

Prioritized Planning for Continuous-time Lifelong Multi-agent Pathfinding

Alvin Combrink¹, Sabino Francesco Roselli¹, Martin Fabian¹

Abstract—Multi-agent Path Finding (MAPF) is the problem of planning collision-free movements of agents so that they get from where they are to where they need to be. Commonly, agents are located on a graph and can traverse edges. This problem has many variations and has been studied for decades. Two such variations are the *continuous-time* and the *lifelong* MAPF problems. In the former, edges have non-unit lengths and volumetric agents can traverse them at any real-valued time. In the latter, agents must attend to a continuous stream of incoming tasks. Much work has been devoted to designing solution methods within these two areas. To our knowledge, however, the combined problem of continuous-time lifelong MAPF has yet to be addressed.

This work addresses continuous-time lifelong MAPF with volumetric agents by presenting the fast and sub-optimal Continuous-time Prioritized Lifelong Planner (CPLP). CPLP continuously assigns agents to tasks and computes plans using a combination of two path planners; one based on CCBS and the other based on SIPP. Experimental results with up to 800 agents on graphs with up to 12 000 vertices demonstrate practical performance, where maximum planning times fall within the available time budget. Additionally, CPLP ensures collision-free movement even when failing to meet this budget. Therefore, the robustness of CPLP highlights its potential for real-world applications.

I. INTRODUCTION

Multi-agent Path Finding (MAPF) is the problem of planning collision-free movements of agents to get them from where they are to where they need to be. Many methods exist for the many variants of the problem. In its most common form, time is discretized and agents are represented by points moving on a graph. Each edge takes unit-time to traverse and all agents move in lockstep. If any two agents occupy the same vertex or traverse the same (undirected) edge in opposite directions at the same time, a collision occurs [9, 17]. This problem has been studied for decades [4] and is NP-hard to solve for minimum *makespan*, *sum-of-arrival-times* and travelled distance [21].

Optimal solvers include M* [19], Increasing-cost tree search [15], and the seminal Conflict-based Search (CBS) [14]. Among the sub-optimal, heuristic-based solvers are Windowed Hierarchical Cooperative A* [16], bounded sub-optimal CBS [2] and Priority Inheritance with Backtracking (PIBT) for fast planning of large numbers of agents with certain theoretical guarantees [11].

We gratefully acknowledge the Vinnova project CLOUDS (Intelligent algorithms to support Circular soLutions fOr sUustainable proDuction Systems), and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

¹Division of Systems and Control, Department of Electrical Engineering, Chalmers University of Technology, Göteborg, Sweden {combrink, rsabino, fabian}@chalmers.se

In recent years, much focus has been directed toward the continuous-time variant MAPF_R, where edges can take any positive time to traverse and agents can traverse them at any time. In many cases, volumetric agents are considered. Safe Interval Path Planning (SIPP) [12] underlies many solution methods for MAPF_R, such as Continuous-time CBS (CCBS) [1], Satisfiability Modulo Theory-CBS [18] and PSIPP [6], amongst others.

Another variant of MAPF is the *lifelong*, or *online*, version. Here, agents must attend to a stream of incoming and a priori unknown tasks, which are completed when an agent occupies a respective task’s target vertex some time after the task has been released to the system. Agents are not assumed to always be assigned a task, thus, methods for this problem must address idle agents. Substantial work has been done in this area too, such as token passing [10], Rolling horizon collision avoidance [8], FM-scheduler [13], Primal₂ which uses reinforcement learning for decentralised path planning in partially observable environments [3], and [22] for concurrent planning and executing in an online setting, where plans are refined for as long as time constraints allow.

Most industrial automation systems (warehouses, assembly lines, etc.) run continuously and indefinitely, in spaces that cannot be captured by graphs with unit-length edges. The lifelong and continuous-time MAPF variants each address one aspect of this, however, the combined problem of lifelong MAPF in continuous-time (LMAPF_R) remains largely unexplored. To fill this gap, we introduce the *Continuous-time Prioritized Lifelong Planner* (CPLP), a fast, sub-optimal solver for the collision-free planning of volumetric agents on a graph in metric 2D space for lifelong MAPF.

The outline of the article is as follows: relevant background is given in Section II, followed by the problem definition in Section III. The planner is described in Section IV and experimentally evaluated in Section V. Finally, Section VI includes discussions and Section VII concludes the article.

II. BACKGROUND

Here we introduce SIPP [12], CCBS [1], and graph pre-computations [6, 20] that the following work builds on.

A. SIPP

Safe Interval Path Planning (SIPP) [12] is a method to plan in continuous time the motion of an agent in an environment with dynamic obstacles. If a dynamic obstacle occupies a vertex or edge during a certain time interval, then it is unsafe for the agent to also do so at any time within this interval as that would cause a collision. Thus, the combined movement of all obstacles provides a sequence of alternating safe and

unsafe time intervals for each vertex and edge. A collision-free trajectory from an agent’s current location and time to a target location can be planned by performing an A* [5] search in the vertex-safe interval space.

B. CCBS

Continuous-time Conflict Based Search (CCBS) [1] extends CBS [14] to the MAPF_R problem. Given a set of agents on a graph, each with their respective start and target vertices, CCBS finds an optimal collision-free trajectory for each agent. CCBS’s high-level algorithm is a best-first search in a binary constraint-tree. A node N in the tree represents a set N_{Π} containing one trajectory for each agent, and a set of constraints N_c . At the root node N^r , N_{Π}^r contains for each agent the shortest path from its start vertex to target vertex without considering other agents, and $N_c^r = \emptyset$. N^r is inserted into the open set. At each iteration of the high-level search, a node N minimizing an objective function over all nodes in the open set is selected. If no collisions are detected between the trajectories in N_{Π} , then N_{Π} is returned as the solution. However, if a collision is detected, say between agent a_1 performing an action m_1 at time t_1 and a_2 performing action m_2 at time t_2 , then two new nodes N^1 and N^2 are spawned. For $i = 1, 2$, $N_c^i = N_c \cup \{c_i\}$ where c_i forbids a_i from performing m_i within the interval $[t_i, t_i^u)$, where t_i^u is the earliest time where m_i can be performed without colliding with the other agent performing its action. All trajectories in N_{Π} are copied into N_{Π}^i except for a_i ’s path which is recomputed using CSIPP (the variant of SIPP used in [1]) to satisfy N_c^i .

C. Graph Pre-computation

Determining if arbitrary geometric shapes overlap can be computationally expensive, which is particularly undesirable in online systems. However, by assuming that all agent volumes are described by circles with the same radius and move with the same speed, [6, 20] describe how unsafe intervals for edge-vertex and edge-edge pairs can be pre-computed. For practical roadmaps, computations can be done in near $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}| \log |\mathcal{E}|)$ and then stored in lookup tables for use during runtime.

III. PROBLEM DEFINITION

An LMAPF_R problem is a tuple $\langle \mathcal{G}, \mathcal{A}, \mathcal{T}, v_s \rangle$ containing a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, agents \mathcal{A} , tasks \mathcal{T} , and a mapping $v_s : \mathcal{A} \rightarrow \mathcal{V}$ defining the agents’ respective start vertices.

\mathcal{G} is connected and directed, with $\mathcal{V} \subseteq \mathbb{R}^2$ being points in 2D space and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ connecting vertices. Agents move along edges with the same constant speed s and have circular shapes with radius r . When traversing an edge $e = \langle v_1, v_2 \rangle \in \mathcal{E}$, agents travel in a straight line from v_1 to v_2 , taking $\frac{|v_1 - v_2|}{s}$ time to do so. A collision occurs when two agents’ volumes overlap. In this case of circular agents, that is when the distance between the positions of two agents is less than $2r$.

\mathcal{T} is defined as a, possibly unbounded, multiset over $\mathcal{V} \times \mathbb{R}$, allowing for multiple instances of the same task. A task $\tau =$

$\langle v_{\tau}, t_{\tau} \rangle \in \mathcal{T}$ specifies a target vertex v_{τ} and a release time t_{τ} . For τ to be completed, an agent must be located at v_{τ} at some time $t \geq t_{\tau}$. Knowledge of τ is only released to the system at $t = t_{\tau}$.

We define a *move-action* as a tuple $\langle e, t \rangle \in \mathcal{E} \times \mathbb{R}$ (starting to traverse an edge e at time t) and a *wait-action* as a triple $\langle v, t_1, t_2 \rangle \in \mathcal{V} \times \mathbb{R}^2$ (occupying vertex v from time t_1 until t_2). A plan π_a for an agent a is a sequence of move and wait-actions.

A satisfying solution to an LMAPF_R problem provides a collision-free plan for each agent $a \in \mathcal{A}$, starting at vertex $v_s(a)$, such that all tasks in \mathcal{T} are completed as time $t \rightarrow \infty$. Throughput is defined as the number of completed tasks per time unit, and is typically used as a measurement of solution quality. In real-world settings, however, computation time is also a critical measurement to consider.

IV. METHOD

In this section, we present the *Continuous-time Prioritized Lifelong Planner* (CPLP) and the two path-planners¹.

A. Continuous-time Prioritized Lifelong Planner

Agent plans are initially empty, $\forall a \in \mathcal{A} : \pi_a = \langle \rangle$, and extensions to these plans are computed by repeatedly calling CPLP as tasks enter the system. When called, CPLP computes plans starting at Δt in the future which are then appended to the end of the agent plans. Thus, actions are never removed. A sufficiently large choice of Δt provides enough time for CPLP to compute plans before they start.

Concretely, at time t , $\text{CPLP}(t_{plan}, \mathcal{T}_{new})$ is called with $t_{plan} = t + \Delta t$ and $\mathcal{T}_{new} = \{\tau \in \mathcal{T} \mid t' < t_{\tau} \leq t\}$ where t' is the last time the planner was called. A set of uncompleted tasks is updated, $\mathcal{T}^U \leftarrow \mathcal{T}^U \cup \mathcal{T}_{new}$, and each task is assigned a priority. Based on these priorities, a specific task $\tau^* = \langle v_{\tau^*}, t_{\tau^*} \rangle$ and an assigned agent a^* is prioritized by computing an extension to a^* ’s plan to complete τ^* while moving all idle agents out of the way. The remaining agents are respectively assigned a task, and a short plan within a time horizon \bar{t} is quickly computed to move them *toward* the task vertex without necessarily reaching it. If the computed plans complete all tasks in \mathcal{T}^U , then None is returned; CPLP is called next time a new task enters the system. Otherwise, CPLP returns t_{next} from which time new plans are required; CPLP is called at $t_{next} - \Delta t$ (such that $t_{plan} = t_{next}$).

Unlike PSIPP [6] which computes an entire plan for each agent in order of priority, CPLP computes an entire plan for only one agent and the remaining agents get a short plan. This has two advantages in this lifelong setting: First, plans based on old information are avoided. New tasks entering the system can be acted upon earlier when decided plans are shorter. Second, by computing shorter plans in less time more often, the computation is spread out over more calls to the planner. This can offer more predictability in the computation time, which is generally advantageous in online settings.

¹The source code, experimental setup, animations, and other supplementary details can be found at <https://github.com/Adcombrink/CPLP-LifelongMAPFR>.

We regard CPLP as more similar to PIBT [11], despite PIBT working with discrete time while PSIPP works with continuous time. In PIBT, one-step movements are planned for all agents at every time-step, in order of agent priority. The highest priority agent remains as such until it reaches its target vertex. Additionally, with minor assumptions on the graph, the algorithm guarantees that the highest priority agent will at every time-step move along its preferred edge. By recursion, the highest priority agent will eventually reach its target vertex. Since the highest priority agent is always an agent that has yet to reach its target vertex, reachability (defined by [11] as each agent reaching its assigned target vertex within a bounded time limit) is guaranteed. However, it is not obvious how to directly apply PIBT to the continuous-time setting with agent volumes, as the core algorithm relies on (1) all agents moving in lockstep, and (2) an agent’s movement disrupts at most one other agent. In LMAPF_R , agents do not generally move in lockstep since edges have non-unit lengths, and an agent’s movements can potentially disrupt several other agents due to their volumes.

CPLP employs two primary strategies. First, we assume that all agents remain idle indefinitely once reaching the end of their plan. The implication of this is that a plan for agent a cannot be decided if it intersects the final position of some other agent a' , unless a plan to move a' out of the way is simultaneously decided. We cannot guarantee that such a plan exists for a' . Thus, if a ’s plan is decided without also verifying that a plan for a' to avoid a collision exists, then collision-free movement is not guaranteed. Therefore, through this strategy we ensure that no agents collide so long as they follow their decided plan and then remain idle.

Second, much like PIBT, one task-agent pair is prioritized over all others. However, unlike PIBT which can rely on graph assumptions to ensure that one-step movements will take the prioritized agent to its task vertex, CPLP does not. Instead, CPLP computes the entire plan for the prioritized agent all the way to its task vertex, while moving idle agents out of the way. Using a solution-complete planner ensures that such a set of plans will be found if it exists. How CCBS is applied for this is discussed in Section IV-B, however, we do not guarantee solution-completeness of our implementation. Given that all agents could in the worst case come to a rest at their last planned positions without collision, if the graph is well-connected and there is sufficient space for agents to maneuver (considering their volumes and the graph’s geometry), then we postulate that such a set of plans exists.

The Pseudo-code for CPLP is presented in Algorithm 1. On lines 3-8, variables are updated with t_{plan} and \mathcal{T}_{new} : tasks in \mathcal{T}_{new} are added to \mathcal{T}^U ; TASKCOMPLETION removes all tasks from \mathcal{T}^U that will be completed by existing agent plans; if there are no remaining uncompleted tasks in \mathcal{T}^U then None is returned since there is no need to plan; ADDWAITACTIONS extends every agent’s plan that ended before t_{plan} with a wait-action at its last vertex until t_{plan} ; and $\text{UPDATETASKPRIORITIES}$ updates the priorities of the tasks. Any prioritization scheme can be used in UPDATE-

Algorithm 1 Continuous-time Prioritized Lifelong Planner

```

1: procedure CPLP( $t_{plan}, \mathcal{T}_{new}$ )
2:
3:    $\mathcal{T}^U \leftarrow \mathcal{T}^U \cup \mathcal{T}_{new}$ 
4:    $\text{TASKCOMPLETION}$ 
5:   if  $\mathcal{T}^U = \emptyset$  then
6:     return  $\text{None}$ 
7:    $\text{ADDWAITACTIONS}$ 
8:    $\text{UPDATETASKPRIORITIES}$ 
9:
10:  if  $\tau^* = \text{None}$  then
11:     $\text{PRIORITIZEDPLANS}$ 
12:
13:   $\text{ASSIGNTASKSTOAGENTS}$ 
14:  if  $\tau^* = \text{None}$  then
15:     $\text{SHORTRANDOMPLANS}$ 
16:  else
17:     $\text{SHORTPLANS}$ 
18:
19:  return  $\text{GETNEXTPLANNINGTIME}$ 

```

TASKPRIORITIES , however, care must be taken to avoid task starvation. A simple scheme is used here; a task’s priority is proportional to the time it has gone without completion.

Only one task τ^* is prioritized at any given time. Thus, $\tau^* \leftarrow \text{None}$ at the time when a^* arrives at v_{τ^*} . If no task is prioritized at time t_{plan} (line 10), PRIORITIZEDPLANS on line 11 selects a new prioritized task-agent pair and computes plans for a^* to complete τ^* and all idle agents to move out of the way: For each task in \mathcal{T}^U (by descending priority), the agent with the earliest arrival time at the task vertex (when traversing the shortest path after following its existing plan) is selected. For this task-agent pair, the path planner in PRIORITIZEDPLANS (Section IV-B) is invoked to find a valid set of plans. The path planner operates under a time limit to ensure that the computational budget in this online setting is not exceeded. If the path planner successfully finds a set of plans within the time limit, then the task-agent pair is prioritized and the plans are used; otherwise, the next task is considered. If no plans are found for any task, the process repeats with agents ranked by subsequent earliest arrival times, until the agent with the α^{th} earliest arrival time. We leave α as an algorithm parameters, where a small α reduces the chance to find a valid set of plans but also limits the worst-case computation time. Within the example set of our experiments, when no solution was found (line 14), it was sufficient to generate short random paths for all agents with SHORTRANDOMPLANS (line 15); this might not be generally true, though.

On line 13, $\text{ASSIGNTASKSTOAGENTS}$ orders tasks in descending priority and assigns to each task the agent with the earliest possible arrival time at the task vertex, when following its existing plan and then traversing the shortest path. An agent can only be assigned one task.

On line 17, SHORTPLANS uses a SIPP-based path planner

(Section IV-C) to compute a short plan for each agent *toward* its respective assigned task vertex. Agents are ordered in descending priority of their respectively assigned task. For each agent, the SIPP-based path planner is called to find an extension to the agent’s existing plan that takes the agent closer to its task vertex. If the agent reaches its task, then a new unassigned task is assigned to the agent. This is repeated until either the path planner is unable to find a plan or the agent’s plan extends beyond the planning horizon (i.e. the plan ends at some time $t \geq t_{plan} + \bar{t}$).

The next time CPLP should be called, t_{next} , is determined in GETNEXTPLANNINGTIME and returned on line 19. If all tasks in \mathcal{T}^U are scheduled for completion, the maximum end time across all agent plans is returned, as no further planning is needed. This ensures any remaining tasks in \mathcal{T}^U can be removed at that time. If not all tasks are scheduled for completion, further planning is required. In this case, the minimum end time of all agent plans is considered, as it is at that time when an agent becomes available for further planning. Since some agents’ plans may not have been extended beyond the horizon \bar{t} due to no valid plans being found, calling CPLP again is unlikely to yield solutions for these agents. Therefore, the returned value is the minimum end time of plans ending at $t \geq t_{plan} + \bar{t}$. If no such plan exists, $t_{plan} + \bar{t}$ is returned instead.

B. CCBS-based Path Planner

Unlike in CCBS, which is designed for the *offline* MAPF_R problem, CPLP plans only one agent to its target vertex while all other agents are idle. The CCBS algorithm does not natively handle idle agents, that is, agents without an assigned target vertex. In PRIORITIZEDPLANS on line 11 of Algorithm 1, a task-agent pair $\langle \tau, a \rangle$ is selected. For this pair, the CCBS-based path planner is called where the root of the high-level CCBS search is initialized with the shortest path from a ’s last location to v_τ . For all other agents, it assigns an infinite wait-action $\langle v', t', \infty \rangle$, where v' is the agent’s last location and t' the last planned time there. If a collision occurs between a moving agent and an idle agent a_i , we compute a new path for a_i using CSIPP. However, instead of finding a path to a target vertex (as in the original CSIPP), we search for a path with the earliest departure time away from the idle agent’s current vertex to another vertex where no other agent is scheduled to arrive at after. The constraint-tree node where the collision was detected is reinserted into the open set with the updated path for a_i and no additional constraints.

C. SIPP-based Path Planner

The SIPP implementation, with a few modifications, follows that of [12, 6]. Recall that this planner is used to find a short plan for agent a *toward*, but not necessarily *to*, its target vertex v_τ . Thus, in the A* search of SIPP, where states are pairs of vertices and safe intervals, the value of a state $\langle v, [t_1, t_2] \rangle$ is equal to $t + \text{SHORTESTPATHTIME}(v, v_{\tau_a})$ where $t \in [t_1, t_2]$ is the arrival time at v and $\text{SHORTESTPATHTIME}(v, v_{\tau_a})$ is the shortest

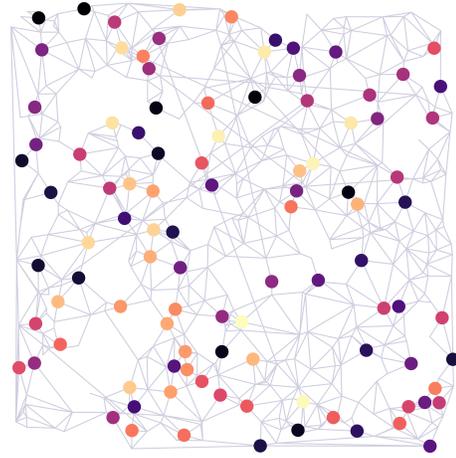


Fig. 1: A generated graph with 100 agents and 500 vertices. See our repository for animations of similar examples¹.

time to traverse from v to v_τ . If a is not assigned a task ($\tau = \text{None}$), then the value of the state is simply t . For a state $\langle v, [t_1, t_2] \rangle$ to be a goal state, it cannot be the root state and $t_2 = \infty$ such that the agent can remain there indefinitely.

Additionally, recall that no agent is planned to intersect with another agent’s last planned position at any time after it arrives there. This is not considered in the CCBS-based path planner since such collisions are inherently handled by adding constraints. In this path planner, however, this is handled when collecting successor states in the SIPP search; any edge and successor that intersects with an agent’s last planned position is removed.

V. EXPERIMENTAL EVALUATION

All experiments are run in Python 3.11 on a 2020 MacBook Air, Apple M1, 16 GB RAM, macOS Sequoia 15.3¹.

A. Instance Generation

Problem instances are generated from a given number of agents n_a and vertices per agent ρ , so that the resulting graph has $n_v = n_a \rho$ vertices. To do so, $\lceil (1 + \gamma_1) n_v \rceil$ 2D points are uniformly sampled from an l -by- l space, forming a preliminary set of vertices. Setting $l = 3\sqrt{n_v}$ results in all generated graphs having the same vertex density. The vertices’ Voronoi cells are then used to create edges between them; any two vertices with bordering cells are connected. To avoid overly uniform graphs, $\lceil \gamma_1 n_v \rceil$ sampled vertices are removed and the graph is ensured to remain connected. Finally, $\lceil \gamma_2 n_v \rceil$ sampled vertex pairs are connected to create crossing edges. In all experiments, undirected edges are used. The parameters γ_1 and γ_2 are set to 0.2 and 0.02, respectively. Agent starting positions are sampled from the set of vertices, ensuring that no two agents start in a collision. Agent radius and speed are both set to 1. Fig. 1 shows a generated graph with agent starting positions.

Although the lifelong MAPF conceptually never ends, for practical reasons, a finite task set is generated with each task being released within a finite time window $[0, T]$. Tasks are released at a rate of $r n_a$, where r is the release rate per

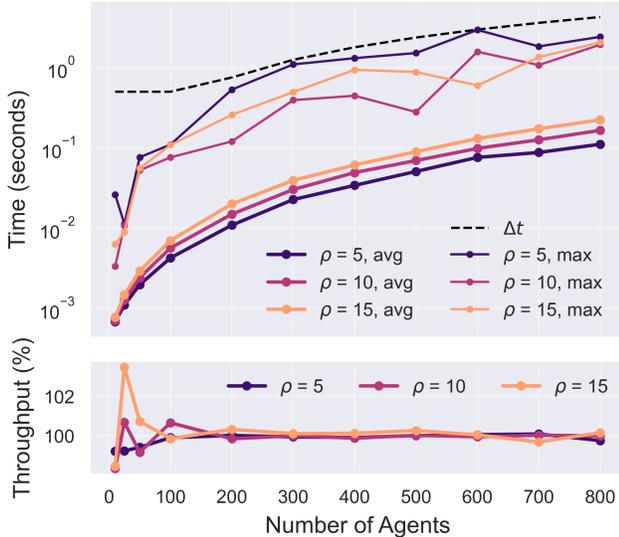


Fig. 2: Average and maximum computation time per call to CPLP and the average percentage of tasks completed versus released during the time interval $[100, 200]$, for each number of agents and vertices per agent ρ .

agent. The total number of tasks released is then $\lceil rn_a T \rceil$. We select $r = 0.05$ so that a task is released on average every 20 seconds per agent, and $T = 200$. Each respective task $\langle v, t \rangle$ is generated by uniformly sampling a vertex $v \in \mathcal{V}$ and time $t \in [0, T]$. Infeasible instances are manually removed, details are provided in the supplementary materials¹.

B. Results

We ran 15 instances for each $n_a \in [10, 25, 50, 100, 200, \dots, 800]$ and $\rho \in [5, 10, 15]$. Since CPLP must compute more plans as n_a grows, requiring more computation time, we find $\Delta t = \max(n_a^{1.25}, 500)$ ms to be sufficient. For PRIORITIZEDPLANS, we set $\alpha = 5$ and a time limit of 25 ms. For SHORTPLANS, we set horizon $\bar{t} = 1$ s.

Fig. 2 shows the average and maximum computation time per CPLP call, and the average percentage of tasks completed versus released during the time interval $[100, 200]$. Importantly, we see that no call to CPLP took more than Δt . Thus, with an appropriate Δt , CPLP can be used in practice without exceeding its computational time budget. The average computation time remains below 250 ms for up to 800 agents, which compared to \bar{t} shows that plans can be computed in shorter time than their durations. In practice, this means that non-stop movement of agents toward tasks can be maintained. The throughput exhibits some randomness due to the variability of task release times, particularly when n_a is low and fewer tasks are released overall. However, the percentage of completed tasks remains near 100%, suggesting that a higher throughput than the tested value ($r = 0.05$) can be maintained.

Fig. 3 shows the average time to perform graph pre-computations, which theoretically grows near $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}| \log |\mathcal{E}|)$, although appears near linear at

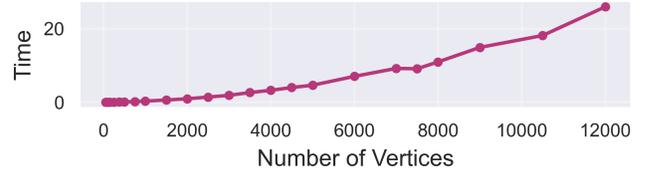


Fig. 3: The average graph pre-computation time, in minutes.

these numbers of vertices. For the largest graphs tested, containing 12000 vertices, pre-computations took around 35 minutes on average. These times are manageable in practical settings where agents move on a single, constant graph that only requires pre-computing once.

VI. DISCUSSION

The proposed CPLP demonstrates computation times within practical ranges for hundreds of agents on graphs with thousands of vertices. Additionally, the parameters \bar{t} and Δt allow for tuning the planner based on observed average and maximum computation times. In a real-world setting, Δt plays an important roll as it determines the time in advance that CPLP must be called, so that the planned paths are computed before they begin. Increasing Δt delays the system’s response to new tasks but provides more time to plan. In our experiments, a Δt under 5 s for the case of 800 agents is likely manageable in most real-world applications. An avenue for future work could be to explore adjusting both Δt and \bar{t} adaptively to observed computation times.

Although rarely, SHORTRANDOMPLANS was called in a number of instances when a path for a^* could not be found by PRIORITIZEDPLANS within the time limit. Shuffling agents randomly was experimentally sufficient for PRIORITIZEDPLANS to eventually find a path, however, this might not always work. In fact, SHORTRANDOMPLANS was called indefinitely on infeasible instances where no valid paths exist¹. Additionally, such inefficient behavior is likely not desirable in practice, nor necessary to shuffle *all* agents. This opens up multiple avenues for future research. For instance, the time limit on PRIORITIZEDPLANS could be temporarily raised, or only agents within the vicinity of the prioritized task-agent pair could be randomly moved instead of moving all agents. Providing theoretical guarantees for CPLP, similar to PIBT, likely hinges on the path planner in PRIORITIZEDPLANS. CCBS was used for this, however, recent findings [7] highlight important considerations regarding CCBS’s terminatability and optimality trade-offs. Thus, the exploration of alternative path planners is motivated.

Finally, the problem of lifelong MAPF as defined here may have limited practical use. The most common warehouse logistics problem would seem to concern transporting goods from one location to another, that is, *pickup-and-delivery* [10]. Lifelong MAPF planners can be adapted for pickup-and-delivery by forcing task assignments (as done in [8]), however, considering *all* task locations could result in better solutions. Future work could look into adapting

CPLP for these types of problems, where, for instance, the prioritized agent’s entire trajectory through all task locations could be planned while still only computing short plans for the remaining agents.

VII. CONCLUSIONS

This work presented CPLP, a fast, sub-optimal solver for the continuous-time lifelong MAPF problem with agent volumes. CPLP combines CCBS and SIPP-based path planning to compute agent plans online, while ensuring collision-free movement even in cases where CPLP is unable to compute plans within time constraints. Experimental results demonstrate computation times within practical ranges for up to 800 agents on graphs with up to 12000 vertices. These findings highlight the potential of CPLP for real-world applications such as warehouse automation and autonomous fleet coordination, where dynamic task assignment and collision-free movements are crucial. Ensuring collision-free movement under computational delays further enhances its robustness in practical scenarios.

In conclusion, this work contributes to the field of MAPF by offering a practical solution for the lifelong MAPF problem in continuous time with volumetric agents.

REFERENCES

- [1] Anton Andreychuk et al. “Multi-agent pathfinding with continuous time”. In: *Artificial Intelligence* 305 (2022), p. 103662.
- [2] Max Barer et al. “Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 5. 1. 2014, pp. 19–27.
- [3] Mehul Damani et al. “PRIMAL₂: Pathfinding via reinforcement and imitation multi-agent learning-lifelong”. In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 2666–2673.
- [4] Michael Erdmann and Tomas Lozano-Perez. “On multiple moving objects”. In: *Algorithmica* 2 (1987), pp. 477–521.
- [5] Peter E Hart, Nils J Nilsson, and Bertram Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.
- [6] Kazumi Kasaura, Mai Nishimura, and Ryo Yonetani. “Prioritized safe interval path planning for multi-agent pathfinding with continuous time on 2D roadmaps”. In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 10494–10501.
- [7] Andy Li, Zhe Chen, and Danial Harabor. “CBS with Continuous-Time Revisit”. In: *arXiv preprint arXiv:2501.07744* (2025).
- [8] Jiaoyang Li et al. “Lifelong multi-agent path finding in large-scale warehouses”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 35. 13. 2021, pp. 11272–11281.
- [9] Hang Ma. “Graph-based multi-robot path finding and planning”. In: *Current Robotics Reports* 3.3 (2022), pp. 77–84.
- [10] Hang Ma et al. “Lifelong multi-agent path finding for online pickup and delivery tasks”. In: *arXiv preprint arXiv:1705.10868* (2017).
- [11] Keisuke Okumura et al. “Priority inheritance with backtracking for iterative multi-agent path finding”. In: *Artificial Intelligence* 310 (2022), p. 103752.
- [12] Mike Phillips and Maxim Likhachev. “Sipp: Safe interval path planning for dynamic environments”. In: *2011 IEEE international conference on robotics and automation*. IEEE. 2011, pp. 5628–5635.
- [13] Francesco Popolizio et al. “Online Conflict-Free Scheduling of Fleets of Autonomous Mobile Robots”. In: *2024 IEEE 20th International Conference on Automation Science and Engineering (CASE)*. IEEE. 2024, pp. 3063–3068.
- [14] Guni Sharon et al. “Conflict-based search for optimal multi-agent pathfinding”. In: *Artificial intelligence* 219 (2015), pp. 40–66.
- [15] Guni Sharon et al. “The increasing cost tree search for optimal multi-agent pathfinding”. In: *Artificial intelligence* 195 (2013), pp. 470–495.
- [16] David Silver. “Cooperative pathfinding”. In: *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*. Vol. 1. 1. 2005, pp. 117–122.
- [17] Roni Stern et al. “Multi-agent pathfinding: Definitions, variants, and benchmarks”. In: *Proceedings of the International Symposium on Combinatorial Search*. Vol. 10. 1. 2019, pp. 151–158.
- [18] Pavel Surynek. “Multi-agent path finding with continuous time viewed through satisfiability modulo theories (SMT)”. In: *arXiv preprint arXiv:1903.09820* (2019).
- [19] Glenn Wagner and Howie Choset. “M*: A complete multirobot path planning algorithm with performance bounds”. In: *2011 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2011, pp. 3260–3267.
- [20] Thayne T Walker and Nathan R Sturtevant. “Collision detection for agents in multi-agent pathfinding”. In: *arXiv preprint arXiv:1908.09707* (2019).
- [21] Jingjin Yu and Steven LaValle. “Structure and intractability of optimal multi-robot path planning on graphs”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 27. 1. 2013, pp. 1443–1449.
- [22] Yue Zhang et al. “Planning and execution in multi-agent path finding: models and algorithms”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 34. 2024, pp. 707–715.