Bachelor's Thesis

Multi-Agent Path Finding with Blocking Containers

Ben Bausch

Examiner: Prof. Dr. Bernhard Nebel
Adviser: Tim Schulte

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Foundations of Artificial Intelligence

August 29th, 2019

Writing period

29. 06. 2019 – 29. 09. 2019

Examiner

Prof. Dr. Bernhard Nebel

Advisers

Tim Schulte

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Zusammenfassung

In den letzten Jahren ist das Interesse an der Wegfindung für Multi-Agenten-Systeme stetig gestiegen, mit Anwendungen, die von automatisierten Lagern bis hin zur optimalen Routenführung autonomer Fahrzeuge durch den Verkehr reichen. In dieser Thesis, werde ich eine neuartige Lagerumgebung formalisieren, in der sich die Agenten unter den gelagerten Container bewegen können, solange sie nicht selbst einen Container transportieren. Ich werde einen CBS-basierten Algorithmus für die Abhol- und Zustellaufgabe in diesem neuen Lager einführen. Darüber hinaus werde ich einen neuen CBS-ähnlichen Algorithmus vorstellen, der eine größere Bandbreite von Probleminstanzen, die unter anderem Planung zur Lösung erfordern, bewältigen kann. Es werden Experimente durchgeführt und die beiden Algorithmen anhand von Benchmarks verglichen. Diese neuen Benchmarks bestehen aus bereits existierenden Benchmarks, die für die Abhol- und Lieferaufgabe erweitert wurden.

Abstract

Over the last years, there has been an ever increasing interest in path finding for multi-agent systems, with applications ranging from automated warehouses to routing autonomous cars optimally through traffic. In this paper, I will formalize a new kind of warehouse environment, where the agents are able to move under the stored containers, as long as they are not transporting a container themselves. I will introduce a CBS based algorithm for the pickup and delivery task in this new warehouse. Furthermore, I will present a new CBS-like algorithm, which can handle a wider range of problem instances, that require planning to solve. Experiments will be conducted and the two algorithms will be compared on benchmarks, created by extending all ready existing benchmarks to fit the pickup and delivery task.

Contents

1 Introduction

With diverse applications ranging from manoeuvring computer characters in video games and efficient path planning for robots in warehouses to autonomously routing cars through traffic, multi-agent path finding (MAPF) has become are very interesting and important topic in computer science. Instead of using planning algorithms, which are often not as efficient, these problems can often be defined as variants of the classical MAPF problem.

In the classic MAPF setting, a set of k, most of the time cooperative, agents $A = \{a_i, a_2, ..., a_k\}$ search for a path from their start positions $S = \{s_1, s_2, ..., s_k\}$ to their designated target positions $T = \{t_1, t_2, ..., t_k\}$. At each time step, agents can either stay at their position or move to an other location. The combination of all the single-agent paths is called a solution, and is the answer to the MAPF problem. A solution is called valid, if every agent is at its target location and there are no conflicts between the agents, for example no two agents are allowed to occupy the same location at the same point in time. A solution is called optimal, if it is the best overall solution for the problem instance given the constrains. Since a solution has to be valid in order to be applicable in real life, it is quite challenging to solve a MAPF problem. If the solution requires to be optimal, finding it becomes even harder. For this task, various algorithms have been proposed over the last few years. SAT-solvers, variants of A* and conflict-based search algorithms are the most popular ones and mostly consider a centralized computing setting, where the solution is being calculated by one single entity (one agent or one meta agent having complete knowledge of all the agents).

SAT-Solvers have first been introduced by Surynek in [13] back in 2012. The main idea is to start from a solution, obtained by other method like A*, divide it into small sub-sequences and then optimize those using a SAT solver. The encoding of the problem in variables plays an important role for such approaches, since SAT-solvers try to solve satisfiability for formulae based on those variables. These methods handle

small grids with high agent density quite well, but drop in performance as the grid size increases.

Variants of A*, as mentioned in [11], try to reduce the number of nodes in the open list of the A* algorithm, to limit the dimensional complexity of the problem. For the classical A* algorithm, we have to consider every possible action for every agent at each time step. This results in a branching of b^k at each time step, if every agent can choose between b actions. Some of these approaches use pattern databases, which calculate heuristics based on an abstraction of the state space, and/or Standley's improvements [12]. He introduced independence ID, which groups conflicting agent into group for which a nonconflicting solution is being calculated, and operator decomposition, considering only the best actions of agents one after another.

Conflict-based search (CBS) algorithms [2] are the main focus of this paper. CBS splits the multi-agent path finding task into k single-agent path finding problems. CBS is a two level algorithm, with a low level algorithm, that computes single-agent paths, and a high level algorithm, which resolves conflicts, like collision, between the k paths. For the low level algorithm, any single-agent path finding algorithm like A^* can be chosen. The high level algorithm compares the single-agent paths and tries to resolve the conflicts. CBS will be further explained in chapter 3. In the last few years, many extensions of the classical CBS have been proposed for example using heuristics in the high level algorithm [4] and assigning the tasks optimally to the agents [6].

In this thesis, we introduce a new formalism to describe a MAPF problem, where agents have to transport a set containers to a set of goal location, such a problem is know as multi-agent pick-up and delivery task (Figure 2). The difference to the new formalism is that containers are defined as agents and therefore planning can be performed from the containers perspective. I will change CBS to fit this task and extend it to fit instances, which require containers, that do not have a target other then their start position, to be moved and therefore require planning to be solved.

2 Background

In this chapter, we describe the classical multi-agent path finding formalism. The formalism will then be extended to the multi-agent pick-up and delivery domain. The two formal descriptions permit to understand the two domains more profoundly and defined a precise formulation of a given problem instance.

2.1 Classical Multi-Agent Path Finding

A classical multi-agent path finding (MAPF) problem can be defined as a tuple:

$$\Gamma =< G, A, s, t >$$

Where:

- $G =< V, E >$ is a graph, representing the environment
  - V is a set of vertices $V = \{v_1, v_2, ..., v_n\}$, representing the set of possible locations
  - E is a set of edges $E = \{e_1, e_2, ..., e_m\}$, describing possible transition between two locations
- A is a set of k agents $A = \{a_1, a_2, ..., a_k\}$
- $s : A \to V$ is an injective function, that maps each Agent $a \in A$ to a start position $v_s \in V$
- $t : A \to V$ is an injective function, that maps each Agent $a \in A$ to a goal position $v_g \in V$

In [1], an action is a function $a : V \to V$, such that $a(v) = v'$. This means, that taking action a in vertex v, results in a position transition of the agent to v'.
The following definitions of actions will serve as more explanatory and planning-like definitions, but will not change the meaning of the classical MAPF action definitions. Actions consist of a precondition χ, which has to be evaluated to true for the action

to be applicable, and an effect e, which describes how the world will change after performing the action. Actions are a tuple: $a =< \chi, e >$.
The precondition and the effect are logical formulae, possibly containing the predicates:

- $at(a, v)$, true if agent a is at vertex v
- $edge(v_i, v_j)$, true if there is an edge between the vertices v_i and v_j

Usually, agents can perform 2 different actions, they can either move or stay:

- the agent moves from vertex v_i to an adjacent vertex v_j:
  $move(a, v_i, v_j) =< (at(a, v_i) \wedge edge(v_i, v_j)), (\neg at(a, v_i) \wedge at(a, v_j)) >$
- the agent stays at the current position for another time step:
  $wait(a, v_i) =< (at(a, v_i)), (at(a, v_i)) >$

We also have to define a cost-function c, that assigns a cost to each action. Typically, a unit-cost-function, that assigns the same cost of 1 to each action, is used.
As in [1], we define a sequence of actions of one agent a_i as followed $\pi(a_i) = (x_1, x_2, ..., x_n)$ and let $\pi_i[x]$ be the vertex v after executing the first x actions of the sequence. A single-agent path is a sequence of actions $\pi(a_i)$, so that $\pi_i[0] = s(a_i)$ and $\pi_i[|\pi(a_i)|] = t(a_i)$. All the actions, that lead to the vertex transitions within the path, need to be applicable. This means, that the agent needs to be able to apply the action $a(v) = v'$ if v precedes v' in the agents path. A solution to the MAPF problem is a set of single-agent paths for all the k agents, so that no pair of single-agent paths results in a conflict.
Many different types of conflicts can be defined. Depending on the setting of the MAPF problem, we can consider different conflict types. The most common conflicts are the following:

- **vertex conflict**: two agents occupy the same vertex at the same time,
  $\pi_i[x] = \pi_j[x]$
- **edge conflict**: two agents move from the vertex v to the vertex v',
  $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$ (implies a vertex conflict before and after execution of action x)
- **Following conflict**: an agent moves to a vertex, just left by another agent,
  $\pi_i[x+1] = \pi_j[x]$

- **Cycle Conflict**: every agent moves to a vertex previously occupied by another agent,
  $\pi_i[x+1] = \pi_{i+1}[x]$ and $\pi_{i+1}[x+1] = \pi_{i+2}[x]$ and ... and $\pi_{k-1}[x+1] = \pi_k[x]$ and $\pi_k[x+1] = \pi_i[x]$

- **Swapping Conflict**: two agents swap vertices over the same edge,
  $\pi_i[x+1] = \pi_j[x]$ and $\pi_i[x] = \pi_j[x+1]$

2.2 Multi-Agent Pickup and Delivery

In the following, we describe a multi-agent pick-up and delivery (MAPD) problem, so that the containers are a new type of agents. This makes it possible to describe algorithms, that plan from the containers perspective. Changing the planning perspective might reduce the complexity of the problem. We adapt the previously introduced formalism to fit the new setting.
A MAPD problem:

$$\Gamma =< G, A, C, s, s', t' >$$

Where:

- G, A and s are defined as for the classical MAPF problem
- C is the set of m containers, that need to be transported to a location, $C = \{c_1, c_2, ..., c_m\}$
- $s' : C \to V$ is an injective function, that maps each container $a \in A$ to a start position $v_s \in V$
- $t' : C \to V$ is an injective function, that maps each container $a \in A$ to a goal position $v_g \in V$

We can formalize a MAPD problem in two different ways, once from the perspective of the agents and once from the perspective from the containers. Since we focus on algorithms, that try to solve the problem instances based on the agents point of view, we will first formalize the MAPD task for the agents perspective.

2.2.1 Agent Perspective

As we introduce a new type of agent, which changes the physics of the environment, we define a new set of possible actions, which can either be a univariate function $a(v) = v'$ or a bivariate function $b(v, x) = (v', y)$. Univariate action functions are actions, that only move the agent. Bivariate action functions describe actions, that move the agent and a specific container. To better understand the new dynamics, we introduce a planning-like formalism with predicates for the preconditions and effects:

- $at(a, v)$, true if agent$a$ is at vertex v

- $at(c, v)$, true if container c is at vertex v
- $edge(v_i, v_j)$, true if there is an edge between the vertices v_i and v_j

To fit the new dynamics of the environment, we introduce the following actions:

- $move_a(a, v_i, v_j)$ moves agent a from vertex v_i to the adjacent vertex v_j
- $move_c(a, c, v_i, v_j)$ moves the agent and the container from vertex v_i to adjacent vertex v_j
- $wait_a(a, v_i)$ lets the agent stay in its current vertex

The preconditions of the actions are the following:

- $pre(move_a(a, v_i, v_j)) = at(a, v_i) \wedge edge(v_i, v_j)$
- $pre(move_c(a, c, v_i, v_j)) = at(a, v_i) \wedge at(c, v_i) \wedge edge(v_i, v_j)$
- $pre(wait_a(a, v_i)) = at(a, v_i)$

The effects of the actions are :

- $\text{eff}(move_a(a, v_i, v_j)) = at(a, v_j) \wedge \neg at(a, v_i)$
- $\text{eff}(move_c(a, c, v_i, v_j)) = at(a, v_j) \wedge \neg at(a, v_i) \wedge at(c, v_j) \wedge \neg at(c, v_i)$
- $\text{eff}(wait_a(a, v_i)) = at(a, v_i)$

We assign a unit-cost of 1 to all the actions, meaning each action will take 1 time step to be executed. This can be changed to better suit real life settings.

To formalize conflicts, we will use the function $\pi_i[t] = v$, the same as in classical MAPF, and a new function $\gamma_i[t] = (t, v_t)$, that returns a tuple of the container affected by the agents action and its position after agent i executed the action at time step t in its plan (see Figure 1).
In MAPD, we will only be interested in the following conflicts:

- **agent vertex conflict**: two agents occupy the same vertex an the same time, if $\exists a_1, a_2 \in A$: $(\pi_{a_1}[t] = \pi_{a_2}[t])$
- **container vertex conflict**: two containers occupy the same vertex, if $\exists a_1, a_2 \in A \wedge \exists c_1, c_2 \in C : \gamma_{a_1}[t] = (c_1, v_i) \wedge \gamma_{a_2}[t] = (c_2, v_i)$

- **container conflict**: a container c is at two different positions for the same time step in two different paths:
  if $\exists a_1, a_2 \in A$: $\gamma_{a_1}[t] = (c, v_1) \neq \gamma_{a_2}[t] = (c, v_2) \wedge v_1 \neq v_2$

- **Swapping Conflict**: two agent swap vertices over the same edge,
  if $\exists a_1, a_2 \in A$: $(\pi_{a_1}[t] = \pi_{a_2}[t-1] \wedge \pi_{a_1}[t-1] = \pi_{a_2}[t])$

As in classical MAPF, we can try to optimize two types of cost functions:

- **Makespan** is defined as the maximum length over all single-agent plans: $max_{1<i<k}(|\pi_i|)$ for all k agents . In the MAPD setting, this corresponds to the time, it took all the containers to reach their goal locations.

- **Sum of costs** is the sum over all paths: $\sum_{n=1}^{k} |\pi_i|$ for all k agents. In the MAPD setting, the sum of costs represents the sum of all the container path lengths.

The agents behaviour upon reaching a goal position has to be formalized as well. Most MAPF algorithms consider the following agent behaviours:

- The agent vanishes upon reaching a goal state.

- The agent does not move after reaching a goal state.

For the containers $c \in C$, the goal state is being at the location defined by the function t', which assigns each container to a goal position. For an agent $a \in A$, a goal state is occupying any position if all of its assigned containers have reached their goal position. A container assignment defines, which agents are responsible for which containers. For planning algorithms, that assign a fix set of containers to an agent, this definition implies the agents goal location to be the goal position of the last transported container, since as soon as the agent reaches the last containers goal location, it has reached its overall goal state.
We change the notion of a well-formed problem instance, introduced in [7], to fit the new formalism of the MAPD domain. A problem instance is called **well-formed** if the following conditions hold:

a) For all containers, there is a path between its start location and its goal location, that is not blocked by another container.

b) For all containers, there is a path between its start location and its goal location, that does not traverse another containers goal position.

c) Each agent can stay in its start location, without blocking another agents path.

Condition a) and b) guarantee, that every container will arrive at its goal position. Condition c) is needed for the same reason if not every agent has been assigned to a container.
A problem instance is called **semi-well-formed** if the following conditions hold:

a) For all containers, there is a path between its start location and its goal location, that does not traverse another containers goal location.

b) Each agent can stay in its start location, without blocking another agents path.

2.2.2 Container Perspective

If we want to formalize algorithms for the MAPD domain, that plan from the containers perspective, we need to introduce a few changes to the action definitions and the conflicts. Actions are either univariate functions $a(v) = v'$, moving only the container, or bivariate functions $b(v, x) = (v', y)$, that move the container and a specific agent.
To better understand the new actions, we use a planning-like formalism with the same predicates as in 2.2.1:

- $move_a(a, v_i, v_j)$ moves agent a from vertex v_i to the adjacent vertex v_j
- $move_c(a, c, v_i, v_j)$ moves the agent and the container from vertex v_i to adjacent vertex v_j
- $wait_c(c, v_i)$ lets the container stay at current position

The preconditions of the actions are the following:

- $pre(move_a(a, v_i, v_j)) = at(a, v_i) \wedge edge(v_i, v_j)$
- $pre(move_c(a, c, v_i, v_j)) = at(a, v_i) \wedge at(c, v_i) \wedge edge(v_i, v_j)$
- $pre(wait_c(c, v_i)) = at(c, v_i)$

The effects of the actions are :

- $\text{eff}(move_a(a, v_i, v_j)) = at(a, v_j) \wedge \neg at(a, v_i)$
- $\text{eff}(move_c(a, c, v_i, v_j)) = at(a, v_j) \wedge \neg at(a, v_i) \wedge at(c, v_j) \wedge \neg at(c, v_i)$

- $\text{eff}(wait_c(c, v_i)) = at(c, v_i)$

We redefine the $y_i[t] = (a, v)$ to return a tuple of the agent affected by the containers action at time step t and its new location. If we plan from a containers perspective, we need to change the formalism of the conflicts as well:

- **agent vertex conflict**: two agents occupy the same vertex an the same time, if $\exists a_1, a_2 \in A$ and $c, c' \in C$: $(\gamma_c[t] = (a_1, v_i) \wedge \gamma_{c'}[t] = (a_2, v_i))$
- **container vertex conflict**: two containers occupy the same vertex, if $\exists c_1, c_2 \in C : (\pi_{c_1}[t] = \pi_{c_2}[t])$
- **agent conflict**: two containers move the same agent to two different positions, if $\exists c_1, c_2 \in C \wedge \exists a \in A : \gamma_{c_1}[t] = (a, v_j) \wedge \gamma_{c_2}[t] = (a, v_k) \wedge v_j \neq v_k$
- **Swapping Conflict**: two agent swap vertices over the same edge, if $\exists a_1, a_2 \in A$ and $c_1, c_2, c_3, c_4 \in C$: $\gamma_{c_1}[t] = (a_1, v) \wedge \gamma_{c_2}[t] = (a_2, v') \wedge$ $\gamma_{c_3}[t-1] = (a_1, v') \wedge \gamma_{c_4}[t-1] = (a_2, v)$

The rest of the formalism does not change depending on the planning perspective.

2.2.3 Examples: Conflict and Problem Instance

In the following, we will show a graphical representation of all the conflicts defined for the MAPD task.

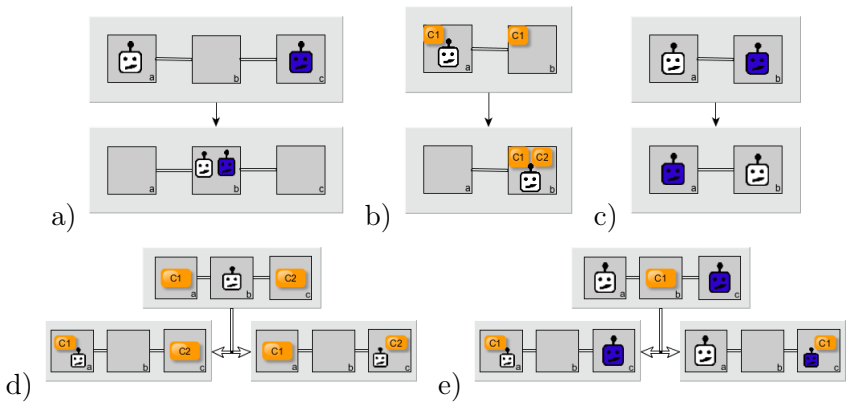


Figure 1: a) agent vertex conflict, b) container vertex conflict, c) swapping conflict, d) agent conflict e) container conflict

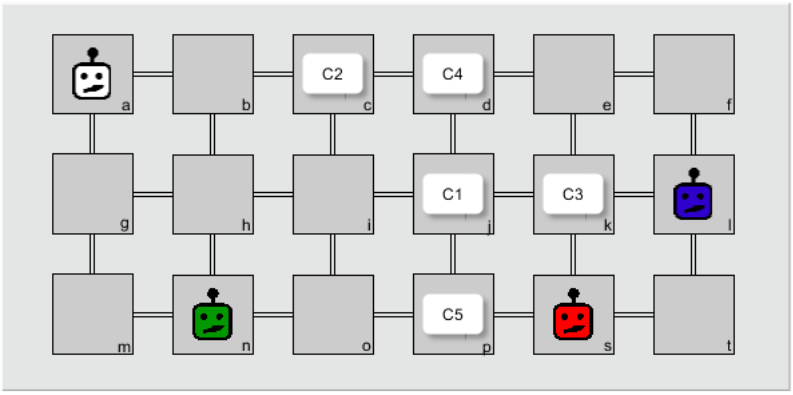


Figure 2: A possible problem instance:

- A = {white, green, blue, red}
- C = {C1, C2, C3, C4, C5}
- s:{(*white* →a),(*green* →n),(*blue* →l),(red→s)}
- s': {(C_1 →j),(C_2 →c),(C_3 →k),(C_4 →d),(C_5 →p)}
- t': {(C_1 →m),(C_2 →g),(C_3 →a),(C_4 →t),(C_5 →i)}

The problem presented in Figure 2 consists of a 6x3 graph, a set of four agents A and a set of five containers C. The agent start function s assigns a start position to each agent. *White* starts at a, *green* at n, *blue* at l and *red* at s. The container start function s' assigns the container start location. $C1$ starts at j, C_2 at c, C_3 at k, C_4 at d and C_5 at p. The container target function t' maps the containers to their target location. C_1 needs to arrive at m, C_2 at g, C_3 at a, C_4 at t and C_5 at j. This problem instance is not well-formed, since the all the paths for C_3 are blocked by another container.

3 Approach

Since multi-agent path finding algorithms are often much more efficient than multi-agent planning algorithms, we use a state of the art algorithm for classical MAPF to solve this problem. Diverse methods, like SAT solvers, variants of A* and conflict based search algorithms, can be applied to solve this task. We chose CBS to solve this MAPD task and extend it to solve problem instances, where all possible paths for a sub task might be blocked by containers. CBS has been shown to significantly outperform A^* and performs on par with other multi-agent path finding algorithms. Furthermore, classical CBS can be improved by introducing heuristics on the high-level search, as shown in [4]. Sub-optimal variants like Greedy CBS [3],which can handle problem instances with a large amount of agents, do also exists, but will not be further discussed in my thesis.

3.1 Conflict Based Search with Fixed Container Assignment CBS-FCA

We used CBS with a fixed container assignment to solve the problem optimally. CBS has first been introduced by [2] in the year 2012, since then further improvements have been published. Due to time restrictions, we focus on the classical variant of CBS, but we will mention further improvements to the proposed algorithms.

3.1.1 The High-Level Algorithm

Instead of finding a solution for the agents all together, as a joint agent, CBS splits the task into multiple single-agent path finding tasks and performs a two level search. The high-level Algorithm 1 does not change with respect to the classical CBS algorithm except for the lines highlighted in blue. It performs a best first search on the search nodes in the OPEN list, which each consist of a set of constraints, a solution and a cost value. The first search node has an empty constraints set, so the agents do not have any restrictions for the low-level search. If no conflict between the single-agent

Algorithm 1: CBS-FCA High-Level

Input: MAPF instance and fixed container assignment for each agent
OPEN ← an empty list
$Root.constraints = \emptyset$
$Root.solution$ = find individual paths with the low-level() for each agent
$Root.cost$ = SIC($Root.solution$)
insert *Root* into OPEN
while *OPEN not empty* **do**
    P ← best node from OPEN
    Validate the paths in P until a conflict occurs.
    if *P has no conflict* **then**
        return *P.solution* //P is goal
    C ← First Conflict found.
    if *(C is a Agent Vertex Conflict)* **then**
        //C is a Conflict of type (a_i, a_j, v, t)
        foreach *agent a in C* **do**
            $new\_constraint = (a, v, t)$
            Create_new_node(P, new_constraint, a)
    else if *(C is a Swapping Conflict)* **then**
        //C is a Conflict of type (a_i, a_j, v_a, v_b, t)
        foreach *agent a in C* **do**
            $new\_constraint = (a, v_l, v_k, t)$
            //v_l is the position of a before time step t
            //v_k is the position of a after time step t
            Create_new_node(P, new_constraint, a)
    else if *C is a Container Conflict* **then**
        //C is a Conflict of type (a,c_a,c_i, v, t)
        $new\_constraint = (c_a, v, t)$
        Create_new_node(P, new_constraint, a)

paths is found, we know that we have found the best possible solution, lines 10-11, due to performing a best first search. Otherwise, the high-level algorithm will generate new nodes depending on the conflict found, see lines 12-28. Each newly generated node copies the constraints of its parent and adds one additional constraint to prevent the conflict found upon validating the solution of the parent. We update the path of the agent affected by the constraint as describes in Algorithm 2. In section 3.1.6, you can find an example for the CBS-FCA algorithm.

Algorithm 2: Create new node

1 **Input**: P the parent node, C a new constraint and an agent a
2 $A \leftarrow$ new node
3 $A.constraints \leftarrow P.constraints + C$
4 $A.solution \leftarrow P.solution$
5 Update $A.solution$ by invoking low-level path finding for a
6 $A.cost$ = SIC($A.solution$)
7 **if** $A.cost < \infty$ *//A solution was found* **then**
8 | Insert A to OPEN

3.1.2 Fixed Container Assignment and Planning Perspective

Let the problem instance have K agents and M containers.

In CBS-FCA, each agent is responsible for n containers, with $n \in [0, M]$, no two agents are assigned to the same container and each container is assigned to exactly one agent. The container assignment function f maps from the set of containers to the set of agents, $f : M \rightarrow K$. We can formalize these properties as follows:

- *if* $M < K$ *then* f is injective
- *if* $M > K$ *then* f is surjective
- *if* $M = K$ *then* f is bijective

The assignment A_{a_i} for an agent a_i is an ordered list of containers, where the first container has to be delivered first, the second container second and so forth. This allows to implement deliver priorities on containers. If the delivery order does not matter, containers can be arranged in any permutation in the list.

Each agent will bring its containers to their goal position in the order defined

in the assignment. The implementation of CBS-FCA focuses on planning from the agents perspective. We could also implement the algorithm from the containers perspective, but this will not be further discussed in this paper. It does not matter from which perspective the path finding is done, since we can transform each action defined for the agents perspective in an action defined for the container perspective.

3.1.3 Adding New Nodes to the OPEN List

The high-level algorithm of CBS-FCA validates a nodes solution until a conflict between two agents is found, but does not check if further agents are involved in the conflict. This is the same procedure as in [2]. The number of nodes added to the OPEN list depends on which conflict is found:

Agent Vertex Conflict (lines 13-17):
As in [2], CBS-FCA will add two new nodes to the OPEN list, if the conflict found is a agent vertex conflict (a_i, a_j, v, t). For each agent a_i involved in the conflict, a new node will be generated prohibiting the agent to enter vertex v at time step t. In other words the tuple (a_i, v, t) will be added to the constraints of the new node.

Swapping Conflict (lines 18-24):
Consider the conflict to be a swapping conflict (a_i, a_j, v_a, v_b, t), CBS-FCA will add one node for each agent in the conflict. The node for agent a_1 will have the additional constraint (a_1, v_a, v_b, t), prohibiting the agent a_1 to move from vertex v_a to the vertex v_b at time step t. The second node will get the constraint (a_2, v_b, v_a, t), preventing the agent a_2 from transition from vertex v_b to vertex v_a at time step t.

Container Vertex Conflict (lines 25-29):
In contradiction to the previous two conflicts, CBS-FCA will only add one new node to the OPEN list, if the conflict found is a container vertex conflict (a, c_a, c_i, v, t). It is important to notice, that a conflict is only classified as a container conflict, if an agent a moves with its container c_a to a vertex v already occupied by container c_i, not yet reached by its agent a_i. If both containers are being transported by their agents and a conflict occurs, it will be classified as an agent vertex conflict. The constraint (c_a, v, t) will be added to the constraints for the new node and will prevent the agent to move to vertex v with its container c_a at time step t.

Container Conflict:
In the container assignment for the CBS-FCA algorithm exactly one agent is assigned to each container, therefore it is impossible for container conflicts to occur. There can not be two agents, that plan to move the same container at the same time.

3.1.4 The Low-Level Algorithm

Algorithm 3: TA* Low-level

Input:

- ordered list of of goal positions L
- start position s
- a graph G
- set of constraints C

$time \leftarrow 0$
$new\_start \leftarrow s$
$path \leftarrow$ empty List
foreach *goal in* L **do**
    $sub\_path \leftarrow$ find path with A^*(G,new_start, $goal$, $time$)
    $time \leftarrow time + length(sub\_path) - 1$
    $new\_start \leftarrow goal$
    Append sub_path to path
return path

For the low-level search, we introduce our new algorithm called *multi-target* A^* (TA^*), see Algorithm 3. Given a set of locations, which have to be visited in a certain order, TA^* will find the optimal path by finding the optimal sub-paths between two locations using A^*. The optimality of the sub-paths implies the path to be optimal as well. Since A^* has to respect the time step, in which the agent transitions to a new vertex in order to keep the path consistent with the constraints, it is important to pass the length of the previous sub-paths (time) to the next invocation of A^*. For the A^* heuristic, we chose the direct distance, which is admissible and therefore guarantees optimality.

In the following, we will show, that this algorithm will return a optimal solution, if the problem instance is well-formed and might not return a solution at all otherwise.

3.1.5 Theoretical Analysis

Assumptions:

- Upon reaching its container, the agent will not move alone anymore until it reaches the containers goal position.
- Let K be the number of agents
- Let M be the number of containers, having to be transported to goal location other than their start location
- The problem instance is well-formed.

First Case: $K \geq M$

Lemma 1: *For every agent, CBS-FCA will find an optimal path to its containers start positions*

Proof 1:

Since CBS is optimal and complete, CBS-FCA will always find a solution, if the problem instance is well-formed. If the problem instance is well-formed, no container or agent will be blocked by another agent without being able to solve the conflict via replanning. Agents, that do not carry a container, can only be involved in two types of conflicts, swapping and vertex conflicts. Finding the optimal path from an agents start position to a containers start position or from a containers goal position to another containers start location, if more than one container is in the agents assignment, does not change the classical MAPF problem. This implies, that CBS-FCA will find such a path for each agent. □

Lemma 2: *For every container, CBS-FCA finds an optimal path to its goal location*

Proof 2:

If there is no conflict for each pair of container paths, it is obvious, that simply applying CBS-FCA with TA* as low-level Algorithm will return an optimal path for each container.

If a conflict is found, we have to consider two environment states, in which the conflict could have occurred:

Each agent transports a container: Path finding will correspond to classical path finding, since the conflict will be resolved by adding two nodes with an additional agent vertex constraint, like in the classical version of CBS. If we have a vertex conflict on vertex v_i between two agent, a_1 and a_2, each transporting a container c_1 and c_2 at time step t, we add two CBS nodes, each with one of the constraints (a_1, v_i, t) and

(a_2, v_i, t). We could also add two container conflict nodes, but since the agent will not drop the container, once they picked it up, these nodes would return exactly the same solution as the nodes with the agent constraints.

Not every agent has reached its container: The only new conflict, that could interfere with the optimality of CBS-FCA, is the container vertex conflict, due to only adding one node for the agent moving with the container. A node with a constraint for the agent, not having reached its container at that point in time, would not return a solution since, the agent already follows the optimal path to the container, and can therefore not find any faster path to be consistent with the constraint. Thus only adding one node does not compromise optimality. □

Second Case: $K < M$

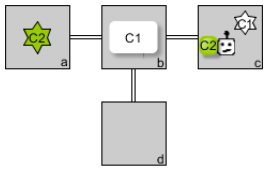


Figure 3: If the agent first moves C1, it cant reach its goal, because of a collision with container C2. First moving C2 is not possible, due to an immediate collision with C1. The agent has to move C1 to vertex d, then C2 to a and finally C1 to c.

If the problem is well-formed, the algorithm will still work, since all the containers will arrive at their goal position no matter the order. It can be demonstrated with a very simple example (Figure 3), that having more containers than agents along with a not well-formed problem instance can lead to unsolvable problem instances for CBS-FCA. Therefore CBS-FCA is only suited for cases, where the problem instance is well-formed, otherwise completeness and optimality can not be guaranteed. Other MAPF algorithms exclude problem instances with completely blocked containers from the domain description, since it would require planning to solve such instances. We introduce a new algorithm called CBP-FCA, which can still solve semi-well-formed problem instances. On account of this, it is recommended to design warehouses, so that problem instances are always well-formed.

3.1.6 Example of CBS-FCA

High-Level Algorithm

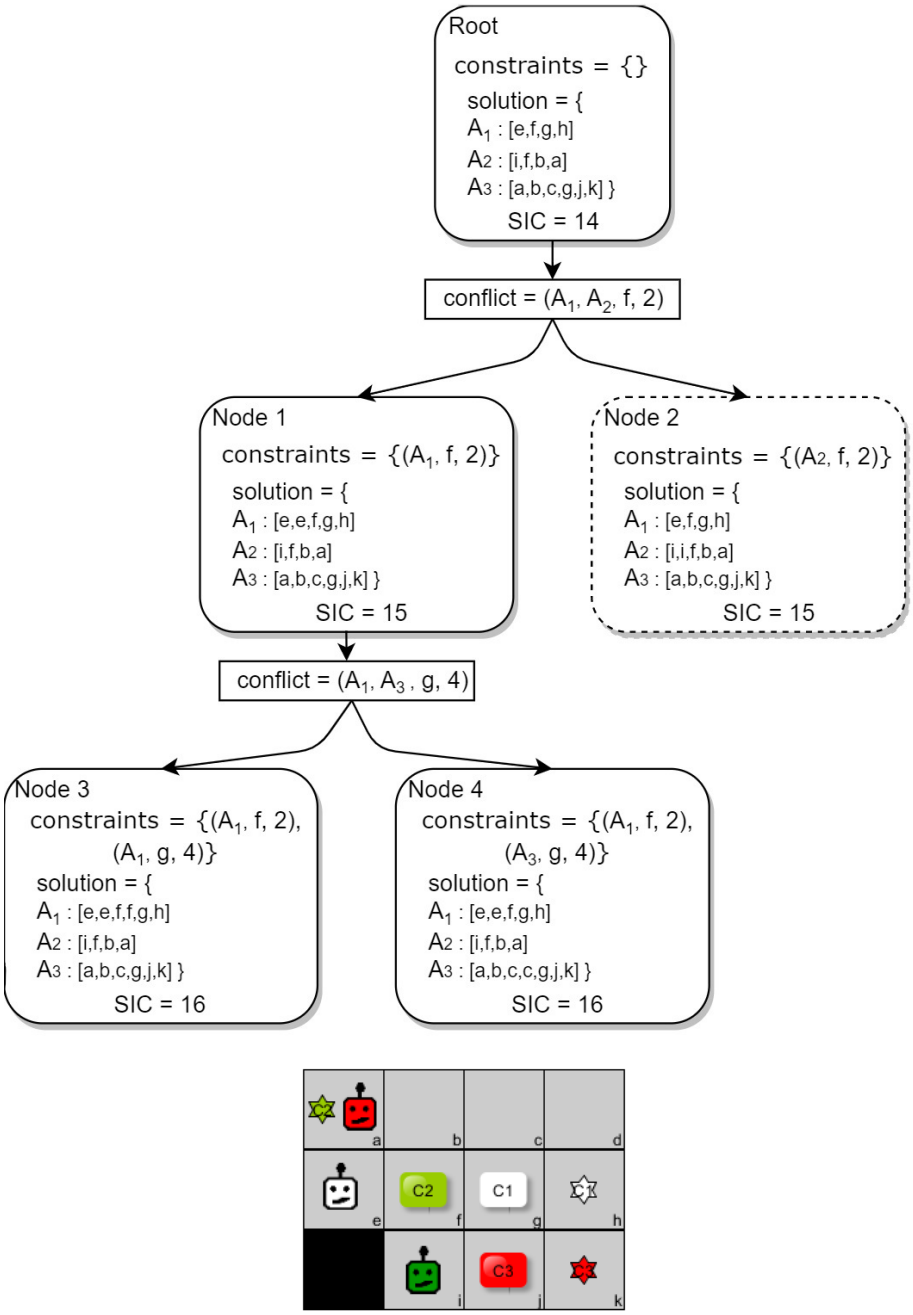


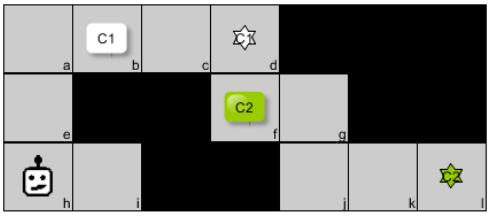
Figure 4

First, we formalize the given problem instance (Figure 4):

- the set of agents $A = \{\text{red, green, white}\}$
- the set of containers $C = \{C_1, C_2, C_3\}$
- agent start positions $s = \{red \to a, green \to i, white \to e\}$
- container start positions $s' = \{C_1 \to g, C_2 \to f, C_3 \to j\}$
- container target positions $t' = \{C_1 \to h, C_2 \to a, C_3 \to k\}$

$A_{white} = [C_1]$, $A_{green} = [C_2]$ and $A_{red} = [C_3]$ describe the assignments between containers and agents. For simplicity, we call the white agent A_1, the green agent A_2 and the red agent A_3 . First a root node is initialized with an empty constraints set and a solution, consisting of three single-agent paths, computed by the low-level algorithm. The SIC, sum of costs, is calculated by adding the lengths of the single-agent paths (lines 2 - 6 in Algorithm 1). Since the root node is the only node in the OPEN list, it is selected as the best node. Upon validating the solution of the root node, a conflict between the paths of the two agents A_1 and A_2 is found. As defined in the lines 13 - 17, two nodes are added to the OPEN list. Each of these two nodes prevents one agent to enter vertex f at the time step 2. After updating the solution and calculating the SIC for the two nodes, both are added to the OPEN list. Since both nodes have the same SIC and the same number of constraints, the algorithm picks node 1 at random. Consequent to validating the solution of node 1, a new conflict, between the agents A_1 and A_2, is found. As before, two new nodes, each having a new constraint for one of the agents involved in the conflict, are generated. Node 3 prevents A_1 to enter vertex f at time step 2 and vertex g at time step 4. Node 4 prohibits A_1 to enter vertex f at time step 2 and A_3 to occupy vertex g at time step 4. After updating the solutions based on the new constrains and calculating the SIC, both are added to the OPEN list. Now the OPEN list consists of three nodes, of which node 2 has the lowest SIC and gets selected as best node. Validating the solution does not return any conflict and therefore the solution is valid and the optimal solution to the problem instance.

Low-Level Algorithm



Iteration	time	start	goal	subpath	path
0	0	h	None	None	[]
1	3	h	b	[h,e,a,b]	[h,e,a,b]
2	5	b	d	[b,c,d]	[h,e,a,b,c,d]
3	6	d	f	[d,f]	[h,e,a,b,c,d,f]
4	10	f	l	[f,g,i,k,l]	[h,e,a,b,c,d,f,g,i,k,l]

Figure 5

Let the agents assignment be the list: [C_1, C_2], then the target list for the low-level algorithm equals [C_1.start, C_1.goal, C_2.start, C_1.goal]. First the algorithm initializes the sub_path and the path as an empty list of vertices, time to be 0 and new_start to be the start location of the agent. In the first iteration, the start vertex is the agents start location and the goal vertex is the first containers C_1 start position. After finding a path from the start to the goal vertex, we append this sub-path to the path and increment the time counter by the length of the sub-path minus 1. In the second iteration, we start from the previous goal location and try to find a path to next goal vertex. It is important to pass the new start time to A^*, to keep the low-level search coherent with the constraints enforced by the high-level algorithm. We increase the time again by the length of the new sub-path minus 1 and add the new vertices to the path. TA^* repeats this process until the last goal vertex is reached, in our example it needs four iterations to do so. After all the iterations, the low-level algorithm returns the path to the high-level algorithm to update the solution of the corresponding node.

3.2 Conflict Based Planning with Fixed Target Assignment CBP-FCA

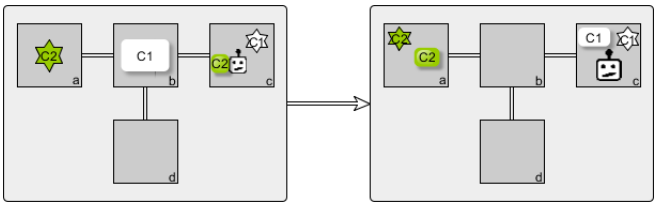


Figure 6: Agent: c → b → d → b → c → b → a → b → c → b → a
C1: b → b → d → d → d → d → d → d → b → c
C2: c → c → c → c → c → b → a

In Figure 3, the agent has to transport two containers to their goal location, but no matter the order of the containers in the assignment, the problem instance can not be solved by CBS-FCA. If the agent wants to transport the container C_2 to its target location, all possible paths are blocked by container C_1. Even first transporting C_1 is not possible, since its target location is being blocked by C_2. To cope with the problem of blocking containers, we introduce a new algorithm called CBP-FCA, which can handle well- and semi-well-formed problem instances. Since the concept of path finding does not achieve to solve all possible problem instances, it has to be replaced by planning, which makes the problem more complex to solve. For CBP-FCA, we do not have to change the structure of the high-level algorithm, but some procedures might become more complex (see section 3.2.3). The low-level path finding algorithm will be replaced by a low-level planning algorithm of choice. We chose a single-agent A* planning variant for the low-level search. To understand the small changes in the high-level algorithm, it makes sense to first introduce the container assignments and the new low-level search algorithm.

3.2.1 Container Assignments

The main difference between CBP-FCA and CBS-FCA, is that the high-level container assignment differs from the one in the low-level algorithm of which the high-level one is a subset.
As in CBS-FCA, the **high-level assignment** is a list of containers, but the order is not of any importance. The assignment specifies the containers, which can exclusively be moved by the assigned agent. Containers, that do not need to change location

(target location equals start location), are not assigned to any agent.

The low-level assignment specifies, which containers need to be considered during planning. Since the agent can move any container around, if it is in the agents high-level assignment or unassigned, the low-level algorithm needs to consider all of these containers during planning. Therefore the high-level assignment specifies only a subset of the low-level assignment.

3.2.2 The Low-Level Planning Algorithm

For the low-level planning algorithm 4, we chose a variant of A*, which expands search nodes only if they are consistent with the constraints. A* will find an optimal plan with respect to the constraints. We define the initial state to be the start locations of the agent and the containers from the low-level assignment. The goal state consists of any position for the agent and the goal locations for the containers. From the resulting plan, the algorithm extracts one path for each container and one path for the agent (Figure 6) These paths will then be returned to the high-level algorithm to be checked for conflicts.

Algorithm 4: Low-level Planning

Input:

- list of containers K
- an agent A
- set of constraints C

$initial\_state \leftarrow (agent.start, c_1.start, ..., c_k.start)$
$goal\_state \leftarrow (any\_agent\_position, c_1.goal, ..., c_k.goal)$
Apply A^*(C,A,K, initial_state, goal_state) to find a plan P
Extract a path for the containers and the agent from P
return paths

3.2.3 The High-Level Algorithm

The overall structure of the high-level algorithm of CBS-FCA can be used for the high-level algorithm of CBP-FCA. However, the two key parts, solution validation and node insertion to the OPEN list, require to be changed.

Path validation becomes more complex, since not only agent and assigned container

paths, but also unassigned container paths have to be compared. As in CBS-FCA, comparing agent paths needs to be done, in order to detect agent vertex and swapping conflicts. The new need to compare unassigned container paths individually results from the fact, that an agent can pickup and drop any containers at locations other than their target locations. Even comparing assigned containers becomes more complex, since container paths are not coherent sub-paths of the agent path. In CBS-FCA, comparing container paths could be efficiently incorporated in the comparing of agent path, without changing the run time by more than a constant factor. Such a container sub-path are all the nodes between the containers start and target location in the agents path. In CBP-FCA, the low-level search returns a path for each container, which we have to compare pairwise. If $|C_u|$ is the number of unassigned containers and $|C_a|$ the number of containers assigned to agent a, then comparing two agents with this procedure has a run time of

$$\mathbf{O}(|C_u|^2 + |C_1| * |C_u| + |C_2| * |C_u| + |C_1| * |C_2|)$$

Inserting Node to the OPEN list does not change for any conflict, but the container conflicts. The fact, that agents can drop containers anywhere, changes how the algorithm has to handle container conflicts. In CBS-FCA, we only have to add one node into the OPEN list, since the container, not being transported, can not be reached any faster by its agent. In CBP-FCA, we do not have this guarantee, since agents can pick up containers and drop them anywhere. Therefore, we need to add two nodes, with additional container vertex constrains, to the OPEN list and change the code highlighted blue in the Algorithm 1 to the lines in Algorithm 5.

Algorithm 5: Replacement Lines

```
else if C is a Container Conflict then
    //C is a Conflict of type (c_1, c_2, v, t)
    foreach container c_i in C do
        new_constraint = (c_i, v, t)
        Create_new_node(P, new_constraint)
```

3.2.4 Examples

low-level planning

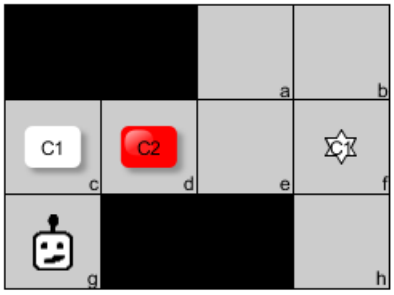


Figure 7

The agents task is to move container C_1 from vertex c to vertex f. There is no path, that is not blocked by container C_2, therefore the agent first needs to move C_2. Executing the low-level search algorithm will return the following plan, where the first vertex corresponds to the position of the agent, the second vertex to the location of C_1 and the last position to the location of C_2:

(agent.position, C_1.position, C_2.position):

$$(g,c,d) \rightarrow (c,c,d) \rightarrow (d,c,d) \rightarrow (e,c,e) \rightarrow (a,c,a) \rightarrow (e,c,a) \rightarrow (d,c,a)$$
$$\rightarrow (c,c,a) \rightarrow (d,d,a) \rightarrow (e,e,a) \rightarrow (f,f,a) \rightarrow (e,f,a) \rightarrow (a,f,a) \rightarrow (e,f,e) \rightarrow$$
$$(d,f,d)$$

After invoking A^*, the low-level algorithm extracts the paths for the agent and each of the containers from the previously computed plan. The three resulting paths of equal length would be the following:

```
agent: [g,c,d,e,a,e,d,c,d,e,f,e,a,e,d]
C1:    [c,c,c,c,c,c,c,c,d,e,f,f,f,f,f]
C2:    [d,d,d,e,a,a,a,a,a,a,a,a,a,e,d]
```

These paths are then passed to the high-level algorithm, where the paths of the agents / the containers are pairwise compared.

3.2.5 Low-Level Planning Heuristics

A first heuristic h_1 for the low-level planning is the sum of the direct distances between the containers start and goal positions, combined with the direct distance between the agent and its nearest assigned container:

$$h_1(state) = dir\_dist(agent\_position, nearest\_container) + \sum_{i=1}^{k} dir\_dist(c_i\_position, c_i\_goal)$$

The heuristic will consider the direct distance between the agent and its nearest container, which has not yet reached its goal. The problem with this approach is that the heuristic will increase dramatically whenever a container reaches its goal position, since the distance to a new nearest container will increase the heuristic value. This heuristic, however, might be a great heuristic if only one container is assigned to an agent.

To solve this problem, we can either take the average distance to all the containers or simply ignore the distance between the agent and the containers. The first attempt is not admissible, since it can overestimate the actual costs. Therefore, this heuristic can not be used in optimal path finding. Ignoring the distance between the agent and the containers never overestimates the actual cost, but does incorporate less information than h_1. Nevertheless, the second approach might still be a good heuristic depending on the problem instance. We formalize the second heuristic h_2 as follows:

$$h_2(state) = \sum_{i=1}^{k} dir\_dist(c_i\_position, c_i\_goal)$$

4 Experiments

4.1 Generating Benchmarks

To evaluate the algorithms, we used the already existing maps from Nathan Sturtevant's Moving AI Lab 2D grid world benchmarks set [14]. The maps also contain up to a 1000 single-agent tasks, which can be combined to create a wide variety of multi-agent path finding tasks. Although the maps can be used to evalutate MAPD algorithms, the predefined tasks can not be utilized. To solve this issue, we randomly assign map locations to the start/goal position of the agent/container, with a uniform distribution over the graph vertices, for up to n tasks. To validate if every possible combination of single-agent tasks to a MAPD task is still solvable, it is possible to simply validate every single-agent task. If a problem instance remains well-formed for a single-agent task, after blocking all locations defined in the other tasks, this single-agent tasks is solvable for any task combination. The following algorithm can be applied to test solvability of a task set:

Algorithm 6: Task Validation

```
1  Input: Graph G and task set T
2  foreach task t in T do
3  |  C ← copy of G
4  |  foreach task d in T \ {t} do
5  |  |  Cut d.agent.start from C
6  |  |  Cut d.container.start from C
7  |  |  Cut d.container.goal from C
8  |  |  #cutting these nodes from C enforces all conditions for well-formed
   |  |   definition
9  |  solution ← CBS-FCA(t, C)
10 |  if Solution is found then
11 |  |  continue
12 |  else
13 |  |  Reassign locations for t and revalidate tasks
14 return "Success: Every combination solvable"
```

Furthermore, we created an own set of maps, which are highly inspired by the classical warehouse layout. The main difference to the maps in the Sturtevant's Moving AI Lab benchmark set, is that there are no solid walls within the maps, but only containers. Since agents can move under containers, path finding for agents, that do not transport containers, becomes a very simple task, because of the absence of obstacles. Once the agent reaches its container, the set of possible movements becomes much more restricted. The evaluation and images of the maps can be found in section 4.2.

4.2 CBS-FCA Empirical Evaluation

For each map, I created 10 set of tasks with up to 29 agents. The tables presented in this section show the results averaged over these 10 different task sets. We implemented both algorithms in python and set a time limit of 5 minutes per task.

4.2.1 Random32 vs Random64

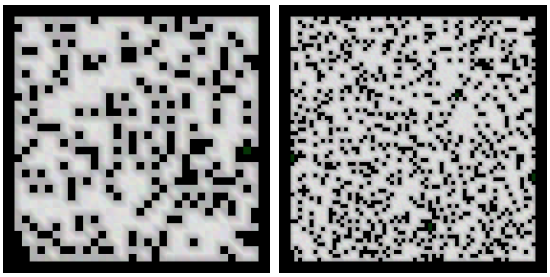


Figure 8: Random32 (left) and Random64 (right) by [14]

Random32 is of size 32x32 and Random64 has the size 64x64. Both maps incorporate obstacles, which are spread out randomly over the map. The density of obstacles in both maps is about the same. In Figure 9 and 10, the average makespan is nearly constant for different amounts of agents and the SIC increases linearly. This indicates, that all the individual single-agent tasks are very similar in terms of complexity. Since no single-agent path is much harder than others, the number of agents involved in a task becomes a more precise metric. There are no single-agent tasks responsible for a drop in the performance of the algorithm.

On **Random32** (Figure 9), CBS-FCA could solve 100% of the problem instances for up to 10 agents. For problem instances, consisting of more than 10 agents, the performance dropped fast. At 21 agents, CBS-FCA did not manage to solve a single

problem instance. If we compare the top right bar chart to the top left line plot in Figure 9, we can observe, that as soon as the performance starts to drop, the number of high level nodes starts to increase significantly. This indicates, that a lot of single-agent paths collide, which is a result of the small map size. We can explain this observation in terms of task density, which indicates the number of tasks per map size. As we increase the number of agents the task density increases, which influences the probability for collisions to occur.

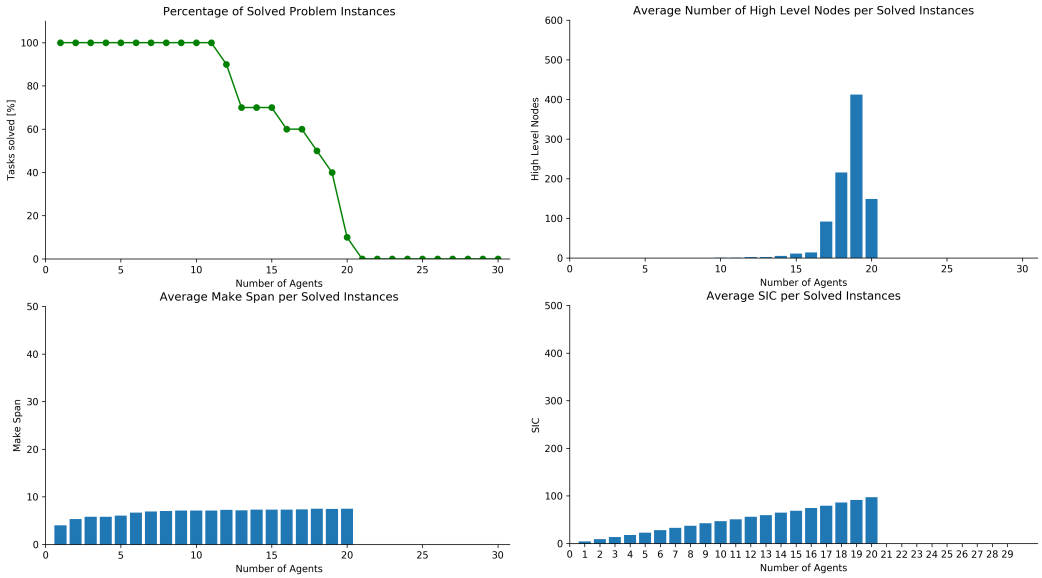


Figure 9: CBS-FCA on Random32

On **Random64** (Figure 10), CBS-FCA manages to solve about 80% of all tasks for up to 20 agents. For tasks with more than 21 agents, the performance begins to drop with every agent added. As for Random32, we can observe an increase in the number of high level nodes as the performance drops. For 29 agents, CBS-FCA still manages to solve 20% of all problem instances.

The fact, that CBS-FCA solves more tasks with a larger amounts of agents on Random64, can be led back to the task density on both maps. The same amount of single-agent tasks on a smaller map implies a higher task density than on a larger

map. On account of this, we can hold the task density to account as limiting factor of CBS-FCA.

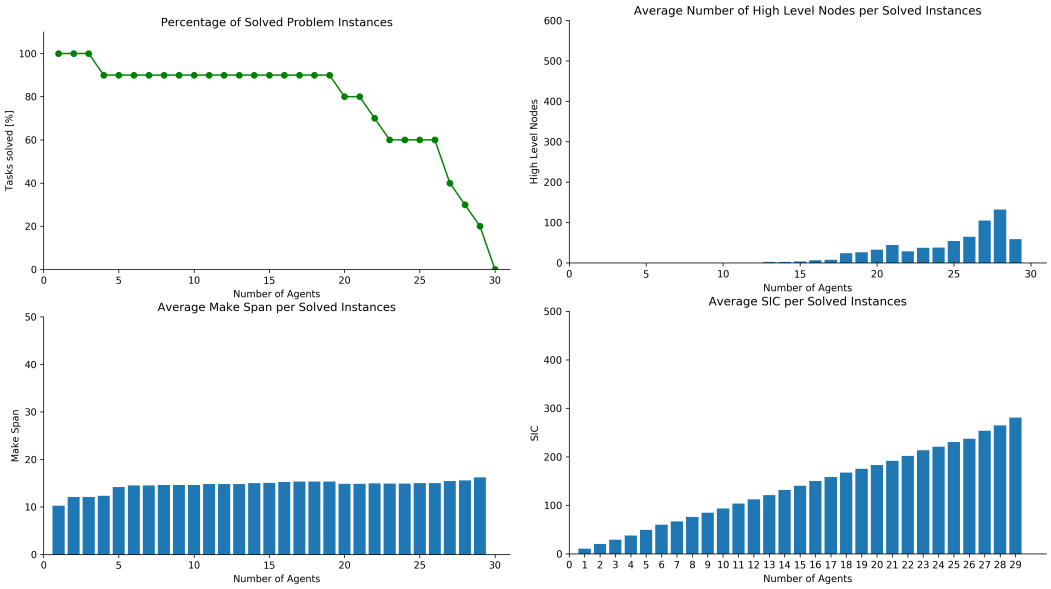


Figure 10: CBS-FCA on Random64

4.2.2 Mansion Map

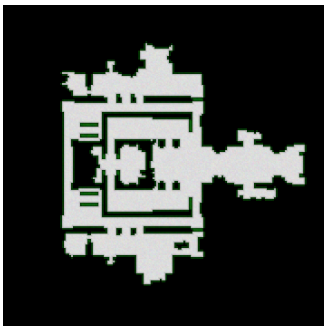


Figure 11: Mansion map by [14]

The Mansion map (Figure 11)is a 141 by 162 grid and consists of a 2D representation of a mansion. It has multiple rooms on the side and in the middle of the mansion layout. These rooms are concave, which sometimes causes difficulties for the A^* algorithm. As before we can observe, that the SIC increases linearly, which hints

again to similar single-agent tasks in terms of complexity. The makespan increases from 17 for 1 agent to 34 for 17 agents, which might be the result of collisions and therefore longer paths.

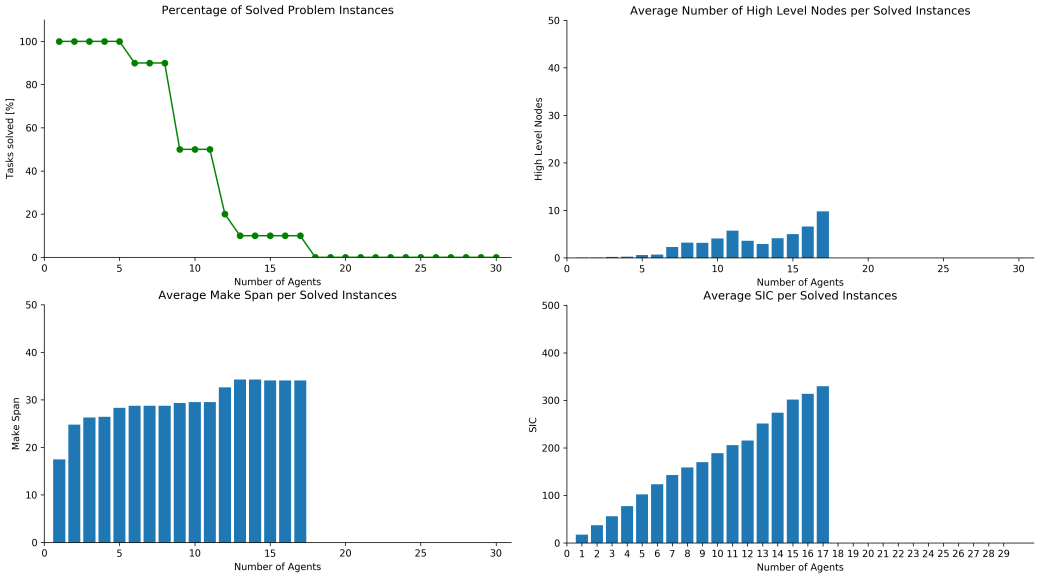


Figure 12: CBS-FCA on Small

In the top left table in Figure 12, we can observe, that CBS-FCA sees major performance drops for tasks with more than 5 agents and can only solve 10% of the tasks with 17 agents. Although CBS-FCA suffers from performance drops, we can not identify a drastic increase of high level nodes. This might be the result of the concave rooms, which can often lead to dead ends in the low level A^* search. This might be improved by using another heuristic in the low level algorithm.

4.2.3 Maze

To further investigate the effect of concave obstacles in maps and their influence on the performance of the CBS-FCA algorithm, we conducted experiments on the Maze 128 map (Figure 13) from Nathan Sturtevant's Moving AI Lab benchmark set. Mazes are map layouts, that contain a lot of concave obstacles and therefore a lot of dead ends. As mazes follow such layouts, path finding becomes a much more difficult task depending on the heuristic used in search algorithms. All the experiments have been conducted using the direct distance as an admissible heuristic.

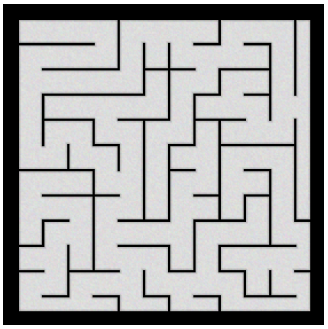


Figure 13: Maze 128x128 by [14]

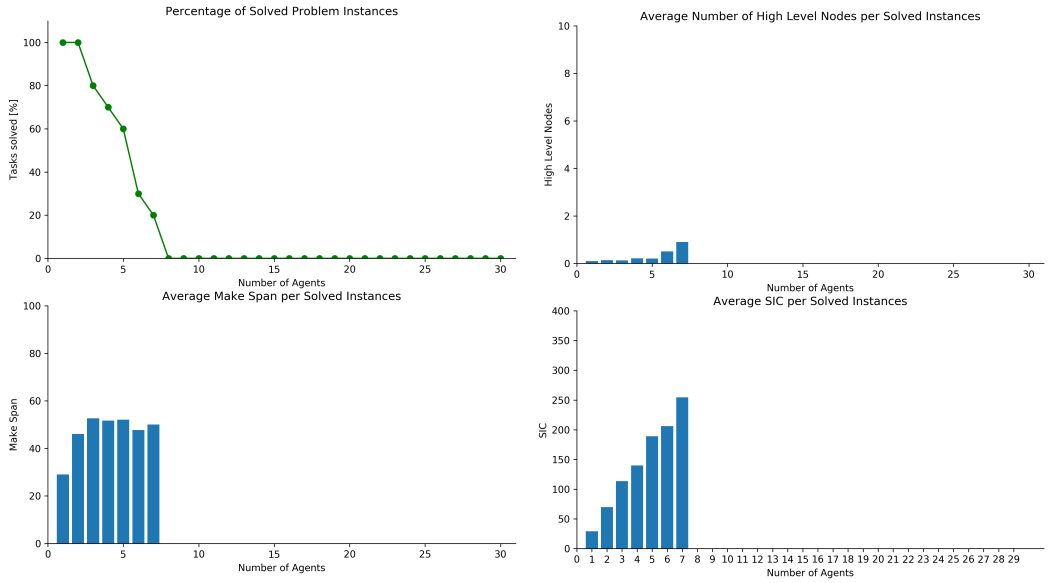


Figure 14: CBS-FCA on Maze128

As previously mentioned, with the help of the makespan and the SIC, we can conclude, that the individual single-agent tasks have a similar level of complexity. Again, we can observe a huge performance drop without any significant increase in the amount of high level nodes. Already at tasks with 8 agents, CBS-FCA is not able to compute a solution within the time limit of 5 minutes. This lets us infer that the performance drop of the CBS-FCA is induced by a poor performance of the low level search. If we increase the amount of concave obstacles in the map, we increase the probability of a poor performance of the low level algorithm, since the search might end up more

often in a dead end. The efficiency of the low level algorithm might be improved by using another heuristic (see chapter 5).

4.2.4 Warehouse Layouts

In the following, we will investigate experiments on warehouse environments of different sizes. In Figure 15, the black boxes represent the containers stored in the warehouse. We constructed the experiments on the following maps of three different sizes:

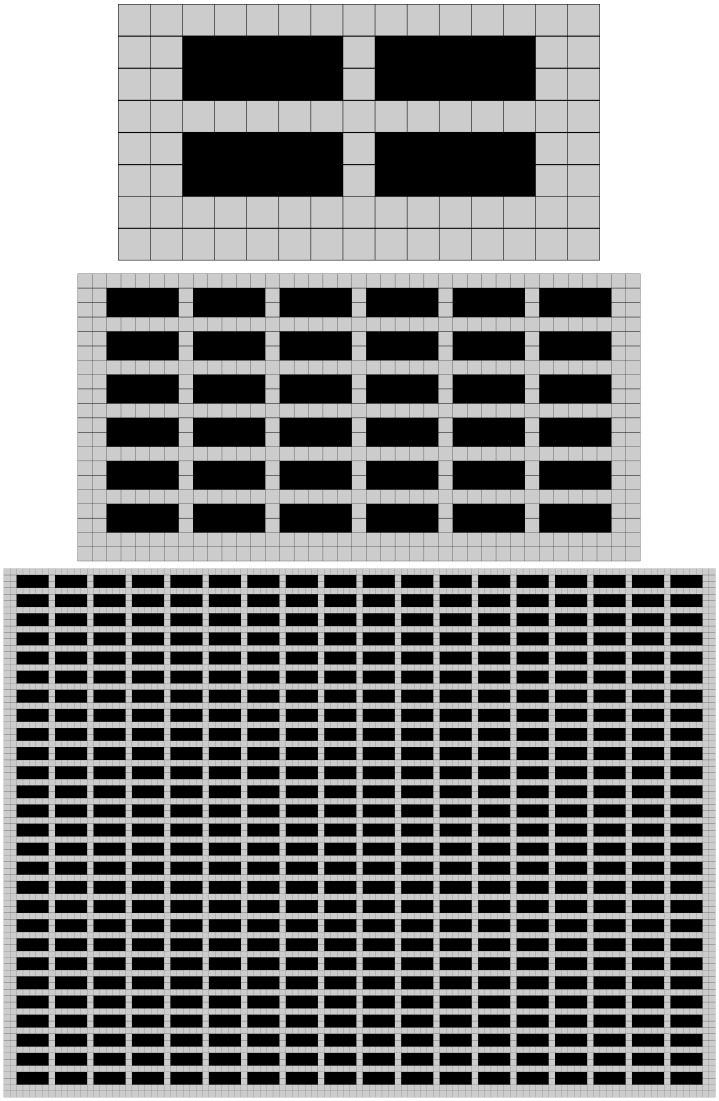


Figure 15: G15x8 (top), G39x20 (middle) and G111x95 (bottom)

In contradiction to the previously seen maps, warehouse environments usually do not have concave obstacles. This enables the low level search to expand search nodes without running the risk to end up in a dead end. Warehouses permit agents to have a very unconstrained set of movements, as long as they do not transport a container. Once an agent picks up a container, they are not allowed to move to a vertex occupied by another container, which reduces the set of allowed movements drastically. With a container, agents are only allowed to move on the gray vertices, shown in Figure 15.

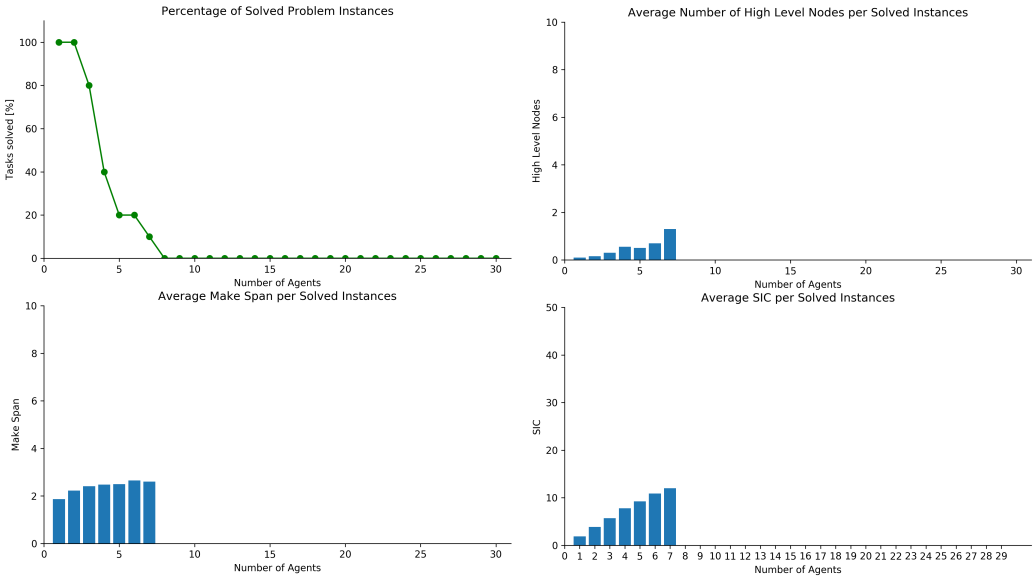


Figure 16: CBS-FCA on G15x8

On **G15x8** (Figure 16), CBS-FCA immediately starts to find itself struggling with tasks, containing more than two agents. It seems like the number of solved tasks decreases exponentially as the number of agents gets bigger. Since the individual single-agent tasks have the same complexity, indicated by the linear increase of the SIC, we think that the performance decrease is due to the task density. As on random32, we can achieve a high task density with a small amount of agents, because the maps are comparably small. On **G39x20** (Figure 17), CBS-FCA is able to solve tasks with up to 16 agents, but still suffers from a strong performance decrease. On **G111x95** (Figure 18), even more tasks with higher agent counts can be solved. This is a clear indication that CBS-FCA's performance directly correlates with the task density, if the tasks are distributed uniformly over the maps. This can obviously change, if the problem instance is designed in a way knowing that no single-agent paths will collide. Comparing the top left to the top right table in Figure 18, we

can see, that performance starts to decrease when the number of high level nodes increases. This indicates again, that on warehouse layouts, the performance is not limited by the low level search, but rather by the high amount of collisions.

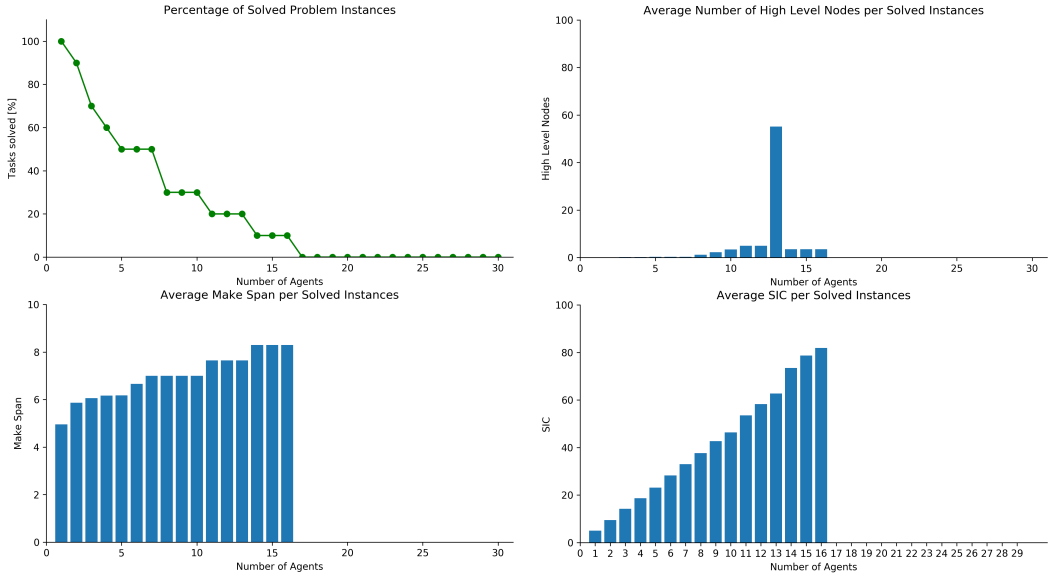


Figure 17: CBS-FCA on G39x20

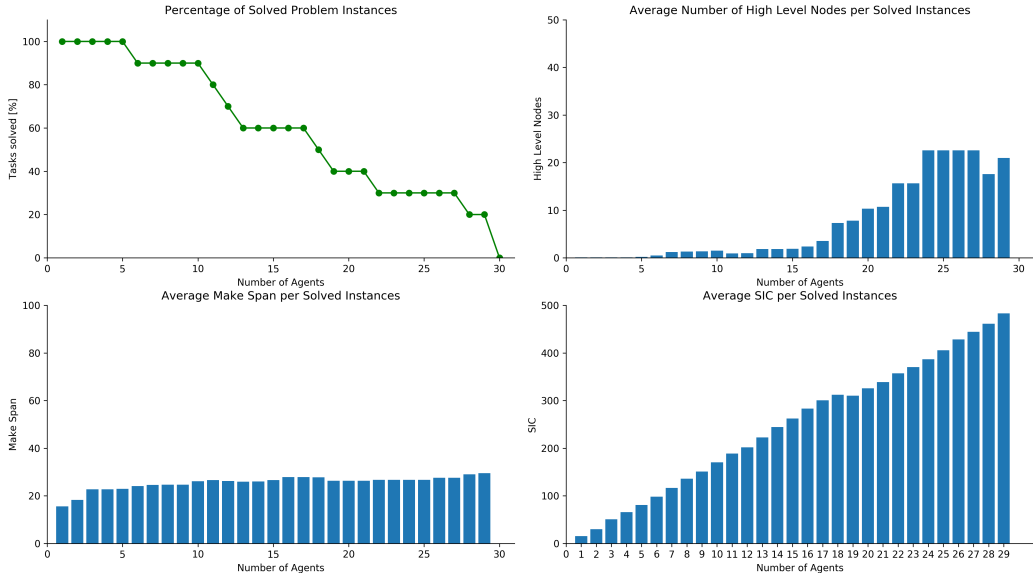


Figure 18: CBS-FCA on G111x95

4.3 CBP-FCA Evaluation

We decided to compare the performance of CBP-FCA on the same maps, that we used for the evaluation of the CBS-FCA algorithm. However, we will only show the results on the smaller maps, because CBP-FCA did not manage to solve any problem instance for bigger maps.

4.3.1 Random32 vs Random64

As before in the evaluation of CBS-FCA, we compare the performance of CBP-FCA on both maps, Random32 and Random64. As we can observe in Figure 19, CBP-FCA can solve problem instances up to 6 agents on Random32. This changes when CBP-FCA tries to solves tasks based on the Random64 map (Figure 20). Here it does not manage to solve tasks with more than one agent. The combination of bad performance and the low number of high level nodes is a strong indication, that the performance of CBP-FCA is limited by the performance of the low level search algorithm.

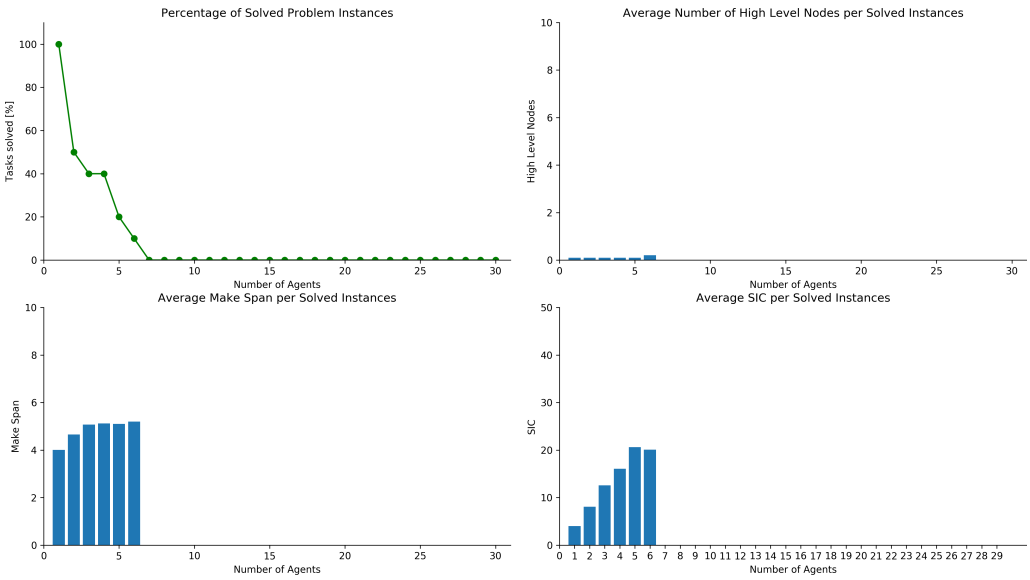


Figure 19: CBP-FCA on Random32

Comparing the performance of CBS-FCA to CBP-FCA on Random32, it is clear that using planning instead on path finding makes the low level search much more complex. CBS-FCA performs way better on the same tasks than CBP-FCA. This indicates, that exchanging the low level path finding algorithm for a low level planning

algorithm should not be done for well-formed problem instances.

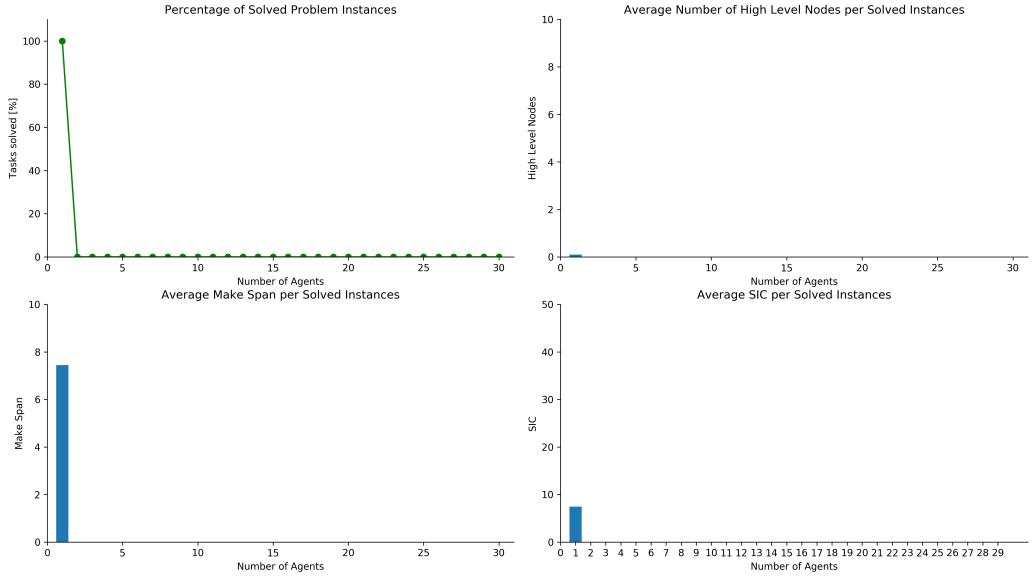


Figure 20: CBP-FCA on Random64

4.4 Evaluation Summary

Domains	N° Tasks	CBS-FCA	CBP-FCA
Random32	290	162	26
Random64	290	230	10
Mansion	290	99	/
Maze	290	46	/
G15x8	290	37	/
G39x20	290	65	/
G111x95	290	179	/

Figure 21: Total Number of MAPD tasks solved per domain

In Figure 21, we can see how many problem instances have been solved in each domain. For **CBS-FCA**, Random64 has been an easier domain than Random32 with 68 more tasks solved. By switching to domains that contain concave obstacles like the Mansion and the Maze domain, we can see that less problem instances got solved even thought these domains do have a low task density. On the warehouse domains G15x8, G39x20 and G111x95, we observe that more tasks get solved as the map sizes increase. This is the result of having a low task density for the same amount of tasks, but with bigger map sizes. Task density is influencing the probability of collision between the single-agent paths. For **CBP-FCA**, we could observe, that switching from the Random32 to the bigger Random64 domain causes performance drops for the CBP-FCA algorithm. This shows, that CBP-FCA is not only limited by the task density and map design, but also by the poor performance of the low level planning algorithm.

5 Future Work

We propose a few improvements to algorithms introduced in this paper, which could be added in a future work. First we propose a solution to the bad performance of the low level algorithm in CBS-FCA on maps with a lot of concave obstacles. The use of another heuristic may result in a greater efficiency of the algorithm. Using the direct distance between two points might be very simple and efficient to implement using the Pythagorean theorem, but completely ignores the appearance of concave obstacles. As alternative, we could calculate a perfect heuristic using a breadth-first search like Dijkstra's algorithm. We would need to calculate the heuristics once for each low level start position. Calculating the heuristic once would have a linear run time in the number of vertices in the graph $O(|V|)$. We would need to calculate the heuristic for each agents start position, then for each containers start and target locations. Except for containers, that are last in an assignment, we only need to calculate the heuristic for the start location, since we do not start a new low level search starting form their target locations. For a task with k agents and m containers, this would have a total run time of $O((k + 2m - k) * |V|)$. The calculation of the heuristic would be much less efficient, but the low level search would find the optimal path with a run time linear in the paths length.

For the second improvement, we want to focus on warehouse layouts. Highways, which have been introduced in [5], serve as a guidance for the low level path finding algorithm and thereby try to reduce the number of collisions. As we can see in figure 15, warehouses do have a lot of narrow corridors, which can only be used by agents moving in the same direction or will lead to collisions otherwise. Highways try to get all the agents, that are in the same corridor, to move in the same direction. Every second horizontal corridors could have a highway pointing to the right and all the others to the left. Analog for the vertical corridors, which could have highways pointing alternatively up and down, see figure 22 for a graphical representation. Highways can be efficiently implemented by changing the cost function for specific actions on certain nodes. This in combination with bounded sub-optimal CBS variants can lead

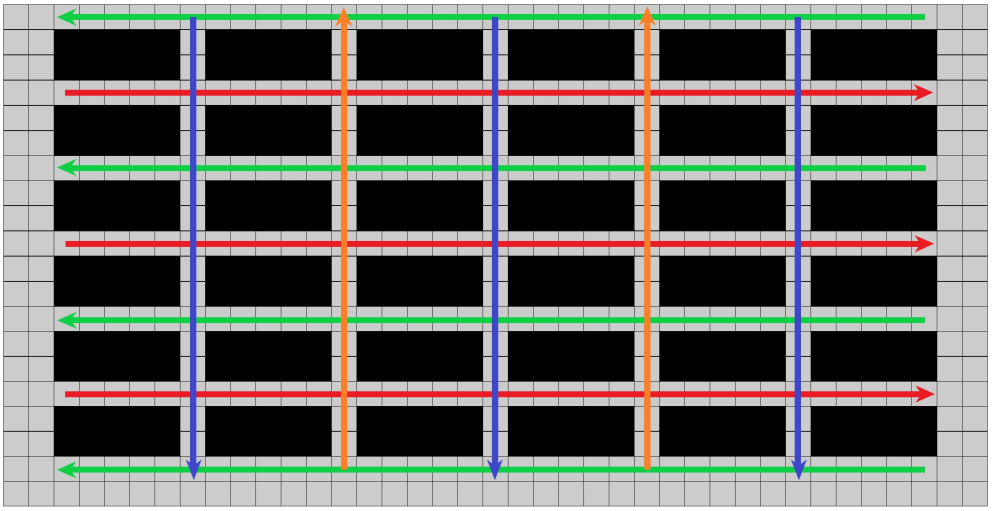


Figure 22: G39x20 with highways

to a considerable speed up.

At last, we thought about assigning the containers optimally to get overall shorter solutions to the problem instances. This idea has been introduced in [6], and requires changes to the high level algorithm. Rather than using a search tree in the high level algorithm, [6] uses a search forest. Each tree in the forest represents one target assignment. Each time a root node gets expanded in the high level search, the next best assignment is computed and added as new root node to the open list. Like in the classical version, once a node is validated and no conflict is found, the algorithm computed the optimal solution and found the optimal target assignment. This algorithm could be extended to fit the MAPD domain by assigning the containers instead of target locations. This can easily be done by applying the Hungarian method on the the agents and containers start positions. If we have more containers than agents, we could create an assignment by first assigning a single container to an agent and then using the Hungarian method to assign the rest of the containers based on the assigned containers target and the unassigned containers start locations. This process can be repeated until no container is left unassigned.

6 Conclusion

In a nutshell, we have empirically shown that using path finding approaches instead of planning searches for the low level algorithm of CBS results in a much better performance. The performance of CBS-FCA and CBP-FCA is limited by multiple factors depending on the map layout and the task to solve. As show in the evaluation, CBP-FCA suffers from poor performance of the low level planning algorithm, therefore it should only be used on semi-well formed problem instances. In some domains, CBS-FCA could solve a large a amount of problem instances like on Random64 and in other domains like the Maze, the performance dropped drastically. Given the results of the evaluations, we recommend to design warehouse layouts in a way, so that there are no concave obstacles and keep the problem instances always well-formed.

7 Acknowledgments

First and foremost, I would like to thank...

- Advisers : Tim Schulte
- Examiner : Prof. Dr. Bernhard Nebel
- Proofreaders: Charel Reinhard and Miguel Da Silva
- My family: Juliette Boussong, Patrick Bausch and Kelly Bausch for the great support

Bibliography

[1] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Eli Boyarski, Roman Bartak . Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks

[2] G. Sharon, R. Stern, A. Felner and N. Sturtevant. Conflict-Based Search for Optimal Multi-Agent Pathfinding. Artificial Intelligence, 219, 40-66, 2012.

[3] M. Barer, G. Sharon, R. Stern and A. Felner. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In Proceedings of the European Conference on Artificial Intelligence (ECAI), pages 961-962, 2014.

[4] A. Felner, J. Li, E. Boyarski, H. Ma, L. Cohen, S. Kumar and S. Koenig. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), 83-87, 2018.

[5] L. Cohen, T. Uras and S. Koenig. Feasibility Study: Using Highways for Bounded-Suboptimal Multi-Agent Path Finding. In Proceedings of the Symposium on Combinatorial Search (SoCS), 2-8, 2015.

[6] W. Hoenig, S. Kiesel, A. Tinka, J. Durham and N. Ayanian. Conflict-Based Search with Optimal Task Assignment. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 757-765, 2018

[7] H. Ma, J. Li, S. Kumar and S. Koenig. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), 837-845, 2017

[8] P. Surynek, A. Felner, R. Stern and E. Boyarski. Efficient SAT Approach to

Multi-Agent Path Finding Under the Sum of Costs Objective. In Proceedings of the European Conference on Artificial Intelligence (ECAI), 810-818, 2016

[9] G. Wagner and H. Choset. M*: A Complete Multirobot Path Planning Algorithm with Performance Bounds. In Proceedings of the International Conference on Intelligent Robots and Systems (IROS), 3260-3267, 2011

[10] M. Barer, G. Sharon, R. Stern and A. Felner. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. In Proceedings of the European Conference on Artificial Intelligence (ECAI), 961-962, 2014

[11] M. Goldenberg, A. Felner, R. Stern, G. Sharon and J. Schaeffer. A* Variants for Optimal Multi-Agent Pathfinding. In AAAI-12 Workshop on Multi-Agent Pathfinding, 2012

[12] Trevor Standley. Finding Optimal Solutions to Cooperative Pathfinding Problems, 2010

[13] P. Surynek. Towards Optimal Cooperative Path Planning in Hard Setups through Satisfiability Solving, 2012

[14] Nathan Sturtevant's Moving AI Lab. 2D grid world benchmarks