Path planning algorithm for levitating planar motion system

Henry Nilsson

Johan Ternerot

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
TFRT-6170
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2022

Abstract

In today's world of ever increasing competitiveness, solutions that include automation and smart production have become a vital part to consider in overall business strategy, specifically for the industry sector. Whether an automated production process will be beneficial or not is dictated by how sub-processes such as inter-production transportation operates. The importance of these processes has been displayed in the late increase of production efficiency when moving from traditional transportation units, such as conveyor belts, to more sophisticated systems, such as transportation robots. However, these new sophisticated systems comes with increased complexity and new challenges when implementing important behaviours such as speed, control and safety.

This thesis is linked to the challenge of developing a safe and time efficient feature for handling a sudden failure or halt in one of these systems, namely the Beckhoff XPlanar levitating planar motion system. Hence, the goal of the thesis was to develop a pathfinding algorithm to easily line up the agents in the XPlanar system from any given position to a pre-specified startup track.

The end-result was a multi-agent pathfinding algorithm that utilizes Conflict-Based Search and A* to move each agent from their start position to a desired end-position whilst avoiding collisions. The algorithm is specifically designed towards the XPlanar system, integrated through ADS communication making it executable from the Beckhoff PC-based control software TwinCAT3.

Acknowledgements

This master's thesis was conducted in 2022 at the Department of Automatic Control, Lund University, and at Beckhoff Automation AB. We would like to thank our academic supervisor Anders Robertsson, for providing valuable insight in the starting phase of the project, as well as general academic support. We would also like to thank Daniel Jovanovski, our supervisor at Beckhoff, Malmö, for investing time in us and in this project.

On behalf of Beckhoff's wishes and IP, this report will mainly focus on the theoretical parts of the developed solution. Thereby, the displayed results will only refer to performance and general functionality of the algorithm rather than the practically implemented end-result. A full technical report, with code included, is available at Beckhoff Automation.

Contents

Nomenclature

ADS	Automation Device Specification
CNC	Computerized Numerical Control
NC	Numerical Control
OPC	Open Platform Communications
PLC	Programmable Logic Controller

1

Introduction

1.1 Background

Automation has been a hot topic in the production industry ever since the introduction of assembly lines and human-machine interaction in production. One of the most important sub-processes in such lines is the ability to quickly and cost efficiently transport sub-assemblies and components from one production unit to the next. Furthermore, these kinds of production systems also reduce unnecessary and potentially wearing tasks within the production line which may be detrimental for workers at the plant. [Neumann et al., 2002]

Production-line transportation has long been done by integrating more traditional machinery, such as one-dimensional conveyor belts, mono-rail carts and transfer machines. However, during later years these old-fashioned systems have been continuously replaced and re-modelled to new more sophisticated technologies, such as multi-agent transporter robots [Kranzberg and Hannan, 2021]. The Beckhoff XPlanar system is one of those multi-agent systems that intends to introduce a new kind of technology, with five additional dimensions instead of conventional one-dimensional systems. This could revolutionize the industry by introducing new opportunities such as improved collaboration between production units and transporters to increase production-precision, and allow for higher freedom in sequencing different production processes. In other words, this kind of system has the capability of merging parts of the production-line such as the transportation and assembly into one continuous system.

Introducing a system as complex as this ensues a myriad of problems. A particularly interesting one is rebooting the system from an unexpected intermission. In a case such as this, the system must be able to return to a predetermined state in order to assure that the quality of the carried product won't be compromised.

1.2 XPlanar overview

The Beckhoff XPlanar system is the successor to their XTS system, which is a linear product transport system. Hence, the XPlanar system also intends to solve the problem of component and sub-assembly transportation between production equipment and machinery, but with added degrees of freedom, and thereby an increased amount of transportational options. With 6 degrees of freedom, magnetically driven movers and customizable tile geometry, the XPlanar system provides benefits such as; increased plant output, simultaneous production of different products and formats, precise positioning during production, low cleaning and maintenance costs. An illustration of the XPlanar movers and tiles can be seen in Figure 1.1. [Beckhoff, 2018a]

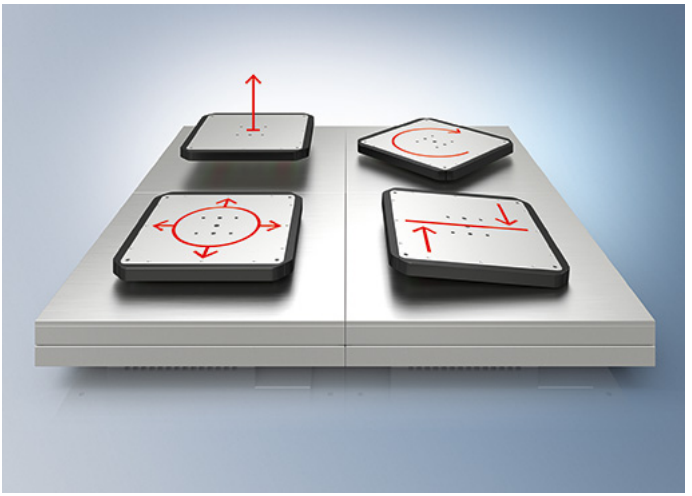


Figure 1.1 illustrates the XPlanar tiles, movers and the possible actions for the movers. [Beckhoff, 2018a]

1.3 Problem Formulation

An important aspect of all production systems, including the XPlanar system, is handling uncontrolled system failures or interrupts. Since the XPlanar system incorporates position lag monitoring, the system will shut down if any uncontrolled or unpredicted mover positioning occurs. In order to reduce consecutive production-delays because of these shutdowns, the system needs to handle an automated reset and alignment of the movers regardless of time and location. This project aims to solve this problem by developing a pathfinding algorithm and integrating the solution into the already existing XPlanar system.

1.4 Goal of the master's thesis

The goal of this master's thesis is to develop a pathfinding algorithm for a multi-agent robot system, which can be implemented into the existing XPlanar system. In order to execute a well-functioning automated reset in regard to safety and time-efficiency, the overall end-goal can be divided into milestone goals as follows:

- Choose and implement an algorithm that allows movers spread all over the system to line up, one by one, on a specified path while avoiding collision
- Optimize the previous solution (one by one) with respect to time
- Optionally, extend the solution to line up all movers simultaneously

1.5 Delimitations

In order to make this project feasible and provide a high-quality solution, the following limitations were set:

- The project will be performed during a timespan of 20 weeks
- The project will be performed by a team of two students
- Safety restrictions will result in physical tests only being feasible towards the end of the project
- Limited access to testing equipment will result in the development of a self produced digital testing simulation
- The solution has to accept specified inputs and generate the requested output, in order to be integratable with the already existing software
- The solution has to be translated from an object-oriented programming language into TwinCAT to be integratable with the already existing software

1.6 Individual contributions

Both team members have been involved in all parts of the project. However, Henry Nilsson was mainly responsible for development of the TwinCAT code whilst Johan Ternerot had extra responsibilities within the academic parts of the project and research for algorithms. Furthermore, the workload was very evenly divided across the two team members.

2

Theory

2.1 TwinCAT3 technology

TwinCAT, short for The Windows Control and Automation Technology, is a software developed by Beckhoff Automation to transform any PC-based system into a real-time control for PLC; NC; CNC and robotics runtime systems. In terms of operating systems, TwinCAT is compatible with Windows CE, Windows 7, Windows 10 and TwinCAT/BSD.

TwinCAT integrates itself into Visual Studio® and can therefore support C/C++ and the IEC 61131-3 languages for programming real time applications. However, data connection via open standards such as OPC and ADS also allows for other languages (Java, Python, etc) to be used for some applications via scripts. Furthermore, linking to MATLAB®/Simulink® is also available. [Beckhoff, 2018b]

2.2 Multi-agent pathfinding

2.2.1 Definition

A classical multi-agent pathfinding (MAPF) problem with a given number of agents k, is defined by a tuple of three elements:

- An undirected graph of vertices and edges, *Graph = (V, E)*
- A vector mapping agents to a start vertex, $Start = [start_1, start_2 .... start_k]$ where $start_k$ is an arbitrary starting vertex.
- A vector mapping agents to a goal vertex, $Goals = [goal_1, goal_2 ... goal_k]$ where $goal_k$ is an arbitrary goal vertex

These elements are then complemented with the attribute time discretization such that each agent is related to a specific state s for the vertex v it occupies at time step t. Actions are then defined as a function which transitions agent k from the current

state s_i to the next state s_{i+1}: $a(s) = s_{next}$. Possible actions thereby include moving the agent from the current occupied vertex to another adjacent vertex or waiting in the same vertex for another time step. A sequence of actions $\pi_i = (a_1, ..., a_n)$ then becomes a single-agent plan for agent i if it fulfills the requirement of moving the agent from the starting vertex to the goal vertex. The number of actions in the sequence are used to define the cost for the plan $c(\pi_i)$. A solution to the classical MAPF problem is thereby defined as a set of single-agent plans $\pi_i, ...\pi_k$, one for each agent present in the graph. Furthermore, the solution will only be valid if none of the single agent plans conflict with each other. [Andreychuk et al., 2022]

2.2.2 Conflicts for classical MAPF

In theory, there are multiple types of conflicts that can occur in a classical MAPF problem. However, there is no guarantee that these will occur in any given MAPF problem since they heavily depend on behavioural aspects of the used algorithm as well as attributes for both the agents and the configuration space.

In the conflict definitions below we denote a single-agent plan as π, vertices as v and time steps as t.

- **Vertex Conflict**
  A vertex conflict is defined as two agents trying to occupy the same vertex at the same time step. Hence, a vertex conflict will occur if $\pi_i(t) = \pi_j(t)$.
- **Following Conflict**
  A following conflict is defined as one agent trying to occupy a vertex occupied by another agent at the previous time step. Hence, a following conflict will occur if $\pi_i(t+1) = \pi_j(t)$. This conflict is heavily dependant on the agents' physical geometry and how they move. This is further expanded upon in Chapter 3.3.3.
- **Cycle Conflict**
  A cycle conflict is defined as a set of agents $\{\pi_i, \pi_j, \pi_k, ..., \pi_l\}$ trying to occupy nodes in a cyclic fashion such that a deadlock occurs. Hence, a cycle conflict will occur if $\pi_i(t+1) = \pi_j(t)$, $\pi_j(t+1) = \pi_k(t)$, $\pi_k(t+1) = \pi_l(t)$ and $\pi_l(t+1) = \pi_i(t)$.
- **Edge Conflict**
  An edge conflict is defined as two agents trying to traverse the same edge at the same time step. Hence, an edge conflict will occur if $\pi_i(t+1) = \pi_j(t)$ and $\pi_j(t+1) = \pi_i(t)$.

If a conflict occurs, a constraint of the same type (vertex-, following-, cycle- or edge-constraint) is generated. The different types of conflicts are illustrated in Figure 2.1. [Stern et al., 2019]

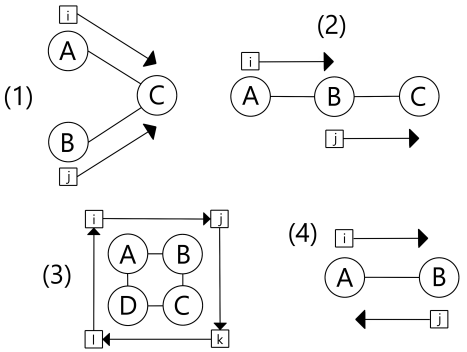


Figure 2.1 illustrates the different situations where each conflict might occur ((1) vertex, (2) following, (3) cycle and (4) edge). The circles are nodes, the small squares are agents, and the arrows represent how they are planning to move. Adapted from [Stern et al., 2019]

2.2.3 Centralized vs Decentralized Algorithm

A centralized method for multi-agent pathfinding is based on implementing one central planner that computes all agent paths simultaneously. In theory, a centralized method will be optimal. However, in practice the method has some issues with complexity and scalability for many units. On the other hand, a decentralized method is based on individual planners for each agent. This is often done by each planner computing an individual path ignoring all other agents, and then handling conflicts as they come, which leads to the requirement of communication between the agents. This method significantly lowers computations which yields a faster but potentially more sub-optimal and less complete result. [Lejeune and Sarkar, 2021]

2.3 Pathfinding algorithms

2.3.1 Low-level algorithms

Dijkstra

The Dijkstra algorithm is one of the earliest well known pathfinding algorithms and intends to solve the problem of finding the shortest path from one source vertex to any other specified vertex in a graph [Misa, 2010; Dijkstra, 1959]. The algorithm is an iterative so-called greedy algorithm that checks and updates the value of adjacent vertices for the current vertex visited based on the weights or distances between them. This is done in an iterative way to find the absolute shortest path from the starting vertex to the specified goal-vertex. [Chen, 2003]

A better understanding of how to implement the Dijkstra algorithm can be acquired by studying the pseudo-code of Algorithm 1.

Algorithm 1 Dijkstra algorithm [Wikipedia, 2022a]

```
1:  Input ← Graph, source
2:  for each vertex v in Graph.vertices do
3:      dist[v] ← ∞
4:      prev[v] ← UNDEFINED
5:      Q ← v
6:      dist[source] ← 0
7:  while Q is not empty do
8:      u ← vertex in Q with min_dist[u]
9:      Remove u from Q
10:     for each neighbour v of u still in Q do
11:         alt ← dist[u] + + Graph.edges(u, v)
12:         if alt < dist[v] then
13:             dist[v] ← alt
14:             prev[v] ← u
15: return dist[ ], prev[ ]
```

To understand the algorithm and its implementation even better, Figure 2.2 is used to illustrate a graph in exemplifying purpose.

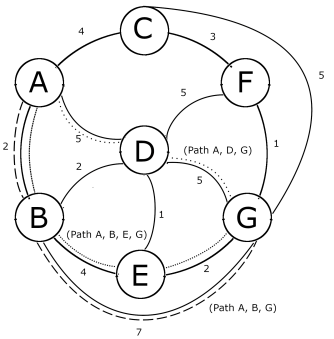


Figure 2.2 illustrates an example of a graph with nodes and their weighted transitions. Adapted from [Javaid, 2013].

In order to use the algorithm accurately, the first step will be to implement a matrix or two lists in order to store both the path from the source vertex to every

other vertex as well as the paths' cost. The unvisited vertices will be used to iterate in the *while* in the pseudo code of Algorithm 1. Which vertex to visit next is represented by the vertex v in the pseudo code where u is the current vertex being visited [Javaid, 2013].

In Figure 2.2 we can see this exact process being executed. The process starts at the source vertex A, where A is first set to *visited* in the visited vertices column. After that, each immediate neighbour to A is located and set in the *Next* column. Hence we get the matrix seen in Figure 2.3 [Javaid, 2013].

Visited	Node					
	B	C	D	E	F	G
A	2	4	5			
Next	B	C	D			

Figure 2.3 illustrates the visited/next matrix for vertex A. Adapted from [Javaid, 2013].

The next vertex to visit will then be B since this has the lowest cost. Therefore, B will be set to *visited* and travel distances from A to the neighbours of B via B are calculated. Since the cost of the path $A \rightarrow B \rightarrow D$ is 4, the value of $A \rightarrow D$ is updated. Thereafter C is set as the next node to visit from A, since it has the same value as node D and is a direct neighbour of vertex A. This process is done in an iterative manner and the shortest distance to each vertex from A is derived as seen in Figure 2.4. [Javaid, 2013]

Visited	Node					
	B	C	D	E	F	G
A	2	4	5			
A,B	2	4	4	6		9
A,B,C	2	4	4	6	7	9
A,B,C,D	2	4	4	5	7	9
A,B,C,D,E	2	4	4	5	7	7
A,B,C,D,E,F	2	4	4	5	7	7
A,B,C,D,E,F,G	2	4	4	5	7	7
Next	B	C	B	B	C	B

Figure 2.4 illustrates the visited/next matrix for vertex A. Adapted from [Javaid, 2013].

A*

The A* algorithm can be seen as a further improvement on the Dijkstra algorithm by implementing heuristic searching in combination with the search for the shortest

path. The algorithm is defined as a best-first algorithm, where each vertex in the graph of the configuration space is evaluated by the value: $f(v) = g(v) + h(v)$. $h(v)$ is the heuristic distance, in our case the Manhattan distance of the current vertex to the goal state, i.e. the sum of their absolute difference in Cartesian coordinates [Wikipedia, 2022f]. $g(v)$ is the cost of the path from the initial state to the goal state for the sequence of vertices that makes up the current path being evaluated, ending in the current evaluated vertex. Each adjacent vertex of the current vertex is then evaluated based on $f(v)$. The vertex with the lowest value of $f(v)$ is then chosen as the next to visit [Duchoň et al., 2014].

The pseudo-code for the A* algorithm is illustrated in Algorithm 2. The open list consists of the vertices that have been visited but not expanded by evaluation of adjacent vertices, whereas the closed list consists of the vertices that have been both visited and expanded.

Algorithm 2 A* algorithm [Wikipedia, 2022b]

```
1:  Input: startNode, goalNode, h
2:  openSet ← startNode
3:  cameFrom = an empty map
4:  gScore = map with default value of ∞
5:  gScore[startNode] = 0
6:  fScore = map with default value of ∞
7:  fScore[startNode] = h(startNode)
8:
9:  while openSet is not empty do
10:     currentNode = the node with lowest f-Score value in openSet
11:     if currentNode == goalNode then
12:         return reconstruct_path(currentNode, cameFrom)
13:     openSet.remove(currentNode)
14:     for each neighbour of currentNode do
15:         d = distance between currentNode and neighbour
16:         tentative_gScore = gScore[currentNode] + d
17:         if tentative_gScore < gScore[neighbour] then
18:             cameFrom[neighbour] = currentNode
19:             gScore[neighbour] = tentative_gScore
20:             fScore[neighbour] = tentative_gScore + h(neighbour)
21:         if neighbour not in openSet then
22:             openSet ← neighbour
23: return failure
```

One great advantage with the A* algorithm is its capability of being modified. Some common modifications are adding weights to reduce computation time, changing the heuristic for different behaviours that you might want depending on the configuration space, and modifying the combinations used in the evaluation of total cost to perform bi-directional search [Patel, 2022]. Some of these modifications have eventually led to new algorithms being sprung from A*. One example is the D* algorithm, which re-uses previous A* computations and re-plans parts of the overall path to account for newly discovered changes in the configuration space. In other words, the D* is an incremental search algorithm [Wikipedia, 2022c].

Breadth/Depth-first search
Breadth- and depth-first search are both primitive low-level search algorithms. They work similarly, in that they both search through trees iteratively. In the case of applying this to pathfinding, the layout or maze needs to be structured in such a way that it behaves like a tree. The easiest, and most widely used way of doing this is by dividing the maze into vertices, and then setting each vertex's children to the vertex next to it, namely its "neighbours". The tree is then structured by putting either the starting vertex, or the goal vertex, at the top, and iteratively searching through the tree until it finds the other part. [Rahim et al., 2018]

The way breadth- and depth-first search differ is how they search through this tree. Breadth-first begins at the start vertex, and then explores all of its neighbours, after that it explores the neighbours' neighbours. This process is performed until it finds the goal vertex. Depth-first search on the other hand explores its way to the bottom of the tree, finding the node furthest from the starting vertex. It then moves on to the neighbours of the "deepest" vertex's parent. After it has searched through all those neighbours and their children, the next vertex being evaluated is the parent of the deepest vertex's parent. This continues until the sought vertex is found. [Wikipedia, 2022d] [Wikipedia, 2022e]

2.3.2 High-level algorithms

Increasing Cost Tree Search
Increasing Cost Tree Search (ICTS) is an algorithm using two levels of search, one high-level and one low-level, and is based on the idea that a complete solution is a collection of single-agent solutions. Therefore, the high-level algorithm uses high-level nodes that are denoted as combinations of single agent costs where every high-level node becomes a vector of individual costs. These high-level nodes are then stored in the increasing cost tree where the root represents the optimal solution in regards to total cost, with each leaf representing the next most optimal solution. To check whether a high-level node contains a valid solution in regard to limitations such as collision avoidance, the algorithm performs a low-level goal test on that high-level node. This goal test is performed by the low-level algorithm searching through combinations of single-agent paths for the specific costs each agent has

in the high-level node. This is done until a complete solution is found or until all combinations have been tried and no solution has been found. [Sharon et al., 2013]

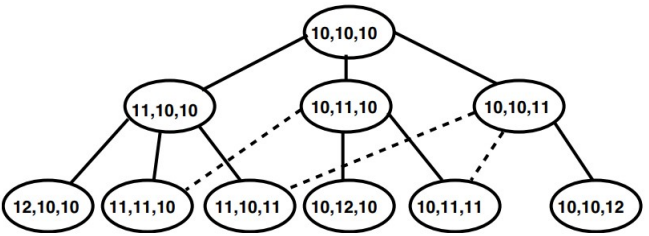


Figure 2.5 illustrates an ICT for three movers [Sharon et al., 2013].

The pseudo-code for the ICTS algorithm can be seen in Algorithm 3, where MDD denotes the multi-value decision graph where the elements are the different solutions with the same cost for a single agent.

Algorithm 3 ICTS algorithm [Sharon et al., 2013]

```
Input: (k, n)
Build the root of the ICT
for each ICT node in breadth-first manner do
    for each agent a_i do
        Build the corresponding MDD_i
    for each pair of agents (a_i, a_j) do
        Perform pairwise search
        if pairwise search failed then
            break

    search the k-agent MDD
    if goal node was found then
        return solution
return
```

Conflict-Based Search

Conflict-Based Search (CBS) is a two level algorithm where the high-level search is performed in a constraint tree with so-called high-level nodes. These high-level nodes represent a collection of all single-agent solutions and the constraints each of these agents have. The constraints are created based on conflicts occurring when single agents are making illegal moves in regards to collision avoidance. A constraint for an agent can be represented as a tuple (a_i, v, t) where agent a_i is prohibited from occupying vertex v at time step t. Likewise a conflict is a tuple (a_i, a_j, v, t) where a_i

and a_j are the agents trying to occupy the same vertex v at the same time step t. A complete solution has only been found if the high-level node contains an individual solution for each agent and does not contain any constraints, meaning all agents have found a path without interfering with other agents' paths. Since conflict-based search is a best-first search algorithm, it will automatically return the first viable complete solution it finds. [Sharon et al., 2015]

Algorithm 4 CBS algorithm [Sharon et al., 2015]

```
1:  Input: MAPF instance
2:  Root.constraints = empty map
3:  Root.solution = find individual path using the low-level algorithm
4:  Root.cost = SIC(Root.solution)
5:  openSet ← R
6:
7:  while openSet is not empty do
8:      P = the node with lowest solution-cost from openSet
9:      Validate the paths in P until a conflict occurs
10:     if P has no conflict then
11:         return P.solution
12:     C = first conflict (a_i, a_j, v, t) in P
13:     for each agent a_i in C do
14:         A = new high-level node
15:         A.constraints ← P.constraints + (a_i, s, t)
16:         A.solution ← P.solution
17:         Update A.solution by invoking low-level algorithm for a_i
18:         A.cost = SIC(A.solution)
19:         openSet ← A
20: return
```

2.4 Benchmarks

In order to provide a reference point and a better understanding of performance for algorithms possibly suited for the XPlanar-problem, data from previous tests and research are presented as benchmarks.

Component	A*	Djikstra	BFS
Path Length	34	34	34
Time (ms)	0.4	0.6	0.8
Computed Blocks	415	618	618

Figure 2.6 illustrates length of path, runtime and explored nodes for different low-level algorithms [Permana et al., 2018].

	Runtime (ms)				
k	A*	A*+OD	ICTS	ICTS3	CBS
3	14	3	1	1	2
4	380	10	2	1	13
5	9 522	50	5	5	13
6	47 783	90	9	10	16
7	N/A	1 122	92	44	186
8	N/A	1 756	271	128	211
9	N/A	7 058	2 926	921	550
10	N/A	21 334	10 943	2 335	1 049
11	N/A	N/A	38 776	5 243	2 403
12	N/A	N/A	N/A	25 537	15 272
13	N/A	N/A	N/A	45 994	36 210

Figure 2.7 illustrates runtime in ms for each algorithm with an 8x8 grid and k number of agents [Sharon et al., 2012].

The terms used in Figure 2.7 that are not further examined in this report are: (1) Operator Decomposition (OD) and (2) Increasing Cost Tree Search + 3E, or ICTS3, which calculates three agents' path simultaneously using ICTS [Sharon et al., 2012].

Coverage (Success rate)				
Model	All	CS	OI	TT
EPEA*	59,67	56,23	53,38	63,1
ICTS	55,39	54,17	50,32	62,49
CBS-H	55,93	51,58	54,11	72,04
Lazy CBS	89,74	89,59	89,09	88,36

Figure 2.8 illustrates the success rate in % for 4 different multi-agent pathfinding algorithms with the different set-ups CS, OI and TT [Kaduri et al., 2021]

The set of algorithms tested in Figure 2.8 are; (1) Enhanced Partial Expansion A* [Goldenberg et al., 2014]; (2) Increasing Cost Tree Search [Sharon et al., 2013]; (3) Lazy-CBS [Gange et al., 2019]; and (4) CBS-H [Kaduri et al., 2021]. These tests include 190,000 solved MAPF problems that were run on different setups and different grids from a publicly available grid-based MAPF benchmark containing: game grids, city maps, mazes, rooms, open grids with randomly placed obstacles and warehouse-grids [Stern et al., 2019]. The different success rate for each set-up can be seen under each abbreviation in Figure 2.8. The description of each abbreviation is; Cross Sides (CS): all agents start on one side and traverse to the other side; Outside In (OI): agents start near the outer edges of the grid and are assigned targets near the center of the grid; Tight to Tight (TT): agents start together and are assigned targets that are as close as possible elsewhere on the grid. [Kaduri et al., 2021]

3

Methodology

The overlaying tasks and different phases for the project can be seen in the gantt chart in Appendix A. The methodology used for each phase or task will be described in further detail in upcoming sections.

3.1 Breaking down the XPlanar into a MAPF problem

A Multi-Agent Pathfinding problem consists of a multitude of different parts that change from system to system. In order to find the optimal algorithm for each system they need to be analyzed, and broken down into parameters relevant to pathfinding theory. These parameters can be configuration space layout, agent geometry, how the agents move and the proportion between the amount of agents and area the configuration space covers.

While the XPlanar can be modified in a lot of ways, most systems follow a certain pattern. Their tile layout is often open spaced, there are few to none corridors, and it does not consist of rooms. Most systems consist of large areas with simple geometries. The XPlanar movers are square shaped, and move consistently with the same speed. TwinCAT is also able to send instructions to different movers with such consistency that they move synchronized. These factors affect the conflicts. The XPlanar is also able to handle a high amount of movers within a smaller area.

3.2 Understanding the problem and developing the test package

3.2.1 Researching algorithms

In order to get a better understanding of the task and usual problems that need to be solved for a multi-agent pathfinding system, a broad explorative research was performed. This was succeeded by a more narrow research to find potentially useful algorithms in regards to both low-level single-agent pathfinding and high-level

multi-agent pathfinding.

Once a solid foundation of information had been acquired, the next step was to gradually boil down the total amount of potential candidate-algorithms. This was done by weighing and comparing the low- and high-level algorithms based on a set of criteria;

1. **Degree of solvability** refers to whether the algorithm generates a complete solution or not

2. **Optimality** refers to the total cost of mover steps for the generated solution

3. **Time complexity** refers to the average and worst-case execution time of the algorithm

4. **Conceptual complexity** refers to how easy or difficult it is to conceptualize the algorithm

5. **Complexity of code** refers to how easy it is to implement the algorithm

6. **Applicability for the XPlanar system** refers to how well the algorithm is suited for the open-space XPlanar system

Furthermore, benchmarks from Section 2.4 were used to supplement these criteria, and expand the decision basis with an overview of the performance for the algorithms in different configuration spaces. In order to make it clear which algorithm to choose for both the low- and high-level search, a concept scoring was performed.

3.2.2 Development of simulation package

To get a visual perception of how the developed code would behave and test whether it would meet specified requirements, a simulation package was developed. The simulation package was based on the python module *pygame* along with classes for visual representation of tiles and movers. To test a maximum of 40 movers and a geometry with tile-layout 100x100 tiles, the usage of factoring was implemented in a class with constants. Furthermore, the constants class included pre-specified mover characteristics to make the simulation as realistic as possible.

The interface for the simulation can be seen in Figure 3.1.

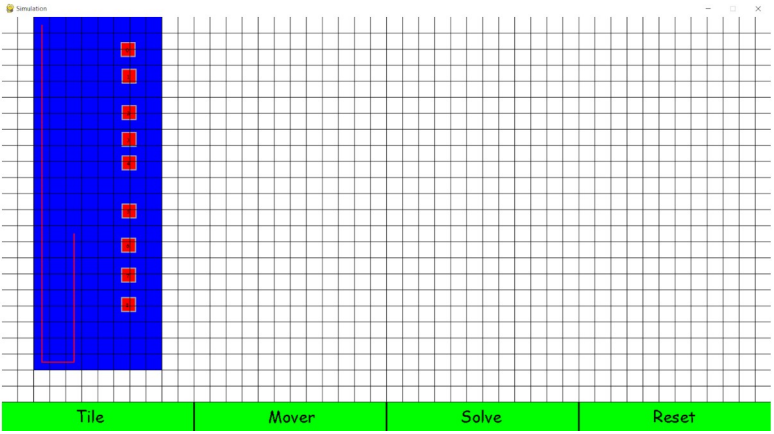


Figure 3.1 illustrates the GUI for the simulation package. Blue colouring indicates the tiles, red squares indicate the movers, grid elements represent nodes in the configuration space and the red line indicates the specified startup-path for movers to align on.

3.2.3 Development of unit tests

In order to assure that behavioural aspects such as collision avoidance were implemented correctly, the first set of unit-tests seen in Figures 3.2, 3.3, 3.4, 3.5 and 3.6 were developed to test collision-avoidance for the different types of constraints; edge-, vertex- and transition-constraint. The transition-constraint which was developed specifically for this system is described in detail in Section 3.2.3. Furthermore, these tests also include testing a basic two-mover case and a function for arranging the movers in a specific sequence on the startup path.

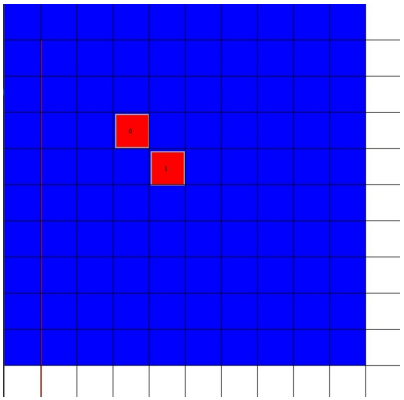


Figure 3.2 illustrates the the case 5x5 tile-layout and 2 movers (testing basic functionality)

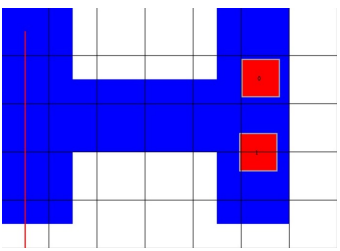


Figure 3.3 illustrates the case H-formed tile-layout with 2 movers (testing vertex- & transition-constraints)

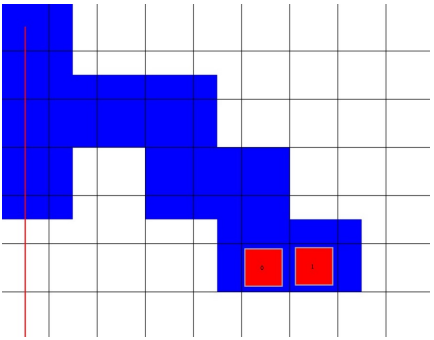


Figure 3.4 illustrates the case zig-zag tile-layout with 2 movers (testing transition-constraints)

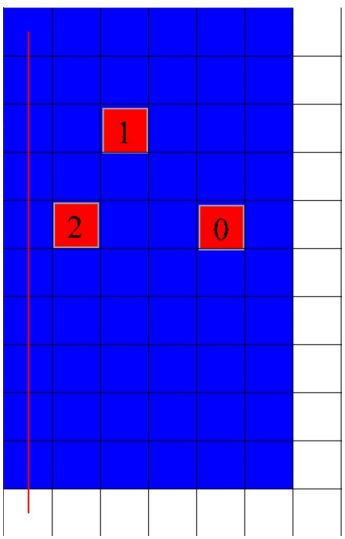


Figure 3.5 illustrates the case 3x5 tile-layout with 3 movers (testing arranging the movers in a specific sequence on starting path $0 \rightarrow 0x1$, $1 \rightarrow 0x2$, $2 \rightarrow 0x0$)

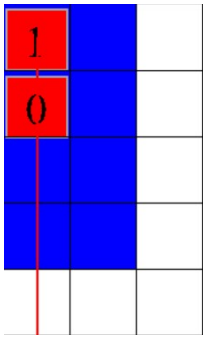


Figure 3.6 1x2 tile-layout with 2 movers and arranged goals $0 \rightarrow 0x0$ & $1 \rightarrow 0x1$ (testing edge constraints)

The second stage of developing the unit-tests included an automated randomized test file outputting whether each run was a success or a fail, time it took the algorithm to execute, and environmental parameters such as number of tiles, number of movers, mean distance of every movers starting position to its goal position, etc. In order to make this test randomized, four of the algorithm inputs were randomly changed before each new run. These inputs were tile-geometry, mover size, number of movers and mover positions. The startup path was kept constant and was made up by three lines defined by the distance in millimeters between one x- and y-coordinate to another in the coordinate system seen in Figure 3.7. Line 1 was defined by the distance between (360,120) and (360, 4200), line 2 by the distance between (360, 4200) and (1200, 4200) line 3 by the distance between (1200) and (1200).

The map was randomized with the help of a random map generator from a 48x24 base layout capped at 100 tiles and excluding groups of tile-isles. The 48x24 base was used in order to make every test-case drawable in the GUI from the simulation package, since visualizing failed cases was extremely helpful in terms of test-driven development and fixing bugs. The mover size was picked based on a randomly generated integer between 1 and 4, each integer representing a different mover size. The number of movers were chosen based on the total number of goal positions present on the startup path, with a minimum of 8 movers. If the configuration space would not support at least 8 movers, a new tile-geometry was generated until the configuration space would support 8 movers. Since the startup path was kept constant, the maximum possible amount of movers would vary based on the mover size due to the node size being dependent on the mover size. The maximum number of movers for the different mover sizes were; 67 for mover size 113, 47 for mover size 127, 47 for mover size 155, and 32 for mover size 245.

It is worth noting that the number of movers was set to be dependent on the number of goal positions available, since the setup becomes completely unsolvable if the number of movers are greater than the number of goal positions. An example of how one of these tests would look can be seen in Figure 3.7.

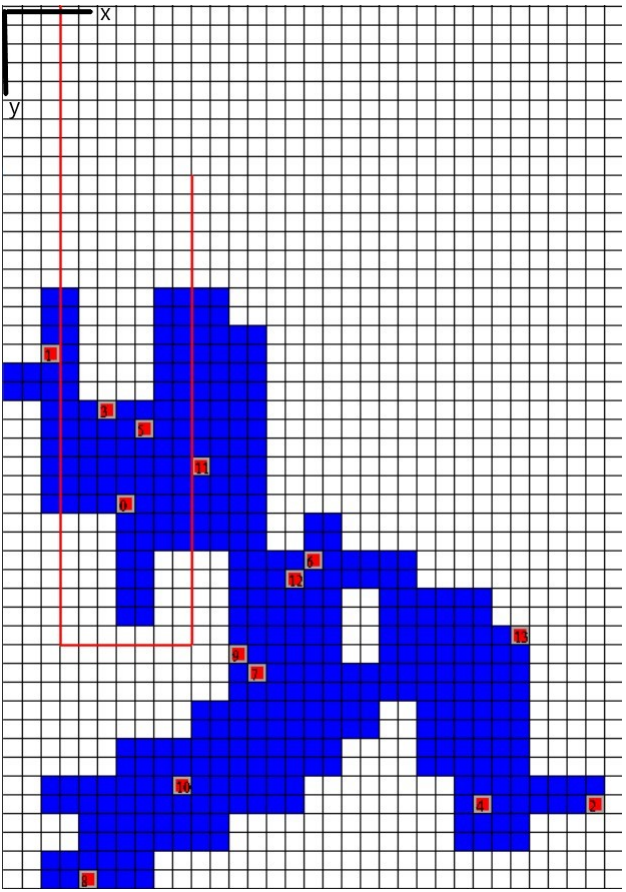


Figure 3.7 illustrates a randomized test with 14 movers

This new and enhanced test was used to detect flaws in functionality, optimize the algorithm, retrieve statistical data and drive continuous improvement of the developed code even further. Retrieving relevant statistics was done by testing a wide range of scattered values for certain parameters to get an overview of the most optimal ranges. Consecutively, the ranges were narrowed down and the tests

were done with a more continuous set of values within the ranges instead of being scattered.

It is worth mentioning that unit-tests were developed at the early-to-mid stages of the project since the simulation provided sufficient testing to drive development in the first stages of the project.

3.3 Developing the algorithm

3.3.1 Development of the A* algorithm

The objective for developing the A* algorithm was one so called mover finding a path from its input start position to an input goal position. This was done by setting up a code skeleton with the basic necessary methods, and input parameters based on the requested specifications from Beckhoff. The development proceeded with iteratively writing and testing the functionality of the written code with the help of our simulation from the test package.

3.3.2 Development of the CBS algorithm

Once a single-agent pathfinding algorithm had been developed, the next natural step was to start developing the multi-agent pathfinding algorithm to enable simultaneous movement for multiple movers and possibly all. This was done by utilizing a low-level algorithm to calculate solutions for each mover individually, then iterate through all solutions pairwise to see if any sort of conflict would occur between two movers. However, two movers' paths crossing is not enough to qualify as a conflict, as the movers must cross each other during the same time-period for a conflict to occur. This introduces a new dimension to the low-level algorithm A*, which is time.

The biggest change for the A* algorithm was going from nodes, which consists of only location, to state, which consists of location and time. This basically teaches movers to wait for another mover to pass, instead of colliding with it. The different kinds of conflicts that can occur are defined in Section 2.2.2.

3.3.3 Refining and customizing the algorithms

Once the first draft for the full CBS had been developed, the next step was to refine and customize the two algorithms in order to provide the desired functionality with consideration to the specified requirements and characteristics of the XPlanar system. The ideas and concepts for these customizations and refinements were derived from the set of criteria mentioned in Chapter 3.1.1 in order to achieve an overall higher-quality solution.

The first customization was the introduction of the third constraint named transition-constraint. Because of the squared geometry of the movers, the transition-constraint class was developed to assure collision avoidance for situations with one agent moving into a node in one direction and another agent moving out of that node in the perpendicular direction. These situations are illustrated in Figure 3.8.

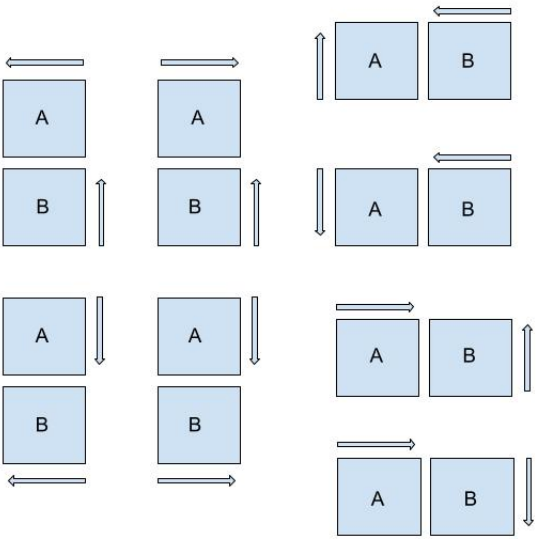


Figure 3.8 illustrates the different situations where a transition constraint can occur between agent A and agent B.

By denoting a single agent plan as π, direction d and time steps as t we can derive the definition of a situation where transition constraints occur as $\pi_i(t+1) = \pi_j(t)$ if $\pi_i(t)$ is diagonal to $\pi_j(t+1)$.

The second customization was changing data structures, mainly for reducing the time of inserting, getting and checking for elements in the open and closed sets in both the CBS and A* algorithm. In the first draft of the solution, the open and closed sets were implemented with lists since they proved to be easy-to-use and versatile since they can be used as both stacks and queues. However, lists proved to be slow when handling a large number of nodes. Hence, these lists were exchanged to Python sets after comparing the time complexity of different actions for different data structures in Python, seen in Table 3.1. To clarify, the first iteration of lists

used sorted insert in order to pop the first element from the list when retrieving the lowest cost node.

Table 3.1 illustrates the average time-complexity for different operations for the data structures lists and sets in python [Python software foundation, 2022].

Time-complexity for operations		
Operation	Lists	Sets
Length	O(1)	O(1)
Containment	O(N)	O(1)
Sorted insert/Add	O(Nlog(N))	O(1)
Pop	O(1)	O(1)
Remove	O(N)	O(1)
Get minimum	O(1)	O(N)

The third customization, introducing a weight into the A* algorithm would also prove to decrease time complexity along with increased solvability and optimality. This customization heavily relied on the introduction of time as a third dimension apart from x and y coordinates. The introduction of time meant that each agent was able to see their current location as a viable state in the next time-step. Hence, providing agents with the option to perform a wait-action. However, since the A* algorithm chooses the next state for an agent with regard to lowest score, there is a possibility the agent will perform perpetual actions of waiting if waiting is cheaper than moving. To counter this a penalty was introduced for every time an agent visits the same location. This discourages agents from visiting the same location multiple times. Since the algorithm uses a score to determine the next viable state, this penalty needed to increase the "bad" states cost somehow.

At first this was done through a flat cost increase for every time a state was visited. This solved cases the algorithm previously could not solve, especially cases with a large amount of tiles where the agents were placed far from their goal locations. However, in cases where the agents were bundled close together near their goal locations the flatly increasing cost affected the result negatively. The penalty score was then reduced to a cost lower than the cost of an actual step. This fixed the clustered case it previously could not solve, but it also made the more spread out case unsolvable. Further optimization of the penalty score decreased the amount of failures, but the algorithm could still only solve one of the cases. The idea then sparked to have a penalty score based on the distance to the target. This new solution proved to be able to solve both kinds of cases. Extensive testing ($n = 50000$) was then performed in order to find the best cost increase in regard to solvability. The results for the testing can be seen in Section 4.5.

In a larger programming project such as this one, keeping the multiple classes

and functions structured is essential for understanding the code. This is usually done with the help of a linter and a formatter. While Python has its own linter, there are more optimized ones available, and in this case one named pylint was used [Python Code Quality Authority, 2022]. The formatter used is named black [Langa, 2022]. These tools helped format the code, consistently use naming conventions and get rid of excessive code.

3.4 Communication between the XPlanar and the algorithm script

3.4.1 Development of TwinCAT program

Once the algorithm was finished, the next step was to test the solution on the TwinCAT3 software. This software contained Beckhoff's simulation for the XPlanar system, which was used to make sure the algorithm worked towards an actual XPlanar system and not just towards our simulation. To do this a TwinCAT program was needed to send the appropriate data to the algorithm and read the instructions that the algorithm produced. The TwinCAT program was developed to read the following data from the XPlanar system; tile layout, mover dimension and the mover positions. The program works by sending all the data to the algorithm and getting the instructions in return. It then starts reading them one time-step at a time, and executes the MoveToPosition command with correct inputs for each mover.

3.4.2 Development of the ADS communication file

Throughout the TwinCAT testing phase some sort of communication was needed between TwinCAT and the algorithm. This was done through an ADS with the help of a python-module called *pyads*. Once the TwinCAT program was started and had initialized, it sent a handshake to the communication program telling it that it was ready to send over the data needed for the algorithm. Once the data was sent and the algorithm was done, the communication program then converted the instructions to a format more fitting for TwinCAT, and sent it back. To access the data, the ADS functions *read_by_name* and *write_by_name* were used.

3.4.3 Refining the CBS algorithm

Once the tests for the algorithm via ADS communication were done, a few modifications were made for the inputs and outputs of the algorithm. These modifications were mainly removing unnecessary inputs such as the number of movers, and dividing the first step for each mover in one step each for x- and y-direction to assure collision avoidance when aligning to the node grid.

3.5 Packaging to exe file and XPlanar tests

In the final stages of development, both tests in the TwinCAT3 simulated XPlanar environment and real XPlanar tests were used to assure a fully functional integration of the algorithm with the ADS communication and TwinCAT3 solution file. It is worth noting that real XPlanar tests were only feasible during one day due to limited access. These real experiments are briefly described in Section 4.4 Once the integration was successfully implemented, the ADS and the algorithm were merged into one file and packaged to an executable (exe) file in order to make it runnable directly from TwinCAT3.

4

Results

The result of this thesis is the Ternerot and Nilsson-algorithm (TaN-algorithm). It uses modified versions of the A* and CBS algorithms to solve multi-agent pathfinding problems and achieve the goals described in Section 1.4.

4.1 Concept scoring for algorithms

Table 4.1 illustrates the concept scoring for different low-level algorithms based on the set of criteria from Section 3.2.1.

Low-level algorithms				
Criteria	A*	Dijkstra	Breadth/depth search	D*
1	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆
2	⋆⋆	⋆⋆⋆	⋆	⋆⋆
3	⋆⋆	⋆	⋆	⋆⋆⋆
4	⋆⋆⋆	⋆⋆	⋆⋆⋆	⋆⋆
5	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	⋆
6	⋆⋆⋆	⋆⋆	⋆⋆	⋆⋆⋆
sum	16⋆	14⋆	13⋆	14⋆

Table 4.2 illustrates the concept scoring for ICTS and CBS algorithms based on the set of criteria from Section 3.2.1.

High-level algorithms		
Criteria	CBS	ICTS
1	⋆⋆	⋆
2	⋆⋆	⋆⋆
3	⋆⋆⋆	⋆
4	⋆⋆	⋆⋆
5	⋆⋆	⋆⋆
6	⋆⋆⋆	⋆
sum	14⋆	9⋆

The scoring in Tables 4.1 and 4.2 uses three degrees of ranking:
⋆ - denoting a low score
⋆⋆ - denoting an average score
⋆⋆⋆ - denoting a high score

The score of each algorithm is relative to one another, with evaluation of criteria 4, 5 and 6 being somewhat subjective.

4.2 Pseudo-code

4.2.1 A* code

To highlight our extentions to the A* we denote it TaN-A* and present the pseudo-code for it in Algorithm 5.

Algorithm 5 TaN-A* algorithm

```
1:  openSet ← startState
2:  while openSet not empty do
3:      currentState = min(openSet)
4:      openSet.remove(currentState)
5:      closedSet ← currentState
6:      if currentLocation == goalLocation then
7:          return path ← RectracePath()
8:      for neighbours of currentState do
9:          if neighbour in closedSet then
10:             continue
11:         if neighbourLocation has been visited then
12:             StateWeight → increase()
13:         else
14:             StateWeight = 0
15:         currentState.hCost ← update()
16:         stepCost = Distance(currentState, neighbour)
17:         newGCost = StateWeight + currentNode.gCost + stepCost
18:         oldGCost = neighbour.gCost
19:         if (newGCost < oldGCost) or (neighbour not in openSet) then
20:             neighbour.gCost ← update()
21:             neighbour.hCost ← update()
22:             neighbour.parent = currentState
23:             openSet ← neighbour
```

The states used in the TaN-A* algorithm are defined as an object containing; (1) a location in the configuration space, (2) a g-cost, (3) a h-cost, (4) a time representing the current time-step in the path being evaluated and (5) a parent state representing the previous state of the current state in the current path being evaluated. Note that the TaN-A* algorithm differs from the traditional A* with the implementation of time discretization and weighted scoring. The weighted scoring is based on the distance from the state location to the goal location and the number of times that location has been visited in the same path.

4.2.2 CBS code

The TaN-CBS used in the TaN-algorithm is very similar to the CBS presented in section 2.3.2 with the difference of implementing high-level nodes in conjunction with a closed set. This is done to prevent the algorithm from searching for the exact same solution more than once. Furthermore, the TaN-CBS continuously updates environment variables such as constraints which is heavily dependant on the XPlanar system.

Algorithm 6 TaN-CBS algorithm

```
1:  startNode = HighLevelNode()
2:  Initialize startNode.constraints
3:  solution = compute_solution(startNode)
4:  if solution not found then
5:      return
6:  startNode.cost = compute_cost(solution)
7:  openSet ← startNode
8:  while openSet not empty do
9:      currentNode = min(openSet)
10:     closedSet ← currentNode
11:     openSet.remove(currentNode)
12:     Environment ← currentNode.constraints
13:     solution = compute_solution(currentNode)
14:     conflicts ← firstConflict
15:     if conflicts empty then
16:         return solution
17:     else
18:         newConstraints = generate_constraints(conflicts)
19:         Environment ← newConstraints
20:     for each constraint in newConstraints do
21:         newNode = HighLevelNode()
22:         newNode.constraints ← currentNode.constraints
23:         newNode.constraints ← constraint
24:         solution = compute_solution(newNode)
25:         if solution not found then
26:             continue
27:         newNode.cost = compute_cost(solution)
28:         if newNode not in openSet then
29:             openSet ← newNode
30: return
```

The high-level nodes used in the TaN-CBS algorithm are defined as an object containing; (1) a solution defined as a collection of individual mover paths, (2) the total cost for the solution in number time-steps and (3) a collection of all constraints for the solution.

4.3 Overview of the architecture

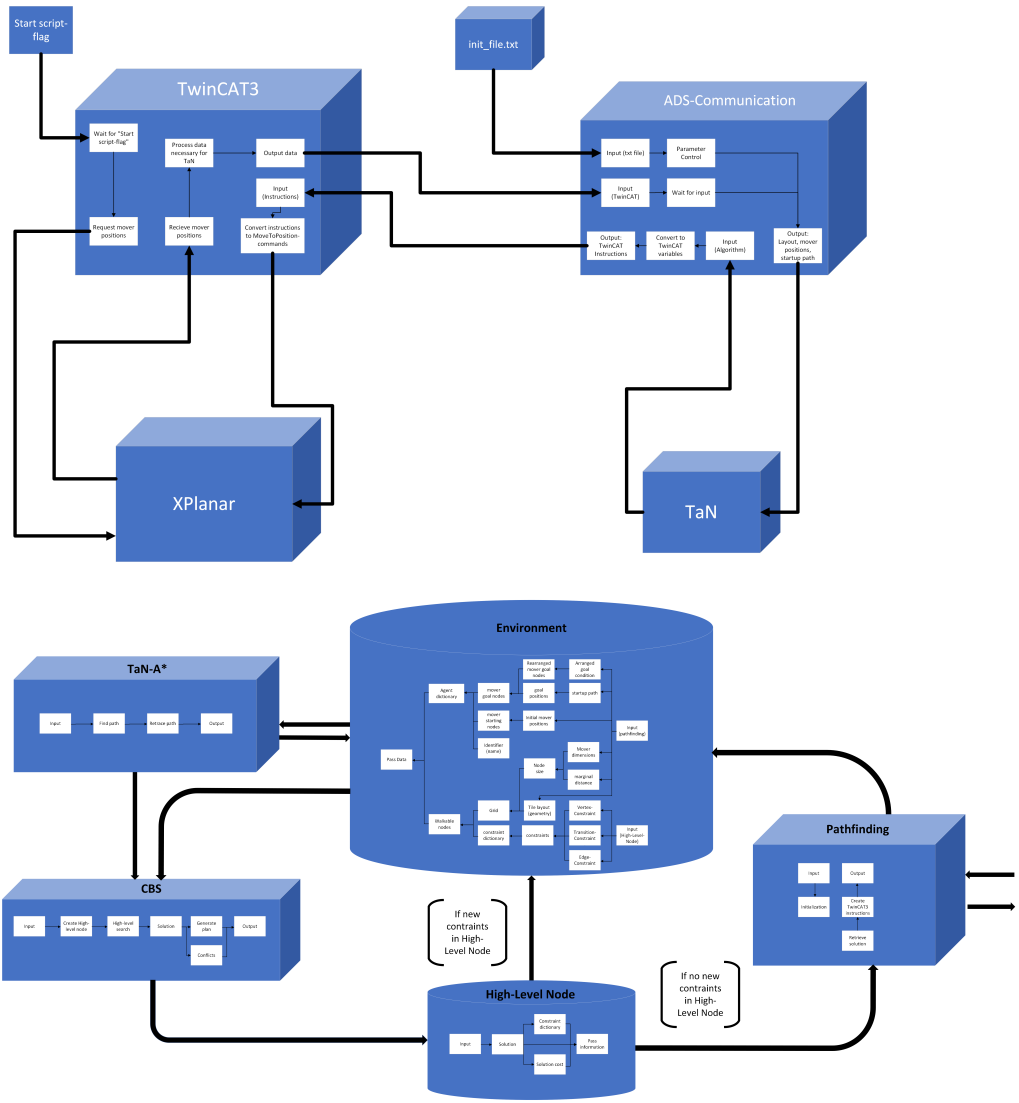


Figure 4.1 illustrates both the passing of data and the overlaying architecture for the complete solution and the entire TaN algorithm

The algorithm starts once it is called from TwinCAT, through a flag. TwinCAT then reads each movers' position and passes these positions along with other necessary data to the TaN algorithm through the ADS communication block. The TaN algorithm then starts by initializing the central storage unit environment. Thereafter, necessary data is passed to the TaN-CBS and TaN-A* algorithm. Once the TaN-CBS and TaN-A* algorithms have retrieved the necessary inputs, the TaN-CBS is called whereby it creates a high-level node. The high-level node is then filled by individual agent paths retrieved by calling and executing the TaN-A* algorithm. The Environment is then updated based on newly derived changes of the configuration space. The process of running the TaN-CBS and TaN-A* as well as updating the environment is performed until a complete viable solution has been found. This solution is then passed back to the pathfinding unit where the solution is reinterpreted as instructions. These instructions are then passed to the ADS block where they are translated to TwinCAT variables and passed right back to TwinCAT. Lastly, these instructions are converted to MoveToPosition-commands, which tells the movers to move step by step.

4.4 Experimental results

Due to limited access to the XPlanar, this section will briefly present the recorded physical test, see Figure 4.2. The video[1] shows four movers starting in one corner of a XPlanar setup, with an L-shaped startup path defining the goal positions located on the opposite corner. The first thing the movers do is line up in the invisible node-grid that is used in the algorithm. After that they move step by step in this node-grid approaching their respective goals. Once each mover reaches its goal node, they leave the node grid and align to the exact specified goal position defined by the startup path, described in millimeters.

Figure 4.2 illustrates the starting positions for the movers in the physical experiment.

[1] `https://youtube.com/shorts/ev-4OqVbQBM`

4.5 Simulation results

In Figure 4.9 the red line represents the minimum number of steps to make every mover move from their starting position to their goal position. To clarify, achieving the absolute minimum number of steps would mean every mover moved unhindered the linear minimum distance from their starting position to their goal position. This would put them exactly on the coloured line. Every test that is not a failure, meaning it lies on zero in the y-axis, is well above the line, and there are no bugs affecting the amount of mover steps. Figures 4.3, 4.4, 4.5, 4.5, 4.7, 4.8 and 4.9 depicts statistical data retrieved from the randomized tests described in Section 3.2.3, where each line is a regression-based approximation based on 1000 data-points each.

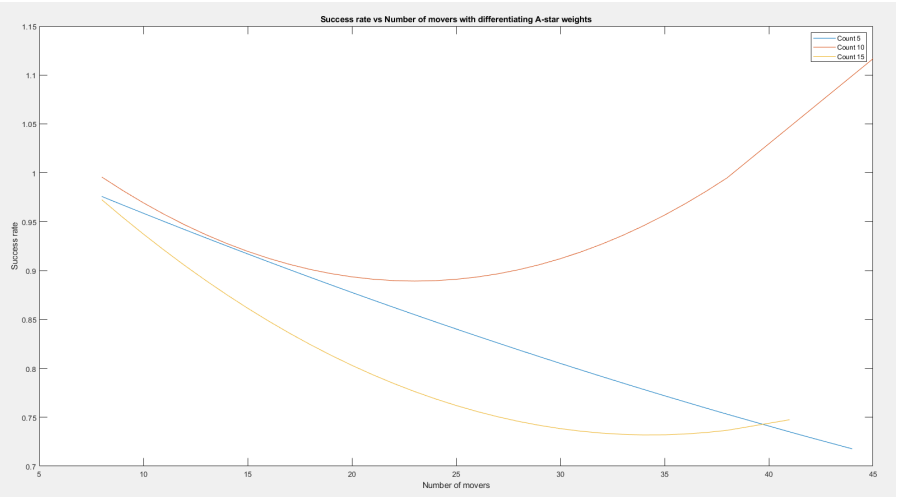


Figure 4.3 illustrates statistics for the randomized tests with the success/fail rate being plotted against the number of movers for different A* weight values (”Count”) 5, 10 & 15

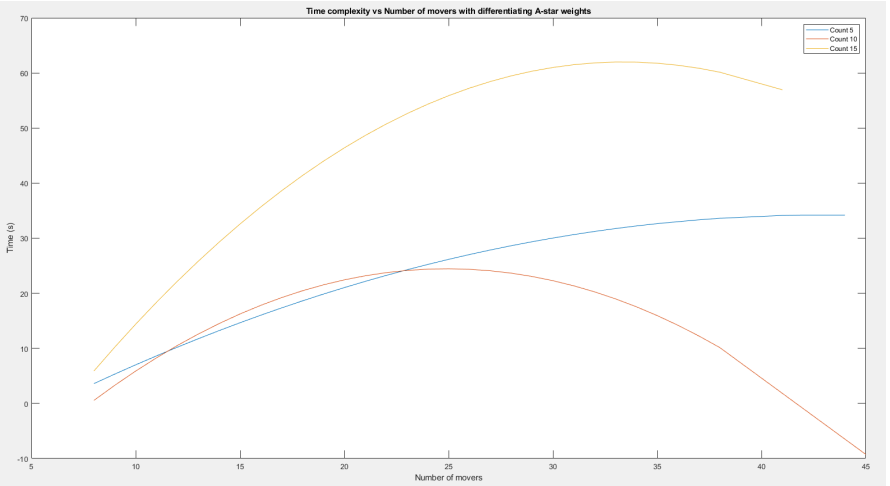


Figure 4.4 illustrates statistics for the randomized tests with the execution time being plotted against the number of movers for different A* weight values (”Count”) 5, 10 & 15

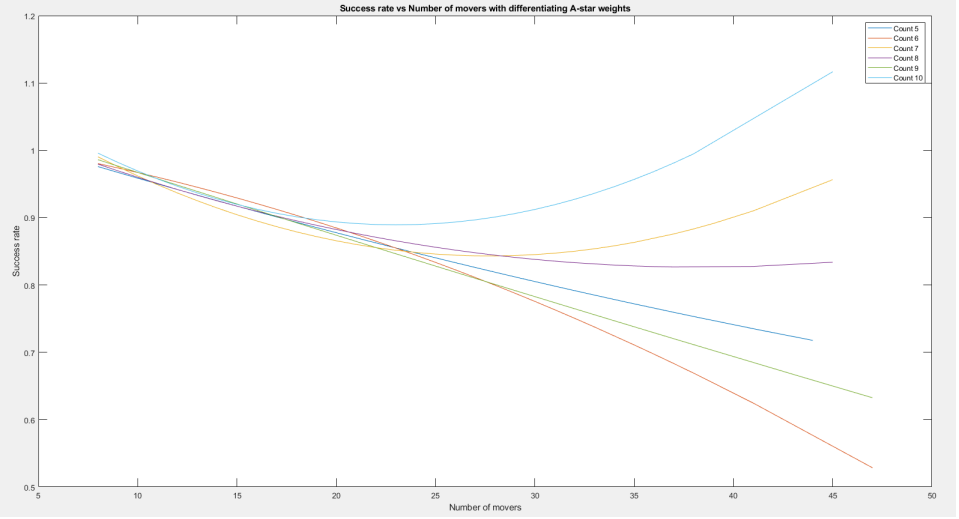


Figure 4.5 illustrates statistics for the randomized tests with the success/fail rate being plotted against the number of movers for different A* weight values (”Count”) 5-10)

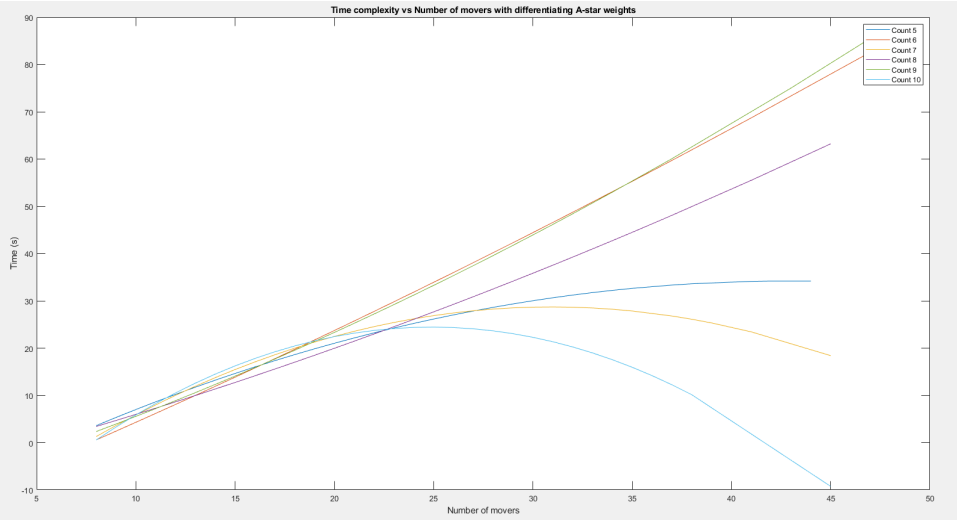


Figure 4.6 illustrates statistics for the randomized tests with the execution time being plotted against the number of movers for different A* weight values (”Count”) 5-10

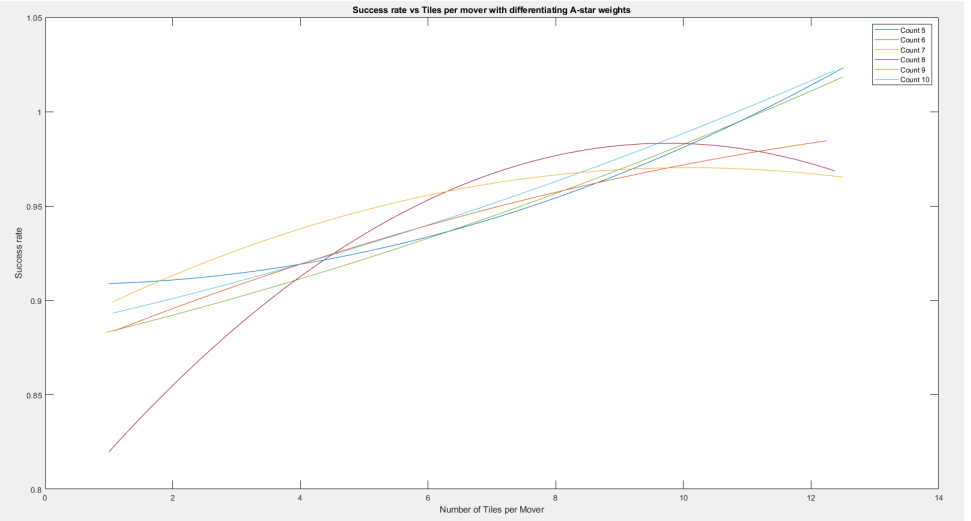


Figure 4.7 illustrates the success/fail rate for the A* weight (”Count”) values 5-10 plotted against number of tiles per mover

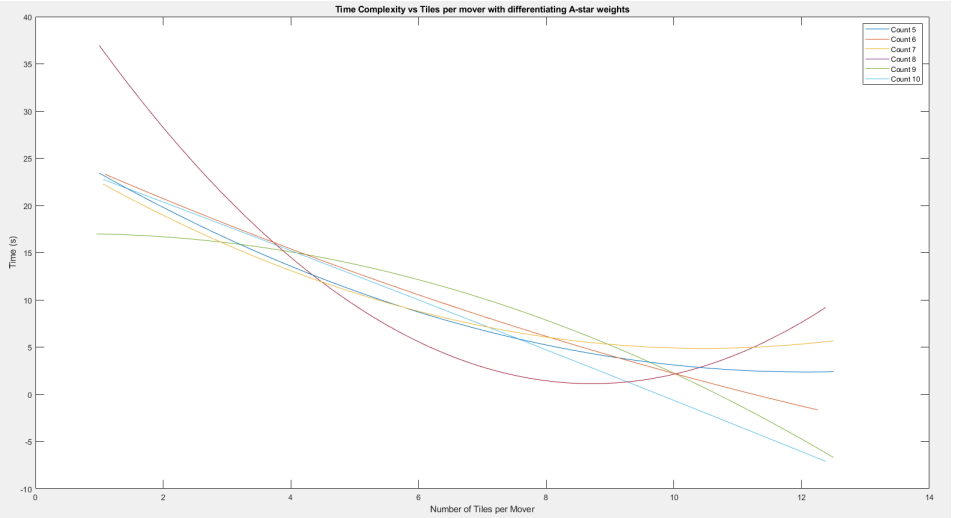


Figure 4.8 illustrates the time-complexity for the A* weight (”Count”) values 5-10 plotted against number of tiles per mover

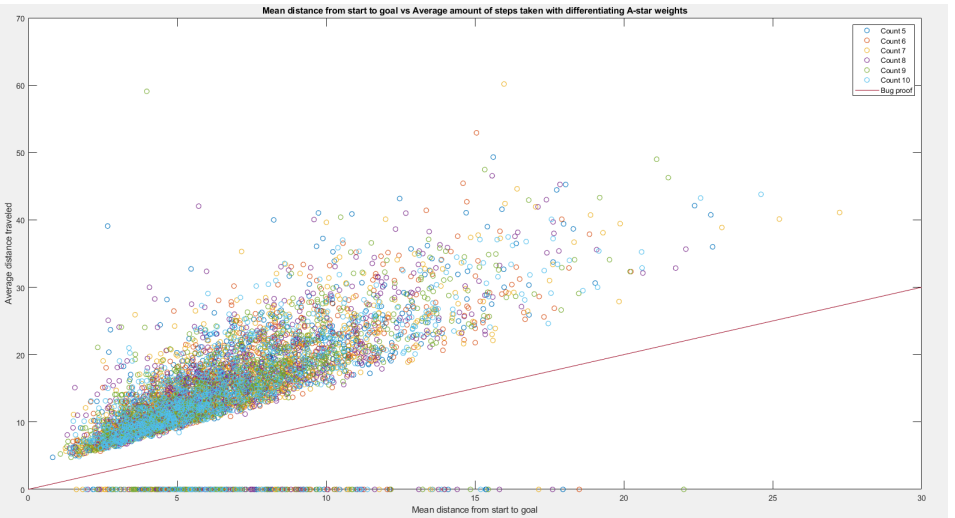


Figure 4.9 illustrates the mean distance for all movers from their starting position to their goal position plotted against the average number of time-steps for the A* weight (”count”) values 5-10

5

Discussion

5.1 Algorithms

5.1.1 Choice of algorithms

The A* and CBS algorithms were chosen based on the set of criteria presented in Section 3.2.1. From Section 2.4 and Table 4.2 it is clear that the CBS algorithm seems to be the top-candidate for both solvability and time-complexity. This is especially true with a higher number of movers which is more relevant in contrary to a low number of movers, since the differentiating execution-time between algorithms for a low number of movers is very low and becomes irrelevant. However, the CBS still needs a collaborative low-level algorithm in order to function as it is supposed to. In this project, the A* was chosen due to it is diversity and the overall scoring seen in Table 4.1. Being able to change heuristics, adding weights, etc. makes A* a well suited low-level algorithm in terms of increasing applicability for the XPlanar system. Furthermore, A* seemed to be very well documented with many previous implementation examples, making it a top candidate for the low-level algorithm in terms of conceptual complexity and complexity of code.

5.1.2 TaN-A* vs traditional A*

The traditional A* is designed to find a path for one agent, from a start position to a goal. The TaN-A* is modified to be able to handle multiple agents, take constraints into consideration, and is optimized to be faster than the average A*. A regular A* often uses two positions as input, the start and the goal. The TaN-A* instead takes a string as input, the name of an agent, and finds this agent's start and goal positions in a dictionary. This dictionary is stored in a separate object, which is used by both the CBS and the TaN-A* to keep track of different things, like agent's start and goal positions, obstacles and constraints. Constraints are something TaN-A* also uses when checking whether neighbouring states can be seen as viable states to visit next. Since TaN-A* is made to fit the TaN-CBS, it introduces time as a factor. This means its nodes become states, and it keeps track of time as well. When exploring neighbours the TaN-A* first evaluates the locations surrounding the current node,

and adds one time unit to these neighbours' current time, since that is how long it takes for a mover to traverse one grid-location. Due to the time discretization, the current location in the next time step is also seen as a viable state and a neighbour.

Another variant of A* is the weighted-A*, in which the scoring equations are modified to favour moving towards the goal, or moving away from the starting position. TaN-A* uses a sub-variant of this principle, the way it does this is by increasing states' score by a dynamic weight. In addition to this, the states' score is increased every time the states are visited. The exact increase of the score each time the state is visited is a penalty score of the distance from that state to the goal state divided by a weight. This weight was set to 10 in the final revision of the code by evaluating the results seen in Section 4.4. These results are further discussed in Chapter 5.2. This value was set to counter-act unsolvable cases, where agents kept moving to the same location instead of moving towards the goal. To hinder agents from moving back and forth between the lowest cost locations, a successively increasing cost was added, to make sure agents eventually started to move towards their goal.

5.1.3 Conflicts

In Section 2.2.2 the different conflicts for a typical MAPF problem were introduced. Whether these are handled correctly or not can be discussed, but from the unit-tests described in Section 3.1.3 it is safe to say that the edge- and vertex conflicts are handled and solved in a successful manner by having implemented constraints for these directly in the developed code. The following conflict and cycle conflict are handled in a slightly different way. The following conflict is implemented and solved by making each mover move one grid-location at a time. This makes sure that each move-action is performed with the same acceleration and speed for all movers. The cycle-conflict is handled and solved by always having one free grid-location available. By doing this, a potential cycle-conflict is automatically solved by having implemented the edge-, vertex- and transition constraints directly in the developed code. The special-case transition conflict in this system, seen in Figure 3.8 is solved by implementing the transition-constraint directly in the developed code. It is worth mentioning that transition conflicts can be neglected if agents have a circular geometry instead of a square geometry.

5.2 Performance evaluation

5.2.1 Solvability

Figure 4.3 depicts that a TaN-A* weight (”count”) value of 5 has a slow exponential performance whilst the weight values of 10 and 15 have a faster exponential performance. Overall, the count value of 10 seems to result in the highest solvability

with 5 as second highest, and 15 as the lowest. However, at 40 movers the values 5 and 15 result in an equal approximation of solvability, with 15 having a higher approximate solvability past 40 movers. Due to the fact that real XPlanar systems will not support more than 40 movers at the given time, these data points can be neglected in further discussion. Figure 4.3 also shows a faulty approximation for the weight value of 10 past 38 movers. Since a success is defined as a 1 and a fail is defined as a 0, a success approximation above 1 or below 0 shows a skewed faulty approximation. Because Figure 4.3 depicts a regression based approximation and not a 100% correct prediction, the faulty approximation does not necessarily reflect an error. The explanation to this is the possibility of a large number of data points with a success/fail value of 1 when the x-value approaches 38 movers. This in turn will increase the inclination and the upwards trend of the graph.

It is worth noting that the different weight values perform very differently for varying number of movers. However, this is expected. A lower TaN-A* weight is expected to affect the behavioural aspect in such a way that movers are less likely to wait whilst a higher value will result in movers being more likely to wait. Furthermore, since the weighted scoring works in conjunction with the distance to goal, the wait capability is enhanced near the goal positions where higher saturation of movers per tile is more likely to occur. From a theoretical perspective, it is difficult to decide whether a high or low wait capability is beneficial in regards to solving conflicts and improving solvability. Therefore, Figure 4.7 is used to closely examine the different TaN-A* weight values and wait capability for different saturation levels and how this in turn affects the solvability. Figure 4.7 shows that a lower TaN-A* weight and thereby a lower wait capability seems to result in a higher success rate for highly saturated areas in regards to mover per tile. On the contrary, a higher TaN-A* weight and higher wait capability has a higher success rate for lower saturated areas. This can be seen in the weight value 5 being the most successful in saturation levels near 1 mover per tile, value 7 being most successful for saturation levels between 2-6 movers/tile, 8 being optimal for saturation levels between 6-9.5 and value 10 being most successful past saturation level 9.5. The explanation for the different success rates likely depends on whether the wait capability helps resolve conflicts or not in more saturated areas. A higher wait capability could help resolve conflicts and improve solvability for a high number of movers as seen in Figure 4.3. However, it could also trigger a behaviour where the TaN-A* makes movers wait for a very long period of time. This likely occurs due to always choosing the same location for the next time step as the next optimal state to visit. This in turn could trigger the time-out interrupt in the TaN-CBS resulting in a failure. This behaviour also comes with restriction of overall movement which could limit the algorithm's ability to rearrange movers in such a way that finding a solution becomes more difficult.

For static weight values, the value of 10 seems to be the most consistently well performing with a very promising solvability approximation, as seen in Figure 4.5. Therefore, this was the weight value that was chosen in the final revision of the

TaN-A* algorithm.

5.2.2 Time complexity

Figure 4.4 depicts that the time complexity differs a lot from different count values, especially for a greater number of movers. This is likely due to the wait capability discussed in Section 5.2.1. A higher wait capability could in one way decrease the computational time by increasing the ability to resolve conflicts. On the other hand, it could also introduce a situation where movers are less prone to moving and rearranging such that a ”stand-still” situation occurs. In the case of a ”stand-still” it could take a lot of time to compute a solution. By comparing Figure 4.7 and Figure 4.8 one could argue that the time for the different TaN-A* weight values 5-10 in Figure 4.8 in highly saturated mover/tile areas would differ more, with higher count values resulting in a higher *time* value. This is a sound reasoning and the indifference could be explained by the time-out interrupt introduced in the TaN-CBS, capped at
240 s. Since highly saturated areas are less likely to be solved, as seen in Figure 4.7 it is also expected that a lot of the test runs will time-out.

If one examines the code instead of the graphs, a lower weight value means that locations' score increases faster, leading to less waiting, and more ”forced exploration”. This leads to more conflicts, as they are encouraged to move rather than wait. If they move more, odds are they reach the goal faster, however with more movement comes more conflicts, which means the CBS needs to run TaN-A* more times. A higher weight leads to the opposite, a slower increasing score, more waiting, more exploring to find the goal, and less conflicts. This in turn means TaN-A* becomes slightly slower, but in general fewer iterations are needed to find a complete solution. It is difficult to say which is better: fewer and longer iterations or more and faster iterations.

5.2.3 Quality control

Figure 4.9 concludes that the successful runs, those that do not have 0 number of time-steps, are placed above the minimum average time-steps line. This proves that no illegal move-actions are performed in regards to moving more than one grid-location at a time. Any data-point below the minimum average line would represent a solution where a mover would move several grid-locations at a time, and not conform to the restricted one grid-location for one time-step. This in turn could be potentially hazardous in regards to collision-avoidance. The reason being that a move-action for at least one of the movers in the configuration space is not performed in the expected and controlled manner it is supposed to.

5.3 Possible future improvements

Supporting obstacles in the tile-layout
The first possible improvement is implementing a function for passing in possible obstacles that might be situated throughout the tile-layout. This would allow the algorithm to work with XPlanar systems that, for example, has a wall separating two sections of the the tile-layout. From a first-thought perspective, this possible improvement could easily be implemented for very basic obstacle geometries. This could be done by:

1. Passing a matrix containing arrays with x- and y-positions in mm defining points at the outline of the geometry.
2. Iterating through each array and finding each grid-location where there is an obstacle-point.
3. Blocking each of the grid-locations found in (2) to deny movers access to those locations.

Dynamic TaN-A* weights
The second possible improvement is introducing dynamic TaN-A* weights. Since the optimization and functionality of TaN-A* seems to depend on different weight values for different amount of movers and tile-layouts, seen in Figures 4.3, 4.4, 4.7 and 4.8. The weights could change dynamically depending on configuration space variables. This could also be easily implemented since these variables already are accessible in the code.

D* implementation
From the A* theory in Section 2.3.1, it is noted that D* is one of the many improved algorithms that have originated from the A* algorithm. The D* is the third suggested improvement since it is supposed to be an overall improvement of the A* algorithm. The D* is in most cases more efficient than the A* in terms of computational time since it re-plans parts of paths once a conflict occurs, instead of scrapping the previous path-search and re-doing the entire path-search all over again. It is also possible that D* could improve solvability due to the incremental search and low-level conflict avoidance.

Multi-threaded solution
One way to potentially greatly decrease the computational time of the algorithm is by making it multi-threaded. A simple way to utilize multiple threads in this algorithm would be to make calculation of A* paths into tasks, which are executable by threads. This would make the computer able to calculate multiple paths simultaneously, and therefore theoretically greatly reduce computational time. A relatively simple way of implementing this would be with a thread pool, since in the case of having more agents than threads the A* tasks would queue up, and several single-agent path-searches could be performed simultaneously. This is however only a

”relatively simple” solution, since large parts of the algorithm would need to be reworked and made thread safe. The size of the thread pool would also be dependent on how many cores the computational unit would have, since having more threads than cores only increases computational time.

Testing

The tests for statistical data described in Section 3.1.3 were performed for randomized configuration spaces. However, in order to optimize the algorithm and further improve solvability, we propose continuing testing solvability and time-complexity with larger data sets for both randomized- and typical configuration spaces. This will also help derive the exact optimal weight value for the TaN-A* in XPlanar systems and whether a slightly lower or higher wait capability is most optimal for solving conflicts.

We also suggest adding a simple unit-test testing the mean distance to the goal against the actual number of instructions for each mover. This will ensure that further development does not introduce bugs connected to illegal multi-node move-actions.

6

Conclusion

By referring to the problem formulation in Section 1.3 and the goal of the project in Section 1.4 it can be concluded that the project was a major success. The final product shows a generally good applicability of the chosen algorithms towards the solution requested by Beckhoff, since it is functional both in theory and in practice. This project also provides a good starting point for continuing already started optimization of the algorithm by following the suggestions made in Section 5.3, Possible future improvements. Furthermore, we suggest using the statistical data presented in Section 5.2, our results, and the easily changeable parameters such as the TaN-A* weight to alter behaviour and optimize performance.

Bibliography

Andreychuk, A., K. Yakovlev, P. Surynek, D. Atzmon, and R. Stern (2022). "Multi-agent pathfinding with continuous time". *Artificial Intelligence* **305**, p. 103662. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2022.103662. URL: https://www.sciencedirect.com/science/article/pii/S0004370222000029.

Beckhoff (2018a). *Beckhoff automation XPlanar: flying motion*. https://www.beckhoff.com/en-en/company/press/pressemitteilungen_77892.html. [Online; accessed 26-June-2022].

Beckhoff (2018b). *Twincat automation software*. https://www.beckhoff.com/en-en/company/press/pressemitteilungen_77892.html. [Online; accessed 26-June-2022].

Chen, J. C. (2003). "Dijkstra's shortest path algorithm". *Journal of Formalized Mathematics* **11**, pp. 237–247. ISSN: 1426–2630.

Dijkstra, E. W. (1959). "A note on two problems in connection with graphs". *Numerische mathematik* **1**:1, pp. 269–271.

Duchoň, F., A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica (2014). "Path planning with modified A star algorithm for a mobile robot". *Procedia Engineering* **96**. DOI: 10.1016/j.proeng.2014.12.098.

Gange, G., D. Harabor, and P. Stuckey (2019). "Lazy CBS: implicit conflict-based search using lazy clause generation". English. In: Benton, J. et al. (Eds.). *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling*. Vol. 29. Proceedings International Conference on Automated Planning and Scheduling, ICAPS. Association for the Advancement of Artificial Intelligence (AAAI), pp. 155–162. URL: https://ojs.aaai.org/index.php/ICAPS/issue/view/239.

Goldenberg, M., A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. Holte, and J. Schaeffer (2014). "Enhanced partial expansion A*". *The Journal of Artificial Intelligence Research (JAIR)* **50**. DOI: 10.1613/jair.4171.

Javaid, A. (2013). "Understanding Dijkstra algorithm". *SSRN Electronic Journal*. DOI: 10.2139/ssrn.2340905.

Kaduri, O., E. Boyarski, and R. Stern (2021). "Experimental evaluation of classical multi agent path finding algorithms". In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 12. 1, pp. 126–130.

Kranzberg, M. and M. T. Hannan (2021). "History of the organization of work". In: *Encyclopedia Britannica*.

Langa, L. (2022). *The uncompromising code formatter*. https://pypi.org/project/black/. [Online; accessed 20-June-2022].

Lejeune, E. and S. Sarkar (2021). *Survey of the multi-agent pathfinding solutions*. DOI: 10.13140/RG.2.2.14030.28486.

Misa, T. (2010). "An interview with Edsger W. Dijkstra". *Commun. ACM* **53**, pp. 41–47. DOI: 10.1145/1787234.1787249.

Neumann, W. P., S. Kihlberg, P. Medbo, S. E. Mathiassen, and J. Winkel (2002). "A case study evaluating the ergonomic and productivity impacts of partial automation strategies in the electronics industry". *International Journal of Production Research* **40**:16, pp. 4059–4075. DOI: 10.1080/00207540210148862. eprint: https://doi.org/10.1080/00207540210148862. URL: https://doi.org/10.1080/00207540210148862.

Patel, A. (2022). *Amit's A* pages*. URL: http://theory.stanford.edu/~amitp/GameProgramming/Variations.html.

Permana, S., K. Bintoro, B. Arifitama, and A. Syahputra (2018). "Comparative analysis of pathfinding algorithms a *, dijkstra, and bfs on maze runner game". *IJIS-TECH (International Journal Of Information System Technology)* **1**, p. 1. DOI: 10.30645/ijistech.v1i2.7.

Python Code Quality Authority (2022). *Pylint*. https://pypi.org/project/pylint/. [Online; accessed 20-June-2022].

Python software foundation (2022). *Python 3.10.4 documentation*. https://docs.python.org/3/. [Online; accessed 20-June-2022].

Rahim, R., D. Abdullah, S. Nurarif, M. Ramadhan, B. Anwar, M. Dahria, S. D. Nasution, T. M. Diansyah, and M. Khairani (2018). "Breadth first search approach for shortest path solution in Cartesian area". *Journal of Physics: Conference Series* **1019**, p. 012036. DOI: 10.1088/1742-6596/1019/1/012036. URL: https://doi.org/10.1088/1742-6596/1019/1/012036.

Sharon, G., R. Stern, A. Felner, and N. Sturtevant (2012). "Conflict-based search for optimal multi-agent path finding". *Proceedings of the AAAI Conference on Artificial Intelligence* **26**:1, pp. 563–569. URL: https://ojs.aaai.org/index.php/AAAI/article/view/8140.

Sharon, G., R. Stern, A. Felner, and N. R. Sturtevant (2015). "Conflict-based search for optimal multi-agent pathfinding". *Artificial Intelligence* **219**, pp. 40–66. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2014.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0004370214001386.

Sharon, G., R. Stern, M. Goldenberg, and A. Felner (2013). "The increasing cost tree search for optimal multi-agent pathfinding". *Artificial Intelligence* **195**, pp. 470–495. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2012.11.006. URL: https://www.sciencedirect.com/science/article/pii/S0004370212001543.

Stern, R., N. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, and R. Bartak (2019). "Multi-agent pathfinding: definitions, variants, and benchmarks". DOI: 10.48550/ARXIV.1906.08291. URL: https://arxiv.org/abs/1906.08291.

Wikipedia (2022b). *A* search algorithm — Wikipedia, the free encyclopedia*. [Online; accessed 4-May-2022]. URL: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1082221896.

Wikipedia (2022d). *Breadth-first search — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Breadth-first\%20search&oldid=1084419557. [Online; accessed 25-April-2022].

Wikipedia (2022c). *D* — Wikipedia, the free encyclopedia*. [Online; accessed 5-May-2022]. URL: https://en.wikipedia.org/w/index.php?title=D*&oldid=1080879163.

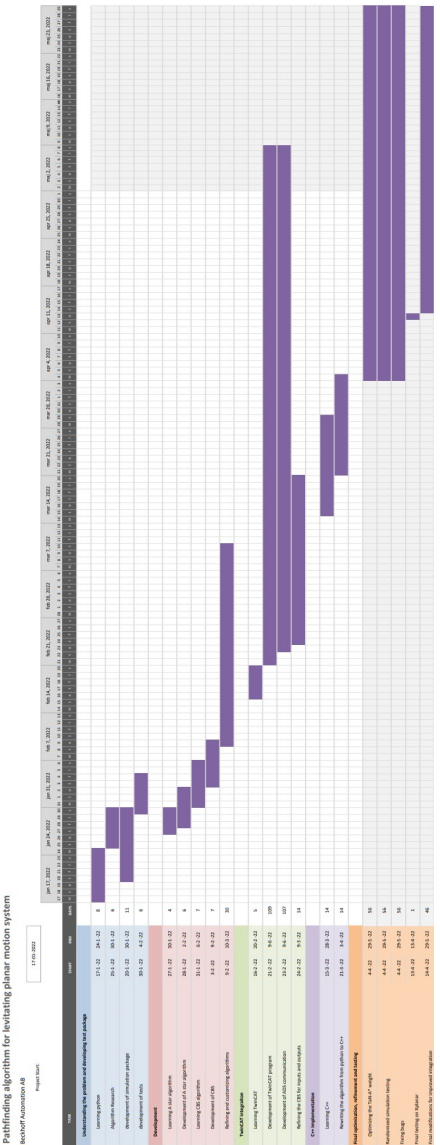
Wikipedia (2022e). *Depth-first search — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/w/index.php?title=Depth-first\%20search&oldid=1074848221. [Online; accessed 25-April-2022].

Wikipedia (2022a). *Dijkstra's algorithm — Wikipedia, the free encyclopedia*. [Online; accessed 4-May-2022]. URL: https://en.wikipedia.org/w/index.php?title=Dijkstra%5C%27s_algorithm&oldid=1086019825.

Wikipedia (2022f). *Taxicab geometry — Wikipedia, the free encyclopedia*. [Online; accessed 18-May-2022]. URL: https://en.wikipedia.org/w/index.php?title=Taxicab_geometry&oldid=1066973902.

A

Gantt chart for the project



Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER'S THESIS

Date of issue
June 2022

Document Number
TFRT-6170

Author(s)
Henry Nilsson
Johan Ternerot

Supervisor
Daniel Jovanovski, Beckhoff Automation AB
Anders Robertsson, Dept. of Automatic Control, Lund University, Sweden
Anders Rantzer, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle
Path planning algorithm for levitating planar motion system

Abstract

In today's world of ever increasing competitiveness, solutions that include automation and smart production have become a vital part to consider in overall business strategy, specifically for the industry sector. Whether an automated production process will be beneficial or not is dictated by how sub-processes such as interproduction transportation operates. The importance of these processes has been displayed in the late increase of production efficiency when moving from traditional transportation units, such as conveyor belts, to more sophisticated systems, such as transportation robots. However, these new sophisticated systems comes with increased complexity and new challenges when implementing important behaviours such as speed, control and safety.

This thesis is linked to the challenge of developing a safe and time efficient feature for handling a sudden failure or halt in one of these systems, namely the Beckhoff XPlanar levitating planar motion system. Hence, the goal of the thesis was to develop a pathfinding algorithm to easily line up the agents in the XPlanar system from any given position to a pre-specified startup track.

The end-result was a multi-agent pathfinding algorithm that utilizes Conflict-Based Search and A* to move each agent from their start position to a desired end-position whilst avoiding collisions. The algorithm is specifically designed towards the XPlanar system, integrated through ADS communication making it executable from the Beckhoff PC-based control software TwinCAT3.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-57

Recipient's notes

Security classification