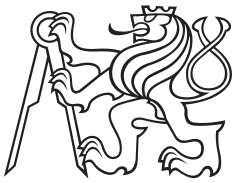
Bachelor Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Execution of Multi-Agent Path Finding Schedules

Jakub Pícha

Supervisor: RNDr. Miroslav Kulich Ph.D.
Study program: Open Informatics
Specialisation: Artificial Intelligence and Computer Science
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Pícha Jakub** Personal ID number: **498817**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Execution of Multi-Agent Path Finding Schedules

Bachelor’s thesis title in Czech:

Realizace plánů multi-agentního plánování hledání cest

Guidelines:

The task of Multi-Agent Path Finding (MAPF) consists in finding an optimal collision-free trajectory for a group of mobile agents (robots) from their starting positions to specified destinations. However, when executing the plan, there are inaccuracies in the control of the robots that make it impossible to execute the plan as it was designed. The aim of the work will be to experimentally verify how the execution of the plan corresponds to the proposed plan. The student performs this verification in the following steps:
1. Get acquainted with current versions of multi-agent planning algorithms, their robust versions and freely available implementations.
2. Get acquainted with the environment (simulator) for performing multi-robotic experiments developed by the IMR group, CIIRC.
3. Create utilities to run schedules generated by selected freely available planners in the IMR simulator.
4. Compare experimentally properties of execution of plans generated by selected planners with different control accuracy settings.
5. Describe and comment on the knowledge gained.

Bibliography / sources:

[1] https://ieeexplore.ieee.org/abstract/document/8620328
[2] https://ieeexplore.ieee.org/abstract/document/5980306
[3] https://arxiv.o- https://www.ijcai.org/proceedings/2021/0568.pdf
[4] https://ojs.aaai.org/index.php/AAAI/article/view/21266/21015r
[5] https://www.ijcai.org/proceedings/2021/0568.pdf
[6] https://github.com/kei18/mapf-IR
[7] https://github.com/Jiaoyang-Li/MAPF-LNS2

Name and workplace of bachelor’s thesis supervisor:

RNDr. Miroslav Kulich, Ph.D. Intelligent and Mobile Robotics CIIRC

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **01.02.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

RNDr. Miroslav Kulich, Ph.D.	prof. Dr. Ing. Jan Kybic	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____ Date of assignment receipt

_____ Student's signature

Acknowledgements

I would like to thank RNDr. Miroslav Kulich, Ph.D. for his patience, willingness to help, compassionate guidance and overall being, at least in my eyes, image of ideal supervisor.

Another person that I would like to extend my gratitude to is Ing. David Zahrádka, for his tireless help with work on the simulator, willingness to answer my, undoubtedly at times tedious, questions and overall being great support throughout the entire time spent with the thesis.

Declaration

I declare that the presented work was written independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Pícha Jakub

.....
In Prague, 24. May 2024

Abstract

The Goal of this thesis was to test the robustness of Multi-Agent Pathfinding algorithms on a simulator. For this task, it was necessary to extend the already existing simulator of logistical robots to be able to simulate parameterised imprecise movement. This is achieved through the implementation of motion models. Further in the thesis can be found a description of the used planning algorithms and experiments done on the simulator. By the end are presented the results and findings from these tests, primarily concerning the relation between real and theoretical cost of the plans, and what affects them.

Keywords: Multi-Agent Pathfinding, simulation of robots, motion models

Supervisor: RNDr. Miroslav Kulich Ph.D.
CIIRC, CTU in Prague,
Jugoslávských partyzánů 1580/3,
Praha 6

Abstrakt

Cílem této práce bylo testovat robustnost plánů multi-agentních plánovačů na simulátoru. Za tímto účelem bylo potřeba rozšířit již existující simulátor robotů o implementaci pohybových modelů, tak aby mohl simulovat parametricky nepřesný pohyb. Po čáasti o simulátoru v práci následuje popis testovaných algoritmů a provedených experimentů. Ke konci jsou prezentováný proznátky z těchto testů, primárně o vztahu teoretické a reálné ceny plánu a jeké jevy tento vztah ovlivňují.

Klíčová slova: multi-agentní plánování, simulace robotů, pohybové modely

Překlad názvu: Realizace plánů multi-agentního plánování hledání cest

Contents

Figures

Chapter 1

Introduction

With a drive for efficiency, there is currently rising interest in robotic-based automation in logistics, such as robot fleets servicing warehouses. This motivated and still does not only developments of robots themselves but also how they will be coordinated. One way to handle the navigation and coordination of robots is to formulate it into a Multi-Agent Pathfinding (MAPF) problem, which was developed to ideally suit this task.

Although studies of Multi-Agent Pathfinding is a relatively new field, there already exists a plethora of algorithms that have been developed to solve it, and there are many solvers applying these algorithms to specific incarnations of MAPF problems. The potential problem lies in the fact that these algorithms presume that agents will follow instructions in solution with absolute precision and no delays, but this is unlikely to be the case in real-world applications.

This thesis aims to test how the solutions provided by different MAPF algorithms perform under imperfect conditions. For this goal simulator, provided by CIIRC (Czech Institute of Informatics, Robotics and Cybernetics), had to be extended to provide these conditions and a testing pipeline had to be established.

The main achieved results were an extension of the simulator by the addition of motion models and features related to them, identifying and fixing of plethora of issues from prior development. Also of note are scripts that form a testing pipeline alongside the simulator, which makes it far simpler than it was initially, to go from the scenario for a planner to the results of

multiple runs in a graph or table.

Concerning tests themselves, experiments done so far do not seem to point favourably to the main hypothesis, But from the gathered data, useful insights were made into the relation between the theoretical cost of the plan and its real cost measured in experiments.

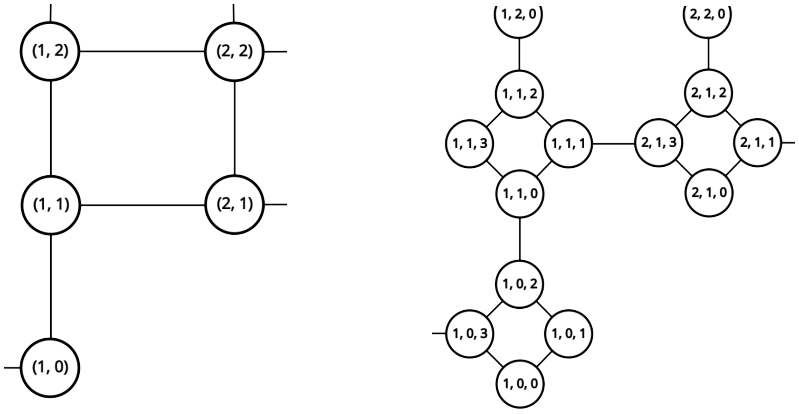
Chapter 2

Problem Specification

2.1 Introduction to Multi-Agent Pathfinding

In general, Multi-Agent Pathfinding (MAPF) is the problem of planning paths for multiple agents in a shared environment, in such a way that collisions between agents or agents and obstacles are preempted. This problem of finding collision free paths quickly becomes very complex with higher densities of agents in the environment. For that reason, specialised algorithms are needed.

MAPF is quite a broad problem and there are many versions and case specific variants, but in this thesis, we will be dealing with what is defined as the classical MAPF problem (defined in [7]). Classical MAPF problem with n agents has a tuple $\langle G, s, t\rangle$ as input. G is the representation of an undirected graph and contains a list of all vertices V and a second list of edges E connecting them, with obstacles represented by the absence of the previous two, visualisation in Figure 2.1a. Both $s = [s_1, s_2, ..., s_n]$ and $t = [t_1, t_2, ..., t_n]$ are lists of vertices from graph, where s_i represent starting position of i-th agent and t_i its destination. When an agent moves between two nodes via a connecting edge, the action takes up one discreet timestep. One quite significant deviation from the classical MAPF problem is that turning in place is an independent action that takes time to complete, and rotations have their vertices, as visualised in 2.1b.



(a) : Graph of classical MAPF problem **(b) :** Graph with rotations

Figure 2.1: Visualization of MAPF graphs

2.1.1 Collisions and k-robustness

The collisions MAPF solvers are trying to prevent are forbidden or impossible interactions between two or more agents, and they depend on the capabilities of what the agents are abstractions for. Depending on the configurations of the MAPF problem, multiple types of conflicts can be present. Conflicts recognised by our planner were node conflict which happens when multiple agents occupy the same spot at the same timestep, or swap conflict where the problem is that agents swap positions across the same edge (more exist and can be seen in [7]). Plan is a valid solution if it has none of the defined collisions for the task.

In addition to having different definitions of collisions, solutions to the MAPF problem can also be **k-robust**. As defined, a k-robust plan has enough reserves and buffers for agents in such a way that each agent can be delayed by at least k timesteps without causing a collision. Where k can be any natural number or a zero. Every valid solution is at least 0-robust. A term that is closely connected to k-robustness is collisions in time t. It is an interaction between two agents, that would result in conflict if one of them were delayed by t timesteps. In a k-robust plan, no t-collisions would happen for $t \in \langle 0, k \rangle$.

2.1.2 Metrics of MAPF plans

To measure solutions to MAPF task, there are two main metrics, sum of costs (SOC) and makespan. Solution for n agents contains a set of paths

$p_1, p_2, ..., p_n$ where each path p_i is a sequence of coordinates i-th agent held each timestep from start in s_i until it definitely reached (meaning that it will not move from it within the scope of the plan) its destination t_i. Path of an agent also contains timesteps where it waits, except those after reaching the target. Sum of costs is calculated as $SOC = \sum_{i=1}^{n} |p_i|$, whereas makespan is $makespan = \max |p_i|$

Every solvable classical MAPF task has an optimal solution, where each agent travels the fastest possible route while avoiding colliding with others (this is most likely different from to fastest path if it was alone in the plan). But finding this optimal solution is proven to be NP-hard (ref. [10]) and potentially very computationally expensive. To lower the solving time to an acceptable amount, several algorithms (such as ECBS and LNS used in this thesis) instead provide a sub-optimal solution.

Conected to sub-optimal solutions is another metric, suboptimality. Sub-optimality is the cost of the current plan, can be both SOC or makespan, divided by the cost, in the same metric, of the optimal plan. Here it will be used mostly in theory, since we rarely have optimal plans to compare to Some algorithms can guarantee bounded suboptimality. This means that the algorithm will produce a plan, the actual suboptimality of which is in the worst case same as the one it was bounded with.

2.2 Hypothesis

The initial motivation for work done in this thesis was to investigate how would MAPF plans, generated under a presumption of ideal circumstances, perform in an environment with errors and imprecisions in movement. These imprecisions, although of small impact individually, could in theory, cascade and compound to create larger issues and start interfering with synchronicity of plan. Main hypothesis was that optimal plans or ones close to them would be most vulnerable, which could cause slightly suboptimal plans to outperform them in such conditions. Reason for this thinking is an assumption that very slightly imperfect plans will have some gaps between moving agents, some organic waiting and that these slight imperfections would cushion execution errors of agents, instead of letting them cascade.

2.3 Means of experimentation

To test the hypothesis, a large number of experiments would be needed would. Also, although it is desirable that the error in movement of agents is realistic, it needs to be controllable, and while random, to some extent predictable. For this reason simulated environment was chosen instead of a laboratory with physical robots.

The environment that the simulation is emulating is that of warehouse robots. These robots have as few sensors as possible and minimal processing power to keep them cheap and deployable in large numbers. Individually they cannot perceive the outside world and instead rely on some localisation system to tell them where they are and some control system to tell them what to do. This makes these robots an ideal match for agents in MAPF. The simulated location system is a set of location marks placed in a grid (in the case of the simulated environment, with a distance of one meter), where robots can get the coordinates of the mark they reach plus visible offset and their heading. Although in most generated plans, movement is quite arbitrary, a requirement was placed that robots must reach their final destination very precisely (with a very small error margin). This is to represent that robots have to be in place to pick up specific packet or enter correctly a charging station.

The simulator as it was initially, lacked the option to simulate errors in movement. This needed to be added via

Chapter 3

Simulator

3.1 State

Simulator was developed from a program for controlling physical robots, by the Intelligent and Mobile Robotics Group at the Czech Institute of Informatics, Robotics, and Cybernetics, under CTU in Prague. Later, an option to run simulated robots within a completely simulated environment was added. The initial motivation was to run scenarios with the software of robots remote from the laboratory. This also made it possible to run scenarios in more complex environments and with more robots than was possible in the laboratory. At the current date, the simulator can run simulated scenarios with multiple robots executing continuous imprecise movement. The transition from real to virtual was necessary to enable running scenarios whose complexity would allow us to properly test the capabilities of MAPF planners, and to completely control how errors in movement happen, instead of having to rely on unpredictable real-world ones to occur.

Input given to the simulator consists of a map file, detailing the environment of the simulation, a plan file, in which it is described what moves robots should perform, and configuration files, where we define constants and parameters. Output is a file containing analytical data of a run. The simulator is at its core a multi-thread program with master-slave architecture. The role of the "master" is fulfilled by the server which oversees and coordinates robots, which are in the role of the "slave". The software of each simulated robot has two parts, controller and simulated movement, where the controller receives a target to reach from the server and gives out inputs to motors, which are

then used by simulated movement to move the robot in the environment. The movement itself is handled by motion models, whose parameters allow to specify what imprecisions can happen and what causes them during the robot's attempt to move.

3.1.1 Server

The main job of the server is to handle input and output and to coordinate individual robots. The coordination is achieved by the server sending only simple and short movement tasks to robots and waiting for their report of completion. The simple movement tasks should be move by just one grid square and/or rotation in place. Since robots can not coordinate themself, sending longer tasks would mean losing coordination. In case the robot cannot proceed in its path, because another robot is blocking it, the server will simply not issue further orders to that robot, making it wait until its path is clear.

To decide which actions of individual robots need to be completed in exact order and when robots can go at their own pace, the server makes use of an Action Dependency Graph (ADG). ADG is a directed graph created at the start of each run from the input plan, where each node represents a move command that should be issued to a specific robot. These commands should not skip positions. The edges between them represent dependencies between commands, and they play a critical role in the functioning of ADG. When in graph one action depends on another one, it means that the depended on action needs to be reported as finished before the orders to execute the dependee can be issued. Edges are added to the graph in two waves at the time of its creation. First are added edges connecting the paths of individual agents, these are directed from start to finish, connecting each node to the next one. Second are edges representing dependencies between agents.

Dependencies between agents arise whenever two paths of two agents contain an order to enter the same position, regardless of the expected time of entry. When this happens, a connection will be drawn between the paths of the agents, with regard to the k-robustness of ADG, with the edge leading to the node that is supposed to be reached second, according to plan. The current simulator has 1-robust ADG, which means that when drawing edges between paths of agents, it leads from the first command that makes the first robot leave the contested position to the one in which the second robot enters the position, visualised in the Figure 3.1. If a node has edges leading to it, its corresponding command cannot be issued, and if it is currently the first node in the agent's path, said agent must wait. When any robot reports it has finished a command, its corresponding node in ADG is removed,

together with all connections leading from it, so if there was a dependency on this action, it can proceed. ADG, as implemented in the simulator, has no predictions or checks and simply assumes that the plan is 1-robust and valid. If a plan does not fulfil that, ADG can enter into a deadlock, where a group of two or more forms a cycle in the graph, which leads to the problem that no node belonging to this group can be removed, leaving them stuck forever.

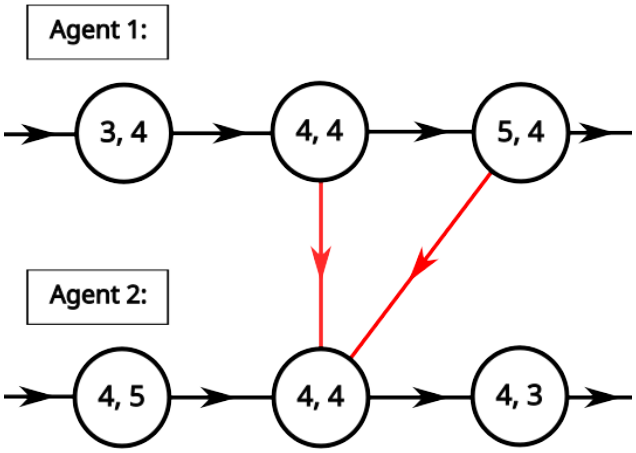


Figure 3.1: Black lines are the dependencies in paths of the agents, red ones are inter-agent dependencies

The server communicates with robots only using buffered messages. It sends commands containing information that the robot with the target ID should move from A to B. In return, the server receives acknowledgments of completion of tasks. These messages contain the ID of the robot the message was relevant to, the coordinates of the goal, and the distance traveled in reaching it, which is accumulated in the robot's overall travelled distance. Acknowledgements can also be used to report critical errors from agents, via sending large negative travelled distance. In addition, the server also tracks the execution and wait times of robots, both in individual sections and their sums throughout a run. Execution time is time sever logged between sending a command and receiving acknowledgement back and wait time is between receiving and sending (if a message gets delayed in a buffer of the server, the delay is counted to wait).

Finally, the server takes responsibility for writing outputs. In the current build, output can have three forms, either error output, one for a multi-agent run, or one for only one agent. All of them take the form of a JSON file. Error output contains info only on α parameters (more about them in Section 3.2 of movement and number of agents. Single and multi-agent output files both contain information based on execution, wait time, and

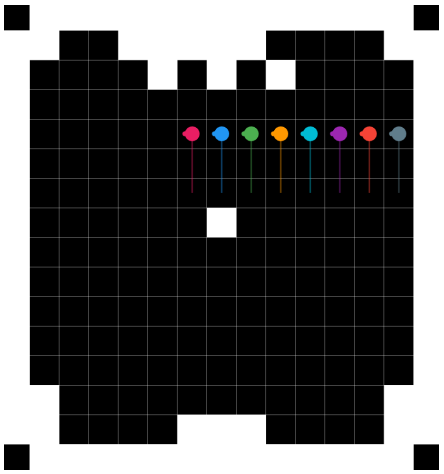
distance traveled, together with information on α-s. The output for a single agent contains detailed information on individual moves the agent has done, together with their sums for the entire run. On the other hand for multi-agent run, information is more focused on finding averages and extremes in values.

Plans and maps

To understand the workings of the simulator, and to run it successfully, it is important to understand the main inputs. These are the plan that the simulator should execute and the map that describes the environment in which the plan plays out.

```
type octile
height 14
width 13
map
T....TTT....T
T...........T
.............
.............
.............
.............
.............
......T......
.............
.............
.............
.............
....T.T.T....
T..TTTTT....T
```

(a) : Map in a text file



(b) : Map in the simulator

Figure 3.2: Representations of simple small map

The map is supplied in a text file, a simple example of which can be seen above in 3.2a. The first four lines are parameters, with the only relevant ones being height and width. These are followed by a block of text where each symbol expresses the status of a field, "." is used for free spaces and "T" for obstacles, important to note is that the simulator flips it upside down, as can be seen by comparing map in file to how simulator renders it in 3.2b. The environment in the map represents a usable area, on loading, edges of obstacles are automatically added around it. By default, the shape of the map is rectangular, but any other shape can be created using obstacles.

```
instance=../instances/sample.txt
agents=8
map_file=lab.map
solver=PIBT
solved=1
soc=3573
lb_soc=3163
makespan=4
lb_makespan=4
comp_time=4
starts=(5,10),(6,10),(7,10),(8,10),(9,10),(10,10),(11,10),(12,10),
goals=(5,10),(6,10),(7,10),(8,10),(9,10),(10,10),(11,10),(12,10),
solution=
0:(5,10),(6,10),(7,10),(8,10),(9,10),(10,10),(11,10),(12,10),
1:(5,9),(6,9),(7,9),(8,9),(9,9),(10,9),(11,9),(12,9),
2:(5,8),(6,8),(7,8),(8,8),(9,8),(10,8),(11,8),(12,8),
3:(5,9),(6,9),(7,9),(8,9),(9,9),(10,9),(11,9),(12,9),
4:(5,10),(6,10),(7,10),(8,10),(9,10),(10,10),(11,10),(12,10),
```

Figure 3.3: Representation of plan without rotations

The plan contains the output of a MAPF solver in a specific format. It stores information on what map the plan was made, how many agents are in it, and what are the SOC and Makespan metrics. It also contains a table, which specifies coordinates, where each agent should be, by time-step (for clarification, lines are for time-steps, columns for agents). Coordinates are positions on the x and y axis, with an option of specifying rotation at the end of move. Although rotations are optional, based on experience, it is strongly recommended to include them, since they help with smoother running for higher agent counts (The project folder includes a python script that can add rotations, though it is recommended that the plan contains free timesteps for turning in place)

3.1.2 Robots and their movement in simulator

Even though plans for robots are discrete and the environment they operate in is a grid, robots move continuously. Real, physical robots simply had to, and since the main intent of simulation is to create realistic real scenarios, robots in simulation also move continuously Hardware wise, robots are thought of as having only one, centered axle, with independently powered wheels, which allows them to turn in place.

When it comes to how robots move, this is the part that has undergone probably the largest development in the time scope of this thesis. At the start, simulated robots moved precisely as instructed. The movement was closer to spread out deterministic move, where just instead of a single jump and then waiting, the robot would move between its initial and goal position. The only thing that was not exact was the time it would take for the robot to act out the action (for example if it needed to turn, move would take longer).

But for the current task of testing how deterministic plans translate to a realistic environment, the previous approach would be insufficient. Outside of some minor, in the end predictable, delays due to turning, no imprecisions or errors in movement, on which we could see the robustness of the algorithm, would happen. This led to the implementation of motion models (see more in Section 3.2), which create imprecision based on current movement and given parameters. This led to the introduction of the perceived position of the robot, alongside the real position of the robot in the environment and that of the current goal, which was already present. The robot decides how it will move based on perceived coordinates, instead of real ones. This is because if the robot was looking at the real coordinates if moved off course due to inaccurate movement, it would simply correct and in large part mitigate the introduced error. Instead robot believes it is moving in an ideal way and the motion model introduces noise to this movement and then applies it to change real positions.

While simulating completely ideal conditions, perceived and real coordinates would not diverge from each other, but in reality, robots would start accumulating error. Without correction, the positional error could spiral to such an extent, that the robot will not be able to follow given instructions, within a relatively small amount of moves (depends on the motion model settings, but with current test parameters, it would be around 3-10 moves). To prevent this, in our simulated environment, there are correction points. These are located at the center of each grid-square and when the robot reaches within a certain radius around the marker, the robot will correct its perceived position. How exactly this happens can change in some minor details, but in general, this correction means overwriting perceived coordinates with real ones. This is an abstraction to a system that exists in the laboratory for physical robots, where there is a grid of markers that robots can read and find their position.

The currently up-to-date robot software is split into two parts running in independent threads. There is the controller, which decides how the robot want to act, then the simulated movement, which acts out the decisions of the controller using one of the motion models. When the robot has target coordinates, it has three movement options to reach them. It can either go in a straight line, turn in place, or travel along a curve. When the robot controller has a new target, it first calculates the angle between its current perceived heading and that needed to reach the target. It then compares the angle to *turn threshold*, a constant, if the angle is less than the threshold, it will use the curved movement, if not, the robot controller will issue commands to stop, turn in place, and then move in direct line. A very situational move is reversing, which is allowed for only very short adjusting moves. The controller runs in cycles with a set frequency (to give an example, the current default is 100Hz). During each iteration, it gives out current instructions and then

waits for its period, so instructions can be carried out.

In the initial version of the simulator, the robot had to reach each goal exactly with minuscule tolerance, for a move to be considered completed. When there were no imprecisions in movement, this worked without problem, since the robot would reach it exactly without corrections. However motion models were introduced and with them imprecise movement, this requirement began to slow robots down considerably. This was because now it was rare for a robot to reach the goal on the first try and once it reached within the correction radius and realised it was heading even slightly of the goal, it had to adjust. Adjusting would mean it had to stop, turn towards goal, often at angles close to 90°, do small adjusting move and only then could the goal be marked as done. But still, the robot once again needed to turn, since the move it made to adjust, almost certainly did not put it in the heading, it desired to continue in.

This was the motivation for the introduction of the "align" feature. With this feature, aligning as described above can be switched on or off for each robot. If disabled, robots will mark moves as completed when it has reached within correction radius around their goal. This means that at the same time aligning robot would realise that it must adjust, non-aligning one would be already marking the move as complete. The reason for making aligning switchable instead of disabling it altogether was that one of the formulated requirements for the plan to be successfully executed is to reach its final destination on the path precisely. Aligning can be switched by specifying it in an extended command form server, the goal for that command will already be affected. In the current state, when the server issues the last order to any robot, it will specify in the message that this robot should reach that goal with aligning enabled.

3.2 Motion Models

To test the robustness of plans, and how they would handle if something interfered with ideal acting out of a plan, it was at first needed to have something that could "go wrong" in a controlled and to some extent reliable, but random way. The way to do that was to introduce noise to the movement of robots. This noise can be viewed as an abstraction of minor problems, such as a robot driving over a wet floor, having one motor low on battery and so on. One way to get this noise into movement is by using motion models. In general, motion models are algorithms that take in the current position of the agent, movement instructions, and parameters and output a new position.

Currently, the simulator implements two motion models, velocity and odometric. When they move robots in the environment, they do not move or rotate them continuously, but in small jumps, though these jumps are so small and done in such high frequency, that they very well imitate continuous movement (default period is set to 5 milliseconds). Both models are parameterised by α parameters. The parameters modify what movement causes what error and to what extent. How many parameters a model has and what each individual does differs but for both implemented models, if all parameters were set to zero, it would disable the noise.

3.2.1 Velocity model

The Premise of the velocity model is quite simple, it takes the speeds at which the agent wants to move to reach the target and slightly modifies them [8]. To visualise this, one can imagine a car doing a curved movement and ending up slightly off target, because the driver turned the steering wheel a few degrees more than he intended or travelled at a bit lower speed. The velocity model as implemented takes input from the robot's controller in the form of desired rotation speed for left and right wheels respectively, marked v_l and v_r. Output is in the form of an updated position of a robot by the enacted movement. In the case of implementation in the simulator, this means updating both real and perceived coordinates of a robot, though the update to perceived does not have any noise or errors in it.

Since the inputs are in the form of rotational speeds, v_l and v_r for the left and right wheels respectively, they need to be converted to the forward and rotational speeds the robot's controller wanted to achieve with them. This is done in following formulas:

$$\hat{v} = \frac{(v_r + v_l)}{2} \tag{3.1}$$

$$\hat{\omega} = \frac{(v_r - v_l)}{len_{axle}} \tag{3.2}$$

The $\hat{x}$ version of variables indicates that it contains value of x as perceived by robot, in this case how fast it wanted to go in $\hat{v}$ and $\hat{\omega}$ marks the turning rate of a robot in radians. Variable len_{axle} is the distance between the wheels of our robot.

Once these are calculated, noise can be introduced into the speeds that will be used for real movement, though the ideal values still need to be preserved to update perceived coordinates. To get random noise, samples from a normal distribution are used. The sample function returns a random

value based on its probability in a normal distribution. In this case, the normal distributions have a mean equal to zero and deviation set in bracket by multiplying intended speeds by relevant α parameters.

$$v = \hat{v} + sample(\alpha_1 * |\hat{v}| + \alpha_2 * |\hat{\omega}|)/\sqrt{dt}$$
$$\omega = \hat{\omega} + sample(\alpha_3 * |\hat{v}| + \alpha_4 * |\hat{\omega}|)/\sqrt{dt}$$
$$\gamma = sample(\alpha_5 * |\hat{v}| + \alpha_6 * |\hat{\omega}|)/\sqrt{dt}$$

The α-s that appear in the pseudocode above are parameters of the model. To explain them further, each type of error can be caused by both translational or rotational movement, even α-s determine the influence of rotation on the error and odd ones determine error caused by translation. During experimentation, relatively low even α-s were used and instead focus was on the odd ones. With the imprecision created by α_1 and/or α_2, the robot would precisely head at its target, but the error will be in how far it travels to it, ending up either short of the desired position. To beware, if we have significant α_2 and the robot wants to only turn in place, it will cause him to slide, which can cause problems. Error from α_3 and/or α_4, will result in robots travelling a precise distance, but with an imprecise heading, causing the robot to land in a semicircle centred around its target. Finally, the last two α parameters would not at all interfere with the robot until it reaches its target, but after that, they will modify the heading it ends up with, whereas without them the robot would be looking exactly in the direction of movement it acted out. In Figure 3.4, we can see the effect of the velocity model on the curved movement of the agent. In the first subfigure, α-s are moderate [8], in the second α_1 and α_2 are higher at the expense of α_3 and α_4 and in the third subfigure it is reversed.

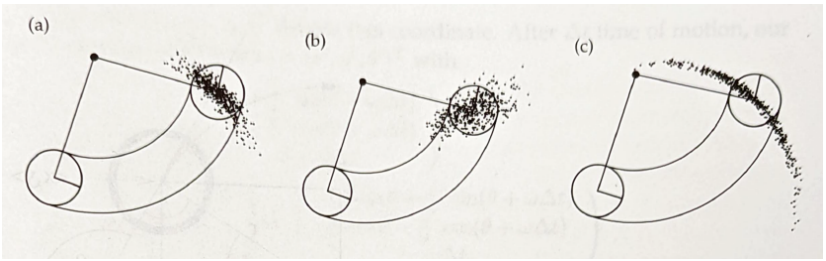


Figure 3.4: Sampling of 500 moves with velocity model, with different parameters taken from [8], page 125, Figure 5.4

Once modified speeds are calculated, they can be used to calculate new positions. Positions need to be calculated separately, so the following code will be run twice, once with real values and once with perceived ones. By default any move with a velocity model is on a curve, but the formula for calculating the curve is undefined for ω equal to zero and is unstable near zero, so if ω is below turning threshold ϵ, it will be ignored and the robot will

move in a straight line (default value of ϵ is 0.0001). Straight movement is simply travelled distance in the initial heading, but with the curved move, the radius of the curve needs to be calculated first r_c and coordinates of its centre, x_c and y_c.

Algorithm 1 Resulting destination

if $(|w| < \epsilon)$ **then**
  $x_{new} = x + v * cos(\theta) * dt$
  $y_{new} = y + v * sin(\theta) * dt$
  $\theta_{new} = \theta$
else
  $r_c = v/\omega$
  $x_c = x - r_c * sin(\theta)$
  $y_c = y + r_c * cos(\theta)$
  $x_{new} = (cos(\omega * dt) * (x - x_c) - sin(\omega * dt) * (y - y_c)) + x_c$
  $y_{new} = (sin(\omega * dt) * (x - x_c) + cos(\omega * dt) * (y - y_c)) + y_c$
  $\theta_{new} = \theta + \omega * dt + \gamma * dt$
end if

At current state velocity model is usable but there are concerns if it performs exactly as it should. In an earlier implementation of the velocity model, the size of the error was dependent on the period of movement. Although that problem has been corrected, the current implementation of the velocity model is not considered altogether reliable.

3.2.2 Odometric model

The odometric model works by calculating where the agent will end up with perfect move, and then retroactively modifying that destination. Compared to the velocity model, this type of error is more abstract, but also more predictable. It needs both old and ideal new positions, then it takes the difference between them and splits it into simple three moves, two rotations and one translation. Then the imprecisions are introduced to these three moves and they are then put back together to get a new position.

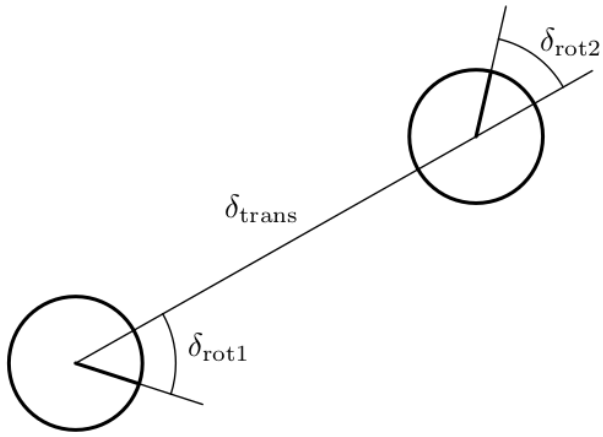


Figure 3.5: Visualisation of rotations and translations

Firstly, ideal destination needs to be calculated. For that, some functions shown in the velocity model will be used. Equations 3.1 and 3.2 to obtain $\hat{v}$ and $\hat{\omega}$. These values are then input into Algorithm 1, to obtain both new perceived location, but more importantly to the odometric model results of ideal move, when perceived speeds are used to update real positions. From difference between old real coordinates, x_{old}, y_{old} and θ_{old}, and newly obtained ideal ones $\hat{x}_{new}$, $\hat{y}_{new}$ and $\hat{\omega}_{new}$, the three moves can be calculated as shown bellow.

$$\begin{aligned}
\hat{\delta}_{rot1} &= atan2\left(\frac{\hat{y}_{new} - y_{old}}{\hat{x}_{new} - x_{old}}\right) - \theta_{old} \\
\hat{\delta}_{trans} &= \sqrt{(\hat{y}_{new} - y_{old})^2 + (\hat{x}_{new} - x_{old})^2} \\
\hat{\delta}_{rot2} &= \hat{\theta}_{new} - \theta_{old} - \delta_{rot1}
\end{aligned}$$

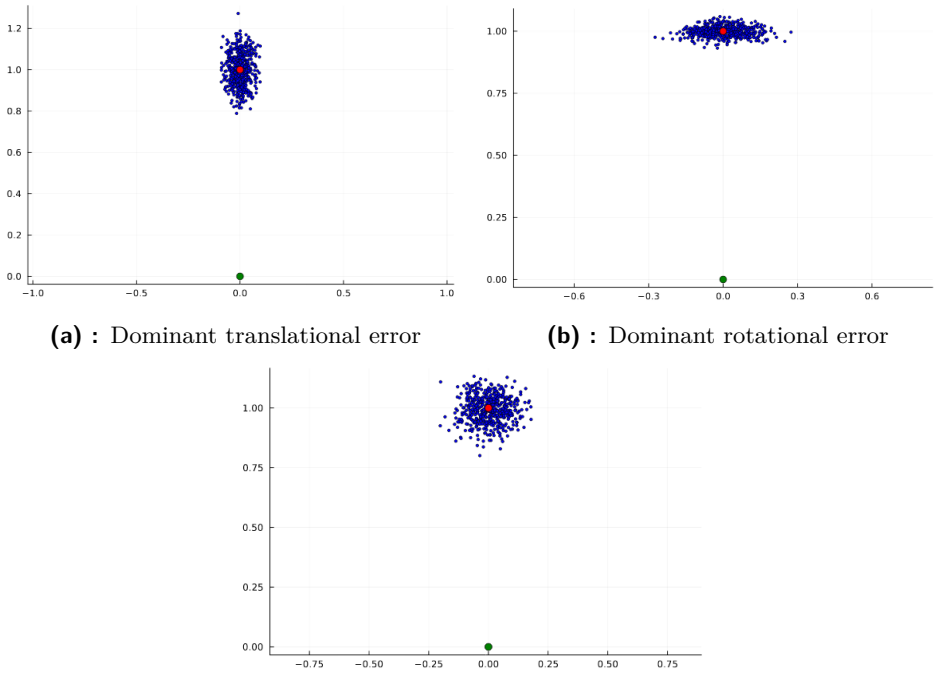
atan2 that can be seen above is special variant of arcus-tangens, defined:

$$atan2(y,x) = \begin{cases}
arctan\frac{y}{x} & \text{if } x > 0 \\
arctan\frac{y}{x} + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\
arctan\frac{y}{x} - \pi & \text{if } x < 0 \text{ and } y < 0 \\
+\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\
-\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\
undefined & \text{if } x = 0 \text{ and } y = 0
\end{cases}$$

Now with the three moves obtained, noise will be introduced to them in 2 bellow. Similar to how we introduced error in the velocity model, the sample of normal distribution but only four α parameters.

Algorithm 2 New v, ω, γ

$$\delta_{rot1} = \hat{\delta}_{rot1} + sample(\alpha_1 * |\hat{\delta}_{rot1}| + \alpha_2 * |\hat{\delta}_{trans}|)$$
$$\delta_{trans} = \hat{\delta}_{trans} + sample(\alpha_3 * |\hat{\delta}_{trans}| + \alpha_4 * (|\hat{\delta}_{rot1}| + |\hat{\delta}_{rot2}|))$$
$$\delta_{rot2} = \hat{\delta}_{rot1} + sample(\alpha_1 * |\hat{\delta}_{rot2}| + \alpha_2 * |\hat{\delta}_{trans}|)$$



(a) : Dominant translational error

(b) : Dominant rotational error

(c) : Even error

Figure 3.6: Results of odometric movement with different types of error

The parameter α_1 represents rotational error caused by rotation and $\alpha 2$ is rotational error caused by translation. Together they both introduce noise to the direction of a move and to final heading, causing robots to fall in a semi-circle centred on the target. On the same note, α_3 and α_4 represent error in translative movement caused by moving or turning respectively. Error in translation means the robot does not travel exactly the travelled distance, either too far or back. It is advised to set α_4 very low in comparison to others or to zero, since if set high, it will cause the robot to rock back and forth as it is turning in place, making it very difficult for the robot's controller to hit targets or even stay in a radius of precision point. In Figure 3.6, can be seen results of move from green point, where agent is oriented to the right, to the red point where he is oriented up. Blue points represent results of 1000 moves with different parameters, 3.6a has high α_3 at the expense of α_1 and α_2, 3.6b is reverse and 3.6c has all values rougly even.

Once modified moves are completed, new real coordinates can be reconstructed from them, as shown in Algorithm 3. Even though odometric model

could act out entire move of robot in one action, it is still being used in very small bits to create continuous movement.

Algorithm 3 assembling final position

$$x_{new} = x + \delta_{trans} * cos(\theta + \delta_{rot1})$$
$$y_{new} = y + \delta_{trans} * sin(\theta + \delta_{rot1})$$
$$\theta_{new} = \theta + \delta_{rot1} + \delta_{rot2}$$

At present time, the odometric model is the default motion model of the simulator, though this can be quite changed by simply replacing the used sub-class of robots. The decision to switch to the odometric model was made after issues with the velocity model and because its configuration has more predictable results on tests.

3.3 Configuration of simulator

Large parts of the simulator's behaviour can be changed. While more complex changes, like switching motion models or switching how aligning is handled need to be done in source code, a plethora of constants and configurations are defined in an easily modifiable file. File *robot.json*, located *warehouse_demo/config*, makes it possible to change the physical properties of robots, precision it needs to achieve, α parameters of motion models (see Section 3.2) and so on. Here is a complete list of available options:

- **dt**: Period of movement cycle, how long in seconds does one movement segment take. needs to be low if robots are to move continuously.
- **precise_loc_radius**: Radius of correction area for robots in meters.
- **axle_length**: Distance between robot's wheels.
- **wheel_radius**: Radius of robot's wheels, affects forward and turning speed .
- **omega_max**: Maximum rotation speed of wheel axle.
- **steering_authority**: What percent of omega_max can be used for steering. Affects movement error and if high it may cause robots to swing from side to side.
- **control_frequency**: Frequency of robot's control cycle, how often can robot's controller change orders.

- **realign_turn_thresh**: When the radian difference between the perceived heading and that needed to reach the destination is greater than this constant, the robot will stop and turn instead of driving on a curve (see at 3.2.1).
- **precision_trans**: At what distance from target is robot's move considered reached exactly-
- **precision_ang**: Turn is considered done when perceived heading is less than this value in radians from desired one.
- α **parameters**: These are $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6$, and they are used by velocity models, but they do not have the same effect in both. Values in them are usually really small, they should never go to whole numbers, or even above 0.001.

Chapter 4

MAPF planners

The MAPF planners are programs implementing algorithms for solving MAPF problems by creating solutions (= plans) that can be then used by the simulator. Since the main goal of this thesis is to examine robustness, it is important to at least briefly look into how these algorithms work, how they are implemented and what changes were made to them.

Two algorithms were used, Enhanced Conflict Based Search (ECBS) and Large Neighbourhood Search (LNS). Both were chosen because they fulfilled the criteria of having publicly available implementations and both could be modified to produce 1-robust plans with actions as independent turns. Furthermore, each had their own added benefit. ECBS can provide a guarantee of subotimalty or even optimality, and while LNS can not do that, it is due to a good combination of single agent pathfinding and heuristics [2] much faster at finding solutions.

4.1 Enhanced Conflict based Search

ECBS (Enhanced Conflict Based Search) is a variant of CBS (Conflict Based Search). CBS itself is a two-level algorithm. On the upper level, it performs a best-first search over a binary tree, where each node N contains a set of constraints, and the value of a node is the SOC of a solution that was created with these constraints. Solutions are created on the lower level using A* algorithm for a specific node N and its constraints. Each agent is planned

```
   Input: MAPF instance
 1 Root.constraints = ∅
 2 Root.solution = find individual paths by the low level()
 3 Root.cost = SIC(Root.solution)
 4 insert Root to OPEN
 5 while OPEN not empty do
 6     P ← best node from OPEN // lowest solution cost
 7     Validate the paths in P until a conflict occurs.
 8     if P has no conflict then
 9         return P.solution // P is goal
10     C ← first conflict (a_i, a_j, v, t) in P
11     if shouldMerge(a_i, a_j) // Optional, MA-CBS only then
12         a_{i,j} = merge(a_i, a_j, v, t)
13         Update P.constraints(external constraints).
14         Update P.solution by invoking low level(a_{i,j})
15         Update P.cost
16         if P.cost < ∞ // A solution was found then
17             Insert P to OPEN
18         continue // go back to the while statement
19     foreach agent a_i in C do
20         A ← new node
21         A.constraints ← P.constraints + (a_i, v, t)
22         A.solution ← P.solution
23         Update A.solution by invoking low level(a_i)
24         A.cost = SIC(A.solution)
25         if A.cost < ∞ // A solution was found then
26             Insert A to OPEN
```

Figure 4.1: Psedocode of CBS, taken from [6] (Section 4.2.6, Algorithm 2)

individually, and cannot see or avoid paths of other agents directly, however, in addition to avoiding obstacles on the map, it is subject to constraints related to it, taken from node N. Due to the properties of the A* algorithm, any solution produced on the lower level is optimal under placed constraints, though it is in most cases not a valid solution in terms of a MAPF problem.

CBS starts with only a single node in the upper tree that has no constraints. In every iteration of the upper level, a node with the lowest value of SOC is selected and its solution is checked on the lower level, if it contains no collisions, that solution is valid and a goal so the algorithm terminates. But once the first collision is discovered, checking stops and the node branches into two new nodes, where each of the two branches resolves the new collision by introducing a constraint blocking one of the agents from entering the conflicted point. Then SOCs of the new nodes are calculated (without dealing with collisions) and they are put into the upper tree as followers of the node that caused them (ref.: [6]. The whole run of the CBS algorithm can be seen in pseudocode in Figure 4.1. Because of the optimal paths on the lower level with a combination of the best first search on the upper level, CBS is guaranteed to find the optimal solution [3].

ECBS builds upon CBS with the introduction of suboptimality variable w. Suboptimality is specified by a number greater or equal to one (if $w = 0$, ECBS behaves exactly as CBS did). In general, works like CBS, except both

lower and upper layers use focal search to achieve bounded relaxation. On a lower level, it works by first using A* to find each agent n, an optimal path with cost $f(best_n)$, where f is a function that returns a cost of a path in length. Then, using relaxed A* (in detail explained in Section 3.2 of [3]) it finds all paths p for this agent which fulfil $f(p) \leq w * f(best_n)$, and selects the one minimising the amount of collisions. This does not improve the lower part, but drastically helps the upper one with branching. Concerning the upper part, nodes now contain also the lower bound of their cost, lb. The lower bound of node N is calculated $lb(N) = \sum f(best_n)$ across all agents. Similar to how this is done in the lower level, all nodes that fulfil $cost(N) \leq w * lb(N_{best}$, are considered for expansion and are instead compared based on the number of collisions they contain.

Since the selected nodes are selected based on lower bound instead of cost, ECBS can guarantee that overall SOC will not be worse than suboptimality times the cost of the optimal solution. Main problem of baseline CBS is that its number of nodes to examine can spiral quickly out of control when confronted with a higher density of agents, which can lead it to unbearable long computational times. ECBS drastically improves upon it in this regard, even though as mentioned in Section 5, long computational times are still a problem.

For this thesis, solver MAPF-IR [4] was used. First, it was modified to produce k+1 robust plans, by simply modifying collision detection. Modification to get rotations as independent actions was much more complex and so was provided to me by Ing. David Zahrádka from CIIRC.

4.2 Large Neighbourhood Search

In this project we made use of Large Neighbourhood Search (LNS) with Safe Interval Path Planning (SIPP) as a single agent path-finding algorithm. LNS is a metaheuristic algorithm that works by finding an initial solution and gradually improving upon it, by the repeating cycle of destroying and repairing [2]. In each iteration we have an input solution, LNS finds, based on the destroy operator, a suitable part of it, called a neighbourhood, destroys it and then tries to repair it while keeping the rest of the solution unchanged. If the new repaired solution is better than the previous one, it is accepted, otherwise, all changes are reverted. There are many ways to define a neighbourhood, but in used implementation, they are a combination of paths of agents, in most plans 2. There are several ways to determine what neighbourhood gets destroyed and rebuilt, but when generating plans, a method which does this

on random was chosen. With LNS there are no optimality guarantees, instead, it simply runs for a certain amount of iterations, each one ending up with a solution that is better or in the worst case same as the previous one, so in high enough time, the local minimum will be reached.

A large part of LNS is the algorithm used to build the initial solution and reconstruct destroyed segments. That is in our case MAPF algorithm called SIPP. SIPP is an algorithm that tries to bypass the space-time approach to dealing with dynamic obstacles (ref.:[5]).n Space-time in this context means that we add another, time dimension to the environment (in this case we extend 2D environment to 3D). With this approach, agents can see each other and so they are theoretically able to avoid collisions even with single-agent pathfinding. This extra time dimension, however, can add a lot of computational complexity, and SIPP tries to work around adding an extra dimension while keeping the benefits. Each node in SIPP contains, in addition to normal information, a table of safe intervals. Initially, all nodes are entirely free, but when an agent enters into a node, the entire time it spent in it will be subtracted from the safe interval, to create a collision interval. No agent can enter a node that will be at the time of entry in collision interval, so it must either go elsewhere or wait in current node until collision interval in the desired node ends. For planning itself, SIPP uses modified A* [5], which checks if the neighbouring node is reachable (the agent can wait in current one long enough to enter the neighbour) and the waiting is counted into the travel cost. Individual agents are planned in sequence, which means that SIPP only finds the optimal solution for the first sequence that gave one, but it does not check if it is the most optimal across all possible sequences of planning robots. SIPP can easily be made to produce k-robust plans by simply extending all collision intervals by k timesteps into free intervals.

The implementation of LNS used in thesis was on by [1]. This implementation already had a selectable level of k-robustnest, so no modification was needed with that. However, independent rotations needed to be added, although this was much easier to do here than with ECBS. Each agent had a direction in which it was travelling last and if it wished to perform a move that would require turning, it that move would be made to take artificially longer in terms of timesteps to allow for a turn to take place. The problem was, that this program used a different input format, than the one from movingAI for scenarios and the output did not contain rotations (they were part of the plan, just not written into the output file). So converter between inputs and script that would add rotation to output were needed.

Chapter 5

Experimentation

Running tests with MAPF plans in a simulated environment is a critical part of the work done in this thesis. After all, almost every change to the simulator was motivated by tests. The following sections details the work that was done with the simulator in chronological order, together with reached conclusions.

5.1 Initial tests

During the early stages of work, the simulator was undergoing quite significant changes and new features were being added. So first test experiments focused on testing new features and overall assessing the capabilities of the simulator. Most noteworthy were the ones focused on how large maps with how many agents the simulator can run. This would inevitably guide further development of the simulator. It was found that simulators, both at the time of testing and now, can run extensive maps, but simulating robots is quite computationally demanding. This means that the number of robots that can be run by simulator is limited by hardware. As an example, on a notebook with a more or less average processor with 6 real cores, 25 agents were the maximum that ran truly smoothly, and 50 agents were around the limit that could be run without serious problems appearing, either due to the server being unable to keep up, or threads not synchronising well.

After the earliest tests on the simulator were done, the focus was shifted to finding what effect motion model parameters have on a run of a simulated

robot. That means how much would different settings slow the robot down, is the slow down evenly distributed throughout the run and how high can we push the α-s before robots start getting lost. Because a large number of tests would be needed, a special plan was handmade, which contained all examples of basic movement while being very compact and taking around 20-30 seconds for its single robot to finish. In this one plan robot drove around a path in the shape of a rectangle ten meters wide and one meter tall. This shape was chosen because it contains a long drive straight, two turns quickly after each other at the short sides and a long straight drive after a turn, as can be seen in Figure 5.1.

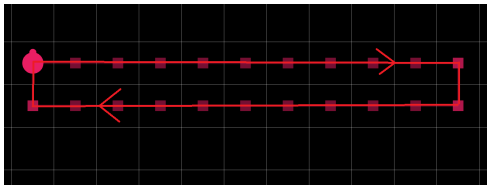
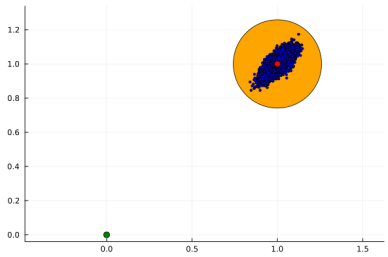


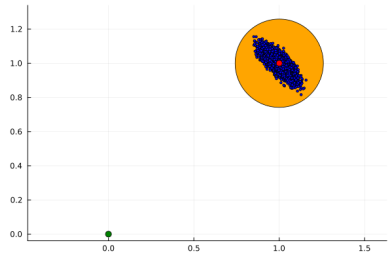
Figure 5.1: Visualisation of testing 10x1 path

This plan offered a good opportunity for studying how one robot moves, and results from experiments run on it, lead us to valuable findings, especially with finding error in our initial implementation of velocity model (for more see Section 3.2.1) and inspiring align feature. However, obtaining large volumes of data proved problematic, so 20 of these paths were combined into one plan, where they would run parallel, greatly speeding up testing.

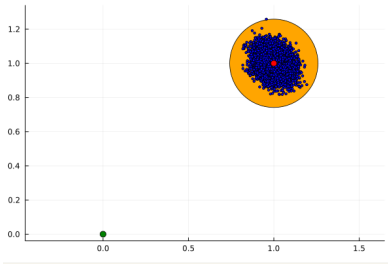
By this time, a switch was made in the used motion model from velocity to odometric model. With that, the simulator was ready to start looking for the best motion model parameters for main tests. Firstly, from the necessity of both simulation and premise of imitating a warehouse, arose the constraint that the robot must hit the align area with every move to keep reliably on track, which limits significantly the size of α parameters. In the simulator this is enforced by aligning, Secondly, we found out that imprecise movement has a lesser effect than anticipated. As a result of these two points, the idea of having three sets of parameters, with different degrees of impact, was abandoned, since these sets would not provide a meaningful difference between them. Instead, It was decided to find three sets of parameters, with each set as high as possible while still being reliable (There was not any clearly defined cut-off point, but sets were discarded if they either failed more than 10% of experiments or if plotting individual moves, agent did not hit within precision radius every out of 10 000 moves). Results were three sets, one with predominant translational error, a second with strong rotational error and a third where they are roughly even.



(a) : Dominant translational error



(b) : Dominant rotational error



(c) : Even error

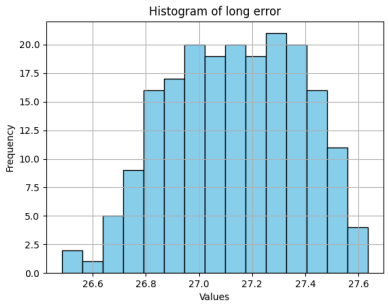
Figure 5.2: Results of odometric movement with different types of error

In Figure 5.2 we can see the result of plotting odometric movement with selected parameters. In each of the three cases, the robot started in the green point at point $(0, 0)$, facing up (in direction of y-axis, towards high values) and with the ideal move, it would transition to $(1, 1)$ and once there, turn again upwards. The red point represents where the agent would end up with the ideal movement. Blue points are each result of one out of 10 000 moves with simulated movement. The orange circle has the same radius as precision point in the simulator. Parameters of all three are $\alpha_1 = 0.0001\alpha_2 = 0.0001\alpha_3 = 0.0012$ for , $\alpha_1 = 0.0007\alpha_2 = 0.0008\alpha_3 = 0.0003$ for and $\alpha_1 = 0.0007\alpha_2 = 0.0007\alpha_3 = 0.0007$ for . The parameter α_4 was for all equal to zero, for reasons explained in Section 3.2.2. Together with these three sets, it was planned to have a fourth comparison set, where all four α-s relevant to the odometric model would be set to zero.

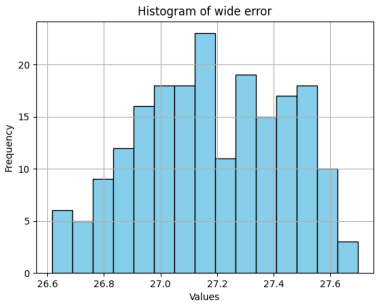
While these tests were being done, most of the work done on the simulator went on behind the scenes. While the development itself is described in Section **??**, It is important to mention here the effect it had on the experimentation process. Tests often had to wait for certain features to come online and there were large batches of measured data that had to be discarded, because they were affected by some previously undiscovered problem or untreated edge cases (often revealed by the problems they caused to the data). Many of the newly implemented features also took longer to fine-tune than expected.

5.2 Main tests

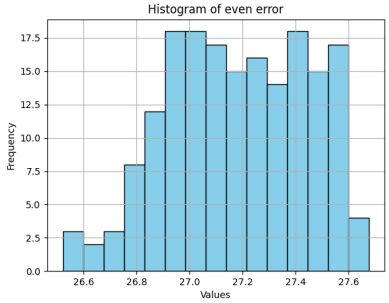
The first of the full-on tests was focused on the effect of different error configurations on plans for the robots, first the tracks, then the real plans. After several confirmation tries, it was found out that to a large extent type of error that is happening to robots has very little impact. As can be seen from Figure 5.3, while there are slight differences in probability distribution, but overall difference in maximum time cost or its average is negligible. Of course, in straight drive only transnational error would be relevant, and in reverse for turning, but in most plans these are more or less in balance. Important is only the average delay set of parameters causes, which was for all three selected sets of α parameters roughly equal. So decision was made to keep the set that had more or less even distribution of rotational and translational error, to prevent from discriminating against plans that for any reason ended up having a lot of turns or straight drives. The one with parameters $\alpha_1 = 0.0007\alpha_2 = 0.0007\alpha_3 = 0.0007$.



(a) : Dominant translational error



(b) : Dominant rotational error



(c) : Even error

Figure 5.3: Probability graph of arrival time with differently parameterised error

Once testing parameters were narrowed down to one, tests specialising in run time cost of plans could begin. The run-time cost of a plan is the sum of the times it took each robot to complete its path, so for n robots, $runtimesum =$

$\sum_{i=1}^{n} runtime_i$. With these tests, the focus is on how suboptimality and SOC affect runtime. The prototype experiments were performed with just plans from ECBS. Scenarios used were already included in repository of ECBS solver [4], and these are base on benchmarks from moving AI. There, five scenarios, with the same map and the same number of agents, would be selected such that for each, the ECBS algorithm could find at least 5 suboptimality values, that produced 5 unique plans. Each of the 25-30 plans would then be run several times (in general 5) in the simulator to get at least a bit of statistically relevant results. This is especially important for plans heavily affected by errors in movement, which are the most important ones, where the results of individual runs tend to have quite a large deviation. The whole process was done for 15, 25 and 50 agents. In the end, these served mainly to test getting from scenario to measured results, with the only established result being that running scenarios with 50 robots is not the best idea, with issues both for the ECBS solver and the simulator.

With the introduction of LNS plans, came the regular tests, again each with a set map and a number of robots. Since the testing uncovered a few deficiencies in planning software, they often had to be repeated throughout the process. Both plans for ECBS and LNS are based on the same five scenarios. The scenarios used were selected manually while making ECBS plans, using the same criteria as in the previous paragraph. This is because due to how ECBS behaves on a relatively small map, only on a few scenarios it was able to find a solution with low enough suboptimality to provide the "over-optimised" solution from the hypothesis and together with it to find four other distinct plans with a differentiable value of SOC. All of these five plans needed to be found manually. Once five valid scenarios and ECBS plans for them are found, scenarios need to be converted to the format used by LNS. The advantage of LNS is that, unlike ECBS, it can produce very high number of plans for any scenario, so finding five with a diverse number of iterations and values of SOC is not difficult.

Once these plans, now around 50 of them, have gone several times through the simulator, we start to arrive at quite a substantial dataset. The problem is that values can't be compared directly between scenarios, since some scenarios simply need longer paths than others to complete, so in the end low suboptimality plan in one might end up with a higher SOC than a high suboptimality plan for another scenario. To combine them, instead of real values of runtime cost and SOC, we will use gap values within each scenario. The gap of values is calculated from a set of values, it this case results from a scenario, where $gap_i = \frac{value_i - value_{lowest}}{value_{lowest}}$

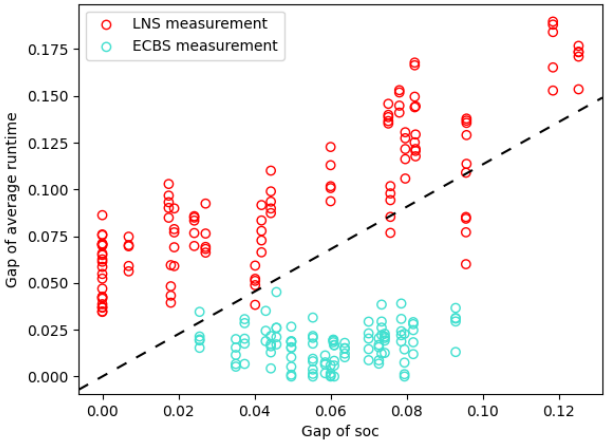
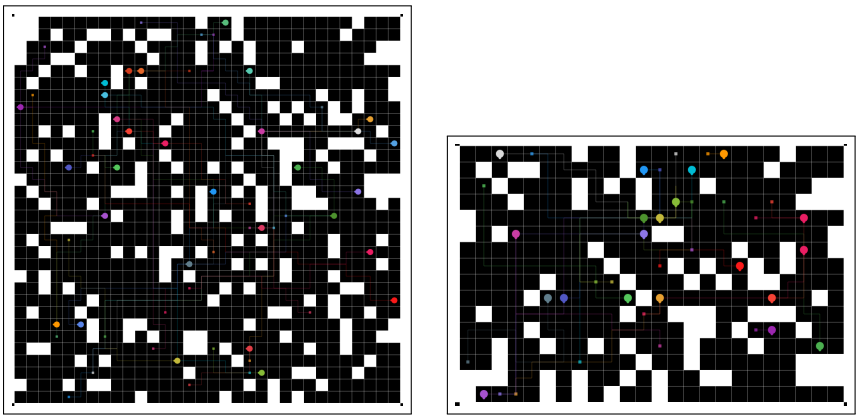


Figure 5.4: Originally promising results, with the difference between solvers.



(a) : Map 32x32 **(b) :** Small 16x24 map

Figure 5.5: The maps used for tests

The first set of results, on a random 32x32 map (Figure 5.5a) with 30 agents, looked at first look very promising. When plotted onto a graph, with few outliers, data points formed into a curve reminiscent of a hyperbole, with the bottom close to the point where it was plausible for the best suboptimality would be. But upon closer inspection, this turned out to be caused by a combination of lucky data hiding mismatch between planers, as can be seen in Figure 5.4. To describe Figure 5.4, each point has gap of runtime cost on the y-axis and SOC of plan it used on the x-axis, colour of points represents the solver that made them, the black line points from (0, 0) to average of points. The problem was caused by what different plans included in SOC calculation and primarily, as it was discovered, plans made by LNS suffered massive delays because its plans did not write desired rotation on coordinates

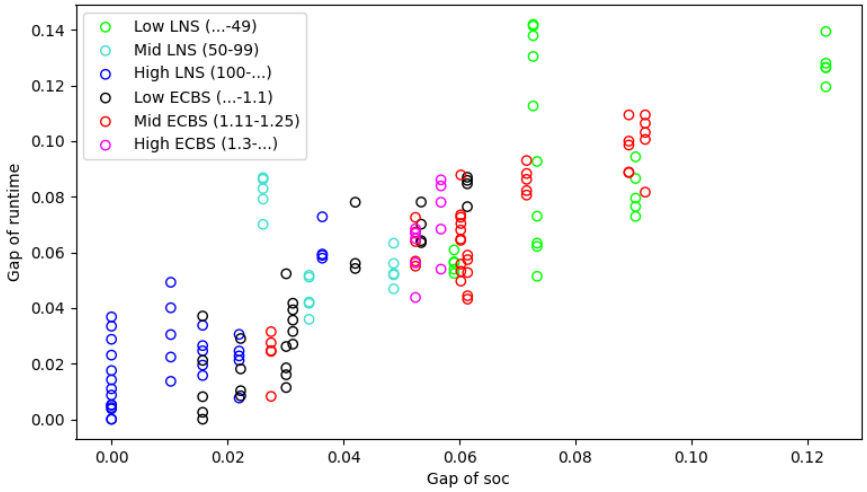


Figure 5.6: Results for 30 agents on 32x32 map

(which is not handled well when the simulator works with a larger amount of robots).

Once the problems mentioned in the previous paragraph were conclusively fixed, the graph settled into, on average, a linear relation between runtime and SOC, as can be seen on Figure 5.6. After that, some other maps and numbers of agents were tried. Most notable was with custom small map 16x24 (Figure 5.5b), with 20 robots (for perspective, that is 37,5%, with 66% robots from the previous experiment). There, it was hoped that the increased density of robots and number of interactions would produce more varied data. This materialised only to some extent, with data remaining on average linear, but it increased the amount of interesting anomalies to norm, results can be seen in 5.7.

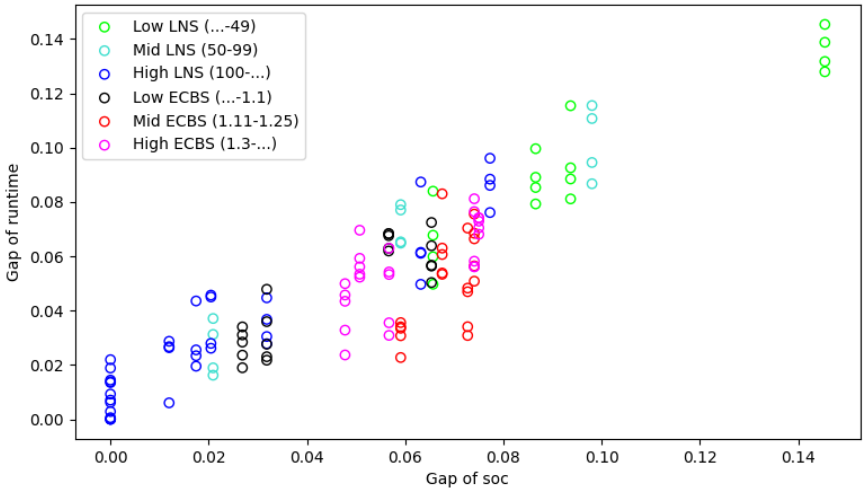


Figure 5.7: Results for 20 agents on 16x24 map.

Ond both Figure 5.7 and ¨5.6, Low/Mid/High LNS refers to numbers of iterations that were used to create plans that lead to these results, with values in brackets shoving intervals. The same goes for ECBS, just with suboptimality instead of the number of iterations.

5.3 Findings

As mentioned in the previous section, with currently measured data, so far gathered data does not point to the initial hypothesis that there is optimal suboptimality that is better than absolute optimum. Although data might be insufficient to downright disprove it. One reason for that is that ECBS did not manage to produce a truly optimal solution or one with very low suboptimality (below 1.03). Although there were LNS plans with SOC and runtime cost lower than the best ECBS produced, it is impossible to say to what extent these were suboptimal. The second reason is that obtained data have a quite high deviation and there can be questions, if enough data has been gathered, to make current results really statistically relevant.

On the other hand, the deviations in measured data provided a good opportunity to study which phenomena, when present, affect the relation between estimated and real cost. There was not enough time left at this time to construct the exact mathematical model. But what seems to have

the greatest effect are, in k-robust plan (in our case they were 1-robust), encounters that would cause $k + 1$ collisions. When these happen, due to synchronisation to plan, the second robot will need to wait for the first one to move past and if the first one is delayed, the second one will inherit this delay by waiting. If the second robot is delayed nothing happens to the first one at this point. The impact of these encounters is not even, it roughly depends on the time since the two robots last encountered each other or if they have not yet, the start of a run. The $k + 1$ collisions happen quite often in probably every plan that was run during testing, but still frequency and severity changes.

The example where this inheriting of delay starts to have very big influence, is when plan contains what will be here called a crossroad.

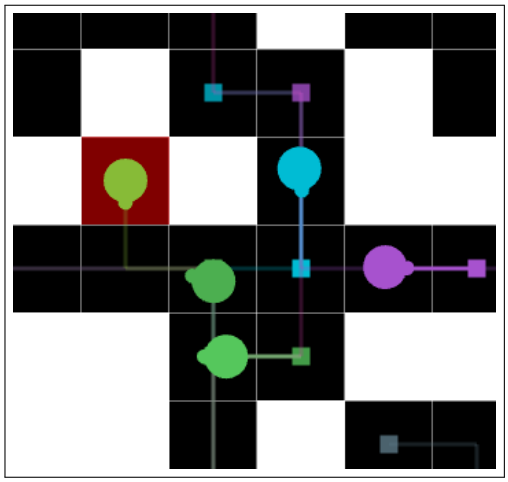


Figure 5.8: Example of a crossroad in a plan

Crossroad is a part of the environment which is in one or more directions surrounded by obstacles, so many robots will want to move through there. Due to the high concentration of robots, anyone arriving with a delay or getting slightly stuck inside the crossroad will cause a cascade of problems for others. In this case, a plan that for whatever reason agents avoid this point, even if it means in theory longer paths and higher SOC, can end up with a lower runtime of robots.

This kind of behaviour was expected from robots by the initial thesis. One reason why the beneficial suboptimality did not appear might be because the crossroads and $k + 1$ collisions are less common than expected or that their impact is not that dependent on suboptimality of plan.

Concerning algorithms, in a relatively small and agent dense environment, ECBS performed poorly. It had a significantly higher computation cost than LNS for plans of similar quality in terms of SOC. Further on, almost universally, the best solution for a scenario, in terms of SOC, was provided by LNS. This is because, in the scenarios used, there is a high probability of collisions between agents, which will cause branching in ECBS. If these are common, amount of possibilities and constraints can spiral to very high numbers in such a way that it is not uncommon to find no solution even after the planning program has been running for twenty minutes.

Chapter 6

Conclusion

Throughout the work on the thesis, the number of problems and bugs related to mainly the simulator and other programs proved to be greater than initially anticipated. This considerably slowed progress down, especially in the earlier parts of the work on this thesis. Due to these delays, some more ambitious initial goals were abandoned. However, all stated steps in the assignment, as listed here:

1. Get acquainted with current versions of multi-agent planning algorithms, their robust versions and freely available implementations.
2. Get acquainted with the environment (simulator) for performing multi-robotic experiments developed by the IMR group, CIIRC.
3. Create utilities to run schedules generated by selected freely available planners in the IMR simulator.
4. Compare experimentally properties of execution of plans generated by selected planners with different control accuracy settings.
5. Describe and comment on the knowledge gained.

All of these have been successfully completed. Steps one and two were necessary prerequisites for the work on extending the algorithms and simulator respectively. The results of the third step can be seen throughout the work. And the results of steps four and five are discussed in Section 5.

In addition to just getting from existing plans to simulator results, the testing pipeline can continue with processing script to evaluation software [9], quite extensively modified, so it would work with data from the simulator, which converts test data to SQL tables. Which in turn can be used to create graphs or tables. So anyone who will use the current simulator should have access to an already working and tested set of programs, which can get them from scenarios for MAPF to graphs and tables for resulting data out of the simulator. When it comes to the results of the performed tests, gathered data does not support the initial hypothesis that in general, slightly suboptimal plans will beat optimal ones, but there is not enough certainty to disprove it either.

In terms of future directions, one of the options is to simply perform more tests. Due to time and hardware limitations, current tests have by far not exhausted the full range of possible tests. Further on, other MAPF algorithms than just ECBS and LNS could be used. One option is to use a specific planning algorithm that can create multiple plans with the same SOC, which would present a perfect opportunity to study how different phenomena in plans affect runtime cost.

Another option is to introduce unplanned agents to the environment. These are agents not part of the plan but are included when creating ADG and have priority in it.

Appendix A

Attachments

- **Simulator** - Contains current version of relevant branch of Simulator. Contains its own README for installation.
- **LaTeX source** - Contains source file of this exact document
- **Additional** - Contains results, plans, maps and extra python scripts. Has its own README
- **PichaJ_Robustness thesis.pdf** - this document.
- **README.txt** - Explains in a bit more details what each folder.

Appendix B

Bibliography

[1] Podlucký Jan. MAPF_LNS+MAPF_SIPP. `https://gitlab.ciirc.cvut.cz/podlujan/mapf_lns-mapf_sipp.h`, 2005.

[2] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):10256–10265, Jun. 2022.

[3] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: A bounded-suboptimal search for multi-agent path finding. 2021.

[4] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. Iterative refinement for real-time multi-robot path planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9690–9697, 2021. Only used for solver, available at `https://github.com/Kei18/mapf-IR`.

[5] M. Phillips and M. Likhachev. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of The 2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*, pages 5628–5635, 2011.

[6] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66, 2015.

[7] Roni Stern, Nathan Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, and Roman Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. 2019.

[8] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[9] Tomáš Verner. An environmnet for evaluation of robotic experiments, May 2023. Available at `https://dspace.cvut.cz/handle/10467/109523`, only resulting program was used.

[10] Jingjin Yu and Steven LaValle. Structure and intractability of optimal multi-robot path planning on graphs. *Proceedings of the AAAI Conference on Artificial Intelligence*, 27(1):1443–1449, Jun. 2013.