throughput for the system using our queue assignment. 25
4.6 Average number of seconds to complete an order in system C. The system is running 10 AGVs, and performing 50 orders per test. The mean time to complete an order is 38.5 s for our queue assignment algorithm and 42.25 s for Kollmorgen's standard algorithm. This is 9.7% increase in throughput for the system using our queue assignment. 26

4.7 Three variations of the Queue Assignment solution, run on system C for 8 vehicles. The standard configuration compared to one where no planned paths are taken into account, and one where the search space is reduced. The means are 36.8, 37.6 and 38.1 respectively, with variances of 1.9, 2.3 and 1.8. 27

List of Tables

4.1 The results for system A. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n the sample size. The indices Q, S and M refer to Queue assignment, Standard and MV. Unfortunately, the failure rate for 5 vehicles is unavailable, along with the failure rate of MV for 3 vehicles, due to the data accidentally being overwritten by the testing platform. The histograms for 2 and 3 vehicles can be seen in Figure 4.1 and Figure 4.2 respectively. 22

4.2 The results for system B. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n is the sample size. The indices Q, S and M refer to Queue assignment, Standard and MV. These numbers are visualized in Figure 4.3. 23

4.3 The results for system C. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n is the sample size. The indices Q and S refer to Queue assignment and Standard. 24

4.4 The results for system C when testing different component combinations. μ is the mean, σ^2 is the variance, and r_{fail} is the failure rate. The indices Q, N and R refer to Queue assignment, no planned paths and reduced search space. 26

4.5 Time to find a Queue position. Data points are the number of queue position requested for the 50 orders of an arbitrarily chosen test run. 27

4.6 Time to find a search space. Data points are the number of search spaces requested for the 50 orders of an arbitrarily chosen test run. . 27

1
Introduction

In the late 18th century, the first industrial revolution changed the world forever with mechanization. A hundred years later, the second industrial revolution brought mass production. During the 20th century, the third industrial revolution was brought about by electronics and information technology, starting an automation of industry that is still going on today. And finally, the latest developments in IT and communication is bringing the fourth industrial revolution, Industry 4.0, with concepts like Smart Factories, Cloud Computing and the Internet of Things. [1]

The same developments that allow this fourth revolution – connectivity and computational power – are of course also allowing automation on an ever-increasing scale. Automation enables companies to increase efficiency and safety [2] through reduced workforce. One of the sectors where automation is currently having a significant impact is logistics, particularly in warehouses and factories [3].

Automation and increasing efficiency is at the heart of this thesis. In this introduction we will give some background on the problem we are faced with, state the aim of our work and make a problem specification of how we plan to achieve that aim.

1.1 Background

The thesis was suggested by Kollmorgen Automation AB. Kollmorgen is a company that produce software for Automated Guided Vehicles (AGVs). Their system consists of a number of AGVs that travel on a graph-based layout. The AGVs are guided by a central controller assigning orders and performing path planning.

The most common solutions to multi-agent path planning problems assumes that every agent is assigned a unique goal [4]. For Kollmorgen it is common for two or more agents to acquire the same goal, which may cause problems. Currently, the system makes no special considerations for this, and allows all vehicles to proceed towards their intended goal. Once one of the vehicles enters the goal stations, all other vehicles en route will simply stop at their current position. The situation is no longer solvable as all solutions must include the goal as the final position and the goal is occupied.

An agent just stopping anywhere can cause several issues. Vehicles stopping in highly trafficked locations might obstruct other vehicles. Now either the vehicle obstructing needs to move or the vehicle being obstructed needs to take a detour, both cases resulting in unnecessary wear and tear on the vehicles. Another example is, if two vehicles are following each other closely and the second vehicle stops just

behind the vehicle that has entered the station. The first vehicle is now blocked in and the second vehicle needs to move out of the way, possibly resulting in significant extra movement.

Some different solutions to this have been proposed. One is to let an engineer manually assign static queue positions. This has significant drawbacks in that one queue position may not be suitable for every situation, and it takes a long time to perform the simulations needed to verify whether the queue position works as intended. Our supervisor has implemented another solution that allows a vehicle to proceed towards its target as far as is possible without preventing another vehicle from exiting said target, but it has not yet been thoroughly tested. It also does not take into consideration some of the other issues previously described, and thus may still cause issue.

1.2 Finding a queue position

The aim of this thesis is to improve the performance of the company's system by assigning suitable queue positions to vehicles trying to reach unavailable goals. The desired result is thus increasing throughput of the system, i.e. more orders completed in the same time span.

To reach this we must first identify what constitutes as a suitable position for a vehicle to queue while it waits for a target destination to become available. Based on this definition on what constitutes as a good queue position, we can then construct methods for identifying these positions.

These methods can then be implemented in the system. Ideally, they should be able to run in real time, dynamically identifying suitable queue positions as needed. Should these calculations prove too costly, however, it would be acceptable to use the solution offline as decision support for engineers manually setting static queue positions.

1.3 Delimitations

Firstly, we are not necessarily looking to find an optimal solution for any given case. We are looking to find solutions that are better than nothing, at as low computational cost as possible. Optimal solutions tend to be costly in terms of computational power, and in the experience of the company and our supervisor the trade-off is rarely worth it. We will focus on speed in our solution, since it will be working within an existing algorithm and with a very limited allotted time.

Secondly, we start by focusing on smaller systems, and gradually look at larger systems as the solution progressed. Starting with smaller example systems from clients with just 2 or 3 AGVs allowed us to verify the tractability of our solution in real time, and make optimizations as needed.

Lastly, we have only tested our solution with Kollmorgen's graph-based layouts. In theory, the method could be applied to any graph-based layout with similar features, but since we do not have any such systems available outside of Kollmorgen's own, we will limit ourselves to those.

1.4 Specification of the problem

As described previously, there are essentially three different steps to this work:

- Designing the model: in order to identify suitable queue positions, we must define what a suitable queue position is. In order to do this, we need a state representation of the system.
- Methods: once we have a state representation and a definition of our desired state, we can construct methods for evaluation positions in the graph based on these criteria.
- Implementation: the concrete implementation of the solution to work with the companies system.

Once an appropriate model has been specified, suitable algorithms can be evaluated and adapted to our specific problem. Once this is done, the concrete implementation can be constructed and tested to evaluate the result.

2
Theory

Here we will present the tools needed to motivate and understand our solution. These tools will primarily consist of graph theory, which is used to describe and evaluate the layouts in which the AGVs operate, as well as path planning, which are the algorithms used to calculate the movement of the AGVs in these layouts. First, however, we will present some previous works and define what we are looking for in a good solution to our problem.

2.1 Previous work

One of the central problems of multi-agent path planning is in avoiding collisions, due to the fact that agents are not allowed to occupy the same space at the same time. This is what separates it from simply planning the paths of several agents independently of each other. Sharon [5] present a conflict-based search method to solve this problem. Their solution, however, assumes two things: that goals are unique and that agents no longer need to be taken into account once they reach their goal. Banerjee [6] propose an algorithm that solves the problem of goal persistence, i.e., the case where agents persist after reaching their goal, thus potentially preventing another agent from reaching their destination. However, it still fails to consider the case of multiple vehicles sharing a common goal. Similarly, Yu [7] assumes that no agents will share a common goal.

Some works concern the specific problem of goal conflicts, although none of them have been applicable to this solution. Tavakoli [8], for instance, treat a similar problem, but like Sharon [5] assume an agent to be removed from the system upon reaching its goal. Unfortunately, agents in our system must be assumed to be persistent, making this solution invalid. Another example by Khaluf [9] relies on switching orders between agents to avoid conflicts altogether. In our case, however, the conflicts often arise due to the fact that a number of agents are transporting their load to a few, or even a single particular station that is unique in the system, thus making it impossible to change their destination.

Instead, we will attempt to build a system where an agent that has a goal that is–or will be–occupied by a different agent relies on finding a suitable position in the system to wait for its target to become available. We were unable to find any previous works with similar solutions.

However, before we can start thinking about how to find a suitable queue position in the system, we need to define what makes a position suitable for queuing.

2.2 Defining a good queue position

As mentioned previously, the goal of this work is to improve the throughput of a system. Since vehicles will need to wait *somewhere* for their target to become available, we need to figure out where they cause the least amount of delays by doing so.

So, what causes delays in the system? First, we have the time it takes for a vehicle to reach its goal position. Since the vehicle will only start moving once the goal position becomes available, the further away from the goal position it is when it starts moving, the longer it will take for it to reach the position. Thus, a good queue position will likely be reasonably close to the target position.

But what if there are two possible queue locations, where one is slightly closer to the target, but it will take the vehicle twice the time to reach the queue position? This will have two drawbacks: First, if the goal becomes available while the vehicle is en route, then the extra travel time to the queue position will be time lost. Second, moving more then necessary will cause unwarranted wear and tear on the vehicle. Thus, a good queue position will be along or reasonably close to the shortest path between the starting position and the goal position.

Furthermore, if another vehicle needs to move past a position, that could cause problems. Either the other vehicle would have to take another route, which could possibly be significantly longer, or the queuing vehicle would have to move out of the way. Both are undesirable for the same reasons as above–they could cause delays and extra wear and tear.

Similarly, we need to make sure that the queue position allows the vehicle occupying the goal position to actually leave. Otherwise, we would at best need to move as in the previous case, and at worst cause a deadlock.

So, to summarize the criteria for a good queue position:

- Vicinity - When the target is once again available it is good if the queuer is in the near vicinity of the target. This minimizes the lost time that the desired target is not occupied and therefore streamlines the vehicle switch.
- Not a Detour - A good queue position is a position that does not require substantial extra movement to reach.
- Not Obstructing - We want the queuer to not obstruct the path of other vehicles in such a way that causes the entire system to be delayed.
- Not blocking - As a opposite force of the vicinity demand, this means that we should not be so close to the target that we block it from exiting either.

Now, once this is settled, we can start to take a look at the tools we will need to figure out what positions will fulfill these criteria.

2.3 Graph theory

A graph or network is a structure which consists of points–commonly referred to as nodes or *vertices*–and edges connecting these vertices. The field of graph theory began in the 18th century with Leonhard Euler and the "Seven bridges of Königsberg" [10], where Euler invented what would become graph theory in order to prove that

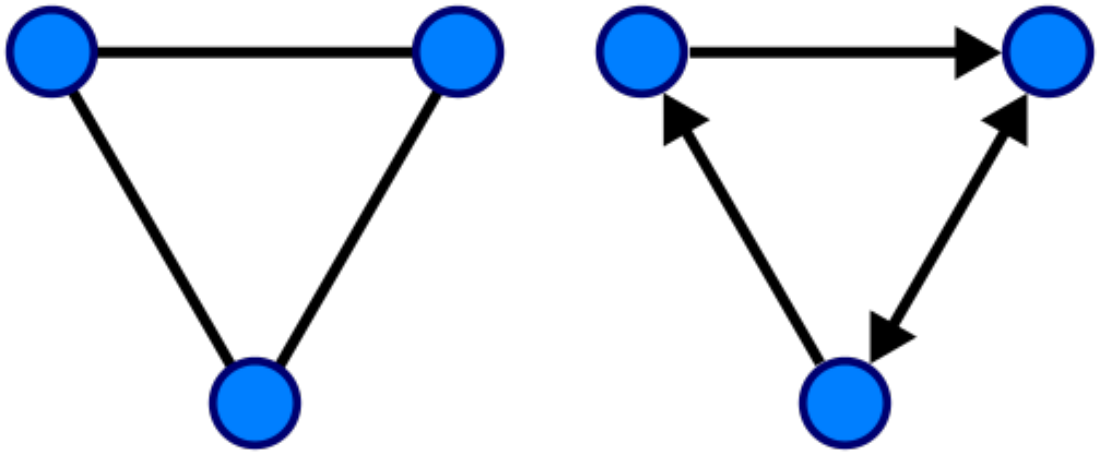
the problem was unsolvable.

In essence, graph theory offers ways of analysing systems which can be described as stationary points (vertices) and means of moving between these (edges). This makes it very useful in describing path finding problems, and many of the most famous algorithms for solving these, such as Dijkstra's algorithm, have been formulated using it.

In this particular case, the layouts are represented by a type of graph, and use some different names for their componants. Vertices, here called *points*, are placed at appropriate points throughout a drawing of the space to be made into a layout. Then edges, referred to as *segments*, are drawn between them. At certain points there are *stations*, where vehicles can stop and perform operations. Along segments they can travel in a certain direction at a certain speed.

2.3.1 Directed and undirected graphs

Until this point, there has been an assumption that if there is an edge between two vertices, it can be traversed in any direction. Such graphs are known as *undirected* graphs. For an example, see Figure 2.1a. For many problems, it becomes beneficial to instead view graphs as directed. These graphs are known as *directed* graphs, or *digraphs*. See Figure 2.1b for an example. This means that a connection from vertex A to vertex B in a graph only allows travel in that direction, while moving from B to A requires an additional connection.



(a) An undirected graph. All edges are symmetrical.

(b) A directed graph. The top and left edges have a single direction, while the right edge is bidirectional.

Figure 2.1: Different types of edges in a graph.

Directed graphs allows for the modeling where travel in both directions is not necessarily possible, like for instance traffic. This distinction has consequences in some of the key aspects of this work, specifically *Network Flow* and *Path Planning*. Before we get in to that, we should probably explain more precisely what a path means in this context.

2.3.2 Paths in graphs

As mentioned above, the edges of a graph and whether they are directed or not, dictates how movement is allowed between different vertices. Stringing multiple such movement together in order to move between vertices that may not be directly connected is what will be referred to as a *path*.

A path is in our case defined as a set of vertices that begin with our starting vertex and end with our goal vertex. The set must be connected in such a way that it is possible to travel on directed edges from one vertex to the next until the last vertex is reached.

Definition 2.1. *A path P consists of a sequence of vertices that are connected by edges such that is is possible to travel from vertex to vertex sequentially.*

If there exists at least one path between every two points in an undirected graph, the entire graph is said to be *connected*. Likewise, a directed graph is connected if there exists a path between every pair of vertices; however, that path only needs to exist in one direction. If a directed path exists between every pair of vertices A and B, both from A to B and from B to A, then that graph is called *strongly* connected. The book *Network flows Theory* by Ahuja, Magnanti and Orlin [11] explains these and more definitions on graphs.

Finding such paths in a graph is a field of its own, which we will dive deeper into in Section 2.4. For now, we will settle for making a few definitions. When finding paths, one is usually interested in finding the shortest possible path. Generally, given a cost for traversing edges in a system, this would mean finding a path such that this cost is minimized. In the case where all such costs are equal, that would simply mean finding the path traversing the least number of edges. In this case, the cost of an edge will be given by the time it takes for a vehicle to travel that edge, plus a constant weight that is defined by the operator of the system. We will refer to the cheapest paths between two vertices as *optimal* paths.

Definition 2.2. *A path is optimal if it is impossible to find a path of lower cost between a given start and goal position in a given graph.*

In addition to this, we will sometimes be interested in finding paths that are not necessarily–or even specifically *not*–optimal. These will be referred to as *suboptimal* paths.

We are further interested in looking at minimizing both total travel time and distance traveled. In this regard it might be useful, or even necessary, to take paths which are not optimal. It may still be desirable to make sure that we are not retracing our steps. In other words we want a path in which no point is repeated. Such paths are referred to as *simple*.

Definition 2.3. *A path is simple if no points appear in the sequence more than once.*

2.3.3 Betweenness Centrality

Something that has been central to this thesis has been *betweenness centrality* [12]. This is the measure of how central a given vertex (or in some cases, edge) is to a network. This is measured by calculating the optimal path between every pair of vertices in the network, and counting how many paths pass through each vertex. The more paths that run through a given vertex, the more central it is to the network.

In our thesis, we will be using a slightly modified version of this measure. Instead of counting the paths between every pair of vertices, we will choose certain vertices of interest. This will be explained in detail in Section 3.2.1.

2.4 Path planning

A central problem in both graph theory and other fields where moving agents are studied is the problem of pathfinding - finding the shortest path between two points. The most famous algorithm for solving this problem is likely Dijkstra's algorithm [13]. While we have not implemented this algorithm ourselves, it has been used extensively in the work, and since it might be interesting in discussing our results, we will describe it below.

2.4.1 Dijkstra's algorithm

Dijkstra's algorithm [13] is a method for finding the shortest path between two points in a graph. It works by dividing all vertices and edges into three categories each. Vertices are divided into categories A, B, and C, where A is the vertices for which the shortest distance from the starting point is known, B is the vertices which are adjacent to A, and C are the rest of the vertices. The edges are divided into categories I, II and III, where I is the edges from the starting point to all vertices in set A, II is the edges which represent the shortest path from each vertex in set B to a vertex in set A, and III is the rest of the edges.

The algorithms starts by placing the starting vertex in set A. It then works in two repeating steps. The first of these steps is to take the vertex most recently moved to set A, and examine all edges connecting in to vertices in set B and C. If a vertex is in set B, check if the edge provides a shorter path than those already known to that vertex. If it is, it replaces the previous edge to that vertex in set II. If a vertex is in set C, it is added to set B and the edge is added to set II.

The second step is to look at the vertices in set B. The one with the shortest path to the starting point, found by combining it's corresponding edge in set II with the edges in set I, is moved to set A, and the edge is moved to set I.

Steps 1 and 2 are repeated until the goal vertex is moved to set A, meaning that the shortest path to the goal vertex has been found.

It is worth noting that in the implementation of this algorithm we have used, the cost of a path is given by its *weight*. This weight is defined by taking the travel time of a segment, and letting the operator manually add a weight to this, in order to "discourage" the system form using certain paths.

2.4.2 Multi-agent path planning

In the last few decades, problems where it has be necessary to plan the paths of several vehicles in one system have become increasingly common, in everything from video games [6] to AGV systems [14, 15]. Naively, one could think that in order to plan the paths of several vehicles, you could simply use the traditional path planning algorithms presented in the previous chapter for each of the vehicles. In reality, however, this is rarely possible due to the constraints which often arise. Conditions such as vehicles not disappearing once they reach their target, and not being able to move through or even past each other, result in the problem being NP-complete [15, 7], meaning an optimal solution cannot be found in polynomial time. Fortunately, several methods for finding solutions that are "good enough" have been constructed, such as the MAPP algorithm by Wang [4].

Papers such as [5] and [7] try to optimize the path finding while still preserving the optimality. The author of [5] presents a conflict-based search which is a two layered algorithm. At the high level a conflict-tree is constructed with constraints on the lower level of the algorithm. The lower level of the algorithm performs individual searches for the agents within the bounds of the constraints set by the higher level of the algorithm. A continuation of the conflict-based search is also presented in [5] which introduces meta-agents. Meta-agents are groups of agents which share a conflict and needs to be solved together as another MAPP problem.

Unfortunately, it is impossible to find an unobstructed path to a position that is itself obstructed, and that is where our work will come in. In short, it will attempt to intercept these situations, and direct vehicles with goals that are obstructed by other vehicles to suitable intermediate goals, allowing the MAPP algorithm to work without interruption.

3
Method

In order to choose a suitable queue position, we use several methods to evaluate the various points in the layout. In order to determine the fitness of a point as queue position, we will calculate the *time loss* for occupying said point. This will be done by calculating how much longer it would take to reach the target, as opposed to using the optimal path; an expected time loss based on the likelihood that the vehicle will have to move from the position and the time it would take to move; and finally, how long it will take to move from the queue position to the target position. Before diving in to the details of how this is accomplished, we will give a quick overview.

3.1 Overview

As we are working with a company and integrating our solution in their system we had some requirements we had to build our solution around. First and foremost, the MAPP algorithm for the entire system will be run every few seconds, leaving relatively small margins for additional calculations. Our algorithm will only have a few hundred milliseconds to work and computational efficiency is a top priority. Therefore, having information pre-calculated and available for simple look-up is desirable. While some information is available to be calculated this way as the system is initialized, some crucial information will require knowing the state of the system at the time of calculation. Thus, we will start by dividing the solution into two parts.

In the first part of the solutions we calculate information that does not depend on the current state of the system. This is performed at system initialization which allows for more exhaustive and time consuming calculations. Analyzing the graph and finding choke points is done in this part of the solution. Information about the flow of orders between stations in the system are used to calculate the the flow and the average traffic of all points. For each station in the system, what other points are used when planning paths from that station to other stations is also calculated and is further explained in section 3.2.2. The first part of the solution will be referred to as the *static evaluation*.

The second part of the solution consists of taking into account the current state of the system to calculate useful information. This means looking at where the vehicle in question is currently located and where it is going, as well as what the other vehicles in the system are doing. We call this part of the solution the *dynamic evaluation*. The dynamic evaluation is performed every time the MAPP algorithm

searches for paths and finds that two vehicles have a common target. The fact that it is performed along with every iteration means that it needs to be light-weight in terms of computational demands in order to not negatively impact the performance of the system as a whole.

3.1.1 Defining a penalty function

After we have analyzed both the static and dynamic information about the system we need a measure to compare the different potential queue positions. As we stated in the introduction we are not looking for a optimal queue position. Without this demand and instead focusing on fast calculations we decided to shift our point of view. To reach our goal of finding the best queue position we are going to focus on penalizing positions which does not meet our definition of a good queue position. The position with the least penalty will be our "good enough solution". Since it is throughput we are interested in maximizing, we decided to give each position a penalty that represents the time lost queuing at that position. This is a estimation of lost time and in many cases a overestimation when we are missing data to decide the time loss more precisely. The penalty function is shown in equation (3.1).

$$T_{tot} = T_{gen} + T_{spec} + T_{other} + T_{detour} + T_q \tag{3.1}$$

Equation (3.1) has the following variables:

T_{gen} =General flow penalty (Section 3.2.1),
T_{spec} =Station specific, exit path penalty (Section 3.2.2),
T_{other} =Other planned paths penalty (Section 3.3.2),
T_{detour} =Detour penalty (Section 3.3.3),
T_q =Time from queue position to target (Section 3.3.4).

3.2 Static evaluation

The static part of the evaluation consists of a statistical analysis of an order list representing distribution and frequency of orders in the system during maximal load[1]. The analysis of the order list is used to approximate the likelihood of a vertex in the graph being passed by a vehicles at any given time, based on a version of the centrality discussed in Section 2.3.3. The likelihood is then converted to an expected loss of time for the queuing vehicle based on the probability of it having to move from its position and the time lost by doing so.

3.2.1 Statistical path usage

At system initialization some basic metrics of the layout can be evaluated. The layout in conjunction with the order list is used to calculate the probability that a

[1]The initial idea for the static evaluation was to simply calculate paths from every station in the system to every other station in the system. After initial tests, this proved unsatisfactory. The initial idea used an incorrect assumption of a uniform distribution of travel between stations.

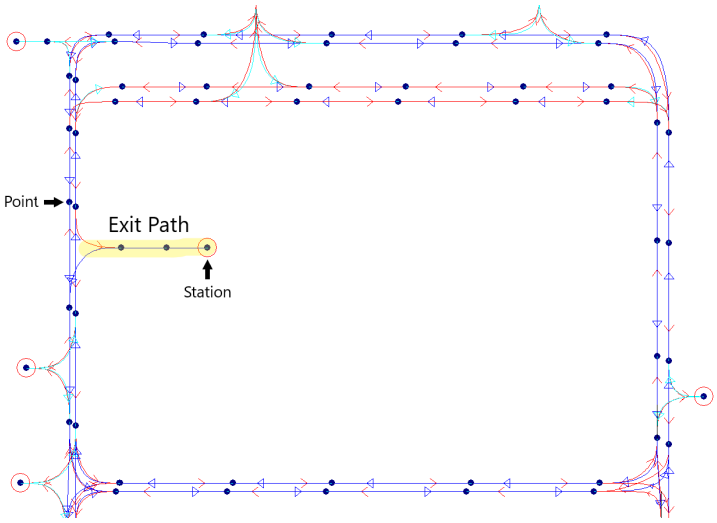


Figure 3.1: An example of a exit path in a layout. If a vehicle is at the marked station it always needs to traverse the exit path to leave the station.

vehicle will pass a certain point at any given time. This can then be converted to concrete expected values of lost time using some additional estimates.

So, how does this work? First of all, all paths for the order list are calculated. For each order in the order list, a path is calculated from the fetch point to the drop point. To account for the movement between drop points and fetch points, a path is calculated between the drop point and the fetch points of the last few orders. Then, for each point in the system, a count is done of how many of the paths block it. It should be noted that the count of the paths from the drop points to the fetch points is divided by how many fetch points it plans paths to. These counts are then used to calculate what fraction of all paths will block a certain point. Once this probability of a road being used has been calculated, the expected value of time lost can be calculated using some more approximations.

At this point it should be noted that some systems are designed with dedicated entrances and exits to their stations, while others are not. The solution described above works fine for systems that do not differentiate between entrances and exits for stations, but it may present a problem for those that do. Presumably, for every vehicle that enters a station, one will also exit. This would mean that the system penalizes queuing by exits as much as it does queuing by entrances. In this case, a bias towards queuing by the entrance while leaving the exit free for the occupying vehicle would be desirable. In order to achieve that, we calculate what paths are used to exit stations.

3.2.2 Exit path

In order to make sure that there is a bias against blocking a path the preceding vehicle will need in order to leave the station, a second statistic is calculated. Here, the paths of all orders with the goal station as start point are calculated and added to the penalty.

We call the points which are always used to exit the station for “exit path”. The

exit path will always be heavily penalized. When the exit path divides to one or more paths agents get the option to choose which direction to go. Points that are only rarely used will be more suitable for a queue position.

Important edge cases here is if a station does not have enough data to show what points are part of the exit path. If only one path leads away from the station this entire path will be looked at as a exit path. This is however not as big of a problem as it seems. Stations which are heavily used and are our prime candidates for needing a queue position and will have most, by definition, order from or to it if we look at sufficiently many orders. We do however want a general solution and stations which do not have much traffic might also need a queue position. The exit path will still be heavily penalized but the rest of the path as well. But since we know that the exit path ends when there is a fork in the graph we also know that we will have points which are not penalized about where the exit path ends.

In the case of no data of orders from a station we are at a risk of assigning a queue position in the exit path. Our thoughts on this is that with enough data this will happen very rarely and when it does the only cost is that the agent will have to move once the agent it is queuing after is done with its operation.

3.2.3 Expected time loss

Once the probabilities of each point having a path planned over it have been calculated, it is then possible to calculate the expected loss of time from standing at that point. The formula is simple: The time it takes to move from that point and back, multiplied with the probability of having to move in a given time unit, raised to the power of how many time units the vehicle will remain at the position. The expected value of time lost due to extra movement when choosing a queue position is given by

$$T_{loss} = T(1 - (1 - p)^K). \tag{3.2}$$

Here, the value of T is how many milliseconds it would cost to move out of the way and back if another vehicle needs to pass through the queue position and the value of K is an estimation of how many seconds the vehicle will occupy the queue position.[2] Thus the probability p_k that another vehicle will pass through the point k in a given second is

$$p_k = \frac{\sum_{j \in P} v_{kj}}{T_s \sum_{i \in V} \sum_{j \in P} v_{ij}},$$

where T_s is the time span of the order list, P is the set of paths, V is the set of vertices, and

$$v_{ij} = \begin{cases} 1 & \text{vertex } i \in \text{path } j \\ 0 & \text{otherwise.} \end{cases}$$

[2] T and K could likely be estimated in real time based on the local characteristics of the layout, but due to time constrains on this project, we opted to use fixed values based on estimations by members of the team who develop the System Manager NG.

3.3 Dynamic evaluation

In the second part of the evaluation, which is performed in real time, several more factors can be taken into account. The starting and goal position of the vehicle in question can be used to determine which points in the system are relevant to evaluate at all, and since the goal position is known, the exit paths of that particular station can be looked up. Further, planned paths of other vehicles can be taken into account.

A dynamic evaluation is performed for each of the agents in need of a queue position every time a path planning takes place. This means that as agents approach their goal their queue position is continuously updated. If a new, better, queue position is found, then this is used instead. This gives the system some more flexibility and an ability to adapt as new routes are planned or the goal position becomes available.

3.3.1 Search space

First of all, in order to reduce computational costs and eliminate some outliers, a search space is computed for a vehicle when a queue position is requested. Outliers are points that would mean a significant detour in order to reach, and thus are unlikely to provide a suitable queue position.

The search space is found by calculating a set of suboptimal paths from the position of the vehicle to its goal, and limiting search for a queue position to a point in the set of points spanned by these paths. This is based on an assumption that most suitable queue positions will be found along the optimal or a suboptimal path from start to goal.

The algorithm starts by placing the starting position in a set we will call A, and then finding the optimal path from a point in set A to the goal point. Then, this path is "blocked" by removing the segment at the middle of the path, and the point immediately after said segment is stored, in a set we will call B. Then, another path is found. This is repeated c times, and that is in turn done for each point in set A. Once that is done, the points in set A are replaced with the points in set B, while set B is emptied, and the entire process is repeated. For each iteration, the number of paths found from each point is also increased by d, in order to increase resolution of the search space closer to the goal position.

This can then be performed to the desired *search depth*. We found that a search depth of 3 gave a reasonable balance between time needed to perform the calculations and sufficient resolution of the search space. For larger values, few additional points were found, at a significant increase in computational cost. As for c and d, we found that values of 4 and 2 respectively, resulting in 4, 6 and 8 repetitions for each of the iterations, gave a good balance between cost and sufficient size of the search space.

3.3.1.1 Algorithm for finding the search space

```
Add starting point to B;
for i < search depth do
    Move all points in B to A;
    for each point in A do
        for j < c + d · i do
            Find path from point to goal;
            if path exists then
                Block middle segment;
                Add point after middle segment to B;
            else
                break;
            end
        end
    end
end
```

For each point in A, the first path found is the optimal. Subsequent paths are only accepted if they are at most four times as long as the optimal path. If no such path is found, the loop breaks. This serves two purposes - it eliminates positions that are unlikely to contain a suitable queue position, while reducing unnecessary computation.

This algorithm does not guarantee that we find the best queue position. It does however give us a set of possible queue position that atleast does not have a high penalty for being on a path which is far from optimal. Since the search space finder is run repeatedly as the queuer is traveling towards the target the middle section that is blocked also changes. This allows for potential new points to be included in the search space in every iteration. If we keep the last queue position in a memory we can also make sure that we do not get a worse queue position.

3.3.2 Paths of other vehicles

While the static evaluation described previously aims to steer queue positions toward points where a vehicle is less likely to obstruct traffic, it does not take into account where the actual vehicles in the system are at any given moment. Thus in order to make sure that we are not only avoiding positions where traffic is to be expected, we look at the currently planned paths of the other vehicles to determine where a queuer would actually obstruct traffic.

Since this step takes place before the paths are planned for this iteration of the system, the only paths available are the paths of the previous iteration. Some vehicles may however have been given a new goal, we must therefore start by validating the paths of the previous iteration by checking them against the goals of the current iteration. If they match, a path is assumed to still be relevant. Otherwise, it is disregarded. In order to calculate the penalties for planned paths for a given point in the search space, a few things need to be calculated.

Firstly, we need to know which other vehicles plan to traverse the potential queue

position, and when. The time of their arrival needs to be compared to the planned arrival time of the queuer. These comparisons will result in a three possible cases listed below.

- The queuer arrives first and the other vehicle arrives while the queuer is still occupying the queue position.
- The queuer arrives first but is able to move to the target before the other vehicle arrives.
- The other vehicle traverses the queue position before the queuer even arrives.

The last two cases cause no problem and only in the first case penalty needs to be added.

The system saves paths as the number of segments (or points) between the start and goal. Every segment has the time it takes to traverse it but the time is unknown to us in this step of the algorithm. Which vehicle arrives first is therefore unknown to us and we have to take into account that for example 14 segments might take longer time than 16 segments.

Next follows a description of how we calculate the risk that a shorter path takes a longer time to traverse than the longer path.

To calculate this risk we start by making the assumption that X_i, the time to reach the next point, is taken from some distribution $\mathrm{D}(\mu,\ \lambda)$ with mean μ and variance λ. This gives the variance

$$\mathrm{Var}\left(\sum_{i=1}^{n} X_i\right) = \sum_{i=1}^{n} \mathrm{Var}(X_i) = \sum_{i=1}^{n} \mathrm{Var}(X) = n\mathrm{Var}(X) = n\lambda, \tag{3.3}$$

where n is the number of points in a path. Now we can use Equation (3.3) and look at the variance between two paths, X and Y.

$$\mathrm{Var}(X - Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) = n\lambda + m\lambda = (n + m)\lambda.$$

Here we have that n is the number of segments in path X and m is segments in Y.

From the central limit theorem we can assume a normal distribution if the path is long enough. Let the difference between two paths be called Z, this means Z is also a random variable as
$Z \sim \mathcal{N}\big(\mu(n - m), \lambda(n + m)\big)$.

We are however not interested in any exact solutions but a simplification is enough for us which makes our solution less complex. Let us therefore only look at Z if Z is less than one standard deviation from the expected value. The lower and upper bound is therefore

$$(m\mu + \mu(m - m)) \pm \sqrt{\lambda(2m)}.$$

From checking the distribution of X_i in one system we could see that a histogram formed a exponential distribution approximately with parameter 1. From this we make the approximation that $\frac{\sqrt{\lambda}}{\mu} \approx 1$. This gives us the very simplified lower and upper bound

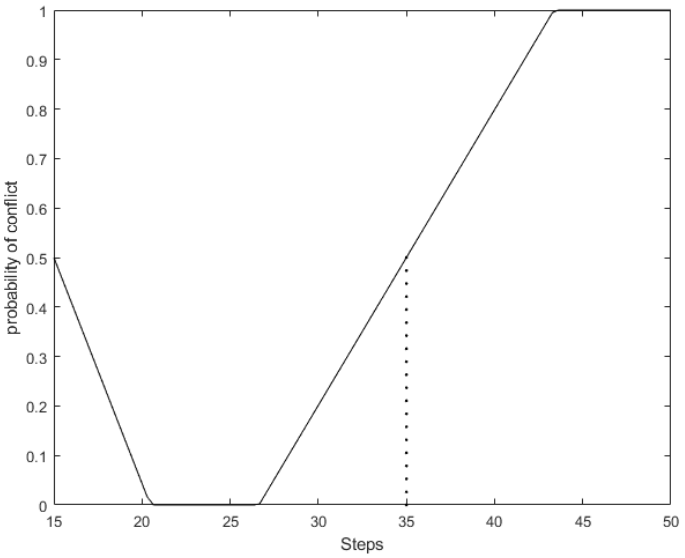


Figure 3.2: A graph showing the probability of conflict depending on when the queuer is expected to arrive to the potential queue position. The closest planned paths (in time) are at the 15 and 50 step marks. The operation time is set to 15, which means that if the queuer arrives to the queue position before the 35 step mark, then it will likely leave the queue position before a conflict arises.

$$m \pm \sqrt{2m}.$$

In this interval we approximate the risk with a linear equation. This can be seen in Figure 3.2 in the intervals $[15, 15 + \sqrt{30}]$ and $[35 - \sqrt{70}, 35 + \sqrt{70}]$.

As it can be seen in Figure 3.2 we have made some more assumptions and simplifications which are worth noting. The probability is often a overestimation for two reasons:

- There should be a lowering of probability as the predicted arrival time goes to the expected arrival time of the other vehicle planned arrival (in this example it's 50 and lets call this vehicle A). The main reason why there isn't is that we do not know if another vehicles (call this B) have a planned path just after. If this was the case, we would have a conflict even if the queuer arrived after vehicle A.
- As stated previously we do not know when the queuers original target will become available. It might be that a queue position is only needed for a few seconds or even just as a new target that the MAPP algorithm can use to plan its path.

These overestimations are made with the assumption that it's better to increases the penalty for points that there is uncertainty about and leaving points which are a better choice for a queue position unpenalized.

3.3.3 Divergence from optimal path

In order to avoid unneeded delays as well as excessive wear and tear on the vehicles, we want to make sure that any unnecessary movement is minimized. It is feasible that an otherwise highly suitable queue position is placed in such a way that the vehicle will need to take a significant detour in order to reach it. If the goal position becomes available while the vehicle is en route to the queue position, then this would lead to decreased throughput.

Since each queue position is placed along either the optimal path or a suboptimal path, a penalty can be added based on how much longer the total travel time of the path will be from the start to the queue, and from queue to goal, compared to the optimal path from start to goal.

In terms of actual time to reach the goal, this is a worst-case estimate. If the goal becomes available while the vehicle is moving along the suboptimal path, with no means of returning to the optimal path, then the full extra time will be lost. If the vehicle reaches the queue position before the goal becomes available, however, throughput is not affected. The bias toward choosing a shorter total path to the goal should still result in vehicles minimizing unnecessary movement.

3.3.3.1 Algorithm

The formula for finding this value is fairly simple. First of, we will call the total travel time for the optimal path from the start position to the goal position T_o. Then, we will find the paths from the starting point to all potential queue positions, and from these positions to the goal position. For each position p, the penalty will be the travel time from start to p, plus travel time from p to the goal, minus T_o. Thus, a point along the optimal path will have no penalty, a point with only a slight detour will have a minor penalty, and a point along a much longer path will receive a significant penalty.

3.3.4 Travel time

This section provides a motivation to why the last penalty is also probably the most important one. We start this motivation with the simplest case.

The optimum solution for one agent to travel to one target is for that agent to use the shortest path to said target. This still holds true for two agents with different targets given that two agents do not cross paths at the same time. When two agents cross paths at the same time we call this a conflict. We also call it a conflict if two agents share a common target. If we have a conflict, the total time to complete all orders is no longer equal to the longest optimum path. If we look at the case where we have a conflict and a common target we have that the agent which reaches the target second has to wait if the preceding vehicle is not done with the target. We denote the time the slower vehicle has to wait by T_{wait}, and denote the operation time at the target station by T_O. The total time for a system with two orders and two vehicles if a queue position is needed, is

$$T = \max_i(T_i) + T_{wait} + T_O, \quad \text{with} \quad i = 1, 2. \tag{3.4}$$

We can further divide the agent's drive times into $T_i = T_i^{to} + T_i^{from}$, where T_i^{to} is the time to drive to the queue position and T_i^{from} is the time it takes for the agent to travel from queue position to target.

We can also show how T_{wait} depends on T_i:

$$T_{wait} + \max_i(T_i^{to}) = \min_i(T_i) + T_O \qquad i = 1, 2 \tag{3.5}$$

Combining equations (3.4) and (3.5), we can see that the total time only depends on $\max T_i^{from}$ and some factors that we can not influence (or is at least out of the scope of this thesis). In a two vehicle system the most important fact about a queue position is that it is close to the target.

If we introduce more vehicles, the equation becomes much more complex and it is hard to draw general conclusions.

Queue positions are often needed for stations which in some way are bottlenecks for the entire system. This means that keeping a low downtime for those stations are especially important and might outweigh the time loss given by obstructing another vehicles path to a less critical station.

3.4 Assigning the queue position

Once all the penalties for the entire search space are calculated we can feed them into equation (3.1) and receive the total penalty for all the points. The lowest penalty is chosen and we almost have our queue position. One additional check still needs to be made.

A vehicle often takes up more space than the point or edge it is occupying. To make sure no collisions occur there is a function that calculates which other points and edges needs to be available for a vehicle to start traversing an edge. We call it, that the edge is *blocked* by said points and edges. Blocked points of a path is always calculated in our algorithm when a path is calculated. The path and the blocked points are all the points a vehicle will block while traveling a certain path.

It is important to note that points and segments that are *occupied* by a vehicle when standing at a certain point–that is, a vehicle standing at point a also covers point b–are not included in this set of blocked points, since they are considered to be *implicitly* blocked. This meant that the points needed for the vehicle to stand on the first point in the path are not included. It is therefore not certain that points that are used by the vehicle standing on the station are excluded from the search space.

The last check we need to make is that the queue position is reachable given that there is a vehicle at the target station. A quick Dijkstra search finds the path and checking the blockings for all of the points in the path gives us the answer to if the queue position is reachable. If it isn't reachable, it is excluded from the search space and the next best queue position is checked for reachability. The reachability checks continue until a viable queue position is found.

4
Results

The performance of the solution is measured by comparing the *throughput* of the system–the amount of orders completed in a set amount of time. A number of different comparisons will be made. First, the complete solution will be compared to the system with no solution implemented, as well as a simple "As close as possible" solution, in order to determine whether the solution offers a meaningful improvement at all. Then, the complete solution will be compared to variations of the solution with certain components removed in order to determine what components are most important.

These tests have been performed in simulations using real customer system. As such, we are unable to show the systems in detail, and will simply be referring to them as system A, B and C.

System A is a relatively small system, running just a few AGVs. System B is a somewhat larger system, running a moderate number of AGVs. System C is the largest of the systems we test in terms of vehicles.

4.1 Optimal queue position vs closest possible vs current solution

First of all, we will be testing our solution against the "standard" solution, which works by simply moving all agents toward their intended goals until that goal is occupied by another vehicle. In addition to that, we will be testing against another solution implemented by our supervisor, *Multiple-vehicles* (MV). In essence, it works by calculating how close an AGV can move to a target without blocking the exit, and moving to that point.

The tests are run using a testing platform where the number of AGVs and number of orders to be completed is specified. The system then tries to complete these orders, and if successful returns the total test time, as well as the average time to complete an order. The latter will be used to compare throughput. If the system is unable to progress–meaning, the MAPP algorithm cannot find a solution–for more than 10 minutes, the test will fail.

4.1.1 System A

System A is a smaller system, running only a few AGVs. In regular testing, the number is modulated between one and five, with the last of these occasionally failing. Initially, our solution had problems with this system, failing frequently at only three

# vehicles	μ_Q	μ_S	μ_M	σ_Q^2	σ_S^2	σ_M^2	$r_{fail}^{(Q)}$	$r_{fail}^{(S)}$	n_Q	n_S	n_M
2	53.9	53.4	-	0.1	0.62	-	0	0	20	14	0
3	40.4	42.5	42.8	4.4	2.8	5.6	0.056	0.050	16	20	19
5	38.5	42.3	-	2.5	5.1	-	-	-	12	10	0

Table 4.1: The results for system A. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n the sample size. The indices Q, S and M refer to Queue assignment, Standard and MV. Unfortunately, the failure rate for 5 vehicles is unavailable, along with the failure rate of MV for 3 vehicles, due to the data accidentally being overwritten by the testing platform. The histograms for 2 and 3 vehicles can be seen in Figure 4.1 and Figure 4.2 respectively.

Figure 4.1: Histogram showing the average time in seconds to complete an order in system A running 2 vehicles. Each data point is an average of 50 orders. More details can be found in in Table 4.1

vehicles. The cause of this was discovered when testing system B, and once a fix was implemented, we saw significant increases in performance. The fix is explained in Section 3.4. The system was tested at two, three and five vehicles.

At two vehicles, our solution actually performed slightly worse than the standard solution, as can be seen in Figure 4.1 and Table 4.1.

At three vehicles, we start to see some possible performance increases. In Figure 4.2, we see a histogram of test runs using our solution (queue assignment), the standard solution, and the simple solution implemented by our supervisor (MV). The throughput for queue assignment is about 5.2% better than the standard solution. This means that in the same time, our solution can do 5.2% more orders. The throughput of queue assignment is 5.9% better than the MV solution. An overview of the results can be seen in Table 4.1.

Figure 4.2: Histogram showing the average time in seconds to complete an order in system A running 3 vehicles. Each data point is an average of 50 orders. More details can be found in in Table 4.1

# vehicles	μ_Q	μ_S	μ_M	σ_Q^2	σ_S^2	σ_M^2	$r_{fail}^{(Q)}$	$r_{fail}^{(S)}$	$r_{fail}^{(M)}$	n_Q	n_S	n_M
5	48.0	49.7	48.8	0.34	2.1	2.5	0	0	0	30	17	37

Table 4.2: The results for system B. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n is the sample size. The indices Q, S and M refer to Queue assignment, Standard and MV. These numbers are visualized in Figure 4.3.

4.1.2 System B

System B is a larger system then system A, both in terms of the size of the graph and intended number of vehicles. We only ran this system with 5 vehicles, which is the amount of vehicles it is designed to handle. Initial results were discouraging with low performance and frequent failures, but after identifying and fixing a potential bug with how our solution treated points that were blocked by other vehicles, performance improved and the failure rate decreased. The results after this fix can be seen in Figure 4.3. An overview of the results are also presented in Table 4.2.

4.1.3 System C

System C is only slightly larger than system B in terms of layout size but running up to 10 agents. We have tested it at 5, 8 and 10 agents, and have seen encouraging results. At 5 AGVs, performing 50 orders, an increase in throughput of about 4.3% compared to the standard system was seen. Figure 4.4 shows this, with the sample size being 15 and 14 runs, respectively. This resulted in a mean of 46.5 seconds per order and a variance of approximately 1.55 for the queue assignment, compared to 48.7 and 0.37 for the standard.

As for 8 vehicles, we saw an even bigger gains, of almost than 9.0%, as can be

Figure 4.3: System B with five vehicles running. Queue assignment has a mean of 48.0 seconds, standard has a mean of 49.7 seconds, and MV has a mean of 48.8 seconds. This puts our solution at 3.5% higher throughput than the standard, and 1.6% higher throughput than MV. These numbers are summarized in Table 4.2

# vehicles	μ_Q	μ_S	σ_Q^2	σ_S^2	$r_{fail}^{(Q)}$	$r_{fail}^{(S)}$	n_Q	n_S
5	46.7	48.7	1.6	0.4	0	0	15	14
8	36.8	40.1	1.9	2.3	0.034	0	18	19
10	38.5	42.3	2.5	5.1	0.48	0.44	11	12

Table 4.3: The results for system C. μ is the mean in seconds, σ^2 is the variance in seconds, r_{fail} is the failure rate and n is the sample size. The indices Q and S refer to Queue assignment and Standard.

seen in Figure 4.5. Like previously, each test consisted of 50 orders, and here the sample sizes are 18 for the queue assignment and 19 for the standard. The Means is 36.8 versus 40.1 average seconds per order, and variances of 1.91 and 1.76.

Finally, we tested the system at 10 vehicles. This results in frequent failures for both queue assignment and standard. Success rate is roughly 50% for each. The histogram of the average order completion time can be seen in Figure 4.6. We see a performance gain of nearly 10.0%, with mean average order completion times of 38.5 versus 42.25, and variances of 2.5 versus 5.1. These results are summarized in Table 4.3.

Figure 4.4: Average number of seconds to complete an order in system C. The system is running 5 AGVs, and performing 50 orders per test. The mean time to complete an order is 46.7 s for our queue assignment algorithm and 48.7 s for Kollmorgen's standard algorithm. This is 4.3% increase in throughput for the system using our queue assignment.

Figure 4.5: Average number of seconds to complete an order in system C. The system is running 8 AGVs, and performing 50 orders per test. The mean time to complete an order is 36.8 s for our queue assignment algorithm and 40.1 s for Kollmorgen's standard algorithm. This is 9.0% increase in throughput for the system using our queue assignment.

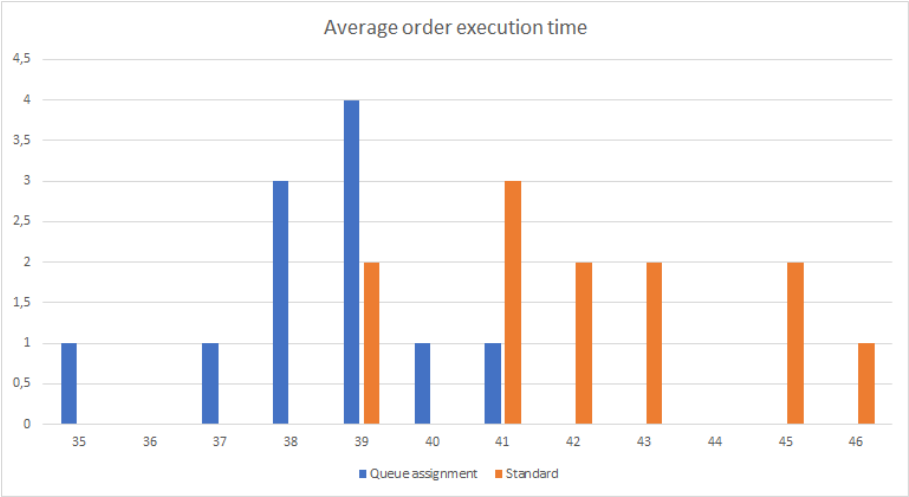


Figure 4.6: Average number of seconds to complete an order in system C. The system is running 10 AGVs, and performing 50 orders per test. The mean time to complete an order is 38.5 s for our queue assignment algorithm and 42.25 s for Kollmorgen's standard algorithm. This is 9.7% increase in throughput for the system using our queue assignment.

# vehicles	μ_Q	μ_N	μ_R	σ_Q^2	σ_N^2	σ_R^2	$r_{fail}^{(Q)}$	$r_{fail}^{(N)}$	$r_{fail}^{(R)}$
8	47.0	49.7	48.8	0.34	2.1	2.5	0.034	0.125	0.26

Table 4.4: The results for system C when testing different component combinations. μ is the mean, σ^2 is the variance, and r_{fail} is the failure rate. The indices Q, N and R refer to Queue assignment, no planned paths and reduced search space.

4.2 Different variations on queue assignment

In order to motivate some of our criteria described earlier in the report, we wanted to vary or remove some features and examine the results.

First we decided to vary two parameters - Planned paths and the search space. For the planned paths, the variation was simply to remove the penalty and see what the effect would be. As can be seen in Figure 4.7, this resulted in a slightly higher mean and the throughput is about 2.2% higher for our solution.

For the search space, we simply reduced the multiple of of much longer than the optimal path any given suboptimal path was allowed to be from 4 to 2. This resulted in a slightly higher mean as well. The throughput of our solution is about 3.5% better. We have summarized these results in Table 4.4.

4.3 Time consumption

In this section the time to find the queue position is shown for the different systems. The times are taken from a random test. The results are presented separately for each of the three systems compared. There is no differentiation between if the agent is located close to the target or very far away. Every search is also done

Figure 4.7: Three variations of the Queue Assignment solution, run on system C for 8 vehicles. The standard configuration compared to one where no planned paths are taken into account, and one where the search space is reduced. The means are 36.8, 37.6 and 38.1 respectively, with variances of 1.9, 2.3 and 1.8.

independently of the previous search.

The total time our algorithm uses to find a queue position can be seen in Table 4.5.

Finding the search space proved to be the most costly operation. The times can be seen in Table 4.6.

System	Average time	Standard deviation	Min value	Max value	Data points
A	4.4 ms	2.7 ms	1 ms	22 ms	230
B	38.0 ms	13.8 ms	1 ms	115 ms	1255
C	68.5 ms	40.9 ms	4 ms	395 ms	1503

Table 4.5: Time to find a Queue position. Data points are the number of queue position requested for the 50 orders of an arbitrarily chosen test run.

System	Average time	Standard deviation	Min value	Max value	Data points
A	3.8 ms	2.4 ms	1 ms	20 ms	230
B	32.2 ms	13.5 ms	0 ms	108 ms	1255
C	63.2 ms	40.5 ms	1 ms	391 ms	1503

Table 4.6: Time to find a search space. Data points are the number of search spaces requested for the 50 orders of an arbitrarily chosen test run.

5
Conclusions

5.1 Throughput performance

Overall, we have seen mainly positive results in terms of throughput gain. Time constraints have unfortunately not allowed us to collect all the date that we would have desired. If we would have had more data, we could have given exact numbers on the improvement of queue assignments and the confidence intervals. With that being said we do believe more data would only strengthen our results.

For system A, shown in Table 4.1, there were minor performance losses for two vehicles, though the sample size is to small to draw any significant conclusions with such small differences. For three vehicles, we started seeing some performance improvements over both the standard solution and the MV solution. It should be noted that the variance of the latter is very high. At five vehicles, we again suffer from little data, but what little data we have does seem to imply significant performance increase. Though, since the variance is extremely high for the standard solution, that is unsure.

For system B we have positive results, of seeming statistical significance. As can be seen in Table 4.2, queue assignment does pull ahead of the standard solution, with about a 3.5% increase in throughput. The MV solution does perform fairly good here, with queue assignment only placing about 1.6% ahead. As can be seen in Figure 4.3, they both appear normally distributed around 48, though the MV has wider distribution–a higher variance–and tend mainly toward higher values.

For system C, we feel more sure of the results, which can be seen in Table 4.3. For five vehicles, we saw a slight increase in performance, though variance was high for the queue assignment solution. For eight vehicles, the normal distributions appear very distinct in the histogram, giving us confidence that the data is somewhat significant. Variances for both solutions are relatively low, and the performance increase of almost 9% is very encouraging. These numbers seem to indicate that the performance increase is larger if more vehicles are involved. More vehicles also means more queue position assignments (see Table 4.5) which probably is one of the reasons for the larger improvement.

The queue assignment solution for system C and eight vehicles experienced rare failures, though we believe they were too few and far between to draw any strong conclusions from. At ten vehicles, we started experiencing significant failure rates for both solutions, though they are of similar size, leading us to believe there may not be enough data to draw any strong conclusions. However, that fact that our solution displays a slightly higher failure rate for both 8 and 10 vehicles is likely indicative of

the fact that it has a slightly higher computational demand, occasionally preventing the system from finding solutions when it otherwise could have.

In this system we also tested some variation on our solution, specifically one without considering the currently planned paths of other vehicles and one where the search space was reduced. The results can be seen in Table 4.4. Overall they are indicative that both components contribute positively to the overall solution, with the complete queue assignment having both lower average time to complete orders, lower failure rate and lower variance. However, it would be prudent to perform much more rigorous test of this, with more combinations of components as well as variations in system and number of vehicles. Due to limited time and resources for testing, we have been unable to do so. As it stands, no significant conclusions can be drawn.

5.2 Time performance

One question that arose along the way of the project was whether the solution would be fast enough to work "online" or only work "offline". Online meaning it would work in real time and make decisions, and offline meaning it could be used as decision support for operators in setting fixed queue positions manually. In general, it seems our solution is suited to work in real time. While it does seem it in certain cases can cause a slight slowdown of calculations, that is inevitable when more work is performed.

As can be seen in the results in Section 4.3 the majority of the time to find a queue position is spent on finding a search space. This means that what was meant as a time saver might actually be costing us time. When looking at the algorithm more exactly, it is the removal of an edge that takes the majority of the time. We suspect that reducing the time to remove an edge significantly is possible. Time constraints and lacking sufficient incentive kept us from implementing such a solution.

If further investigation should conclude that performance could be gained by speeding up the algorithm an alternative to our search space finder is presented in Section 5.5.2.

5.3 Data

As has been discussed somewhat previously, we have suffered from limited data for our results. This has been due to two major factors. First and foremost, each test takes around 45 minutes on average, which means that it takes significant amounts of time to gather significant data for even one solution, in one system, running one number of vehicles. Secondly, we have experienced issues with the testing platform of the company, sometimes resulting in us spending days trying to get a test working before any results are produced at all, and on one occasion causing us to lose days worth of data. We had no prior experience working with the testing platform which might explain our difficulties.

In order to ensure statistical significance of all our results, it is likely that several more weeks of tests would be required, if not more. We do however believe that our

results are consistent enough to indicate that the work has been generally successful, even if the exact numbers of how much better it performs are not certain.

5.4 Challenges

There were a lot of challenges we faced during our work. There were multiple instances of us having ideas regarding a solution, which in the end turned out to be impractical or even impossible to implement in the current framework. Instead we had to find clever alternative solutions.

5.4.1 Time constraint

Searching for paths non-stop is very time consuming and the company's MAPP algorithm is already using significant amount of time and computational power. It is hard to say how much resources our solution would be allowed to to use. As of now the MAPP finds a path and then tries to improve it until it has no more time. The best solution is then used and the process starts again.

The time it takes for our algorithm to find a queue position might take just the time MAPP would otherwise use to improve the throughput on a similar level. It could therefore be interesting to change the time limit for MAPP to see how this affects the throughput of the system.

5.4.2 Layout intentions

The layouts or graphs that we are working on are made by an application engineer. It requires some skill to draw good layouts and the engineer often has a lot of experience in the area. This is of course a good thing in most cases as the engineer can tailor the graph to the transport structure and the factory layout. This also means that the engineer has thought about which stations are regularly going to have multiple AGVs traveling to it and designed the graph with an "intended" queue position. This might help the normal path finder but could cause some complications for our queue position finder. It creates a bias for the intended queue position and prevents dynamic features from influencing the decision. Since all layouts are made this way it is hard for us to test the effect of this.

One of the tools used in the design of the layouts is weights, as mentioned in Section 2.4.1. They are used in order to discourage vehicles from using certain paths unless absolutely necessary to reach the intended goal. This may have unintended consequences for our algorithm, which we have been unable to test or control for.

5.5 Ideas that were not implemented and suggested future work

There are always more things to test and more functions to add to a solution like ours. We have had many ideas along the work of this project that we have not had the time to implement or that were simply not prioritized.

5.5.1 Analyzing the flow in the system

In this work we used a order list to gather information about how the flow in the system would look like. This made it very easy to take input as it only required a list. This list is however created by randomly selecting orders from a transport structure. The transport structure is a table which specifies from which stations or groups of stations to which stations or groups of stations transport is going to be needed. It also specifies how often orders of every type will arrive. While a sufficiently long order list is going to give the same information as the transport structure this is a unnecessary step. The transport structure could instead be feed directly in to the program and analyzed directly to give the flow of the system. Two challenges will be needed to consider before this can be done:

- How will the data be represented and read by the program?
- How do we analyze the transport structure to get valuable information efficiently?

5.5.2 Prioritizing the search space

Finding a search space was our first attempt at minimizing the time it would take to evaluate all possible queue positions. As our work moved forward we noticed that one of the closer points to the queue position was almost always chosen. This can be explained by the fact that the other penalties have a much lower cap if the queuer starts far away from the target. Traveling from one end of the system can take minutes while just moving out of the way for another vehicle does not. This was a fact that we could possibly have used to make the queue position finder more robust. Searching out from the target and ordering the possible queue positions in the order they are found gives us a higher chance to find a reasonable good queue position early in the search. This is because we order them by perhaps the most influential penalty factor.

Bibliography

[1] "The 4 industrial revolutions," https://www.sentryo.net/the-4-industrial-revolutions/, accessed: 2019-05-29.

[2] "Health risks at work should belong to history," https://www.kollmorgen.com/en-us/blogs/_blog-in-motion/articles/emma-anderson/health-risks-work-should-belong-history/, accessed: 2019-05-17.

[3] "The automated guided vehicle (agv) market is on the rise," https://www.kollmorgen.com/en-us/blogs/_blog-in-motion/articles/samuel-alexandersson/automated-guided-vehicle-agv-market-on-the-rise/, accessed: 2019-05-22.

[4] K. H. C. Wang and A. Botea, "MAPP: A scalable multi-agent path planning algorithm with tractability and completeness guarantees," *Journal of Artificial Intelligence Research*, vol. 42, pp. 55–90, 2011.

[5] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artificial Intelligence*, vol. 219, pp. 40–66, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2014.11.006

[6] B. Banerjee and C. E. Davis, "Multiagent path finding with persistence conflicts," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 9, no. 4, pp. 402–409, 2017.

[7] J. Yu and S. M. LaValle, "Optimal Multirobot Path Planning on Graphs: Complete Algorithms and Effective Heuristics," *IEEE Transactions on Robotics*, vol. 32, no. 5, pp. 1163–1177, 2016.

[8] Y. Tavakoli, H. Haj Seyyed Javadi, and S. Adabi, "A cellular automata based algorithm for path planning in multi-agent systems with a common goal," *International Journal of Computer Science and Network Security*, vol. 8, 05 2019.

[9] Y. K. B, C. Markarian, P. Simoens, and A. Reina, "Advances in Practical Applications of Cyber-Physical Multi-Agent Systems: The PAAMS Collection," vol. 10349, pp. 144–156, 2017. [Online]. Available: http://link.springer.com/10.1007/978-3-319-59930-4

[10] G. Alexanderson, "About the cover: Euler and Königsberg's Bridges: A historical view," *Bulletin of the American Mathematical Society*, vol. 43, no. 4, pp. 567–573, 2006.

[11] J. B. Orlin, *Ahuja, Magnanti, Orlin - Network flows Theory, Algorithms and Applictions*, 1993.

[12] L. Freeman, "A set of measures of centrality based on betweenness," *Sociometry*, vol. 40, pp. 35–41, 03 1977.

[13] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, dec 1959. [Online]. Available: http://link.springer.com/10.1007/BF01386390

[14] H. Ma, W. Hönig, T. K. S. Kumar, N. Ayanian, and S. Koenig, "Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery," 2018. [Online]. Available: http://arxiv.org/abs/1812.06355

[15] H. Ma, C. Tovey, G. Sharon, T. K. S. Kumar, and S. Koenig, "Multi-Agent Path Finding with Payload Transfers and," *AAAI Conference on Artificial Intelligence*, pp. 3166–3173, 2016.