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

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 game and efficient path planning for robots in warehouses to autonomously routing cars trough 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 move to an other location or stay at their position. 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 there are no conflicts between the agents, for example no two agents want 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 Solution have 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 a solution becomes even harder, that is why over the last few years various algorithms have been proposed. 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, representing all the agents).

Sat-Solvers have first been introduced by Surynek in [12] back in 2012. The main idea is to start from a solution, obtained by other method, like A*, divide it into small subsequences 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 [10], 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 [11], called independence ID, which groups conflicting agent into group for which a non conflicting 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 the approaches chapter. In the last few years, any 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 [5].

I 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 1). The difference to the new formalism is that containers are described as passive agents and therefor 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 to be moved that do not have a target other then their start position and therefor require planning to be solved.

write about how many nodes are open in my algorithms, for planning astar added either 5 or 10 by expanding one node

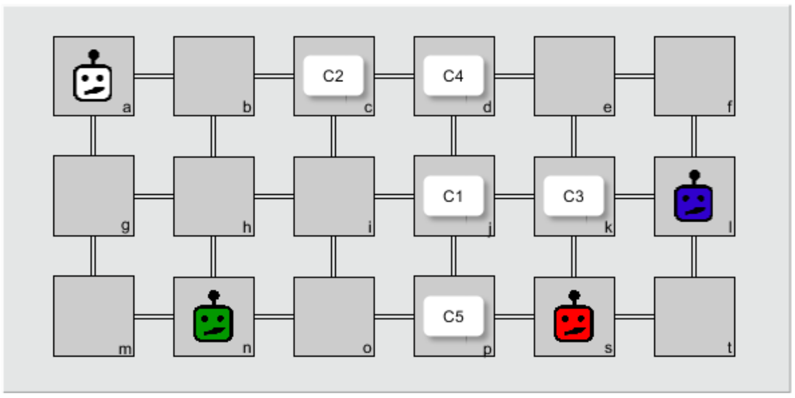


Figure 1: A possible problem instance:

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

2 Background

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 $v_* \in V$

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 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:
  $wait(a, v_i) =< (at(a, v_i)), (at(a, v_i)) >$

I also have to define a cost-function c, that assigns a cost to each action. Typically, a unit-cost-function, that assigns each action the same cost of 1, is used.
As in [1], I define a sequence of actions of one agent a_i as followed $\pi(a_i) = (a_1, a_2, ..., a_n)$ and let $\pi_i[x]$ be the vertex v after executing the first x actions of the sequence. A single-agent-plan is a sequence of actions $\pi(a_i)$, so that $\pi_i[0] = s(a_i)$ and $\pi_i[|\pi(a_i)|] = t(a_i)$.
A solution to the MAPF problem is a set of single-agent-plans for all the k agents, so that no pair of single-agent-plans results in a conflict.
Many different type of conflicts can be defined, but the most spread are the following:

- **vertex conflict**: two agents occupy the same vertex an the same time,
  $\pi_i[x] = \pi_j[x]$

- **edge conflict**: two agents move from the same vertex v to the same vertex v', $\pi_i[x] = \pi_j[x]$ and $\pi_i[x+1] = \pi_j[x+1]$ (implies a vertex conflict before and after exec of action x)

- **Following conflict**: an agent move to a vertex, just left by an other agent,
  $\pi_i[x+1] = \pi_j[x]$

- **Cycle Conflict**: every agent moves to an agent previously occupied by an other 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 agent swap vertices over the same edge,
  $\pi_i[x+1] = \pi_j[x]$ and $\pi_i[x] = \pi_j[x+1]$

2.2 MAPD with blocking containers

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

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

Where:

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

Since I introduce, a new type of agent, which change the physics of the environment, I introduce new set of possible actions, along 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, I 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 container from vertex v_i to adjacent vertex v_j
- $wait_a(a, v_i)$ agents stay in its current vertex

- $wait_c(c, v_i)$ container stays at current position

The preconditions of the actions are the following:

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

The effects of the actions are :

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

An agent $a \in A$ can execute the actions from the action set $Action_a = \{move_a, move_c, wait_a\}$, an agent $c \in C$ can execute the actions for the action set $Action_c = \{move_a, move_c, wait_c\}$. I assign a uni-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, I 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 agent affected by the action and its position after agent i executed the action at time step t in its plan. (see figure **??** for an example)
In MAPD-BC, I will only be interested in the following conflicts:

- **agent vertex conflict**: two agents occupy the same vertex an the same time, if $\exists a_1, a_2 \in A$ and $c, c' \in C$: $(\pi_{a_1}[t] = \pi_{a_2}[t])$ or $(\gamma_c[t] = (a_1, v_i) \wedge \gamma_{c'}[t] = (a_2, v_i))$
- **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 $c_1, c_2, c_3, c_4 \in C$: $(\pi_{a_1}[t] = \pi_{a_2}[t-1] \wedge \pi_{a_1}[t-1] = \pi_{a_2}[t])$ or $(\gamma_{c_1}[t+1] = (a_1, v_i) \wedge \gamma_{c_2}[t] = (a_2, v_i) \wedge \gamma_{c_3}[t] = (a_1, v_j) \wedge \gamma_{c_4}[t+1] = (a_2, v_j))$

I introduce $p : \{A, C\} \rightarrow \{active, passive\}$ a bijective function, that determines from which agent type's perspective the planning is performed. At most one set can be active. An agent is active if $e \in E$ and E is active. It is important to note, that agent conflicts may only occur if the set of containers is the active planning set, since for the formal definition of the conflict, containers are the agents executing the actions. As in classical MAPF, I 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 agents i in the active planning set. In the MAPD setting, this corresponds to 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 agents i in the active planning set. In the MAPD setting, the sum of costs represents the sum of all the container path lengths, no matter the active planning set. clear enough?

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 position.

For the agents 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 to occupying any position if all of its assigned containers have reached their goal position. 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.
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 an other container.

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

c) Each agent can stay in its start location, without blocking an other agent's 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 an other containers goal location.

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

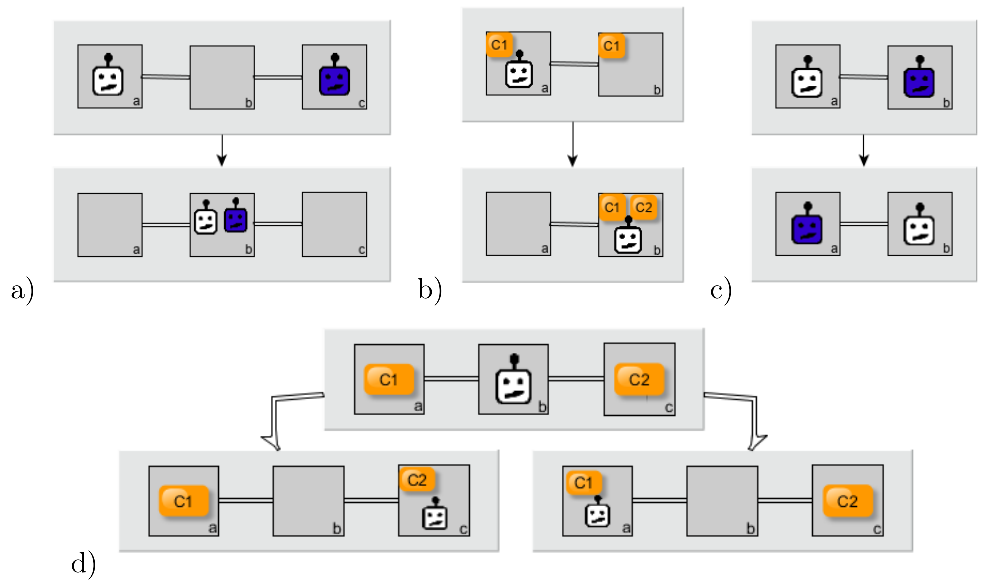


Figure 2:
a) Agent vertex conflict, b) container conflict, c) swapping conflict, d) action conflict

add classical MAPF conflict examples ?

3 Approach

Since multi-agent path finding algorithms are often much more efficient than multi-agent planning algorithms, I 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 optimally. I 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 perform 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 exits, but will not be further discussed in my thesis.

3.1 Conflict Based Search with Fixed Container Assignment CBS-FCA

add vocabulary explanation. As a first attempt, I 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 introduced. Due to time restrictions, I focus on the classical variant of CBS, but I will mention further improvements to the proposed algorithms.

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 high lighted 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 constraints for the low level search. If no conflict between the single agent 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

Algorithm 1: CBS-CA High Level

```
1  **Input:** MAPF instance and fixed container assignment for each agent
2  OPEN ← an empty list
3  $Root.constraints = \emptyset$
4  $Root.solution$ = find individual paths the low level() for each agent by
5  $Root.cost$ = SIC($Root.solution$)
6  insert $Root$ into OPEN
7  while $OPEN$ not empty do
8      $P$ ← best node from OPEN
9      Validate the paths in $P$ until a conflict occurs.
10     if $P$ has no conflict then
11         return $P.solution$ //$P$ is goal
12     C ← First Conflict found.
13     if (C is a Agent Vertex Conflict) then
14         //C is a Conflict of type ($a_i$, $a_j$, v, t)
15         foreach agent a in C do
16             $new\_constraint = (a, v, t)$
17             Create_new_node(P, new_constraint, a)
18     else if (C is a Swapping Conflict) then
19         //C is a Conflict of type ($a_i$, $a_j$, $v_a$, $v_b$, t)
20         foreach agent a in C do
21             $new\_constraint = (a, v_l, v_k, t)$
22             //$v_l$ is the position of $a$ before time step t
23             //$v_k$ is the position of $a$ after time step t
24             Create_new_node(P, new_constraint, a)
25     else if C is a Container Conflict then
26         //C is a Conflict of type ($c_i$, v, t)
27         $new\_constraint = (c_i, v, t)$
28         Create_new_node(P, new_constraint, a)
29
```

generate new nodes depending on the conflict found, see lines 12-28. Each new node generated copies the constraints as its parent node with one additional constraint to solve the conflict found upon validating the solution of the parent node make this sentence shorter. We update the path of the agent affected by the constraint as describes in Algorithm 2. In section 3.1.5, you can find an example for the CBS-CA algorithm.

Algorithm 2: Create new node

```
1 Input: P the parent node, C a new constraint and an agent a
2 A ← 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
```

3.1.1 Fixed Container Assignment and Planning Perspective

Let the problem instance have K agents and M containers.

In CBS-CA, 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, $f : M \to K$. 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 containers 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-CA focuses on planning from the agents perspective. If the number of agents equals the number of containers, it does not matter from which perspective the path finding is done, since the assignment function is bijective and the agents actions can be directly transformed into container actions of its container. reformulate this sentence

3.1.2 Adding New Nodes to the OPEN List

The high level algorithm of CBS-CA validates a node's 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-CA 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-CA 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 two the previous two conflicts, CBS-CA 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 conflict are 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, if will be classified as an agent vertex conflict. The constraint (a_i, c_i, v, t) will be added to the constraints for the new node and will deny the agent to move to vertex v with its container at time step t.

Agent Conflict:
In the container assignment for the CBS-CA algorithm exactly one agent is assigned

to each container, therefore it is impossible for agent conflicts to occur. There can not be two containers, that plan to move the same agent at the same time.

3.1.3 The Low Level Algorithm

Algorithm 3: TA* Low level

```
1  Input:
       • ordered list of of goal positions L
       • start position s
       • a graph G
       • set of constraints C
2  $time \leftarrow 0$
3  $new\_start \leftarrow s$
4  $path \leftarrow$ empty List
5  foreach goal in L do
6      $sub\_path \leftarrow$ find path with $A^*$(G,$new\_start$, goal, time)
7      $time \leftarrow time + length(sub\_path) - 1$
8      $new\_start \leftarrow goal$
9      Append $sub\_path$ to path
10 return path
```

For the low level Algorithm , I introduce a 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 2 locations using A^*. The optimality of the sub paths implies the for the path to be optimal as well. Since A^* has to respect the time step, in which the agent transitions to a new node 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, I chose the direct distance, which is admissible and therefore guarantees optimality.

In the following, I 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.

3.1.4 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.
- The problem instance is well-formed.

First Case: $K \geq M$

Lemma 1: *For every agent, CBS will find an optimal path to its containers start positions*
Proof 1:
Since CBS is optimal and complete, CBS will always find a solution, if the problem instance is well-formed. 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 an other containers start location, does not change the classical MAPF problem. This implies, that CBS will find such a path for each agent. □

Lemma 2: *For every container, CBS 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 with A* 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 not adding those nodes.
Not every agent has reached its container: The only new conflict, that could

interfere with the optimality of CBS-CA, 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 therefor not find any faster path to be consistent with the constraint. Thus only adding one node does not compromise optimality. □

First Case: $\underline{K < M}$

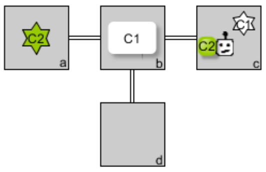


Figure 3: If the agent first moves C1, it cant reach its goal, because of a collision with container C2. First moving C2 is not possible, due to an immediate collision with C1. The agent has to move C1 to vertex d, then C2 to a and finally C1 to c.

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. If the problem is not well formed, it can easily be demonstrated (figure 3) for both cases, that the problem instances might not be solvable. I introduce a new algorithm called CBP-FCA, which can still solve semi-well-formed problem instances. It can be demonstrated with a very simple example (figure 4), 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. On account of this, it is recommended to design warehouses, so that problem instances are always well-formed.

3.1.5 Example of CBS-FCA

High Level Algorithm

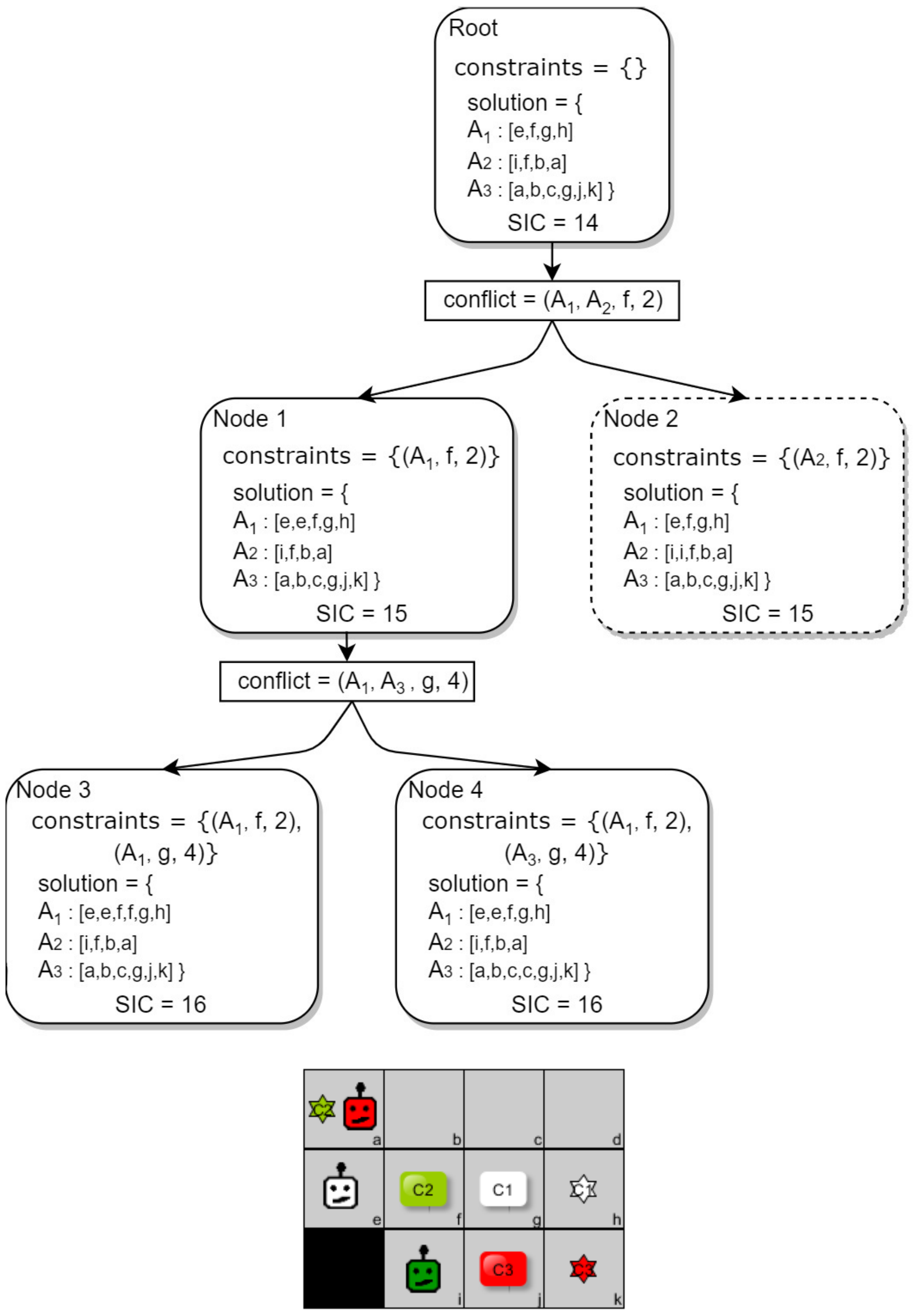


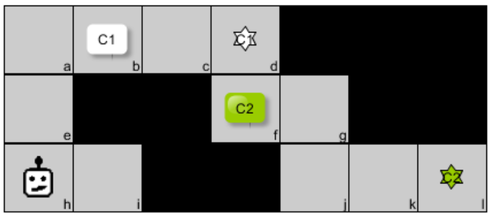
Figure 4

First, I 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, I call the white agent A_1, the green agent A_2 and the red agents A_3 . First a root node is initialized with an empty constraints set and a solution, consisting of 3 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). 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:	initial	1	2	3	4
time	0	3	5	6	10
start	h	h	b	d	f
goal	None	b	d	f	l
subpath	None	[h,e,a,b]	[b,c,d]	[d,f]	[f,g,i,k,l]
path	[]	[h,e,a,b]	[h,e,a,b,c,d]	[h,e,a,b,c,d,f]	[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 algorithms 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 timer 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

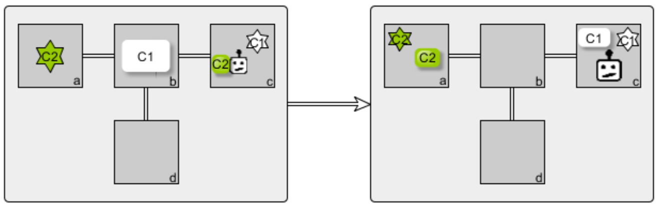


Figure 6: Agent: c → b → d → b → c → b → a → b → c → b → a
C1: b → b → d → d → d → d → d → d → b → c
C2: c → c → c → c → c → b → a

In figure 3, the agent has to transport 2 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, I 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 replace by a low level planning algorithm of choice. I chose 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 new low level search algorithm.

3.2.1 Assignments and low level complexity

3.2.2 The Low Level Planning Algorithm

For the low level planning algorithm 4, I 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. I define the initial state to be the locations of the agent and the containers. The goal state consists of any position for the agent and the goal locations for the containers. Form 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: TA* Low level

```
1 Input:
      • list of containers K
      • an agent A
      • set of constraints C
2 $initial\_state \leftarrow (agent.start, c_1.start, ..., c_k.start)$
3 $goal\_state \leftarrow (any\_agent\_position, c_1.goal, ..., c_k.goal)$
4 Apply $A^*$(C,A,K) to find a plan P
5 Extract a path for the containers and the agent from P
6 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 paths, but also 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 container paths individually results from the fact, that an agent can drop containers at locations other than their target locations, therefore 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 start and the target location in the agents path. Since 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 to agents with this procedure has a run time of

$$\mathbf{O}(|C_u| * (|C_u| - 1) + |C_1| * |C_u| + |C_2| * |C_u| + |C_1| * |C_2|)$$

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 reach 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 2 nodes to the OPEN list and change the code highlighted blue in the algorithm 1 to the lines in algorithm 5.

Algorithm 5: Replacement Lines

```
1 else if C is a Container Conflict then
2     //C is a Conflict of type ($c_1$, $c_2$, v, t)
3     foreach container $c_i$ in C do
4         $new\_constraint = (c_i, v, t)$
5         Create_new_node(P, new_constraint)
```

3.2.4 Low Level Planning Heuristic

A first heuristic for the low level planning is is the sum of the direct distances from the containers to their goal position, combined with the direct distance from the agent its nearest assigned container:

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

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 is that the heuristic will increase dramatically when the agent reaches the first goal for a container.

4 Experiments

4.1 Generating Benchmarks

To evaluate the algorithms, I used the already existing maps from Nathan Sturtevant's Moving AI Lab 2D grid world benchmarks set. 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, I 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 the 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 $T \setminus \{t\}$ 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"
```

4.2 CBS-FCA Empirical Evaluation

5 Conclusion

6 Acknowledgments

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

- advisers
- examiner
- person1 for the dataset
- person2 for the great suggestion
- proofreaders

7 References

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, 2015

[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] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

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

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

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

Bibliography