Enhancing Lifelong Multi-Agent Path-finding by Using Artificial Potential Fields

Arseniy Pertzovsky, Roni Stern, Roie Zivan, Ariel Felner

Abstract

Artificial Potential Fields (APFs) is a well-known method for deploying mobile agents. We explore the use of APFs to solve Multi-Agent Path Finding (MAPF) and Lifelong MAPF (LMAPF) problems. In these problems, a team of agents must move to their goal locations without collisions, whereas in LMAPF new goals are generated upon arrival. We show that direct usage of APFs is ineffective for solving challenging cases. As an alternative, we incorporate APFs into Temporal A* (TA*+APF) and SIPPS (SIPPS+APF), which are single-agent path-finding algorithms that are used by common MAPF algorithms. We implemented TA*+APF and SIPSS+APF within standard MAPF algorithms and evaluated them experimentally. Our results show that while APFs are not beneficial when solving MAPF problems, they are very effective when solving LMAPF, yielding up to a 7-fold increase in overall system throughput.

1 Introduction

Artificial Potential Fields (APFs) (Khatib 1986) is a physics-inspired approach for the deployment of mobile agents in an environment with obstacles. The standard way to use APFs is to consider each agent as a particle that is moved based on attraction and repulsion forces that are applied to it. Typically, obstacles and other agents exert repulsive forces while the goal applies an attractive force (Koren and Borenstein 1991). APFs have been used to solve many motion planning problems (Mac et al. 2016; Hagelback and Johansson 2009; Daily and Bevly 2008), with successful applications in obstacle avoidance of an unmanned aircraft (Rezaee and Abdollahi 2012), collision avoidance systems for automated vehicles (Wahid et al. 2017) and more (Mac et al. 2016). In this work, we explore how APFs can be used to solve Multi-agent Pathfinding (MAPF) problems.

MAPF is the problem of finding a set of paths for a group of agents such that if each agent follows its path it ends up in its goal location without colliding with any other agent (Stern et al. 2019). Instances of MAPF exist in robotics (Barták et al. 2019), automated warehouses (Wurman, D'Andrea, and Mountz 2008; Salzman and Stern 2020), digital entertainment (Ma et al. 2017) and more (Morris et al. 2016). Solving MAPF problems can be computationally difficult since all agents may move concurrently, and thus the number of actions to consider at every point in time is exponential in the number of agents. Indeed, for common optimization criteria finding an optimal solution for a given MAPF problem is NP-hard (Yu and LaValle 2013; Surynek 2010). Popular optimal and suboptimal algorithms, such as Prioritized Planning (PrP) (Silver 2005), Conflict-Based Search (CBS) (Sharon et al. 2015), and Large Neighborhood Search (LNS) (Li et al. 2021a), mitigate to some extent the combinatorial challenge of planning for multiple agents by path planning for each agent individually and imposing different forms of constraints to avoid collisions.

Still, using existing algorithms often results in congestion occurring in areas of the environment that are shared by the shortest paths of multiple agents. This congestion problem is exacerbated when solving MAPF problems in a *lifelong* setting (LMAPF) (Li et al. 2021a), which is an important type of online MAPF (Švancara et al. 2019) in which an agent receives a new goal location whenever it reaches its current goal location. When solving LMAPF, congested areas tend to get more and more congested over time, resulting in diminishing overall system efficiency. Using APFs is a natural approach to encourage agents to avoid congested areas, adding repulsion forces not only for avoiding obstacles but also for avoiding the paths of other agents.

We begin our investigation of using APFs for solving MAPF problems by following a simple, myopic approach: each agent plans its next move based on the forces of APFs, where an agent's goal has an APF that attracts it and all other agents have an APF that repulses it. The resulting MAPF algorithm (Direct APF) compares poorly with existing MAPF algorithms on standard MAPF benchmarks. Apparently, in such cases, relying on attraction and repulsion forces is too myopic, and longer-horizon path planning is needed.

As an alternative, we explored using APFs within existing MAPF solvers. First, we consider PIBT (Okumura et al. 2022), LaCAM (Okumura 2023a), and LaCAM* (Okumura 2023b), which are MAPF algorithms that search the space of *configurations*, where a configuration is a set of all agents' locations in a given time-step. We introduce APFs to these algorithms by incorporating APFs into the process of configurations' creation.

Second, we consider using single-agent pathfinding algorithms to plan for the individual agents under different types of constraints. Temporal A* (TA*) (Silver 2005) and SIPPS (Li et al. 2022) are the most prevalent examples of such algorithms. We propose modified versions of these algorithms, called TA*+APF and SIPPS+APF, that change the cost of agents' actions to consider the APFs created by other agents who have already planned their paths. Consequently, the path returned for an agent does not optimize only for allowing the agent a fast arrival to its goal, but also for avoiding areas congested by other agents.

We evaluated our approaches experimentally on a range of standard MAPF benchmark problems. The results show that our APF-augmented algorithms have marginal merit when solving classical MAPF problems. Yet they are very effective when solving LMAPF, yielding up to a 7-fold increase in overall system throughput.

2 Definition and Background

A classical MAPF problem is defined by a tuple $\langle k, G, s, g\rangle$ where k is the number of agents, $G = (V, E)$ represents an undirected graph, $s : [1, ..., k] \rightarrow V$ maps an agent to a start vertex, and $g : [1, ..., k] \rightarrow V$ maps an agent to a goal vertex. Time is discretized into time-steps. In every time-step, each agent occupies a single vertex and performs a single *action*. An action is a function $a : V \rightarrow V$ such that $a(v) = v'$. There are two types of actions: $wait$ and $move$. The result of a $wait$ action at some time-step is that the agent will stay at the same vertex v at the next time-step. The result of a $move$ action at some time-step is that the agent will move to an adjacent vertex v' in the graph (i.e., $(v, v') \in E$). A *single-agent path* for agent a_i, denoted π_i, is a sequence of actions π_i that is applicable starting from s_i and ends up in g_i. A solution to a MAPF is a set of single-agent paths $\pi = \{\pi_1, \ldots, \pi_k\}$, one per agent, that do not *conflict*. We consider two types of conflicts: *vertex conflict* and *swapping conflict*. Two single-agent paths have a vertex conflict if they aim to occupy the same vertex at the same time, and a swapping conflict if they aim to traverse the same edge at the same time from opposing directions. The *sum-of-costs* (SOC) of a MAPF solution π is the sum over the lengths of its constituent single-agent paths. It is often desirable to find MAPF solutions that have minimal SOC.

Some MAPF algorithms are *complete and SOC-optimal*, i.e., guaranteed to return a valid optimal solution, if such a solution exists, w.r.t. its SOC. Primary examples of such complete and optimal MAPF algorithms are CBS (Sharon et al. 2015), ICTS (Sharon et al. 2013), A*+OD+ID (Standley 2010), M* (Wagner and Choset 2011), BCP (Lam et al. 2022), and SAT-MDD (Surynek et al. 2016). Other MAPF algorithms are complete but suboptimal, i.e., they are guaranteed to find a solution if such exists but its cost may not be optimal. Examples of algorithms of this family include Push-and-Swap (Luna and Bekris 2011) and Kornhauser's algorithm (1984). In this work, we focus on incomplete MAPF algorithms, which are very common in real-world MAPF applications and are often much faster and can scale better than any complete algorithm (Li et al. 2021a; Leet, Li, and Koenig 2022a). We present some examples of incomplete algorithms next.

Prioritized Planning (PrP) and LNS2 PrP (Bennewitz, Burgard, and Thrun 2001) is a simple yet very popular MAPF algorithm to grasp and implement (Laurent et al. 2021; Leet, Li, and Koenig 2022b; Varambally, Li, and Koenig 2022; Zhang et al. 2022; Chan et al. 2023). In PrP, the agents are first ordered, and they plan sequentially based on this order. When the i^{th} agent plans, it is constrained to avoid the paths chosen for all $i - 1$ agents that have planned before it. PrP is agnostic to how a single-agent path is found for each agent that satisfies these constraints. Such single-agent paths are found by a low-level search algorithm such as Temporal A* (TA*) or SIPPS, which are described below. PrP is simple and fast but might not be very effective in very dense environments due to possible deadlocks.

Large Neighborhood Search (LNS2) (Li et al. 2022) is an incomplete MAPF algorithm that aims to overcome some of the pitfalls of PrP. LNS2 starts by assigning paths to the agents even though they might conflict. LNS2 then applies a *repair* procedure, where PrP is used to replan for a subset of agents, aiming to minimize conflicts with other agents. LNS2 repeats this repair procedure until the resulting solution is conflict-free.

Temporal A*, SIPP, and SIPPS Temporal A*, SIPP, and SIPPS are algorithms that plan a path for a single agent under constraints. They are used in many MAPF algorithms such as PrP and LNS2 described above but also by CBS (Sharon et al. 2015), PBS (Ma et al. 2019).

Temporal A*(TA*) uses A* on the *spatio-temporal state-space* in which a state is a pair (v, t) where v is a vertex in G and t is a time-step. TA* receives the start and goal locations of an agent, along with a list of vertex- and edge constraints. A vertex constraint is a pair (v, t) specifying that the agent must not plan to be at v at time-step t. An edge constraint (e, t) specifies that the agent must not traverse edge e at time t in either direction. TA* returns the shortest paths that avoids the given constraints.

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) improves upon TA* by dividing time into intervals instead of timesteps to represent the time dimension. SIPP performs an A* search on a state space where each state is defined by a vertex and a safe (time) interval, representing that the agent occupies that vertex in this time interval, and that this does not violate any given constraint.

Safe Interval Path Planning with Soft Constraints (SIPPS) (Li et al. 2022) generalizes SIPP to accept both *hard* and *soft* constraints. The hard constraints are to avoid conflicts with the other agents selected for replanning, and the soft constraint is to avoid conflicts with all other agents. It returns a path that does not violate any hard constraints and minimizes the number of soft constraints violated. SIPPS is designed to run within LNS2.

PIBT and LaCAM PIBT (Okumura et al. 2022) is a state-of-the-art MAPF algorithm that solve MAPF problems by searching in the *configuration space*. A configuration here is a vector representing the agents' locations in some time-step. PIBT searched in this space in a greedy and myopic manner, starting from the initial configuration of the agents and iteratively generating a configuration for the next time-step until reaching a configuration where all agents are at their goals. PIBT generates configurations recursively, moving every agent toward its goal while avoiding conflicts with previously planned agents. To avoid deadlocks, PIBT employs priority inheritance and backtracking techniques.

PIBT is very efficient computationally but is incomplete since it searches greedily in the configuration space. LaCAM (Okumura 2023a) also searches the configuration space using a similar approach to generate configurations. To ensure completeness, LaCAM adds constraints to the configuration generation process to ensure it eventually can reach every possible configuration.

2.1 Lifelong MAPF (LMAPF)

Lifelong MAPF (LMAPF) (Li et al. 2021a) is an important type of *online MAPF* (Švancara et al. 2019). In LMAPF agents continuously receive new tasks from a task assigner (outside of our path-planning system). When an agent reaches its current goal it receives a new goal to travel to from the task assigned. The task in classical MAPF is to minimize the SOC or makespan, as explained above. Since solving LMAPF involves solving multiple MAPF problems, the efficiency of LMAPF algorithms is usually measured by the overall system *throughput* achieved when using them, which is measured by the number of tasks fulfilled in a given period of time (Li et al. 2021b; Morag, Stern, and Felner 2023).

The Rolling-Horizon Collision Resolution framework (RHCR) (Li et al. 2021b), solves MAPF problems by repeatedly planning the next k steps (k is the *horizon* parameter) to execute based on the agents' current locations and goals. Agents then move $w \leq k$ steps, where w is a *window* parameter, and the process repeats. RHCR is particularly useful in LMAPF as new goals are generated on the fly and there is no point in planning long paths.

Congestion avoidance techniques Several approaches were proposed to avoid the emergence of congested areas in LMAPF. Skrynnik et al. (Skrynnik et al. 2024) developed a reinforcement learning (RL) approach for a partially observed MAPF variant. Shuai et al. (Han and Yu 2020) used database heuristics tailored for every map separately. Chen et al. (Chen et al. 2024) exploit time-independent routes to compose a heuristic for the PIBT algorithm that predicts future expected congestion. Finally, Hen et al. (Han and Yu 2022) present a Space Utilization Optimization (SUO) heuristic for clever tie-breaking and allows to preserve optimality guarantees. In our work, we do not restrict observability, there are no requirements for learning procedures or optimality guarantees and we are not tied to any specific algorithm. Therefore, we do not compare our work with these approaches.

2.2 Artificial Potential Fields

A majority of the work that was done about APFs is about single-agent motion planning in continuous spaces (Barraquand, Langlois, and Latombe 1992; Fox, Burgard, and Thrun 1997; Wang and Chirikjian 2000; Daily and Bevly 2008; Mac et al. 2016; Zhang, Lin, and Chen 2018; Shin and Kim 2021). Vadakkepat et al. (2000) studied continuous spaces with moving obstacles. Hagelbäck et al. (2008; 2009) used APFs in real-time strategy games to avoid obstacles. Among those works that dealt with multiple agents: Liu et al. (2017) used APFs in a problem of formation control; Dinh et al. (2016) introduced Delegate MAS that simulated food foraging behavior in ant colonies; and many works (Semnani et al. 2020; Fan et al. 2020; Agrawal et al. 2022; Dergachev and Yakovlev 2021) introduced algorithms for reinforcement learning (RL) tasks that incorporated together with Optimal Reciprocal Collision Avoidance (ORCA) (Van den Berg et al. 2011) or Force-based motion planning (FMP) (Semnani et al. 2020) in them. Some RL environments include the APFs as part of the environment (Bettini et al. 2022). To the best of our knowledge, we are the first to apply APFs to solve MAPF problems.

3 Direct Artificial Potential Fields

In this section, we present the *Direct APF* (DAPF) algorithm. DAPF embodies a direct application of APFs to solve MAPF problems. In DAPF, at every time-step each agent a_i sums a set of repulsion and attraction "forces" and moves in the next time-step in the corresponding direction. These "forces" consist of repulsion from the locations of all other agents and attraction to the location of the goal of a_i. We experimented with different functions for these repulsion and attraction "forces" in our MAPF context. The following functions worked reasonably in our experiments.

Repulsion forces For every agent a_i, we create a repulsion function APF_i based on its current location v_i.

$$APF_i(v) = \begin{cases} 0 & \text{if } d(v_i, v) > d_{max} \\ w \cdot \gamma^{-d(v_i,v)} & \text{otherwise} \end{cases} \tag{1}$$

where d_{max}, γ, and w are predefined parameters and $d(v_i, v)$ is the minimal distance between v and v_i.[1] The w parameter controls the strength of the repulsion, γ controls the rate of decay, i.e., how fast its intensity of the repulsion declines while moving away from its source, and d_{max} defines how far away from v_i the repulsion affects the cost.[2]

Attraction forces For every agent a_i, we create an attraction function $GoalAPF_i$, based on its goal g_i:

$$GoalAPF_i(v) = \begin{cases} h(g_i, v) & \text{for agent } a_i \\ 0 & \text{for other agents} \end{cases} \tag{2}$$

where $h(g_i, v)$ is a precalculated heuristic estimation on the distance from vertex v to g_i. Given these repulsion and attraction functions, DAPF moves in every time-step each agent a_i located at v_i to the adjacent location v'_i that minimizes

$$Total\_APF(v'_i) = \sum_{j \neq i} APF_j(v'_i) + \sum_{j} GoalAPF_i(v'_i) \tag{3}$$

[1]This could be the exact minimal distance in the graph or some estimation heuristic on it. In our experiments, every agent location is associated with a cell in a grid. We used Manhattan distance.

[2]We illustrate this in the supplementary material.

Ties are broken randomly. Collisions are avoided in a prioritized planning manner. The agents plan sequentially (with some random order), where every agent cannot occupy the current locations of other agents and the locations reserved by previously planned agents.

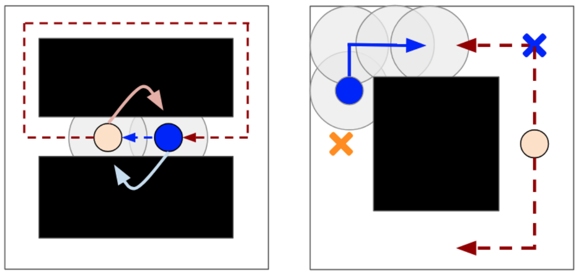

(a) (b)

Figure 1: Example illustrations. (a) An instance that cannot be solved by DAPF. Solid lines point to goals. The dotted lines depict a PrP solution. (b) Two agents solve LMAPF. The X shapes represent goal locations. A solid blue line shows the direction of a chosen path for the blue agent. Dashed red lines represent alternative k-length paths for the orange agent. The agent will prefer the bottom path because of the APFs of a blue agent.

DAPF is very efficient computationally. In every iteration, each agent incurs a runtime of $O(k \cdot d_{max}^2)$, where k is the number of agents and d_{max} is the radius of influence of APFs. So, the overall runtime complexity of DAPF is $O(k^2 \cdot d_{max}^2 \cdot N)$, where N is the number of iterations.

In spaces with relatively few obstacles and a small number of agents, using DAFP is very fast. However, it performed poorly in our experiments when the environment became more dense. This is because there are cases where long-term planning is needed and a naïve usage of APFs, as done by DAPF, is insufficient no matter which parameters we choose. For example, consider the MAPF problem in Figure 1 (a). Solid lines point to the agents' goal locations. Here, DAPF fails to find a solution. Agents are pushed in and pulled back to the middle corridor, preventing agents from swapping positions. The possible solution is for the orange agent to bypass the obstacle and reach its goal (dashed line in Figure 1 (a)). In the following sections, we propose a range of techniques for using APFs within existing MAPF algorithms.

4 PIBT and LaCAM with APFs

Both PIBT (Okumura et al. 2022) and LaCAM (Okumura 2023a) use a heuristic to prioritize the agents' actions when generating configurations. In these algorithms, each agent chooses its next action by sorting the vertices adjacent to it based on a heuristic estimate of their distance to the goal.

We propose to add APFs in these heuristic estimates. Specifically, when sorting the vertices adjacent to the k'-th agent, we consider the actions chosen by the previously planned agents $(1, \dots, k' - 1)$ with Equation 1. Then, we sum all the APFs for every neighboring vertex as follows:

$$cost_{APF}(v) = \sum_{i \in \{1,\dots,k'-1\}} APF_i(v) \tag{4}$$

where $APF_i(v)$ is defined above (Eq. 1). Finally, we sort the vertices according to $h(v) + cost_{APF}(v)$ values. This use of APFs is reminiscent of the DAPF algorithm presented above, and may suffer from similar limitations. The overhead incurred by APFs in terms of runtime complexity is $O(k \cdot d_{max}^2)$ per time-step, corresponding to computing $cost_{APF}$ for all k agents and nodes in radius d_{max}. Therefore, the total overhead is $O(k \cdot d_{max}^2 \cdot l)$, where l is the length of the longest path.

5 Temporal A* with APFs

As noted above, many MAPF algorithms internally use TA* to find paths for individual agents. We propose to use APFs in TA* such that the resulting path not only avoids collisions with the paths of other agents (which is a hard constraint) but also attempts to keep distance from them by considering the repulsion of their APFs. We refer to our TA* variant as *Temporal A* with Artificial Potential Fields* (TA*+APF).

TA*+APF is a single-agent path-finding algorithm. It accepts as input a tuple $\langle G(V, E), s, g, \{\pi_1, \dots, \pi_{k'}\}\rangle$ where G is the graph of possible locations (V) the agent can occupy and allowed transitions between them (E); s and g are the start and goal locations of the path planning agent, respectively; and π_i is a path for agent a_i for every $i = 1, \dots, k'$. The output of TA*+APF is a path from s to g that does not conflict with any of the paths $\pi_1, \dots, \pi_{k'}$. The given set of paths $\{\pi_1, \dots, \pi_{k'}\}$ depend on the particular MAPF algorithm in use. For example, in PrP TA*+APF will be given the paths planned for the higher-priority agents. In LNS2, the given set of paths includes the paths already planned for the other agents within the neighborhood of the planning agent.

To bias the resulting path to keep distance from these paths, TA*+APF creates for every path $\pi_i \in \{\pi_1, \dots, \pi_{k'}\}$ a repulsion APF function APF_i that maps every location-time pair (v, t) to a real number representing the added penalty incurred by planning to occupy v at time t. TA*+APF considers these penalties when computing the cost of every move. There are multiple ways to define these repulsion APF functions and to aggregate them into a single penalty cost. We experimented with several options, and have found the following implementation to work best in our experimental setup. Similar to DAPF, the APF induced by agent a_i on location v and time t is computed as follows:

$$APF_i(v, t) = \begin{cases} 0 & \text{if } d(v, \pi_i[t]) \geq d_{max} \\ w \cdot \gamma^{-d(v, \pi_i[t])} & \text{otherwise} \end{cases} \tag{5}$$

where d_{max}, γ, and w are predefined parameters and $d(v, t, \pi_i[t])$ is the minimal distance between v and $\pi_i[t]$.[3] The purpose of each parameter is as described in Section 3. The only difference between this computation method compared to the one used by DAPF (Eq. 4) is the introduction

[3]Again, we used Manhattan distance.

of the time dimension. Hence, in each time-step t' an agent considers only APFs that are calculated at t'. In the special case in which $d_{max} = 0$, $APF_i(v,t)$ is always 0. This corresponds to plain TA* that only considers conflicts where agents are exactly at the same location (distance 0 for each other). To aggregate all the APFs we used a simple sum. That is, the APF cost of moving into location v at time is

$$cost_{APF}(v,t) = \sum_{i \in \{1,\dots,k'\}} APF_i(v,t) \tag{6}$$

We pondered incorporating our APF-inspired cost function within TA* in two places: either as part of the heuristic evaluation function or as part of the edge cost function. For a node n generated by TA*, the former adds a penalty to $h(n)$ and the latter adds a penalty to $g(n)$. We initially incorporated our APF-inspired cost function in $h(n)$ and observed poor results. To understand these results, consider what each value (h and g) represents and its objective. $h(n)$ represents an estimate of the cost from n to the goal. It is designed to guide the search towards finding the goal faster. In contrast, $g(n)$ represents the cost of the best path found so far from the start to n. A*, and subsequently TA*, are designed to minimize the edge cost function, and correspondingly find the path to the goal with the lowest g-value. Thus, incorporating APFs in the heuristic functions does not, directly, affect what path TA* returns, but rather, how quickly the search finds it. In contrast, incorporating APFs in the edge cost function (and thus in the g value) directly results in TA* returning paths that optimize for avoiding other agents' paths.

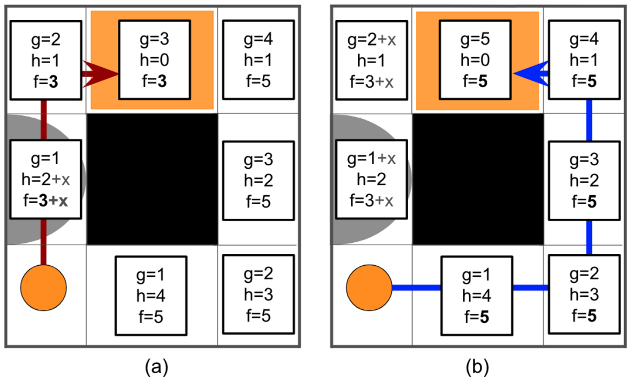


Figure 2: The orange circle and the orange square are the agent's start and goal locations, respectively. x represents the cost of APFs. A* is executed. g, h and f are the components of A* nodes. (a) With $x \leq 2$ an agent always picks a red path, wherever x is added. (b) With $x > 2$ the agent picks a blue path if x is added to g. Otherwise, if x is added to h, it continues to choose a red path, nonetheless.

Examine Figure 2, where an agent needs to go from the orange circle to the orange square. The g, h, and f components of an A* algorithm are presented inside the locations. The gray zone shows APFs of another agent that adds to the cost x of moving via the location in the middle left of the figure. Consider the case where $x > 2$. A red line is an optimal path that A* would choose if the costs were added to an h component (Fig. 2 (a)). A blue line is the optimal path if the cost is a part of a g component, and this is the variant that we want to implement (Fig. 2 (b)). In other words, h is not being aggregated along paths. It is only defined for individual locations and only attracts the agent towards the goal. By contrast, g is being aggregated and an edge with a large weight will be carried over to all its descendants, and this is what we want.

Therefore, in TA*+APF we chose to use our APF-inspired cost function when computing the g value of a search node, as follows. Let $cost(parent, n)$ be the cost of moving the agent from $parent$ to n, and let (v,t) be the vertex and time-step that node n represents. In TA*+APF we compute the $g(n)$ as follows:

$$g(n) = g(parent) + cost(parent,n) + cost_{APF}(v,t) \tag{7}$$

TA*+APF runs TA* according to $f(n) = g(n)+h(n)$, using this modified $g(n)$ function.

Figure 1(b) illustrates an example case, where APFs help to pick a better path out of two options. In this example, two agents work in a RHCR framework. The blue agent plans first and goes directly to its goal. An orange agent has two alternative paths around the obstacle. However, the APFs of the blue agent (gray circles) will cause the orange agent to prefer the bottom path and therefore avoid future conflicts.

Using APFs incurs some computational overhead, compared to vanilla TA*. This overhead is due to the need to compute APFs for every newly generated path. The computational complexity of generating an APF for the path of a single agent is $O(d_{max}^2 \cdot l)$, where d_{max}^2 is the maximal number of nodes affected by the APF induced by an agent occupying a single vertex at a specific time step; and l is the length of the longest path. This computation is done for each of the k agents, adding a total overhead of $O(k \cdot d_{max}^2 \cdot l)$.

6 SIPPS with APFs

A more recent state-of-the-art low-level solver in multiple MAPF algorithms is SIPPS (Li et al. 2022). As was described in the background section, SIPPS is an enhanced version of TA*, that uses time intervals instead of time-steps and accepts a set of *soft constraints* defined by paths of previously planned agents. Moreover, SIPPS orders its nodes in the open list differently from TA*. First, it sorts the open list according to $c(n)$, which tracks the amount of soft collisions (violated soft constraints, caused by a collision with another path). Then, the secondary sort (between the nodes with the same $c(n)$ value) is executed according to $f(n)$ as in TA*. The $g(n)$ component equals the lowest value in the node's time interval. This way, SIPPS ensures that the found path minimizes the number of soft collisions. SIPPS does not guarantee optimally of the length of the path, and thus is often used within a suboptimal MAPF algorithm such as EECBS (Li, Ruml, and Koenig 2021).

Similar to TA*+APF we propose to use APFs so that the agents will try to keep a distance from each other and, hopefully, arrive faster to their goals. We refer to our SIPPS variant as *SIPPS with Artificial Potential Fields* (SIPPS+APF).

Because the sorting of the open list in SIPPS+APF is executed according to two components, first $c(n)$ and then $f(n)$, we explored incorporating APFs separately in each of these components. We first tried to append APFs to $c(n)$ only and the effect was modest. Then, we tried to append APFs only to the $g(n)$ component as in TA*+APF and the improvement was also small. Nevertheless, the combination of both resulted in a significant boost. Hence, the final formal definition is presented as follows.

$$cost_{APF}(n) = \max_{t \in [t^n_{start}, t^n_{end})} \sum_{i \in \{1,\ldots,k'\}} APF_i(v,t) \quad (8)$$

Where $APF_i(v,t)$ is defined in TA*+APF, and $[t^n_{start}, t^n_{end})$ is the current safe interval of the n node. t^n_{start} and t^n_{end} are the beginning and the end time-steps of the interval. In other words, $cost_{APF}(n)$ represents the highest APFs an agent can encounter during its time interval. In SIPPS+APF we compute the $g(n)$ as follows:

$$g(n) = t^n_{start} + cost_{APF}(n) \quad (9)$$

And $c(n)$ is computed as follows:

$$c(n) = count\_soft\_collisions(n) + cost_{APF}(n) \quad (10)$$

Here, t^n_{start} is the beginning of the n's safe interval, and *count_soft_collisions(n)* is an internal SIPPS function that counts soft collisions. SIPPS+APF runs SIPPS and first prioritizes nodes according to small $c(n)$ values. In case of a tie, it moves to the secondary priority and prefers nodes with smaller $f(n) = g(n) + h(n)$.

The analysis of the runtime complexity overhead incurred by APFs in SIPPS+APF is similar to the described above analysis for TA*+APF. An additional computational cost is incurred due to maxing over the time steps in the relevant safe interval (Eq. 8). So, the overhead equals $O(k \cdot d^2_{max} \cdot l^2)$, where k, d_{max}, and l are defined as earlier.

7 Experimental Study

We conducted an experimental evaluation comparing the use of APFs within PrP (Silver 2005), LNS2 (Li et al. 2022), PIBT (Okumura et al. 2022), LaCAM (Okumura 2023a), and LaCAM* (Okumura 2023b), where PrP and LNS2 are implemented twice: once with TA* and once with SIPPS. With APF-enhanced versions, this adds up to a total of 14 algorithms. The versions that use APFs are denoted in our plots by dashed lines. All experiments were performed on four different maps from the MAPF benchmark (Stern et al. 2019): *empty-32-32*, *random-32-32-10*, *random-32-32-20*, and *room-32-32-4*, as they present different levels of difficulty. The number of agents used in our experiments varied from 50 to 450. We executed 15 random instances per every number of agents, map, and algorithm. The APF parameters used were $w = 1$, $d_{max} = 4$, and $\gamma = 2$ for TA*+APF, $w = 0.1$, $d_{max} = 3$, and $\gamma = 3$ for SIPPS+APF, and $w = 0.1$, $d_{max} = 2$, and $\gamma = 1.1$ for APFs in PIBT, LaCAM, and LaCAM*, which were observed to work best in general.[4] All algorithms were implemented in Python and ran on a MacBook Air with an Apple M1 chip and 8GB of RAM.

[4]A sensitivity analysis of the impact of these parameters is discussed later and in the supplementary material.

Negative Results in Standard MAPF In all our standard MAPF experiments, using APFs in PrP, LNS2, PIBT, and LaCAM, yielded either identical or inferior results.[5] For PIBT, LaCAM, and LaCAM* algorithms, we conjecture that this is due to the myopic nature of these algorithms, choosing a single step ahead in every iteration. For PrP and LNS2, we explain these poor results by the fact that the APFs encourage the single-agent path planning algorithm (TA* or SIPPS) to avoid the plans of agents that have already chosen a plan. Thus, the APFs only make the planning harder while avoiding plans that are already chosen to be part of the solution. Thus, the expected benefit of APFs — avoiding congested areas — did not manifest in performance gains. However, finding plans that avoid congested areas can bring significant gains in the lifelong setting, where agents continuously receive new tasks over time. We demonstrate this in the next set of experiments, which evaluate our APF-augmented algorithms within the RCHR framework, in a lifelong MAPF setting.

Lifelong MAPF In this set of experiments, we solved LMAPF problems using the RHCR framework with the *window* and planning *horizon* parameters set to 5. Every algorithm was limited by 10 seconds for the planning phase. In planning-failure events, i.e., when an agent could not find a path within the time limit, we followed Morag et al.'s (2023) robust MAPF framework using the *AllAgents* + *iStay* + *Persist* configuration. This corresponds to having all agents plan in every planning period (*AllAgents*); failing agents stay in their place (*iStay*); and all agents that do have a path and can follow it without conflicts, do so (*Persist*). In the planning phase, we executed all algorithms and our experiment halted after each agent performed 100 steps. The main metric in the LMAPF experiments is the average *throughput* of each algorithm, i.e., the number of times when an agent reaches its goals before the aforementioned 100 time-steps limit is reached. As noted above, throughput is the common metric for measuring the quality of algorithms in the LMAPF literature (Li et al. 2021b; Song, Na, and Yu 2023; Morag, Stern, and Felner 2023).

— Figure 3 plots the average throughput of the different algorithms as a function of the number of agents. The results for PIBT, LaCAM, and LACAM* with and without APFs were virtually the same, due to the myopic way in which these algorithms work. Therefore, we only plot in Figure 3 the results of these algorithms without APFs.

As can be seen in Figure 3, using APFs significantly increases the throughput of all other algorithms in all maps. For example, in the *empty-32-32* LNS2 with TA*+APF reaches a throughput of around 1400 with 450 agents, which is approximately 7 times more than the throughput of vanilla LNS2 for the same number of agents. The observed significant advantage of using TA*+APF within the RHCR framework may suggest that it indeed achieves its intended purpose: biasing agents towards paths that avoid other agents' paths, reducing future collisions and congestion. As an interesting side phenomenon, we observed the superior per-

[5]Detailed description of results can be found in the supplementary materials.

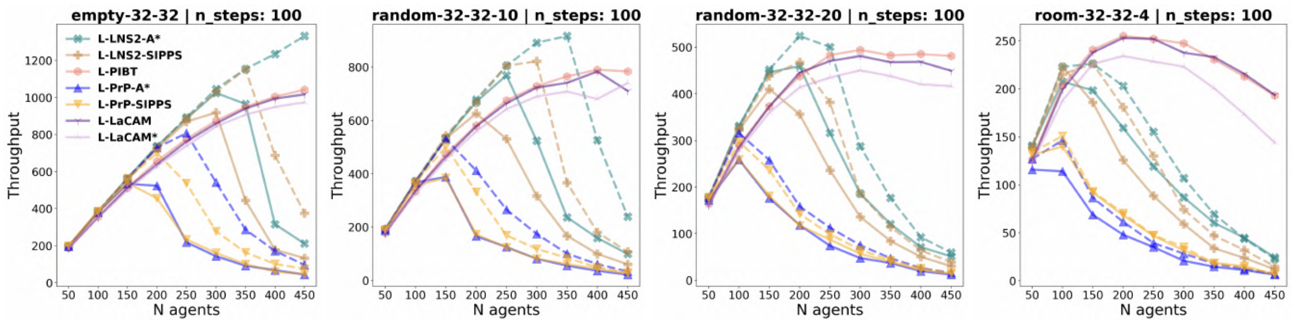


Figure 3: LMAPF: Average Throughput. Dashed lines - APF-enhanced; Solid lines - no APFs

formance of LNS2 algorithms using A*, compared to LNS2 with SIPPS. It might be due to different search techniques of A*, where it prioritizes short paths, and SIPPS, where it prioritizes paths with minimum soft constraints.

Parameter Sensitivity Analysis Our APF algorithms require setting several parameters, d_{max}, γ, and w. Next, we present a sensitivity analysis for these parameters. As a representative example, we focus on the algorithm with the best performance in our experiments — LNS2 with TA*+APF. We show the impact of the TA*+APF parameters — d_{max}, γ, and w — on the overall performance in our LMAPF experiments. We report here only results on the *random-32-32-10* map but similar trends were observed when using other algorithms and other maps.

Figure 4 plots an average throughput as a function of the number of agents, for different values of d_{max}, γ, and w, respectively. Consider first the impact of varying the d_{max} parameter (Fig. 4(a)). Recall that this parameter defines how far away from an agent's planned location a potential field reaches. The results show that setting d_{max} to either extreme value — too small (d_{max}=1) or too high (d_{max}=10) — yields significantly worse results than setting d_{max} to 4. Yet, even in these cases, APF-enhanced LNS2 significantly outperformed plain LNS2.

Next, consider the impact of the γ parameter (Fig. 4(b)). Recall that γ defines how fast the impact of the APF decreases with the distance from its origin. Setting $\gamma = 1$ corresponds to an APF that has a uniform effect regardless of distance, as long as the distance is smaller than d_{max}. This assignment for γ =1 yielded even worse results than vanilla LNS2. All other values of γ yielded much better results, where γ =2 worked best in this setting.

Finally, consider the impact of the w parameter (Fig. 4(c)), which determines how much weight should be given to the cost stemming from APFs as opposed to the regular cost of moving. The impact of the value of this parameter is interesting because its preferred value depends on the number of agents in the problem. For example, $w = 2$ yielded the best results for problems with 400 agents, while setting $w = 1$ was best for the other. We also explored setting different w values for different agents according to their traveled path length or future path lengths. These methods did not find any significant improvements, and thus their results are not reported.

8 Conclusion and Future Work

We investigated whether MAPF can be solved efficiently using artificial potential fields (APFs). First, we showed that a direct implementation of APFs in a myopic manner is fast but may yield poor results. Then, we proposed a way to incorporate APFs into the PIBT and LaCAM algorithms. The resulting algorithms add a bias to the search to avoid passing near paths of other agents. Next, we proposed to use APFs in TA* and in SIPPS, key components of many MAPF algorithms. Specifically, we incorporated APFs in the calculation of the g component of TA*'s node evaluation function and c and g components of SIPPS's node evaluation functions.

Experimentally, we showed that APFs do not provide any advantage for PIBT and LaCAM. Also, APFs in other algorithms did not yield any advantage when solving a single offline MAPF problem. However, in the context of lifelong MAPF, we showed that using APFs in TA* and SIPPS is highly beneficial, resulting in significantly higher overall system throughput.

There are several interesting directions for future work. One such direction is to explore a more efficient way to incorporate APFs into PIBT and LaCAM algorithms. Another direction is to study how to set the d_{max}, γ, and w parameters effectively for APFs in TA* and SIPPS.

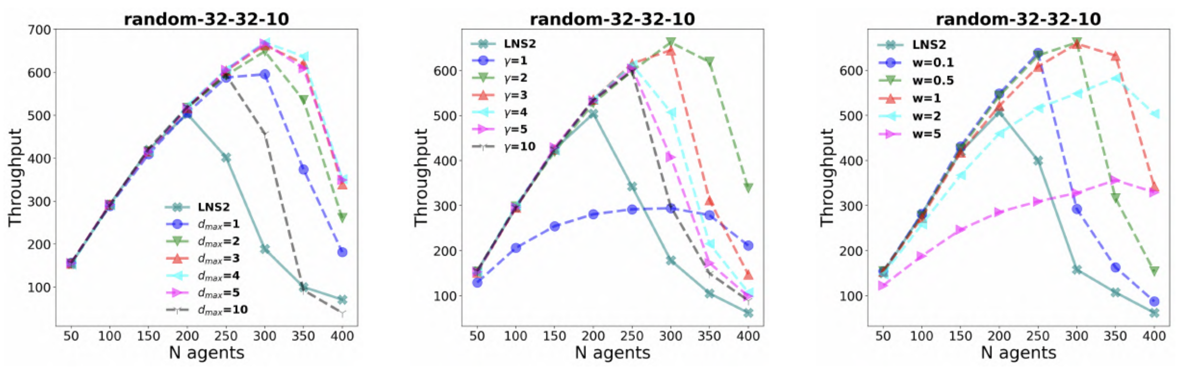
References

Agrawal, A.; Hariharan, S.; Bedi, A. S.; and Manocha, D. 2022. DC-MRTA: Decentralized Multi-Robot Task Allocation and Navigation in Complex Environments. In *IROS*, 11711–11718. IEEE.

Barraquand, J.; Langlois, B.; and Latombe, J.-C. 1992. Numerical potential field techniques for robot path planning. *IEEE Transactions on Systems, Man, and Cybernetics*, 22(2): 224–241.

Barták, R.; Švancara, J.; Škopková, V.; Nohejl, D.; and Krasičenko, I. 2019. Multi-agent path finding on real robots. *AI Communications*.

Bennewitz, M.; Burgard, W.; and Thrun, S. 2001. Optimizing schedules for prioritized path planning of multi-robot systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 1, 271–276.



(a) d_{max} values (b) γ values (c) w values

Figure 4: Throughput for different parameter values of APFs for LNS2 with TA*+APF .

Bettini, M.; Kortvelesy, R.; Blumenkamp, J.; and Prorok, A. 2022. VMAS: a vectorized multi-agent simulator for collective robot learning. *arXiv preprint arXiv:2207.03530*.

Chan, S.-H.; Stern, R.; Felner, A.; and Koenig, S. 2023. Greedy Priority-Based Search for Suboptimal Multi-Agent Path Finding. In *SoCS*, 11–19.

Chen, Z.; Harabor, D.; Li, J.; and Stuckey, P. J. 2024. Traffic flow optimisation for lifelong multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 20674–20682.

Daily, R.; and Bevly, D. M. 2008. Harmonic potential field path planning for high speed vehicles. In *2008 American Control Conference*, 4609–4614. IEEE.

Daniel Kornhauser, P. S., Gary Miller. 1984. Coordinating pebble motion on graphs, the diameter of permutation groups, and applications. In *FOCS*.

Dergachev, S.; and Yakovlev, K. 2021. Distributed multi-agent navigation based on reciprocal collision avoidance and locally confined multi-agent path finding. In *CASE*.

Dinh, H. T.; van Lon, R.; and Holvoet, T. 2016. Multi-agent route planning using delegate MAS. In *Workshop on Distributed and Multi-Agent Planning*, 24–32. London, UK.

Fan, T.; Long, P.; Liu, W.; and Pan, J. 2020. Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios. *The International Journal of Robotics Research*, 39(7): 856–892.

Fox, D.; Burgard, W.; and Thrun, S. 1997. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1): 23–33.

Hagelback, J.; and Johansson, S. 2009. A multi-agent potential field-based bot for a full RTS game scenario. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 5, 28–33.

Hagelbäck, J.; and Johansson, S. J. 2008. Using multi-agent potential fields in real-time strategy games. In *AAMAS*, 631–638.

Han, S. D.; and Yu, J. 2020. Ddm: Fast near-optimal multi-robot path planning using diversified-path and optimal sub-problem solution database heuristics. *IEEE Robotics and Automation Letters*, 5(2): 1350–1357.

Han, S. D.; and Yu, J. 2022. Optimizing space utilization for more effective multi-robot path planning. In *2022 International Conference on Robotics and Automation (ICRA)*, 10709–10715. IEEE.

Khatib, O. 1986. The potential field approach and operational space formulation in robot control. In *Adaptive and Learning Systems: Theory and Applications*, 367–377. Springer.

Koren, Y.; and Borenstein, J. 1991. Potential field methods and their inherent limitations for mobile robot navigation. In *ICRA*, 1398–1404.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144: 105809.

Laurent, F.; Schneider, M.; Scheller, C.; Watson, J.; Li, J.; Chen, Z.; Zheng, Y.; Chan, S.-H.; Makhnev, K.; Svidchenko, O.; Egorov, V.; Ivanov, D.; Shpilman, A.; Spirovska, E.; Tanevski, O.; Nikov, A.; Grunder, R.; Galevski, D.; Mitrovski, J.; Sartoretti, G.; Luo, Z.; Damani, M.; Bhattacharya, N.; Agarwal, S.; Egli, A.; Nygren, E.; and Mohanty, S. 2021. Flatland Competition 2020: MAPF and MARL for Efficient Train Coordination on a Grid World. In *NeurIPS*.

Leet, C.; Li, J.; and Koenig, S. 2022a. Shard systems: Scalable, robust and persistent multi-agent path finding with performance guarantees. In *AAAI*, 9386–9395.

Leet, C.; Li, J.; and Koenig, S. 2022b. Shard Systems: Scalable, Robust and Persistent Multi-Agent Path Finding with Performance Guarantees. *AAAI*, 36(9): 9386–9395.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P.; and Koenig, S. 2021a. Anytime multi-agent path finding via large neighborhood search. In *IJCAI*.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *AAAI*.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: Bounded-Suboptimal Search for Multi-Agent Path Finding. In *AAAI*.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. K. S.; and Koenig, S. 2021b. Lifelong Multi-Agent Path Finding in Large-Scale Warehouses. *AAAI*.

Liu, X.; Ge, S. S.; and Goh, C.-H. 2017. Formation potential field for trajectory tracking control of multi-agents in constrained space. *International Journal of Control*, 90(10): 2137–2151.

Luna, R. J.; and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *AAAI*, 7643–7650.

Ma, H.; Yang, J.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2017. Feasibility Study: Moving Non-Homogeneous Teams in Congested Video Game Environments. In *AIIDE*.

Mac, T. T.; Copot, C.; Tran, D. T.; and De Keyser, R. 2016. Heuristic approaches in robot path planning: A survey. *Robotics and Autonomous Systems*, 86: 13–28.

Morag, J.; Stern, R.; and Felner, A. 2023. Adapting to Planning Failures in Lifelong Multi-Agent Path Finding. In *SoCS*.

Morris, R.; Pasareanu, C. S.; Luckow, K. S.; Malik, W.; Ma, H.; Kumar, T. S.; and Koenig, S. 2016. Planning, Scheduling and Monitoring for Airport Surface Operations. In *AAAI Workshop: Planning for Hybrid Systems*.

Okumura, K. 2023a. Lacam: Search-based algorithm for quick multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 11655–11662.

Okumura, K. 2023b. LaCAM: Search-Based Algorithm for Quick Multi-Agent Pathfinding. *AAAI*.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310: 103752.

Phillips, M.; and Likhachev, M. 2011. Sipp: Safe interval path planning for dynamic environments. In *2011 IEEE international conference on robotics and automation*, 5628–5635. IEEE.

Rezaee, H.; and Abdollahi, F. 2012. Adaptive artificial potential field approach for obstacle avoidance of unmanned aircrafts. In *AIM*, 1–6. IEEE.

Salzman, O.; and Stern, R. Z. 2020. Research challenges and opportunities in multi-agent path finding and multi-agent pickup and delivery problems blue sky ideas track. In *AAMAS*.

Semnani, S. H.; Liu, H.; Everett, M.; De Ruiter, A.; and How, J. P. 2020. Multi-agent motion planning for dense and dynamic environments via deep reinforcement learning. *IEEE Robotics and Automation Letters*, 5(2): 3221–3226.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195: 470–495.

Shin, Y.; and Kim, E. 2021. Hybrid path planning using positioning risk and artificial potential fields. *Aerospace Science and Technology*, 112.

Silver, D. 2005. Cooperative Pathfinding. In *AIIDE*.

Skrynnik, A.; Andreychuk, A.; Nesterova, M.; Yakovlev, K.; and Panov, A. 2024. Learn to Follow: Decentralized Lifelong Multi-Agent Pathfinding via Planning and Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 17541–17549.

Song, S.; Na, K.-I.; and Yu, W. 2023. Anytime Lifelong Multi-Agent Pathfinding in Topological Maps. *IEEE Access*, 11: 20365–20380.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SoCS*, 151–158.

Surynek, P. 2010. An Optimization Variant of Multi-Robot Path Planning Is Intractable. In *AAAI*.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*, 810–818.

Švancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Barták, R. 2019. Online multi-agent pathfinding. In *AAAI*.

Vadakkepat, P.; Tan, K. C.; and Ming-Liang, W. 2000. Evolutionary artificial potential fields and their application in real time robot path planning. In *Congress on Evolutionary Computation*.

Van den Berg, J.; Guy, S. J.; Lin, M.; and Manocha, D. 2011. Reciprocal n-Body Collision Avoidance. In *Robotics Research*, 3–19.

Varambally, S.; Li, J.; and Koenig, S. 2022. Which MAPF Model Works Best for Automated Warehousing? In *SoCS*.

Wagner, G.; and Choset, H. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *2011 IEEE/RSJ international conference on intelligent robots and systems*, 3260–3267. IEEE.

Wahid, N.; Zamzuri, H.; Rahman, M. A. A.; Kuroda, S.; and Raksincharoensak, P. 2017. Study on potential field based motion planning and control for automated vehicle collision avoidance systems. In *ICM*, 208–213.

Wang, Y.; and Chirikjian, G. 2000. A new potential field method for robot path planning. In *ICRA*, volume 2, 977–982 vol.2.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1): 9–9.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *AAAI*.

Zhang, H.-y.; Lin, W.-m.; and Chen, A.-x. 2018. Path planning for the mobile robot: A review. *Symmetry*, 10(10): 450.

Zhang, S.; Li, J.; Huang, T.; Koenig, S.; and Dilkina, B. 2022. Learning a Priority Ordering for Prioritized Planning in Multi-Agent Path Finding. In *SoCS*.

Supplementary Material: Enhancing Multi-Agent Path-finding by Using Artificial Potential Fields

Anonymous submission

A Justification for Parameters

To demonstrate the role of each parameter, consider Figure 1. The x-axis is the distance from the agent (in the middle), the y-axis is the value of APFs, and the height of the red bars represents the specific APFs cost for the locations of the specific distance from the agent. All parameters of APFs are constant, except those we want to stir a bit to see the impact. We stir each parameter separately. We can clearly see that the w parameter controls the strength of the repulsion APF (Fig. 1(a)), γ controls the rate of decay, i.e. how fast its intensity declines while moving away from its source (Fig. 1(b)), and d_{max} defines how far away from v_i the repulsion APF will influence the cost (Fig. 1(c)).

B Supplementary Sensitivity Analysis

In addition to the examination of TA*+APF parameters in the main paper, we present a similar analysis of SIPPS+APF and PIBT+APF parameters. Recall, that as a representative example, we focus on the algorithm with the best performance in our experiments — LNS2 with TA*+APF. Here, we present the results only with *random-32-32-10* grid, however, analogous trends were observed in other algorithms and other grids as well.

B.1 SIPPS+APF

Figure 2 plots an average throughput as a function of the number of agents, for different values of d_{max} (Fig. 2(a)), γ (Fig. 2(b)), and w (Fig. 2(c)). Regarding d_{max} and γ, almost all the values resulted in better throughput, when values $d_{max} = 3$ and $\gamma = 3$ were of the best performance. The trend in w was different. For low values, the results were almost always better than a vanilla version. For high values of w the results were inferior to almost every number of agents, except the dense scenarios, where, for example, $w = 3$ succeeded in reaching the highest throughput. In our experiments, we chose $w = 0.1$.

B.2 PIBT+APF

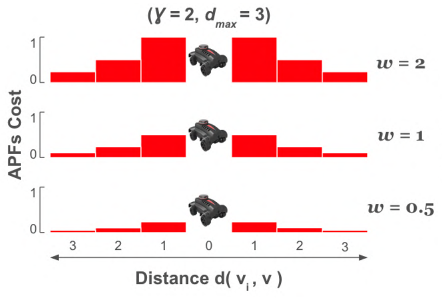
Figure 3 also plots an average throughput as a function of the number of agents, for different values of d_{max} (Fig. 3(a)), γ (Fig. 3(b)), and w (Fig. 3(c)). Regarding d_{max} and γ, there was no difference in influence between different values of those parameters. We used $d_{max} = 3$ and $\gamma = 1.1$. Regarding w, the influence was worse with higher values and remained the same with very low values. We chose $w = 0.1$ in our experiments.

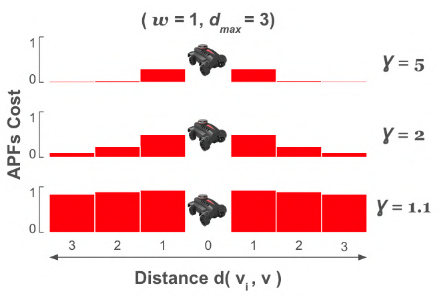
C MAPF Experiments

In this section, we present results in a standard MAPF setting. The implementation of APFs without the RHCR (Li et al. 2021) framework (i.e. original algorithms) yields inferior results. This can be described by the fact that the first agents do not consider others that come afterward and, therefore cannot take advantage of their APFs to, potentially, escape congestion. Whereas in RHCR, the agents may reconsider several times their paths before arriving at a goal. Hence, we omit the results without RHCR. For the rest of the section, all PrP and LNS2 algorithms are implemented within RHCR, where the window size and horizon depth were set to 5, which we found to be effective in our experimental setup. In this set of experiments, a time limit of one minute was imposed. For the sake of clarity, we do not report on APF-enhanced versions of PIBT, LaCAM, and LaCAM*, as they showed no substantial difference from their baseline versions. As we mentioned in the main paper, we conjecture that this is due to the myopic structure of these algorithms, choosing a single step ahead in every iteration. All experiments were performed on four different maps from the MAPF benchmark (Stern et al. 2019): *empty-32-32*, *random-32-32-10*, *random-32-32-20*, and *room-32-32-4*, as they present different levels of difficulty. The maps are visualized in the plots in Figure 4.

Success Rate Figure 5 presents the *success rate* (SR) of algorithms in different grids, where the SR is the ratio of problems that could be solved within the allocated time limit. In many cases, APFs helped to boost the performance, such as for PrP versions. But, in some cases, the results were worse with APFs, than without it, such as in LNS2 versions. Regarding the overall view, LaCAM versions succeded in solving the majority of the problems outperforming others.

Runtime Figure 6 plots the runtime (y-axis) required to solve a given number of instances (x-axis). This is also known as a “cactus” chart. The trend is similar to SR metric. In most cases, the versions of algorithms that used APFs were able to solve more instances in less time than their


(a) Different w values


(b) Different γ values

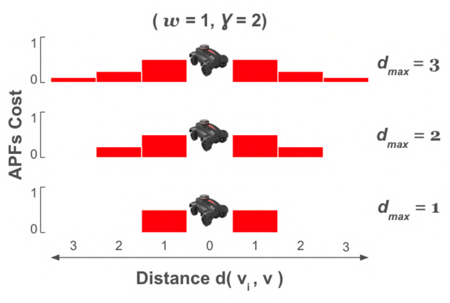
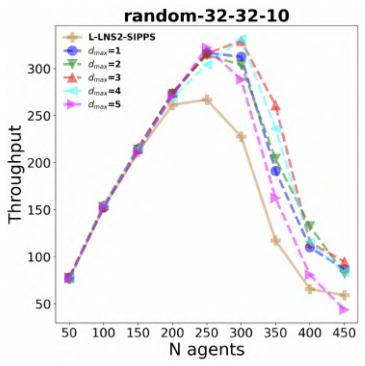
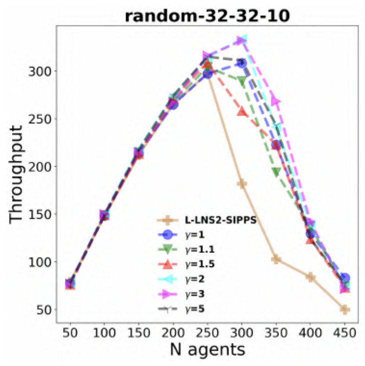

(c) Different d_{max} values

Figure 1: The influence of parameters' values on APFs. (a) w controls the strength; (b) γ controls the rate of decay; (c) d_{max} defines the radius of influence.


(a) d_{max} values


(b) γ values

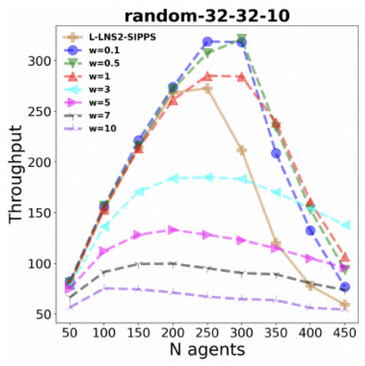
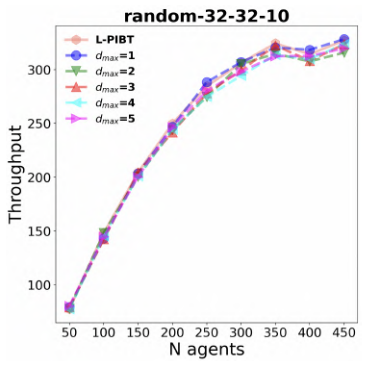
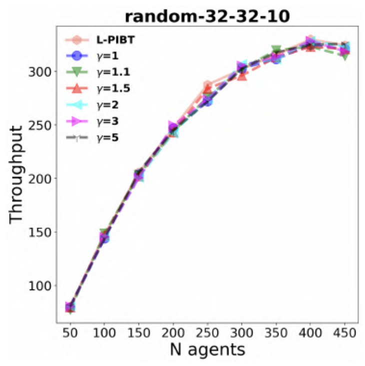

(c) w values

Figure 2: Throughput for different parameter values of APFs for LNS2 with TA*+APF .


(a) d_{max} values


(b) γ values

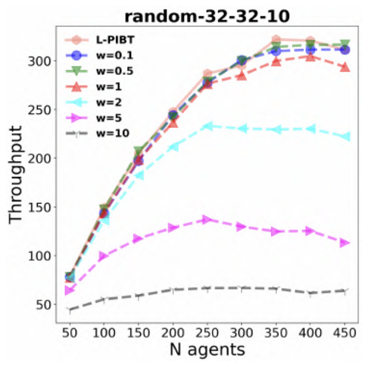

(c) w values

Figure 3: Throughput for different parameter values of APFs for LNS2 with TA*+APF .

counterparts. For example, in *empty-32-32* the APF-version of PrP with A*succeeded in solving almost twice as many instances compared to its vanilla variant. On the other hand, in case of LNS2 in *random-32-32-20*, the results of APF-enhanced versions are the same or even poorer.

RSOC Let RSOC denote the ratio between the sum of costs obtained by an APF-enhanced algorithm and the sum of costs obtained by the version of the same algorithm that does not use APFs. That is, RSOC for PrP is the sum of the costs of PrP with APFs divided by the sum of the costs of vanilla PrP. This is intended to evaluate the potential impact of using APFs on solution cost. RSOC smaller than one means APFs reduced the solution cost. Note that we only compute this ratio for problem instances that were solved

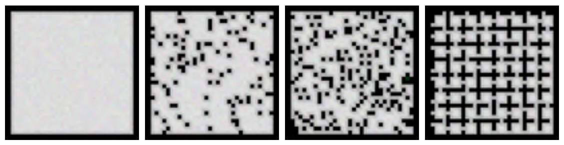


Figure 4: MAPF Grids

by both algorithms. Figure 7 presents the RSOC (y-axis) of all the benchmark algorithms for each number of agents (x-axis) for each of our maps. We can see that using APFs does not reduce the solution cost compared to vanilla versions. On the contrary, in many cases, it improves the cost. Sometimes, the improvement is two-fold, e.g., PrP versions in *random-32-32-10* grids with 200 agents.

References

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P.; and Koenig, S. 2021. Anytime multi-agent path finding via large neighborhood search. In *IJCAI*.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Bartak, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *SoCS*, 151–158.

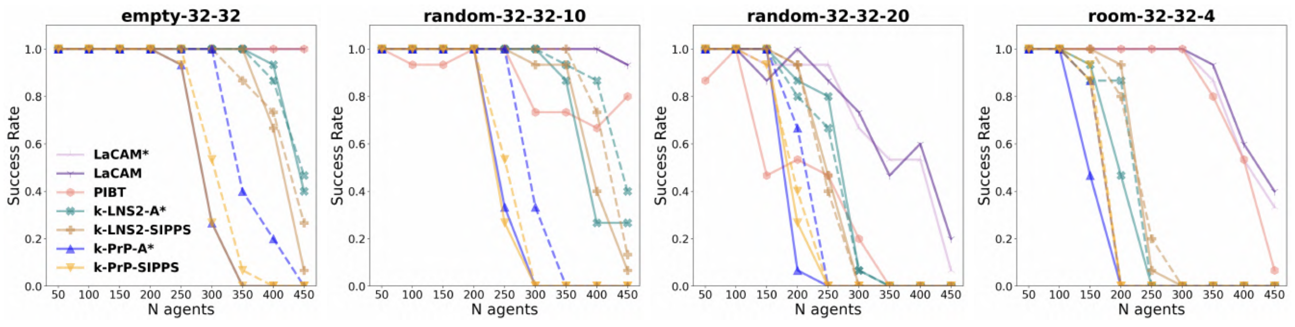


Figure 5: MAPF: Success Rate. Dashed lines - APF-enhanced; Solid lines - no APFs

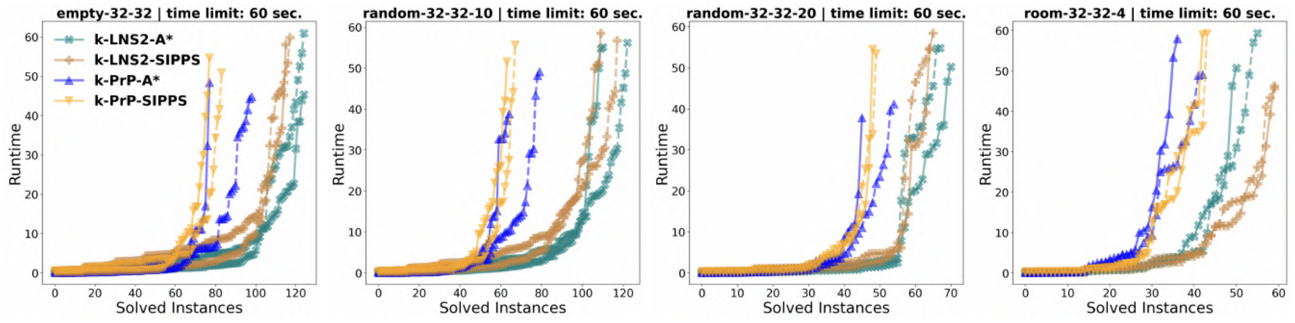


Figure 6: MAPF: Runtime. Plotting the runtime required to solve a given number of instances.

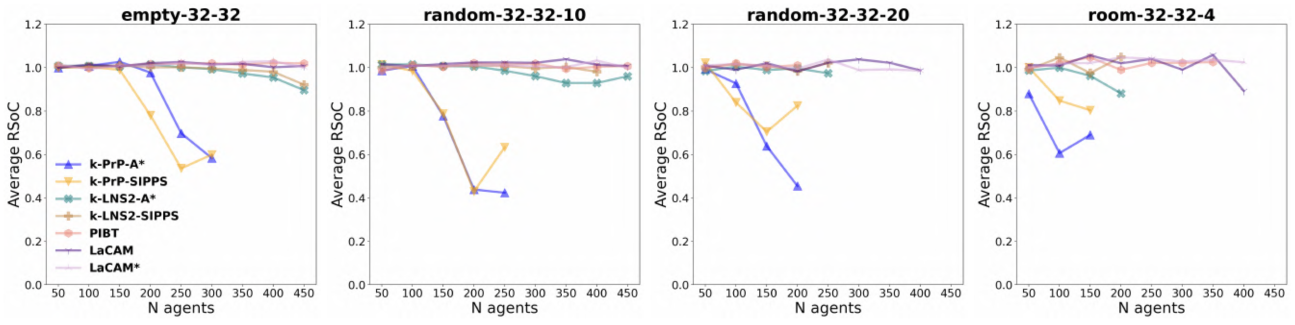


Figure 7: MAPF: RSOC