AALBORG UNIVERSITY
STUDENT REPORT

JAWA
Just Another Warehouse Agent

Aalborg University
Department of Computer Science
5th term
Embedded Systems
sw502e14

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://cs.aau.dk

Title:
Just Another Warehouse Agent

Theme:
Embedded Systems

Project period:
Autumn semester 2014

Project group:
sw502e14

Participants:
Christian Friis Lyngbo Andersen
Kasper Langeland Michelsen
Mathias Johns Toustrup
Mike Gersvang Eiersholt Larsen
Rasmus Hove Johnsen
Thomas Skovsborg Jørgensen

Supervisor:
Brian Nielsen

Copies: 8

Pages: 75

Date of completion:
19. December 2014

Abstract:

Robots are a common part of many industries. Robots replace work tasks from humans that are mundane, dangerous, or can otherwise be automated. This project proposes an autonomous robot, JAWA, which allows for transporting and interacting with pallets in a warehouse environment. The robot will use a map to traverse the warehouse and an algorithm to determine the best path possible relevant to a warehouse scenario. It will then transport pallets to designated areas, based on the colour of said pallet. The warehouse may have unforeseen obstacles, misplaced pallets, employees etc. that the robot needs to actively avoid.
The robot will be made using the LEGO® Mindstorms NXT box-kit, and programmed to the leJOS firmware, version 0.9.1, using Java.

Christian Friis Lyngbo Andersen

Kasper Langeland Michelsen

Mathias Johns Toustrup

Mike Gersvang Eiersholt Larsen

Rasmus Hove Johnsen

Thomas Skovsborg Jørgensen

Preface

Six software engineering students wrote this report during their semester project for the 5th semester at Aalborg University. The title of this semester project was *Embedded Systems*.

In order for the reader to benefit from reading this report, it is advantageous for the reader to have prior knowledge of methods and terminology of the core subjects of software development, including machine intelligence, embedded systems, algorithms and real time systems.

Reading Guide

The report is written in sequentiel order and should be read as such. All figures in the report are made by the authors, unless stated otherwise.

Citation Style

All references throughout the report are in Chicago style, more specifically the author-year style. Every source is in lexicographical order sorted by first authors surname or, in the case of an organisation, the company name. The in-line references are written as the authors' surnames and year of publication, in square brackets. The bibliography contains all references and can be found in the end of this report.

Source Code

Source code is written in code listings. Long lines are broken and indicated with "↪".

The authors have used Eclipse Luna as IDE for developing this project, and as such this project is easily imported into Eclipse. The firmware used was version 0.9.1. The program has been tested on both Microsoft Windows and on the target hardware.

Disc

The CD will contain all files used for the creation of JAWA. The contents of the CD may need to be copied to a local directory, due to the CD being read-only.

JAWA is an accronym that stands for *Just Another Warehouse Agent*

Contents

1 Introduction

Embedded systems have become a big part of software development. They can be found everywhere from consumer, to industrial, to commercial use. Embedded systems are usually based on small units that are limited in memory and have computing constraints. They can sometimes be part of a bigger system and are used to handle a particular task [Barr & Massa, 2007].

Robots are common in industrial use to perform tasks that are mundane, dangerous, or can otherwise be automated. Robots also reduce costs of employees and increase efficiency. Robots that can perform tasks without human intervention are called autonomous robots. Autonomous robots depend on some way of observing its environment, either by doing this on its own or having a map of the environment [Yanco & Drury, 2004].

Autonomous robots can for example be found in storage departments and warehouses. Fully automated machines can be tasked with picking up and moving pallets to delivery or conveyor belts. These machines are also autonomous and therefore need a way to observe their environment to avoid collisions. They also need a way to know which pallet to pick up next by either hard coded destinations or scanning barcodes on the pallets [Egemin Automation, 2014].

This report will focus on the development of an autonomous robot. The robot will be using the LEGO® NXT hardware that allows for manipulation on both the hardware and software of the robot. The objective for the robot, like the autonomous robot found in the warehouse, is to move pallets and avoid collision with obstacles in its environment. The pallets will be scanned by the robot and moved to a specified destination. This task requires the robot to have sensors for both scanning its environment and a technique for identifying the pallets. Because the robot is autonomous, this will have to be done without human interaction.

1.1 Case Environment

In order to find the requirements for the robot. The context it should work in must be analysed. By doing this, it is possible to determine some of the requirements and tasks the robot should perform. A rich picture has been made to represent where in a warehouse process the robot could be implemented see Figure 1.1.

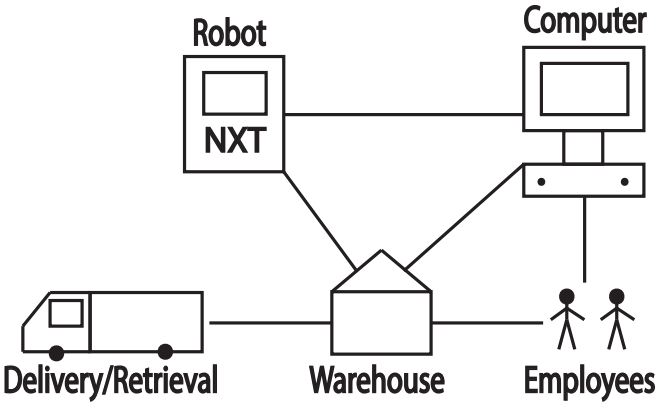


Figure 1.1: A rich picture of the robots environment

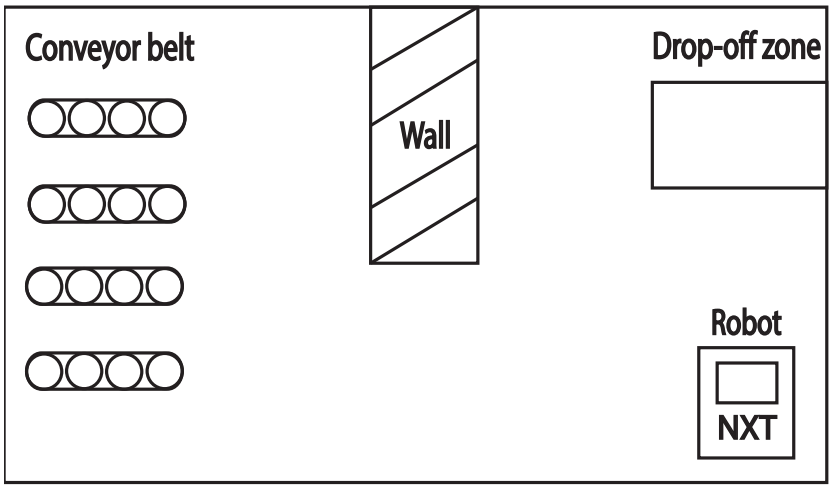


Figure 1.2: A model of the warehouse environment

A delivery truck carries pallets with goods that are transported to the warehouse. The pallets are placed in a specific drop-off point in the warehouse for the robot to move to its designated location. The employees that work at the warehouse do not interact with the robot but are still in the proximity of the robot. The employees keep track of what goods come in to the warehouse using a computer. The computer will update the robot with what goods need to be moved in the warehouse. The computer sends a signal to the robot when a pallet has entered the warehouse. The robot moves delivered goods around the warehouse for either storage or retrieval.

Figure 1.2 shows an example of an environment the robot should be able to operate in. The robot starts at a predefined position at each start-up. If a pallet is delivered to the warehouse, the robot will navigate to the drop-off zone. The pallet will then be scanned by the robot for the appropriate delivery at the different conveyor belts. The robot then needs to navigate from the drop-off zone to the conveyor belts whilst avoiding obstacles such as walls and employees in the warehouse. When at the correct conveyor belt the pallet will be placed and the robot will return to the drop-off zone to check for any remaining pallets that need to be moved. The robot will then continue this process until there are no more pallets and will then return to its start position.

1.2 Preliminary Problem Statement

Based on the motivation for building and programming such a robot, the following problem statement has been made:

The process of moving pallets in a warehouse today is largely done by people. This is, however, relatively mundane and can be automated by an embedded system, to save resources. This system needs to be able to perform the same core tasks as a pallet lifter in the same situation. This includes navigating the warehouse, identifying the pallets, moving them and avoiding obstacles.

We will investigate how to make an autonomous robot that has such features and which limitations our hardware and software choices have.

2 | Analysis

For JAWA to be able to achieve the preliminary problem statement, knowledge of the LEGO® NXT hardware is needed. To gain this knowledge, an overview of the hardware and the software API will be examined. Having this knowledge is important before the initial design of JAWA begins. Knowing how the sensor and actuators work is crucial to make sure that the right design decisions are made. The various sensors and actuators available for LEGO® NXT will be tested, to determine how they perform.

2.1 LEGO® NXT

The LEGO® NXT is a part of the LEGO® Mindstorm series. LEGO® NXT presents an easy way of building functional robots from LEGO® blocks, sensors and motors. The LEGO® unit has an integrated battery pack, as opposed to six AA batteries. All the specifications for the LEGO® NXT unit can be seen in [LEGO GROUP, 2014]. Below are the specifications that are important to this project:

- 256 KB Flash memory
- 64 KB RAM
- 48 MHz CPU
- 4 input ports for sensors
- 3 output ports for actuators
- 100 x 64 pixel led display

This hardware is both limited in memory and computation speed compared to the hardware of a PC, as the ones used to implement JAWA on a PC. When designing JAWA this will be taken into consideration as a quick algorithm for the developer computer might take extended amounts of time on the LEGO® NXT unit.

2.1.1 Hardware Platform

The LEGO® unit is used as a mediator between the actuators and the sensors. The schematics of JAWA can be seen in Figure 2.1.

The arrows on Figure 2.1 represent data flow throughout the LEGO® NXT unit, the sensors and the actuators. The sensors give analogue input to the AVR, which is a micro controller that is capable of converting analogue signals to digital signals and vice versa. This step enables the software to process the information from the sensors and react according to the programming. The digitalised data from the AVR is transmitted via the I2C bus to the memory controller, where it is processed by the ARM CPU with the instructions from the application. The instructions from the application are fetched from the software container. The fetching goes through the firmware, then the operating system and finally the application itself. Signals to the actuators are sent according to the instructions of the application through the peripheral bridge and back to the I2C bus. The converted analogue signals are then sent to the actuators, which conclude the schematics of the LEGO® NXT unit. [Nielsen, 2014, pp. 82–83].

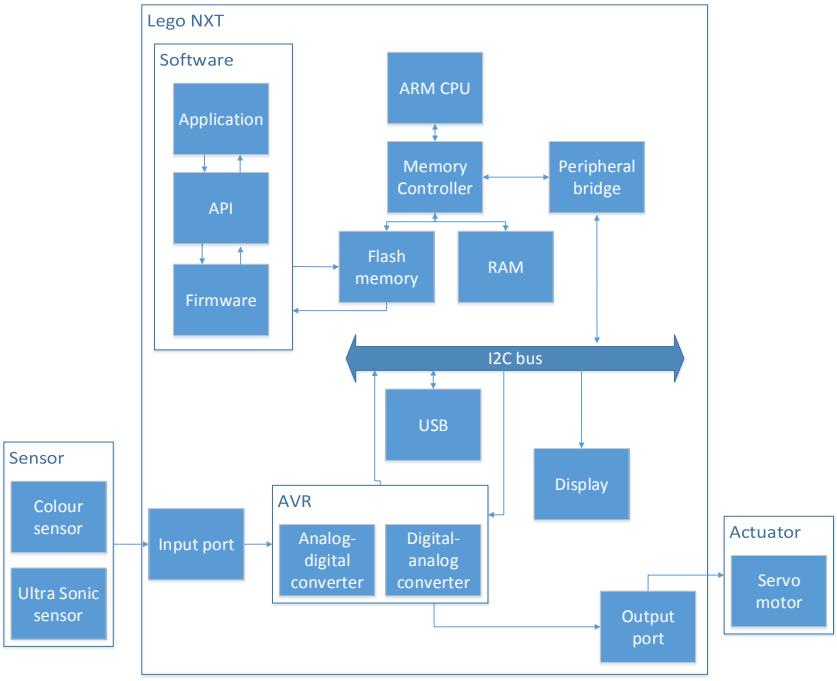


Figure 2.1: Schematics of the hardware of the NXT unit

2.2 Software Platform

Different software options can be used for creating programs on a LEGO® NXT unit. There are both visual –and text programming languages for LEGO® NXT. The visual based programming language comes with the drivers for the LEGO® NXT unit that allows for drag-and-drop type programming. The text based programming languages are APIs that allow for programming in C++ or Java. Two of the options for writing code for creating NXT programs are shown in the following sections. The differences and similarities between these two options will be outlined for the design in Chapter 3.
The two options that are discussed here are OSEK and leJOS. These were chosen because they are the most widely used languages for creating robots with the LEGO® NXT. There exists other platforms as mentioned before, but these will not be examined because they were deemed too similar or inappropriate.

2.2.1 OSEK

OSEK, *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen*, is an operating system for embedded control units [OSEK/VDX, 2014]. OSEK was created for coding automated embedded controllers for cars, but also has a branch for the NXT unit known as nxtOSEK. NxtOSEK controllers are written in the programming languages C or C++. The nxtOSEK platform consists of drivers of leJOS NXT C/Assembly source code [Chikamasa, 2013]. Since nxtOSEK uses the same driver code as leJOS, the main driver functionality between nxtOSEK C/C++ code and leJOS Java code are equivalent. OSEK supports fixed-priority pre-emptive scheduling for task at each priority level. At run time the task with the highest priority are selected for execution. If a task with a higher priority is dispatched it will be placed at the front of the ready queue for the current priority. Tasks that have the same priority level is dispatched into a first in first

out queue based on their static priority. OSEK also supports a priority ceiling to avoid priority inversions and mutual deadlocks. OSEK also supports event synchronisation allowing for waiting on multiple events. Each task can have multiple WaitEvent calls that each waits for a subset of events to finish execution [Feiler, 2003]. OSEK API is seen in [OSEK, 2014].

2.2.2 LeJOS

LeJOS, Java for LEGO® Mindstorms, is a Java Virtual Machine for programming LEGO® Mindstorm units [leJOS, 2014]. The leJOS Nxt platform allows users to code programs for the LEGO® NXT unit in Java. Contrary to OSEK, leJOS was made for the specific purpose of creating NXT software. Both leJOS and OSEK had a plugin that allowed for implementing classes and methods to software development tools such as Eclipse. That way it's easy to access the many features that the sensors and actuators have in a known interface. LeJOS scheduling policy is completely priority based. The scheduler for leJOS will always execute the thread with the highest priority. Each thread is given a priority at creation with a value between one and ten, one being the lowest and ten being the highest. If their exits more than one thread with that priority the leJOS scheduler will time slice them allowing them both to execute at a certain amount of time. For a lower priority thread to run a higher priority thread must cease to run, it must either exit, sleep or wait on a monitor. LeJOS also supports standard Java events allowing for handling external event to occur. LeJOS API is seen in [LeJOS, 2012].

Since both OSEK and leJOS are based on the same assembly code, meaning the functionality of the two approaches are the same. OSEK and leJOS also provides their own firmware for the Nxt unit. The main difference between OSEK and leJOS is the programming language used to create programs. OSEK is for C and C++ while leJOS is for Java. Both approaches had a well-defined documentation for looking up examples of code. Because the authors experience with programming in Java is better than C or C++, it has been deemed appropriate to choose leJOS for programming JAWA.

2.3 Function Analysis

In this section the functionality of JAWA will be further specified. These requirements are based on the preliminary problem statement, and will be used to specify which features on the sensors and actuators need to be tested. The requirements of JAWA are the following:

- Move pallets from a designated drop off point to some predefined location in the warehouse.
- Detect unforeseen obstacles, employees for instance, and be able to steer around them.
- Perceive the environment, to see static obstacles, such as walls, shelves and other stationary objects in a warehouse.
- Identify pallets, in order to sort them and know where the specific pallet is to be placed.

These functionalities require certainty that the LEGO® NXT sensors perceive the environment correctly and actuators yield the same result on the environment as expected. To verify this, these sensors and actuators has to be tested. Given the fact that JAWA has to

be able to perceive an environment, i.e. gain an understanding of the physical space it is in, JAWA will need a sensor to detect its surroundings. An ultrasonic sensor would suffice for this. Furthermore, JAWA will need a way to identify pallets. In the example given in Chapter 1 the method of identification was a barcode that was scanned. This may be difficult to achieve using LEGO® NXT hardware, so a colour sensor would have to suffice, and the barcodes will then have to be different coloured LEGO® bricks.
As for the actuators, JAWA needs to be able to move through the environment. This can be achieved by using the LEGO® NXT motors. The precision of these requires testing.
There are two ways of having JAWA navigate in the environment. Either through the use of a map of the environment with all of the walls, the drop off point and the conveyor belts already mapped for JAWA to traverse through. The other approach is to have JAWA use its sensors to navigate by itself. Not providing a map of the environment is more challenging, but also more attractive as it is the more flexible option. Both methods will be looked into and the tests will reflect that. The first option to be tested is the design with an unknown environment, if this is possible, this option will be used. If the tests conclude this to be implausible, a map will be provided to JAWA to traverse through.

The relevant tests for JAWA are therefore:

- Testing how accurate the ultrasonic sensor is at measuring distances.
- Testing how wide the spread of the ultrasonic sensor is.
- Testing whether different material has an effect on the ultrasonic sensor.
- Testing whether objects facing the ultrasonic sensor at an angle has an effect on the measurements.
- Testing which distance the colour sensor is most precise.
- Testing how the colour sensor represents the colours it reads.
- Testing how the motors perform when adding weight to JAWA.
- Testing whether driving many short bursts or one continuous stretch affects the motor accuracy.
- Testing the turning accuracy of the motors.

In addition to the individual sensor and actuator tests, it will also be determined whether the sensors and actuators can work together.

2.4 Testing Sensors

The sensors are what allow JAWA to perceive the environment. Being able to depend on the input, and that all measurements are precise, is critical for JAWA to act in the environment. If there is a difference in what is perceived, and how the environment actually is, JAWA might drive into walls or employees. To ensure that the challenges of the sensors are already known before the implementation, tests are conducted. These tests will act both as a verification that the sensors work as intended as well as giving an understanding of what actions might be necessary to be able to implement the functionality from the section above.

2.4.1 Ultrasonic Sensor

JAWA is using the ultrasonic sensor to perceive the environment. The ultrasonic sensor sends out ultrasonic sound waves through a tube and picks up the reflection of those waves with another tube, both pointing in the same direction. The ultrasonic sensor has an integrated circuit that calculates the distance using the time between sending and receiving sound waves. According to the specifications, the sensor ranges from 0 cm to 255 cm and has a deviation of ± 3 cm The ultrasonic sensor calculates distance in integers only. If a sensor cannot see an object or th object is out of range, the ultrasonic sensor returns 255cm
All specifications of the ultrasonic sensor are based on information from [Generationrobots, 2014].
For testing the ultrasonic sensor, it was important to test that the sensors worked as expected. The tests of the ultrasonic sensor involved returning the distance to the nearest object, finding the range in terms of spread, what materials the sensor could see, whether or not the ultrasonic waves reflect on objects and the sampling frequency of the sensor. All these tests are required to verify if JAWA can navigate in an unknown environment with current hardware.

Testing for distance

Objective: To figure out the minimum and maximum range at which the ultrasonic sensor could measure and what error margin is to be expected.
Expectations: The results were expected to be within 0 cm and 255 cm with an error margin of ±3 cm as mentioned in Section 2.4.1. The different heights were expected to show different results where the mounted sensor would lose precision at greater distances.
Setting: The sensor was placed on an even surface with no wall within 3 m for the first test, the sensor was placed flat against the ground. For the second test, the sensor was attached to the JAWA roughly 1.5 cm above the ground. The sensor was placed at the end of a 255 cm line with every 15 cm marked. A barrier was then moved from mark to mark exactly in front of the sensor and the ultrasonic sensor output was compared to the distance. Figure 2.2 shows a model of how the distance test was conducted, "—" represent no results as the sensor is no longer able to detect the object.
Results: All results can be seen in Table 2.1 and Table 2.2. The results shows a greater loss in precision with lower distances for the sensor lying flat on the ground while the sensor mounted on the JAWA lost more precision on greater distances. The turn over point was at 45 cm where both placements had near the same margin of error. The mounted sensor could measure up to 210 cm before getting imprecise while the sensor flat on the ground could measure up to 240 cm.
Conclusion: It was preferred to have the ultrasonic sensor placed on the ground since it allowed for greater precision at both long as well as short distances. However, this placement would not be used since having the ultrasonic sensor scrape along the ground was not viable.
Therefore the ultrasonic sensor would be placed as close to the ground as possible, even though some distances are not measurable it was still viable since objects would most likely be more than 15 cm away from the sensor.
For the following ultrasonic sensor tests all measurements will be done with 50 cm intervals.

Sensor lying flat on the ground

Distance (cm)	Measured (cm)	Difference (%)
15	21	28.57
30	32	6.25
45	46	2.17
60	61	1.64
75	76	1.32
90	90	0.00
105	104	-0.96
120	119	-0.84
135	134	-0.75
150	149	-0.67
165	164	-0.61
180	179	-0.56
195	194	-0.52
210	209	-0.48
225	224	-0.45
240	239	-0.42

Table 2.1: A table depicting the difference in the measured data and the expected results. The sensor is lying flat on the ground.

Sensor mounted on the robot

Distance (cm)	Measured (cm)	Difference (%)
15	17	11.76
30	29	-3.45
45	44	-2.27
60	58	-3.45
75	73	-2.74
90	88	-1.27
105	103	-1.94
120	118	-1.69
135	133	-1.50
150	148	-1.35
165	163	-1.23
180	178	-1.12
195	193	-1.04
210	208	-0.96
225	220	-2.27
240	—	5.88

Table 2.2: A table similar to the table above, however, here the sensor is mounted on the robot.

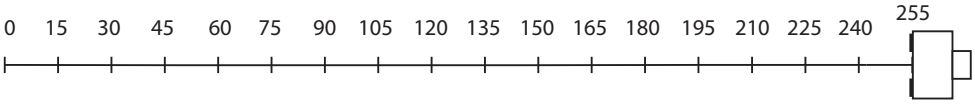


Figure 2.2: A model of the ultrasonic distance test. The maximum range of the sensor is 255 *cm*, and a line was marked every 15 *cm*.

Testing for spread

Objective: The spread of the ultrasonic sensor refers to the angle at which the receiver of the sensor was able to pick up the reflected sound of an object. The objective of this test was to determine the actual angle at which the sensor was able measure objects and if this angle was consistent for all distances. In addition, the accuracy of the sensor was also to be determined.
Expectations: Based on the specification used for the ultrasonic sensor, a standard ultrasonic sensor has a measuring cone of approximately 30°. Therefore, this was the measuring angle expected when testing. The accuracy was expected to be similar to what expected from the distance test in Section 2.4.1
Setting:
It was unknown whether the specifications had the measuring cone from the middle of the sensor, the transmitter or the receiver of the ultrasonic sensor. The ultrasonic sensor was mounted on JAWA facing a 60° and 2.5 *m* long cone with no objects were placed within it. From the base of the sensor, cones with 10°, 20° and 30° on either side was drawn out to the maximum measuring distance of the sensor. A barrier like object was placed at distances with intervals of 50 *cm* for each of the drawn cones. Figure 2.3 shows a model of the test, as seen from a top down perspective.

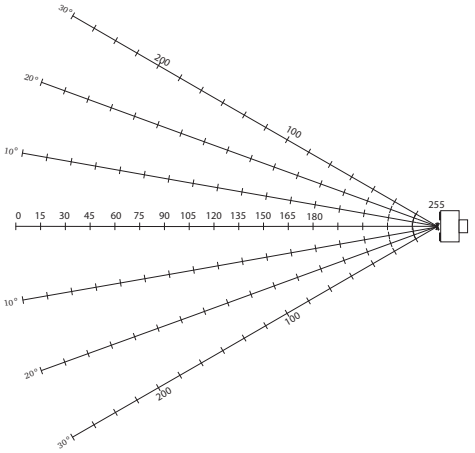


Figure 2.3: A model of the spread test.

Results: It was deducted that measurements was within a 0 – 6 % margin between all distances from 50 to 200 *cm*. More precision was lost at higher range until over 200 *cm* as seen in Table 2.2 where it was noticed that the sensor struggled to measure distances over 200 *cm* when attached to JAWA. It was also deducted that the sensor was not able to see any objects for the 30° angle on either side of the sensor from distances over 150 *cm*.
For short distances of 100 *cm* and less, the ultrasonic sensor has a measuring cone of 60°. This cone decreases to 40° as the distances increase beyond 100 *cm*.
Table 2.3 shows the data collected from the test. A "—" in the table denotes a measurement of 255 *cm* which was considered as the sensor not being able to see the object.
Conclusion: The actual measuring cone of the ultrasonic sensor is larger than expected, it was necessary to keep this in mind when developing JAWA, since it most likely would be difficult to tell where the object actually was within the cone. Continuous checks with the ultrasonic sensor would be necessary to ensure that JAWA was on course according to the objects location.

Distance(cm) \ Degrees		-30°	-20°	-10°	10°	20°	30°
50		52	51	51	52	52	51
100		98	99	102	103	101	102
150		—	150	150	146	148	—
200		—	195	194	196	196	—
250		—	—	—	—	—	—

Table 2.3: Results from the spread test of the ultrasonic sensor.

Testing for materials

Objective: To specify whether the material of an object would have an effect on the data retrieved from the sensor.
Expectations: It was expected that the density of the material would have an impact on the measurement; lower density would mean that the sensor would have a harder time measuring the distance to the object.
Setting: The test was conducted with a cube made of LEGO® blocks and a chalkboard eraser with a rough and soft surface. These objects were placed at a 50 cm. directly in front of the sensor.
Results: The results showed that the ultrasonic sensor was able to measure the LEGO® cube but not the chalkboard eraser at 50 cm. Further testing of the chalkboard eraser showed that it was invisible to the ultrasonic sensor at all ranges.
Conclusion: The test showed that there were limitations to what type of objects the sensor could do measurements on. The reason why the chalkboard eraser was invisible would be because of the low density, the sound waves sent from the ultrasonic sensor were simply absorbed. This also meant that it was necessary to conduct tests on all objects in an environment to make sure that they were visible to the sensor.

Testing for angle

Objective: The purpose of this test was to determine if objects set on angles would appear to be further away or even invisible to the sensor. The test was performed both on a cube representing a pallet and a hard LEGO® ball. The round shape could symbolise objects like oil drums.
Expectations: The results for the cube were expected to have it become unmeasurable at some angles as the ultrasonic waves are dispersed and not caught by the sensor. The results for the LEGO® ball were expected to always return the correct distance as there would always be a surface pointing in the direction of the sensor.
Setting: The cube was turned with a 5° interval until the object was turned 90 degrees. The object was placed 50 cm directly in front of the sensor. Spinning a cube would result in equal offset for both the transmitter and receiver of the sensor; therefore a 90° would be enough for the sensor to have measured all possibilities. The angle test was performed with an 8×8 cm cubic LEGO® brick with a smooth surface. Figure 2.4 shows a model of the test. The LEGO® ball was placed 50 cm away from the sensor and had a diameter of 5 cm.
Results: The results, shown in Table 2.4, indicate that the cube was only measurable when spun less than 15° in either direction. The results of the LEGO® ball did not give any results for a distance of 50 cm.
Conclusion: Wrong measurements were easy to get for sharp objects. As expected, the object became completely invisible to the sensor at one point, the angle in which the sensor lost track of the cube was any angle above 15°, with no preference to the

Degrees spun	Measured (cm)
0°	50
5°	50
10°	50
15°	—
30°	—
45°	—
60°	—
75°	—
80°	50
85°	50
90°	50

Table 2.4: Test of ultrasonic sensor on a cubic object rotated at different angles.

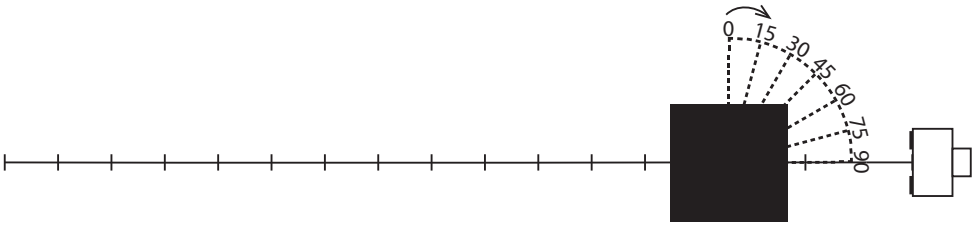


Figure 2.4: A model of the angle test performed on a cubic object at various degrees.

transmitter and receiver. The LEGO® ball was invisible at the distance of 50 cm. The ball was moved closer to get an idea of the range at which a ball was visible. It was noticed that the sensor was able to detect balls at distances of approximately 20 cm.

Testing for sampling frequency

Objective: To determine the sampling rate of the ultrasonic sensor as well as the time needed to scan all degrees for a complete 360° turn.
Expectations: It is expected that the sensor performs by the specification from LEGO®, which showed a sensor sampling rate of 3 *ms* for most sensors, but did not have any frequency data for the ultrasonic sensor [LEGO GROUP, 2014]. It was expected that the test would show results similar to the sampling rate of the other sensors as seen in the specifications from LEGO®.
Setting: The ultrasonic sensor was set to measure continuously for a specified time in *ms* while at the same time reading the distance. The method used for measuring the distance was made in such a way that the sensor would wait for the distance data to become available before returning. A while-loop was constructed to read the distance and count the number of times it was run through. The while loop was set to stop when the last result returned after 1,000, 10,000 and 1,000,000 *ms*, which because of the delay became 1,020, 10,015 and 1,000,025 *ms*. The test was run five times for both 1,000 *ms* and 10,000 *ms*
Results: For both 1,000 *ms* and 10,000 *ms* the test yielded a consistent result for each run where both the time spend in the while loop, and the amount of measurements were consistent. In addition with the results from the 1,000,000 *ms* test it was noticed that increased run time resulted in lower sampling frequency. The test showed a frequency about 10 times slower than the expected sampling frequency, even for the 1,000 *ms* run

times.
The results of the test can be found in Table 2.5.
Conclusion: If JAWA were to turn 360° and measure distances for all angles it would require about 12.24 seconds for a complete turn with an average of 34 *ms* per sample.

Time *ms*	number of measurements	time per sample $\left(\frac{ms}{samples}\right)$
1,020	31	32.90
10,015	288	34.77
1,000,025	28,574	35.00

Table 2.5: Test of sampling rate for the ultrasonic sensor.

2.4.2 Colour Sensor

JAWA uses a colour sensor to determine whether the object, it has picked up, is the same colour as expected.
The colour sensor for the LEGO® NXT provided in the box-set can distinguish between the six basic LEGO® colours red, blue, green, yellow, white and black.
The colour sensor uses an RGB LED which alternates between red, green and blue to shine on an object. Based on the reflection returned, the sensor is able to determine the colour. The colour readings are updated 100 times a second, which is one sample every 10 *ms* [University of Windsor, 2011]. The colour sensor had to be able to determine the colour of the objects to distinguish them. For this reason, it is prudent to test for the accuracy of the sensor and what effects might trigger a bad reading. First off, it is interesting to see what effect the distance between the sensor and the object has on the readings. Additionally, it is valuable to test the sensor output since the output has 8 colour IDs with only 6 colours. The last test for the colour sensor was the impact from room lightning.

Testing for distance

Objective: To determine at which distance the colour sensor, needs to be away from the object to be most accurate.
Expectations: It was expected that greater distances would result in less accurate readings.
Setting: The test was conducted on LEGO® bricks of the colours red, green and blue in a well lit room. These three colours was chosen because they were the same colours used for the LED in the sensor. The bricks were placed at distances from 0 – 4 *cm* away from the colour sensor. The test was conducted using `getColorID()`, at this point it was still not known what ID belonged to what colour, but it was not necessary for this test.
Results: The results of the test can be seen in Table 2.6.
It was noticed that at both 0 and 4 *cm* the sensor returned the ID 7, this was interpreted as the ID for lack of colour, or black. The green and blue bricks were only measurable on distances 1 and 2 *cm*, whereas readings on 3 *cm* returned ID 7. The red brick was measurable on distances from 1 – 3 *cm*.
Conclusion: It was concluded that all the bricks were noticeable at both 1 and 2 *cm* distance while the red block in addition was readable on 3 *cm*. Furthermore it was assumed that the most accurate distance to do measurements of colours would be at 1 *cm*. This was because of the fact that only bricks of the same colour of the sensors LED had been used, therefore it was possible that some of the remaining colours wouldn't be

readable at 2 *cm*. Because of the green and blue brick only being readable on distances 1 and 2 *cm*, it was chosen to do further testing using another method for determining the colours of a brick, namely `getColor()`. This will be detailed further in the following test.

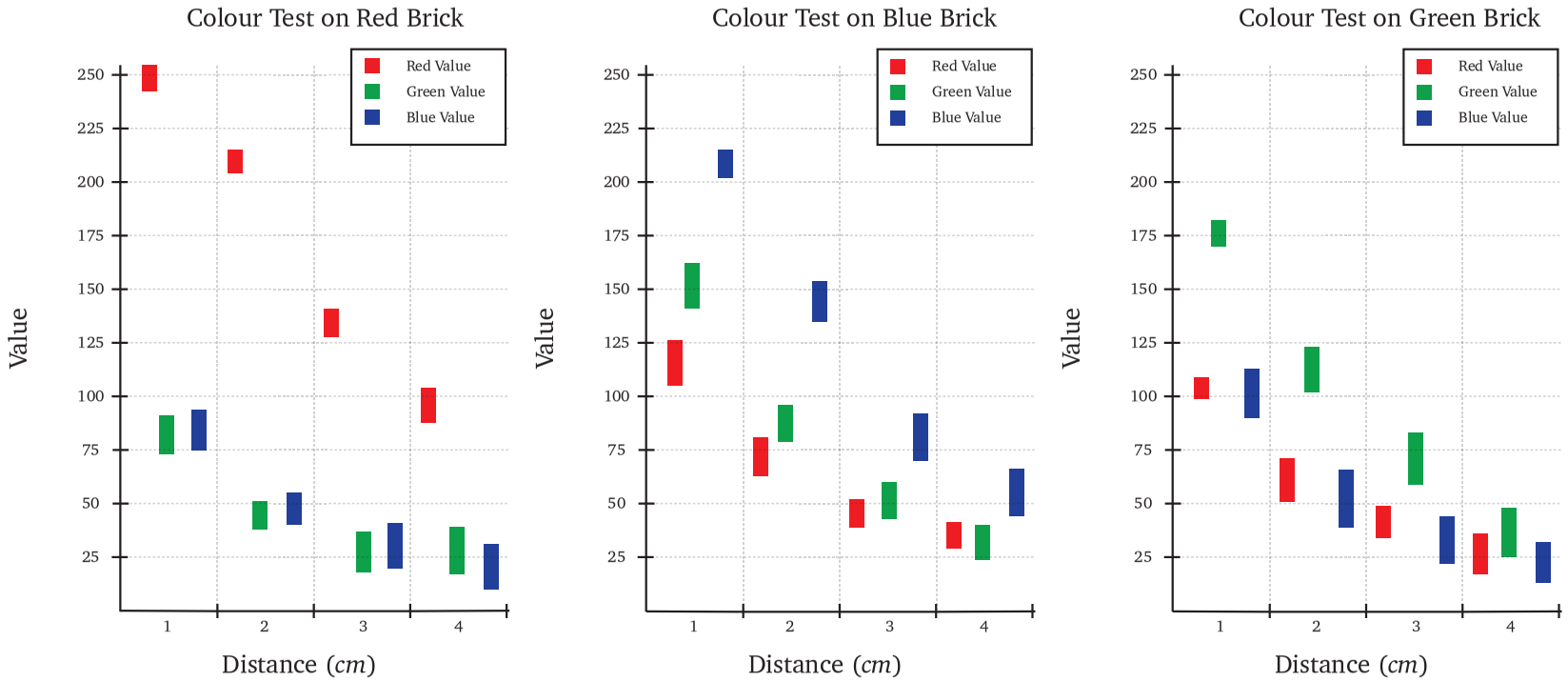
Brick colour / Distance to brick (*cm*)	Red	Green	Blue
0	7	7	7
1	0	1	2
2	0	1	2
3	0	7	7
4	7	7	7

Table 2.6: Test of colour sensor precision based on distance. Each number represents the ID of colour.

Objective: To further determine at what distance the colour sensor readings were most accurate by using `getColor()`.
Expectations: It was expected that this test would provide a more precise view of the actual distance at which the sensor was most precise. Furthermore it was expected that higher numbers of a specific RGB value would prove it easier to read the actual colour of the brick.
Setting: The setup for this test was identical to the previous test except for the `getColor()` method instead of `getColorID()`. `getColor()` works by returning a colour object which holds the colour ranges for each of the RGB values of the measured object. For the colour object it is possible to use a method; `getRed()`, `getGreen()` or `getBlue()` to retrieve the corresponding colour values between 0 – 255.
Results: The results of the test can be seen in Figure 2.5. The results shows a decrease in precision for all colours for distances above 1 *cm*.
Conclusion: The colour sensor has the highest precision if placed 1 *cm* away from the object it evaluates, this distance will be used in the remaining colour sensor tests. Measurements on the red brick are significantly more precise than the other colours tested.

Testing for colours

Objective: To determine what colour a specific ID corresponds to using the `getColorID()` method.
Expectations: It was expected that each coloured brick would have its own specific ID within the values from 0 – 5. From the test above it can be expected that red has ID 0, green has ID 1, blue ID 2 and black/no colour has the ID of 7. What the remaining colours corresponds to must be found out from the test results.
Setting: Each coloured LEGO® brick was placed 1 *cm* away from the colour sensor as it was found to be most precise, as seen in the test above. The ID returned for each brick was then noted and put in a table.
Results: It was discovered that each coloured brick corresponds to its own ID, but that the IDs does not range from 0 – 5, but rather from 0 – 7. It is unknown what colour, ID 4 and 5 corresponds to, if any at all.
The results of this test are presented in Table 2.7.
Conclusion: It could be concluded that each of the colours red, green, blue, yellow and white had its own specific ID. The ID of black was the ID for no result. There is no way to telling whether or not there is a black pallet. Though the IDs of 4 and 5 were not discovered, it could not be concluded what colour these would correspond to, if any at



(a) A range plot of the colour sensor distance test performed on a red LEGO® brick.

(b) A range plot of the colour sensor distance test performed on a blue LEGO® brick.

(c) A range plot of the colour sensor distance test performed on a green LEGO® brick.

Figure 2.5: Three range plots depicting the distance test for the colour sensor. These are the result of the RGB test. The x axis is the distance from the sensor to the brick, measured in centimetres. The y axis is the raw value of the colour ranging from 0 to 255. The height of the bars represents the range from the lowest to the highest measured value.

Brick Colour	Colour ID
Red	0
Green	1
Blue	2
Yellow	3
White	6
Black	7

Table 2.7: Result of using `getColorID()` on the colour sensor.

all. To make sure that the measurements would be consistent, the usage of the IDs 0 – 2 would be used for colour marking the pallets for JAWA.

Testing for room lighting

Objective: To determine if dimmer room lighting has any impact on the precision of the colour sensor. This will decide whether or not the warehouse has to have uniform lighting.
Expectations: It is expected that the colour sensor will still be able to determine the colour of an object since the RGB LED gives of a small amount of light itself.
Setting: To simulate a completely dark room, the colour sensor was covered by a bucket, along with an object of a specific colour. All test cases had the colour sensor 1 *cm* away from the objects since it was determined in Section 2.4.2 that this gave the best results. The results can be seen in Table 2.8
Results: The results shows no differences between a normal room as shown in Table 2.7, and a dark room.
Conclusion: Room lighting between a normally lighted warehouse and a completely dark warehouse has no difference in the precision of the colour sensors measurements. JAWA can work in complete darkness.

Brick colour	Colour ID
Red	0
Green	1
Blue	2
Yellow	3
White	6
Black	7

Table 2.8: Test of colour sensor in the dark.

2.5 Testing Actuators

For JAWA to be able to traverse the environment, it needs actuators. These actuators are the motors in the LEGO® NXT hardware. They need to be tested, to gain knowledge of the performance of them. As such this section is for testing these actuators.

2.5.1 Motor

The motors needs to be tested to ensure that the reaction, JAWA makes, in the environment is what is expected from each action. If the motors are out of sync, the robot could end up taking longer to solve the task than needed. For acting in an unknown environment, the sensors are much more important for navigating than motors, therefore only a single test will be performed since the main goal of the motors is to be efficient. To test the efficiency of the motors, a weight test will be conducted.

Testing for Weight

Objective: To ensure the motors are able to reach the starting destination after picking up a pallet.
Expectations: If the motors are exposed to too heavy objects, they are expected to stop and/or break. This test, however, was for investigating whether a lighter object might have an effect on the efficiency of the motors. The results were expected to show no variation between distances travelled with different weight as long as the wheels can spin.
Setting: JAWA was set up against the wall. It was set to drive a set amount of rotations and stop. The distance travelled from the wall was noted and can be seen in Table 2.9
Results: The results shows a loss in precision with increasing weight.
Conclusion: When JAWA picks up a pallet weighing roughly the same as itself, it will have to travel roughly 2% more to get back to the starting position. This is important to keep track of if the navigation back to the starting position is based on the distance travelled.

Weight(g)	570	773	964	1106	1245	1332
Distance(cm)	136	135,5	135	134	133	133

Table 2.9: Test of distance driven, with weight.

2.6 Radar

This is an evaluation of a radar design with an unknown environment based on the tests performed in the previous sections. For JAWA to be able to explore the environment, it would have to spin around itself like a radar and return the distance to all objects. Based on the results from the angle test Section 2.4.1 on page 12, the radar test will most likely fail in all cases where the pallets are placed with angles greater than $\pm 15^\circ$ away from JAWA. The results of the LEGO® balls showed that these objects cannot be used instead. Based on these results, it can be deduced that using a radar for an unknown environment is unfeasible, since both sharp and round objects are rarely or never detected by the ultrasonic sensor. Changing from an unknown environment to a known environment poses a new set of challenges for JAWA. One of the requirements of a known map is for JAWA to be able to traverse through a map. This requires precise steering as any error in motor rotation will accumulate over time. New tests are created for measuring the precision of the motors. These will be presented in the following section.

2.7 Accuracy Tests

The following tests will focus on the precision that the motors must have in order for JAWA to traverse the warehouse environment. All tests in this section has been conducted using both `getTachoCount()` and `rotate()` to control the rotations of the wheels of JAWA. `getTachoCount()` was used in combination with `forward()` and `backward()` to specify how much the motors should rotate.
It was chosen to do so because it was necessary to determine which option was more precise for either driving straight or turning.

2.7.1 Testing for Distance Accuracy

Objective: To determine whether or not JAWA would be able to travel a specific distance without any loss in distance covered.
Expectations: It was expected that both rotating options would result in a no loss of precision as both methods used the wheels diameter to calculate the total rotations that the motors had to perform. If there was loss in precision, the test for steps of 10 cm were expected to result in greater loss than for the continuous test.
Setting: For both control-methods, two types of tests were performed. For the first test, JAWA was programmed to drive 10 times of 10 cm with a break between each step. For the other test it would drive 100 cm continuously. To use the `getTachoCount()` method, a while loop was needed to check a condition, and in this while loop each motor was set to rotate backwards as long as the condition held. Each test was conducted three times.
Results: Small deviations was noticed for `rotate()` test, whereas larger deviations were observed when `getTachoCount()` was used. For `rotate()` The 10×10 cm was almost as precise as the continuous, but resulted in a small offset of an average of 2 cm centimetres more than expected, whereas continuously driving only deviated with $+1$ cm for one of three tests.
Using `getTachoCount()` had a deviation of $+2-3$ cm and a slight turn when stopping for driving continuously, while JAWA for the 10×10 cm test had driven 100 cm forward and rotated about $25°$.
Conclusion: Using `getTachoCount()` for driving straight was completely dismissed. The problem with this was that JAWA would rotate a bit to one side when stopping. This was noticed for both the continuously and 10 times 10 cm test.
For using `rotate()` the results showed that driving continuously was a few centimetres more precise than starting and stopping the motors frequently. Because of these observations, JAWA would use `rotate()` in addition to being programmed to drive with as few stops as possible.

2.7.2 Testing for Turning Accuracy

Objective: To determine how accurate JAWA was able to turn around its own axis.
Expectations: It was expected that both options to rotate, would result in a loss in precision, similar to the previous test. Whether one option was more precise than another was unknown and would be determined after the test was completed.
Setting: JAWA was programmed to turn with one wheel and with both wheels to figure out which option was most precise. It was decided to do the test with a turning degree of $90°$, which allowed for easier observation of the precision. During the test, JAWA would turn $8 \times 90°$ around its own axis for a total of $720°$.
Results: For turning with both wheels it was discovered that both `getTachoCount()`

and `rotate()` was very imprecise. Using the `rotate()` method resulted in JAWA rotating a total of 660°, whereas the use of `getTachoCount()` resulted in JAWA rotating about 800° total degrees.
Rotating with only one wheel when turning resulted in much more precise rotations for both methods. One wheel rotation with `rotate()` resulted in JAWA rotating about 700°, where `getTachoCount()` made JAWA turn about 735°.
Conclusion: From this test it can be concluded that one wheel rotation using `getTachoCount()` is the most precise way of turning JAWA, even though one wheel rotation using `rotate()` is almost as precise. However, this poses a problem. If JAWA has to follow a map, it has to know when it is in a specific point. This is much simpler if JAWA turned around its own axis, so as to follow the map more closely. therefore, to make JAWA turn accurately with both wheels, some compensation has to be implemented, so it arrives at the points it is supposed to.

2.7.3 Testing for Combined Distance and Turning

Objective: To determine if JAWA was able to drive and turn in combination without losing too much precision.
Expectations: It was expected that JAWA would be slightly off the origin point, but still close enough so that it would not cause any problems.
Setting: For this test, the tests in Section 2.7.1 and Section 2.7.2 were combined to make JAWA drive in a 1×1 m box. The results in the two previous test showed that `rotate()` was best for driving straight while `getTachoCount()` was better for turning, therefore this test was conducted under these conditions.
JAWA drove 1 m then turned 90° and repeated this four times so that it had returned to its origin point.
Results: The worst test resulted in JAWA being 12 cm away from the origin point with only a slight loss in turning precision. The best, on the other hand, resulted in JAWA being almost precisely on the origin point but again with a slight loss in turning precision.
Conclusion: The test showed that it was possible to make JAWA drive with only a slight loss in precision. The loss was mostly based on the loss in precision each time JAWA turned, but also based on how precise JAWA was placed at the beginning of the test.
It could be concluded that this way of controlling JAWA was viable, but some slight tweaks were necessary. Problems were likely to occur when JAWA was tasked with driving multiple distances with a lot of turns, therefore the code had to be tweaked to make up for this, so it would remain as precise as possible even for long work sessions.

2.7.4 Summary

After the testing of the two sensors and the motors it can be concluded that the sensors should be handled carefully and the results of the tests taken into consideration.
The colour sensor is most accurate if the range between the sensor and the object is 1 cm, as seen in Section 2.4.2 on page 14, and the detectable colours are the colours in Table 2.7 on page 17.
JAWA will travel a shorter distance if the weight applied to it increases, as seen in Section 2.5.1 on page 18. A deviation in distance travelled also occurs when the motors are tasked with rotating a specific degree with multiple stops and starts in between, as seen in Section 2.7.1. Therefore JAWA should strive to travel distances in one consecutive run.
Objects to be located by JAWA must be between 15 and 210 cm away from the sensor, as seen in Section 2.4.1 on page 9. The precision of the spread of the ultrasonic waves should be accounted for, seen in the test in Section 2.4.1 on page 11. In addition, sharp objects are not detectable on angles larger than 15°, because the ultrasonic waves are not

reflected back to the receiver, as seen in Section 2.4.1 on page 12. The ultrasonic sensor is not able to detect soft materials, such as the chalkboard erasers as seen in Section 2.4.1 on page 12, therefore there should be refrained from using objects like these. In addition, all objects in the environment will have to be tested for visibility to prevent unforeseen outcomes.

From the knowledge gathered of the hardware and the knowledge obtained after changing JAWA to compensate for this, it is noticed that a map is the best way of navigating an area.
The precision of JAWA can be tweaked to make JAWA precise enough to navigate an area using nothing but straight movement and 90° turns.

2.8 Map

As determined in Section 2.6, JAWA will follow a map of the warehouse. Two ways of doing this is presented below; bitmap and graph. These will be explained and the approach will be chosen.

2.8.1 Bitmap

By making a matrix of, for instance, **`boolean`**s it is possible to represent an environment. An entry in the matrix will then look like coordinates, $[x][y]$, and the value of that coordinate tells whether it is walkable or not, i.e. whether it is possible to go there. This can be seen as a map of the primitive type **`boolean`** to coordinates in the environment, and this can be called a bitmap. To represent obstacles, such as walls, in the environment, the coordinates that correspond to this can be set to false. By doing this the robot will never consider going to that coordinate because it is not walkable, and thus the robot will not drive into the wall, given it follows the map precisely.

2.8.2 Graph

Another approach is to make a graph representing the environment. The warehouse environment would be modelled as an undirected, uniformly weighted graph, where the vertices are coordinates, the edges are paths between the coordinates, and the weights denote the distance between coordinates. It is undirected so that the robot can freely move from vertex to vertex, and uniformly weighted means all edges have the same weight. Furthermore the graph would only have horizontal and vertical edges so the graph becomes a grid. As with the bitmap if an obstacle needs to be modelled, it would suffice to remove connections to the coordinates making up to obstacle. This could be done by either specifying the weight on the edge, going to the wall, to be near infinity. Another way would be to entirely remove the edge. Both ways would result in a traversing algorithm not considering the points, where there are obstacles.

Due to the abundance of traversal algorithms that already exists for graph problems, and the flexibility of it, a graph has been chosen for representing the warehouse. Furthermore, leJOS provides a whole library for creating and manipulating graphs, so it can be assumed that it is optimised for LEGO®NXT.

2.9 Algorithms

Since it was chosen to use a graph, for mapping the environment, an algorithm for calculating the shortest path is needed. In this section, different algorithms to find the shortest path are presented. Three algorithms will be detailed: Dijkstra's, A* search and jump point search.

2.9.1 Dijkstra's Algorithm

Dijkstra's algorithm is often synonymous with shortest path algorithms. It is one of the most well-known path finding algorithms, because it guarantees a shortest path and is relatively fast. It is used to find the shortest path through a weighted, undirected graph, with non-negative edges. In terms of complexity, Dijkstra's algorithm has a running time corresponding to number of vertices squared, i.e. $O(|V|^2)$[Wikipedia, 2014b].

```
1  function Dijkstra(Graph, source):
2     dist[source]  := 0                     // Distance from source to source
3     for each vertex v in Graph:            // Initializations
4         if v \neq source
5             dist[v]  := infinity           // Unknown distance function from
                  ↪ source to v
6             previous[v]  := undefined      // Previous node in optimal path from
                  ↪ source
7         end if
8         add v to Q                         // All nodes initially in Q
              ↪ (unvisited nodes)
9     end for
10
11    while Q is not empty:                  // The main loop
12        u := vertex in Q with min dist[u]  // Source node in first case
13        remove u from Q
14
15        for each neighbor v of u:          // where v has not yet been removed
              ↪ from Q.
16            alt := dist[u] + length(u, v)
17            if alt < dist[v]:              // A shorter path to v has been found
18                dist[v]  := alt
19                previous[v]  := u
20            end if
21        end for
22    end while
23    return dist[], previous[]
24 end function
```

Listing 2.1: Dijkstra's algorithm.

2.9.2 A* Search Algorithm

This algorithm is an optimised version of Dijkstra's algorithm. The optimisation lies within implementing heuristic reasoning into the existing algorithm.
The algorithm uses a knowledge-plus-heuristic cost function, $h(x)$, which finds the heuristic estimate between two functions in the following way:

- The path cost from the initial node to the current node.

- The future path cost, which is a heuristic estimate from the current node to the goal node. This is often a straight line through nodes. Because it is usually the shortest admissible physical distance between two nodes.

$h(x)$ is then used to determine which node should be visited next.
The algorithm uses the heuristic estimate to prioritise the priority queue. This ensures that it always tries to move in a straight line to the goal. If it is obstructed it takes the second highest priority node in the queue.
The time complexity of this algorithm is $O(log(h^*(x)))$, i.e. the logarithm of the optimal heuristic function. Due to this relatively low time complexity it is widely used in artificial intelligence[Wikipedia, 2014a].

```
1  function A*(start,goal)
2      closedset := the empty set    // The set of nodes already evaluated.
3      openset := {start}    // The set of tentative nodes to be evaluated,
           ↪ initially containing the start node
4      came_from := the empty map    // The map of navigated nodes.
5
6      g_score[start] := 0    // Cost from start along best known path.
7      // Estimated total cost from start to goal through y.
8      f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)
9
10     while openset is not empty
11         current := the node in openset having the lowest f_score[] value
12         if current = goal
13             return reconstruct_path(came_from, goal)
14
15         remove current from openset
16         add current to closedset
17         for each neighbor in neighbor_nodes(current)
18             if neighbor in closedset
19                 continue
20             tentative_g_score := g_score[current] +
                   ↪ dist_between(current,neighbor)
21
22             if neighbor not in openset or tentative_g_score < g_score[neighbor]
23                 came_from[neighbor] := current
24                 g_score[neighbor] := tentative_g_score
25                 f_score[neighbor] := g_score[neighbor] +
                       ↪ heuristic_cost_estimate(neighbor, goal)
26                 if neighbor not in openset
27                     add neighbor to openset
28
29     return failure
30
31 function reconstruct_path(came_from,current)
32     total_path := [current]
33     while current in came_from:
34         current := came_from[current]
35         total_path.append(current)
36     return total_path
```

Listing 2.2: A* search algorithm.

2.9.3 Orthogonal Jump Point Search

Jump point search is an optimised version of the A* algorithm. Jump point search tries to look ahead and skip nodes that are not important to look at. Assumptions are made

for the current node and its immediate neighbours. This is called *graph pruning*. Firstly the parent node can be ignored, because it was the last node visited. Then the nodes diagonally behind the current node can be ignored, also the nodes above and below the current node can be ignored, because they would have been reached optimally, from the parent. The nodes diagonally in front of the current node can be reached by the node above and below, for the same cost. Therefore only the node in front of the current node needs to be examined.

When there are no obstacles, the algorithm can jump ahead, without having to save nodes, but if an obstacle appears above or below the current node the algorithm has to stop and re-examine the node diagonally upward or downward depending on the placement of the obstacle. This is because the neighbours cannot be reached from the parent, without going through the current node. This node is called a forced neighbour, because it needs to be considered. When reaching a forced neighbour the algorithm stops jumping in that direction, and saves the forced neighbour in the priority queue and runs the A* algorithm. When the A* algorithm runs, one of the jump points in the queue gets selected, then the algorithm jumps to this node and continues. The algorithm expands by searching horizontally, after its done with a line it moves up vertically and so forth[Witmer, 2013].

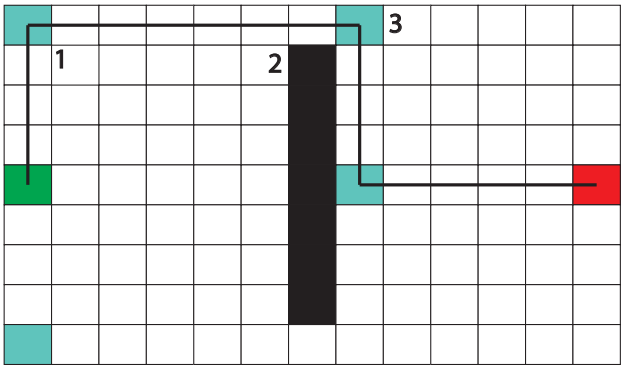


Figure 2.6: The jump point algorithm, with a green start node and a red end node.

```
1 function JPS(start, goal) {
2 //run A* with the following modification to the if statement on line 18
3
4 if neighbor in closedset AND neighbor reachable from parent without going
      ↪ through current
5     continue
6 }
```

Listing 2.3: Pseudo code for jump point search. label

In Figure 2.6 an orthogonal jump point search has calculated a route from the start point, green, to the goal, red. When the algorithm runs it meets the obstacle, **2**, and makes a forced neighbour, **3**. Between the starting point and **3**, a jump point is made, **1**, to connect these.
The worst case time complexity of the algorithm is the same as A*. However, the average case is better as the pruning of jump point search is in constant time, the orthogonal jump point search, will be referred to simply as jump point search.

2.9.4 Algorithm Evaluation

Based on the algorithms presented above, it has been deemed most appropriate to select the jump point search algorithm specifically for this project. When analysing the presented algorithms they all found the shortest path. The A* and jump point search algorithms both had a better time complexity than Dijkstra's algorithm. In Section 2.7.1 the tests showed that having JAWA stop many times would result in a loss of accuracy. The issue with the A* and Dijkstra's algorithms is that when calculating the shortest path, they depend on making many turns. The shortest path is often an oblique line to the goal, but if A* or Dijkstra's should be used for this project the cost of making a turn or a stop, should have a reasonable cost, to make as few turns as possible. In the orthogonal Jump point search this feature is already included, but still the algorithm makes more turns than necessary. Finally, Jump point search is faster than A* and Dijkstra's algorithm, in average case. This all results in jump point search being the most appropriate algorithm for JAWA.

2.10 Problem Statement

Based on the results from the analysis a more concise problem statement has been made. This problem statement will be based on the warehouse case presented in Section 1:

Can an autonomous robot be implemented in a environment to traverse through a map of a warehouse, using the shortest available path with the least amount of turns, while avoiding obstacles, using the limited memory and computational power offered by the LEGO® NXT platform?

In Figure 1.1 a computer is shown to communicate with JAWA. However, this will not be the case for the final product.

Following the problem statement, certain requirements must be made. These requirements will be based on the results of the tests in addition to the environment being a warehouse.

Follow a map - JAWA must have a map that resembles a warehouse environment. The jump point search algorithm will be used to calculate the shortest distance to and from different locations in the warehouse, and do this with the least amount of turns.

Identify pallets - JAWA must determine the distance to and colour of the pallets in order to distinguish them from each other and afterwards place them on the right conveyor belt.

Move pallets - JAWA must be able to move the pallets that have been identified. This includes both picking up and driving around with the pallet in addition to placing the pallet and leaving it.

Avoid collision - JAWA must be able to detect obstacles, like walls, employees etc. and avoid colliding with them.

These requirements will be used as design guidelines for building JAWA. The requirements will also be evaluated at the end of the project to see if they have been satisfied in the final product.

3 | Design

In this chapter the design of JAWA will be described. The different design decisions will be explained in regards to the tests done in Section 2.4 and through Section 2.7 and a description of how the different sensors and motors have been placed will be detailed. The design is a critical part of making sure that its possible to achieve the goal. It is therefore important that the design decisions are chosen carefully to reach of the best possible end result.

3.1 Design Prerequisites

Before the design of JAWA is finalised, some clarification regarding to the functionalities of JAWA should be explained. This will follow up on the requirements that were established in Section 2.10.

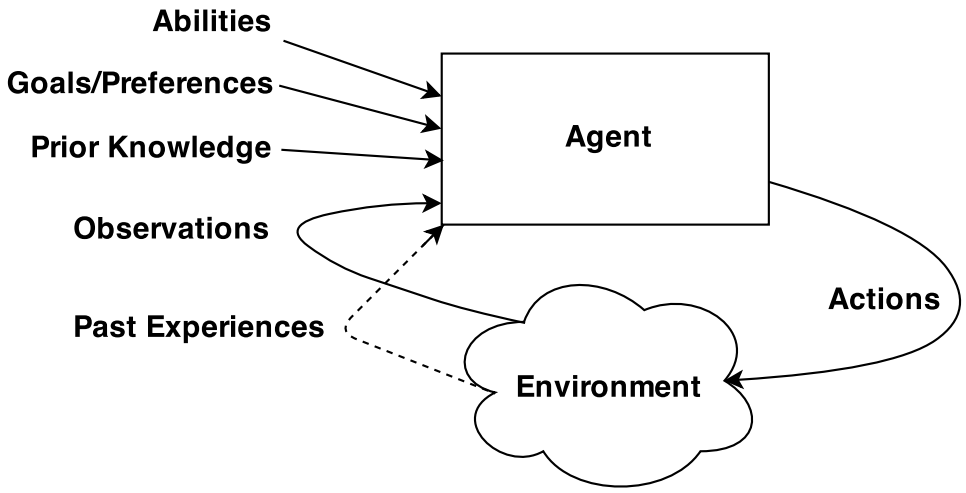


Figure 3.1: Agents Situated in Environments [Poole & Mackworth, 2010, p. 11].

Figure 3.1 shows the inputs and outputs of an agent. An agent can in JAWA's case be a computational engine with physical sensors and actuators and the environment is the physical setting. During the execution the agent depends on certain aspects as detailed in Figure 3.1.

Prior Knowledge — Refers to what the agent knows about the environment and what it knows about itself. For JAWA this is the knowledge of where the different areas in the warehouse are, such as conveyor-belts and drop-off zones. JAWA will only begin if it is known that there is a pallet in the warehouse ready for pickup.

Observations — Are the knowledge that the agent is currently obtaining from the environment. In the requirements it was stated that JAWA should be able to avoid collisions. This should be done by using the ultrasonic sensor to determine the distance between JAWA and obstacles.

Past Experiences — Past experiences of previous observations or actions that the agent can learn from. For JAWA this is triggered when a obstacle has been met as JAWA will avoid that exact obstacle in the future.

Goals/preferences — The goal is the main objective that JAWA has to try and achieve. For JAWA this is moving pallets from a pick-up zone to a drop-off zone, both located in the warehouse.

Abilities — The abilities are the primitive actions that JAWA is capable of doing. The requirements stated that JAWA should be able to move pallets, traverse the warehouse and identify pallets. This should be done with the actuators and sensors such as the ultrasonic sensor and colour sensors.

3.2 Mechanical Design of JAWA

This section deals with the design decisions, and the compromises that had to be made in conjunction with the construction of JAWA. In regard to the choices of sensors, motors, and the positions of these.

3.2.1 The Design of JAWA

In Figure 3.2 the mechanical design of JAWA can be seen. The figure contains numbers with the name of the different parts to give an overview of where the different parts are placed. More pictures of the final design of JAWA can be seen in Appendix B.

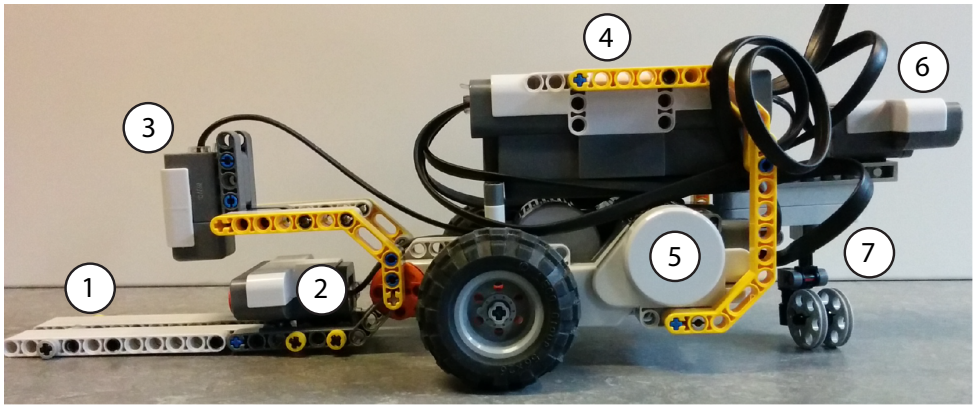


Figure 3.2: A picture of JAWA taken from the side.

1. Pallet collector
2. Ultrasonic Sensor for detecting pallets
3. Colour sensor for detecting the colour of pallets
4. LEGO® NXT Unit
5. Motors for wheels and pallet collector
6. Ultrasonic Sensor for detecting obstacles
7. Castor wheel

The thought process behind the placements of the different parts will be further explained in the following sections.

3.2.2 Motors

There are a number of tasks that requires the interaction of a motor to succeed. The motors of JAWA must be able to:

- Drive in a straight line as precise as possible.
- Turn 90 degrees allowing for navigating the map.
- Pick up a pallet using a lifting mechanism.

As mentioned in Section 2.1, these tasks must be solved with a maximum of three motors from the base LEGO® NXT unit.
If JAWA needs to drive and make turns, it needs at least two motors. This leaves one motor port in the LEGO® NXT unit for other tasks, further decisions will be explained in the following sections.

Steering

It is possible to make JAWA able to steer in different ways, but they all require at least two motors to be precise and work correctly. Considering the requirements of JAWA it is known that great precision is a necessity, the simplest and most precise way of doing this is with two motorised wheels parallel to each other as seen in Figure B.4 on page 72.
The NXT unit is constructed in such a way, that two wheels easily can be mounted parallel to each other on the NXT unit. Having only two wheels on JAWA poses a new problem: balancing. To eliminate this problem a castor wheel was mounted on the front of JAWA. Though there might be extra resistance, a triangle shape of the wheels and the castor wheel creates balance. By doing this it allows for only two motor ports to be used in the NXT unit and have a fully functional steering mechanism.

Collecting objects

The lifting mechanism on JAWA is build up of two separate arms with a small space in between to compensate for the normal construction of an EUR-pallet.
The lifting mechanism lifts the pallet by tipping them towards JAWA. The motors have a locking mechanism that keeps the motor in place for the rotation specified, even when weight is applied.
Two ways of placing the motor for the lifting mechanism was possible, but only one was considered, since the castor wheel takes up too much space. This means that the motor is placed at the bottom of the chassis in between the other two motors. It is placed outwards so that the mechanism can be extended enough to grab a pallet.

3.2.3 Sensors

Three sensors are used for JAWA and like the motors of JAWA, the sensors also have some specification that they must satisfy. The sensors of JAWA must be able to:

- Determine the distance between the robot and obstacles.
- Determine the distance between the robot and pallets.
- Determine the colour of a pallet.

As mentioned in Section 2.1, the LEGO® NXT unit allows for four sensors to be attached, which is more than enough to satisfy these specifications. The challenge is the positioning of the sensors; further decisions will be explained in the following sections.

Determining distance

To satisfy the specification of determining the distance to a pallet, an ultrasonic sensor facing the pallet from the lifting mechanism is needed. This poses a problem, because there is no room to place the sensor in the direction of the pallet without placing it on the lifting mechanism itself, the pallet will obscure the view whenever JAWA is to pick up a pallet. To avoid colliding with objects or obstacles, another ultra sonic sensor was needed because whenever a pallet is picked up, the first ultra sonic sensor is blind. The second sensor needs to determine the distance to objects or detect obstacles, but it can not be placed on the same side as the first sensor. Therefore the second sensor has to be placed on the other end of the chassis, this complicates the driving of JAWA because it will have to drive backwards to detect objects. This means that JAWA must spin 180° before being able to pick up a pallet.
The placement of both sensors can be seen in Figure 3.2.

Determining colours

JAWA needs to be able to determine the colour of a pallet to deliver it to the correct place. To do this a colour sensor needs to be mounted on the lifting mechanism without blocking the ultrasonic sensor. As seen in Section 2.4.2 on page 14, the colour sensor has the best accuracy at 1 cm from the colour to be determined.
To compliment with this, the lifting mechanism was extended in such a way that the colour sensor could be placed above the ultrasonic sensor, with approximately 1 cm between the top of the pallet and the sensor. This allows JAWA to determine the colour of a pallet, before or after it is picked up. The placement of the colour sensor can be seen in Figure 3.2.

3.2.4 Designing the Environment

The environment that JAWA will traverse is designed to look like the warehouse in the case scenario specified in Section 1.1. A scaled prototype of the environment will be made, with the same properties as Figure 1.2. By doing this JAWA can still be tested for functionality without having to build a full scale environment.

Limitations to the environment

Based on some of the tests performed in Section 2.4 and Section 2.5 some limitations to what JAWA should be able to do in the environment must be made. Due to the limitations of the ultrasonic sensor and the actuators, some actions cannot be accounted for. To ensure that JAWA can achieve its goals, the following list of limitations has been made:

- Objects in the environment must not be sound absorbing, such as a chalkboard eraser.
- No round objects must be present in the environment.

- There must be at least 15 *cm* between JAWAs ultrasonic sensors and an object in order for JAWA to react to it.

The first limitation is based on the test in Section 2.4.1, which showed that only hard surface objects, could be detected by the ultrasonic sensor. Therefore only hard surface and non-sound absorbing objects will be allowed in the environment. The second limitation is based on the fact that the ultrasonic sensor had issues with detecting round shaped objects such as LEGO® balls. The third limitation is based on the frequency test of the ultrasonic sensor, showing that it has problems detecting sudden objects placed in front of it. It takes time for the ultrasonic sensor to adjust the distance and in certain cases it can result in JAWA driving into walls, objects etc. It has therefore been deemed appropriate to place obstacles in the environment well ahead of JAWA.

3.3 Agent Design

In this system JAWA is an agent and the warehouse is its environment. As such, this can be modelled as a machine intelligence problem. These following sections will focus on the design of JAWA as an agent and the specification of the warehouse as viewed by JAWA. This includes a description of the dimensions of complexity, and a definition of JAWA's state space.

3.3.1 Dimensions of Complexity

In machine intelligence, the concept of designing an intelligent agent can be summarised into smaller categories, which can be considered separately, but ultimately has to be combined to create an agent. These categories are called the *dimensions of complexity*. It outlines different aspects that define the agent, such as how the agent chooses an action and how it predicts the future [Poole & Mackworth, 2010, p. 19]. Table 3.1 shows basic aspects of an agent, and helps create JAWA as an agent in this system.

Dimension	**Value**
Modularity	flat, modular, hierarchical
Representation Scheme	states, features, relations
Planning Horizon	non-planning, finite-, indefinite- infinite- stage
Sensing Uncertainty	fully-, partially observable
Effect Uncertainty	deterministic, stochastic
Preferences	goals, complex preferences
Learning	knowledge is given, knowledge is learned
Number of agents	single-, multi-agent
Computational limits	perfect rationality, bounded rationality

Table 3.1: A table detailing the various dimensions of complexity specifying the level of complexity for an agent.

The following is a list of the dimensions, with a short description and how JAWA will make use of them. These are based on [Poole & Mackworth, 2010, pp. 19–29] and follows Table 3.1 sequentially.

Modularity concerns the structure of the agent's reasoning. It comes in three different structures: flat, modular, and hierarchical. Flat modularity is used for systems with no

need for organisational structure, i.e. the whole system reasons as one. The modular approach divides the structure into small subtasks. A hierarchical structure recurses the modular approach to create modules within the modules that each have more defined tasks, until it grounds out into basic instructions.

JAWA: JAWA needs to be able to calculate a path, drive it, avoid obstacles and so forth. These can be seen as subtasks, and as such a flat modularity is not complex enough. However, it would be detrimental to recurse it further, and as such it would exclude a hierarchical modularity. A modular structure can therefore be used. For JAWA, this is constructed by having multiple smaller modules, i.e. jump point search, traversal, obstacle detection and so forth.

Representation scheme concerns the way the agent perceives its environment. It can be divided into three values; states, features, and relational. States are the raw representation of the world and can be useful to describe trivial environments with very few elements. Features are used when a system becomes more complex and there are too many states. A feature is a way to summarise states into larger groups. For instance, a specific coordinate can be a state, but instead of concerning about each individual coordinate it is easier to call it a location. That way every coordinate can be summed up as a location. The last approach is the relational representation scheme. This is when there are too many features to properly represent the environment. This is done by calling features individuals instead and have a relation between the feature and its respective value. For instance a light switch can have a feature that tells its position, $position\_s_1$ that can be either up or down. If there are many switches it would be easier to just reason with the relation between s_1 and its value, making a relation *position(s_1, up)*.

JAWA: Given the fact that being in position $(0, 4)$ and having the lift up is not the same state as being in the same position with the lift down, this results in an immense amount of states, excluding a raw state representation. Therefore, features could be location and lift, to describe the example above. However, there would not be enough features to justify a relational representation scheme, therefore a feature based representation scheme is used for JAWA.

Planning horizon concerns the rate at which an agent plans its actions. It comes in four different values; non-planning, finite -, indefinite -, and infinite stage. The non-planning agent does not plan its steps and does not take the future into account. If the tasks of the agent can be divided into different stages it uses a finite stage planner, where it knows exactly how many steps it needs to take to reach that goal. An indefinite stage has a goal but no defined number of steps necessary to reach the goal. An infinite stage has a goal but an infinite amount of steps to reach it, as such it keeps on going forever.

JAWA: In the case of JAWA the goal is known; pick up all the pallets and return them to the designated locations. Given the fact that the warehouse has unforeseen obstacles, the total amount of steps necessary for JAWA to achieve its goal is obscured. As such JAWA has an indefinite stage planning horizon.

Sensing uncertainty concerns the actual sensing of the agent's surroundings, and whether it is complete or partial. Sensing uncertainty can have the following values; fully observable or partially observable. If an environment is fully observable an agent has complete knowledge of all states. However, if it is only partially observable it means that there is something that limits, or obscures the agent's perception of the environment.

JAWA: Since the warehouse can contain unforeseen obstacles, and the readings of the sensors are not perfect and can give error readings, the environment for JAWA is therefore

only partially observable.

Effect uncertainty concerns whether the agent knows the effect of the action it is taking. This can have two values; deterministic or stochastic. An effect is deterministic if the agent knows exactly what the state will be after taking an action. It is stochastic if it is not completely certain how the action affects the state. l **JAWA:** Because sensing uncertainty is partially observable, it does not make sense to model this dimension according to [Poole & Mackworth, 2010, p. 24].

Preference concerns how the agent determines whether a goal has been reached in the best way. There are two ways of modelling this; goals or complex preferences. A goal based preference can then have two additional categories; achievement goals or maintenance goals. The achievement goal, when reached, is when the agent has fulfilled its task completely. A maintenance goal is, where the goal is to reach and maintain a goal. When a preference is complex it is usually divided into two parts; ordinal- or cardinal preferences. An ordinal preference can be seen as a general preference, for instance $a > b > c$ where it is more preferable to choose a. A cardinal is a set of more specific preferences. For example, an ordinal preference could be; $cappuccino > coffee > tea$ where cappuccino is more preferable with higher reward, but making cappuccino takes a longer time. This makes cappuccino the better decision when the agent has time, while making coffee and tea is better when the beverage needs to be served immediately.

JAWA: In terms of the path finding algorithm for JAWA, an ordinal preference would be to make the shortest path in the warehouse. A cardinal preference, however, could be to make as few turns as possible. However, the overall goal of JAWA would be to pick up pallets and return them to the designated locations, i.e. an achievement goal.
As such, the system as a whole has an achievement goal, but the algorithm has complex preferences.

Learning concerns the knowledge of the agent. There are two different ways of learning; knowledge is given or knowledge is learned. When knowledge is given the agent knows the whole environment. When knowledge is learned the model of the environment is not complete and the agent needs to learn the missing parts of the model.

JAWA: Given the fact that the environment is directly modelled in a graph, the knowledge is given to JAWA. However, obstacles are discovered and added to the map, thus updating the model, this is only for the specific session, and is reset for the next run. Hence, there is no retention of knowledge.

Number of agents concerns whether the agent has to take other agents into account when reasoning. There exists two domains in this dimension; single agent, and multiple agents. In a single agent system the agent only takes its own reasoning into account, and assumes that other agents are just part of the environment. In a multiple agent system the agent needs to know how the other agents reason, and base its reasoning on that.

JAWA: A multi-agent system is not necessary to model a solution for the given problem statement.

Computational limits concerns whether the agent needs to take its computational limits into account when choosing an action. In computational limits there are two values; perfect rationality, and bounded rationality. If the agent has perfect rationality it always wants to find the optimal solution to a problem, regardless of computation time.

With bounded rationality it only concerns itself with finding a satisfying solution to the problem.

JAWA: based on the hardware limitations mentioned in Section 2.1 it would seem that bounded rationality was the best choice. However, the environment is rather limited and it is not a priority that it drives immediately after receiving an instruction. So based on that JAWA will use perfect rationality.

To summarise, the dimensions of complexity for JAWA is as follows:

- **Modularity:** Modular.
- **Representation Scheme:** Features.
- **Planning Horizon:** Indefinite Stage.
- **Sensing Uncertainty:** Partially Observable.
- **Effect Uncertainty:** Irrelevant.
- **Preferences:** Overall system: Achievement Goal. Pathfinding algorithm: Complex Preference.
- **Learning:** Knowledge is Given.
- **Number of Agents:** Single Agent.
- **Computational Limits:** Perfect Rationality.

As seen, the representation scheme for JAWA is a feature based system. To further detail the environment that JAWA needs to be in a state space definition is needed.

3.3.2 State Space

A state space is a definition of the entire environment and what the agent needs to reason with. As mentioned above the states of JAWA is summed up in features, therefore section will define the various features and their respective values that JAWA needs to perceive its environment. Furthermore, the available actions, that transitions JAWA from one state to another, is also defined later in this section. A more formal definition of a state space can be seen below.

> A state-space problem consists of:
>
> - A set of states;
> - A distinguished set of states called the start states;
> - A set of actions available to the agent in each state;
> - An action function that, given a state and an action, returns a new state;
> - A criterion that specifies the quality of an acceptable solution. For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost. This is called an optimal solution. Alternatively, it may be satisfied with any solution that is within 10% of optimal.
>
> *Source:* [Poole & Mackworth, 2010, p. 74]

State Space of Environment

The features of the environment can be divided into features specific for JAWA and features specific for the warehouse. First the features describing JAWA.

- **Colour Sensor:** BluePallet, RedPallet, GreenPallet, Nothing.
- **Lifting Mechanism:** Up, Down.
- **Pallet Acquired:** True, False.
- **Direction:** North, South, East, West
- **Robot location:** coordinates(x-axis, y-axis).
- **Front Ultrasonic Sensor:** Obstacle, NoObstacle
- **Back Ultrasonic Sensor:** Pallet, NoPallet

The features describing the warehouse are as follows:

- **Pallets in environment:** 0,1,2....

Most of these features are self explanatory, such as where JAWA is and its direction. However, the colour sensor has a value called *Nothing*, which means that until the sensor reads a pallet there is no colour.

Start / End States

In a feature based system a state is a tuple of all features. A start state, or initial state, is the state where the agents perception of the environment is the most limited. As the agent moves through the environment it expands this perception and gains a wider understanding of the environment. An end state, or goal state, is the state where the agent has fulfilled its goal. A start state in this instance would be; *(Nothing, Down, False, South, (0,0), NoObstacle, NoPallet, 1)*. This would be that JAWA initially detects no colours, the lifting mechanism is down and there is no pallet on it. JAWA faces south in location *(0,0)*, the front ultrasonic sensor detects no obstacles and the back doesn't detect a pallet. JAWA knows that there is at least one pallet in the environment.
An end state for JAWA would be for instance; *(Nothing, Down, False, South, (0,0), NoObstacle, NoPallet, 0)*. This looks almost identical to the start state, however, the feature *Pallets in environment* is set to 0. This means that JAWA returns to its start location after retrieving the pallet and placing it at the designated conveyor belt.

Actions

To transition from a state to another JAWA needs a set of actions available in every state it transitions to. The actions available are:

- *RotateLeft*, *RotateRight*.
- *DriveForward*, *DriveBackwards*.
- *LiftingUp*, *LiftingDown*.
- *MeasureColour*.

- *DetectDistance*

RotateLeft and *RotateRight* can be used in any given state and change the feature *Direction*. *DriveForward* and *DriveBackwards* can only be called, if it does not bring JAWA out of bounds. These change the *Coordinates* based on the *Direction*.
A state with the lifting mechanism up cannot perform the action *LiftingUp*, and vice versa. These changes the state of the *Lifting Mechanism*. *LiftingDown* also sets the *Pallet Acquired* to *False*, while *LiftingUp* sets it to *True* if *Measure Colour* does not return *Nothing*.
If *MeasureColour* returns Nothing, *Pallet Acquired* is *False*. If *MeasureColour* returns a colour, the *Lifting Mechanism* must be set to up and *Pallet Acquired* must be *True*. This changes the state of the *Colour Sensor* to the measured colour. When the pallet is picked up the feature *Pallets in environment* is decreased by one.
If *DetectDistance* returns a distance to *Back Ultrasonic Sensor*, and is expecting a pallet, the value changes from *NoPallet* to *Pallet*. If a distance is returned to *Front Ultrasonic Sensor*, the value becomes *Obstacle*. The action to go to this coordinate from that point is then removed as the robot cannot access this location.

3.4 Software Design

The following sections will describe the design of the architecture, which will make it possible for JAWA to achieve its goals. Here, the models for the architecture supporting JAWA will be explained. Also an overview of the system will be made by presenting an abstract class diagram. Furthermore, a sequence diagram of the expected program flow is presented. This will provide a sturcture for the implementation of JAWA.

3.4.1 Software Architecture

For the overall software architecture, the model FroboMind is used. FroboMind was developed to create a standard architecture for developing autonomous robots. Though the actual model was developed for robots in agriculture, it can be used for JAWA as well. The model is documented in [Jensen et al., 2014].

In this section this model will be described. The descriptions are largely based on the source, where applicable. However, the source does not provide comprehensive descriptions of all the parts in the model. Thus, some of the descriptions are based on the definitions of the words, in a software context, and some conjecture.

External

External components describe how knowledge and tasks are given to the agent. The external knowledge is used for Localization & Mapping as well as the Mission Planner.
Mission Description is a statement of the purpose of an agent, and spells out its overall goal and guides decision-making. The Mission Description contains start- and end-state. For JAWA this is the tasks it needs to fulfil for the current session.
Shared Knowledge shares the knowledge between entities in the environment. JAWA does not share knowledge, since it is the only entity that requires knowledge.
A priori Knowledge is the knowledge that is known to be true, requiring no evidence for its validation or support. In the case of JAWA, these are things such as knowledge of the map, where the walls, pick-up and drop-off zones are placed, as explained in Section 3.3.2.

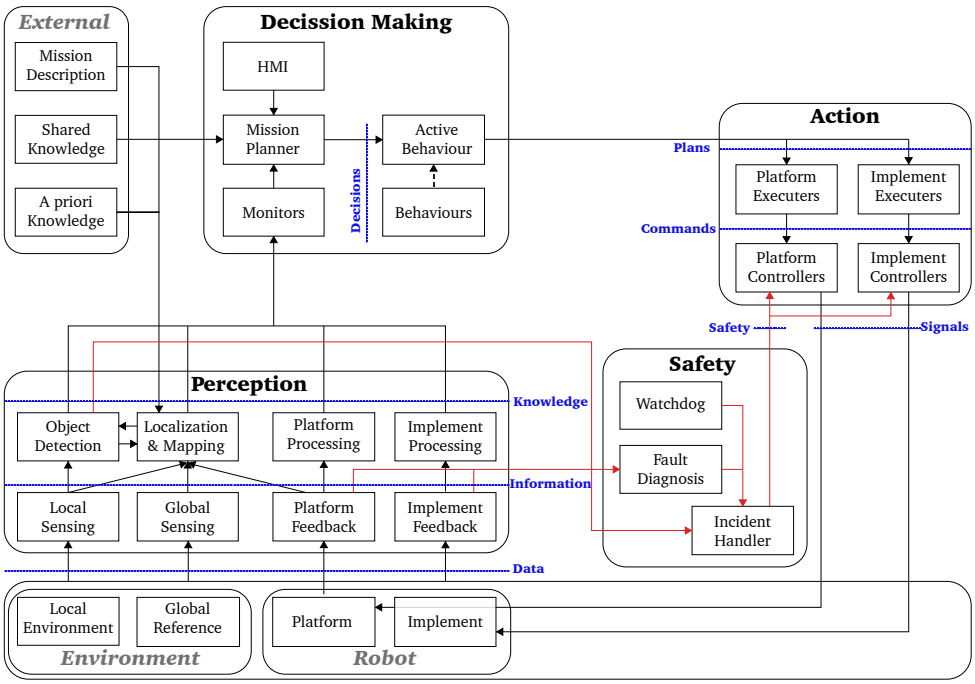


Figure 3.3: FroboMind Architecture.

Decision Making

The process of selecting a logical choice from the available options. These decisions are made based on the Externals from the previous section as well as the Perception. The Active Behaviour is then taken to the Action centre where it later will be processed.
HMI(Human Machine Interface) displays information to humans through, for instance, a GUI. As JAWA does not interact with humans other than avoiding colliding with them, this becomes irrelevant.
Mission Planner is a list of waypoints, events and tasks required to be completed. This is created from the Monitors and the Externals. The planner for JAWA is the amount of actions required to reach the end-state, specified in Section 3.3.2.
Monitors observe and keep systematic review of the different forms of knowledge acquired from Perception. This includes knowledge from Object Detection, Localization & Mapping, Platform Processing and Implement Processing. This information is used for the Mission Planner. Monitors contain all data about the state of the agent, and would be comparable to the sensors and the map representation in JAWA.
Behaviours contains a list of available actions to the agent. From these, an action can be chosen by the Mission Planner and be set active. The active behaviour is based on the highest priority of the behaviours in the Mission Planner. For JAWA the behaviours in the Mission Planner will be the actions mentioned in Section 3.3.2.
Active Behaviour is the behaviour currently in execution. This behaviour could, in JAWA's instance, be collecting a pallet.

Action

The Active Behaviour is broken into smaller components, called plans. Plans from the Active Behaviour are converted into platform and implement commands. These commands, together with commands from the Incident Handler, are converted into signals for the robot. Here the two categories, Platform and Implement, are very similar.

However, Platform pertains closer to the software, i.e. at a higher level, while Implement refers more to the hardware level. This is more or less the same for all agents.
Platform Executer puts the Active Behaviour into effect. The Platform includes the hardware architecture, the firmware, and runtime libraries. This could be compared to the use of leJOS for JAWA.
Implement Executers puts the active behaviour into effect. The Implement includes all actuators.
Platform Controllers sends signals to the robot to update the robots state, for instance, play motor failure tone in a frequency of 700 Hz. The controllers takes commands from both the Executors and the Incident Handler from the Safety group.
Implement Controllers sends signals to the actuators to update the robot state, for instance, stop motor A.

Robot

The actual hardware of the robot. Receives signals from Action and acts in the environment. The actions can later be measured in the Perception part of the robot.
Platform receives signals from the Platform Controllers, altering the state of the platform.
Implement receives signals from the Implemented Controllers, altering the state of the actuators.

Environment

The environment is the surroundings in which the robot operates. Information of the environment is gained through Local and Global Sensing, referring to the knowledge gained from the sensors of JAWA as well as the information from the map.
Local Environment contains knowledge available to the sensors; what the sensors detects in any given state. The Local Environment refers to what JAWA knows about the environment in its current position, such as an obstacle being present.
Global Reference is the knowledge of the map, along with the current knowledge obtained from the current session. For instance, JAWA should know if it had passed an obstacle, and if it must navigate past some walls to reach the goal, but it does not for certain know if a pallet is ready for pickup.

Perception

The robot perceives its surroundings by the use of sensors, or by referencing a map of the environment or a combination of both. Perception takes input from Robot, Environment and Externals then presents that information and knowledge to the Monitors in the Decision Making, and the Fault Diagnosis and Incident Handler of the Safety.
Object Detection takes input from Local Sensing and uses this data in both the Incident Handler and Monitors. The object detection is used for the ultrasonic sensor on JAWA, sensing local obstacles and using that data for preventing collisions. Object Detection sends and receives data from Localization & Mapping.
Localization & Mapping takes input from the Local and Global sensing, as well as Platform Feedback to create a map and update the position for JAWA. This knowledge is tracked by the Monitors.
Platform Processing acquires information from the Platform Feedback and makes the knowledge accessible to the Monitors.
Implement Processing acquires information from the Implement Feedback and makes the knowledge accessible to the Monitors.

Local Sensing takes input from the Local Environment with the use of sensors. This information is used by the Object Detection. Local sensing is also used by the Localization & Mapping. The Local Sensing is performed by the sensors attached to JAWA.
Global Sensing takes data from Global References and passes the information to Localization & Mapping. When JAWA computes a route, it retrieves information from Global References and navigates around walls.
Platform Feedback is measured data from the agent's runtime. The agent updates the location of itself and informs the Localization & Mapping. This information is also evaluated in the Fault Diagnosis, which will evaluate whether the agent has performed a legal action. For instance, JAWA cannot drive outside of the map.
Implement Feedback is the measured data from the agent's runtime, this is sent to Implement Processing. The data is supervised by Fault Diagnosis. For JAWA this could be a wheel being blocked.

Safety

To reduce the risk of failing or crashing before reaching a certain goal, some safety mechanisms are required. Safety takes input from the Perception, based on both the Robot and the Environment. It then outputs to the Action with commands for the controllers.
Watchdog is a device used for observing, checking, and keeping a continuous record of something. For the agent, it keeps track of the current state. This can be used to detect errors in the system, such as low battery, and send signals to the Incident Handler to resolve problems. For JAWA, this is already implemented in leJOS.
Fault Diagnosis takes input from the Feedback in the Perception part. If a wheel is blocked and the tachometer does not increase after rotate is called, the Fault Diagnosis will send a signal to the Incident Handler to solve this. JAWA uses Fault Diagnosis, when it is expecting a blue pallet, but sees a green one, which it perceives as a fault.
Incident Handler will receive necessary data from Object Detection, Watchdog and Fault Diagnosis with problems that can bring the agent into a state that is problematic. The Incident Handler then sends an overriding signal to the controllers to resolve the incidents. If the example mentioned in Fault Diagnosis occurs, JAWA will drive back to the starting point.

3.4.2 Class Structure

With the ideas from the FroboMind architecture, an early structure of the problem domain has been created in the form of an abstract class diagram, which can be seen in Figure 3.4. The `Warehouse` class generates the map that JAWA needs to follow. It contains a method to create the map and one to represent an obstacle in it, if such one is found. This is related to the *External* and *Localization & Mapping* parts of the FroboMind architecture. The `JumpPointSearch` class contains the algorithm, which finds the shortest path through the map. This is used together with `Warehouse` and pertains to the same categories in FroboMind. The system has an event `AvoidObstacle`, which is raised when the sensors detect an obstacle, and this would correspond to the *Perception* part of the architecture. The `Driver` class provides methods, for controlling the actuators, such as driving forward, backward and stopping. The `Driver` class also controls the lifting mechanism. This would be the *Action* category in FroboMind. Finally the main method of the system is the `Pilot` class, which would be equivalent to the *Decision Making* category in FroboMind. This class uses the path computed by the Jump point search algorithm and the methods from `Driver` to physically follow this path, and

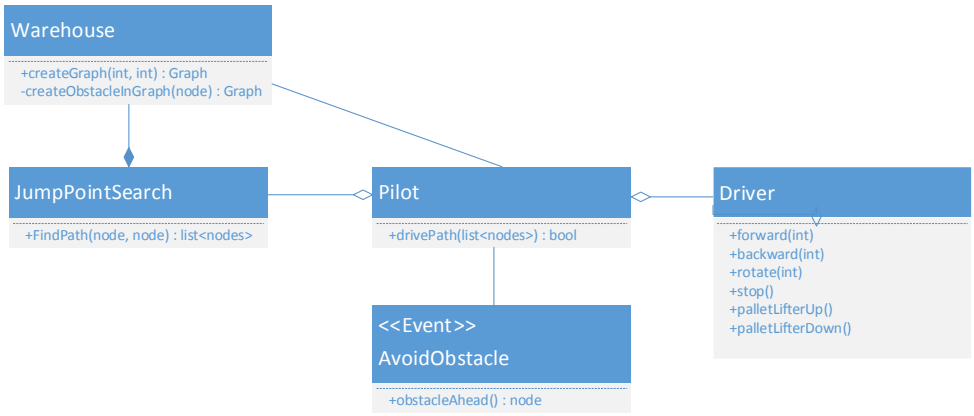


Figure 3.4: Abstract class diagram.

while it drives it checks if there is an obstacle, and raises the event if there is. If an obstacle is detected a call is made to `Warehouse` to add that obstacle in the map.

3.5 State Diagram

To represent the states that JAWA can be in and the execution cycle for it, a state diagram has been made in Figure 3.5.

For JAWA to operate properly, the order in which certain actions are executed must be detailed. At certain points during execution, JAWA must know, what the next action is, to make sure that the goal is reached. This includes when to compute a new route, pick up a pallet, identify the colour of the pallet or avoid an obstacle. The state diagram describes the behaviour of the system and the different states it can be in. This follows up on Figure 1.1, which presents an example of an environment that JAWA could operate in.

The action sequence of JAWA will be detailed in the following, it is implied that each time JAWA drives a check for obstacles are made:

1. Use the jump point search algorithm to get to the pickup zone
2. Scan the pallet using the colour sensor and determine the colour
3. Use the jump point search algorithm to get to the correct dropoff zone
4. If an obstacle is detected using the ultrasonic sensor then compute a new route
5. When at the correct dropoff zone, place the pallet and compute a route back to the pickup zone.
6. If a pallet is available, then deliver it to the dropoff zone, if not, then return to the starting position.

Points two through six will continue to execute until no pallet is detected at the drop-off zone, and JAWA returns to the starting position where it will wait for a new signal.

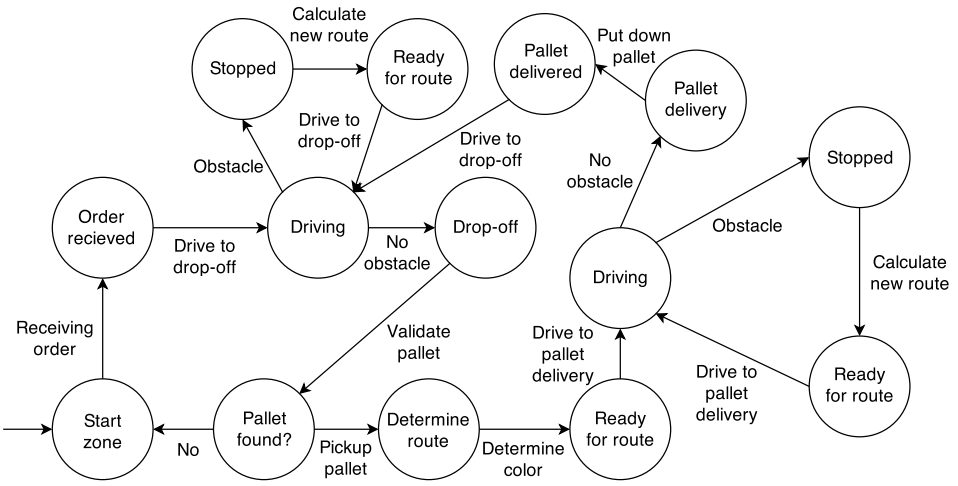


Figure 3.5: State diagram.

3.6 Sequence Diagram

Figure 3.6 represents the sequence diagram from when the operator tells JAWA to retrieve a pallet of a specific colour to the actual retrieval process throughout the system. The necessary objects such as `Driver`, `Warehouse` and `Pilot` are instantiated. The ultrasonic sensors are instantiated in the `ObstacleDetect` class. This is an event with an eventlistener, which executes if an object is detected within a danger-zone. If an object is detected, the `Pilot` uses `JumpPointSearch` to find a new path and continues to drive with the new path instead of the old. If no obstacles were detected, JAWA drives the most direct route with as few turns as possible. Eventually, in the best case scenario, JAWA will reach the pallet. `Driver` is opted to pick up the pallet. It checks whether the pallet has the colour that is wanted or not and takes corresponding action of picking it up or leaving it. `Driver` signals the `Pilot` that it is ready to continue. `Pilot` gets a path from `JumpPointSearch` and drives to its new destination. As before if any obstacles are detected a new path is calculated by `JumpPointSearch`. When the destination is reached the `Driver` sets down the pallet if the fork is lifted. The operator gets a signal that the job is done.

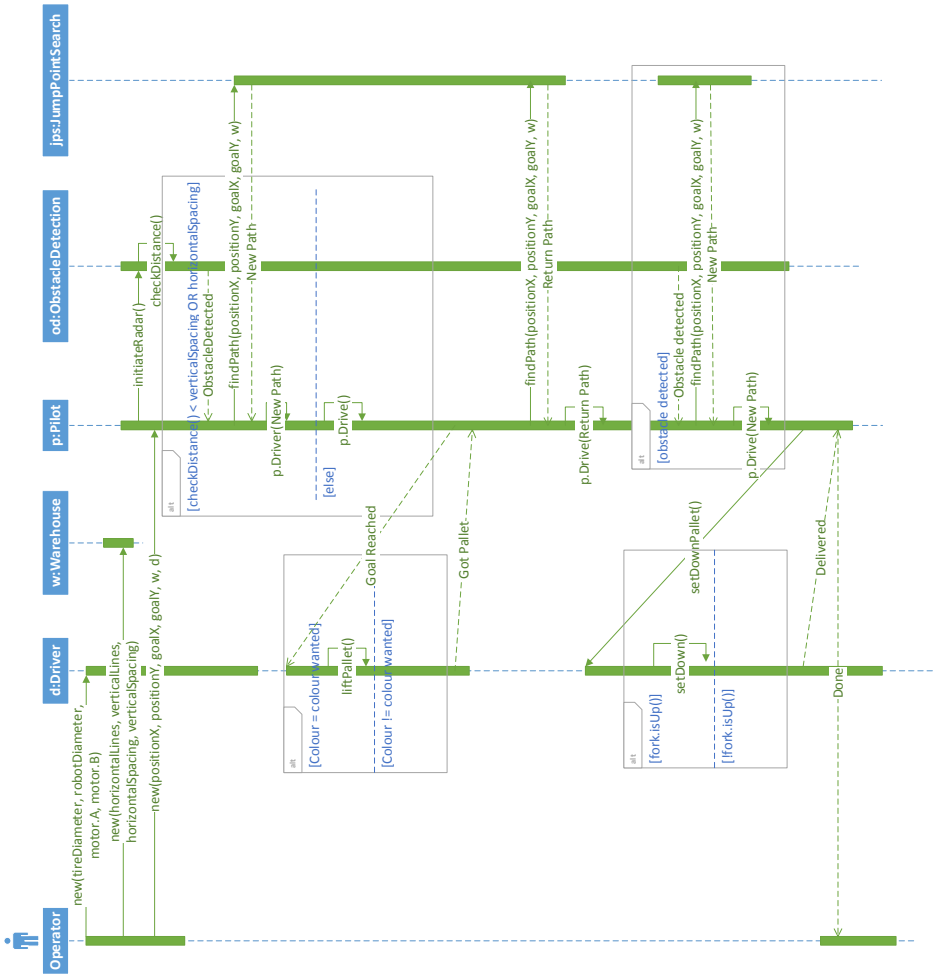


Figure 3.6: Sequence Diagram.

4 | Implementation

This chapter describes the implementation of JAWA. This includes how JAWA is programmed to traverse, the environment and how the map is implemented. The approaches chosen will be explained, and code examples will be presented. As an embedded system, both the hardware and software needs to work together. The implementation is important to make sure, that the software aspect of JAWA is working.

4.1 Map Implementation

This section describes the implementation, of the map for JAWA. This consists of the `Warehouse` class. For the implementation of the Warehouse a number of class structures are provided by leJOS. These include `Line`, `Node` and `LineMap`. To create a representation of a graph, a `LineMap` is build by `Lines`, corresponding to the edges of a graph. The vertices are represented by `Node`s, which are instantiated with the type `float`, for the x- and y-coordinates. When creating a graph using a `LineMap`, a height and width must be specified. This is used to create a `Rectangle`, which contains the `Line`s. The `LineMap` can be seen as a coordinate system, where length and height correspond to the x- and y-axis respectively.

4.1.1 Warehouse Class

The Warehouse class, handles creating a graph, with a specified number of lines, `float horizontalLines` and `float verticalLines` in Listing 4.1, for either axis, and the length of these lines, `float xLineLength` and `float yLineLength`.

```
1  public Warehouse(float horizontalLines, float verticalLines, float
       ↪ xLineLength, float yLineLength) {
```

Listing 4.1: The constructor for the Warehouse class.

The height and length of the map is computed, by using the number of lines on the axis multiplied by the amount of lines on that specific axis, for instance `float horizontalLines` *times* `float xLineLength`.

The most used methods are the `linesForNodeExists` and `lineBetweenNodesExists` methods. They can be seen in Listing 4.2 and Listing 4.4 respectively and was made by the authors of this report.
`linesForNodeExists` checks whether the specified node, has any lines going to or from it. The node is specified by the x- and y-coordinates, denoted by the two `float` input parameters. In line 2 a foreach loop iterates through the `ArrayList` of horizontal lines. In line 3, it checks if there exists a line in the list, that either has the specified node as starting point or end point, and if so it returns `true`. In Listing 4.2 the check is only for the horizontal directions, the check for the vertical directions are however identical.

`lineBetweenNodesExists` has some of the same functionality as `linesForNodeExists`, but instead of checking whether a specified node is start or end node for a line, it checks

whether a line exists between two specified nodes. At line 1 in Listing 4.4 it can be seen, that the methods takes four `float` parameters, these denotes the start and end, x- and y-coordinates of the two nodes. On line 3 it is determined, whether the line is horizontal or not, by iterating through the `ArrayList` of horizontal lines. On line 5 a if statement check is necessary, because a node can be both start and end node of a line, therefore it is determined which is the case. The last if check on line 6 determines whether a line between the two nodes exist by checking against the start and end, x- and y-coordinates of the line and returns `true` if it is the case.
In Listing 4.4, only the code for checking horizontal lines are shown because the only difference is the `ArrayList` that is used.

In Listing 4.3 and Listing 4.5 the public versions of the above mentioned methods can be seen. These have been created because the naming convention provides a better understanding of their use in accordance to a graph representation. They call their respective private variants, and returns what they return. These methods are used excessively in the Jump Point Search algorithm to determine whether a jumppoint has been reached.

```
1  private boolean linesForNodeExists(float x, float y){
2    for(Line line : _listOfLinesHorizontal){
3      if(x == line.x1 && y == line.y1 || x == line.x2 && y == line.y2){
4        return true;
5      }
6    }
```

Listing 4.2: A method that returns `true` if the specified `Line` exists in the map.

```
1  public boolean isWalkableAt(float x, float y) {
2    if (linesForNodeExists(x, y))
3      return true; // some lines to or from the node still exist.
4    else
5      return false; // all lines to and from the nodes is removed.
6  }
```

Listing 4.3: A method that checks whether the specified `Node` is accessible from at least one direction.

```
1  private boolean lineBetweenNodesExists(float x1, float y1, float x2, float
       ↪ y2){
2    // must check for all lines in both horizontal and vertical if both x's and
         ↪ y's is equal.
3    if(y1 == y2){
4      for(Line l : _listOfLinesHorizontal){
5        if(x2 < x1){ // must have this check because lines starts from lowest x
             ↪ to highest.
6          if(l.x1 == x2 && l.y1 == y2 && l.x2 == x1 && l.y2 == y1) // switch
               ↪ start point to be (x2, y2)
7            return true;
8        }
9        else{
10         if(l.x1 == x1 && l.y1 == y1 && l.x2 == x2 && l.y2 == y2)
11           return true;
12       }
13     }
14   }
```

Listing 4.4: A method that checks whether there exists a `Line` between two `Node`s.

```
1  public boolean isWalkableBetween(float x1, float y1, float x2, float y2){
2    if(lineBetweenNodesExists(x1, y1, x2, y2))
3      return true;
4    else
5      return false;
6  }
```

Listing 4.5: A method that checks whether it is possible to traverse from between two specified `Node`s.

4.2 Robot Implementation

In this section the various aspects of the implementation of JAWA are described. This includes the movement, pathfinding and evading obstacles.

4.2.1 Movement

The Lejos NXT API provides a class called `DifferentialPilot` which contains methods to control the movements of a robot with the wheels parallel to each other, which is how JAWA is designed. The class proved to be insufficient since it did not rotate JAWA by the specified amount. As such, a new class called `Driver`, was created to immitate the functinonality of the `DifferentialPilot`.
The `Driver` class is based on the `DifferentialPilot` class, but is not extended from it. Instead it only provides methods which are relevant and complements the way in which JAWA must traverse the map. The `Driver` class has methods that can make JAWA drive forward, backward and rotate around its own axis. The calculations of how much either motor should rotate for one wheel to reach 360° is based on the simple fraction: $\frac{\text{wheel diameter} \cdot \pi}{360} = \text{Degrees for one rotation.}$
The `Driver` also provides methods to stop the motors, set the rotation and acceleration speed, wait until all movement operations are done and a method to check whether the motors are in motion. `stop`, `setSpeed`, `setAcceleration`, `waitComplete` and `isMoving` methods respectively. These are some examples of the methods thay has been implemented in the `Driver` class.

For driving, the method `drive` is provided and takes, a distance and a **`boolean`** `immediateReturn`. This method has an overloaded counterpart, without the `immediateReturn` parameter, which calls the other method with the parameter set to **`false`**. `immediateReturn` determines whether it should be possible to perform other actions while the robot is driving forwards or backwards. When true, it is possible to execute code for the ultrasonic sensor to check for obstacles in the environment.

4.2.2 Pathfinding

To have JAWA traverse the map, the orthogonal Jump Point Search is used to create a shortest path from a start-node to a goal-node. The most important method in `JumpPointSearch` is the `jump` method.
In Listing 4.6 the recursive `Jump` method can be seen. This method returns a node when a forced neighbour of this node is found in the map or null if no such node exists. Line 7 is a check to see if a node is accessible or not and returns **`null`** if the node is outside of the grid, inside a wall or if the current node is not reachable from the parent node. Line

10 checks if the current node reached is the goal and returns if that is the case.
Lines 13–23 is the actual jump point check. This particular one is for the horizontal direction, i.e. moving along the x-axis. The actual jumppoint check uses the `isWalkableBetween` method to determine forced neighbours. If one is found the current node must be returned as a jumppoint.
If there are no jumppoints the method is called recursively, to move one step down the y-axis.

```
1   private MyNode jump(float x, float y, float px, float py){
2     // gets the distance between the node and its parent.
3     LogJava.writeToFile("in jump", System.currentTimeMillis());
4     float dx = (x - px);
5     float dy = (y - py);
6
7     if(!_grid.isWalkableAt(x, y) || !_grid.isWalkableBetween(x, y, px, py)) //
          ↪ if the grid isn't walkable at the current node, and if it is not
          ↪ possible to go there from the parent, return null
8       return null;
9
10    if(_grid.getNode(x, y) == this._goalNode) // if goal node is reached return
          ↪ it.
11      return _grid.getNode(x, y);
12
13    if(dx != 0){ // searching along x-axis.
14      // This checks whether it is possible to go either up or down in the
            ↪ graph, and that it is not possible for our parent to go up or down
15            //Furthermore if our parent is able to go up or down, we check
                  ↪ whether it is possible to go from there to the node above
                  ↪ or below the current node, respectively
16      if(((_grid.isWalkableBetween(x, y, x, y + _dy) &&
            ↪ !_grid.isWalkableBetween(px, py, px, py + _dy)) ||
17        (_grid.isWalkableBetween(x, y, x, y - _dy) &&
              ↪ !_grid.isWalkableBetween(px, py, px, py - _dy))) ||
18        (_grid.isWalkableBetween(x, y, x, y + _dy) &&
              ↪ _grid.isWalkableBetween(px, py, px, py + _dy) &&
19        !_grid.isWalkableBetween(px, py + _dy, x, y + _dy)) ||
              ↪ (_grid.isWalkableBetween(x, y, x, y - _dy) &&
20        _grid.isWalkableBetween(px, py, px, py + _dy) &&
              ↪ !_grid.isWalkableBetween(px, py - _dy, x, y - _dy))) {
21
22        return _grid.getNode(x, y);
23      }
```

Listing 4.6: The method responsible for finding jumppoints.

The rest of the class works as an A* algorithm, but instead of expanding on all neighbours, it uses the `jump` method to prune unnecessary neighbours. This means that when A* would normally identify the successors, `jump` is called instead.

4.3 Combined Map & Robot

This section will describe how the `Warehouse`, `Driver` and `JumpPointSearch` classes were combined to make a class called `Moves` which makes JAWA drive from jumppoint to jumppoint to reach a defined goalnode. The methods used in this class will be described in detail.

4.3.1 Moves

Moves provides three constructors. Listing 4.7 shows the base constructor that takes the most arguments.

```
1  public Moves(int direction, float startX, float startY, int palletColor,
       ↪ final UltrasonicSensor uss, Driver pilot, Lift lift, Warehouse w){
```

Listing 4.7: The base constructor of the `Moves` class.

One of the overloaded constructor omits `int direction` to facilitate a case where JAWA starts in default direction, south.
The last constructor omits both `int direction` and `int palletColor` to facilitate a case where JAWA starts in the default direction, south and is opted to pickup pallets from the dropoff zone. This is most commonly used constructor.

Directions are provided as integers with the value 0–3: $North = 0$, $East = 1$, $South = 2$ and $West = 3$. The default direction is South. If the `palletColour` is defined it determines that JAWA should pick up pallets from the conveyerbelt as opposed to the dropoff zone. The default value of `palletColour` is 7, the integer value for black and no colour.

After moves has been instantiated in the `Pilot` class, calling `traverse` becomes available. This method, rotates JAWA towards the direction of the first jumppoint and starts traversing trough an array of jumppoints. Whenever a jumppoint is reached, a variable containing the current position of JAWA is updated. When reaching the goal, whether it is the conveyorbelt or pallet dropoff zone, JAWA will change direction such that the lifting mechanism is facing the pallet. Then JAWA uses the ultrasonic sensor to detect if there is a pallet. If there is no pallet, JAWA will find a new path back to the starting point and terminate. If there is a pallet, JAWA backs up into it and picks it up, moving it to the right location depending on whether the task is to move pallets from conveyor belts to the dropoff zone or vice visa.

Evading Collisions

For JAWA to navigate autonomously in a given environment it has to avoid colliding with objects or walls. This is implemented in the `travers` method in the `Moves` class. When traversing, the `drive` method is set with the parameter `immediateReturn` to true. This lets a while loop be executed, using a `isMoving()` method as a boolean check. As soon as JAWA reaches the destination it is driving towards, it will escape the while loop. Inside the loop, the ultrasonic sensors check continuously for the distance towards the nearest object. If the distance to the nearest object cross the threshold, the motors of JAWA will stop. This ends the while loop and sets the backtrack action in motion, starting the jump point search for the current location to the goal while breaking out of the current traversal.

4.4 Log File

A log file was created to serve as a debugging tool and to ascertain the program flow when JAWA is running. This file is static and is written to at key points in the code, and is used after the session to see if the code executed as expected. If a session is prematurely terminated it can be deduced where and sometimes how the error occured, by reading the last entry.

4.5 Implementation Status

In this section statistics for the program is presented. This is to give a sense of the size of the software system that was presented through the implementation chapter. This section contains the total number of code lines without comments and whitespace, the total number of methods and a list of all the classes implemented in the program.

Here is a list of the implementation status:

- Lines of code - 1233
- Number of methods - 85
- List of implemented classes
  - Driver
  - Lift
  - MyNode
  - Warehouse
  - LogJava
  - Moves
  - Pilot
  - JumpPointSearch

5 | Test

In this chapter, the final program controlling JAWA will be tested. These tests was conducted to determine, whether JAWA can act in accordance to the problem statement in Section 2.10. Testing is crucial to see if the design and implementation have been successful. The testing should be done carefully, to make sure that most issues are solved.

5.1 Test Tools

An array of test tools have been used to test JAWA. The following sections explain the usage of the tools, but not the tools themselves.

Logging

The log file contains a differentiated timestamp and a corresponding message. This offers some debugging features:

- To see how long each action takes to execute
- To see the latest action before a crash
- To see if parameters retains correct values
- To see if the actions occur in the right order

After a crash or program termination, the log file can be retrieved via Bluetooth or USB cable and reviewed. Calls to the `writeToFile()` method, needs to be placed in the code where testing is needed.

Screen printing

Printing to the LCD screen on the LEGO® NXT unit, offers information on runtime. However, due to the small size of the screen, not much text can be printed. The LCD screen has mainly been used during analysis tests and implementation of lesser methods, to review input from a sensors, in order to monitor their behaviour in real time.

Running on a PC

If leJOS specific classes and functions are removed, the program can be executed and compiled as a normal Java program. Setting aside the computation power of a PC, compared to the NXT unit, more than one developer can perform tests and develop on non-leJOS specific code. This proved convenient when the `Warehouse` and `JumpPointSearch` classes were developed and tested.

Ad hoc testing

This test was performed, without planning and documentation. Whenever JAWA did not behave as intended and neither a log file nor the screen printing options were sufficient nor adequate, an ad hoc test was performed. For instance, when JAWA turns 130° instead of 180° around its own axis. The testing consisted of changing parameters and the code, in order to get JAWA to behave as close to expected as possible.

5.2 Module Test

The module tests involve testing if all the features of JAWA works by themselves, and to determine if they work as intended. This is done to segment debugging time. It is relatively easier to determine bugs in a smaller module than the entire program as a whole.
In this section only one module will be documented thoroughly. The other modules follow the same procedure, and as such only have an introductory text, the results and a discussion of the results.
The module that is to be thoroughly documented will be the jump point search algorithm. The other modules of JAWA are as follows:

- Driver
- Warehouse
- Movement
- Pallet pickup and validation
- Return pallet
- Avoid obstacle

Jump Point Search test

Jump point search has been chosen for extensive testing. The tests will examine whether the algorithm can make a path through different environments, how long does it take to find the path and how scalable is it, when it has to find a path in a large map. These are core functionalities of the algorithm, and thus, important to test. The tests are smaller acceptance tests of the module.
First is whether the jump point search algorithm can find a path.

Pathfinding test

Objective: To determine whether the algorithm is able to find a path through different warehouse environments.
Expectations: It is expected that it makes no difference how the map looks, as long as there exists a path from the start point to the destination.
Setting: There are four cases, represented by the four graphs in Figure 5.1e. They are 6×6 grids, where the vertices are coordinates and the edges connect them. The edges have de facto weights, as they have a length that represent the real world. The edges in these graphs are 45 cm long. The algorithm is then tasked to find a path from $(0, 0)$, denoted by the circle, to a destination, marked by an $\times$. This was uploaded to JAWA and the test was conducted. The graphs and the paths were then extracted from JAWA.

- Case 1: The graph shown in Figure 5.1a. This is a graph representation of the rich picture in Figure 1.2. This graph is the main case scenario for JAWA.
- Case 2: The graph shown in Figure 5.1b. This graph was made to more closely resemble how a warehouse would actually look like.
- Case 3: The graph shown in Figure 5.1c. This graph was made to check if the jump point search could navigate in a more closed warehouse.
- Case 4: The graph shown in Figure 5.1d. This graph was made as an unsolvable scenario where the jump point search algorithm can not find a path.

Results: The paths are shown in Figure 5.1i. It can be seen that the algorithm finds adequate paths in the maps. Figure 5.1d is notably missing in the results, and that is because a path did not exist from the start to the destination. This is documented in the log file presented in a later test.
Conclusion: The paths are what was expected, and there were no path in Figure 5.1d, which was expected as well. The paths do, however, make turns that are not necessary, for instance in Figure 5.1f the path runs along the wall in the middle and turns to the right to reach the goal. If it had continued one point, when it cleared the wall, and then turned it could have saved one turn.

Execution time test

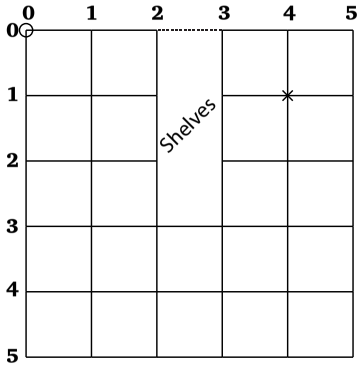
Objective: This test was performed to determine how much time the algorithm spends, when finding a path.
Expectations: It is expected that the algorithm finds a path within a reasonable time frame. For the maps created in the test above, a reasonable time frame will be about 2 seconds.
Setting: This test was performed in conjunction with the previous test. A log entry was made before and after each method was called. Then the timestamps were compared. This was performed in all of the test cases.
Result: The results are shown in Table 5.1.
Conclusion: As seen in the table the algorithm was very close to the expected time frame for case 1, while the rest of the cases were faster than expected. As such the algorithm is reliable and fast for smaller graphs.

Map Cases	Time (ms)	Time (s)
Case 1	2090	2
Case 2	841	0.8
Case 3	1349	1.3
Case 4	845	0.8

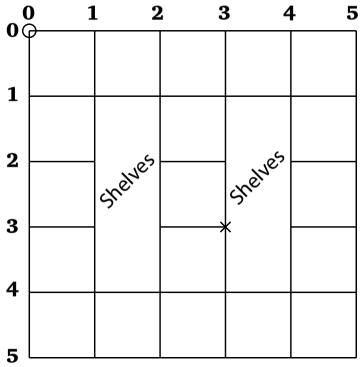
Table 5.1: A table of the execution times for four pathfinding cases.

Scalability test

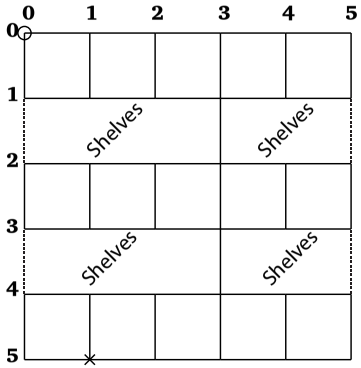
Objective: The purpose of this test, is to determine the scalability of the jump point search algorithm. This is important to see whether it would be realistic to increase the complexity and size of the environment.
Expectation: Given the fact that the warehouse is significantly larger than the previous cases, it is expected that it will be able to make paths, but it will take a long time. An expected time frame is about 2–3 minutes.
Setting: A large map, 20×10 nodes, is made with 45 cm between each node. The map is made to resemble a warehouse more closely, with rows of shelves, enclosed in a room that is open in two directions. Instead of a single pick-up zone there are two zones in this



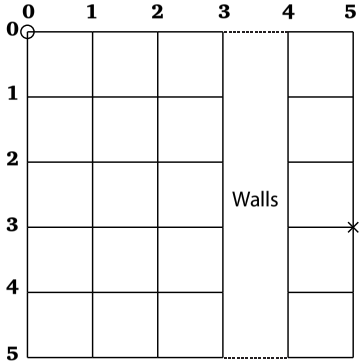
(a) The first graph, made to resemble the rich picture made in Figure 1.2.



(b) A graph made to more closely resemble how a warehouse might actually look.

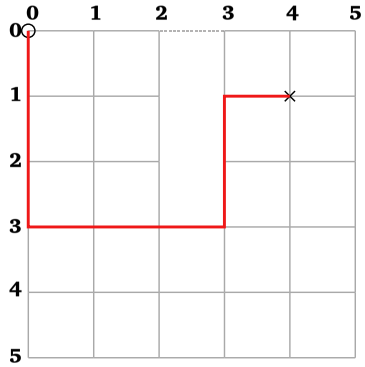


(c) A graph where JAWA have to find a path down two narrow alleys between shelves.

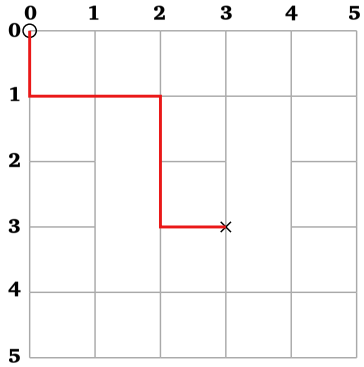


(d) A graph where JAWA needs to find a path to a destination it cannot reach.

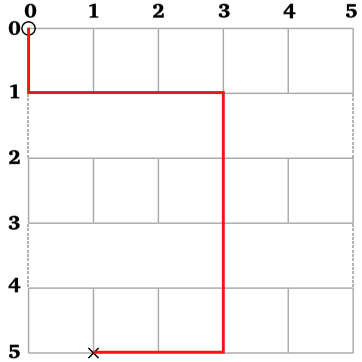
(e) The test cases for the jump point search algorithm, represented by four test warehouses. The circle marks the starting location and the × marks the destination. The dotted lines in figures **(a)**, **(c)**, and **(d)** are not walkable.



(f) A path through Figure 5.1a from $(0,0)$ to $(4,1)$.



(g) A path through the graph in Figure 5.1b from $(0,0)$ to $(3,3)$.



(h) A path through Figure 5.1c starting from $(0,0)$ to $(1,5)$.

(i) The paths found by the jump point search. The reason why there is no path through Figure 5.1d, is because it was not possible to pass the wall. As with Figure 5.1e the dotted lines in figures **a** and **c** are not walkable.

Figure 5.1: The test cases for the jump point search. Figures **(a)** through **(d)** are the warehouses, and figures **(f)** through **(h)** shows the paths through the warehouses.

case. JAWA needs to find a path from a starting point, $(0,0)$, to one pick-up point, then deliver that pallet to a shelf. From there it needs to check the other pick-up point and then return to the start point. This case was used to test how JAWA would function in a more realistic scenario that looks more like how a real warehouse would look. Figure 5.2a shows the map created by JAWA. Furthermore the test also includes logging the process, to determine how much time JAWA spends making the map and calculating the routes.
Result: Figure 5.2b shows the paths found in the warehouse. The red line is the first path from the circle to the first × in $(20,2)$. Then it follows the green line from $(20,2)$ to the second × in $(5,4)$. After that it follows the blue line to the third × in $(20,7)$. Lastly it returns to the circle via the yellow line, from $(20,7)$ to $(0,0)$. In Table 5.2 the times for the four paths can be seen.
Conclusion: Even though this test is a more extreme test case, it is still a valid test. The fact that JAWA can make a path in the map is not particularly important, though it is interesting to see how the paths look, to determine if it still makes as few turns as possible. As seen most of the paths takes less than 30 seconds to compute. The third path, however, takes about 2 minutes to compute. The paths is not significantly longer than the second path, but takes about 4 times longer. The reason for this is unknown at the moment. The entire process of finding a path and return to the start, takes 200.4 *ms* or about 3 minutes and 20 seconds. This is close to the highest expected estimate, so it can be concluded that the algorithm is scalable to some extend.

Path	Time *ms*	Time *s*
$(0,0) \rightarrow (20,2)$	27973	27.9
$(20,2) \rightarrow (5,4)$	21411	21.4
$(5,4) \rightarrow (20,7)$	124391	124.3
$(20,7) \rightarrow (0,0)$	26989	26.9
Total	200464	200.4

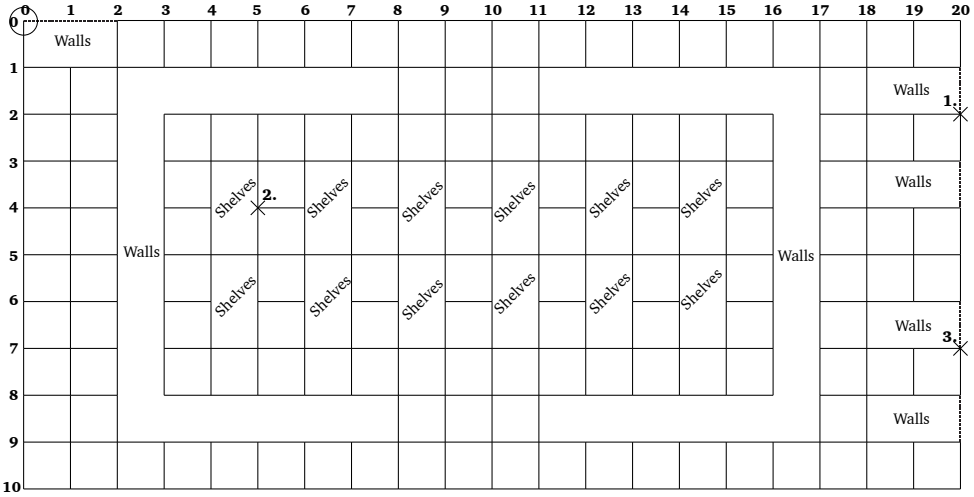
Table 5.2: The execution times for the four paths in Figure 5.2b.

Driver test

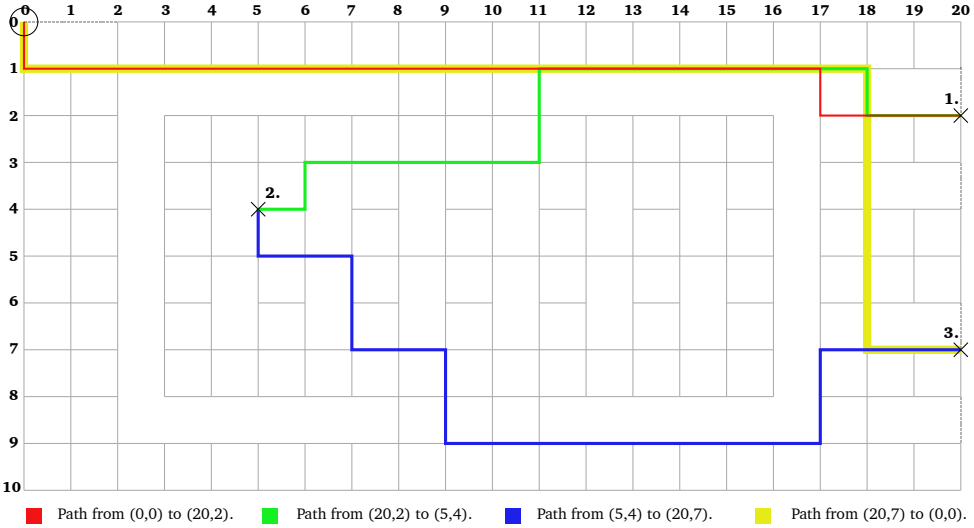
In Section 2.7.4, it was mentioned that the precision of the actuators needed to be tweaked in order to compensate for the loss in precision. The test of the `Driver` class determined the offset that was needed in order to make JAWA more precise. JAWA was placed in a small grid, and a series of tests were conducted. The tests primarily consisted of variations of parameters, making JAWA drive either in a straight line or turn. This resulted in inaccurate turning, and different outcomes of the same test. The tests resulted in making an approximate offset that makes JAWA more accurate.

Warehouse

Given the fact that a warehouse layout does not change often, loading from a pre-made map would be prudent. The warehouse test was made to determine the time spent creating the maps, in addition to testing when JAWA saves the map and reloads it later. Table 5.3 shows the time spent creating and loading the graphs from previous tests. The test cases are the four maps created in the pathfinding test and the larger map in the scalability test. It can be seen that the time spent creating the small graphs are slightly longer than the time loading them, though not by much. However, it takes about 11



(a) A map of a warehouse. It shows rows of shelves enclosed in a room open in two dirrection. To the right there are two drop-off zones where there can be pallets.



(b) The fulle path from start to finish. The path starts from the circle and proceeds to the first drop-off zone, **1**. Then it places that pallet on a shelf, **2**. Then it checks the other zone, **3**, and determines there is no pallet and returns to the start point.

Figure 5.2: The map and path in a larger map of a warehouse.

seconds to create the large map in case 5, but only less than 1 second to load it. The test is therefore a success.

Graphs	Time spent creating *ms*	Time spent loading *ms*
Case 1	161	135
Case 2	193	133
Case 3	327	124
Case 4	241	130
Case 5	10932	827

Table 5.3: The time spent creating and loading the graphs from the previous tests.

Pallet pickup/validate pallet

The purpose of this test is to pick up a pallet and determine its colour. Firstly the ultrasonic sensor determines whether there is a pallet ahead or not. If a pallet is detected, JAWA drives towards it until the colour sensor registers a colour change and JAWA can determines whether this is the pallet it is looking for. If it is, then JAWA picks it up, if not it drives back to the starting position. The tests were successful, JAWA is able to pick up and evaluate pallets.

Avoid obstacle

This test was made to determine if JAWA could avoid obstacles in its path. In the test, JAWA drove a path from a starting point to the pallet dropoff zone. On the path an obstacle was placed in front of JAWA. The test was successful, JAWA was able to stop at the obstacle, calculate a new route and reach the original goal node.

5.3 Integration Test

The integration test involved combining the module tests, to make sure that they work together and the correct results were documented. This was checked by using the log file during the code execution, to determine if a certain input gave the expected results. The benefits of this type of test is that the majority of errors and bugs could be eliminated, before doing an acceptance test. If an error occurred during one iteration of the tests, that had not occurred before, it could be determined that the latest module did not work with the rest of the modules. The base integration test was to combine the Jump Point Search -, the Driver -and the Warehouse modules, to have JAWA follow a path. After that the test was extended to have obstacles on the path, and JAWA was supposed to avoid them. It was further extended to have JAWA pick up a pallet at the end of the path.. Figure 5.3 and Figure 5.4 shows the expected sequences through the integration test.

The integration tests have all been conducted, using the map as seen in Figure 5.1a from position $(0, 0)$ to $(5, 1)$. This map was drawn onto a floor area, and the positions of conveyorbelts, walls and pallet pickup locations were marked onto the map.

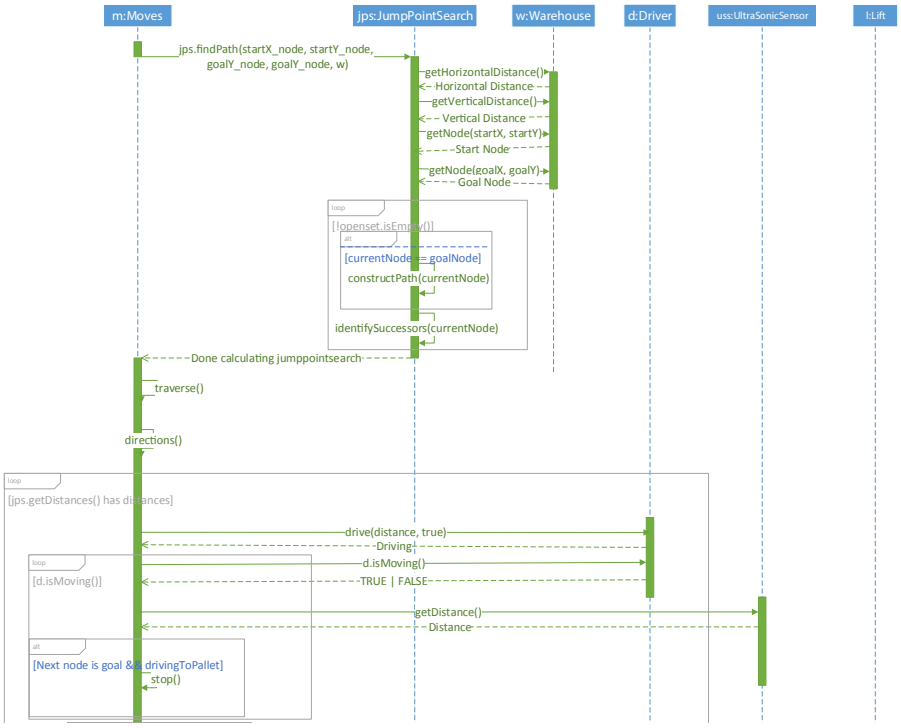


Figure 5.3: Sequence Diagram for following path, avoiding obstacles and pick up pallet.

Follow path

Objective: Here it was tested if the Warehouse -, Jump Point Search -and Driver modules were combinable.
Expectations: A satisfying result would be, if JAWA follows the path as expected and reaches the goal.
Setting: JAWA was given a map, which was created by the `Warehouse` class, and then the shortest path was found using the Jump Point Search algorithm. JAWA was then tasked to traverse the map using the `Moves` class.
Results: The test showed that JAWA was able to create a map, and compute the correct distances from jumppoint to jumppoint. The directions and distances can be seen in Listing 5.1.
Conclusion: During JAWA's physical traversing of the path, a rather large error margin in the precision was discovered. Both the distances driven and the turns performed were imprecise. The turning angles were smaller than expected. This resulted in JAWA stopping closer to position $(5, 0)$, rather than $(5, 1)$, which was a difference of approximately 45 cm.

```
1 557 ms     Instatiated warehouse
2 641 ms     Made walls in warehouse
3 2889 ms     Found path
4 3760 ms     Instatiated Moves - Finding path
5 3798 ms     Driving distance: 135cm with direction: south
6 12839 ms     Driving distance: 135cm with direction: east
7 21923 ms     Driving distance: 90cm with direction: north
8 28377 ms     Driving distance: 90cm with direction: east
9 34715 ms     Reached goal - Done
```

Listing 5.1: The log file for the pathfollowing test.

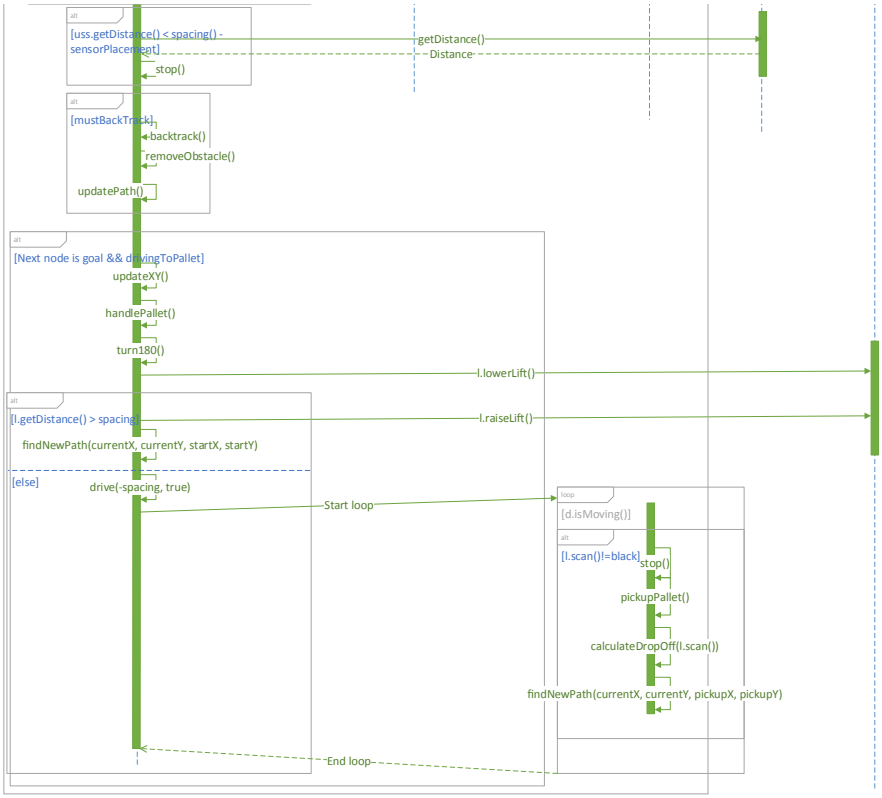


Figure 5.4: Figure 5.3 continued.

In the remaining integration tests, JAWA's position was adjusted each time it deviated from its path. This should not interfere, with the results as it will still be possible, to see if JAWA reaches its goal as expected.

Follow path and avoid obstacle

Objective: The second test involved extending the previous test with the Avoid Obstacles module.
Expectations: A satisfying result would be if JAWA did not collide with any obstacle. It should always be able to find a new path if an obstacle is met, no matter how many times the map is updated, as long as a path exists.
Setting: The test was performed on the same map as the previous test, with 3 obstacles placed in the path.
Results: The results, seen in Listing 5.2, shows that JAWA was still able to find a path to its goal, while backtracking and calculating a new path, each time an obstacle was met.
Conclusion: From the log file it can be seen that JAWA encounters an obstacle in three different occasions and the path is then updated with the current coordinates. The newly generated warehouse, with the lines to the obstacles removed can be seen in Figure 5.5. From this test and the log file, it can be concluded that JAWA can avoid obstacles.

```
1 557 ms     Instatiated warehouse
2 641 ms     Made walls in warehouse
3 2889 ms     Instatiated Moves - Finding path
4 3760 ms     Found path
5 3730 ms     Driving distance: 135cm with direction: south
6 9981 ms     Saw an obstacle
```

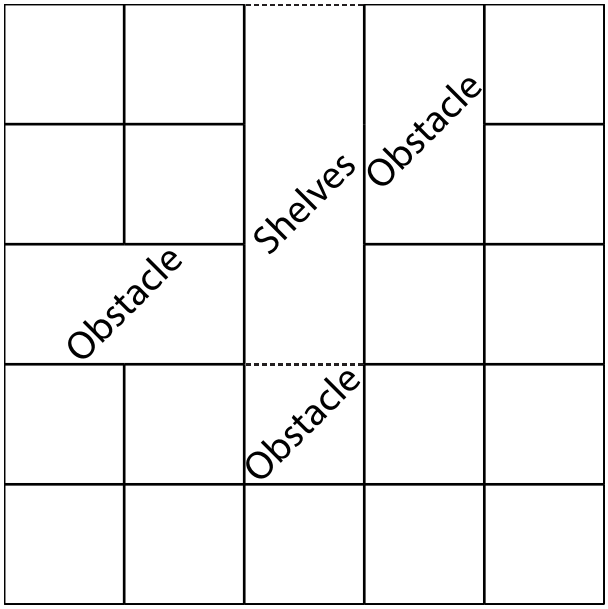


Figure 5.5: New warehouse after lines to obstacles are removed.

```
10001 ms    Backtracking: -12cm
11188 ms    Removed line between x: 0.0, y: 2.0 and x: 0.0, y: 3.0
11220 ms    Updating path from x: 0.0, y: 2.0 to x: 5.0, y: 1.0
15142 ms    Driving distance: 45cm with direction: east
19066 ms    Driving distance: 45cm with direction: south
22976 ms    Driving distance: 90cm with direction: east
27202 ms    Saw an obstacle
27223 ms    Backtracking: -22cm
28872 ms    Removed line between x: 2.0, y: 3.0 and x: 3.0, y: 3.0
28912 ms    Updating path from x: 2.0, y: 3.0 to x: 5.0, y: 1.0
32234 ms    Driving distance: 45cm with direction: south
36228 ms    Driving distance: 45cm with direction: east
40203 ms    Driving distance: 135cm with direction: north
49237 ms    Driving distance: 90cm with direction: east
50619 ms    Saw an obstacle
50640 ms    Backtracking: -17cm
52000 ms    Removed line between x: 3.0, y: 1.0 and x: 4.0, y: 1.0
52031 ms    Updating path from x: 3.0, y: 1.0 to x: 5.0, y: 1.0
54134 ms    Driving distance: 45cm with direction: north
58046 ms    Driving distance: 45cm with direction: east
62073 ms    Driving distance: 45cm with direction: south
66100 ms    Driving distance: 45cm with direction: east
69906 ms    Reached goal - Done
```

Listing 5.2: The log file for following the path and avoiding obstacles.

Follow path, avoid obstacles and pickup pallet

Objective: In addition to the previous sequence, this test adds the Pick Up Pallet module to it.
Expectations: A satisfying result would be if JAWA traverses the map, using the shortest path and avoids colliding with any obstacles. If an obstacle is met, a new path should be found. When reaching the goal it should be able to pick up the designated pallet.

Setting: As with the previous tests, JAWA was to follow a path to a goal node, while avoiding one obstacles. When it reaches the goal, it is opted to turn 180° and pick the pallet up.
Results: From the log file seen in Listing 5.3 it can be seen that when an obstacle is detected, a new path is calculated and traversed. When the node before the pallets pickup zone is reached, it can be seen that the ultrasonic sensor sees a pallet and drives towards it until the colour sensor sees that the pallet is green.
Conclusion: It can be concluded that JAWA can pick up the pallet. Further testing for picking up the pallet shows that it is possible to both pick up a pallet when driving straight in a line against it, or turning in a jump point next to the pallets position.

```
557 ms     Instatiated warehouse
641 ms     Made walls in warehouse
2889 ms     Instatiated Moves - Finding path
3692 ms     Found path
3730 ms     Driving distance: 135cm with direction: south
12787 ms     Driving distance: 135cm with direction: east
21874 ms     Driving distance: 90cm with direction: north
28331 ms     Driving distance: 90cm with direction: east
29549 ms     Saw an obstacle
29570 ms     Backtracking: -14cm
30767 ms     Removed line between x: 3.0, y: 1.0 and x: 4.0, y: 1.0
30799 ms     Updating path from x: 3.0, y: 1.0 to x: 5.0, y: 1.0
32490 ms     Driving distance: 45cm with direction: north
36604 ms     Driving distance: 45cm with direction: east
40594 ms     Driving distance: 45cm with direction: south
44535 ms     Driving distance: 45cm with direction: east
44681 ms     Reached node before goal
47105 ms     Pallet detected with ultrasonic sensor
49589 ms     Drove 37cm before seeing a colour change
50482 ms     Picked up a blue pallet
51360 ms     Done
```

Listing 5.3: The log file for following the path, avoiding an obstacle and picking up a pallet.

Conclusion

All the functionality in the integration test worked as intended. When the different modules were combined and tested together, the expected results were achieved. A common issue during the testing was the imprecision of the actuators. As a result, JAWA can not traverse the map on its own and requires to be adjusted after each turn as the accumulated imprecision makes JAWA unable to pick up the pallet or reach the goal destination.

5.4 Acceptance Test

The acceptance test involves testing the full system with all its functionalities, to make sure that it lives up to the expectations. This is where all the previous tests are combined, every functionality of JAWA should be tested during this final test. The advantage of the acceptance test, is that the final errors and faulty behaviour between the modules, can be seen as JAWA traverses the physical warehouse, as shown in Figure 5.6. One disadvantage is that if an error occurs, it can be hard to locate where the failure occurs, due to the whole system being executed. But with the use of logging, locating these

errors becomes easier.

If the acceptance test was completed and fulfilled all of the requirements stated in the problem statement Section 2.10 have been met.

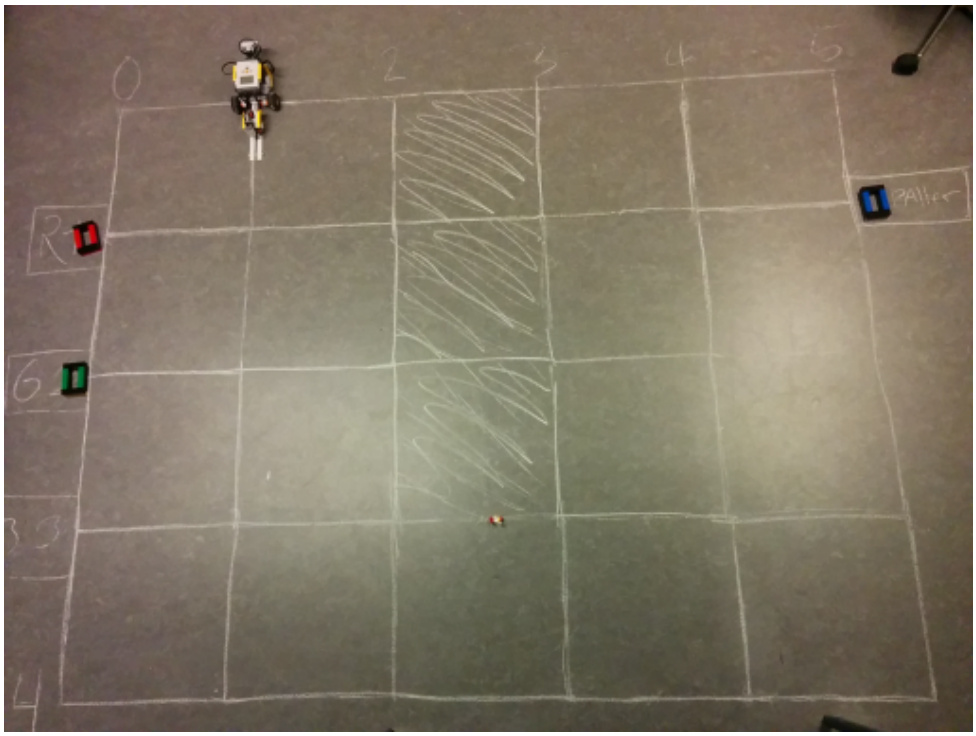


Figure 5.6: A picture of the acceptance test done in the group room.

Objective: To determine if the entire program will work as intended and the program flows as expected.

Expectations: It is expected that JAWA can navigate through a warehouse and autonomously avoid obstacles. It is also expected that JAWA will be able to pick up pallets from designated locations, and deliver them to a drop-off point.
Setting: There was four test cases of the acceptance test, each testing the interaction between different functionality. In all cases, JAWA started in $(0, 0)$. After every turn, JAWA had to be readjusted back to the path. The cases are as follows:

- **Case 1:** Drive towards the blue conveyor belt expecting a blue pallet. Avoid one obstacle. Find blue pallet and deliver it to the pallet drop-off zone. Drive towards blue conveyor belt again, find no pallet. Return to start
- **Case 2:** Drive towards the blue conveyor belt expecting a blue pallet. Find red pallet. Leave pallet and return to start.
- **Case 3:** Drive towards the pallet drop-off zone. Avoid one obstacle. Find blue pallet. Drop blue pallet off at blue conveyor belt. Drive towards pallet drop-off zone again. Find no pallet and return to start.
- **Case 4:** Drive towards the pallet drop-off zone. Avoid one obstacle. Find no pallet. Return to starting position.

Results: The results can be seen in the log files in Appendix C. It shows the log files for all four cases, from `Warehouse` has been instantiated to JAWA has returned to the start.

Conclusion: As seen in the log files, the flow of the program was as expected. Additionally, JAWA did physically follow the same flow. As such it can be concluded that the acceptance test passed.

6 | Conclusion

In this report an embedded system, designed for operating in a warehouse, is described. Specifically, we developed an autonomous robot that can move pallets in a warehouse, to and from specific locations. The motivation for creating such a robot, is that the task of moving pallets could be automated.

In the analysis chapter, we explored the opportunities and limitations of the hardware and software platform. The exploration of the platform consisted of a series of tests that clarified the properties of the sensors and actuators. Furthermore, a number of traversal algorithms were researched and analysed to find the best suited for our robot.

In the design chapter, we provided the prerequisites of designing the robot as an agent. After that, the mechanical design of the robot was documented, as well as the actual agent design. Last was an overview of a software architecture implemented in the robot.

In the implementation chapter, we explore some of the approaches chosen, and code examples for important methods. Furthermore, a status of how many lines of code there is in the program, how many methods and all the classes implemented.

In the test chapter, we provided the test tools that were used during the testing phase. Then a module -, integration -and an acceptance test were made.

To conclude, the requirements presented in Section 2.10 have been adequately satisfied.
Follow a map — The robot follows a map as closely as possible. It can interpret the path created from the pathfinding algorithm and follow it. The precision of turning is the only remaining issue that we were not able to solve. This can be seen in Section 5.4.
Identify pallets — It identifies the colour of the pallets and returns them to the designated conveyour belts. This, too, can be seen in Section 5.4.
Move pallets — This is also met, because the robot is able to move the pallets to and from drop-off zones.
Avoid collision — The robot is able to avoid obstacle and compute a new path, as seen in in Section 5.3.
Based upon the results of the tests, it has been concluded that the requirements are satisfied.

6.1 Reflection

In this section the progression of our project will be discussed, to evaluate the work process and see what we could have done differently. The chapters and the work process in each of these will be discussed separately in the following sections.

Reflection of the Analysis

The initial idea was to have the robot search the environment for LEGO® balls, collect and sorting them by colour. This was supposed to be done using a radar feature to scan the environment for the LEGO® balls. Without much prior testing, the radar feature was implemented and the initial testing began. We concluded that the robot was not able to detect the LEGO® balls whilst scanning the environment. A test that determined if the ultrasonic sensor could detect the LEGO® ball was conducted. The test showed that the

ultrasonic sensor had issues detecting the LEGO® balls at longer distances. If this test was conducted before the radar test, it could have been concluded that the radar feature would not work.

When we began using the sensors it was found, that we did not know how to gather the correct information from them. Following this, a series of tests were conducted, to calibrate or determine the error margin of all the needed LEGO® NXT sensors. We started the testing of the sensors by considering, what kinds of tests we could conduct on each specific sensor. This resulted in a lot of ad hoc testing, without much consideration of why we needed the specific tests or expectations to the result. Some of the sensors were tested, for the same thing in different ways and earlier tests were not used as foundation for later tests. Instead we should have started by considering, what we needed the sensors for and what kind of tests would satisfy this. The tests were also conducted without a clear outline of, how they were to be conducted and how to collect the data. The chapter was restructured and rewritten, and a clearer guideline for the testing set up was created and followed for all test cases. If some considerations for the tests were made, it could have saved us a great deal of time not having to sort through all the unnecessary tests.

After redefining the goals of the project, a graph was needed to represent the environment. As such, an algorithm to determine the best path had to be found. Therefore, a few shortest path algorithms were analysed and compared. Since we knew that the robot had issues bound with the rotation of the actuators, we had to prioritise algorithms with less turns. Therefore the Jump Point Search algorithm was chosen because it fits the criterion. However, we found no official pseudo code for the algorithm, or any information as to how it should be implemented. Instead we researched definitions, found a visualisation tool, and read Javascript and Lua code for the algorithm. This helped us understand the algorithm and translate it to fit our case.

Reflection of the Design

Like stated above, the initial idea was to have the robot search the environment for LEGO® balls and determine their colour. This meant that the design had to be made to accommodate this problem. When we later found that it was not possible to achieve this, a second design was made, being the design of the current robot. This means that the mechanical design had to be remade, and a lot of time was used to build these designs.

What could have been done instead was to make a test design that satisfied all the needs for testing the sensors. That way the initial results of the tests could be gained and then later create the final design of the robot.

The most questionable design decision was the castor wheel, which makes the robot imprecise after turns, because it has to correct itself after turning. Several attempts at making a replacement was made. One replacement had the robot using a ski at the front. This provided promising results at first, however, after more usage, the ski became worn and rendered the robot incapable of turning at all. The castor wheel therefore ended up being the best solution bringing low friction without too much trade-off in turning ability.

Reflection of the Implementation

In the implementation of the software architecture each class was written separately. When the classes were done, they were linked together according to the class diagram. Early in the project we did not have any way, of discovering where and why code

executions had stopped. We spend a lot of time on running small physical tests, to check if the code worked correctly. Therefore we created a log file, to help locate possible errors. Though, the first implementation of the log file was faulty and it added to the problems. However, after the log was fixed the tests, that included the file, needed to be redone.

Reflection of the test

The structure of the testing phase worked particularly good. The chronology of the module tests, going into an integration test and resulting in an acceptance test, helped identifying bugs and errors more effectively.

6.2 Future Works

This section presents some improvements that would make the robot more efficient. The improvements suggested are concepts that could make the robot more advanced instead of just core features.

Multi-Agent

When designing the robot it was decided that there would only be one agent in the environment, moving pallets. In a real life scenario only having one unit to move pallets would be too slow, if the warehouse has many pallets coming and going. Having multiple agents in the environment would increase the amount of pallets moved and the system as a whole would be more efficient. This could be done by having the map and algorithm off site, on a server, and have the agents ping the server with their locations and their goal. This way the server could keep the agents from colliding.

Detailed Environment

The environment that the robot operates in, only has a few elements that the agent needs to take into consideration. The only thing other than avoiding obstacles, that the robot has to know, is where to deliver the pallets. It was chosen that the pallets would be delivered to conveyor belts. In another case scenario, the pallets might have to be delivered to shelves at different heights. In this scenario, the level the pallet should be delivered at, should be taken into consideration. It was also chosen that the level in which the robot traverses the environment is static. A more detailed environment could have different levels with ramps that make the robot drive both up and down.

Machine Learning

A feature that was not implemented to the robot was the ability to have machine learning. The intention was to have the robot learn over time what the best route was for the different colour pallets. This would also allow for the robot to skip calculating a new route every time a pallet needed delivery. When the robot encounters an obstacle, it should remember where the obstacle was and take into consideration the next time a new path is created. Over time the robot should learn that certain areas on the graph is likely to have obstacles and therefore avoid that area.

Bibliography

Michael Barr & Anthony Massa (2007). *Programming Embedded Systems: With C and GNU Development Tools*. 2nd edition. O'Reilly media. ISBN: 978-0-596-00983-0.

Takashi Chikamasa (2013). *nxtOSEK/JSP*. Accessed the 17'Th September, 2014. URL: http://lejos-osek.sourceforge.net/.

Egemin Automation (2014). *Egemin Automation*. Accessed the 17'Th September, 2014. URL: http://www.egemin-automation.com/en/automation/material-handling-automation_ha-solutions/agv-systems.

Peter Feiler (2003). *Real-Time Application Development with OSEK: A Review of the OSEK Standards*. Accessed the 3'Th December, 2014. URL: http://resources.sei.cmu.edu/asset_files/TechnicalNote/2003_004_001_14129.pdf.

Generationrobots (2014). *Ultrasonic sensors for robots*. Accessed the 16'Th September, 2014. URL: http://www.generationrobots.com/en/content/65-ultrasonic-sonar-sensors-for-robots.

Kjeld Jensen, Morten Larsen, Søren H. Nielsen, Leon B. Larsen, Kent S. Olsen & Rasmus N. Jørgensen (2014). "Towards an Open Software Platform for Field Robots in Precision Agriculture". In: *Robotics* 3.2. Accessed the 17'th November, 2014, pp. 207–234. ISSN: 2218-6581. DOI: 10.3390/robotics3020207. URL: http://www.mdpi.com/2218-6581/3/2/207.

LEGO GROUP (2014). *LEGO MINDSTORM NXT Hardware Developement Kit*. Accessed the 16'Th September, 2014. URL: http://www.csd.uoc.gr/~hy428/reading/lego_nxt_hw_dev_kit.pdf.

– (2012). *leJOS NXJ API documentation*. Accessed the 17'Th September, 2014. URL: http://www.lejos.org/nxt/nxj/api/.

leJOS (2014). *NLeJOS, Java for Lego Mindstorms / NXJ*. Accessed the 17'Th September, 2014. URL: http://www.osek-vdx.org/.

Brian Nielsen (2014). *NXT HW & Sensors and Actuators*. Accessed the 14'Th October, 2014, pp. 82–83. URL: https://www.moodle.aau.dk/pluginfile.php/385665/mod_resource/content/1/embedded-hardware-and-sensorsactuators.pdf.

nxt OSEK (2014). *ECRobot C/C++ API for nxtOSEK*. Accessed the 17'Th September, 2014. URL: http://lejos-osek.sourceforge.net/html/index.html.

OSEK/VDX (2014). *nxtOSEK/JSP*. Accessed the 17'Th September, 2014. URL: http://www.osek-vdx.org/.

David Poole & Alan Mackworth (2010). *Artificial Intelligence - foundations of computational agents*. 1st edition. Cambridge University Press, pp. 11,19–29,74. ISBN: 978-0-521-51900-7.

University of Windsor (2011). *LEGO NXT: Features & Limitations*. Accessed the 30'Th September, 2014. URL: http://nxt.cs.uwindsor.ca//499football/features_limitations.pdf.

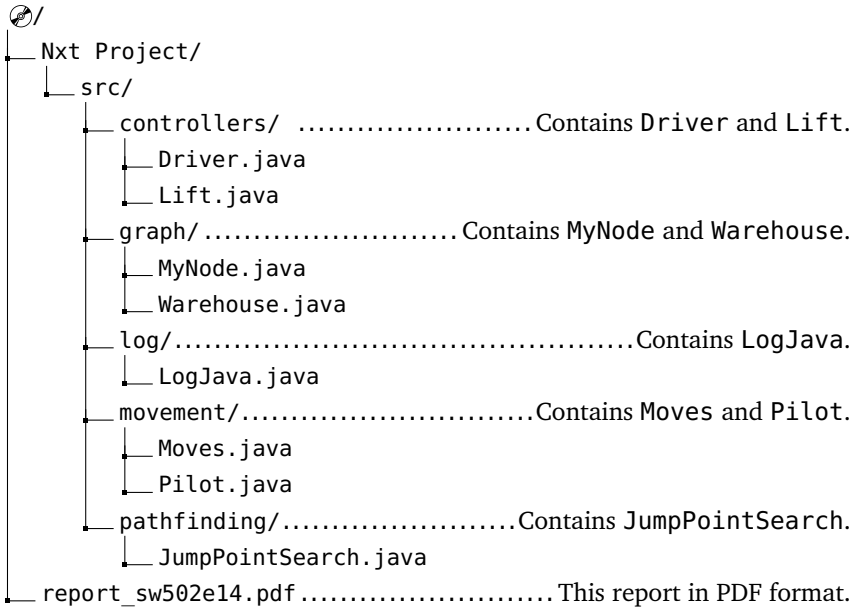
Wikipedia (2014a). *A* search algorithm*. URL: http://en.wikipedia.org/wiki/A*_search_algorithm.

– (2014b). *Dijkstra's algorithm*. URL: http://en.wikipedia.org/wiki/Dijkstra's_algorithm.

Nathan Witmer (2013). *Jump Point Search Explained*. URL: http://zerowidth.com/2013/05/05/jump-point-search-explained.html.

Holly Yanco & Jill Drury (2004). *Classifying Human-Robot Interaction: An Updated Taxonomy*. Accessed the 2'Th October, 2014. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1400763.

A | Disc


/
Nxt Project/
src/
controllers/ ..... Contains Driver and Lift.
Driver.java
Lift.java
graph/ ..... Contains MyNode and Warehouse.
MyNode.java
Warehouse.java
log/ ..... Contains LogJava.
LogJava.java
movement/ ..... Contains Moves and Pilot.
Moves.java
Pilot.java
pathfinding/ ..... Contains JumpPointSearch.
JumpPointSearch.java
report_sw502e14.pdf ..... This report in PDF format.

B | Pictures of JAWA

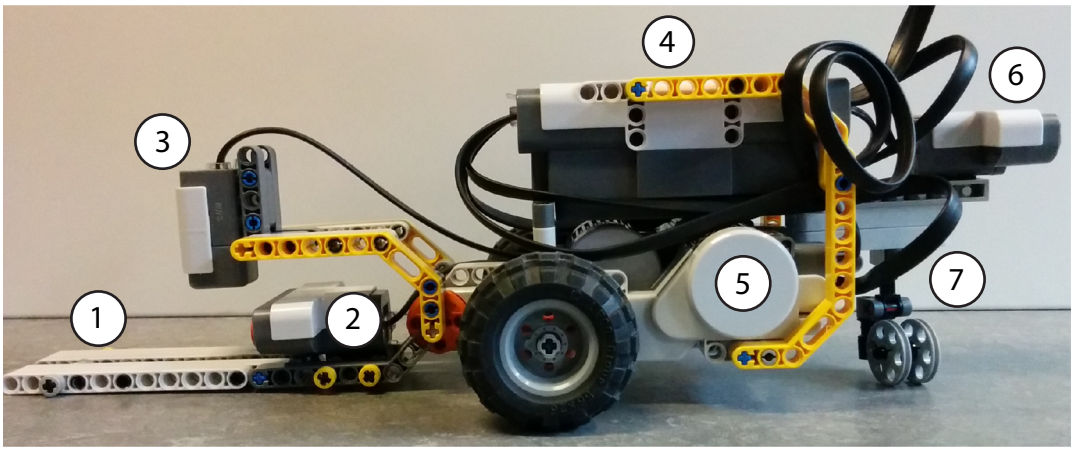


Figure B.1: A picture of JAWA taken from the side

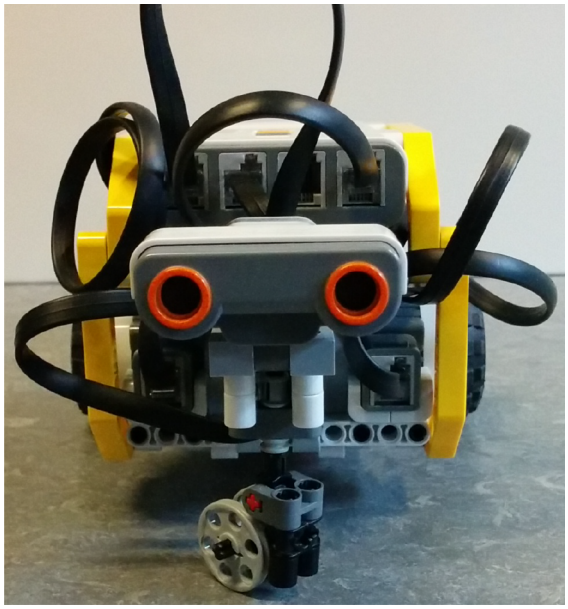


Figure B.2: A picture of JAWA taken from the front

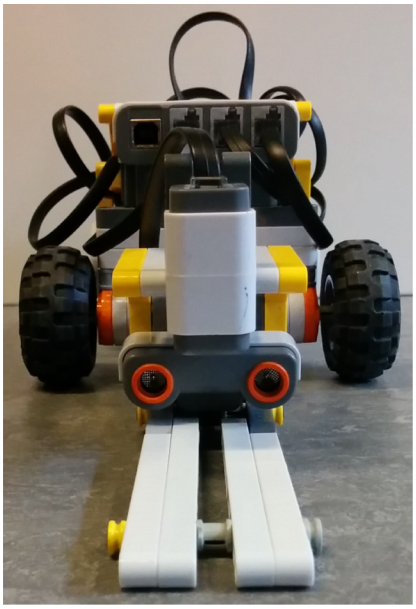


Figure B.3: A picture of JAWA taken from the back

Figure B.4: A picture of JAWA taken from above

C | Acceptance Test Logs

```
1  541 ms     Instatiated warehose
2  626 ms     Made walls in warehose
3  1533 ms     Found path
4  2474 ms     Instatiated Moves - Finding path
5  2502 ms     Driving distance: 135cm with direction: south
6  12423 ms     Pallet detected with ultrasonic sensor
7  14121 ms     Drove 23cm before seeing a colour change
8  15017 ms     Picked up a blue pallet
9  15040 ms     Backtracking: 23cm
10 16741 ms     Saw blue pallet at blue conveyor - finding path to dropoff zone
11 16774 ms     Finding path to pickup from x: 0.0, y: 3.0 to x: 5.0, y: 1.0
12 19555 ms     Driving distance: 135cm with direction: east
13 28630 ms     Driving distance: 90cm with direction: north
14 35138 ms     Driving distance: 90cm with direction: east
15 40641 ms     Reached dropoff with pallet - delivering
16 43382 ms     Finding path to blue dropoff, from x: 5.0, y: 1.0 to x: 0.0, y: 3.0
17 47301 ms     Driving distance: 90cm with direction: south
18 53777 ms     Driving distance: 135cm with direction: west
19 55415 ms     Saw an obstacle
20 55436 ms     Backtracking: -22cm
21 57085 ms     Removed line between x: 5.0, y: 3.0 and x: 4.0, y: 3.0
22 57122 ms     Just backtracked
23 57142 ms     Updating path from x: 5.0, y: 3.0 to x: 0.0, y: 3.0
24 61363 ms     Driving distance: 45cm with direction: north
25 65312 ms     Driving distance: 45cm with direction: west
26 69274 ms     Driving distance: 45cm with direction: south
27 73270 ms     Driving distance: 90cm with direction: west
28 78778 ms     Driving distance: 90cm with direction: west
29 86596 ms     No pallet detected
30 87478 ms     Expected blue pallet saw: nothing
31 90214 ms     Finding path back to start from x: 0.0, y: 3.0
32 92054 ms     Driving distance: 135cm with direction: north
33 102437 ms     Reached goal - Done
```

Listing C.1: The log file for the pathfollowing test.

```
1  541 ms     Instatiated warehose
2  626 ms     Made walls in warehose
3  1533 ms     Found path
4  2474 ms     Instatiated Moves - Finding path
5  2502 ms     Driving distance: 135cm with direction: south
6  12374 ms     Pallet detected with ultrasonic sensor
7  13484 ms     Drove 12cm before seeing a colour change
8  14380 ms     Picked up a red pallet
9  14403 ms     Backtracking: 12cm
10 15500 ms     Expected blue pallet saw: red
11 18236 ms     Finding path back to start from x: 0.0, y: 3.0
12 20478 ms     Driving distance: 135cm with direction: north
13 30921 ms     Reached goal - Done
```

Listing C.2: The log file for the pathfollowing test.

```
1  541 ms     Instatiated warehose
```

```
626 ms     Made walls in warehose
3525 ms     Found path
4466 ms     Instatiated Moves - Finding path
4505 ms     Driving distance: 135cm with direction: south
13564 ms     Driving distance: 135cm with direction: east
22639 ms     Driving distance: 90cm with direction: north
29139 ms     Driving distance: 90cm with direction: east
32213 ms     Reached node before goal
34563 ms     Pallet detected with ultrasonic sensor
37489 ms     Drove 45cm before seeing a colour change
38377 ms     Picked up a blue pallet
38407 ms     Backtracking: 44cm
41366 ms     Finding path to blue dropoff, from x: 4.0, y: 1.0 to x: 0.0, y: 3.0
45286 ms     Driving distance: 90cm with direction: south
48420 ms     Saw an obstacle
48441 ms     Backtracking: -3cm
49007 ms     Removed line between x: 4.0, y: 2.0 and x: 4.0, y: 3.0
49037 ms     Just backtracked
49058 ms     Updating path from x: 4.0, y: 2.0 to x: 0.0, y: 3.0
52960 ms     Driving distance: 45cm with direction: west
56867 ms     Driving distance: 45cm with direction: south
60779 ms     Driving distance: 45cm with direction: west
63803 ms     Driving distance: 90cm with direction: west
69309 ms     Reached dropoff with pallet - delivering
72048 ms     Finding path to pickup from x: 0.0, y: 3.0 to x: 5.0, y: 1.0
77140 ms     Driving distance: 135cm with direction: east
86242 ms     Driving distance: 45cm with direction: north
90170 ms     Driving distance: 90cm with direction: east
96628 ms     Driving distance: 45cm with direction: north
96774 ms     Reached node before goal
99175 ms     No pallet detected
100052 ms     Finding path back to start from x: 5.0, y: 2.0
104887 ms     Driving distance: 90cm with direction: west
111348 ms     Driving distance: 45cm with direction: south
115328 ms     Driving distance: 45cm with direction: west
119340 ms     Driving distance: 135cm with direction: north
128424 ms     Driving distance: 90cm with direction: west
135755 ms     Reached goal - Done
```

Listing C.3: The log file for the pathfollowing test.

```
541 ms     Instatiated warehose
626 ms     Made walls in warehose
3525 ms     Found path
4466 ms     Instatiated Moves - Finding path
4505 ms     Driving distance: 135cm with direction: south
13611 ms     Driving distance: 135cm with direction: east
22640 ms     Driving distance: 90cm with direction: north
25759 ms     Saw an obstacle
25780 ms     Backtracking: -3cm
26333 ms     Removed line between x: 3.0, y: 2.0 and x: 3.0, y: 1.0
26364 ms     Just backtracked
26385 ms     Updating path from x: 3.0, y: 2.0 to x: 5.0, y: 1.0
27979 ms     Driving distance: 45cm with direction: east
31945 ms     Driving distance: 45cm with direction: north
35850 ms     Driving distance: 45cm with direction: east
35996 ms     Reached node before goal
38339 ms     No pallet detected
39225 ms     Finding path back to start from x: 4.0, y: 1.0
43170 ms     Driving distance: 45cm with direction: south
```

```
46193 ms     Driving distance: 45cm with direction: south
50192 ms     Driving distance: 90cm with direction: west
56658 ms     Driving distance: 135cm with direction: north
65695 ms     Driving distance: 90cm with direction: west
73031 ms     Reached goal - Done
```

Listing C.4: The log file for the pathfollowing test.