2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

CGA: Corridor Generating Algorithm for Multi-Agent Environments

Arseniy Pertzovsky[1], Roni Stern[1], Roie Zivan[1]

2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) | 979-8-3503-7770-5/24/$31.00 ©2024 IEEE | DOI: 10.1109/IROS58592.2024.10802519

***Abstract*— In this work, we consider path planning for a team of mobile agents where one agent must reach a given target as soon as possible and the others must accommodate to avoid collisions. We call this practical problem the *Single-Agent Corridor Generating* (SACG) problem and explore several algorithms for solving it. We propose two baseline algorithms based on existing Multi-Agent Path Finding (MAPF) algorithms and outline their limitations. Then, we present the Corridor Generating Algorithm (CGA), a fast and complete algorithm for solving SACG. CGA performs well compared to the baseline approaches. In addition, we show how CGA can be generalized to address the lifelong version of MAPF, where new goals appear over time.**

I. INTRODUCTION

Consider a team of mobile agents operating in a physical domain, e.g., a transportation system with autonomous vehicles picking up and delivering passengers to their destination. Suddenly, a medical emergency arises and one of the mobile agents must arrive as soon as possible to the nearest hospital. To do so, we must plan paths for all the agents such that the respective agent reaches the hospital as soon as possible. This work focuses on how to solve this type of multi-agent path planning problem, which we denote as the *Single-Agent Corridor Generating (SACG) problem*.

More generally, in a SACG problem, a single *main agent* aims to reach a given goal location, while the other agents in the environment are required to move away from the path of the main agent to facilitate its movement. The objective is to find a plan for all agents, such that the main agent reaches its goal, while the other agents avoid collisions with it and between themselves. SACG manifests in settings where there is a main agent whose importance significantly outweighs the other agents.

The SACG problem is situated between two well-studied problems: single-agent path finding and multi-agent path finding (MAPF) [30]. In single-agent path finding, the objective is to find a path in the graph from a given source vertex to a given destination vertex. In MAPF, we are given a set of source and destination pairs of vertices and the objective is to compose a path for each source-destination pair such that agents can follow these paths concurrently without collisions [30]. Single-agent path finding is usually solved with classical search algorithms such as Dijkstra's algorithm and A^* [10]. There is a variety of approaches to solve MAPF, including search-based [8], auction-based [1], and rule-based algorithms [24, 19]. In SACG, we only require a single main agent to reach its goal but a joint plan is required since other agents may need to move from their current position to enable the main agent to reach its goal.

The first contribution of this work is an efficient, polynomial-time procedure denoted as the Corridor Generating Algorithm (CGA) for solving a wide range of SACG problems. CGA is an iterative algorithm that plans a single step ahead. In every iteration, it moves the main agent towards its goal, pushing the other agents out of the way and ensuring the main agent's step does not block the escape route of the other agents. We prove that CGA is sound and complete under very mild restrictions. We also propose tailored versions of two well-known MAPF algorithms, PrP [28] and PIBT [24], for solving SACG. These will serve as baselines for comparison of the SACG algorithm that we introduce, i.e., CGA.

The second contribution of this work is to show how a SACG solver, in our case CGA, can significantly improve the performance of Lifelong Multi-Agent Path Finding (LMAFP) [17] in some cases with crowded grids. LMAPF is a practical generalization of MAPF in which agents are constantly provided with new goals upon completing tasks. Many existing LMAPF algorithms suffer from potential deadlocks and livelocks, especially when the agents occupy a densely populated region, e.g., the packing station in an autonomous warehouse. With some minor modifications, CGA offers a natural way to solve such cases by selecting a single, arbitrary agent, as the "main agent" and applying CGA to move it towards its goal. We analyze the resulting LMAPF algorithm and show that under certain conditions it ensures that all agents will eventually finish their tasks in a finite amount of time.

The third contribution of our work is a comprehensive experimental evaluation of CGA for solving SACG problems as well as CGA for solving LMAPF problems. Our results show that indeed CGA solves SACG problems very quickly and outperforms baseline algorithms in terms of success rate, and solution quality. Moreover, using CGA to solve LMAPF proves to be highly effective in some cases, outperforming baseline LMAPF algorithms in terms of throughput.

II. BACKGROUND AND RELATED WORKS

The term *corridor* has been used in the context of path planning in different ways. In some cases, it refers to imposing constraints over the path planner to guide it towards a desired location without enforcing a specific path [2, 25]. In other cases, it refers to pre-planning routes that are known to be safe [26, 29]. Our work is significantly different since we deal with a multi-agent scenario, where we can control both the main agent as well as the other agents.

[1]Agriculture, Biological, Cognitive Robotics Initiative, Ben-Gurion University of the Negev

The SACG problem we study in this work is similar to Single-Agent Path Finding (SAPF) in that the objective of both problems is to find a path for a single agent from its current location to a given goal location. However, since in SACG we plan for moving multiple agents, it is closer to the Multi-Agent Path Finding (MAPF) and Multi-robot Path Planning (MRPP) literatures. MAPF and MRPP are different names for the problem of finding paths for multiple agents such that each agent reaches its designated goal without collisions with the other agents.

Many algorithms exists for solving this problem including complete algorithms [33, 18, 5, 23, 22], incomplete algorithms [35, 21, 24], rule-based algorithms [24, 19], RL-based algorithms [7, 9, 3], and combinations of planning and learning [6]. Kottinger et al.[12], Wen et al. [34] and Saxena et al.[27] examine cases where agents have kinodynamic constraints.

For completeness, we briefly describe two MAPF algorithms, namely PrP [28] and PIBT [24] since they perform well and can be easily adapt to solve SACG problems.

Prioritized Planning (PrP), also known as Cooperative A* [28], is a popular algorithm that is widely used in many recent suboptimal MAPF papers [13, 14, 32, 36, 4]. In PrP, the agents plan one after another according to some predefined order. That is, when the i^{th} agent plans, it is constrained to avoid the paths chosen for all $i-1$ agents that have planned before it. The advantage of PrP is that it is simple to implement and has a small runtime. However, it is suboptimal and incomplete since its predefined priority ordering can sometimes result in solutions of bad quality or even fail to find any solutions for solvable MAPF instances.

PIBT [24] is a greedy MAPF algorithm, planning ahead a single time-step. In every iteration, it generates where every agents should move to in the next step, ensuring that at least one agents ends up getting closer to its goal. It does so by applying a recursive function that tries to move every agent towards its goal while using the Priority Inheritance techniques and a backtracking technique to break potential deadlocks. Under some conditions, PIBT is guaranteed to result in each agent reaching its goal, possibly perhaps at different times.

One of the contributions of this work is an adaptation of the proposed CGA algorithm to Lifelong MAPF (LMAPF) [15], that is tailored for online MAPF [31]. In LMAPF, agents continuously receive new tasks from a task assigner (the assigner is not part of our path-planning system). When an agent reaches its current goal it receives the next goal to travel to from the task assigner. The efficiency of a LMAPF system is measured in terms of throughput, i.e., the number of tasks that can be fulfilled in a given period of time [16, 20]. One of the popular approaches to solving LMAPF is to use the Rolling-Horizon Collision Resolution (RHCR) framework [16]. RHCR allows algorithms to repeatedly plan the next h steps (the horizon) based on the agents' current locations and goals. Agents then proceed $w \leq h$ steps, where w is a *window* parameter, and the process repeats.

III. SINGLE-AGENT CORRIDOR GENERATING

A Single-Agent Corridor Generating (SACG) problem is defined by a tuple $\langle G, s, g, k, \tilde{S} \rangle$, where $G = \langle V, E \rangle$ is a strongly connected undirected graph representing the possible locations the agents may occupy and the allowed transitions between them; s and g are vertices in G representing the start and goal locations of the *main agent*, respectively; k is the number of agents in addition to the main agent; and $\tilde{S} = (\tilde{S}_1, \ldots, \tilde{S}_k)$ is the vector of vertices in G representing the initial locations of other agents. A solution to a SACG is a set of paths $\pi_0, \ldots, \pi_k$, one per agent, where π_0 is the path of the main agent and (1) the paths do not conflict, (2) the path of every agent starts from its start location, and (3) the path of the main agent ends in g. Paths do not conflict if the agents following them do not occupy the same vertex or edge at the same time. Our objective in this work is to find solutions to SACG problem such that the path of the main agent is minimal. The example of SACG instance is shown in Fig. 1.

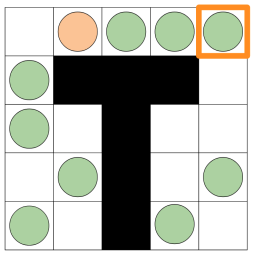


Fig. 1: An example of a SACG instance. The orange circle represents the *main agent*. The orange square is the goal g of the main agent. The rest are obstacles (black areas) and other agents (green circles) in a grid.

IV. BASELINE ALGORITHMS

In this section, we propose two baseline algorithms for solving SACG that are based on simple adaptations of existing MAPF algorithms, namely PrP [28] and PIBT [24].

The first algorithm that we call *PrP_SACG*, is a very simple adaptation of PrP. We set the agents' priorities such that the main agent is the higher-priority agent, and select the priorities of the other agents randomly. This will result in planning the path for the main agent first, ignoring all other agents, and having all other agents find a plan that avoids conflicts with the main agent's path (and the paths of all other higher priority agents). As in standard PrP, the choice of priorities for the other (non-main) agents is critical, and in some cases, the algorithm will fail to find any solution. To mitigate this to some extent, we incorporate the *random restart* mechanism, i.e., *PrP_SACG* is performed multiple times, choosing a different priority order for the other (non-main) agents, until finding a valid solution. *PrP_SACG* is a sound algorithm for SACG, and it runs in polynomial time. However, it is incomplete, in the sense that there are cases where regardless of the priorities chosen by the agent a solution may not be found. Such a case is presented in Fig.

2. The orange circle is the main agent that plans first. The other two green agents plan afterward. There is no order of planning by green agents that allows them to avoid collisions with the main one.

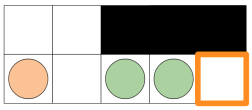


Fig. 2: Example of a failed instance for PrP_SACG. No order in the planning of green agents allows for the avoidance of collisions with the main (orange) agent.

The second baseline algorithm we propose is called *PIBT_SACG*. **PIBT_SACG** is an adaptation of PIBT [24] in which the main agent plans first in every iteration. In addition, in *PIBT_SACG* we limit the Priority Inheritance mechanism used in PIBT such that no other agent can inherit the priority higher than the main agent. This is needed to ensure that the main agent follows its shortest path. Like *PrP_SACG*, *PIBT_SACG* is also sound and runs in polynomial time. In addition, in a graph G where every pair of vertices has a simple cycle of length at least 3, PIBT ensures that every goal will eventually be visited [11] (Theorem 4). Consequently, in such graphs *PIBT_SACG* is a complete SACG algorithm, i.e., the main agent is guaranteed to reach the goal. The requirement that every pair of vertices has such a cycle is strong. For example, the SACG instance depicted in Fig. 2 does not satisfy this requirement. Indeed, PIBT_SACG fails to solve this instance since the main agent will be stuck in two steps, causing the deadlock. However, CGA will successfully handle the instance, by moving others out of the corridor to the goal location.

V. CORRIDOR GENERATING ALGORITHM

In this section, we present the Corridor Generating Algorithm (CGA), a polynomial-time algorithm for solving SACG that is complete even under a very limited set of assumptions. To explain CGA, we introduce the following terms.

Definition 1 (Separating Vertex): A vertex v in a graph G is called a separating vertex (SV) if removing v from G results in a graph with more connected components than G. A vertex that is not an SV is called a non-SV. Fig. 3 shows all the SV vertices in several grids.

Definition 2 (Corridor): A corridor in a graph $G = (V, E)$ is a path $(v_1, \ldots, v_n) \subseteq V$ in G such that all the vertices $v_2, \ldots, v_{n-1}$ are separating vertices.

That is, a corridor in this work is a path in which all vertices except the first and last must be SVs. The first and last vertices may or may not be SVs. A *trivial* corridor is a path comprising only a pair of vertices (v_1, v_2), where either v_1 or v_2 is not a separating vertex.

The CGA algorithm performs the following steps iteratively until the main agent reaches its goal: (1) select a corridor c starting from the current location of the main agent and ending in either the goal g or a non-separating vertex, (2) evacuate the other agents from c, and (3) move the main agent through c. Next, we describe how the corridor is selected in step (1) and how it is evacuated in step (2).

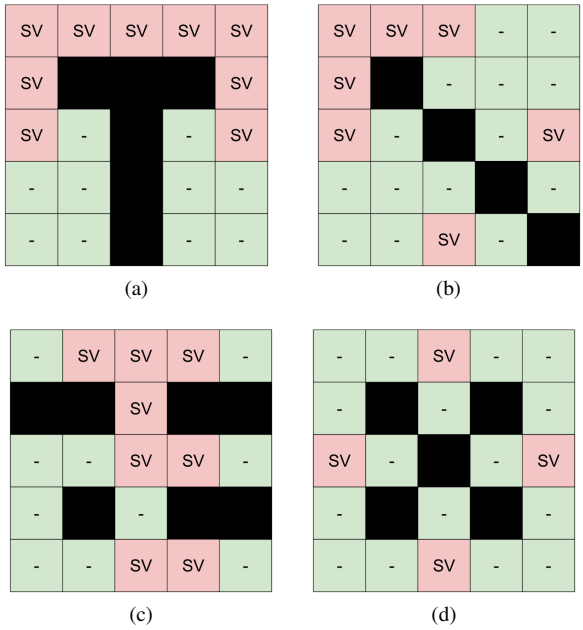
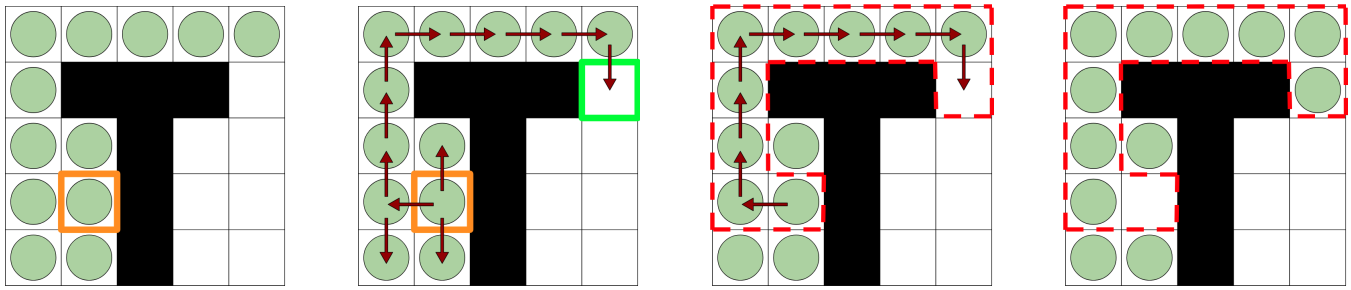


Fig. 3: Examples of the set of all the SVs for different graphs. The red cells marked by *"SV"* are the SVs.

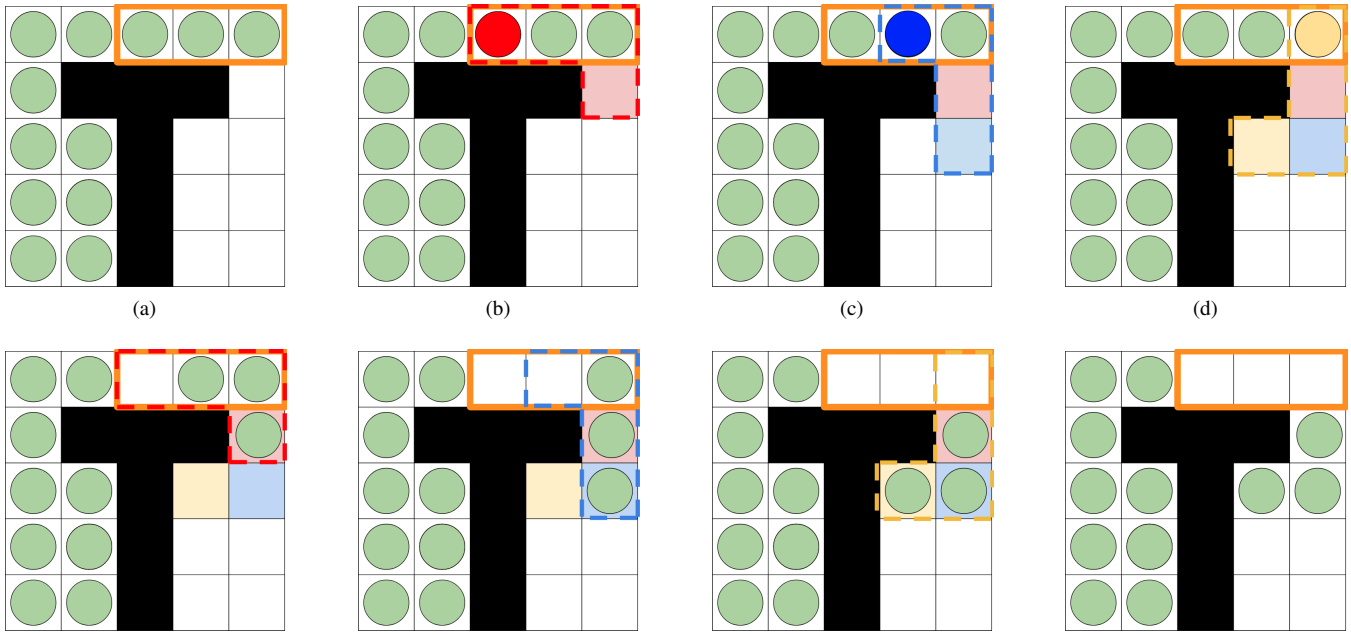
a) Corridor Selection (CorSel): CGA maintains an optimal path π^* from the agent's current location to the goal g. This path ignores the other agents and can be found using standard shortest path algorithms such as A*. In every iteration of CGA, it selects the maximal prefix of π^* that forms a corridor. Let c_{max} be this corridor. By definition, c_{max} ends either in g or in a non-separating vertex. If all the vertices in c_{max} are not occupied by any agent, the main agent simply moves along c_{max} and we continue to the next iteration of CGA. Otherwise, CGA employs the *Corridor Evacuation* (CorEvac) procedure described below to evacuate all the agents from c_{max} (except the main agent), before moving the main agent along c_{max}.

b) Corridor Evacuation (CorEvac): The CorEvac procedure iterates over the vertices in c_{max}, evacuating any agent that occupies them except the main agent. This is done as follows. Let v be a vertex in c_{max} that is currently occupied by a (non-main) agent and let V_e be the vertices in c_{max} we already iterated over. CorEvac finds the shortest path in G from v to a vertex v_f such that (1) v_f is not occupied by any agent, (2) v_f is not in V_e, and (3) the path to v_f does not go through the location of the main agent. Finding such a path can be done via a simple Breadth-First Search from v. We call this path the *Evacuation Path* (EP) of v. Now, all that is left to do is to move each of the agents in this EP, in order of their distance to v_f. Consequently, this step ends when v and V_e are no longer occupied by any agent, and the entire corridor evacuation procedure ends when c_{max} does not include any agent except the main agent, as required. Fig. 4 demonstrates an example of finding such an EP.



(a) (b) (c) (d)

Fig. 4: (a) A grid and a vertex v to evacuate (orange square); (b) run a BFS starting from v until a the first unoccupied vertex v_f is found (green square); (c) identify the EP from v to v_f (dotted red line); (d) move all agents in EP towards the unoccupied vertex v_f, starting from the neighbor of v_f.



(a) (b) (c) (d)

(e) (f) (g) (h)

Fig. 5: (a) The initial state with the *corridor* to evacuate; (b) - (d) the Evacuation Paths (EPs) are created one by one from the agents inside the corridor; (e) - (g) movement of agents along the EPs; (h) final state

The pseudo-code of CGA is presented in Algorithm 1. The input to the algorithm is the main agent a, the other agents $\tilde{A}$, and the graph G. First, CGA computes and stores the set of all SVs. This is easily done in polynomial time via dynamic programming. Then it runs the Algorithm 2 (described below) for each step of the algorithm (line 4), which selects the next corridor the main agent should traverse and computes appropriate EPs as needed. The next steps of all agents are updated accordingly (lines 5-6). If the last node of the updated path is the goal node, the algorithm halts (lines 7-8).

Algorithm 2 lists the pseudo-code of a step in CGA, *CGA-Step*. The input of CGA-Step is the main agent a, the other agents $\tilde{A}$, the graph G, and the set of SVs. The algorithm returns the dictionary of additional future steps per agent. The blue lines in Algorithm 2 are the adaptations for the LMAPF case, which will be discussed later. The algorithm starts with initializing the `next_steps` dictionary, the `EP` list, and the corridor for the main agent c_{max} (lines 3-4). If there are no agents in c_{max} the main agent just proceeds forward to line 21. Otherwise, we follow our CorEvac procedure, creating evacuation plans (EPs) for every agent in c_{max} and executing them (lines 9-20). Finally, the main agent moves after them (line 21).

VI. CGA IN LIFELONG MAPF

With some modifications, CGA can be particularly useful in the context of Lifelong MAPF. We refer to this modified version of CGA as *CGA(L)* and describe it in this section.

The pseudo-code of CGA(L) is presented in Algorithm 3. The algorithm follows the Prioritised Planning frame-

Algorithm 1 CGA

Input: $\langle a, \tilde{A}, G := (V, E) \rangle$
$SVS \leftarrow$ `create_svs(`G`)`
while $True$ **do**
  `next_steps` $\leftarrow$ `CGA-Step(`$a, \tilde{A}, G, SVS$`)`
  `add next_steps(`a`) to` a`.plan`
  `add next_steps(`$\tilde{A}$`) to` $\tilde{A}$`.plans`
  if a`.last_node is` a`.goal` **then**
    return a`.plan,` $\tilde{A}$`.plans`
  end if
end while

Algorithm 2 CGA-Step

Input: $\langle a, \tilde{A}, G := (V, E), SVS \rangle$
Output: `next_steps`
`next_steps` $\leftarrow$ `empty dictionary`
`ev_paths` $\leftarrow$ `empty list`
$c_{max} \leftarrow$ `get_corridor(`a, G, SVS`)`
if c_{max} `is` $\emptyset$ **then**
  return $\emptyset$
end if
`c_agents` $\leftarrow$ `agents in` c_{max}
for `c_a in c_agents` **do**
  `ev_path` $\leftarrow$ `get_EP(c_a,` c_{max}`,` $\tilde{A}, G$`)`
  if `ev_path is` $\emptyset$ **then**
    return $\emptyset$
  end if
  `add ev_path to ev_paths`
end for
for `ev_path in ev_paths` **do**
  `ev_agents` $\leftarrow$ `agents in ev_path`
  `next_steps(ev_agents)` $\leftarrow$
  `move(ev_agents, ev_path,`
  `next_steps(`$\tilde{A}$`))`
end for
`next_steps(`a`)` $\leftarrow$
`move(`a`,` c_{max}`, next_steps(`$\tilde{A}$`))`
return `next_steps`

Algorithm 3 CGA(L)

Input: $\langle A, G := (V, E), SVS, t \rangle$
`planned` $\leftarrow$ `agents with an active plan`
for $a \in A$ **do**
  If $a \in$ planned: **continue**
  $G' \leftarrow$ `prohibit(planned,` G`)`
  $A' \leftarrow A \setminus$ `planned,` a
  `next` $\leftarrow$ `CGA-Step(`a`,` A'`,` G'`,` SVS`)`
  if `next is` $\emptyset$ **then**
    continue
  end if
  `update(`A`, next)`
  `add agents in next to planned`
end for
`unplanned` $\leftarrow A \setminus$ `planned`
`For each agent` $a \in$ `unplanned: stay`
`Put finished agents at the end of` A

work, where the agents are assigned priorities and higher-priority agents plan before lower-priority agents. The input of CGA(L) is the graph G and a group of agents A where every agent $a_i \in A$ is associated with its current goal g_i and the plan it is currently following π_i. We refer to π_i as the *active plan* of agent i. Initially, the active plan of all agents is empty. CGA(L) is called in each time step t, outputting the next location each agent should go to and potentially updating the active plans of some agents.

When CGA(L) is called, it loops through the agents in order of their priorities (lines 3-13). If the agent a_i has an active plan it will follow it in the next time step, and CGA(L) continues to the next agent (line 4). Otherwise, CGA(L) generates an active plan for a_i and possibly other lower-priority agents by running a single iteration of CGA (Algorithm 2) for that agent (line 7), considering previously planned paths as obstacles (line 5). If this iteration of CGA succeeds, the algorithm updates the plans of agents (line 11) and puts the updated agent into the planned list (line 12). Otherwise, i.e., when the agent cannot find a corridor or cannot evacuate it (blue parts in Algorithm 2), then its active plan remains empty in this time step. Every agent with an empty active plan remains in its current location for the next time-step (lines 14-15). All agents that reach their goal location are placed at the end of the order of agents for the next iteration so that eventually every agent will enjoy being first in the order (line 16).

VII. THEORETICAL RESULTS

First, we analyze the runtime of CGA. The runtime complexity of the Corridor Selection is $O(|V| + |E|)$ as it simply runs a breadth-first search. Similarly, finding an EP for a single vertex requires $O(|V| + |E|)$. CorEvac searches for an EP at most $|A|$ times, and thus its runtime is $O(|A|(|V|+|E|))$. The number of iterations in CGA is at most the number of vertices in the graph. Thus, the runtime of CGA is $O(|A|^2(|V| + |E|))$.

Next, we analyze the completeness of CGA.

Lemma 1: In CGA, if the main agent is occupying a non-SV and the number of unoccupied vertices in a graph G is greater than or equal to the length of the longest corridor in G then the CorEvac procedure will successfully evacuate all agents from the corridor c_{max}.

Proof outline. Since the main agent is not occupying an SV, there exists a path from every vertex in c_{max} to any vertex in G that does not go through the main agent's location. As there are more unoccupied vertices than vertices in c_{max}, there exists an unoccupied vertex in G for every vertex in C_{max}. Thus, an EP will be found for every vertex in c_{max}, as required. □

Theorem 1: [Completeness] If the main agent is not occupying an SV and the number of unoccupied vertices in a graph

is equal to or greater than the length of the longest corridor in G then CGA is guaranteed to solve any solvable instance.

Proof outline. The Corridor Selection (CorSel) procedure in CGA ensures that the main agent moves from one non-SV vertex to another along an optimal path to the goal. Due to Lemma 1, CorEvac will successfully evacuate the corridor connecting these two non-SV vertices. Consequently, after a finite number of steps the main agent will reach its goal. □

Note that the requirement for completeness in Theorem 1 is strictly weaker than the requirement for PIBT_SACG. That is, every problem for which PIBT_SACG is complete also satisfies the requirement in Theorem 1 and thus will also be solvable by CGA. This is because if a graph G has a simple cycle of size 3 between every vertex, then G has no SV.

Theorem 2 (Reachability of CGA(L)): In CGA(L), if the number of unoccupied vertices is larger than the longest corridor then every agent is guaranteed to reach its next goal location in a finite amount of time.

Proof: Following Theorem 1, the agent with highest priority will reach its goal location in a finite amount of steps, as it applies CGA without any restrictions. CGA(L) assigns the lowest priority to agents that has reached their goals. Thus, eventually every agent will be the highest priority agent and reach its goal. □

VIII. EMPIRICAL RESULTS

We conducted two sets of experimental evaluations: one for solving SACG problems and one for solving LMAPF problems. All algorithms were implemented in Python and ran on a MacBook Air with an Apple M1 chip and 8GB of RAM.

A. SACG Experiments

This set of experiments was performed on four different grids from the MAPF benchmark [30]: *empty-32-32*, *random-32-32-20*, *maze-32-32-4*, and *room-32-32-4*, as they present different levels of difficulty. The grids are visualized in Fig. 6. The number of agents varied from 100 to 1000. SACG problems were created as follows. All agents were placed in random start locations in the given grid, and a goal location was selected randomly for the main agent. 25 random instances were generated in this way for every number of agents and grid.

To solve the generated SACG instances, we implemented CGA and the two baselines, *PrP_SACG* and *PIBT_SACG*. For *PrP_SACG* we allowed 100 random restarts before declaring that no solution has been found. We considered the following standard metrics for comparison: success rate and sum-of-costs. *Success rate* is the number of SACG instances out of all the algorithms succeeded in solving. An additional metric to consider is to examine the *Sum-of-costs* (SoC) which is a sum of all movements of agents that were needed to solve SACG. SoC embodies the cost of a solution that we prefer to minimize.

Fig. 6 shows the success rate results as a function of the number of agents. As can be seen, on all grids CGA solved all instances while PIBT_SACG and PrP_SACG solved a fraction of instances with a larger number of agents. For instance, in a *room* grid with 600 agents, PrP_SACG could not solve any instance in our given time limit, PIBT_SACG solved around 90% and CGA solved all 25 instances.

Fig. 7 compares the SoC obtained by each algorithm. The x-axis is the number of solved instances. The y-axis is the solution quality of a given instance (lower is better). We can clearly see that CGA outperforms other algorithms. For example, in a *random* some instances executed close to 2500 moves, while the highest number for CGA is under 500.

TABLE I: LMAPF: Throughput

Fig. 8	Alg.	50	75	100	125	150	175	200
(a)	PrP	12.92	6.28	0.32	-	-	-	-
	PIBT	**79.84**	58.24	13.56	-	-	-	-
	CGA(L)	60.00	**61.24**	**33.00**	-	-	-	-
(b)	PrP	8.40	8.64	6.00	1.96	0.36	-	-
	PIBT	**67.00**	**63.52**	**56.32**	33.36	13.40	-	-
	CGA(L)	46.32	44.76	46.12	**45.92**	**27.28**	-	-
(c)	PrP	22.80	17.76	11.48	3.84	1.60	0.08	-
	PIBT	**122.40**	**133.92**	**133.80**	**130.92**	**83.16**	14.84	-
	CGA(L)	94.92	98.72	99.64	92.04	80.12	**27.68**	-
(d)	PrP	34.84	36.48	25.48	10.56	3.88	0.68	0.20
	PIBT	**146.00**	**176.44**	**187.08**	**205.88**	**207.12**	**165.80**	78.28
	CGA(L)	107.00	122.32	134.56	134.00	142.40	137.00	**107.92**

B. LMAPF Experiments

Next, we conducted a set of LMAPF experiments. The main metric for comparison in the LMAPF experiments is the *throughput* obtained after 100 iterations. *Throughput* is measured as the sum of agents' reached goals during the execution.

Initially, CGA was developed to solve SACG, not LMAPF. Indeed, in our experiments on MAPF benchmark [30] PIBT significantly outperforms CGA(L) in terms of throughput. The reason for this is that PIBT aims to push every agent towards its goal even when the agent is required to free the way for higher priority agents. Unlike PIBT, CGA does not consider the preferences of other agents except the *main* one, while moving them away from corridors. Nevertheless, there are cases where PIBT cannot find a solution, as shown in Fig. 2. To demonstrate the performance of CGA(L) in such cases compared to PIBT, we created four small 15×15 grids of different arrangements of rooms with narrow single entries. These grids are shown in Fig. 8. The number of agents used in our experiments varied from 50 up to 200, depending on the capacity of a grid. Random goal locations were created for every agent that reached its goal. As in SACG experiments, we executed 25 random instances per every number of agents and grid. Consequently, the baseline algorithms for comparison are PrP and PIBT. PrP works in a RHCR framework with $w = h = 5$, which we found the best. PIBT has no modifications.

Table I shows throughput results per grid (4, 8, 6, 2 rooms), algorithm, and number of agents. From the table, it is clear that with an increasing number of agents the advantage transfers from PIBT to CGA(L). For example, in *15-15-four-rooms* grid, PIBT succeeds in completing 13.25 goals on average, while CGA(L) completes 33, which is 2.4 times better. One of the future directions of research is to try to

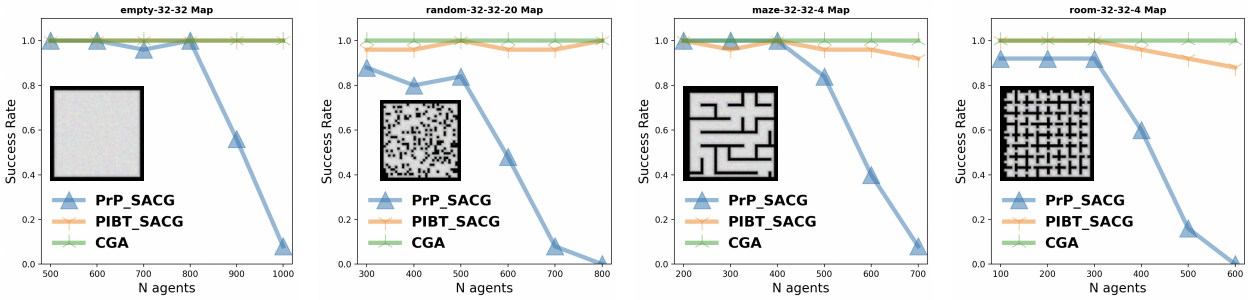


Fig. 6: SACG: Success Rate

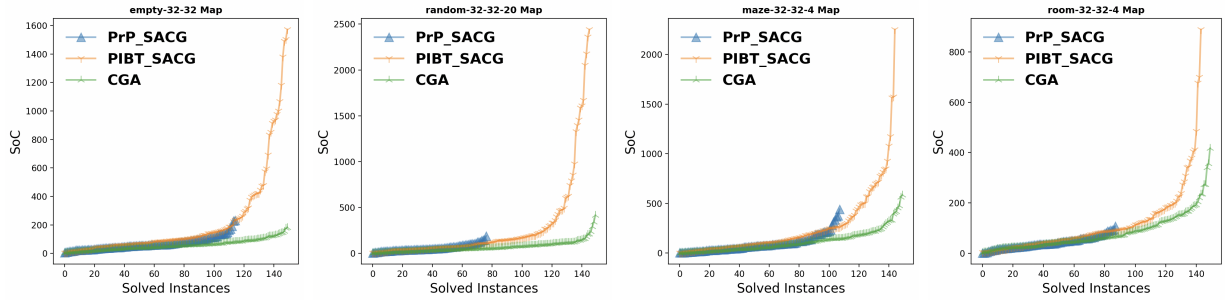
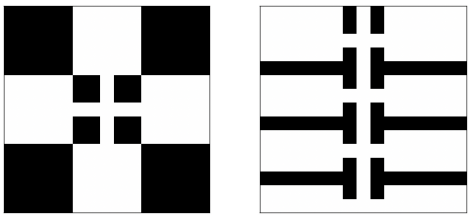


Fig. 7: SACG: Sum-of-Costs (SoC). x-axis is a number of solved instances, and y-axis is the SoC of a given instance. The instances per algorithm are ordered from lowest to highest regarding SoC.



(a) 15-15-four-rooms (b) 15-15-eight-rooms

(c) 15-15-six-rooms (d) 15-15-two-rooms

Fig. 8: 15×15 grids for LMAPF experiments.

combine the advantages of both PIBT and CGA into a single algorithm.

IX. CONCLUSIONS AND FUTURE WORK

In this work, we introduced the Single-Agent Corridor Generating (SACG) problem, which is a multi-agent path planning problem in which a single main agent must move to its goal location and the other agents move to avoid colliding with it. We proposed two baseline algorithms for solving SACG problems that are based on existing Multi-Agent Path Finding (MAPF) algorithms. Then, we proposed the Corridor Generating Algorithm (CGA), a dedicated algorithm for solving SACG problems. CGA runs in polynomial time and is complete under certain conditions. Experimentally, we showed that CGA solves SACG problems more efficiently than the baseline approaches in terms of success rate and solution cost. Then, we introduced CGA(L), a Lifelong MAPF (LMAPF) algorithm that is based on CGA. CGA(L) is fast to compute and we show experimentally that it can be more effective than baseline LMAPF algorithms in densely populated scenarios. Yet, in sparser scenarios CGA(L) is outperformed by existing LMAPF algorithms. Future work can integrate ideas from existing LMAPF algorithms and CGA(L). Another direction for future work is to develop distributed methods for solving SACG.

ACKNOWLEDGEMENTS

This research was partly supported by the Helmsley Charitable Trust through the Agricultural, Biological and Cognitive Robotics Initiative and by the Marcus Endowment Fund both at Ben-Gurion University of the Negev, and the ISF fund #1238/23.

REFERENCES

[1] Ofra Amir, Guni Sharon, and Roni Stern. “Multi-agent pathfinding as a combinatorial auction”. In: *AAAI*. Vol. 29. 1. 2015.

[2] Zack Butler. “Corridor planning for natural agents”. In: *ICRA*. 2006, pp. 499–504.

[3] Eduardo Candela et al. “Transferring multi-agent reinforcement learning policies for autonomous driving using sim-to-real”. In: *IROS*. IEEE. 2022, pp. 8814–8820.
[4] Shao-Hung Chan et al. “Greedy Priority-Based Search for Suboptimal Multi-Agent Path Finding”. In: *SoCS*. 2023, pp. 11–19.
[5] Boris De Wilde, Adriaan W Ter Mors, and Cees Witteveen. “Push and rotate: cooperative multi-agent path planning”. In: *International conference on Autonomous agents and multi-agent systems*. 2013, pp. 87–94.
[6] Stepan Dergachev, Konstantin Yakovlev, and Ryhor Prakapovich. “A combination of Theta*, ORCA and push and rotate for multi-agent navigation”. In: *International Conference on Interactive Collaborative Robotics*. Springer. 2020, pp. 55–66.
[7] Tingxiang Fan et al. “Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios”. In: *IJRR* 39.7 (2020), pp. 856–892.
[8] Ariel Felner et al. “Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges”. In: *SoCS*. 2017.
[9] Huifeng Guan et al. “AB-Mapper: Attention and BicNet based Multi-agent Path Planning for Dynamic Environment”. In: *IROS*. IEEE. 2022, pp. 13799–13806.
[10] 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.
[11] Kazumi Kasaura, Mai Nishimura, and Ryo Yonetani. “Prioritized Safe Interval Path Planning for Multi-Agent Pathfinding With Continuous Time on 2D Roadmaps”. In: *IEEE RAL* 7.4 (2022), pp. 10494–10501.
[12] Justin Kottinger, Shaull Almagor, and Morteza Lahijanian. “Conflict-based search for multi-robot motion planning with kinodynamic constraints”. In: *IROS*. IEEE. 2022, pp. 13494–13499.
[13] Florian Laurent et al. “Flatland Competition 2020: MAPF and MARL for Efficient Train Coordination on a Grid World”. In: *NeurIPS*. 2021.
[14] Christopher Leet, Jiaoyang Li, and Sven Koenig. “Shard Systems: Scalable, Robust and Persistent Multi-Agent Path Finding with Performance Guarantees”. In: *AAAI* (2022), pp. 9386–9395.
[15] Jiaoyang Li et al. “Anytime multi-agent path finding via large neighborhood search”. In: *IJCAI*. 2021.
[16] Jiaoyang Li et al. “Lifelong Multi-Agent Path Finding in Large-Scale Warehouses”. In: *AAAI* (May 2021).
[17] Jiaoyang Li et al. “MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search”. In: *AAAI*. 2022.
[18] Ryan Luna and Kostas E Bekris. “Efficient and complete centralized multi-robot path planning”. In: *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2011, pp. 3268–3275.
[19] Ryan J Luna and Kostas E Bekris. “Push and swap: Fast cooperative path-finding with completeness guarantees”. In: *IJCAI*. 2011.
[20] Jonathan Morag, Roni Stern, and Ariel Felner. “Adapting to Planning Failures in Lifelong Multi-Agent Path Finding”. In: *SoCS*. 2023.
[21] Jonathan Morag, Roni Stern, and Ariel Felner. “Adapting to Planning Failures in Lifelong Multi-Agent Path Finding”. In: *SoCS*. Vol. 16. 1. 2023, pp. 47–55.
[22] Keisuke Okumura. “Improving LaCAM for scalable eventually optimal multi-agent pathfinding”. In: *International Joint Conference on Artificial Intelligence*. 2023, pp. 243–251.
[23] Keisuke Okumura. “Lacam: Search-based algorithm for quick multi-agent pathfinding”. In: *AAAI*. 2023, pp. 11655–11662.
[24] Keisuke Okumura et al. “Priority inheritance with backtracking for iterative multi-agent path finding”. In: *Artificial Intelligence* 310 (2022), p. 103752.
[25] Yunfan Ren et al. “Bubble planner: Planning high-speed smooth quadrotor trajectories using receding corridors”. In: *IROS*. 2022, pp. 6332–6339.
[26] K Sangho, G Betsy, and S Shashi. “Evacuation route planning: scalable heuristics”. In: *ACM international symposium on Advances in geographic information systems*. 2007.
[27] Dhruv Saxena and Maxim Likhachev. “Planning for Manipulation among Movable Objects: Deciding Which Objects Go Where, in What Order, and How”. In: *arXiv preprint arXiv:2303.13385* (2023).
[28] David Silver. “Cooperative Pathfinding”. In: *AIIDE*. 2005.
[29] Jakub Sláma, Petr Váňa, and Jan Faigl. “Generating Safe Corridors Roadmap for Urban Air Mobility”. In: *IROS*. IEEE. 2022, pp. 11866–11871.
[30] Roni Stern et al. “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks”. In: *SoCS*. 2019, pp. 151–158.
[31] Jiří Švancara et al. “Online multi-agent pathfinding”. In: *AAAI*. 2019.
[32] Sumanth Varambally, Jiaoyang Li, and Sven Koenig. “Which MAPF Model Works Best for Automated Warehousing?” In: *SoCS*. 2022.
[33] KC Wang and Adi Botea. “MAPP: a scalable multi-agent path planning algorithm with tractability and completeness guarantees”. In: *JAIR* (2011), pp. 55–90.
[34] Licheng Wen, Yong Liu, and Hongliang Li. “CL-MAPF: Multi-agent path finding for car-like robots with kinematic and spatiotemporal constraints”. In: *Robotics and Autonomous Systems* 150 (2022).
[35] Qinghong Xu et al. “Multi-goal multi-agent pickup and delivery”. In: *IROS*. IEEE. 2022, pp. 9964–9971.
[36] Shuyang Zhang et al. “Learning a Priority Ordering for Prioritized Planning in Multi-Agent Path Finding”. In: *SoCS*. 2022.