Multi-Agent Path Finding with Blocking Containers

Bachelor Thesis

Ben Bausch

July 23, 2019

1 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 the paper, I will formalize a warehouse problem with two different types of Agents, which can influence the behaviour of each other. This requires a new problem formalization MAPF and combines multiple challenges of classical MAPF. I will conduct experiments on the performance of current state-of-the-art algorithms in this new environment.

2 Introduction

3 Background

3.1 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 vertexes $V = \{v_1, v_2, ..., v_n\}$, representing the set of possible positions
  - E is a set of edges $E = \{e_1, e_2, ..., e_m\}$, describing all possible position transitions
- A is the 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 definition of an action will serve as a more explanatory and planning-like definition, but will not change the meaning of the previous action definition.

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 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 each action the same cost of 1, is used.

As in [1], we 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 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 $\pi_i[x] = \pi_j[x+1]$

3.2 MAPF with blocking containers

To describe a MAPF with blocking containers (MAPF-BC) problem, we have to change the previously introduced formalization. It can be described as follows.
A MAPF-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, $C = \{c_1, c_2, ..., c_m\}$
- $s' : C \to V$ 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 $v_* \in V$

Since we introduce, a new type of agent, which change the physics of the environment, we introduce new set of possible predicates for the preconditions and effects of the actions:

- $at(a, v)$, 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 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:

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

The effects of the actions are :

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

We will still assign a uni-cost of 1 to all the actions, meaning the action will take 1 time step to be executed. This can be changed to better suit real life settings, but has to be considered in the planning algorithms.
To formalize conflicts, we will use the function $\pi_i[t] = v$, the same as in classical MAPF, and the function $\gamma_i[t] = (t, v_t)$ returns a tuple, where the first position indicates the affected agent and its position after agent i executed the action at time step t in the plan of i. Look at figure add figure!, for some examples.
In MAPF-BC, we will only be interested in the following conflicts:

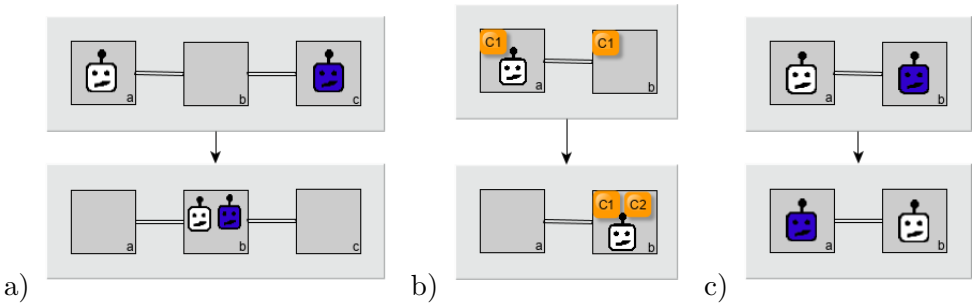
- **agent vertex conflict**: two agents occupy the same vertex an the same time,
  if $\exists a_1, a_2 \in A$ and $c, c' \in C$: $(\pi_{a_1}[t] = \pi_{a_2}[t])$ or $(\gamma_c[t] = (a_1, v_i) \wedge \gamma_{c'}[t] = (a_2, v_i))$
- **container vertex conflict**: two containers occupy the same vertex,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

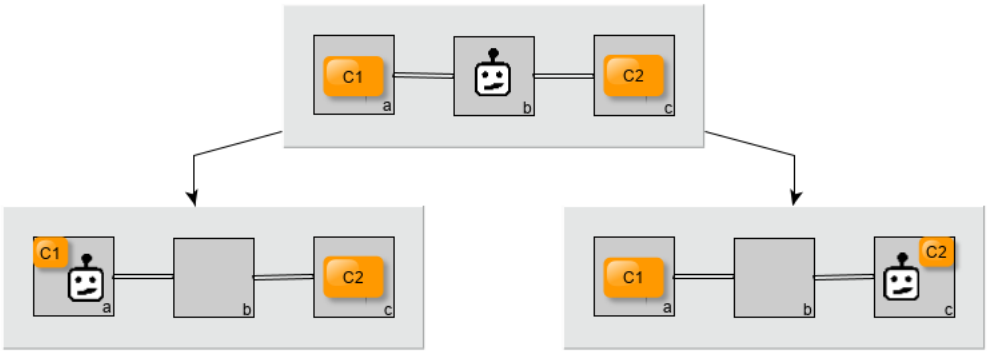
- **Swapping Conflict**: two agent swap vertices over the same edge,
  if $\exists a_1, a_2 \in A$ and $c_1, c_2, c_3, c_4 \in C$: $(\pi_{a_1}[t] = \pi_{a_2}[t-1] \wedge \pi_{a_1}[t-1] = \pi_{a_2}[t])$ or $\gamma_{c_1}[t+1] = (a_1, v_i) \wedge \gamma_{c_2}[t] = (a_2, v_i) \wedge (\gamma_{c_3}[t] = (a_1, v_j) \wedge \gamma_{c_4}[t+1] = (a_2, v_j))$

We introduce $p : \{A, C\} \rightarrow$ active, passive a bijective function, that determines which Agent set does the planning. At most one set can be active. An agent is active if $e \in EandEisactive$. It is important to note, that agent conflicts may only occurs if the set of containers is the active planning set. As in classical MAPF, we can try to optimize the makespan or the sum of cost:

- **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.
- **Sum of costs** is the sum over all the actions for each active agent: $\sum_{n=1}^{k} |\pi_i|$ for all agents i in the active planning set.

add agent behaviour upon reaching the goal and describe feasible path





4 Approach

add motivation of why choosing CBS 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.

4.1 Conflict Based Search with Fixed Container Assignment CBS-FCA

add vocabulary explanation. The first attempt to solve this problem is using CBS with a Fixed Container Assignment to solve the problem optimally. CBS has first been introduced by [2] in the year 2017.

Algorithm 1: CBS-CA High Level

Input: MAPF instance and fixed container assignment for each agent
OPEN ← an empty list
$Root.constrains = \emptyset$
$Root.solution$ = find individual paths by the low level()
$Root.cost$ = 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_i in C* **do**
      $new\_constraint = (a_i, v, t)$
      Create_new_node(P, new_constraint)
  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_l in C* **do**
      $new\_constraint = (a_l, v_l, v_k, t)$
      //v_l is the position of a_l before time step t
      //v_k is the position of a_l after time step t
      Create_new_node(P, new_constraint)
  else if *C is a Container Conflict* **then**
    //C is a Conflict of type (a_i, c_i, v, t)
    $new\_constraint = (a_l, v_l, v_k, t)$
    Create_new_node(P, new_constraint)

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 performs a best first search on the search nodes, which each consist of a set of constrains, a solution and a cost value. The first search node has an empty set of constrains, so the agents do not have any constrains 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, due to performing a best first search. Otherwise, the high level algorithm will generate new nodes depending on the conflict found, see lines add line numbers. For each new node, we update the path of the agent, affected by the constrain, as describes in Algorithm 2.For

further Information and an example of the CBS Algorithm can be found in [2]

Algorithm 2: Create new node

Input: P the parent node and C a new constraint
A ← new node
$A.constrains \leftarrow P.constrains + C$
$A.solution \leftarrow P.solution$
Update $A.solution$ by invoking low level(a_i)
$A.cost$ = SIC($A.solution$)
if $A.cost < \infty$ *//A solution was found* **then**
  Insert A to OPEN

4.1.1 Add new nodes to the OPEN list

CBS-CA validates a path until the first conflict between two agents is found and does not check for more agents to be 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:

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 constrains for the new node.
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.
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 may only be classified as a container conflict, if an agent a moves with its container c_a to a vertex v already 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 constrains for the new node and will deny the agent to move to vertex v with its container at time step t.

4.1.2 the low level algorithm

For the low level Algorithm, I introduce a new algorithm called TA^*, see algorithm 3. Given a set of locations, which have to be visited in a certain order, TA^* will find the optimal path 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 constrains, it is important to pass the length of the previous sub paths (time) to the next invocation of A^*.
Let k be the number of agents and m be the number of containers. In the following, I will show, that this algorithm will return a optimal solution, if $K \leq M$, and might not return a solution at all, if $K > M$.

Algorithm 3: TA* Low level

Input:
- ordered list of of goal positions L
- start position s
- a graph G
- set of constrains C

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

4.1.3 Theoretical Analysis

Assumptions:
- Each container is assigned to exactly one agent and vise versa.
- Upon reaching its container, the agent will not move alone anymore.
- Let K be the number of agents
- Let M be the number of containers.

These assumptions hold the way CBS is implemented.

First Case: $K \geq M$

Lemma 1: *Each agent will find an optimal path to its containers start position*
Proof 1:
Since CBS is optimal and complete, CBS will always find a solution, if there is one. 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, does not change the classical MAPF problem. This implies, that CBS will find such a path for each agent, if there is one. □

Lemma 2: *Each container find 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 has reached its 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 had two CBS nodes, each with one of the constrains (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: $K < M$

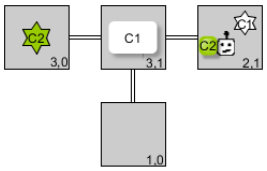


Figure 1: If the agent first moves C1, it cant reach its goal, because of a collision with container C2. First moving C2 is not possible, due to an immediate collision with C1. The agent has to move C1 to vertex c, then C2 to d and finally C1 to a.

It can be demonstrated with a very simple example, see figure 1, that having more containers than agents can lead to unsolvable problem instances for CBS-CA. Therefore CBS-CA is only suited for cases, where $K \geq M$, otherwise completeness and optimality cant be guaranteed.

4.2 Conflict Based Search with Optimal Target Assignment CBS-OCA

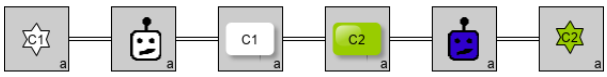


Figure 2: The assignment $\{A_{white} \rightarrow C_2\}$ and $\{A_{blue} \rightarrow C_1\}$ will not lead to a solution. However assigning $\{A_{white} \rightarrow C_1\}$ and $\{A_{blue} \rightarrow C_2\}$ will find the optimal solution.

Although CBS-FCA solves the problem optimally for a given fixed container assignment, most of the time it will lead to a better solution, if the tasks are assigned optimally. It can even be the case, that ,for a fixed container assignment, a problem instance might not be solvable at all, but becomes solvable using an other assignment, see figure 2

5 References

References

[1] Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks -Stern et al.

[2] Conflict-based search for optimal multi-agent pathfinding - Sharon, Stern, Felner, Sturtevant