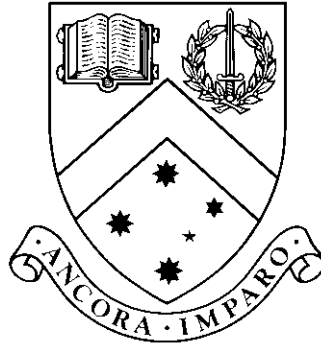


School of Computer Science (BICA)  
Monash University



Literature Review, 2017

## Review of optimal multi-agent Pathfinding algorithms and usage in warehouse automation

Phillip Wong

Supervisors: Daniel Harabor,  
Pierre Le Bodic

**Contents**

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Related Work</b>	<b>1</b>
<b>3</b>	<b>Algorithm</b>	<b>1</b>
3.1	High Level . . . . .	1
3.2	Path Generation . . . . .	1
3.3	Path Assignment . . . . .	2
3.4	Temporal A* with collision penalties . . . . .	2
<b>4</b>	<b>Resolving conflicts</b>	<b>3</b>
<b>5</b>	<b>Master problem formulation</b>	<b>3</b>

# 1 Introduction

In our multi-agent pathfinding problem, we have an environment containing a set of  $k$  agents on a grid-map. Each agent aims to find a path to their goal without colliding with another agent (a path collision).

Hence we have a centralized agent coordinator which aims to resolve path collisions.

## 2 Related Work

CBS

ICTS

Centralised A\*

That Network Flow paper

## 3 Algorithm

Each agent knows of a number of paths to the goal. Using a Mixed Integer Program (MIP) we are able to solve the assignment problem of choosing a combination of paths such that no assigned two paths has a collision. Hence the two main parts of the algorithm:

Any specifics about the MIP I should mention here?

1. **Path Assignment** The Mixed Integer Program, Responsible for choosing a assigning paths to agents, while finding a conflict-free solution.
2. **Path Generation:** Responsible for generating alternative paths for agents whose current paths are in collision.

### 3.1 High Level

At the high-level we describe the cycle of path generation and assignment. Simply: while the path assignment MIP fails to find a solution, we run the path-generation method.

---

**Procedure 1** high-level of the Algorithm

---

**Input:**  $\vec{a}$  : list of agents

**Output:** *void*

- 1:  $collisions \leftarrow \{\}$
  - 2: **do**
  - 3:    $pathCollisions \leftarrow GeneratePaths(\vec{a})$
  - 4:    $failedAgents \leftarrow AssignPaths(paths, pathCollisions)$
  - 5:    $collisions \leftarrow failedAgents$
  - 6: **while**  $collisions$  is not empty()
- 

### 3.2 Path Generation

This method aims to generate paths for an agent. The first iteration of the path generation will use A\* to quickly find the shortest path to the goal. This path will likely be in conflict. Next we use Temporal A\* which aims to avoid collisions by applying penalties to tiles which are in conflict. We increments the penalties by 1 and iteratively finds paths of the next cost.

A generated path may have a longer length than the optimal single-agent solution. Hence we describe this difference to the optimal path as  $(\Delta)$ . We do not generate a path for this agent until other agents in the deadlock have the same  $\Delta$ . In this way we find an optimal solution.



$a1$		
$t2$		$t1$
		$a2$

Table 1: A deadlock occurs here. The optimal solution is

$a1$ :  $d, r, r, u, d, w$

$a2$ :  $w, w, w, u, l, l$

This deadlock occurs as the solution for  $a1$  requires the algorithm to search past the goal.

However our implementation of Temporal A\* is not complete. There are instances which it is unable to find a solution. If the optimal shortest path requires the agent to move past the goal tile. Then Temporal A\* fails and will never expand past the goal. An example of this is can be seen in Figure 1. Hence in this algorithm we determine that agents are in **deadlock** when they have  $n = 100$  number of collisions. If this occurs we start finding paths with Centralized A\*.

---

#### Procedure 2 GeneratePaths

---

**Input:**  $\vec{a}$  : list of penalty agents

**Output:**  $pathCollisions$ : list of paths which are in collision: a tuple of (agent, path)

```

1:  $pathCollisions \leftarrow \{\}$ 
2: for agent  $a$  in  $\vec{a}$  do
3:    $path \leftarrow \{\}$ 
4:   if agent.paths is empty then
5:      $path \leftarrow AStar$ 
6:   else if  $\exists$  agents in deadlock with  $a$  then
7:      $path \leftarrow CentralizedAStar(\text{agents in deadlock})$ 
8:   else
9:      $path \leftarrow TemporalAStar$ 
10:   $path \leftarrow GeneratePath(a, firstRun)$ 
11:   $a.\vec{p}.append(path)$ 
12:   $pathCollisions.append(a, CheckCollisions(path))$ 
```

---

### 3.3 Path Assignment

At the core, this method simply calls the Master problem (5). The objective here is to detect if any agents were unable to be assigned paths (meaning there was no combination of paths which resulted in a collision-free solution). These agents are assigned the penalty variable by the mip and are returned as output of this method.

---

#### Procedure 3 AssignPaths

---

**Input:**  $\vec{a}$  : list of agents  $\vec{c}$  : list of collisions

**Output:**  $\vec{p}$  : list of agents who were assigned the penalty

```

1:  $solution \leftarrow RunMasterProblem(\vec{a}, \vec{c})$ 
2: for  $a$  in  $solution.assignedAgents$  do
3:   assign collision-free path to  $a$  as described by  $solution$ 
4: return  $solution.penaltyAgents$ 
```

---

### 3.4 Temporal A\* with collision penalties

Here we describe our time-expanded A\* implementation. The goal of this algorithm is to incrementally find paths of increasing cost to ensure optimality. We do this applying a

penalty to tiles when collisions occur. Collision detection is described in Section 3.2.

---

**Procedure 4** Temporal A\* with collision penalties

---

**Input:**  $s$  : start,  $g$  : goal,  $\vec{c}$  : collisionPenalties, a vector of tile at time to penalty

**Output:**  $f$  : comparator

1:  $f(n) = g(n) + h(n) + c(n)$

2: **return**  $path$

---



---

**Procedure 5** Temporal A\* with collision penalties

---

**Input:**  $s$  : start,  $g$  : goal,  $\vec{c}$  : collisionPenalties, a vector of tile at time to penalty

**Output:**  $p$  : path

1:  $path \leftarrow \{\}$

2: **if**  $s$  or  $g$  is not valid or  $s$  equals  $g$  **then**

3:     **return**  $path$

4:  $open \leftarrow PriorityQueue()$

5: add  $TileTime(start)$  to  $open$

6: **while**  $open$  is not empty **do**

7:      $current \leftarrow open.pop()$

8: **return**  $path$

---

## 4 Resolving conflicts

1. Given a set of paths,  $S$  which contains all agent's path, find a new path for each agent their goal and add it to  $S$
2. Detect any path collision for each path
3. Convert the paths to MIP variables and path collisions to constraints
4. Repeat 1. if there is not a valid solution found i.e the optimal solution contains a path collision

## 5 Master problem formulation

### Branch and bound

#### Column generation

#### Branch and price

Each agent is given *one or more* paths to their goal. The master problem aims to assign one path to every agent while minimizing the path distance and avoiding path collisions.

- **Agents:** The MIP is given a set of agents,  $A$
- **Potential paths:** Every agent has a set of paths,  $P$  describing a unique path from an agent's position to their goal.
- **Penalty:** A penalty variable,  $q_i$  is added for every agent in the case that all the agent's paths are in collision. If the MIP chooses this penalty in the solution, then the MIP was unable to find an assignment of paths which resulted in a conflict-free solution. The cost of the penalty is set to be larger than the expected maximum path length (here it is 1000).
- **Conflicts:** A set of conflicts,  $C$  is provided to the MIP. A conflict is a set of paths, describing that these paths are not allowed to be assigned at the same time as they will cause a conflict.

We generate a variable for each path and the cost is set to the path length.  
We specify an agent's path as  $p_{ij}$ . Penalty  $q_i$ . Path collision as  $c_{nm}$ .

$$\min \sum_{i \in A} \sum_{j \in P_i} (d_{ij} * p_{ij}) + q_i \quad (1)$$

$$\text{subject to } \sum_{j \in P_i} (p_{ij}) + q_i = 1, \forall i \in A \quad (2)$$

$$\sum_{m \in P_c} p_{cm} \leq 1, \forall c \in C \quad (3)$$

$$p_{ij} \in 0, 1, \forall i, j \quad (4)$$

$$q_i \in 0, 1, \forall i \quad (5)$$

For example ?, our generated variables are:  $5a_0p_0 + 5a_0p_1 + 1000q_0 + 2a_1p_0 + 2a_1p_1 + 1000q_1$ .

Agents are assigned

## References