CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Multi-agent pathfinding with discrete speeds

and its application for vehicle road networks

Master's thesis in Computer science and engineering

Johan Gerdin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Multi-agent pathfinding with discrete speeds

and its application for vehicle road networks

Johan Gerdin

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Multi-agent pathfinding with discrete speeds
and its application for vehicle road networks
Johan Gerdin

Supervisor: Elad Michael Schiller, Department of Computer Science and Engineering
Examiner: Nir Piterman, Department of Computer Science and Engineering.

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2020

Multi-agent pathfinding with discrete speeds
and its application for vehicle road networks
Johan Gerdin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This work contributes to the multi-agent path-finding problem (MAPF) by extending existing algorithms with optimal and efficient routing for roads with multiple lanes and intersections. It also considers the continuous case, where agents are forbidden from performing actions during a non-discrete interval of time. Further, a method to plan for the speed of agents is introduced. In order to create a strong base for this problem, rigorous mathematical models are defined and proved to have properties of both completeness and optimality. Experiments are used to validate the abstract models.

Keywords: Multi-agent path-finding, kinematic constraints, speed.

Acknowledgements

I would first of all like to give a big thank you to my supervisor Elad, who has supported this project and myself with a lot of time and energy and also great enthusiasm. I would also like to thank my examiner Nir for taking an interest in this project. Furthermore, I would like to give credit to Andreas Rosenfeld, Andreas Kuszli and Jesper Åberg who have contributed to discussions during the project and also helped with programming of the evaluation.

Johan Gerdin, Gothenburg, July 2020

Contents

List of Figures

List of Tables

1 Introduction

Optimal route planning for autonomous agents, called a *Multi-agent path-finding problem (MAPF)*, is an NP-hard problem [15]. This means that, in general, problem instances can take exponential time to compute. If during execution there are deviations to the plan, agents might need to halt for a long time while recomputing their paths. Existing attempts to mitigate the risk of recomputation during the planning phase considers, for example, unexpected yet bounded delays [3].

We consider the problem of routing vehicles in a road network. The network is represented by a graph, consisting of straight two-lane roads and intersections. A MAPF solver will create an optimal plan for the vehicles that schedules them to move without colliding with each other. The solvers described in this project are based on *conflict based search (CBS)* by Sharon *et al.* [14]. It plans for agents by focusing on eliminating conflicts in the form of collisions, where two agents occupy the same space at the same time.

This algorithm is the basis for a framework, consisting of three models in sections 2, 3 and 4 respectively. Each model adds to or changes the previous, bringing in more kinematic details such as time and speed. A way to translate from the *target system* of road networks into the model and back will be presented in Section 5. The third model described in Section 4 is one of the novelties of this project. It focuses on planning for the speeds of agents under continuous time intervals. During refinement in Section 5, the other two models will be brought closer to this third model. This is another novelty, refining all models in the framework to consider speeds of agents. A preliminary evaluation was done of the framework. The evaluation shows evidence of gains in execution time performance from considering the speeds of agents.

An example of previous work in the field of vehicle routing is the work of Petig *et al.* [12]. Their work provides a polynomial-time algorithm for planning lane changes on a highway. They provide a way to reason about the movement of vehicles in a road but it does not account for the kinematics of the vehicles, which this thesis is attempting to do. Another example is Ekenstedt *et al.* [4], which involves a protocol for how vehicles should behave in intersections. Her work focuses on coordinating vehicles in a running system, while this thesis focuses on the planning phase. Ma *et al.* [11] presents a sub-optimal algorithm allowing for both rescheduling and encoding kinematic constraints for disk-shaped agents operating on a grid. This algorithm could potentially be extended to road networks, but due to its restriction to grids and agents of one shape, is not immediately suited for road networks.

1.1 Problem description

We model the problem of routing vehicles (agents) in a road network as an instance of a multi-agent path-finding problem. The road network is represented by a graph consisting of vertices (locations) and edges (intermediate positions) between each vertex. The goal is to compute a *solution* (also called a *plan*) consisting of collision-free *paths* for multiple individual agents. Each agent has a start and goal state, which consists of a vertex in the graph, and in the case of the CTDS model also a start and goal speed. Paths in a solution are called collision-free if they do not schedule agents to occupy, at the same time, the same location (or concurrently move between locations in a way the violates safety conditions).

This thesis considers different models, some of which, such as the *target system (TS)* consider all the kinematic constraints that the agents must satisfy when running in the environment (e.g. road network) in which the vehicles are planned to operate in. These constraints include detailed descriptions regarding continuous positions, velocities and acceleration. The other modeling approaches in this thesis, vary from the most basic that do not consider any kinematic constraints to ones that consider more of these constraints. The goal, in all of these models, is to find optimal collision-free paths for all agents.

Since some of our models consider several layers of abstractions, post-processing of the set of planned paths is required to properly account for the kinematics of the agents and environment [10]. By considering several models, we can select to include the right level of details about the environment and agents and balance the trade-offs between computation costs of the MAPF algorithms vs. the execution time when the agent follows the plan in the target system.

1.2 Models

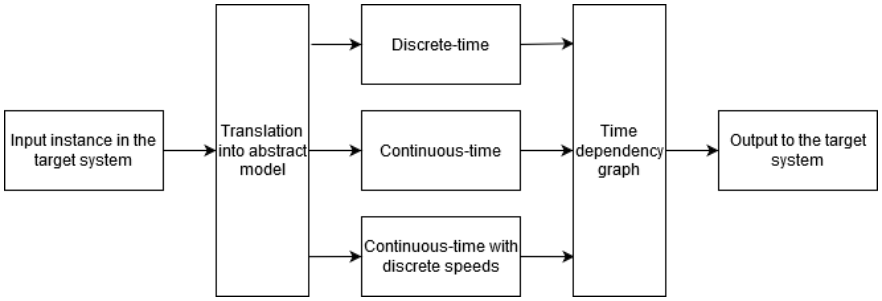


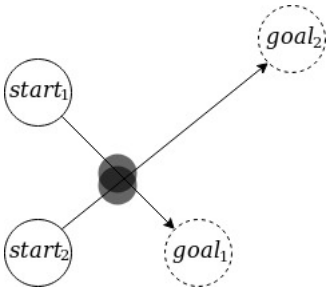
Figure 1.1: Overview of transformation from and to the target system.

	Discrete			Continuous			
	Time	Speeds	Positions	Time	Speeds	Positions	Geometry
Discrete-time (DT)	x		x				
Continuous-time (CT)			x	x			x
CT with discrete speeds (CTDS)		x	x	x			x
Target system				x	x	x	x

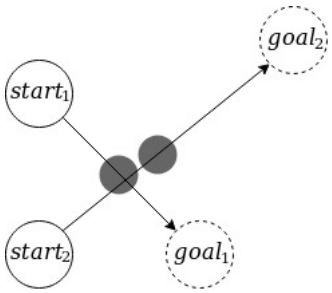
Table 1.1: Summary of what kinematic constraints are considered in the three models.

The models differ in the manner in which they define safety conditions. One of the most basic ways to define this condition is by requiring no two agents to be present simultaneously at the same vertex or edge. Sharon *et al.* [14] follow this approach and assume that time can be divided in a discrete manner. A solution in this *discrete-time (DT)* model considers a set of paths, such that each path is the itinerary $\{location_1, \ldots, location_n\}$, which lists locations and times an agent is scheduled to go through. The safety conditions of the DT model are easy to verify. For example, consider one agent with path $\{location_1, location_3\}$ and another agent with path $\{location_2, location_3\}$. In this example, both agents are scheduled to reside in $location_3$ at time 2, and thus the solution is not valid. Sharon *et al.* [14] use the DT model for capturing inherent scheduling conflicts, such as the one in the example above. Sharon *et al.* demonstrated the existence of an attractive MAPF algorithm that offers valid and optimal solutions for many relevant cases.

Since the DT model does not consider any kinematic constraints, Hönig *et al.* [10] proposed a transformation of Sharon *et al.*'s solutions from the DT model to the one of the target system by encoding the missing kinematic constraints. We note, however, that there are many optimal solutions that are safe at the target system but are considered not valid by the DT model. Consider Figure 1.2, where two agents have paths that cross. Figure 1.2a illustrates how the agents will collide if they start to move at the same time. This can be resolved by letting one of the agents wait in its starting location before moving. An example of this can be seen in Figure 1.2b, where the agent moving from $start_1$ to $goal_1$ waited in $start_1$ before moving. A solution to this in the DT model would either be represented as by paths $\{start_1, start_1, goal_1\}$ and $\{start_2, goal_2\}$ or $\{start_1, goal_1\}$ and $\{start_2, start_2, goal_2\}$. Both paths represent one agent waiting until the other agent is completely done moving before moving itself. This is sub-optimal if we consider time as non-discrete. A non-discrete solution is Figure 1.2b, where the agent moving from $start_1$ to $goal_1$ waited for a continuous-time interval before moving.



(a) The two agents collided during movement.



(b) One of the agents waited before moving, allowing the ohter to pass.

Figure 1.2: A graph consisting of four locations and two agents with crossing paths.

With this as a motivating example, Andreychuk *et al.* [1] refine the safety conditions used by Sharon *et al.* by treating time as continuous. Agents are now represented as objects with a given size and shape, moving and colliding at any point in time. A path in a *continuous-time (CT)* model includes a time-point (positive real number) along with each location: $\{(location_1, time_1), \ldots, (location_n, time_n)\}$. This allows a solver in this model to create paths in an environment with locations placed at a non-uniform distance from each other and for agents with different maximum speed, size and shape. Consider Figure 1.2, where instead of waiting for the other agent to completely finish its move, the agent waits until it can move without colliding. The interval of time the agent waits can be a smaller interval of time than the duration of the other agents movement. It could, for example, be based on the collision interval as seen in Figure 1.3. However, they make one important assumption; that an agent has infinite acceleration and will reach maximum speed from zero speed instantly and vice versa. Agents in a target system will require time to reach their target speeds. Further, there is a solution for the agents in Figure 1.2 where both start to move at the same time, but at different speeds. This solution would result in both agents reaching their locations and target speeds earlier than if one agent stalls until the other is done moving.

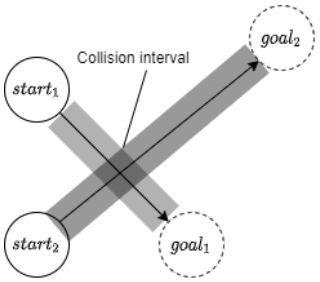


Figure 1.3: Example of how a collision interval might be computed.

The *continuous-time with discrete speeds (CTDS)* model in Section 4 is an attempt to remedy this. It extends the CT model by refining the concept of speeds. Assuming constant acceleration, agents can travel by a set of speeds. This treatment of speeds can gain performance by catching opportunities of not coming to a complete stop to avoid a collision. Also, agents in the target system are subject to safety conditions (friction on the road, forces upon the vehicle, etc) that can be better captured here. An action by an agent in the CTDS model will take a different amount of time and is validated based on the speed and distance traveled. This is achieved by providing both a time-point and a speed along with each location in a path: $\{(location_1, time_1, speed_1), \ldots, (location_n, time_n, speed_n)\}$. A solver can now encode that an agent is not able to go from top speed to zero in too short of a distance or that going from 1 m/s to 3 m/s takes longer than going from 2 m/s to 3 m/s for example.

The three models and target system are summarized in Table 1.1. The table gives an overview of the

search-space of each model and how each model builds on the next, bringing it closer to the target system. A solution that has accounted for all kinematics of the target system would have to search in both continuous-time, speed and position. In general, approaches that account for all the details do not scale very well with larger input sets. So, as is apparent in the table, no model explores the space of continuous speed and position. An abstraction made by all models is that positions are discrete points in the geography of the target system.

An overview of transformation from and to the target system is given in Figure 1.1. The transformation and refinement of each model into the target system of a *road network* is described in Section 5. The general procedure is as follows:

1. The input in the target system is transformed into a graph in either the DT, CT or CTDS model.
2. Graph and to a small extent algorithms are refined with information relevant to the choosen abstract model.
3. A solution is created based on the refined graph.
4. The solution is post-processed based on the work of Hönig *et al.* [10] in order to solve for continuous speeds.

1.3 Research questions

The purpose of this work is to investigate how to provide an optimal and safe plan for many vehicles in a target system, *i.e.,* the road. One solution to this is to plan for the target system directly, using a detailed physical simulation. This can be very expensive, however, and another approach is use a high-level model of the problem. In order to decrease running time, these models try to capture properties of the target system without doing a full simulation. This work attempts to answer what properties of the target system we model and how we model them. Moreover, there is a trade-off of how detailed these properties should be. For example, how fine-grained should the set of speeds be that an agent can travel by? Lastly, the studied algorithms should be suitable for a large number of agents. The aim is to answer this by seeing how runtime is affected by having larger sets of agents.

This can be summarized as the following research questions:

- How to provide an optimal and safe plan for many vehicles on the target system, *i.e.,* the road?
- How to consider the details of the target system when modeling the problem?
- How to balance the running time trade-off between the studied model and the target system?
- How well do the studied solutions scale in the number of agents

1.4 Our contribution

We contribute to the field of MAPF by describing a novel model and algorithms for planning for the speeds of agents. This work is based on two previous contributions, *conflict-based search (CBS)* [14] and *continuous conflict-based search (CCBS)* [1]. Both of these are modeled in this text. Moreover, we give a novel approach to solve for the target system of road networks for all three models.

1.5 Applications

This thesis focuses on the specific application of road networks. The network is separated into two components: a two-lane road and a small intersection with four entrances and four exits. All models and solutions presented are intended to be general however, and can be adapted to many fields such as video games or warehouses [2].

2 Discrete-time (DT) model

Paths are computed in a given graph, where an agent traversing the graph represents, for example, a robot operating on a grid. The agent use actions to traverse the graph. Each action can let an agent either stand still or move from one location to another. More generally, these actions transition the agent from one state to another state, where a state in the DT model refers to the agent's current location. Each transition has an associated cost, which in the DT model is uniform for all actions. Agents are assumed to perform actions simultaneously. The notion of time refers to an integer that is used when representing conflicts between agents. In the DT model, a conflict between two agents exists if they are scheduled to reside at the same location at the same (discrete) time. For a given solution in the DT model, the costs related to time is the maximum number of state transitions any agent has to go through until the end of its path, *i.e.,* makespan. The cost related to the total amount of work can be defined as the sum of all individual costs, which is the sum of the lengths of all individual paths in a given solution. Next, we provide a more detailed version of these definitions.

2.1 Definition

The model considers a set of k agents $\mathsf{A} = \{a_1, \ldots, a_k\}$ and a directed *non-weighted* graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$. Each agent $a_i \in \mathsf{A}$ traverse the graph by transitioning from one state st to another state st', representing e.g. changing the location of the agent.

Definition 2.1 (State)**.** A $state\{v \in \mathsf{V}\}$ is defined to consist of

- A vertex $state.v \in \mathsf{V}$ representing the location of the agent.

The transition from state st to state st' has an associated cost $tranCost(st, st')$. This is used in Section 2.2 to define what the cost of a path is.

Definition 2.2 (State transition cost)**.** A *state transition cost* $tranCost(st, st')$ from state st to st' is defined to be equal to 1.

Further, each agent $a_i \in \mathsf{A}$ starts at location $v_i^{start} \in \mathsf{V}$ and it needs to traverse the graph to reach location $v_i^{goal} \in \mathsf{V}$, via a series of *actions*.

Definition 2.3 (Action)**.** An *action* is defined to be either:

- STOP(st): Remain in the same state st.
- MOVE(st, st'): Move to a new state st' from st, such that $(st.v, st'.v) \in \mathsf{E}$.

In other words, agent a_i moves from v_i^{start} to v_i^{goal} along the path $p_i = (state\{v_i^{start}\}, \ldots, st, st', \ldots, state\{v_i^{goal}\})$, where the move that a_i takes in state st is MOVE(st, st') when $st.v \neq st'.v$ and STOP(st) otherwise.

This model treats time as discrete, meaning that agents perform actions simultaneously. Therefore, time can be represented as follows:

Definition 2.4 (Time)**.** The *timed representation* of a path p_i is $p_i = (st_1, st_2, \ldots, st_t, \ldots, st_\ell)$, where $st_t.v$ is the location of agent a_i at time t and ℓ is the path length.

We say, in this paper, that $s = \{p_1, p_2, ..., p_k\}$ is a solution, consisting of the individual paths of *all agents*, to the problem of *Multi-Agent Path-Finding* (MAPF) if it satisfies the safety condition of no *conflicts*.

Definition 2.5 (Conflict)**.** A *conflict* between two agents a_i and a_j in states st_i and st_j at time t is one of two types:

- (Vertex conflict) (a_i, a_j, v, t) where agents a_i and a_j plan to occupy the same vertex $v = st_i.v = st_j.v$ at the same time step t.
- (Edge conflict) $(a_i, a_j, (v, v'), t)$ where agents a_i and a_j plan to occupy the same edge. That is, at time step t a_i plans to move from $v = st_i.v$ to v' and a_j plans to move from $v' = st_j.v$ to v.

To the end of finding a conflict-free solution, Sharon *et al.* [14] propose to use a *constraint tree (CT)*. Given a solution s that has a conflict, one can refine s by allowing a_i to keep its current plan with state st_i at time t while forbidding the a_j from transitioning to state st_j at time t. Each node in a CT refines the solution by adding constraints.

Definition 2.6 (Constraint)**.** A *constraint* for an agent a_i is one of two types:

- (Vertex constraint) Given a vertex conflict (a_i, a_j, v, t), (a_i, v, t) is a constraint where agent a_i is forbidden from occupying vertex v at time-step t.
- (Edge constraint) Given an edge conflict $(a_i, a_j, (v, v'), t)$. $(a_i, (v, v'), t)$ is a constraint where agent a_i is forbidden from traversing edge (v, v') by being at v at time t and then at v' at time $t+1$. Similarly, $(a_j, (v', v), t)$ is a constraint for agent a_j.

Consider a conflict between two agents a_i and a_j, a CT considers *both* the constraint (a_i, v, t) or $(a_i, (v, v'), t)$ and (a_j, v, t) or $(a_j, (v', v), t)$. Specifically, starting from a the CT root, which includes no constraints, each node encodes a different list of constraints that refines the solution. That is, the list of any non-root node is the result of appending a constraint to the list of its parent.

A path $p_i \in s$ is called *consistent* if it satisfies all constraints for agent a_i and a solution is called consistent if all single-agent paths in the solution are consistent. The solution in a CT node is always consistent with its constraint list and a CT node is called a *goal node* when the solution has no conflicts. Therefore, we say that goal nodes provide *valid* solutions.

2.2 Cost function for MAPF

A solution s to a MAPF problem instance is called optimal, in this text, if it is a minimum of the *multi-agent objective function cost*(s). All three models will use this function to order potential solutions, processing ones with the lowest cost first. *cost*(s) is computed using a composite value of a *single-agent objective function*, considering each agent in the solution. The single-agent objective function computes the optimal cost of an agent moving from its starting location to its goal location.

2.2.1 Single-agent objective function

A single-agent objective function, $h(v)$, computes the optimal cost of a single agent moving from $v \in \mathsf{V}$ to a goal vertex $g \in \mathsf{V}$. Computing $h(v)$, however, might be expensive in that many graph vertices must be processed. To the end of reducing the computation cost, the heuristic function $\hat{h}(v)$ is used. The aim of $\hat{h}(v)$ is to estimate the cost of $h(v)$. It can be implemented in many ways, for example: The euclidean distance from v to g or the time it would take to travel in a straight line from v to g at a certain speed. If $\hat{h}(v)$ is an *admissible heuristic* it has the property: $\forall_{v \in \mathsf{V}} \hat{h}(v) \leq h(v)$. Meaning, $\hat{h}(v)$ never overestimates the cost of moving from v to goal vertex g. The path-finding algorithms in this work also considers a constraint set C. To this end, we introduce $h_C(v)$ as the true cost of moving from v to a goal vertex g while respecting constraint set C. We now prove that, given a set of constraints C, an admissible heuristic for $h_C(v)$ still exists.

Theorem 2.1. Let C be a set of constraints and $h_C(v)$ an individual objective function. Suppose that $h_\emptyset(v)$ has an admissible heuristic $\hat{h}(v)$. Then, $h_C(v)$ has an admissible heuristic $\hat{h'}(v)$ also for the general case in which $C = \emptyset$ may not hold.

Proof: Let $\mathsf{G}(\mathsf{V}, \mathsf{E})$ be a graph, $g \in \mathsf{V}$ be a goal vertex and C a constraint set. We argue that $\forall_{v \in \mathsf{V}} h_\emptyset(v) \leq h_C(v)$, which gives us that $\forall_{v \in \mathsf{V}} \hat{h}(v) \leq h_\emptyset(v) \leq h_C(v)$, meaning $\hat{h}(v)$ is actually an admissible heuristic for $h_C(v)$. By definition of $h_\emptyset(v)$, it is true that $h_\emptyset(v)$ always computes the optimal cost of moving from v to g. Since $h_\emptyset(v)$ considers no constraints, the path that $h_C(v)$ computes the cost of is a possible candidate for $h_\emptyset(v)$ as well. This means that it can never be the fact that $h_\emptyset(v) > h_C(v)$ for any $v \in \mathsf{V}$. ■

Corollary 2.1. If there exists an algorithm that computes $h_\emptyset(v)$ using an admissible heuristic $\hat{h}(v)$ then the same algorithm can use $\hat{h}(v)$ to compute $h_C(v)$.

2.2.2 Multi-agent objective function

We are now ready to define a cost function for multiple agents. Let *cost*(s) be a function that takes a solution, s, and computes the cost of s via a composite value of the costs of the cost of individual paths in $s = \{p_1, \ldots, p_k\}$. Note that we consider a discrete set of costs in this work. Further, we require *cost*(s) to

satisfy Property 2.1. The definition of Property 2.1 considers a constraint set C and the minimal $cost[C]$ of function $cost(s)$ for a solution s that satisfies constraint set C.

Property 2.1. Let $cost(s)$ be a cost function, then $\forall_{C,C'} C \subseteq C'$ we have that $cost[C] \leq cost[C']$.

Examples of $cost(s)$ are the *sum of individual costs (SIC)* and the *Makespan*. Let s be a solution and the cost of a path $p \in s$ be

$$pathCost(p) = \sum_{st_i, st_{i+1} \in p} tranCost(st_i, st_{i+1})$$

The SIC given a solution s is:

$$SIC(s) = \sum_{p_i \in s} pathCost(p_i) \tag{2.1}$$

Then, SIC optimality is defined as: $SIC^*_C = \sum_{a_i \in \mathsf{A}} h_C(v_i^{start})$ is the optimal SIC for agents A in a given MAPF problem instance. If a solution s satisfies constraint set C and has a cost $SIC(s) = SIC^*_C$, then s is called optimal w.r.t. SIC and the constraint set C.

The Makespan given a solution s is:

$$Makespan(s = \{p_1, \ldots, p_k\}) = \max(pathCost(p_1), \ldots, pathCost(p_k)) \tag{2.2}$$

Then, Makespan optimality is defined as: $Makespan^*_C = \max_{a_i \in \mathsf{A}}(h_C(v_1^{start}), \ldots, h_C(v_k^{start}))$ is the optimal Makespan for agents A in a given MAPF problem instance. If a solution s satisfies constraint set C and has a cost $Makespan(s) = Makespan^*_C$, then s is called optimal w.r.t. Makespan and the constraint set C.

The fact that the SIC and Makespan satisfy Property 2.1 follows trivially from that the set of solutions that satisfy C' also satisfy the constraint set C.

2.3 Single-agent path-finding: Constrained-A*

A very common path-finding algorithm, when computing a path for a single agent, is A* [8, 9]. The algorithm is a best-first search. It explores the state-space by ordering states from low to high cost. The cost of a state is based on both the time it takes to get to that state and also the estimated future cost of moving from that state to the goal state. Constrained-A* is a modified version of A* that also takes as input a constraint set C. When computing successors to a node in the search tree, the successor must be valid under constraints C.

Constrained-A* can be formulated using the discrete-time model. The pseudo-code can be seen in Algorithm 1. Solvers described in the two other models in this text will be seen using this structure as well, but adding to the functionalities of the algorithm. The procedure Constrained-A*$(v_i^{start}, v_i^{goal}, \mathsf{G}(\mathsf{V}, \mathsf{E}), C)$ takes a starting vertex v_i^{start}, a goal vertex v_i^{goal}, a graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ and a constraint set C (line 1). Let v be the vertex of a node explored by Constrained-A* and $\hat{h}(v)$ is the heuristic function defined in Section 2.2.1. Then, $g(v)$ is the current minimum cost of moving to v from v_i^{start} and $f(v) = g(v) + \hat{h}(v)$. We observe that if $\hat{h}(v) = 0$ for all $v \in \mathsf{V}$, the algorithm is not guided by $\hat{h}$ anymore and becomes a regular best-first search. $g(v_i^{start})$ and $f(v_i^{start})$ are initialized on lines 2 and 3. Then, the root node is created and inserted into the priority queue $OPEN$ on lines 4- 6. The main iteration loop starts on line 7 and continues until either $OPEN$ is empty (line 7), meaning no solution was found, or that the current node explored is found to be the goal node on line 9 and a solution is returned. The iteration begins by finding the node with the lowest f-value on line 8, and proceeds by calling $getSuccessors(N, C)$ on line 11. $getSuccessors(N, C)$ are all reachable nodes N' where $(N.vertex, N'.vertex) \in \mathsf{E}$ and the same node N representing waiting in place. Each successor must be valid under constraint set C. How successors are computed will be the key modification to this procedure in later sections. For each successor N', $f(N'.vertex)$ is set to ∞ if it has not been explored yet (lines 13- 14). In order to compute the cost of transitioning to a neighbour, N and N' are mapped to states and passed to the state transition function as $tranCost(state\{N.vertex\}, state\{N'.vertex\})$. If the cost of moving to $N'.vertex$ from $N.vertex$ is better than previous route (line 15), $g(N'.vertex)$ and $f(N'.vertex)$

are updated accordingly and N' is inserted into open on lines 16-18.

Algorithm 1: A high-level description of Constrained-A*, identical to A* by Hart *et al.* [9] except for how successors are computed.

```
1  function Constrained-A*(v_i^start, v_i^goal, G(V, E), C)
2  g(v_i^start) = 0;
3  f(v_i^start) = ĥ(start);
4  OPEN = ∅;
5  Root.vertex = v_i^start;
6  insert Root into OPEN;
7  while notEmpty(OPEN) do
8      N = node with lowest f(N.vertex) in OPEN;
9      if N.vertex ≡ v_i^goal then
10         return reconstructPath(N);
11     successors = getSuccessors(N, C);
12     foreach N' ∈ successors do
13         if notVisited(N') then
14             f(N'.vertex) = g(N'.vertex) = ∞;
15         if g(N'.vertex) > g(N.vertex) + tranCost(state{N.vertex}, state{N'.vertex}) then
16             g(N'.vertex) = g(N.vertex) + tranCost(state{N.vertex}, state{N'.vertex});
17             f(N'.vertex) = g(N'.vertex) + ĥ(N'.vertex);
18             insert N' into OPEN;
19 return no solution;
```

2.3.1 Validity and optimality

Hart *et al.* [8, 9, Section 2, Subsection C, Theorem 1] prove the optimality of A* to be true as long as $\hat{h}$ is an admissible heuristic. Constrained-A* operates exactly like A* except for how successors are computed. Given an empty constraint set $C = \emptyset$, Constrained-A* behaves exactly like A* and will compute $h_\emptyset(v)$ for any given $v \in \mathsf{V}$. As shown in Corollary 2.1, an admissible heuristic $\hat{h}(v)$ for Constrained-A* will then also be an admissible heuristic given a constraint set $C \neq \emptyset$. Lastly, since a successor must be legal under constraints C for a path to be consistent under constraints C, excluding successors is the only way to get a path that is both consistent and optimal.

2.4 Multi-agent solution: Conflict based search

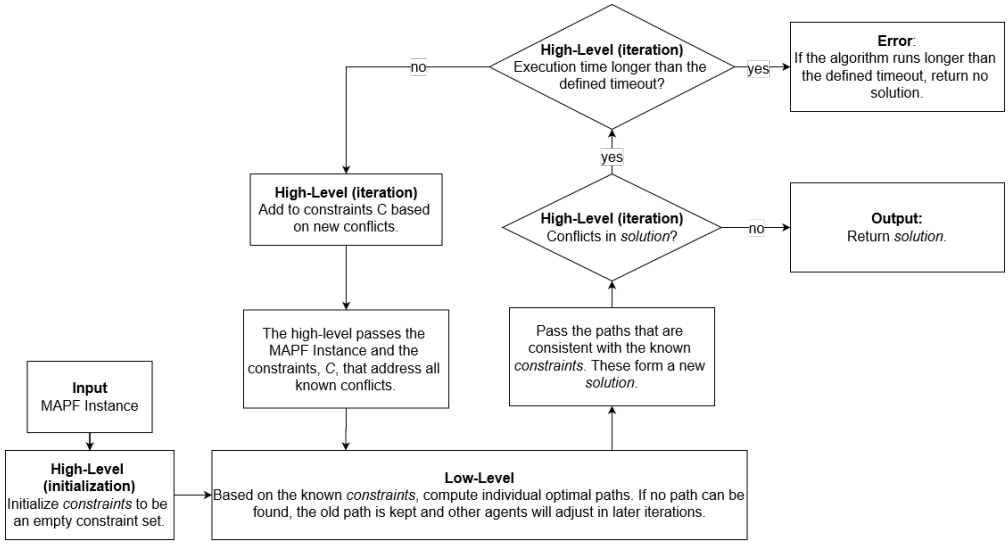


Figure 2.1: Architecture of CBS.

Sharon *et al.* [14]'s algorithm solves a MAPF problem instance and is called *conflict based search (CBS)*. CBS is divided into two levels. One level called the High-Level builds the CT by using a Dijkstra *et al.* [5] best-first search using the cost of solutions in the nodes. The other level, called the Low-Level, uses A* with constraints (Constrained-A*) to compute individual paths for all agents given the constraints in a node. The algorithm repeatedly interleaves these two levels until a conflict-free solution is found. The architecture and interaction between the two levels are illustrated in the example in Figure 2.2.

2.4.1 Low-Level

Given a set of constraints C and an individual agent a_i, the Low-Level uses Constrained-A* to compute the optimal path of a_i. The Low-Level procedure is defined as LowLevel($v_i^{start}, v_i^{goal}, \mathsf{G}(\mathsf{V}, \mathsf{E}), C$), where $\mathsf{G}(\mathsf{V}, \mathsf{E})$ is a graph, $v_i^{start} \in \mathsf{V}$ and $v_i^{goal} \in \mathsf{V}$ and C is a set of constraints. It outputs a path for a_i, that is consistent with constraints C. Note that, the procedure does not consider other agents in the $MAPF$ instance that the High-Level solves. If no solution can be found it outputs a signal to the caller.

2.4.2 High-Level

The High-Level takes parameters corresponding to a MAPF problem instance as input on line 2. These are defined as follows:

$$
\begin{aligned}
&agents : \text{set of agents } \{a_1, \ldots, a_k\} \\
&\mathsf{G}(\mathsf{V}, \mathsf{E}) : \text{a graph} \\
&start : \text{start vertices } \{v_1^{start}, \ldots, v_k^{start}\} \subseteq V \text{ of agents} \\
&end : \text{goal vertices } \{v_1^{goal}, \ldots, v_k^{goal}\} \subseteq V \text{ of agents}
\end{aligned}
$$

It then proceeds to build a binary constraint tree for the rest of the algorithm (lines 3 to 22). Each node N in the tree consists of a set of constraints $N.constraints$ and a solution $N.solution$ (consistent with $N.constraints$).

The High-Level processes nodes by adding to and popping from a processing queue called $OPEN$. $OPEN$ is initialized to $\{Root\}$ on lines 6 and 7 where $Root$ is a node initialized on lines 3 to 5 to have no constraints.

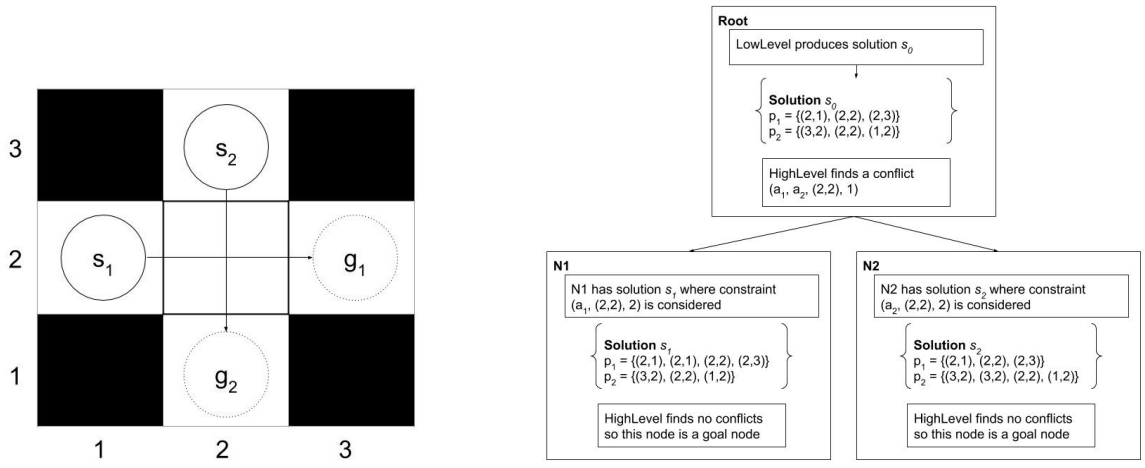
The algorithm then proceeds popping the best node from $OPEN$ on line 9, using a best-first approach of processing nodes. Nodes are ordered by their costs $cost(N.solution)$.

Processing a node N means first validating $N.solution$. If $N.solution$ is determined to be valid on line 11 by having been assigned an empty set of conflicts on line 10, it is returned on line 12. Otherwise, the first found conflict $c = (a_i, a_j, v, t)$ or $c = (a_i, a_j, (v, v'), t)$ on line 13 is considered, where v and possibly v' are vertices in the paths of the agents and t is the time at which when the conflict happened. For both agents a_i and a_j in c new nodes N_i and N_j are created in the for-loop on lines 14 to 22. An example of a solved MAPF problem instance is presented in Figure 2.2.

There are ways to improve this algorithm, some of which are discussed in Appendix A which deals with delays during execution and Appendix B which introduces a modified version of CBS.

Algorithm 2: A high-level description of conflict based search by Sharon et al. [14]

```
1  required function cost(s) returns a real number, which is the cost of s, e.g. SIC or Makespan
2  function CBS(agents, start, goal, G(V, E))
3  Root.constraints = ∅;
4  foreach agent a_k ∈ agents do
5      Root.solution[k] = LowLevel(v_k^start ∈ start, v_k^goal ∈ goal, G(V, E), Root.constraints);
6  OPEN = ∅;
7  insert Root to OPEN;
8  while OPEN not empty do
9      N = OPEN.popMin(cost);
10     conflicts = FindConflicts(N.solution);
11     if conflicts = ∅ then
12         return N.solution;
13     c = first conflict in conflicts;
14     foreach agent a_k ∈ c do
15         N' = new node;
16         if c ≡ (a_i, a_j, v, t) then
17             N'.constraints = N.constraints ∪ {(a_k, v, t)};
18         else if c ≡ (a_i, a_j, (v, v'), t) then
19             N'.constraints = N.constraints ∪ {(a_k, (v, v'), t)};
20         N'.solution[k] = LowLevel(v_k^start ∈ start, v_k^goal ∈ goal, G(V, E), N'.constraints);
21         if LowLevel found a solution then
22             insert N' to OPEN;
```



(a) An example of a MAPF problem instance and its initial non-valid solution. **(b)** Solution to the example in Figure2.2a.

Figure 2.2: Example of the Low-Level and High-Level interacting to create a solution.

2.4.3 Validity and optimality

CBS is proven optimal in [14, Section 5] and completeness of CBS is proven in [14, Section 5.2]. Their proof includes two claims: (a) That CBS will find a solution if there exists one and (b) that CBS will identify an unsolvable problem. (a) is shown as a formal proof in [14, Section 5.2.1] while (b) does not hold for CBS in the general case. They propose to use another algorithm by Yu and Rus [16] as a pre-processing step, to detect an unsolvable instance.

3 Continuous-time (CT) model

Based on the work of Andreychuk *et al.* [1] we refine the previous DT model into a continuous-time (CT) model. In this model, a point in time is represented by a real number instead of a discrete time-step.

The problem considers a set of k agents $\mathsf{A} = \{a_1, \ldots, a_k\}$ and a directed *weighted* graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$. Each agent $a_i \in \mathsf{A}$ traverse the graph by transitioning from one state st to another state st'. A state st is refined to also encode the time t (positive real number) an agent is present in location $st.v$. This lets the solver produce a plan for actions that take varying amounts of time.

Definition 3.1 (State)**.** A $state\{v \in \mathsf{V}, t \in \mathbb{R}\}$ is defined to consist of

- A vertex $state.v \in \mathsf{V}$ representing the location of the agent.
- A time $state.t \in \mathbb{R}$ representing the time when the agent is in the state.

The transition from state st to state st' has an associated cost $tranCost(st, st')$. This is used in Section 2.2 to define what the cost of a path is. This cost can now be computed with the relative time an agent is in present state st and previous state st'.

Definition 3.2 (State transition cost)**.** A *state transition cost* $tranCost(st, st') \in \mathbb{R}$ from state st to st' is defined to be $st'.t - st.t$.

Further, each agent $a_i \in \mathsf{A}$ starts at location $v_i^{start} \in \mathsf{V}$ and it needs to traverse the graph to reach location $v_i^{goal} \in \mathsf{V}$, via a series of *actions*. Since actions take place during transitions between states, the previous definition of an action is refined to also consist of a time interval.

Definition 3.3 (Action)**.** An *action* taken during interval $[action.t_{start}, action.t_{end}]$ is defined to be either:

- STOP$(st, [st.t, t^{ub}])$: Remain in the same state st during time interval $[action.t_{start}, action.t_{end}] = [st.t, t^{ub}]$, where t^{ub} is the time the next action takes place from $st.v$.
- MOVE$(st, st', [st.t, st'.t])$: Move to a new state st' from st, such that $(st.v, st'.v) \in \mathsf{E}$ during time interval $[action.t_{start}, action.t_{end}] = [st.t, st'.t]$.

In other words, agent a_i moves from v_i^{start} to v_i^{goal} along the path $p_i = (state\{v_i^{start}, 0\}, \ldots, st, st', \ldots, state\{v_i^{goal}, t_{goal}\})$, where the action that a_i takes in state st is MOVE$(st, st', [st.t, st'.t])$ when $st.v \neq st'.v$ and STOP$(st, [st.t, t^{ub}])$ otherwise. Time is now directly correlated to the time parameter $st.t$ of a state st in the path.

Definition 3.4 (Time)**.** The *timed representation* of a path p_i is $p_i = (st_1, \ldots, st_\ell)$, where $st_j.v$ is the location of agent a_i at time $st_j.t$ and ℓ is the path length.

We say, in this paper, that $s = \{p_1, p_2, ..., p_k\}$ is a solution, consisting of the individual paths of *all agents*, to the problem of *Multi-Agent Path-Finding* (MAPF) if it satisfies the safety condition of no *conflicts*. To the end of finding a conflict-free solution considering continuous time, Andreychuk *et al.* [1] extends the notions proposed by Sharon *et al.* [14]. This model consider agents moving continuously, and actions made by agents can conflict during any interval of time. For example, an agent can collide in the middle of traversing an edge. Therefore, conflicts and corresponding constraints are now defined with respect to actions instead of states.

Definition 3.5 (Conflict)**.** A continuous *conflict* which would result in a collision between two agents a_i and a_j taking actions $action_i$ and $action_j$ is defined as $(action_i, action_j)$.

Given a solution s that has a conflict, one can refine s by allowing a_i to keep its current plan while forbidding the a_j from taking $action_j$ during the *unsafe interval* of time $[action_j.t_{start}, t) \subset [action_j.t_{start}, action_j.t_{end}]$ when it is not safe to perform $action_j$. Each node explored by the solver refines the solution by adding constraints.

Definition 3.6 (Constraint)**.** A *constraint* for an agent a_i is defined as: $(a_i, action_i, t)$, where agent a_i is forbidden from executing action $action_i$ during time interval $[action_i.t_{start}, t)$. $[action_i.t_{start}, t)$ is the interval of time that if a_i executes $action_i$ during this interval, a_i will collide with another agent.

For example, consider the conflict in Figure 1.3, the length of $[action_i.t_{start}, t)$ would be the same as the length of the collision interval. The upper bound t in the unsafe interval $[action_i.t_{start}, t)$ can be computed using any collision-interval-detection algorithm. Andreychunk *et al.* uses an algorithm by Guy *et al.* [6] in their experiments, which detects collisions between disk-shaped agents.

Note that both $(a_i, action_i, t')$ and $(a_j, action_j, t'')$ are explored, *i.e.,* one constraint for each agent in a conflict $(action_i, action_j)$. As in the DT model, starting from the root node, which includes no constraints, each node encodes a different list of constraints that refines the solution. That is, the list of any non-root node is the result of appending a constraint to the list of its parent.

A path $p_i \in s$ is called *consistent* if it satisfies all constraints for agent a_i and a solution is called consistent if all single-agent paths in the solution are consistent. The solution in a CT node is always consistent with its constraint list and a CT node is called a *goal node* when the solution has no conflicts. Therefore, we say that goal nodes provide *valid* solutions.

A multi-agent objective function $cost(s)$ and an individual objective function $h(v)$ for a vertex $v \in \mathsf{V}$ are defined in Section 2.2.

3.1 Single-agent path-finding: Safe interval path planning

Safe Interval Path Planning (SIPP) is an algorithm for solving a continuous path-finding problem with dynamic obstacles for a *single agent*. SIPP was developed by Phillips *et al.* [13], and works like A* but considers continuous-time and obstacle collision.

To avoid collisions, SIPP uses safe intervals for each vertex in the graph. A safe interval can be looked up during path-finding and gives the solver information about when or if an action is possible or not in the graph.

For example, consider Figure 3.1, where agent a_1 is trying to get from cell $(4, 2)$ to $(1, 2)$. a_2 is an obstacle that is moving from $(3, 1)$ to $(3, 4)$. During an interval of time $[t_{enter}, t_{exit}]$, a_2 will occupy $(3, 2)$. If a_1

enters cell $(3, 2)$ at any point between t_{enter} and t_{exit}, it runs the risk of colliding with a_2. To counter this, we include the intervals $[0, t_{enter})$ and $[t_{exit}, \infty)$ as safe intervals for cell $(3, 2)$. This means that during path-finding, SIPP will know that it cannot let a_2 enter $(3, 2)$ during $[t_{enter}, t_{exit})$.

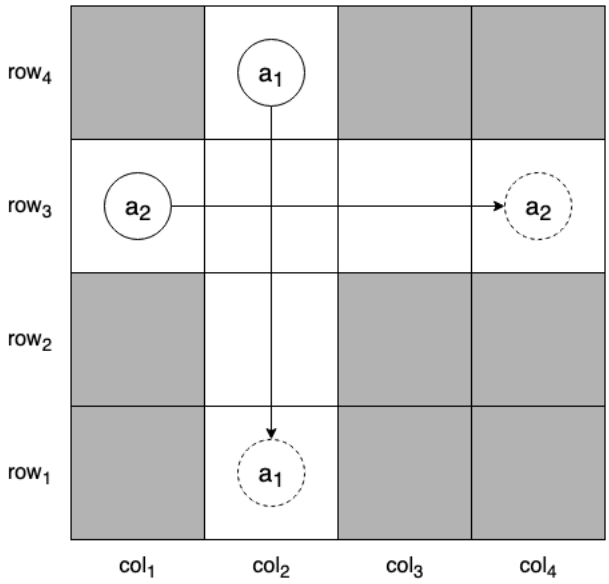


Figure 3.1: An example of a path-finding problem on a 4x4 grid.

With that introductory example in mind, we define the path-finding algorithm and safe intervals. Let $[t_i^{lb}, t_i^{ub})_v$ be a safe interval for vertex v where t_i^{lb} is the lower bound and t_i^{ub} is the upper bound on when it is safe to be positioned in v. Given a vertex $v \in \mathsf{V}$, $safeIntervals(v)$ is a lookup-function that returns a set $\{[t_1^{lb}, t_1^{ub})_v, \ldots, [t_n^{lb}, t_n^{ub})_v\}$ of n safe intervals for v. The lower and upper bound of an interval i are also referred to as $i.start$ and $i.end$.

Recall from Section 2.2.1 the definition of an admissible heuristic $\hat{h}(v)$. SIPP explores the search space in a best-first fashion, guided by the estimate $\hat{h}(v)$ of the true continuous-time $h(v)$ it will take to reach the *goal* vertex. This can, for example, be the time it would take to travel the euclidean distance from v to *goal*. Just like A*, SIPP keeps track of $g(v)$ as well. $g(v)$ is the current best continuous time at which v can be reached from the *start* vertex. Combining these is a mapping $f(v) = g(v) + \hat{h}(v)$, which is used to sort the priority queue containing nodes to be explored in the search space.

The pseudo-code for the algorithm can be seen in Algorithm 3 and is close to identical to Algorithm 1 with a few differences. Lines 2-7 initialize the $OPEN$ priority queue with the *Root* node. Then, as long as we either have more nodes to explore (line 8) and the goal node have not been reached (line 10), we iterate, popping nodes of of $OPEN$ based on $f(N.vertex)$ (line 9). On lines 13-19, each successor N' is checked and if it is determined to create a path of lower cost, $f(N'.vertex)$ and $g(N'.vertex)$ are updated and the successor N' is added to $OPEN$. Note that, just like in Constrained-A* N and N' are mapped into states and passed as $tranCost(state\{N.vertex, N.t\}, state\{N'.vertex, N'.t\})$. Difference being that $tranCost$ is defined differently in the continuous-time model.

The part that makes SIPP different is how successors are computed for a vertex (line 12). Instead of just considering what vertices are connected to the current vertex, SIPP also checks if the arrival time falls within a safe interval. This procedure can be seen in Algorithm 4. The $safeIntervals(v)$ discussed above can be seen as "required" on line 1. Initially, *successors* is set to be empty on line 4. Then, we iterate over each possible maneuver m that can be performed from N (line 5). A maneuver is another vertex $m.vertex$, such that edge $(N.vertex, m.vertex) \in \mathsf{E}$ which has an edge cost of $m.timeToExecute$. An $earliestArrivalTime$ and $latestArrivalTime$ are computed on lines 6 and 7 respectively. $earliestArrivalTime$ is the earliest time at which the agent can arrive at $m.vertex$ and $latestArrivalTime$ is the latest time at which the agent must

leave $N.vertex$ (as to not violate the safe interval $N.interval$). Then, for each safe interval i (line 8) in the neighbouring vertex $m.vertex$ we check to see if we can perform the maneuver during i on line 9. Now, if an arrival time t is found on line 10, a new successor is initialized on lines 12 to 16.

Algorithm 3: A high-level description of safe interval path planning by Philips *et al.* [13]

function SIPP($v_i^{start}, v_i^{goal}, \mathsf{G}(\mathsf{V}, \mathsf{E})$)
$g(v_i^{start}) = 0$;
$f(v_i^{start}) = \hat{h}(v_i^{start})$;
$OPEN = \emptyset$;
$Root.vertex = v_i^{start}$;
$Root.t = 0$;
insert *Root* into $OPEN$;
while $notEmpty(OPEN)$ **do**
  N = node with lowest $f(N.vertex)$ in $OPEN$;
  if $N.vertex \equiv v_i^{goal}$ **then**
    return $reconstructPath(N)$;
  $successors = getSuccessors(N)$;
  foreach $N' \in successors$ **do**
    if $notVisited(N')$ **then**
      $f(N'.vertex) = g(N'.vertex) = \infty$;
    if $g(N'.vertex) > g(N.vertex) + tranCost(state\{N.vertex, N.t\}, state\{N'.vertex, N'.t\})$ **then**
      $g(N'.vertex) = g(N.vertex) + tranCost(state\{N.vertex, N.t\}, state\{N'.vertex, N'.t\})$;
      $f(N'.vertex) = g(N'.vertex) + \hat{h}(N'.vertex)$;
      insert N' into $OPEN$;
return *no solution*;

Algorithm 4: Function that returns possible and safe maneuvers to take from N

required function $safeIntervals(v)$ returns a set of intervals during which it is safe to occupy vertex v.
required function $possibleManeuevers(N)$ returns a set of maneuvers that can be performed starting from $N.vertex$.
function $getSuccessors(N)$
$successors = \emptyset$;
foreach $m \in possibleManeuevers(N)$ **do**
  /* A manuever m from a node N is defined as */
  /* $m.vertex \in \mathsf{V}$ and $(N.vertex, m.vertex) \in \mathsf{E}$ */
  /* $m.timeToExecute \in \mathbb{R}$ */
  $earliestArrivalTime = N.arrivalTime + m.timeToExecute$;
  $latestArrivalTime = N.interval.end + m.timeToExecute$;
  foreach $i \in safeIntervals(m.vertex)$ **do**
    if $i.start \leq latestArrivalTime$ *and* $earliestArrivalTime \leq i.end$ **then**
      t = earliest arrival time at $m.vertex$ during interval i;
      if t *exists* **then**
        N' = new node;
        $N'.vertex = m.vertex$;
        $N'.arrivalTime = t$;
        $N'.interval = [t, i.end]$;
        insert N' into $successors$;
return $successors$;

3.1.1 Validity and optimality

Phillips *et al.* prove that SIPP is complete in [13, Section 3, Subsection B, Theorem 1]. The theorem states that arriving in a state at the earliest time yields the largest set of successors. Using this, they prove that SIPP is optimal.

3.2 Multi-agent solution: Continuous conflict based search

Andreychuk *et al.* [1] solves a MAPF problem instance with a modified version of CBS called *Continuous conflict based search (CCBS)*. They use SIPP as a Low-Level solver instead of Constrained-A*. To illustrate how their algorithm works, we reiterate the example in Figure 3.1 used to explain SIPP.

As explained in the SIPP example, a_1 and a_2 would collide if they start to move at the same time. Let $t = 1$ second be the time that it takes to move from one cell to the next for each agent. Both moving at the same time would lead CCBS to detect a conflict $(action_i, action_j)$ where $action_i = \text{MOVE}(\{v_{(4,2)}, 0\}, \{v_{(3,2)}, 1\}, [0, 1])$ and $action_j = \text{MOVE}(\{v_{(3,1)}, 0\}, \{v_{(3,2)}, 1\}, [0, 1])$. This conflict can, for example, be resolved by first computing the collision interval between the agents. Then, this interval is used to create two unsafe intervals $[action_i.t_{start}, t)$ and $[action_j.t_{start}, t)$, where $action_i.t_{start} = action_j.t_{start} = 0$ and t is the upper-bound derived from the collision interval such that $0 \leq t \leq 1$. These unsafe intervals are used to create two constraints: $(a_i, action_i, t)$ and $(a_j, action_j, t)$, which are both explored in the constraint tree. Next we explain the concepts in this example in more detail.

To detect conflicts, CCBS uses a geometry-aware collision detection mechanism. They use an algorithm by Guy *et al.* [6] in their experiments. CCBS applies the collision detection several times with a given $\Delta > 0$ to compute a collision interval from the time point when the collision was first detected, and ends when no collision is detected anymore. This collision interval is then used to derive the upper bound t in an unsafe interval $[action.t_{start}, t)$.

Let $(a_i, action_i, t)$ be a constraint and $[action_i.t_{start}, t)$ its corresponding unsafe interval. They resolve the constraint differently if it the action part of the constraint is MOVE or STOP.

If $action_i$ is $\text{STOP}(st, [st.t, t^{ub}])$ they they forbid a_i from waiting at $st.v$ during interval $[st.t, t)$ by removing this interval from the safe intervals associated with $st.v$. For example, if $st.v$ is associated with a single safe interval $[0, \infty)$, this interval is split into two safe intervals $[0, st.t]$ and $[t, \infty)$.

They modify SIPP so that it will never schedule a MOVE action for agent a_i such that it is executed during $[action_i.t_{start}, t)$. Let v be the source vertex and v' the target vertex, instead of scheduling the MOVE action between v and v' during $[action_i.t_{start}, t)$, they add a STOP action, which has a_i waiting in v until t and then move to v'.

Given the current constraint set and the procedures explained above, CCBS uses SIPP to compute the optimal single-agent paths. In the paper made by Andreychuk *et al.* [1] there is no pseudo-code supplied. Algorithm 5 attempts to illustrate the main procedure by highlighting the lines changed in Algorithm 2. The MAPF input instance is defined the same as for CBS but under the new definitions in the CT model.

Algorithm 5: CCBS by Andreychuk *et al.* [1].

required function $cost(s)$ returns a real number, which is the cost of s, e.g. *SIC* or *Makespan*

function CCBS($agents, start, goal, G(V, E)$)
$Root.constraints = \emptyset$;
foreach *agent* $a_k \in agents$ **do**
    Create safe intervals using $Root.constraints$;
    $Root.solution[k] = \text{SIPP}(v_k^{start} \in start, v_k^{goal} \in goal, G(V, E), Root.constraints)$;
$OPEN = \emptyset$;
insert $Root$ to $OPEN$;
while $OPEN$ *not empty* **do**
    $N = OPEN.\text{popMin}(cost)$;
    $conflicts = \text{FindConflicts}(N.solution)$;
    if $conflicts = \emptyset$ **then**
        return $N.solution$;
    $C =$ first conflict $(action_i, action_j)$ in $conflicts$;
    foreach *agent* $a_k \in C$ **do**
        $N' =$ new node;
        Compute $[action_k.t_{start}, t)$ for $action_k$ w.r.t. the other action;
        $N'.constraints = N.constraints \cup \{(a_k, action_k, t)\}$;
        Create safe intervals using $N'.constraints$;
        $N'.solution[k] = \text{SIPP}(v_k^{start} \in start, v_k^{goal} \in goal, G(V, E), N'.constraints)$;
        if *SIPP found a solution* **then**
            insert N' to $OPEN$;

3.2.1 Validity and optimality

Andreychuk *et al.* [1, Section 3.2, Theorem 1] prove optimality and completeness for CCBS using the notion of a sound pair of constraints defined by Atzmon *et al.* [3]. In [1, Section 3.2, Lemma 2] they provide a lemma proving that the pair of continuous constraints created by CCBS is a sound pair of constraints. With this, they prove both optimality and completeness of CCBS.

4 Continuous-time with discrete speeds (CTDS)

This model is one of the main contributions of this project. The previous models in Sections 2 and 3 made an important assumption that agents can always stop and come to full speed instantly. Constrained-A* always has the option to add a WAIT action, unless a constraint forbids it. In the case of SIPP, Phillips *et al.* specifically states that "Inertial constraints (acceleration/deceleration) are negligible. The planner assumes the robot can stop and accelerate instantaneously." [13]. This model is an attempt to relax this assumption and will consider agents moving at many speeds and accelerating between them.

The problem considers a set of k agents $\mathsf{A} = \{a_1, \ldots, a_k\}$ and a directed graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$. Each $v \in \mathsf{V}$ encodes a position $pos = (x_1, \ldots, x_p)$ of p dimensions referred as $v.pos$ and a set of $v.speeds = \{speed_{min}, \ldots, speed_{max}\}$. This gives the solver information it can use to determine the distance between vertices and if a speed change is possible within a certain distance for example. Each agent $a_i \in \mathsf{A}$ traverse the graph by transitioning from one state st to another state st'. A state st is further refined to encode the speed $st.speed$ the agent was traveling at location $st.v$ at time $st.t$.

Definition 4.1 (State)**.** A $state\{v \in \mathsf{V}, t \in \mathbb{R}, speed \in v.speeds\}$ is defined to consist of

- A vertex $state.v \in \mathsf{V}$ representing the position of the agent.
- A time $state.t \in \mathbb{R}$ representing the time when the agent is in the state.
- A time $state.speed \in speeds_{st.v}$ representing the speed the agent is traveling by.

The transition from state st to state st' has an associated cost $tranCost(st, st')$. This is used in Section 2.2 to define what the cost of a path is. This cost is the relative time between states based on the speed in the first and second state.

Definition 4.2 (State transition cost)**.** A *state transition cost* $tranCost(st, st') \in \mathbb{R}$ from state st to st' is defined to be $st'.t - st.t$.

The start and goal location of an agent is refined to also include a start and goal speed. That is, each agent $a_i \in \mathsf{A}$ starts at location $v_i^{start} \in \mathsf{V}$ with speed $speed_i^{start} \in v_i^{start}.speeds$ and it needs to traverse the graph to reach location $v_i^{goal} \in \mathsf{V}$ at speed $speed_i^{goal} \in v_i^{goal}.speeds$, via a series of *actions*. Keeping speed or changing speed is signified by adding a source and target speed for all MOVE actions.

Definition 4.3 (Action)**.** An *action* taken during interval $[action.t_{start}, action.t_{end}]$ is defined to be either:

- STOP$(st, [st.t, t])$: Remain in the same state st during time interval $[action.t_{start}, action.t_{end}] = [st.t, t]$, where t is the time the next action takes place from $st.v$. This action implies that the speed of the agent is 0 and is only possible if $speed_{min} = 0 \in st.v.speeds$.
- MOVE$(st, st.speed, st', st'.speed, [st.t, st'.t])$: Move to a new state st' from st, such that $(st.v, st'.v) \in \mathsf{E}$ during time interval $[action.t_{start}, action.t_{end}] = [st.t, st'.t]$. If $st.speed \neq st'.speed$, then this implies that the agent changes its speed between locations $st.v$ and $st'.v$. Otherwise if $st.speed = st'.speed$, then the agent keeps its speed when moving from $st.v$ to $st.v'$.

In other words, agent a_i starts at speed $speed_i^{start}$ from v_i^{start} and ends up in speed $speed_i^{goal}$ at v_i^{goal}, along the path $p_i = (state\{v_i^{start}, 0, speed_i^{start}\}, \ldots, st, st', \ldots, state\{v_i^{goal}, t_{goal}, speed_i^{goal}\})$. An action that a_i takes in state st is MOVE$(st, st', [st.t, st'.t])$ when $st.v \neq st'.v$ and STOP$(st, [st.t, t])$ otherwise. Time is as before directly correlated to the time parameter $st.t$ of a state st in the path.

Definition 4.4 (Time)**.** The *timed representation* of a path p_i is $p_i = (st_1, \ldots, st_\ell)$, where $st_j.v$ is the location and $st_j.speed$ is the speed of agent a_i at time $st_j.t$ and ℓ is the path length.

We say, in this paper, that $s = \{p_1, p_2, ..., p_k\}$ is a solution, consisting of the individual paths of *all agents*, to the problem of *Multi-Agent Path-Finding* (MAPF) if it satisfies the safety condition of no *conflicts*. In the CTDS model, collisions are allowed to occur at different speeds. However, conflicts and constraints can be defined in the same way as the CT model. The only difference is how the unsafe interval $[t, t')$ is computed, which now has to be computed while taking different speeds into account.

Definition 4.5 (Conflict)**.** A continuous *conflict* which would result in a collision between two agents a_i and a_j taking actions $action_i$ and $action_j$ is defined as $(action_i, action_j)$.

Given a solution s that has a conflict, one can refine s by allowing a_i to keep its current plan while forbidding the a_j from taking $action_j$ during the *unsafe interval* of time $[action_j.t_{start}, t) \subset [action_j.t_{start}, action_j.t_{end}]$ when it is not safe to perform $action_j$. Each node explored by the solver refines the solution by adding con-

straints.

Definition 4.6 (Constraint)**.** A *constraint* for an agent a_i is defined as: $(a_i, action_i, t)$, where agent a_i is forbidden from executing action $action_i$ during time interval $[action_i.t_{start}, t)$. $[action_i.t_{start}, t)$ is the interval of time that if a_i executes $action_i$ during this interval, a_i will collide with another agent.

For example, consider the conflict in Figure 1.3, the length of $[action_i.t_{start}, t)$ would be the same as the length of the collision interval. The upper bound t in the unsafe interval $[action_i.t_{start}, t)$ can be computed using any collision-interval-detection algorithm. Andreychunk *et al.* uses an algorithm by Guy *et al.* [6] in their experiments, which detects collisions between disk-shaped agents.

Note that both $(a_i, action_i, t')$ and $(a_j, action_j, t'')$ are explored, *i.e.,* one constraint for each agent in a conflict $(action_i, action_j)$. As in the DT model, starting from the root node, which includes no constraints, each node encodes a different list of constraints that refines the solution. That is, the list of any non-root node is the result of appending a constraint to the list of its parent.

A path $p_i \in s$ is called *consistent* if it satisfies all constraints for agent a_i and a solution is called consistent if all single-agent paths in the solution are consistent. The solution in a CT node is always consistent with its constraint list and a CT node is called a *goal node* when the solution has no conflicts. Therefore, we say that goal nodes provide *valid* solutions.

A multi-agent objective function $cost(s)$ and an individual objective function $h(v)$ for a vertex $v \in \mathsf{V}$ are defined in Section 2.2.

4.1 Single-agent solution: Discrete-speed SIPP

Discrete-speed SIPP (DS-SIPP) is a modification of SIPP that also computes the optimal path for one agent. The main contribution to SIPP is to relax the assumption that agents can start and stop instantly. DS-SIPP considers a set of one or more $speeds = \{0, \ldots, speed_{max}\}$ that agents can travel by. This lets an agent slow down to avoid a collision instead of stopping. The hope is to gain performance at the target system, where stopping is more expensive than slowing down. Moreover, DS-SIPP can encode that an action is possible or not.

DS-SIPP uses a function $actionTime(v, speed, v_{next}, speed_{next})$ to compute the time of an action. Assuming constant acceleration, this function returns the time it takes to move from a vertex v at $speed$ to another vertex v_{next} ending up in $speed_{next}$. For example, if we let $t = actionTime(v, speed, v_{next}, speed_{next})$, then t can be described by the kinematic equation:

$$\begin{aligned} distance(v, v_{next}) &= t * (speed + speed_{next})/2 \\ t &= 2 * distance(v, v_{next})/(speed_{next} + speed) \end{aligned}$$

A function *actionPossible* determines if an action is possible w.r.t. movement and speed. Given a previous vertex v_{prev}, a current vertex v and $speed$, $actionPossible(v_{prev}, v, speed, v_{next}, speed_{next})$ returns true if v_{next} can be reached at $speed_{next}$ and false otherwise. This function allows DS-SIPP to know more about constraints in the target system. For example, consider a U-turn in an intersection modeled as (v_1, v_2) and (v_2, v_3), where v_1 is the entry point, $v2$ is the "peak" of the turn and v_3 the exit. A vehicle would have to keep its speed low enough to perform the maneuver. Then, *actionPossible* can encode this by returning false for combinations of $speed$ and $speed_3$ that are too high.

The computation of safe intervals are extended to two functions: $safeIntervalsVertex(v)$ and $safeIntervalsEdge(v, v')$. $safeIntervalsVertex(v)$ is identical to $safeIntervals(v)$ as described in Section 3.1. $safeIntervalsEdge(v, v')$ is similar, but returns the safe continuous time intervals for which it is safe to move from vertex v to v'. A safe interval for and edge (v, v') is identical to the constraint of a MOVE action. It is formally described as $[t_i^{lb}, t_i^{ub})_{(v,v')}$ where t_i^{lb} is the lower bound and t_i^{ub} is the upper bound on when it is safe to move from v to v'. Then, $safeIntervalsEdge(v)$ is a lookup-function that, given an edge $(v, v') \in \mathsf{E}$, returns a set $\{[t_1^{lb}, t_1^{ub})_{(v,v')}, \ldots, [t_n^{lb}, t_n^{ub})_{(v,v')}\}$ of n safe intervals for (v, v').

The main procedure of DS-SIPP operates exactly like SIPP seen in Algorithm 3, but with the explored nodes storing more information:

- Nodes store a speed as $N.speed$.

- Nodes store its parent node as $N.parent$.
- Check both if in the goal vertex and speed on line 9.
- On line 15 the mapping of N and N' into states is updated according to the new definition.

The key difference is the *getSuccessors* function. *getSuccessors* for DS-SIPP is presented in Algorithm 7. DS-SIPP iterates over manuevers m created by $possibleManuevers(N)$ on line 6. Let $(N.vertex, v) \in \mathsf{E}$ and $speed \in speeds_v$, then a manuever m is possible if $actionPossible(N.parent.vertex, N.vertex, N.speed, v, speed)$ returns true. $m.timeToExecute$ is computed using $actionTime(N.vertex, N.speed, v, speed)$.

For each maneuver m iterated over on line 6, the earliest and latest departure and arrival times are computed on lines 7 to 13. If speed is 0 (line 7), the earliest time to depart for the agent is $N.arrivalTime$. The latest time to depart is $N.interval.end$, since an agent cannot stay in $N.vertex$ past the end of the safe interval in $N.vertex$. If speed is larger than 0 (line 10), the agent has a single departure time $N.arrivalTime$ since it is not allowed to wait in the vertex due to currently being in motion. The earliest time to arrive and the latest time to arrive are both defined in terms of the respective departure times on lines 12 and 13.

Then, we iterate over each safe interval i for $m.vertex$ (line 14) and each j for edge $(N.vertex, m.vertex)$ (line 15). We check if the intersection between intervals i, j and $[earliestArrivalTime, latestArrivalTime]$ and the intersection between j and $[earliestDepartureTime, latestDepartureTime]$ exists (line 18), meaning that we can move and arrive without collision from $N.vertex$ to $m.vertex$. The intersection $x = a \cap b$ of two intervals a and b is computed as $x = [max(a.start, b.start), min(a.end, b.end)]$ such that $x.start \in a$, $x.start \in b$, $x.end \in a$ and $x.end \in b$, otherwise x does not exist. Then, we compute the two possible earliest arrival times $arrivalTime_1$ and $arrivalTime_2$ on lines 19 and 20. This is done such that we depart from $N.vertex$ during $safeDepartureI$ and arrive during $safeArrivalI$ in $m.vertex$.

If either $arrivalTime_1$ (line 21) or $arrivalTime_2$ (line 23) exist, a successor is created in identical fashion to SIPP on lines 28- 32, with the exception that the speed $m.speed$ is stored on line 32 and parent N on line 33.

Algorithm 6: A high-level description of discrete-speed safe interval path planning.

```
1  function DS-SIPP(v_i^start, speed_i^start, v_i^goal, speed_i^goal, G(V, E))
2  g(v_i^start) = 0;
3  f(v_i^start) = ĥ(v_i^start);
4  OPEN = ∅;
5  Create Root node for vertex v_i^start and speed speed_i^start;
6  insert Root into OPEN;
7  while notEmpty(OPEN) do
8      N = node with lowest f(N.vertex) in OPEN;
9      if N.vertex ≡ v_i^goal and N.speed ≡ speed_i^goal then
10         return reconstructPath(N);
11     successors = getSuccessors(N);
12     foreach N' ∈ successors do
13         if notVisited(N') then
14             f(N'.vertex) = g(N'.vertex) = ∞;
15         cost = tranCost(state{N.vertex, N.t, N.speed}, state{N'.vertex, N'.t, N'.speed});
16         if g(N'.vertex) > g(N.vertex) + cost then
17             g(N'.vertex) = g(N.vertex) + cost;
18             f(N'.vertex) = g(N'.vertex) + ĥ(N'.vertex);
19             insert N' into OPEN;
20 return no solution;
```

Algorithm 7: Function that returns possible and safe maneuvers to take from N

required function $safeIntervalsVertex(v)$ returns a set of intervals during which it is safe to occupy vertex v.
required function $safeIntervalsEdge(v, v')$ returns a set of intervals during which it is safe to occupy edge (v, v').
required function $possibleManeuevers(N)$ returns a set of maneuvers that can be performed starting from $N.vertex$ at speed $N.speed$.
function $getSuccessors(N)$
$successors = \emptyset$;
foreach $m \in possibleManeuevers(N)$ **do**
    /* A manuever m from a node N is defined as */
    /* $m.vertex \in \mathsf{V}$ and $(N.vertex, m.vertex) \in \mathsf{E}$. */
    /* $m.speed \in \{0, \ldots, speed_{max}\} \subseteq speeds_{m.vertex}$. */
    /* $m.timeToExecute$ is the time it takes to perform m. */
    if $N.speed \equiv 0$ **then**
        /* If speed is 0, move can be executed during interval. */
        $earliestDepartureTime = N.arrivalTime$;
        $latestDepartureTime = N.interval.end$;
    else
        /* If speed is not 0, move can only be executed at time $N.arrivalTime$. */
        $earliestDepartureTime = latestDepartureTime = N.arrivalTime$;
    $earliestArrivalTime = earliestDepartureTime + m.timeToExecute$;
    $latestArrivalTime = latestDepartureTime + m.timeToExecute$;
    foreach $i \in safeIntervalsVertex(m.vertex)$ **do**
        foreach $j \in safeIntervalsEdge(N.vertex, m.vertex)$ **do**
            $safeArrivalI = [earliestArrivalTime, latestArrivalTime] \cap i \cap j$;
            $safeDepartureI = [earliestDepartureTime, latestDepartureTime] \cap j$;
            if *$safeArrivalI$ exists and $safeDepartureI$ exists* **then**
                $arrivalTime_1 = safeDepartureI.start + m.timeToExecute$;
                $arrivalTime_2 = safeArrivalI.start$;
                if $arrivalTime_1 \in safeArrivalI$ **then**
                    $arrivalTime = arrivalTime_1$;
                else if $arrivalTime_2 - m.timeToExecute \in safeDepartureI$ **then**
                    $arrivalTime = arrivalTime_2$;
                else
                    $arrivalTime$ does not exist;
                if *$arrivalTime$ exists* **then**
                    $N' =$ new node;
                    $N'.vertex = m.vertex$;
                    $N'.arrivalTime = arrivalTime$;
                    $N'.interval = [arrivalTime, i.end]$;
                    $N'.speed = m.speed$;
                    $N'.parent = N$;
                    insert N' into $successors$;
return $successors$;

4.1.1 Proof of validity and optimality

To prove optimality and completeness of DS-SIPP, we must first show that $getSuccessors(N)$ called on line 11 returns all valid successors with the earliest arrival time. We do this by showing that, for each safe vertex and edge interval, we pick the earliest arrival time if and only if it exists. This will prove Theorem 1 [13, Section 3.B, Theorem 1] by Phillips *et al.* for DS-SIPP, leading to completeness. Theorem 2 [13, Section 3.B, Theorem 2] by Phillips *et al.* will then be true for DS-SIPP as well. This proves both optimality and completeness of DS-SIPP.

Proof:

First of all, if the agent cannot arrive in the safe interval $safeArrivalI$ (checked on line 18) there is no possible time to safely arrive in $m.vertex$. Likewise, if the agent cannot depart during $safeDepartureI$ (checked on line 18) there is no possible time to safely arrive in $m.vertex$ either.

Now, given that $safeArrivalI$ and $safeDepartureI$ are determined to exist, we show that the earliest arrival time is either $arrivalTime_1 = safeDepartureI.start + m.timeToExecute$ on line 19 or $arrivalTime_2 = safeArrivalI.start$ on line 20, otherwise no arrival time exists.

Let $moveInterval_1 = [safeDepartureI.start, safeDepartureI.start + m.timeToExecute]$ and $moveInterval_2 = [safeArrivalI.start - m.timeToExecute, safeArrivalI.start]$. These are the two intervals of time that moves the agent from $N.vertex$ and has it arrive in $m.vertex$ in $arrivalTime_1$ or $arrivalTime_2$ respectively. We now prove that these are the only two possible intervals of time that has the agent both arriving safely and as early as possible. This is done by proof of contradiction. We assume that another interval $moveInterval_3 = [t_{departure}, t_{arrival}]$, where $t_{arrival} - t_{departure} = m.timeToExecute$, $t_{departure} \in safeDepartureI$ and $t_{arrival} \in safeArrivalI$. This interval is both valid and has the agent arriving earlier at $m.vertex$ than either $moveInterval_1$ and $moveInterval_2$, whether they are valid or not.

In the cases where $moveInterval_1$ or $moveInterval_2$ are valid, it must be true that $moveInterval_3$ has the agent departing earlier than $moveInterval_1.start$ or $moveInterval_2.start$. This leads to a contradiction in assuming that $t_{departure} \in safeDepartureI$ and $t_{arrival} \in safeArrivalI$, since either $t_{departure} < safeDepartureI.start$ due to $moveInterval_1.start = safeDepartureI.start$ or $t_{arrival} < safeArrivalI.start$ due to $moveInterval_2.start = safeArrivalI.start - m.timeToExecute$.

In the case where both $moveInterval_1$ and $moveInterval_2$ are invalid, meaning that for $moveInterval_1$ we have that either

$$safeDepartureI.start + m.timeToExecute > safeArrivalI.end \tag{4.1}$$

or

$$safeDepartureI.start + m.timeToExecute < safeArrivalI.start \tag{4.2}$$

and for $moveInterval_2$ we have that either

$$safeArrivalI.start - m.timeToExecute > safeDepartureI.end \tag{4.3}$$

or

$$safeArrivalI.start - m.timeToExecute < safeDepartureI.start \tag{4.4}$$

In cases 4.1 and 4.3 it is impossible to construct $moveInterval_3$, since any $t_{departure} > safeDepartureI.end$ and $t_{arrival} > safeArrivalI.end$. In case 4.2 it must be that $t_{departure} = safeArrivalI.start - m.timeToExecute$ since this is the earliest time larger than $safeDepartureI.start + m.timeToExecute$ that has the agent arriving in $safeArrivalI$. However, this contradicts the fact that $moveInterval_2$ is invalid. Case 4.4 is analogous for $t_{arrival} = safeDepartureI.start + m.timeToExecute$ and the earliest departure time being $safeDepartureI.start$, which contradicts the fact that $moveInterval_1$ is invalid.

■

4.2 Multi-agent solution: Continuous conflict based search

We propose to use Continuous conflict based search (CCBS) described in Section 3.2 with DS-SIPP as the Low-Level to solve a MAPF problem instance. This will produce a solution that has planned for agents both

moving in continuous time and at one or more speeds. We also refine the MAPF input to pass speeds in the *start* and *goal* sets of agents.

$agents$: set of agents $\{a_1, \ldots, a_k\}$
$G(V, E)$: an undirected graph
$start$: start vertices and speeds$\{(v_1^{start}, speed_1^{start}) \ldots, (v_k^{start}, speed_k^{goal})\}$ of agents
end : goal vertices and speeds$\{(v_1^{goal}, speed_1^{goal}), \ldots, (v_k^{goal}, speed_k^{goal})\} \subseteq V$ of agents

Algorithm 8: Highlighted usage of DS-SIPP instead of SIPP in CCBS.

required function $cost(s)$ returns a real number, which is the cost of s, e.g. *SIC* or *Makespan*

function CCBS($agents, start, goal, G(V, E)$)
  $Root.constraints = \emptyset$;
  foreach *agent* $a_k \in agents$ **do**
    Create safe intervals using $Root.constraints$;
    $Root.solution[k]$ = DS-SIPP($a_k, v_k^{start}, speed_k^{start}, v_k^{goal}, speed_k^{goal}, G(V, E)$), where $(v_k^{start}, speed_k^{start}) \in start$ and $(v_k^{goal}, speed_k^{goal}) \in goal$;
  $OPEN = \emptyset$;
  insert $Root$ to $OPEN$;
  while $OPEN$ *not empty* **do**
    $N = OPEN$.popMin($cost$);
    $conflicts$ = FindConflicts($N.solution$);
    if $conflicts = \emptyset$ **then**
      return $N.solution$;
    C = first conflict $(action_i, action_j)$ in $conflicts$;
    foreach *agent* $a_k \in C$ **do**
      N' = new node;
      Compute $[action_k.t_{start}, t)$ for $action_k$ w.r.t. the other action;
      $N'.constraints = N.constraints \cup \{(a_k, action_k, t)\}$;
      Create safe intervals using $N'.constraints$;
      $N'.solution[k]$ = DS-SIPP($a_k, v_k^{start}, speed_k^{start}, v_k^{goal}, speed_k^{goal}, G(V, E)$), where $(v_k^{start}, speed_k^{start}) \in start$ and $(v_k^{goal}, speed_k^{goal}) \in goal$;
      if *DS-SIPP found a solution* **then**
        insert N' to $OPEN$;

4.2.1 Proof of validity and optimality

Andreychuk *et al.* [1] proves the optimality and validity of CCBS in their paper. However, in this section, we provide a reformulated proof of optimality and validity of CCBS. It was made to be easy to extend and modify if the model or algorithms were to change and inspired by the proof made by Sharon *et al.* for CBS.

For a given constraint tree, T, and its node $N \in T$, the set $CV(N)$ includes all the valid solutions that satisfy $N.constraints$. Note that the solutions in $CV(N)$ are not necessarily found in a descendant of N in T. We say that N permits solution s, if $s \in CV(N)$. Note that the root has no constraints, and thus, it permits all (valid) solutions.

Denote by $minCost(CV(N))$ the minimum $cost(s)$ of all solutions $s \in CV(N)$.

Lemma 4.1. For a given constraint tree, T, and its node $N \in T$, if $cost(s)$ has Property 2.1 then $cost(N.solution) \leq minCost(CV(N))$.

Proof: Looking at Algorithm 8, we observe that $N.solution$ stores a solution that is consistent with $N.constraints$ (lines 6 and 20). Moreover, the algorithm constructs tree T in such a way that for any node N' that is descendant of N, it holds that $N.constraints \subseteq N'.constraints$ (since only lines 3 and 18 make an assignment to $N.constraints$). Therefore, we can prove the lemma by showing that the cost of the solution encoded in $N.solution$ is smaller or equal to any (valid) solution that satisfies the set of constraints $C' = N.constraints \cup C$, where C is any set of constraints (possibly empty). This proves the lemma according to the definitions of the functions $minCost()$ and $CV()$.

The rest of the proof is followed by $cost(s)$ having Property 2.1, which says that adding constraints never lowers the value of $cost(s)$. ■

In Lemma 4.2 the fact that the root permits all valid solutions is used and the set of all valid solutions is denoted as $CV(Root)$.

Lemma 4.2. $\forall_{s \in CV(Root)} \; \exists_{N \in OPEN} \; s \in CV(N)$. That is, all valid solutions are permitted by at least one node in OPEN.

Proof: We prove this by induction on the number of iterations of the while-loop (line 10 to 22). Specifically, we prove that predicate $P(n) = \forall_{s \in CV(Root)} \; \exists_{N \in OPEN_n} \; s \in CV(N)$ holds, where n is the n-th iteration and $OPEN_n$ is the value of the variable $OPEN$ on the n-th time that the algorithm executes line 22.

Base case: $P(1) = \forall_{s \in CV(Root)} \; \exists_{N \in OPEN_1 = \{Root\}} \; s \in CV(N)$ is trivially true because the only node in $OPEN_1$ is the root node which permits all valid solutions. This is because before the first execution of the while-loop (line 10 to 22), lines 7 and 22 are the only lines in the code that sets the value of the variable $OPEN$.

Induction Hypothesis: We assume that Equation 4.5 is correct.

$$P(n-1) = \forall_{s \in CV(Root)} \; \exists_{N \in OPEN_{n-1}} \; s \in CV(N) \tag{4.5}$$

Induction step: We show that Equation 4.6 is correct.

$$P(n) = \forall_{s \in CV(Root)} \; \exists_{N \in OPEN_n} \; s \in CV(N) \tag{4.6}$$

We show that if all valid solutions are permitted by the nodes in $OPEN_{n-1}$ (Equation 4.5) then all valid solutions are also permitted by the nodes in $OPEN_n$ (Equation 4.6).

In iteration $n-1$, $N' \in OPEN_{n-1}$ was popped from $OPEN_{n-1}$ at line 10 and conflicts were found on line 11. For each conflict $(action_i, action_j)$ considered on line 14, the algorithm creates two new nodes N'_i and N'_j in the for-loop on lines 16-22. These new nodes where constraints

$$N'_i.constraints = N'.constraints \cup \{(a_i, action_i, t')\}$$

and

$$N'_j.constraints = N'.constraints \cup \{(a_j, action_j, t''))\}$$

were modified on line 18. The rest of the proof is implied by Claim 4.1, because in iteration n, the value of $OPEN_n$ is $\{N'_i, N'_j\} \cup OPEN_{n-1} \setminus \{N'\}$ on line 10.

Claim 4.1. Let $N \in T$ be a node on a constraint tree that the algorithm expands into N_i and N_j due to the conflict $(action_i, action_j)$, then $CV(N_i) \cup CV(N_j) = CV(N)$.

Proof: Let S_C be the set of all solutions, possibly not valid, that satisfy C. Moreover, $conf(action_i, action_j)$ is the set of all solutions, valid or not, that schedule both a_i to take $action_i$ during time interval $[action_i.t_{start}, t')$ and a_j to take $action_j$ during time interval $[action_j.t_{start}, t'')$. We show that $S_{N_i.constraints} \cup S_{N_j.constraints}$ includes all the solutions in $S_{N.constraints}$ expect the ones in $conf(action_i, action_j)$.

Equation 4.7 is true since the algorithm added the constraints $(a_i, action_i, t')$ and $(a_j, action_j, t'')$ to N_i, and respectively, N_j when creating these nodes (lines 16-22).

$$S_{N_i.constraints} \cup S_{N_j.constraints} = S_{N.constraints \cup \{(a_i, action_i, t')\}} \cup S_{N.constraints \cup \{(a_j, action_j, t''))\}} \tag{4.7}$$

Note that $S_{N.constraints \cup \{(a_i, action_i, t')\}} \cup S_{N.constraints \cup \{(a_j, action_j, t''))\}}$ contains all solutions consistent with $N.constraints$ and do not schedule both a_i to take $action_i$ during time interval $[action_i.t_{start}, t')$ and a_j to take $action_j$ during time interval $[action_j.t_{start}, t'')$ (because of constraints $(a_i, action_i, t')$ and $(a_j, action_j, t'')$)).

Then, equations 4.8 and 4.9 are equal because the conflict $(action_i, action_j)$ can only be resolved by respecting either constraint $(a_i, action_i, t')$ or $(a_j, action_j, t'')$.

$$S_{N_i.constraints} \cup S_{N_j.constraints} = S_{N.constraints \cup \{(a_i, action_i, t')\}} \cup S_{N.constraints \cup \{(a_j, action_j, t'')\}} \tag{4.8}$$

$$= S_{N.constraints} \setminus conf(action_i, action_j) \tag{4.9}$$

Which is what we wanted to show. Next, we prove the claim by using the fact that $S_{N_i.constraints} \cup S_{N_j.constraints} = S_{N.constraints} \setminus conf(action_i, action_j)$. Note that $CV(N)$ is the set of all valid solutions in $S_{N.constraints}$. Furthermore, $CV(N)$ is also the set all the valid solutions in $S_{N.constraints} \setminus conf(action_i, action_j)$ since no valid solution can include the conflict $(action_i, action_j)$. This means that $CV(N)$ is also the set of all valid solutions in $S_{N_i.constraints} \cup S_{N_j.constraints} = S_{N.constraints} \setminus conf(action_i, action_j)$. The set of all valid solutions in $S_{N_i.constraints}$ are $CV(N_i)$ and $S_{N_j.constraints}$ are $CV(N_j)$. This gives us the equality $CV(N_i) \cup CV(N_j) = CV(N)$. ■

■

Theorem 4.1. If CCBS finds a valid solution $G.solution$ for a goal node G, it will be optimal in terms of $cost(G.solution)$ if $cost(s)$ has Property 2.1.

Proof: If a goal node G is chosen for expansion on line 10, all valid solutions are permitted by at least one node in $OPEN$ as shown in Lemma 4.2. Since the High-Level search explores nodes in a best-first manner w.r.t. cost we have that $\forall_{N \in OPEN}\ G.cost \leq N.cost$. Since $cost(s)$ has Property 2.1, Lemma 4.1 gives us that, $\forall_{N \in OPEN}\ N.cost \leq minCost(CV(N))$. So we get that $\forall_{N \in OPEN}\ G.cost \leq N.cost \leq minCost(CV(N))$. I.e. $G.cost$ is the lower bound for all nodes in $OPEN$. Since G is a goal node it will hold a valid solution, and given the above it is also the optimal solution. This means the algorithm, since $G.solution$ has no conflicts, will pass the check on line 12 and will return $G.solution$ on line 13. ■

5 Target system: Model of a road network

In this text, a road network is modeled as graph $\mathsf{G}(\mathsf{V},\mathsf{E})$ that consist of a combination of connected road components and intersection components. Each $v \in \mathsf{V}$ encodes a position in the road network $pos = (x, y)$ referred as $v.pos$. Using this position an edge length (*i.e.,* distance between positions) can be deduced. Each edge $(v, v') \in \mathsf{E}$ has an associated $distance(v.pos, v'.pos)$, being for example $\sqrt{(x-x')^2+(y-y')^2}$ if the edge is a straight trajectory.

When solving for an optimal solution, path-finding might consider Road and Intersection components a bit differently. However, the resulting solution will still consist of contiguous paths in $\mathsf{G}(\mathsf{V},\mathsf{E})$. Construction of a component consists of mapping the real-world road network into the parameters of vertices and edges that form $\mathsf{G}(\mathsf{V},\mathsf{E})$. Road components focus on the ability for vehicles to be safely position in the road and switch lanes in order to reach the target exit in the road. Intersection components are created to allow vehicles to wait safely for passing obstacles (other vehicles).

5.1 Road component

A road component $\text{RoadComponent}_i(\mathsf{V}^i_{road}, \mathsf{E}^i_{road}) \subseteq \mathsf{G}(\mathsf{V},\mathsf{E})$ consists of two lanes *Left (L)* and *Right (R)*. Vertices $v \in \mathsf{V}^i_{road}$ encodes, in addition to a position, a lane v^{lane} where $lane \in \{Left(L), Right(R)\}$. V^i_{road} contains two *entrance vertices* $v^L_{entrance}$ and $v^R_{entrance}$ and two *exit vertices* v^L_{exit} and v^R_{exit}. Both joined by a set of *lane vertices*. The two lanes are labeled *Left (L)* and *Right (R)*. For a given road component a set of possible speeds $\{speed_0, \ldots, speed_{max}\}$ are defined for all $v \in \mathsf{V}^i_{road}$, where $speed_{max}$ is the speed limit. Each lane vertex is connected by an edge to the two next lane vertices, where an edge to the next vertex on (1) the same lane is called a *forward edge* and (2) the other lane is called a *switch edge*. Note that a switch edge can connect not only to the next vertex, but any succeeding vertex on the other lane. The entrance vertices connect to the first lane vertices by forward edges, likewise the last lane vertices connect to the exit vertices by forward edges.

An example of an instantiated road component can be seen in Figure 5.1. Here $v^L_{entrance}$ (abbreviated to v^L_{entr}) and $v^R_{entrance}$ (abbreviated to v^R_{entr}) are the entrance vertices and v^L_{exit} and v^R_{exit} are the exit vertices and the other five are lane vertices. The forward edges feed from one lane vertex on the same lane to the next (e.g. v^L_1 to v^L_2) and the switch edges feed from one lane vertex to the next vertex on the opposite lane (e.g. v^R_2 to v^L_3).

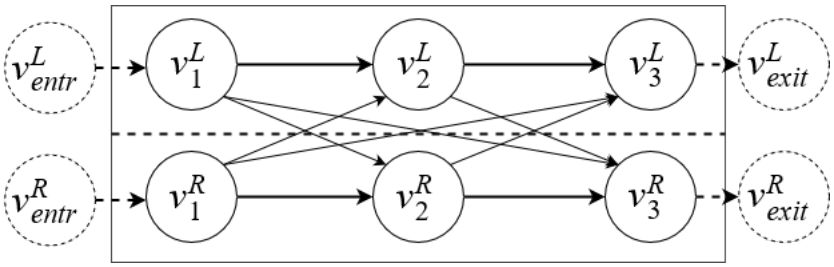


Figure 5.1: Modeling of a straight road with three location on each lane.

When creating a road component, the following factors are important to consider:

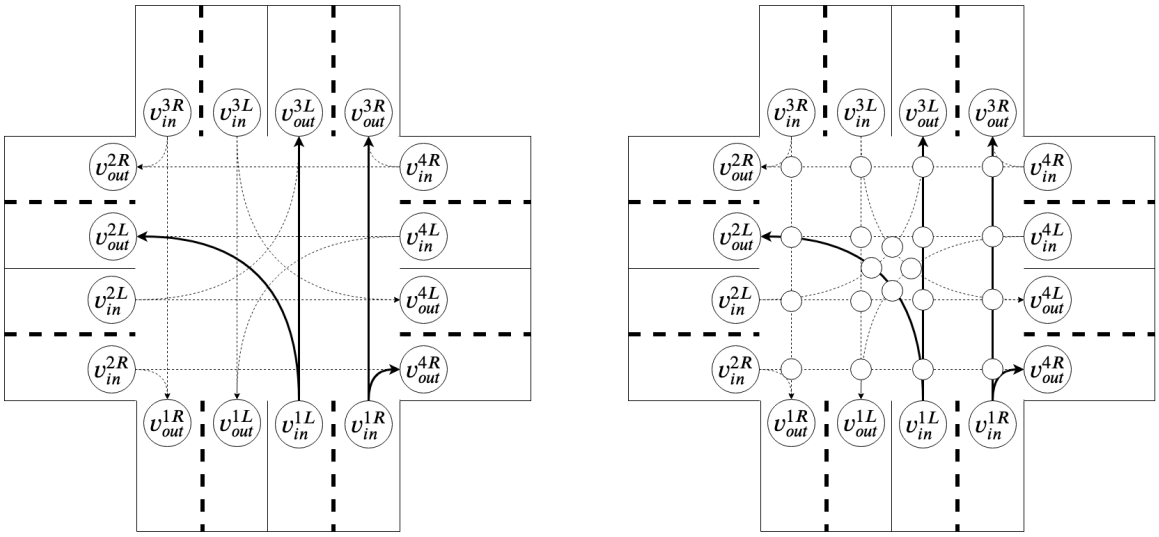
- Length of the road and dimensions of vehicles, that translate into positions in vertices.
- Speed limit of the road, translating to possible speeds $\{speed_0, \ldots, speed_{max}\}$ where $speed_{max}$ would be the speed limit.

5.2 Intersection component

An intersection component $\text{IntersectionComponent}_i(\mathsf{V}^i_{intersection}, \mathsf{E}^i_{intersection}) \subseteq \mathsf{G}(\mathsf{V},\mathsf{E})$ consists of four entrances and four exits. Each entrance is a pair of *in-vertices*. For the i-th entrance, $v^{i,L}_{in}$ is the left in-

vertex and $v_{in}^{i,R}$ is the right in-vertex. These represent traffic entering the intersection. Each exit is a pair of *out-vertices*. For the i-th exit, $v_{out}^{i,L}$ is the left in-vertex and $v_{out}^{i,R}$ is the right out-vertex. These represent traffic leaving the intersection. Connecting the in- and out-vertices are auxiliary vertices, which encode positions inside the intersection. For a given intersection component a set of possible speeds $\{speed_0, \ldots, speed_{max}\}$ are defined for all $v \in \mathsf{V}^i_{intersection}$, where $speed_{max}$ is the speed limit.

An example of an instance of an intersection component can be seen in Figures 5.2a and 5.2b. The four pairs of entrance vertices can be seen as $v_{in}^{1,L}$ and $v_{in}^{1,R}$ to $v_{in}^{4,L}$ and $v_{in}^{4,R}$. Similarly, the four pairs of exit vertices can be seen as $v_{out}^{1,L}$ and $v_{out}^{1,R}$ to $v_{out}^{4,L}$ and $v_{out}^{4,R}$. In Figure 5.2b, auxilliary vertices are shown to encode locations in the intersection. Agents moving through the intersection will be able to stop at these vertices to allow other agents to pass.



(a) Modeling of an intersection.

(b) Modeling of an intersection with auxiliary vertices.

Figure 5.2: Illustration of the graph representation of an intersection.

5.3 Solution: Multi-agent path-finding with kinematic constraints

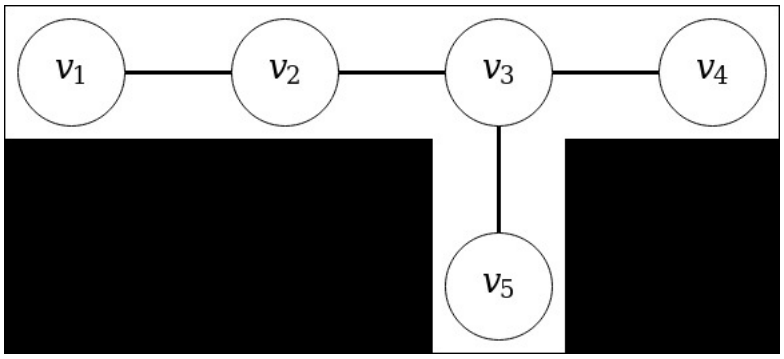


Figure 5.3: Graph used as an example.

The target system will execute a solution that has been post-processed according to the work of Hönig *et al.* [10]. They describe a procedure called *MAPF-POST* that encodes kinematic constraints for a solution. What this does in the context of this work is to relax the assumption that only a discrete set of speeds is

used. MAPF-POST will optimize for a continuous interval of possible speeds, based on the planned speeds in the models. Further research into this area showed limitations related to the work in this thesis, which are discussed in Section 5.3.1.

As a running example, consider Figure 5.3 and a solution in that graph consisting of paths $\{v_1, v_2, v_3, v_4\}$ for $Agent_1$ and $\{v_2, v_3, v_5, v_3\}$ for $Agent_2$. The distance between locations in this graph is uniform and equal to 1 m. This example will be processed by this procedure, which is as follows:

Given a solution s, create a *temporal plan graph (TPG)* that model the dependencies of each path $p_i \in s$. Edges that connect the vertices in the path of one agent are called *Type 1 edges*. The length of such edges is $l(e = (v, v'))$. The edges that connect from one path to the other are special edges called *Type 2 edges*, which have 0 length. All edges in this graph encode that the earlier action must take place before the later action. To illustrate this, consider the TPG in Figure 5.4 for the solution in our running example. The Type 2 edge between v_2 for $Agent_2$ and v_2 for $Agent_1$ models that $Agent_2$ must leave v_2 before $Agent_1$ enters it.

The TPG also stores the maximum and minimum speed for each Type 1 edge. The maximum speed is denoted as $v_{max}(e)$ and the minimum as $v_{min}(e)$. $Agent_1$ in our example has $v_{max}(e) = 1/4\ m/s$ for all its Type 1 edges and $Agent_2$ has $v_{max}(e) = 1/16$ for all its Type 1 edges. Neither agent has a minimum speed and can always come to a complete stop.

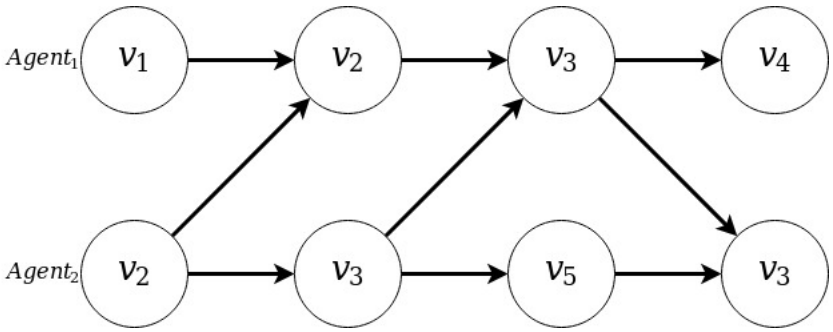


Figure 5.4: A *temporal plan graph (TPG)*.

Next, the TPG is augmented to include auxiliary vertices called *safety markers*. These indicate that an agent has reached an intermediate location $\delta > 0$ distance before or after a vertex. δ is only valid if the length of every edge is greater than $2 * \delta$. The length of the new edges in this augmented graph are either $l(e) - 2 * \delta$ or δ. Type 2 edges that previously connected two vertices v and v', now connect from the safety marker after v to the marker before v'.

The TPG for our running example is augmented in Figure 5.5. All vertices are replaced by a path from a pre-vertex to the vertex which then connects to a post-vertex. All incoming edges are diverted to the pre-vertex and all outgoing edges are diverted from the post-vertex. An example of how type 2 edges change can be observed for the edge between v_2 for $Agent_2$ and v_2 for $Agent_1$. This models that $Agent_2$ must leave the safety marker after v_2 before $Agent_1$ enters the safety marker before v_2.

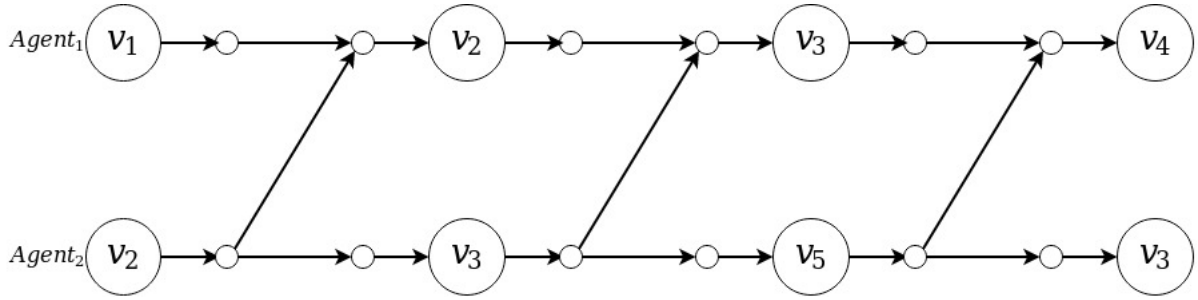


Figure 5.5: An augmented TPG.

A *simple temporal network (STN)* is created from this augmented TPG by assigning continuous time intervals to each edge $e = (v, v')$. These consist of a lower and upper bound $[LB(e), UB(e)]$, where $LB(e) = l(e)/v_{max}(e)$ and $UB(e) = l(e)/v_{min}(e)$. The bounds indicate that event v must be scheduled between $[LB(e), UB(e)]$ time units before event v'.

An STN is created for our running example in Figure 5.6, with $\delta = 0.25\ m$. An example of a dependency in this graph is that $Agent_1$ must be scheduled to be at v_2 between $[0.25/(1/4) = 1, \infty]$ time units after being at its second safety marker. Note that $UB(e) = \infty$, since both agents have no minimum speed.

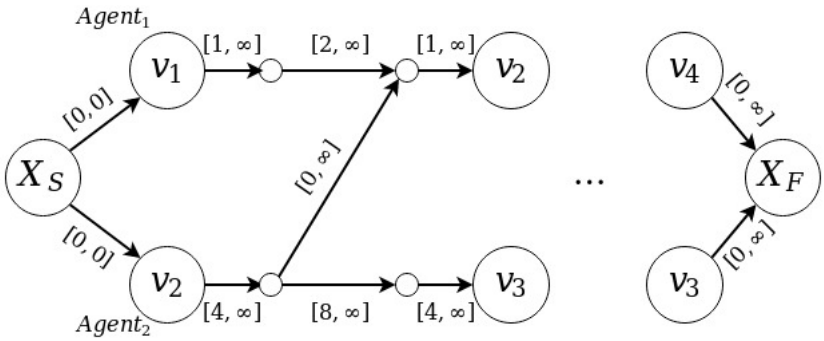


Figure 5.6: A *simple temporal network (STN)*

The next step is to compute the minimum flow-time through this graph with either a linear program or a graph-based optimization algorithm. Let $t(v)$ be the time at the last location in a path, E' be the set of edges of the augmented graph. Then the linear program looks as follows:

$$\begin{aligned} \text{Minimize } & \sum_{j=1}^{K} t(v_j) \\ \text{such that } & t(X_S) = 0 \\ \text{and, for all } e = (v, v') \in \mathsf{E}', & \\ & t(v') - t(v) \geq LB(e) \\ & t(v') - t(v) \leq UB(e) \end{aligned}$$

This creates a solution that has each agent arriving as early as possible at its goal location. Furthermore, the solution will respect the kinematic constraints in the STN.

5.3.1 Limitations and possible solutions

The MAPF-POST algorithm only models dependencies between locations in the graph. This means that conflicts where two agents collide during movement are not encoded in the final STN. As an example, consider Figure 5.7, where two agents have crossing edges. A solver will create a solution for both agents where one waits for the other before moving. When this solution is passed to the MAPF-POST algorithm, a TPG such as in Figure 5.8 will be created. Here we see that the dependency (wait for the other agent) is not present and the two agents might be scheduled to collide. To solve this, the TPG must be modified to encode such information.

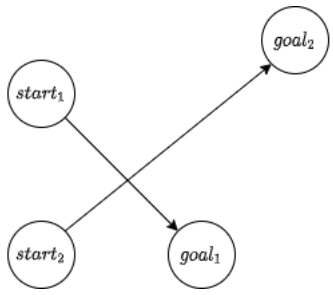


Figure 5.7: Example of crossing which is not modeled as a dependency in basic MAPF-POST.

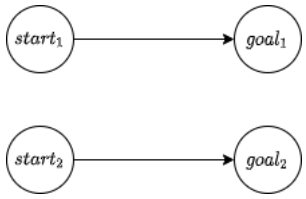


Figure 5.8: Resulting TPG of Figure 5.7

We suggest to solve this by adding vertex at the crossing point to the solution. The time at which the agent passes through this vertex can be inferred from the given solution. This would then be modeled as the TPG in Figure 5.9. The MAPF-POST algorithm can then be used as described in this section.

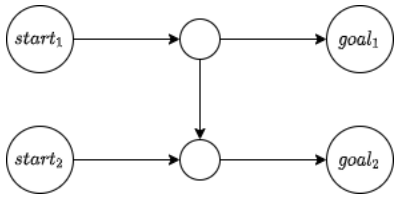


Figure 5.9: Resulting TPG of Figure 5.7 when a vertex representing the crossing is added.

5.4 DT refinement into target system

Here we describe how to create a solution for the target system of road networks in the DT model. Given an input graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ that models a road network, we create a transformed graph in the DT model. In this refinement we treat road and intersection components the same. The transformed graph models the possible speeds in the road network as discrete time-steps. Moreover, a few modifications are needed to make the Low- and High-Level produce a solution that encodes speed. This solution is then post-processed with the procedure in Section 5.3.

The input graph encodes the possible speeds in each given vertex. A slow speed translates into more vertices and a fast speed into fewer vertices. The procedure to transform the input graph is as follows:

1. Replace all vertices v with new vertices $\mathsf{V}_v = \{\mathsf{v}_0, \ldots, \mathsf{v}_{max}\}_v$ for each speed in $\{speed_0, \ldots, speed_{max}\}$.
2. Replace all edges $e = (v, v')$, for each $speed \in \{speed_0, \ldots, speed_{max}\}$, with n connected *intermediate vertices* $\mathsf{I}_{e,speed} = \{\mathsf{i}_1, \ldots, \mathsf{i}_n\}_{e,speed}$. The number of intermediate vertices should be more for a slow speed and less for a fast speed. Also, the number of vertices must scale to correctly model differences between speeds in the target system. For example, 50 km/h would be represented by twice as many intermediate vertices as 100 km/h.
3. For each possible speed transition, $\mathsf{i}_n \in \mathsf{I}_{e,speed}$ then connect to the new vertices $\mathsf{V}_{v'} = \{\mathsf{v'}_0, \ldots, \mathsf{v'}_{max}\}_{v'}$ that replace v'.

Figure 5.10 illustrates this procedure for two locations v_1 and $_2$ with possible speeds $\{0, 1, 2, 3\}$. Here speeds scale in power of two, meaning going one speed faster is twice as fast.

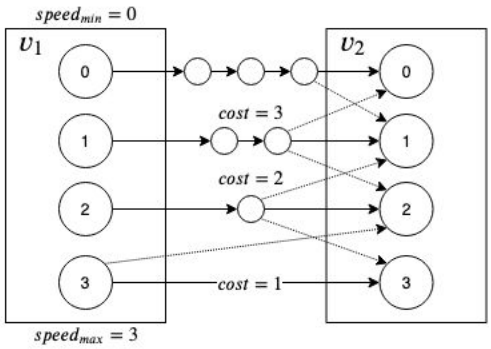


Figure 5.10: Detailed view of how speeds are treated given given two vertices v_1 and v_2.

CBS only checks for conflicts in single vertices and edges for two agents. After the transformation of the input graph, two agents can be in the same location at once. Using the example above, one agent in $\mathsf{v}_0 \in \mathsf{V}_{v_1}$ and $\mathsf{v}_1 \in \mathsf{V}_{v_1}$ would be ok given the definition of a conflict in Section 2. To solve this, we modify this definition:

Definition 5.1 (Conflict)**.** A *conflict* between two agents a_i and a_j in states st_i and st_j at time t is one of two types:

- (Vertex conflict) (a_i, a_j, v, t) where agents a_i and a_j plan to occupy the same vertex set $\mathsf{V}_v \cup \mathsf{I}_{(v,v'),speed}$ at the same time step t.
- (Edge conflict) $(a_i, a_j, (v, v'), t)$ where agents a_i and a_j plan to occupy the same edge. That is, at time step t a_i plans to move from $v = st_i.v$ to v' and a_j plans to move from $v' = st_j.v$ to v.

This means, for example, that an agent cannot enter $\mathsf{v}_0 \in \mathsf{V}_v$ if another agent is present in any of the vertices in V_v or $\mathsf{I}_{(v,v'),speed}$.

In the target system, an agent moving at a speed larger than 0 will need time to come to a complete stop. To enforce this, we modify Constrained-A* so that a path can only include a WAIT action in a vertex that encodes speed 0.

A solution of the transformed input is then transformed back into the target system by using the procedure described in Section 5.3. To simplify this procedure, intermediate vertices are removed from the solution. The resulting solution will still encode the speed an agent has in each location. For a given MOVE action from $\mathsf{v}_i \in \mathsf{V}_v$ to $\mathsf{v'}_{i'} \in \mathsf{V}_{v'}$, we say that $e = (\mathsf{v}_i, \mathsf{v'}_{i'})$. We assign $v_{min}(e)$ and $v_{max}(e)$, such that $v_{min}(e) \leq min(speed_i, speed_{i'})$ and $v_{max}(e) \geq max(speed_i, speed_{i'})$. This allows post-processing to compensate for the fact that the solver only considers discrete speeds. It can optimize for a continuous speed interval $[v_{min}(e), v_{max}(e)]$ instead.

5.5 CT refinement into target system

The procedure to create a solution for the target system is similar to the CT model. It takes an input graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$ and transforms it into the CT model. Road and intersection components are treated the same in this refinement as well. The same modification made to Constrained-A* is made to SIPP. The solution produced by CCBS is then post-processed using the procedure in Section 5.3.

We represent speeds as the cost of edges. A high cost is equal to more time and a slow speed. A low cost is equal to less time and a fast speed. The procedure to transform the target system into the CT model is as follows:

1. Replace all vertices v with new vertices $\mathsf{V}_v = \{\mathsf{v}_0, \ldots, \mathsf{v}_{max}\}_v$ for each speed in $\{speed_0, \ldots, speed_{max}\}$.
2. Replace all edges $e = (v, v')$ with new edges that represent maintaining speed or changing speed between v and v'. For each $speed_i, speed_{i'} \in \{speed_0, \ldots, speed_{max}\}$ and possible speed change, create

new edges with costs that represent an agent moving from v at $speed_i$ to v' at $speed_{i'}$. The cost of each edge should be based on how long this takes in the target system.

Figure 5.11 illustrates this for two locations v_1 and $_2$ with possible speeds $\{0, 1, 2, 3\}$.

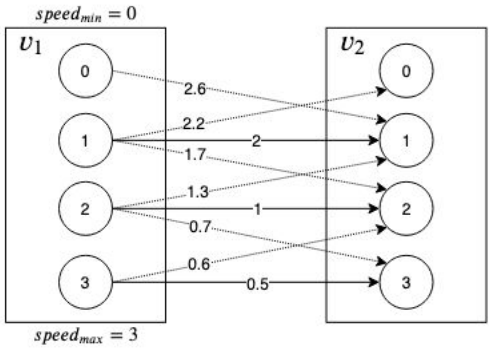


Figure 5.11: Converting an edge with speeds.

Collision-checking in CCBS now needs to account for the fact that all vertices in V_v represent the same location. This means that if two agents are present in any $\mathsf{v}_i \in \mathsf{V}_v$ and $\mathsf{v}_j \in \mathsf{V}_v$ respectively at the same time, they will collide. SIPP is modified so that the path of a single agent can only include a WAIT action in a vertex encoding 0 speed.

As in the DT refinement, a solution of the transformed input is then transformed back into the target system by using the procedure described in Section 5.3. For a given MOVE action from $\mathsf{v}_i \in \mathsf{V}_v$ to $\mathsf{v}'_{i'} \in \mathsf{V}_{v'}$, we say that $e = (\mathsf{v}_i, \mathsf{v}'_{i'})$. We assign $v_{min}(e)$ and $v_{max}(e)$, such that $v_{min}(e) \leq min(speed_i, speed_{i'})$ and $v_{max}(e) \geq max(speed_i, speed_{i'})$. Again, this allows post-processing to compensate for the fact that the solver only considers discrete speeds.

5.6 CTDS refinement into target system

The CTDS model can create a solution for the target system by defining the functions *actionTime* and *actionPossible* for the input graph $\mathsf{G}(\mathsf{V}, \mathsf{E})$. *actionTime* is created for each pair of vertices and possible speeds. *actionPossible* is created based on conditions of the road and/or specification of the vehicles. Note that we treat road and intersection components the same again.

Then, CCBS and DS-SIPP can create a solution without modification. The solution is then, just as in the case of the DT and CT model, post-processed by the procedure in Section 5.3. For a given MOVE action from v at $speed_v$ to v' at $speed_{v'}$, we assign $v_{min}(e)$ and $v_{max}(e)$, such that $v_{min}(e) \leq min(speed_v, speed_{v'})$ and $v_{max}(e) \geq max(speed_v, speed_{v'})$. As before, this compensates for the fact that the solver only considers discrete speeds.

6 Evaluation

We made a preliminary evaluation to investigate if having more than two speeds (including 0) would prove beneficial on a two-lane road. Our running hypothesis was that more speeds would not increase performance. This was also what we found. However, we did find evidence of small performance gains.

The evaluation was made on an Intel NUC7i7BNH that uses an i7-7567U processor. Only the CTDS model was implemented and tested. The metric used for the performance of a solution was the makespan. We considered a finite segment of an infinite straight two-lane road. Figure 5.1 is an example of such a segment. Since we did not investigate queuing behavior, agents were assumed to disappear from the segment when reaching an exit vertex. A segment had a random number of lane vertices. The vertices were placed at a uniform distance from each other. Agents were placed at random start vertices and assigned a random exit vertex (on the left or right lane). Parameters for the experiment were:

- Number of speeds ranging from 2 to 10 including speed 0.
- Number of agents ranging from 3 to 25.
- Number of vertex pairs, called locations, ranging from 5 to 48.

The results show that very little can be gained from having more than two speeds. The average makespan is close to identical for all speeds, seen in Table 6.1. The execution time also gets worse when considering more speeds, which can be seen in Figure 6.2.

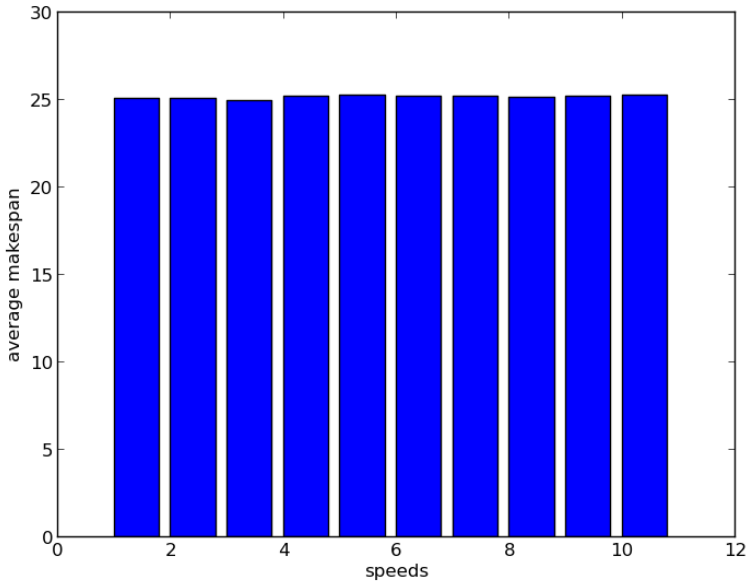


Figure 6.1: Average makespan for each speed over all trials.

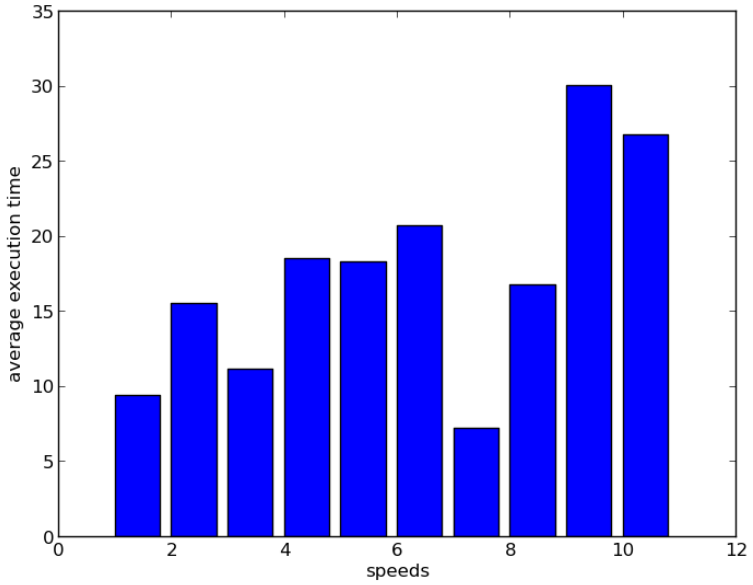


Figure 6.2: Average execution time for each speed over all trials.

There is evidence for some small performance gain, however. Table 6.1 lists the average makespan for each number of speeds 1 to 9 and agent count ranging 3-5, 5-10, 10-15, 15-20 to 20-25. As before, it is evident that the difference in makespan is very small but there seem to be some consistent small gains with having more than two speeds. The red area in Figure 6.3 plots the difference between the best makespan for a given agent count and the average makespan when only using two speeds (including speed 0). The blue area is the difference between the best makespan for a given agent count and the worst makespan out of all speed sets. The values are plotted based on Table 6.1. For example, the average makespan using two speeds (row 1) for agent count 3-5 (column 1) is 5.57. The best makespan for 3-5 agents was found to be 5.37 using the set of three speeds (row 2). This is plotted as $5.57 - 5.37 = 0.2$ in Figure 6.3. Even though the loss in performance is very small, it appears that only considering two speeds is almost always the option that yields the worst makespan.

	Agent count				
	3-5	5-10	10-15	15-20	20-25
2	5.57	10.67	18.16	25.97	33.81
3	5.37	10.45	18.16	25.81	33.65
4	5.48	10.47	18.04	25.43	33.33
5	5.48	10.63	18.03	25.77	33.10
6	5.46	10.62	18.11	25.60	33.15
7	5.49	10.72	17.91	25.72	33.58
8	5.55	10.57	17.84	25.42	33.46
9	5.53	10.60	18.00	25.56	33.49
10	5.57	10.61	18.17	25.64	33.41

Table 6.1: Average makespan aggregated over each speed and agent count.

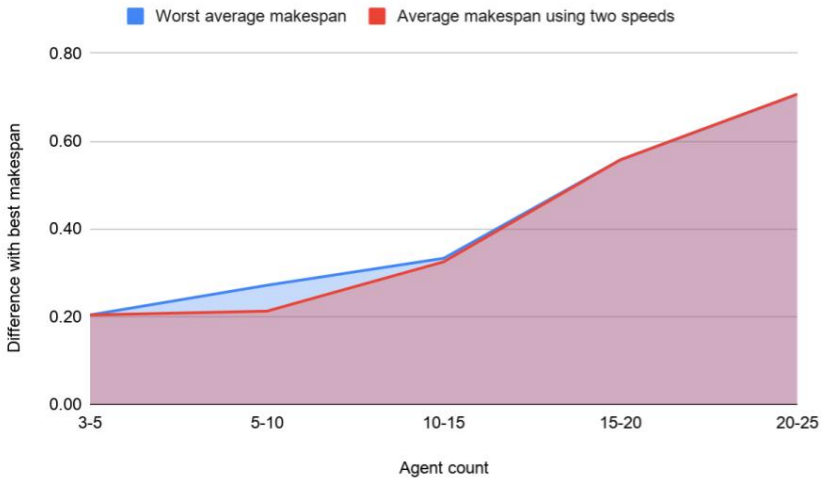


Figure 6.3: Comparison of difference in makespan for Table 6.1

In summary, vehicles traveling on two-lane straight roads do not gain much in performance from having more speeds to choose from. However, even in this simple example, more speeds show consistent performance gains. This implies that if another problem instance was used, the performance gain would be larger. Examples of such instances are highway-entrances, where vehicles need to adapt their speeds to cars already on the highway. Another example is a queue that is dense enough to make going at full speed impossible.

7 Conclusion

This project dealt with an extensive literature review of the field of conflict-based path-finding algorithms. Using this review, we defined a novel model for planning for the speeds in a multi-agent path-finding setting. A novel approach of transforming and then solving a path-planning problem in a road network for a framework of three models was described. We showed empirically that a gap between existing models and best performance at the target system of road networks existed.

In conclusion, this report has described how to use abstract models for solving problems for the target system of road networks. We showed three different models that considered a different level of detail. Each model refined the previous, adding or changing definitions and algorithms. For each model, a single-agent solution was described together with a multi-agent solution that used the single-agent solution. We presented a proof of optimality and validity for each algorithm, either a new one or an existing one.

A novel CTDS model, along with single- and multi-agent path-planning algorithms, was introduced. The model and algorithms were modifications to previous work, resulting in a new model and algorithms for solving for the speed of multiple agents. The two algorithms developed were a modified version of *safe interval path-planning (SIPP)* called *discrete-speeds SIPP (DS-SIPP)* and a version of *continuous conflict-based search (CCBS)* that uses DS-SIPP. DS-SIPP contributes with a method of planning for a discrete set of speeds of an agent. Functions were described that encode the time and possibility of movement of agents given location and speed. CCBS using DS-SIPP can create a multi-agent solution where agents move at many speeds without colliding.

Another contribution of this work was a novel method of creating a solution for the target system in all three models. This was achieved by a transformation of an input graph from the target system into each model. Solvers in two of the models were modified to handle speed. Each model could then produce a solution that could be transformed back into the target system via post-processing.

Future work for this project would include refining the models to solve for other target systems. This can, for example, be path-finding in three dimensions, while considering drones in open air or spacecrafts. Aspects of how to model movement and conflicts in such higher dimension problems would be interesting to explore. In regards of road networks, adding more traffic rules and dynamics would move the solution further towards real-world traffic.

References

[1] Anton Andreychuk, Konstantin Yakovlev, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 39–45, 2019.

[2] ROBOTS Association. Kiva systems, 1999.

[3] Dor Atzmon, Roni Stern, Ariel Felner, Glenn Wagner, Roman Barták, and Neng-Fa Zhou. Robust multi-agent path finding. In *Eleventh Annual Symposium on Combinatorial Search*, 2018.

[4] Antonio Casimiro, Emelie Ekenstedt, and Elad Michael Schiller. Membership-based manoeuvre negotiation in autonomous and safety-critical vehicular systems. *arXiv preprint arXiv:1906.04703*, 2019.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[6] Stephen J Guy and Ioannis Karamouzas. Guide to anticipatory collision avoidance. *Game AI Pro*, 2, 2019.

[7] Joakim Gyllenskepp and Kevin Nordenhög. Delay-tolerant multi-agent path finding, 2019.

[8] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[9] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1972.

[10] Wolfgang Hönig, TK Satish Kumar, Liron Cohen, Hang Ma, Hong Xu, Nora Ayanian, and Sven Koenig. Multi-agent path finding with kinematic constraints. In *Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

[11] Hang Ma, Wolfgang Hönig, TK Satish Kumar, Nora Ayanian, and Sven Koenig. Lifelong path planning with kinematic constraints for multi-agent pickup and delivery. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7651–7658, 2019.

[12] Thomas Petig, Elad M Schiller, and Jukka Suomela. Changing lanes on a highway. In *18th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[13] Mike Phillips and Maxim Likhachev. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE International Conference on Robotics and Automation*, pages 5628–5635. IEEE, 2011.

[14] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.* AAAI Press, 2012.

[15] Jingjin Yu and Steven M LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[16] Jingjin Yu and Daniela Rus. Pebble motion on graphs with rotations: Efficient feasibility tests and planning algorithms. In *Algorithmic foundations of robotics XI*, pages 729–746. Springer, 2015.

A Stalling

Work has been done in the past to improve CBS in the DT model. If delays should arise during the execution of a solution agents might run the risk of colliding with each other. Therefore, Gyllenskepp and Nordenhög [7] developed a method to resolve delays in a live environment called *stalling*. Given an agent a_i that is delayed by a number of time-steps δ, stop all other agents for δ time-steps. This method can be refined into *component stalling* which only stalls agents with paths affected by the delay, defined as:

1. Agents crossing the path of a_i at some time-step $\pm bound$ the delay. We call these agents *affected* by the delay.
2. Agents crossing the path of an agent affected by the delay at some time-step $\pm bound$. Meaning agents not directly crossing the path of a_i can also be affected by the delay.

B Improvement of CBS: MA-CBS

Here we describe an improvement to basic CBS by Sharon *et al.*, which might also be applied to the other algorithms in this text. Experiments made by Sharon *et al.* [14] shows that basic CBS performs poorly for *strongly coupled* agents *i.e.,* agents with high rate of conflicts between them. They solve this by modifying CBS into a new algorithm called *meta-agent CBS (MA-CBS)* that identifies strongly coupled agents. Strongly coupled agents are merged into what is called a *meta-agent*. The Low-Level will solve a separate MAPF problem instance if the meta-agent consist of several merged agents. Otherwise, an algorithm such as Constrained-A* is used. The choice of multi-agent Low-Level procedure must be an algorithm that detects an unsolvable problem. So CBS cannot be used directly without the pre-processing step discussed in Section 2.4.3.

MA-CBS adds lines 14 to 24 to the original algorithm and the new input to MA-CBS is a MAPF problem instance instance defined as:

$$MAPF = \{\mathsf{A} : \text{set of meta-agents } \{MA_1 = \{a_1\}, \ldots, MA_k = \{a_k\}\},$$
$$\mathsf{G}(\mathsf{V}, \mathsf{E}) : \text{graph } \mathsf{G}(\mathsf{V}, \mathsf{E}),$$
$$start(MA_i) : \text{start vertices of meta-agent } i,$$
$$end(MA_i) : \text{end vertices of meta-agent } i, \}$$

The set of constraints that were added due to conflicts between agents in meta-agent MA_i and MA_j are denoted as $internal(i, j)$.

Algorithm 9: A high-level description of meta-agent conflict-based search by Sharon et al. [14]

required function $cost(s)$ returns a real number, which is the cost of s, e.g. *SIC* or *Makespan*
Function $MA\text{-}CBS(MAPF)$
$Root.constraints = \emptyset$;
foreach *meta-agent* $MA_k \in MAPF.A$ **do**
    $Root.solution[k] = \text{LowLevel}(start(MA_k), goal(MA_k), \mathsf{G}(\mathsf{V}, \mathsf{E}), Root.constraints)$;
$OPEN = \emptyset$;
insert *Root* to $OPEN$;
while $OPEN$ *not empty* **do**
    $N = OPEN.\text{popMin(cost)}$;
    $conflicts = \text{FindConflicts}(N.solution)$;
    if $conflicts = \emptyset$ **then**
        return $N.solution$;
    $C =$ first conflict (a_x, a_y, v, t) in $conflicts$;
    if $shouldMerge(a_x, a_y)$ **then**
        $MA_i = MA_i \in MAPF.A$, such that $a_x \in MA_i$;
        $MA_j = MA_j \in MAPF.A$, such that $a_y \in MA_j$;
        $MA_{i,j} = MA_i \cup MA_j$;
        $MAPF.A = (MAPF.A \cup \{MA_{i,j}\}) \setminus \{MA_i, MA_j\}$;
        $N.constraints = N.constraints \setminus internal(i, j)$;
        $mergedSolution = \text{LowLevel}(start(MA_{i,j}), goal(MA_{i,j}), \mathsf{G}(\mathsf{V}, \mathsf{E}), N.constraints)$;
        for *agent* $a_k \in MA_{i,j}$ **do**
            $N.solution[k] = mergedSolution[k]$;
        if $cost(N.solution) < \infty$ **then**
            Insert N into $OPEN$;
    else
        foreach *agent* $a_k \in C$ **do**
            $N' =$ new node;
            $N'.constraints = N.constraints \cup \{(a_k, v, t)\}$;
            $N'.solution[k] = \text{LowLevel}(start(MA_k), goal(MA_k), \mathsf{G}(\mathsf{V}, \mathsf{E}), N'.constraints)$;
            if *LowLevel found a solution* **then**
                insert N' to $OPEN$;