Path Optimizations for Storage Systems

HENRIK MÅNSSON

Master’s Thesis at the Department of Computer Science
Supervisor/Examiner: Roger Henriksson
Supervisor: Bo Lindblom

April 2, 2009
Abstract

In production industry there is a constant struggle for gaining higher efficiency, that also includes the storages and especially the buffer storages between the different machines in the production. As the demands for production flexibility increases as well as the efficiency demands, smarter storage systems are needed. That efficiency requirement is a direct result of the economical aspect, which also must be reckoned with. Using automated guided vehicles can be a good compromise for these requirements.

In this project an attempt to create a flexible, yet efficient automatic storage system has been made. It includes proposed solutions to problems like route planning, collision avoidance and deadlock strategies. We also discuss advantages and disadvantages for different techniques, with the main goal of finding the optimal solutions for this specific task.
Referat

Vägoptimering för lagersystem


4 Deadlocks
4.1 Deadlock Conditions ........................................ 25
4.2 Different Strategies ......................................... 26
4.3 Prevent-Strategy .............................................. 26
   4.3.1 Mutual Exclusion ....................................... 27
   4.3.2 Hold and Wait .......................................... 27
   4.3.3 No Preemption ......................................... 28
   4.3.4 Circular Wait .......................................... 28
   4.3.5 Example Implementation of the Prevent-Strategy .... 29
   4.3.6 Summary ................................................. 29
4.4 Avoid-Strategy .............................................. 29
4.5 Detect and Recover-Strategy ................................. 30
   4.5.1 Detecting a Deadlock .................................. 30
   4.5.2 Recovering from a Deadlock ............................ 30
   4.5.3 Summary ................................................. 31
4.6 Deadlock Experiences ....................................... 31
4.7 Conclusions ................................................. 32

5 Future Work .................................................. 33
5.1 Communication with the AGVs ................................. 33
5.2 The Client .................................................... 34
5.3 Visions ....................................................... 34

6 Conclusions ................................................... 37

Bibliography ..................................................... 39
Chapter 1

Introduction

Every company struggles to be more efficient. This is a very big issue for production companies, due to the competition from low salary countries. The companies have two options to choose from: Either they move the production to low salary countries, or they automate the factory as much as they can. The goal is to minimize the number of employees but still keep up the production rate. The price of the product can now be kept as low as the market demands.

But not every job can be made automated, mainly the monotonous jobs where you have to do the same things over and over again. An example of such a job is internal transportation between the machines in the production. Usually there are so called buffer systems between the machines to stabilize the production rate. If e.g. machine A is down, machine B should not starve and vice versa. One solution for this is automated guided vehicles (AGV). That means they are totally automated and guided by a computer system. The main advantage using AGVs is that it is much cheaper to have this automated, compared to having an employee do it manually. A system like this will repay itself very fast.

Such a system still demands certain flexibility. The production rate could change, or rather will change in the future. This will be a major issue if the production rate is going up more than expected and it has not been planned for. You do not want to rebuild the whole system.

One solution for expanding the system could be to install an extra AGV to handle the extended production rate. But having multiple AGVs in the system increases the complexity level considerably, which also creates new problems. Is there a possibility that the AGVs will collide? Maybe the number of possible travel routes needs to be increased to keep the AGVs apart from each other. Is it possible to custom build a solution which automatically adjusts itself to different tracks, number of AGVs and efficiency demands? How should the track be built to get maximum efficiency from each AGV? Is there a possibility deadlocks can occur when having multiple AGVs on the same track?

The keywords are flexibility and efficiency. Is there a chance these could be combined or is it just an unreachable vision?
Chapter 2

The Base System

The base system is a prototype system developed by Texo Application for a company in the packaging industry. They will use two AGVs at first, but may perhaps need a third (and a forth) one later. The time demands won’t be that hard to meet, considering the first machine only produces approximately three items per hour. But it may sound easier than it is, because there is more than just collecting items from a machine that the AGV should do. For example structuring the storage will require quite some time from the AGVs. The AGVs also needs time to charge, which also has to be accounted for. But for a prototype system it is considered good to have time demands like these. All the mechanical work including building the AGVs is done. In a project like this the programming part is most demanding, since the resulting system will become quite big.

2.1 Abstractions

These are the main abstractions in the complete system:

- There are a number of spots to place pallets on.
- The pallets are created so the AGV can run underneath. If it wants to pick one up, it runs up underneath and lift the pallet a few centimeters. Otherwise it just continues forth to the next pallet.
- Several storage spots is connected to form a storage row.
- Each row should (at the moment) be ordered in FIFO (First In First Out).
- To travel between these spots the AGV optically follows a line on the floor.
- There is a home spot for each AGV, containing a charger.
- Every storage spot and turn/crossing contains RFID-tags for positioning.
2.1. ABSTRACTIONS

- The AGV is not running at the exact middle of the line. The position is side shifted a few centimeters in one direction (offset). The reason for that is the fact that the wheels are placed in the middle, and will destroy the lines after some time.

- The AGVs can only run through and pick up pallets in one direction, due to the offset of the above abstraction.

- In the front of the AGV there is a safety system, see the unit on the front left in Figure 2.1 or 2.2. It will first slow down the AGV if something is in front of it and ultimately stop if it gets too close.

- The AGV can execute the following commands:
  
  - The AGV can run forward.
  - The AGV can also run backwards. But this should be avoided, because there is only one line-sensor which is placed in the front.
  - A normal turn for the AGV is a 90 degrees turn in any direction.
  - The AGV is also able to do a 180 degrees turn.
  - The AGV can lift up a pallet if standing underneath.
  - The AGV can of course also lower the pallet.

For a picture of the AGV and a pallet, see Figure 2.1.

These abstractions must be accounted for in the software, although the command part do have potential to change. That part must be made agile.

There are more problems than just lacking a rear line-sensor to be able to run backwards. The AGV only has three wheels, see Figure 2.2. There are two wheels in the back which are used for running the AGV, the third wheel is placed in the front and it is used for steering.

At first it would be more expensive, having to add not just another line-sensor but also a second safety system, which is very expensive. Steering backwards with a three wheeled vehicle is not the same as running forward. It is a bit harder to do and results in different steering routines. Developing this also costs money and keeping down the costs is a must.

2.1.1 Remarks

The AGVs have the ability to automatically dock the charger when they enter their home spots, see the docking part close to the left rear wheel in Figure 2.2. This will make the AGV charge whenever it has nothing to do.
Figure 2.1. The AGV running underneath the pallet.

Figure 2.2. The AGV from underneath.
2.2. NEED FOR A NEW SYSTEM

2.1.2 Definitions

The home spot for each AGV will from now on be described as the home node.

A vehicle is equivalent to an AGV.

An item is considered equivalent to a pallet.

2.2 Need for a New System

The first system was developed using only PLCs.

A PLC (Programmable Logic Controller) is a simple form of computer used in automation and control applications. It consists of a number of input/outputs, which can be both digital and analog. There are multiple languages to program a PLC, where the most used is the so called ladder logic. It is a visual programming language where the programmer connects different components to get the desired output. That language was created to describe relay logic which is fairly outdated today. For more data handling tasks, this language is not recommended.

In this system PLCs was used both to control the AGVs, as well as the server which sends orders to the AGVs. At first it did work out quite well. But later on, with more functions and data handling the system became too complex for using only PLCs. The solution was to put a PC as the server of the system.

2.2.1 PC vs. PLC

There are multiple advantages (in this case) of using a PC instead of a PLC.

- Advantages using a PC
  
  + The complete software system will be easier to implement.
  + The software can be made much more agile, with support for e.g. an arbitrary number of AGVs.
  + Data handling will be much easier.
  + Data can be accessed from a higher abstraction level, e.g. by administration programs.
  + The database can be accessed from the network, with possibility to use web-based services for control and supervision.

- Disadvantages using a PC
  
  – The stability can be a problem. The system must not fail at any time, and if it does it must be handled safely.
The advantages clearly exceed the disadvantages. Regarding the stability problem, there is much more in a PC that can go wrong compared to a PLC. In production industry they have been using PLCs to a great extent since the early 80s. But using PC:s in that environment is not very common. The problem is that a PC is much more complex, and higher complexity has an increasing potential of errors.

Is it possible to trust the system all the way from the hardware, to the virtual machine, to your own program? What happens if there is a disk crash? Such errors must be handled safely and fast by the system, as any downtime costs money.

2.3 Requirements of the New System

The main objective was to create a system which replaces the old PLC system. Due to this, all previous requirements on the PLC system are inherited by the PC system. Those requirements will not be described. However the main requirement of the inherited ones are handling the product flow inside the factory. So the PC system must be optimized at least enough for coping with that flow.

The new requirements of the system are:

- Being able to automatically adjust itself to different track designs.
- Handling an arbitrary number of AGVs and preventing them from colliding.
- Making it easy for the crew installing the system.
- The stability of the system is very important, especially in the economical aspect.
- In a future revision it might be possible for the AGVs to travel in both directions. Because of that the system should be planned intelligent, so minimal changes needs to be done.
- The solution must be implemented using C# .NET.

There are no other specific requirements on the software implementation itself. But as stated in Chapter 1; flexibility and efficiency must be considered a high priority at all times.
Chapter 3

Current Implementation

The goal was to create a system which was both flexible and efficient. If a complete system should be flexible in itself, so must the software solution. It must be considered in every choice the developer make. But of course some choices are more important than others.

One consideration was to give the AGVs more intelligence, e.g. giving them the ability to guide themselves through the track. This would naturally give the server less control, which is not very desirable. The reason for that is first of all the route-planning. An AGV would not as easy be able to find the optimal route on its own since knowledge of the complete track would be needed, plus knowing the positions of the other AGVs.

Secondly, collision avoidance would be hard to do within the AGVs and unplanned events could occur, which also is not very desirable. Doing this with a server solution would be more suitable.

Changing the software could also be a problem. If you e.g. have to update the system you have to do this on every AGV in whole the system. Updating using a server solution would in most cases only have to be done at one place, depending on what is updated of course. There must still be some software in the AGVs.

To gain efficiency the keyword is planning and that is the main reason to have central intelligence. It is simply not possible to plan a route without having enough information of the environment.

The solutions are divided into three separate programs. The first program developed was the track designing utility called "Path Constructor". Secondly a simulator was created to be able to see what is happening by monitoring the database, called "Path Viewer". Letting the viewer know if there has been any crash and where it has happened. The third and main program is the server software, called "Path Server". Around 80% of the development time has been spent on this since it is the main part as well as the most complex part of the project.
3.1 Data structures and Algorithms

The choice of data structures is very important because that is the foundation of the system. Much will depend on the data structures and using an unsuitable one will result in a more complex solution, lacking speed and having bad time complexity.

3.1.1 Graphs

Looking at the requirements, making an agile solution do have certain demands on the solution. At first it must be possible to dynamically decide which path to choose and in some way find the smartest. The solution was to use graphs. There are a lot of studies made about graphs and many proved algorithms for different tasks. For this particular project there are a number of reasons for using graphs:

- Nodes are used as either turn/crossing or storage. There are also nodes used as home nodes for the AGVs as mentioned before.
- Weighted edges are used as the lines between the storage nodes and the turns/crossings.
- Possibility to use a majority of algorithms e.g. Dijkstra’s shortest path algorithm.

It felt natural to use graphs in this project. As for representing the graph an adjacency list (Weiss [1], pp. 450) was used, due to the fact that the representation should be stored in a database. It is more suitable for database storing to use lists than matrices, as well as for saving to an ASCII file (which also is an option). The matrix would be so sparse anyway since every node only can have maximum four edges connected, being another reason not to use an adjacency matrix.

3.1.2 Dijkstra’s Algorithm

Dijkstra’s algorithm is the main algorithm used in this project for calculating the shortest path. In the server program it is used quite often both for allocating a path, as well as for finding the most suitable vehicle to do a certain order. There are many fields of application for this algorithm in these kind of programs. The graphs used are not that big and they are also pretty sparse, making the algorithm execute fast. So there is no problem considering its time complexity.

3.2 Specifying a Track

To be able to guide the AGVs, the system must know the complete track where they will be running. This could easily be done by specifying which nodes and edges do exist and how they are connected. Doing this in a simple text-file would do the job,
but with growing tracks this would not be the most efficient nor the most cognitive solution. As for more functionality (e.g. RFID-tag specifications), that file would be a mess quite soon. There will be a need for more information than was thought of at the beginning.

At this planning state every mistake will cost a lot of time. You cannot just rush into things and start creating a solution before giving it enough time thinking it through. Considering the requirements is one thing, but the implementation is also possible to do in many ways. There are certain things to consider.

- How much time will it take to implement?
- How often will it be used?
- How much time will the user save by using this implementation?
- Do the system have the need for continuous changes?

This is something to think about every time something new is created, as it will save time both developing and using, which results in saving money.

3.2.1 The Track-designing Utility

The solution was to create a graphical track-designing utility, which was supposed to facilitate not just the floor painting and RFID-tag placing, but also have the possibility to load the designed track into the server program. This utility contains more functionality with the ability to upload to any MySQL database. The system needs a lot of information in order to work desirably.

- The system must know the track.
- The system has to know which nodes that can be used as storage, as well as which storage row they are a member of.
- The rows must have unique identities.
- The edges between each turn/crossing or storage node, must be known.
- The length of the edges must also be known. That is required to be able to calculate the optimal route.
- The RFID-tag ids must be editable.
- The system need to know which AGVs that should be working in the system.
- The homenode of the different AGVs must also be specified, as well as which nodes the AGV is allowed to visit.
- The edge restrictions, e.g. if they are one way only.
3.2. SPECIFYING A TRACK

The following Sections (3.2.2 - 3.2.4) describes the graphical representations. These representations are also used in Path Viewer but not in Path Server, since it does not require any graphical components.

3.2.2 Representation of a Node

The nodes were in Path Constructor designed as new C# components extending UserControl, which is a template control in C# made for creating own controls. To implement nodes (see Figure 3.1) the first strategy was to make them round, as seen in the most drawings. But it became quite hard to make them look good because the area surrounding them was impossible to get transparent. It was possible to get them statically transparent, but if you should be able to move the nodes (which you should) then it became impossible to keep them transparent, especially when moving across other nodes (or other controls). The problem was simply solved by making the nodes squares instead of circles, which eliminated the need for any transparency. This is also suitable considered that the AGV only can do 90 degrees turns and there will thus only be a maximum of four edges connected a node anyway, which makes it one edge entry per side.

There are plenty of settings for the nodes (see Figure 3.3); e.g. name and node type. If the node is a home node or a start/end (e.g. a machine) node which only has one edge there is an exit option to choose from. Either the AGV exits the node running backwards to the target node, or it simply turns around 180 degrees and runs forward. This option exists because in some machines you have to exit running backwards. There might not be enough space for a complete turn around.

It is also possible to assign a storage node to a storage row. This node must also have a unique id inside the row, for the system to be able to order the row.

3.2.3 Representation of an Edge

The edges did seem easier, but there were some issues regarding them as well. Basically an edge is just a plain line from node A to B. But every edge have a length and can be assigned a name. This must be shown in some way near the edge’s line. The solution was to create one component which just like the node component were implemented extending UserControl. The upper text is the name of the edge and the lower one is the length of the edge, see the box on top of any edge’s line in Figure 3.1. There is also an arrow on some edges. This arrow describes if the edge is travelable in just one way and which direction that is allowed. So far the edges was quite straight forward. But there should also be a line describing where the edge starts and where it ends. That line would not be possible to create as a part of the earlier name and length box, considering the earlier problem using transparency (or opaque) in controls. As this line will follow a node when being moved, it would also have that same problem. The solution was to draw the lines inside the main panel of the program. To be able to do that a modified panel must be created, which extends a normal panel. In the new panel the OnPaint method
extended from the other panel must be replaced, or overridden as it is called in C#. This method is called every time something happens in that panel, e.g. when a control is moved and that do suit the purpose. Drawing in this method also solves the transparency problem. The disadvantage doing it this way is that the program takes quite some processing power when having a large amount of edges, and e.g. the user is moving a node control. Then that method is run all over again until the control is still, also creating a form of line flicker on the edges 'lines. However the control movement is not affected nor the behavior of the rest of the program.

An edge does not have as many settings as a node. It is mainly the length (or distance) of the edge that is interesting, but as said earlier an edge can also be assigned a name (see Figure 3.4). A direction restriction can also be set on each edge, letting through traffic in only one direction. The RFID-tag assignment seen in Figure 3.4 is discussed later in upcoming Section 3.2.4.

3.2.4 Node Points

The connections between the nodes and the edges are called node points, see the square dot connecting a node and edge in Figure 3.1. The colour of the node point is either black or blue, depending on whether it is connected or not. An edge is created by first clicking an unused node point (black) and after that clicking another. The entries going into or the exits going out from a node is defined as a node point. The number at the side of the occupied node points describes the RFID tag of that node point. Every node point must be assigned a unique tag number, as described in Section 2.1. To do this in Path Constructor double click either the node or the edge that is connected. When a double click event is triggered (for either a node or an edge), their properties windows are shown (see Figure 3.3 and 3.4).

3.2.5 Designing a Good Track

Considering the abstraction of the system (see Section 2.1), there are certain things to keep in mind when designing a track. Since all pallets must be entered from the same direction every time, the edges inside a row must be one way only. But if all rows are one way only in the same direction there must be a way for the AGVs to travel in the other direction. This must not necessary be one way only, but that is recommended, since an AGV traveling the other direction would break the item flow. It is however not dangerous but it will cost unnecessary time. That return way is called a highway. For a typical design see Figure 3.2.

This is the recommended design which is guaranteed to work with the Path Server. Many of these could be combined for meeting all the needs the customer has.
3.2. SPECIFYING A TRACK

**Figure 3.1.** A part of a screenshot taken in Path Constructor, showing some nodes and edges in an incomplete track (graph).

**Figure 3.2.** A typical storage design where (not seen in figure) all storage edges are one way right and the mid highway is one way left. The vehicles standing at their home nodes can be seen to the right. Note that this system is not complete since there is no entry nor exit to any machine or other buffer system.
CHAPTER 3. CURRENT IMPLEMENTATION

Figure 3.3. A screenshot of the node properties dialog that could be seen in Path Constructor.

Figure 3.4. A screenshot of the edge properties dialog that could be seen in Path Constructor.
3.3 Simulating the System

Having the possibility to simulate a solution is a great advantage. There are multiple reasons of simulating before making the system run live:

- Fast error feedback.
- Not as risky as running directly on hardware, if looking at the safety aspect.
- The sources of the errors are in a smaller area, making them easier to find.
- Logical errors are easier to find.
- Possibility of finding unusual errors (e.g. deadlocks) by setting the system in unusual states.

But of course not all errors can be detected using simulation. However most of the logical parts can be tested, verified and made ready for live testing.

3.3.1 The Simulation Utility

The simulation utility is called Path Viewer (the second program created) is able to connect to a database created by Path Constructor. That database contains a lot of information, e.g. the complete track design, AGV information (including current position etc.) and also information about the storage (including all items currently in the system and where these items are placed). The simulator can get all information it needs from the database in order to make a reliable simulation.

In Path Viewer the database is constantly watched. The program shows the whole track as drawn in Path Constructor, and also all AGVs in the system. If an AGV move is registered in the database, the AGV also moves in the simulation program. Even all storage items (including item type) are shown on its current storage node, and if the item is lifted by an AGV, that item is shown on the back of the AGV. Complete orders could be watched in this program, which is very useful for verifying that the system behaves desirable. If a crash occurs (meaning two AGVs is concurrently on the same node or edge), the program will abort and show an error message. The program will stop reading the database, leaving everything as it was when the crash occurred. That feature is useful in marathon testing, when the system is left unsupervised for a long time. If an error do occur during a test, it is possible to see in what state the system was.

The most important logical issues to test is collision avoidance, and the possibility for deadlock. Different states of the system could be tested to try triggering these events. You can through the database place the AGVs in such an order to simulate that a deadlock have occurred and see how the system reacts.

This has been an important tool during the development, facilitating debugging a lot.
3.4 The Server

The third program called Path Server is the main program of the project. It is also the most advanced program, demanding most time developing. It is the controller software that sends commands to the AGVs. It constantly monitors the database, just like the simulator. Gathering all information it needs from the database, still having its own internal datastructures. The reason for that is to speed up execution, using the database only when needed.

3.4.1 Definitions

When a row needs packing it means that there are spots free, either in the front of the row or between items in the row. The system basically wants to have all items in the front spots at all times, making it possible to input more pallets at the free spots. If that is not the case a packing order will be created, moving the items forth the row. See Section 3.4.6 for more about packing.

An allocation is basically a list of nodes where a vehicle is scheduled to run. It could be compared to a priority list, because the vehicles only yields the right to the allocations before their own allocation in the list. First in the list has the highest priority. This is explained further in Section 3.4.4.

A half cycle order is an order running only half the order cycle, e.g. a 'move item'-order is divided into two half cycles orders, where the first order collects the item and the second leaves the item. All orders in the database are written as full cycle order and divided later on in the server program.

3.4.2 Description

As this program is quite big only a general description will be given.

The main method running is the server method. It contains the read functions from the database and it also owns a list of vehicles, which are used in the system. There is one method (periodically called) for synchronizing with the database. The period is at the moment set to two seconds, which means that it will take two seconds at worst for e.g. an order to reach the server. In this application that is an acceptable delay. In the safety aspect there might be a complain that two seconds is too much, e.g. if you want to abort the current order. But it is considered no problem since there are multiple emergency stops on the AGV, which can be used if you want to stop it really fast.

When the periodic method is triggered the server does the following:

1. Updates the settings from the database.
2. Updates the items in the storage from the database.
3. Reads the orders from the database.

4. If packing is enabled, the system checks if any packing is needed and creates new packing orders.

5. Executes the orders for each vehicle.

Parts of this will be event driven when the system is fully implemented, e.g. the communication with the AGVs, decreasing the use of this periodic event, with the database communication as the only exception. It is e.g. not possible to know there is a new order in the database without looking for it at a certain interval. In this program there is no graphical components whatsoever. The user interface simply consists of a log, a connect/disconnect button and start/stop buttons, see Figure 3.5.

### 3.4.3 The Use of Graphs

There is one main graph class having all nodes in a list. It also contains a list representing all edges to facilitate edge iterating. Every node has a list of edges going in, just as any normal graph structure. The graph class contains a shortest path function, to be used either between graph objects or between graph ids. The graph class also contains a list representing all items, which either can be assigned a storage node or a vehicle (whether the item is being moved or not). The reason the item list is put in the graph class is that many methods (inside the graph class)
have the need for reading/editing the items. In that way we avoid having a lot of parameters sent back and forth. It also contains a list of all allocations in the system, see Section 3.4.4. That is for the same reason as with the items; many methods needs access to the allocations.

3.4.4 Allocations

One problem that had to be solved was which vehicle had the priority to run, if e.g. multiple vehicles had to run through the same crossing (at the same time). Such occasions will often occur and must be safely resolved. The solution was called allocations.

An allocation is basically a list containing an arbitrary number of nodes where a vehicle plans to travel. A class Allocation was implemented containing that node list as well as the vehicle owning that allocation. The active allocations are stored in a list where only one allocation is allowed per vehicle. The first allocation in the list has a higher priority compared to the others. If e.g. two vehicles simultaneously wants to enter a crossing, the first one in the allocation list will first be allowed to enter. The time the order was issued gives the allocation its priority.

This is a place for further efficiency improvements. Consider an order traveling through many nodes has been issued. Just after that order was created another order is created, sharing some nodes with the previous. Although this order is not that big (not visiting as many nodes) and is just sharing the last nodes with the first order. It still has to wait at some place, until that previous allocation is not allocating the shared nodes anymore. That results in that the second AGV must wait unnecessarily long, when it would have had time to run before the first one. That is all because the first order was issued first, giving the second one lesser priority. For security reasons it is left as it is, due to collision avoidance and other unplanned events that could occur.

3.4.5 Order Handling

The orders are written to the database by some client software, see Section 3.5. There are two different order types in the database. The first type is a simple move order which moves a vehicle to a specific node. The second order type is "move item", which basically assigns a vehicle to move an item from one node to another. When the order is created it also gets a time stamp. It is possible to assign a specific vehicle to take the order. If no vehicle has been stated, the most suitable will take the order.

In the server the orders are divided in so called half cycle orders, see Section 3.4.1. The full cycle order read from the database gets split up as soon as they are read. The cause of having half cycle orders are mainly because of the allocation strategy. If full cycle orders would have been used the whole traveling route would be allocated, preventing the other vehicles from crossing that route. That may result in unnecessary long wait times. But by using half cycle orders, other vehicles'
3.4. THE SERVER

Priorities would become higher when the first vehicle completes its first half cycle order, which will speed up the flow.

Within the program there is one special order that will be executed without being read from the database. That order is called the "run home"-order. Every vehicle has a list of orders to execute. When this list is empty, and the vehicle is standing somewhere out on the track, a "run home"-order will automatically be created. Since it is not in the database, it will not beabortable through the client software. But it do have another feature: If this order runs and another order is created (for this vehicle), the "run home"-order will be interrupted. This order does not have to be completed, which saves some time. Hence, the vehicle does not have to run home before the next order is started.

Having such a feature can however create some critical situations. One critical situation could be if a vehicle is executing a "run home"-order and another vehicle (having lower priority) is running behind it. Suddenly a new order for the first vehicle is issued, interrupting the "run home"-order. Now a new allocation is created crossing the second vehicle’s allocation. The system will now enter a deadlock state where both of these vehicles are blocked by the other.

A solution to this is to not interrupted the order if there is another allocation on the vehicle’s current node, continuing the "run home"-order until no other allocation is at its current node, and aborting if that is the case. The worst case would be that the vehicle must run the whole way home (since no other vehicle would be allowed to enter another vehicle’s home node), which might happen if there are many vehicles in the system. This solution has been implemented.

3.4.6 Packing

Since the vehicles only travels well in the forward direction, all rows must be ordered in FIFO (First In First Out). Consider the case that one row is full, meaning every spot in the row occupied. If an AGV removes the first item in the row, there would be one spot free. But that free spot is not possible to use since the rows are one way only and that spot is in the wrong end of the row. That free spot must be at the entry of the row to be usable. Every item must be moved one spot forward, and doing this becomes very time costly as it is many item that has to be moved. That is the major disadvantage of the system, but it is a consequence of both of having the FIFO order, as well as the AGVs only running forward.

The system automatically creates these packing orders when needed. There are however a few settings the client can change. At first there is a possibility to disable packing completely as well as re-enabling it. There is also an option about when the system starts giving packing orders. Seen from the end of the row, you can set the minimum free nodes before the system starts packing, e.g if that number is set to two and there is only one free node in the back of the row the system will start packing until there are two free nodes. It would however still be possible to create a normal client order at that row.
Note that the packing orders are (just like normal orders) created in the database, and not just internal in the server. This gives the client the ability to abort a packing order as well. Note that if a packing order is aborted there will be a new one created directly, supposing packing is enabled of course. So to abort a packing order completely, packing must be disabled before the abortion is issued. This is an issue for the client software. Maybe packing could automatically be disabled when an abort on a packing order is issued.

### 3.4.7 Collision Avoidance

One important issue is the stability of the system, keeping the system flowing and not having any breakdowns that has to be fixed manually. As stated earlier, every breakdown costs money. Collision avoidance is one big part of the stability. There could be severe consequences for a collision, especially in the safety aspect.

Having more than one AGV will require collision avoidance in some form. In this application most collisions are avoided using the allocation system. As the AGVs are blocked they wont cross another AGV’s allocation, preventing the collisions in most cases. But consider the case when an AGV has completed its last order. The new order, which is a "run home"-order, is not yet created and on the node behind it comes another AGV in full speed. Since the first AGV does not have an order it wont have an allocation either, and having no allocation results in being invisible on the track. While the first AGV is invisible the second one wont be able to see it, which will create a critical situation. The onboard safety system would have taken care of this situation anyways, but in e.g. a simulation the AGVs would have crashed.

A solution to this is to create another condition that must be fulfilled for an AGV to get permission to run. It is not allowed to run, if there is another AGV standing on its target node. Note that an AGV owns a node until it has acquired another one, even while it has left the node (entered an edge). But adding more conditions brings new complications, since new deadlock situations can be created. An AGV can now be blocked in two ways; firstly by an allocation with higher priority, and secondly by another AGV standing on the target node. For more about deadlock see Chapter 4.
3.5 The Client

To be able to create orders and change settings there must be a client interface as well. At first there are some options for the items, including types and positions etc. Secondly is order handling, the most basic feature to create orders for the system. There are also some settings for the server as well as for packing. Only a demo has been created to test the system, see Figure 3.6. The future production version will be more user friendly and have more functionality. See Chapter 5 for more about the future implementations.

3.5.1 Item Handling

To insert a new item into the system, the item type must first be created. There can exist multiple items of the same item type. A company usually produces an arbitrary number of product types and that is the main reason for having different item types. So when the system first is installed, the expected item types should be added. Later when the different items are produced, and inserted into the system, the item type must be specified for each new item. It is of course possible to add more item types later on, as part of the flexibility vision.

3.5.2 Order Handling

Orders are (as described in Section 3.4.5) created in the database, see the mid part of Figure 3.6. There are two different order types to create; move vehicle and move item. Note that when creating a "move item"-order, the option "Most suited vehicle" can be selected. This allows the system to calculate which vehicle is most appropriate for this specific task. This calculation is done by the server and takes the distances to the targets nodes into consideration, including the distances of the other orders the vehicles currently has. It is highly recommended to use this feature and in a future production client that might be set as the only option available.

There is also an option for removing existing orders. Every order in the database can be removed, including active "move item"-orders, even when a vehicle already has picked up an item. That must however be used with caution, since the vehicle will keep this item and continue to the next order with it. In a production version a warning may be issued if the vehicle is carrying an item, or it should not be possible to abort an order at that stage.

3.5.3 Settings

At first there is an option to reset the server, see the bottom left part in Figure 3.6. It will be equivalent to stop the service, disconnect, reconnect and start the service again. A complete restart of the server must be done manually.

As for the packing settings, first there is a button to enable and disable packing completely. There is also an option to set how many free spots (counted from the end) there must be at minimum at a row before packing should start. The reason
for those options is to gain more control over the packing, and be able to make a perfect setup for the customer. There is also an option to always pack which should be preferred.

Figure 3.6. A screenshot from Path Demo.
3.6 Conclusions

The current implementation is not a complete system, but it do fit its purpose. The goal was to show that a system like this could work, and that the different logical issues are solved. That has been accomplished in accordance to all abstractions described in Section 2.1. There is however more work to be done before the system is complete, see Chapter 5.
Chapter 4

Deadlocks

A deadlock is a phenomena which not only can occur in computer systems, but in every automated system. The deadlock issue is a well studied problem that is not that hard to solve. The most important is to know that deadlocks exists and the most suitable ways to handle them. Many studies have been made on deadlocks in a variety of subject areas, as this issue has been known for quite some time. In the 60s the issue was first detected, as multitasking in the operating systems started developing. Early problem statements and solutions was suggested by Havander [2] and Coffman, Elphick and Shoshani [3].

The main issue is resource sharing. Having a flexible and efficient system requires resource sharing between the units. In every system where resources are shared, the deadlock issue must be investigated. Every resource can be allocated by a task, which is given to a unit. As for this project the vehicles are the units and e.g. the track (both nodes and edges) as well as the items are shared resources. The tasks are the orders the units are executing. It is however not guaranteed they will occur, since that is the difficulty with these issues; there might only be a one in a thousand risk they occurs. The rarest issues are the hardest to solve, because you cannot simply know that the solution works, unless there is a proof of it.

4.1 Deadlock Conditions

It is however possible to know whether the system has a risk for deadlock or not. Stated by Coffman, Elphick and Shoshani [3], there are four conditions that must be true to enable occurrence of deadlocks:

1. Mutual exclusion of the resources (mutual exclusion).

2. The units holds the current resource when requesting a new (hold and wait).

3. An unit holding a resource cannot be preempted by another unit (no preemption).
4.2 DIFFERENT STRATEGIES

4. A circular chain can be formed by two or more units, where a unit requests the next units resource (circular wait).

At this stage it might seem fairly easy to find a deadlock issue. But consider more advanced examples with more shared resources that the units can allocate. Forcing all conditions to be checked for each resource and every unit that can allocate this resource. The resource combinations the units can adopt must be properly checked to ensure that it is not creating any new problems. Using different resource the units can answer different to these conditions, which also has to accounted for.

4.2 Different Strategies

There are multiple approaches to solve the deadlock issues which can be divided into three categories (Coffman, Elphick and Shoshani [3]):

- Prevent the deadlocks from occurring.
- Avoid deadlock situations.
- Detect and recover the deadlock situations.

All strategies do have advantages and disadvantages. Below an analysis has been made of these strategies with this project’s issues as the main concerns.

4.3 Prevent-Strategy

As stated in Section 4.1 there are four conditions for a deadlock to occur. So making sure at least one of these conditions fails ensures there is no risk for deadlocks. This is the prevention strategy which according to Coffman, Elphick and Shoshani [3] is the preferable strategy. However it is possible to switch the condition that should fail, meaning that it must not be the same condition that fails at each specific time, only that at least one fails. This might be possible to use for some systems, but not for all. It is e.g. not as easy to fail the mutual exclusion condition for a practical system as it might be in a theoretic computer system. The same goes to the "hold and wait"- as well as the "no preemption"-condition. The last condition "circular wait" is the only condition that is fairly easy to fail for a practical system.

It is however a good start by investigating these conditions for one of the most obvious resources; the nodes. The vehicles will serve as the units, and the orders as the tasks. It also represent the current build very well, resulting in a fairly accurate study. Notice that there are more resources that could be investigated.
4.3.1 Mutual Exclusion

Consider the first condition; mutual exclusion. Two or more vehicles cannot share a node. If considering a pallet on that same node makes it even more obvious. It is also practically impossible for a vehicle just to disappear while standing on a node. That node cannot be reached by other vehicles at the concurrent time otherwise a crash will occur which not is very desirable. The "mutual exclusion"-condition will always be true for a system like this.

4.3.2 Hold and Wait

As for the second condition, the same applies. While a vehicle has a resource (node) and waiting for another resource the vehicle currently owns that resource and will not give it up until it has gotten the next resource (see Figure 4.1). So currently this condition is also true. There is however a possibility to avoid this condition. Consider when a vehicle is waiting for permission to enter a node, that vehicle could possibly exit its current node and enter the edge and stand there waiting instead until the next node is free. That will result in freeing the current node concurrently, while requesting the next node (see Figure 4.2) and by that the "hold and wait"-condition is false. Note the differences between the Figures 4.1 and 4.2. There could however be other issues using this solution, e.g. where on the edge the vehicle should stop before it gets permission to enter the next node. Just exiting the node by two vehicle lengths might be enough. Using this solution also makes the edges another allocatable resource, which must be analyzed in the same way as the nodes.
4.3. PREVENT-STRATEGY

Figure 4.3. A vehicle (the lower) standing outside the track, a fictional result of a preemption by another vehicle (the upper) having higher priority.

4.3.3 No Preemption

The third condition is pretty obvious in this case, since it is impossible to replace a vehicle at a specific node by force. Although a solution moving a vehicle off the track could be used (see Figure 4.3), but that would not be very practically. It would create new complications, e.g. guarantees there are enough space for a vehicle to exit around the whole track. This is not very desirable since there must be enough free space in e.g. narrow passes, for at least two vehicles in width. The valuable space between the storage rows must also be extended, resulting in lesser rows in width, which means lesser spots to use as storage. So this solution is not an option and the "no preemption"-condition will still be true.

4.3.4 Circular Wait

The forth and last condition is hard to completely prevent. Looking at the nature of the vehicle movements, when the vehicle e.g. is packing, it first must pick up and leave the first item, then exit the row and completely go around the row to reenter it (see Figure 4.4). That flow is circular and vehicles will run this route quite often. But it is not only the packing that is the problem. Say a vehicle just has finished a "move item"-order and gets a new order to pick up another item, it must not be the same target row but it might be a neighboring row. That will also result in a circular behavior. These abilities are required and thereby it wont be possible to fail the "circular wait"-condition for all system designs.
4.3.5 Example Implementation of the Prevent-Strategy

The current implementation is using the allocation technique (described in Section 3.4.4), which gives the system certain properties. A vehicle is e.g. not allowed to cross another vehicle’s path if that order was received earlier than the other. This results in a predictable behavior, which makes it easier to find potential deadlock situation. As seen in Figure 4.4 and having the allocation behavior in mind, there is a possibility of deadlocks in a situation like this, but there must be at least two more vehicles in that system to make it possible. By not having that amount of vehicles will prevent a deadlock situation, while the allocation technique is applied. It would also not be very effective having that many vehicles in the same small system. The track must be inspected for these loops, where the circular wait condition is jeopardized. A good rule is to only allow as many vehicles as the least number of storage node members of a unique row, since that will automatically eliminate the risk for circular wait.

4.3.6 Summary

The only acceptable solution among the proposed non-implemented solutions is the one stated in Section 4.3.2, which solves the second condition (hold and wait). The others are simply not good enough. They are either too hard to implement, or inefficient in other aspects. There are also other approaches for the prevention strategy, like graph or petri net solutions, as e.g. shown by Viswanadham, Narahari and Johnson [4]. In this project such a solution would only complicate matters more.

4.4 Avoid-Strategy

The avoid strategy is the second ranked strategy. It is not the strategy that is currently used in this project. Just like the prevent-strategy the goal is to not have any deadlocks happening at all. It could however theoretically be possible for a
deadlock to occur using this strategy, but that is avoided in every critical situation. That is the difference between this strategy and the prevent-strategy: All conditions (stated at the beginning of Section 4) will be true but the deadlocks are still being avoided. There is always a risk using a strategy like this, although deadlocks should be avoided it is not easy to guarantee that they are. Many stated solutions, e.g. by Zhang [7], tries to prevent upcoming critical situations. But the solution must be able to solve even immediate issues as well as upcoming. Having this wide time spectra creates even more difficulties.

4.5 Detect and Recover-Strategy

This strategy should only be used if no other strategy is implementable. At first there is more work to be done, including creating two algorithms; one for detecting and another for recovering. Secondly a lot of testing must be done verifying both algorithms. But it might in some cases be the only possible strategy.

4.5.1 Detecting a Deadlock

This strategy will not be used for this particular project, but multiple solutions are possible. The first and preferred solution is to inspect every vehicle for any blocking. If all active vehicles are blocked, there must be some kind of deadlock. This detection must be run every time a vehicle block is detected. If a deadlock is detected, proceed to the recovery routine. Another way of detecting is to after any event check the track for any inconsistencies, where a deadlock could appear. This is however not recommended since that algorithm would be harder to implement, as well as less efficient compared to the method stated earlier. However, detecting a deadlock is much easier than recovering from a deadlock situation.

4.5.2 Recovering from a Deadlock

If a deadlock has been detected, what is at that stage the best solution to use? One solution would be to abort the order for the vehicle having a non-physically blocked way to its home node, not taking any allocation in account. This way, there must be at least one vehicle having this prerequisite. But that creates a new problem: What will happen to the order that was aborted? It is better to just create a "run home"-order and advance this order in the order priority list? That would result in the vehicle running home, and after that continuing its aborted order, but now standing at its home node. The reason for sending the vehicle home is that the home node is the only node that only is accessible by one unique vehicle, running the whole distance home might not even be needed. This vehicle is now not a part of the deadlock, but however it is not certain this has solved the complete deadlock. If there still is a deadlock existing, the recover routine must be rerun. Sending another vehicle home and try detecting any continuous deadlocks. If no other deadlocks are detected, the normal execution will run again, completing all previously given
orders. Even if the vehicle is in the middle of a "move item"-order (having a pallet on its back heading for the target node) that neither will be a problem. Since the orders are half cycle orders, the first half (collect) is already completed which means there is only the second half (leave) left, which is the required behavior. This might however not be the most efficient way of recovering from a deadlock. But it can however guarantee to solve the deadlocks, as well as being fairly easy to implement.

4.5.3 Summary

There are many proposed solutions for detecting and recovering from deadlocks, including Cho, Kumaran and Wysk [5] and Wu and Zhou [6]. The solution stated here is not general, but it is a good specific solution for this particular system. It is however not currently used but it may in future revisions. As detecting and recovering will be complex to implement and requires a lot more computing power compared to the other strategies, that is the reason this strategy is not recommended, if there are other options.

4.6 Deadlock Experiences

One of the reasons that the simulation program was created was to test the strategies both for collision avoidance, as well as deadlock situations. This is a great advantage for testing different deadlock strategies, since these situations could be created by statically, configuring the vehicles positions, their current orders and so on. It is of course very hard to find every critical situation, but at least there is a way to test them.

Once when the system only used the allocation technique, a deadlock did occur. One vehicle was ordered to move an item to row A. Another vehicle had a packing order issued to that same row, but some storage spots further ahead, while running after the first vehicle (see the situation in Figure 4.5). When the first vehicle got to its target node and put down the item, the current order was completed. A new order was created directly, ordering that vehicle to run home. But that new order made the first vehicle’s allocation have lesser priority compared to the other vehicle, but it was still in front it, which resulted in the two of them locking each other. A deadlock was created since neither of them could reach their next resource. But there are however ways to avoid this situation.

At first, the only time this situation can occur is when both a conventional and a packing order is issued on the same row. It can only occur inside a row since that is where every first half cycle order is finished, and the second half cycle order gets a new allocation (with the lowest priority). That is the main problem, the second half cycle order gets lower priority lower compared to the succeeding vehicle. But as it only can happen inside a row, and the rows are one way only, those allocation method has no use. A simple solution is to ignore all allocations, while running for a storage node. This will however not be possible for a row that is travelable in
4.7. CONCLUSIONS

Figure 4.5. A preventable deadlock situation. The right vehicle is about to start its second half cycle order (leave "Item 2" at spot A:1) of the packing order as the left vehicle tries to reach "Item 1".

both directions, but since that is one of the system’s abstractions (see Section 2.1), that method can be used.

After analyzing the system, using the track structure as stated in Section 3.2.5, as well as the described methods there should be no deadlock issues.

4.7 Conclusions

Deadlocks are an issue that cannot be ignored, the consequences could be too big and not worth risking. It do however require more development time, but just one occurring deadlock in a production version would cost more. The solutions however are fairly simple, which is another reason not to ignore. It might however not be so easy to find every resource, task and unit, that possibly can deadlock. That is the biggest challenge.
Chapter 5

Future Work

There is a high probability that the system will go live. After having a conversation with the supervisor as well as the customer an agreement has been made to start up the system in the late first quarter of 2009. At the moment it is possible to simulate the customer’s storage, confirming that it works as they expect. The next step is the communication with the vehicles. The communication will use a wireless system developed by Texo Application. As there will be PLCs in the vehicles which another developer will program an agreement must be made regarding the communication. The client that the operators should use must also be developed.

5.1 Communication with the AGVs

The main issue making the system run live is the communication part. There are multiple issues that has to be further investigated, e.g. when the commands should be sent to the vehicles or how many commands every vehicle is allowed to buffer. Letting the vehicles only have one command at the time might slow down the execution, since only when the current command is completed, the next one can be issued which forces the vehicle to just stand and wait. A solution to this might be to allow the vehicle to store two commands, the current and the next. This may however not be safe, since the server loses direct control of the system but it may be a good compromise between direct control and execution speed. This must be investigated more practically before the system goes live.

How and when to send different status packages is another issue. The battery level is the most critical status information that needs to be sent to the server. When is the best time to do this? It must anyhow be transmitted often enough. It would be possible to include battery level information into every packet the vehicle sends to the server. In that way it is possible to have control all the time on this issue. However, the vehicle is not always transmitting packets, but the only time not transmitting is while the vehicle is on its home node, and standing on the home node means that the vehicle is docked to the charger. This might not be the most flexible solution since not every system might have a charger dock at the home node.
5.2 The Client

The client part has to be created almost from scratch, requiring quite some development time. Some parts can however be taken from the demo program (described in Section 3.5), e.g. many SQL calls. The solution is meant to be web-based, which has a lot of advantages:

- Cheap clients can be used, e.g. small industry PC:s.
- Almost no software installation required on the clients.
- The system can be reached from all computers allowed, e.g. by the supervisors at the office. It can be even be reached over the internet, although not recommended due to the safety issue, but still possible.

Other useful features would e.g. be to be able to see the complete track, items at their specific spots in the storage, as well as the AGVs moving. This way it is possible to view the system just as the system sees it, and any inconsistencies to the real world can be adjusted.

There must of course also be a part for order handling, preferably more specialized for the customer’s needs. A reoccurring "move item"-order, e.g. from a machine, can be made so the operator does not have to select the collect node every time, only the destination node. But the possibility of moving from one specific node to another, must still exist. Regarding the remove order functionality, some features must be added. While e.g. running the second half cycle order of a "move item"-order (as stated in 3.5.2), there must either be a message box popping up, asking if the operator really want to abort this order, or simply not allow abortion of an order in that state. There are advantages and disadvantages with both solutions, however the safest is to not allow the abortion.

There must also be a password protected service part, where certain special settings can be changed, e.g. direct database changes that must be done with caution.

5.3 Visions

There will with no doubt be systems like this is the future. The level of automation is constantly increasing, and it will much likely continue like this. The demands on the systems will increase (as well as the systems themselves), and get more and more advanced. Larger supervision systems will be needed, and we will probably see more computers (not only PLCs) in the industry in the future. Especially, the old ladder program language used in PLCs will probably be replaced by some text-based language, since larger systems are not suitable for ladder programming. Software engineers will surely be needed in the production industry as well.
If looking at this specific system, there are major plans for further development. At first discussions regarding construction of a new version of the AGV is ongoing, having more features like e.g. running in both directions, which will speed up certain orders a lot. That will unfortunately also require changes in the software, but one of the objectives has been to keep it flexible, so hopefully these changes would not be that major. Inside the rows the vehicles must be able to travel both directions, but while carrying a pallet there must be no other pallet standing in its way. In the current system that never is a problem, and thereby that condition is never checked.

The software system as whole will be connected to different other systems including supervising administrative programs. It may also be ported into a 3D solution, stacking the storages in multiple levels, having elevators and different types of AGVs for different tasks. This results in high demands on the software, especially on the stability of all these features. That will become the main concerns.
Chapter 6

Conclusions

The main objective was to investigate if it was possible to create a both flexible and efficient system. The result must be considered a success, since the system actually was successfully simulated, as well as living up to the expectations. It does however, still remain some work to get a live version to work, but most of the theoretic parts are done. There are however more room for optimizations, which maybe must be done in a future version. This is especially true in the path allocation part, where currently there are some unnecessary waiting times. But as for now, the efficiency of the system is more than enough.

The plan was to make the AGVs change path if they had to cross another AGV's path, having higher priority. That was changed quite fast due to the fact that such a solution would become much more complicated. Another reason for not having it that way was the small amount of different paths an AGV could choose. They simply had to choose the same path in most cases, making the path changing ability not very useful. Currently the AGVs are assigned an order and allocates a path, if another vehicle (having higher priority) is running in that path, the others simply has to wait. It would also create new deadlock situations, which not is very desirable.

At first the collision avoidance was considered the hardest part, since the AGVs never should be able to collide. That was later discovered to only be a minor problem compared to other potential issues, e.g. deadlocks. But in a more optimized solution the collision avoidance would once again be a problematic issue, almost like the deadlocks.

The deadlock problems are really special. Since they rarely occur they are often bypassed in most automation systems. But in a system like this, such issues must be investigated, as a deadlock can be very expensive. The biggest problem is however not to solve the deadlock situations, but finding all potential situations. It cannot just only occur at the most obvious places. Thereby every shared resource must be investigated.
Bibliography


