Maximize Throughput in Lifelong Multi-Agent Path Finding with Unassigned Agents

Omer Onn, Ariel Felner and Roni Stern
Faculty of Computer and Information Science
Institute of the Foundations of Artificial Intelligence
Ben-Gurion University of the Negev, Israel
omeron@post.bgu.ac.il, felner@bgu.ac.il, roni.stern@gmail.com

Abstract

Multi-agent Path Finding with Unassigned Agents (MAPFUA) is a recently introduced practical variant of MAPF. In MAPFUA there are two types of agents: *assigned agents* that have targets to go to and *unassigned agents* that do not have targets but are allowed to move to clear the way for the assigned agents. In this paper, we deal with the lifelong variant of MAPFUA (LMAPFUA) where new tasks arrive over time and the aim is to maximize the throughput. We first provide a new variant of LaCAM for MAPFUA and a general framework for LMAPFUA. We then focus on the best balance between the total number of agents available and how many of them should be assigned for achieving high throughput. Our experiments show that usually a higher throughput is achieved if some assigned agents are transferred to be unassigned agents that may be better positioned to perform future tasks.

1 Introduction and Overview

In Multi-Agent Path Finding (MAPF) (Stern et al. 2019) the aim is to find paths for multiple agents from their start to their target locations without *conflicts*. Conflict occur when two agents occupy the same location or traverse the same edge in opposite directions at the same time. Finding optimal solutions for common cost functions is NP-hard (Surynek 2010; Yu and LaValle 2013). MAPF is highly applicable e.g., in automated warehouses, vehicle routing, and multi-robot systems. Consequently, MAPF has been extensively studied. In the *lifelong variant* of MAPF (LMAPF) (Li et al. 2021) new tasks arrive over time and the aim is to maximize throughput (number of completed tasks).

Most prior work on LMAPF assumed that all agents are assigned a target at all times. That is, when an agent reaches its target it is immediately assigned a new target. However, in some real-world applications of LMAPF, some agents may not be assigned a target. This may occur at times when the rate of incoming tasks is small and the number of agents is sufficiently large. The unassigned agents may be moved from their location if this is helpful for the assigned agents.

*Multi-Agent Path Finding with Unassigned Agents (*MAPFUA*)* (Felner and Stern 2026; Gabay et al. 2026) is a recently introduced variant of MAPF where only a subset of the agents are *assigned* targets they must reach. The other agents, referred to as *unassigned* agents, have no targets they must reach, but they may still move to clear way for the assigned agents. The challenge in MAPFUA is to coordinate the paths of all agents to allow the assigned agents to quickly reach their targets. MAPFUA has been studied recently, but not in a lifelong setting where new tasks arrive over time.

The topic of our paper is thus the *lifelong* version of MAPFUA (denoted LMAPFUA). We introduce a general framework for solving LMAPFUA where up to a fixed number of tasks can be concurrently assigned and executed by the agents while the rest of the agents remain unassigned. When an assigned agent completes its task it becomes unassigned, and when a new task arrives, it is assigned to one of the unassigned agents. A MAPFUA solver is repeatedly applied to find paths for completing the assigned tasks. Specifically, we use a new variant of LaCAM (Okumura 2023) for LMAPFUA, for guiding the unassigned agents.

Additionally, we provide a new goal condition for LMAPFUA solvers (and LMAPF in general). Here, we halt the search at the time step where the first agent reaches its goal as this will trigger new task assignments and a new up-to-date search will be conducted. We show that this decreases runtime significantly without reducing throughput.

Our final main focus is on increasing the throughput by correctly adjusting the number of assigned and unassigned agents. We show that it is sometimes beneficial to lower the rate of incoming tasks, intentionally creating unassigned agents that may be better available to perform future tasks. We raise fundamental questions for LMAPFUA applications: how many agents should be used and how many of them should be assigned targets. We provide empirical evidence that appropriately answering these questions can yield significant increases in throughput.Similar trends were also reported by (He et al. 2024) while summarizing the 2024 League of Robot Runners. Nevertheless, we perform a comprehensive and systematic research on this phenomenon.

2 Background: MAPFUA

A MAPFUA problem instance is defined by a graph $G = (V, E)$, a set of agents $A = \{a_1, \ldots, a_n\}$, a start vertex $s_i \in V$ for every agent $a_i \in A$, a subset of agents $A' \subseteq A$, and a target vertex $t_i \in V$ for every agent $a_i \in A'$. The agents in A' are referred to as the *assigned agents*. Agents not in A', referred to as the *unassigned agents*, have no targets to go to. A solution π to MAPFUA maps each agent

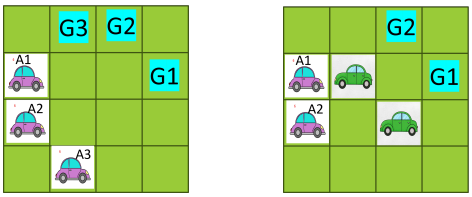

Figure 1: Left: MAPF. Right: MAPFUA.

(assigned or unassigned) to a path in G, where for each assigned agent $a_i \in A'$, the path π_i starts at s_i and contains t_i. For each unassigned agent $a_j \in A \setminus A'$, the path π_j only needs to start at s_j, but since they have no target, they can end up anywhere. The paths of all agents, assigned or unassigned, must be conflict-free. Figure 1 shows a classic problem instance (left) as well as a MAPFUA problem instance (right) where the labeled (purple) agents are assigned agents and the unlabeled (green) agents are unassigned agents.

MAPFUA is applicable for warehouses where some of the robots do not have tasks, or for modern warehouses where packages themselves have moving abilities but only some of the packages have targets assigned to them. Another example is a dense parking lot where a few cars need to exit, but other cars are blocking them. Another application is a heterogeneous team of robots, only some of which are needed at a time, depending on their functions (e.g., construction site, airport, or mine).

Several solution cost functions were suggested for MAPFUA (Felner and Stern 2026). The *Sum of Service Time* (SST) is one such cost function, which is computed as the sum over all assigned agents of the earliest time step it visited its target (ignoring subsequent steps). This quantifies how quickly the targets are visited. Since unassigned agents do not have targets, their paths do not contribute to the SST. Other solution cost functions for MAPFUA include the number of unassigned agents that are moved (e.g., unassigned cars in the parking lot), and the number of move actions (Fuel) of all the agents or only of the assigned agents (as was done by Pertzovsky, Stern, and Zivan (2024)).

Recent work proposed optimal solutions for different MAPFUA variants using heuristic search algorithms (Felner and Stern 2026) or Integer-Linear Programming (Okoso et al. 2022). Inspired by real world warehouse scenarios Fu et al. (2025) deal with the Block Rearrangement Problem (BRaP) which is very similar to MAPFUA. They experimentally compared several (suboptimal) solvers and especially recommended using a solver based on the LaCAM algorithm (Okumura 2023). Similarly, LaCAM variants for MAPFUA were studied by Gabay et al. (2026).

2.1 The LaCAM Algorithm

LaCAM (Okumura 2023) is a complete and highly scalable MAPF algorithm with two levels. The high-level searches in a depth-first manner over the *configuration space*, where a configuration is a vector containing a location for each agent. The high-level search continues until it reaches a node with the goal configuration; i.e., each agent is at its target. When expanding a high-level node, LaCAM invokes a low-level search which chooses a (conflict-free) configuration for the next successor. The low-level commonly uses PIBT (Okumura et al. 2022), a fast MAPF algorithm that efficiently moves the agents towards their targets. LaCAM may call the low-level search to generate a configuration for the same high-level node multiple times, e.g., when the high-level search backtracks (when a dead-end is reached by the search) or when the same configuration is reached from different paths. Whenever this occurs, LaCAM ensures that a different configuration is generated by maintaining relevant knowledge in every high-level node. Consequently, LaCAM is guaranteed to eventually visit all configurations in the worst case, and is therefore complete. LaCAM was reported as a fast algorithm with high quality solutions.

2.2 LaCAM for MAPFUA

The low-level of LaCAM (with PIBT) tries to move each agent towards its target. To apply LaCAM with PIBT to MAPFUA one must identify a target for each unassigned agent. Indeed, a number of target identification mechanisms were suggested for the low-level of LaCAM for MAPFUA (Gabay et al. 2026). One mechanism chose dummy targets for the unassigned agents. Another mechanism assigned the start state of an unassigned agent to be its target, i.e., it may move away but must return back to its start state. In this paper, we use a different mechanism where every vertex in G is considered a target for the unassigned agents. Thus, for each unassigned agent a_j the low-level first chooses the *wait* action. But, if moving a_j to one of the neighbors helps one or more of the assigned agents then that move is chosen. This algorithm is referred to as LaCAM$_{UA}$.

3 LMAPFUA

Lifelong MAPF (LMAPF) (Li et al. 2021) is a setting where new tasks for agents arrive over time. Usually, there is a stream of tasks (targets to go to or a stream of pickup and delivery tasks) and once an agent reaches its target it is assigned a new task from the stream. In this paper we extend LMAPF to the setting where unassigned agents exist and introduce the Lifelong Multi-agent Path finding with Unassigned Agents (LMAPFUA).

3.1 LMAPFUA Definition and Framework

The input to LMAPFUA includes: (1) a graph $G = (V, E)$, (2) a set of agents $A = \{a_1, \ldots, a_n\}$, where $|A| = N$ (3) a start vertex $s_i \in V$ for every agent $a_i \in A$, (4) T: a stream of tasks to be fulfilled by the agents, and (5) *Task capacity*: a fixed number of allowed concurrent tasks, C. That is, at all times, C tasks from T are assigned to agents to fulfill. By definition $C \leq N$. Otherwise, if $C > N$ then C is reduced to be N, as N agents cannot simultaneously process more than N tasks. So, at all times exactly C agents are *assigned* while $N - C$ agents are unassigned.[1] For now we assume

[1] N, C are listed here as inputs. But, sometimes the user has the choice to decide how many agents to have (N) and how many of them should be concurrently assigned (C). We deal with this below.

Algorithm 1: Framework for LMAPFUA

```
1  Input(A, C, T)
2  for i = 1 to C do
3      AllocateAgent(Pop(T))
4  SolveMAPFUA(G, C)
5  foreach time step do
6      Move agents according to the plan
7      Fulfilled = 0
8      foreach a ∈ AssignedAgents do
9          if a Fulfilled its task then
10             a becomes unassigned
11             Fulfilled++
12 for i = 1 to Fulfilled do
13     AllocateAgent(Pop(T))
14 if Fulfilled > 0 then
15     SolveMAPFUA(G, C)
```

that a task is to reach a target. Below we expand this to other possible tasks such as pickup and delivery.

We next introduce a general framework that solves LMAPFUA and the pseudocode is presented Alg. 1. Two calls to external functions are made and we deal with them below. Initially (Lines 2-3), the first C tasks from T are popped and are assigned to C agents from A. These C agents are currently the *assigned agents*. The rest $N - C$ agents are the *unassigned agents*. A MAPFUA solver is then executed (Line 4) to solve the current problem (with the first C tasks). The agents then start following their paths. Then, at every time step if $L \geq 1$ assigned agents fulfill their task they becomes unassigned (Line 10). Then, L new tasks are popped from T, and the algorithm allocates these tasks to L unassigned agents (Line 13).[2] Every time new assignments are made, a MAPFUA solver is executed from the current *configuration* (i.e., from the locations of all assigned and unassigned agents, Line 15) to accommodate the new tasks. This way, at all times exactly C tasks are concurrently assigned.

There are two external algorithmic choices in the LMAPFUA framework: (1) *Agent allocation:* which unassigned agent to allocate to a new task (Lines 3,13); and (2) *Path finding:* which path each agent should follow, possibly also modifying the paths of assigned agents already on their way to a previous task (Lines 4,15). There are many possible choices for these algorithms and we deal with them next.

3.2 Agent Allocation

We tried two allocation policies. The first, used as baseline, is *Random*: randomly choose an unassigned agent to fulfill the task that was popped. The Random allocation can also simulate the case where the allocation is an external process beyond our control. A more informed policy is *Closest*: choose the unassigned agent that is closest to the task location. This can be done by a heuristic, such as Manhattan/Airline distance. Alternatively, we can precalculate and use the *all-pairs shortest-path* data which stores all shortest-paths under the relaxation that only one agent exists in the graph.

3.3 Path Finding

The common approach for path planning in LMAPF is to use Rolling Horizon Collision Resolution (RHCR) (Li et al. 2021). RHCR has two parameters: *Planning horizon* H and the *Replanning period* R. In RHCR, paths are planed until agents reach their targets but these paths are constrained to be conflict free only up to the first H time steps. Beyond H time steps, shortest paths are planed but they may conflict. Then, given the replanning period $R \leq H$ agents follow the first R steps and immediately replan thereafter. The rationale for RHCR is that in LMAPF the system is dynamic since agents receive new tasks once they reach their targets. Thus, finding full paths for all agents is not advisable as suffixes of paths may not be relevant once new tasks are present.

To find non-conflicting paths up to the horizon H, RHCR calls a *windowed MAPF* algorithm. Several such algorithms have been proposed for this purpose, including algorithms based on MAPF-LNS (Li et al. 2021) and LaCAM (Veerapaneni et al. 2025). Morag et al. (2025) observed that even within the planning horizon, there is no need to plan to reach a specific goal configuration where all agents are at their targets, since when an agent reaches its target it will be assigned a new task and will likely not stay in its target. They proposed a modified objective for MAPF called the *Reachability Objective* (MAPFRO), where the agents only need to *reach* their targets but do not have to stay there afterwards. Several algorithms for solving MAPFRO were proposed, including an efficient version of LaCAM which have also been extended to solve MAPFUA (Gabay et al. 2026).

We continue the MAPFRO reasoning and observe that at a time step where one agent (or several agents) has reached its target, new tasks may be assigned to some agents and all other paths may be modified. Thus, planning after that time step may be redundant. Therefore, we propose an even more extreme halting condition for our path planner:

In our new LaCAM$_{UA}$ variant, initially, we search for paths that place all agents at their target. However, a configuration Q triggers a halt if (at least) one agent in Q is located at its target. Once such a configuration is found our LaCAM variant halts and returns the paths to configuration Q. We refer to this variant of LaCAM as LaCAM with Conflict Free Early Termination (CFET). This can be seen as a dynamic horizon that is determined on the fly based on the first target arrived by an agent. That is, H is actually the number of time steps until to the first goal configuration Q is found. Then, we replan (equivalent to setting $R = H$).

Experiments with different path-findings We experimented with a number of RHCR variants and found that our new CFET variant runs much faster than all others. Figure 2 provides representative results of the average CPU

[2] This is a common setting in LMAPF. But, it can be generalized to the option of popping more than L tasks from T and choosing which ones we want to handle. In this scenario we will have to avoid starvation of far-away tasks. In addition, in the limit, the entire stream may be popped and this will cancel the lifelong nature of the setting. We leave such settings for future work.

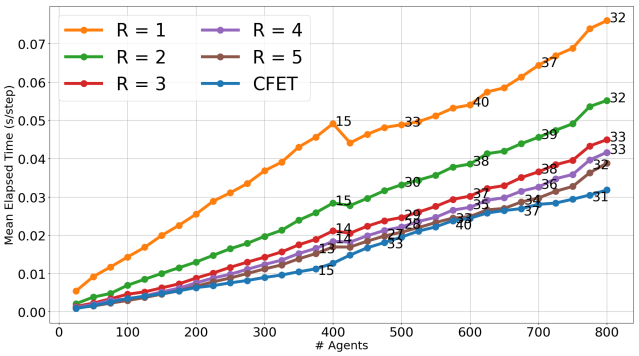

Figure 2: Time per time step for different variants.

time (over 20 instances) spent per time-step (y-axis, in seconds) as a function of the number total agents N (x-axis) in a 32×32 empty grid. Importantly, the runtime of the pathfinding algorithm is a small fraction of a second, which is negligible in real-world environments (e.g., robots in warehouses). For $N < 400$ then we set $C = N$. For $N \geq 400$, We set $C = 400$ and $N - 400$ agents were designated as unassigned. The different curves are for different RHCR instantiations of LaCAM$_{UA}$ for LMAPFUA as well as for our new CFET approach. For RHCR we varied the replaning period R from 1 to 5 and always set the planning horizon to $H = 2 \times R$. Clearly, CFET (blue curve) is up to 5 times faster per time step than all other RHCR variants. Deeper treatment on throughput will be given in the main experimental section below. Nevertheless, the throughput achieved (in thousands) after 500 time steps is written on the curves for N of hundreds. One can observe that very similar throughput is achieved for the different path finding variants with the same mixture of agents. Similar trends were observed for other RHCR settings and on other graphs.

The CFET criterion can be used in conjunction with RHCR and any combination of H and R. In fact, it can be used in other algorithms (besides LaCAM$_{UA}$) that work in the configuration space and for both LMAPF and LMAPFUA. A deeper study of all these is left for future work. Nevertheless, in the remainder of this paper we focus on the best balance between the number of assigned and unassigned agents and their influence on the throughput of the system. Thus, below we only report the throughput of the CFET variant as it performed best in our experiments in terms of both time and throughput.

3.4 Tasks in LMAPFUA and Their Algorithms

In this paper, we deal with three types of tasks that may be present in the task stream. We next describe each of them coupled with the modification needed for the LMAPFUA framework and LaCAM$_{UA}$.

Arrive at Target (AT) Here, tasks in the stream consist of a target $t \in G$. The agent that is assigned to the task needs to reach t as soon as possible. This is along the reachability objective studied by (Gabay et al. 2026). LaCAM$_{UA}$ described above is directly applicable here.

Arrive and Wait (AW) Here, the task is to arrive at t but the agent is also required to wait at its target for k consecutive time steps, where k is a parameter. This corresponds to cases where, e.g., a local job should be done at the target (repair, give aid etc.) and it takes k consecutive time steps for the agent to perform the job. Adjusting LaCAM$_{UA}$ for AW is done as follows. First, in the definition of the configuration space then for each agent a_i we also add a variable q_i. When a_i is an assigned agent, q_i counts how many consecutive time steps the agent still needs to wait at t_i. Upon assigning a task to agent a_i, q_i is initialized to k and is decremented every time a_i performs a *wait* action in t_i. if a_i leaves t_i then q_i is reset to k. [3] An AW task is completed only when a_i is located at t_i and $q_i = 0$. Another modification is to the low-level of LaCAM$_{UA}$. Let h_i denote the heuristic used by PIBT for agent a_i. With this task we set $h'_i = h_i + q_i$ and use h'_i as the heuristic for agent a_i. The rest of the algorithm remains the same. Note that when an agent leaves its target after a number of waits actions then its heuristic may be significantly increased. This will be considered by PIBT when choosing the next step of the agent.

Pickup and Delivery (PD) Here a task in the stream consists of two location $p, d \in V$. The assigned agent needs to arrive at p and then go to d. This corresponds to the *pickup and delivery* tasks that are very common in warehouses (Ma et al. 2017). We modified the LMAPFUA framework to work in two stages. When a new PD task is popped from T we assign an agent a_i to it. We first set p as the target for a_i and activate the MAPFUA solver (LaCAM$_{UA}$ in our case) accordingly. When the agent reaches p then a_i is immediately assigned a new target d (and the agent allocation procedure is not activated) and the MAPFUA solver is executed again with this new target assigned to a_i. LaCAM$_{UA}$ is directly applicable here too in the two stages.

3.5 Balance Between the Agents

An important issue to deal with is the following two possible additional choices that the user (e.g., the owner of the warehouse) needs to make to maximize the throughput. Sometimes N, the number of agents, is fixed. But, the user needs to choose the exact task capacity C. That is, the user needs to choose how many agents from A will be assigned at each time and how many will be unassigned. Alternatively, another (dual) version of this dilemma is the case where C is given as input but the user needs to to choose $N \geq C$ (and have C assigned agents and $N - C$ unassigned agents). That is, the owner of the warehouse needs to choose how many agents to purchase and place in the warehouse. A third version is that the user needs to choose both N and C.

Adding more agents have pros and cons. Intuitively, increasing C (and having more assigned agents) will likely increase the throughput because more tasks can be concurrently processed. Similarly, having more unassigned agents will improve the allocation procedure as there are higher chances that one of the unassigned agents is very close to

[3] Alternatively, if we do not reset q_i to k in such cases, then the constraint is that the agent should wait at its target k time steps but these k time steps need not be consecutive.

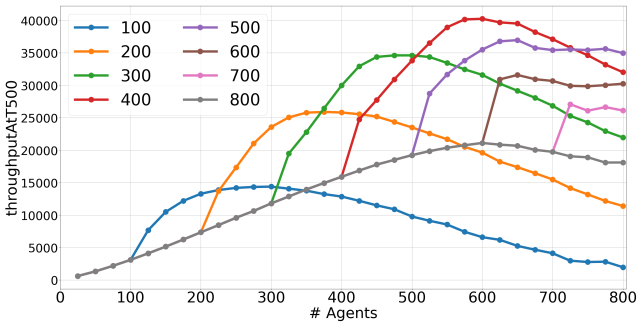

Figure 3: 32×32. Closest Allocation, AT tasks.

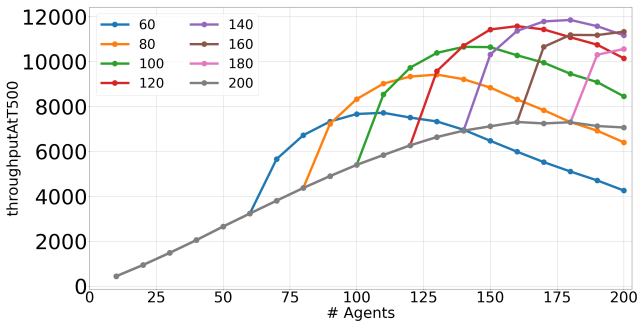

Figure 4: 16×16. Closest Allocation, AT tasks.

the new task location. By contrast, adding more agents will cause the environment to be more dense. This will increase the number conflicts, increase the lengths of the paths and will likely reduce the throughput. So, how many agents (N) do we use? and how should they be partitioned to assigned and unassigned (by choosing C)? This is dealt next in our experimental section.

4 Experimental Results

We experimented with our LMAPFUA framework on several types of domains from the Moving AI repository (Sturtevant 2012). We first concentrate on an empty 32×32 grid which serves as a good representative. We then present results on other domains too. We used LaCAM$_{UA}$ as the MAPFUA solver with the *Closest* allocation policy with the AT task. We also experimented with other sizes of grids, with the *Random* allocation, and with the AW and PD tasks. The aim is to study the tradeoff between the number of existing agents N and the number of tasks that are executed concurrently C and their influence on the total throughput. Results here are for CFET but other RHCR variants produced similar trends.

4.1 Arrive at Target Throughput

Figure 3 presents the throughput of different agent configurations on our 32×32 grid. Each point in the figures reports the throughput after 500 time steps, averaged over 20 instances. The x-axis is the total number of all agents that are present (N, both assigned and unassigned), and the y-axis shows the resulting throughput. Different curves represent different values of C which is the number of assigned agents that are allowed to concurrently execute tasks. When the total number of agents N (x-axis) is larger than the number of concurrent tasks C then $N - C$ are unassigned agents. Likewise, when $N < C$ then C is reduced to N.

The baseline curve is the gray curve of $C = 800$. For all values of N in the x-axis it holds that $N \leq 800$ and therefore this curve corresponds to the case where *all* agents are assigned and execute a task (denoted hereafter as *all agents assigned* AAA), and no unassigned agents exist. Interestingly, the best throughput for AAA is achieved at $N = 600$ with a throughput of 21,000 completed tasks. In general, increasing the number of agents allows more tasks to be executed. Thus, adding more agents up to 600 improves the throughput. But, adding more agents and more tasks beyond 600 hurts the throughput. This is because as the environment becomes denser, more conflicts arise and the consequently paths get longer. Next, consider the blue curve of $C = 100$. This curve converges with the $C = 800$ gray curve up to $N = 100$ because for $N \leq 100$ all N agents are assigned. Consider the blue curve for the point $N = 200$. Here we have 100 assigned agents and 100 unassigned agents and this has better throughput than having all 200 agents being assigned (the gray curve at $N = 200$). The reason is as follows. For $N = 200$ and $C = 100$ assume that a single assigned agent reached its target. We now have 101 unassigned agents. Next a new task t is popped from T. The closest agent among these 101 agents is assigned to t and its task is easy as it is very close to the target. This is better than having 199 assigned agents and only one unassigned agent. That unassigned agent must be picked and it might be very far away from the goal. The sweet spot for $C = 100$ is $N = 300$. Beyond $N = 300$, adding more agents will not help a lot for having closer agents to the popped tasks, but will certainly add more density to the graph and more conflicts. Thus, more bypasses will be made by the assigned agents, the paths get longer and this hurts the throughput. Notably, the best throughput is achieved for $N = 600$ with $C = 400$. That is, for a 32×32 grid, even if one has an infinite number of agents it is best to only place 600 of them in the grid and only allow 400 of them to execute tasks concurrently. Figure 4 shows the same results but for 16×16 grid. Very similar trends are observed.

Explanation Figure 5 explains the above trends. We have curves for C values of 200 (yellow), 400 (red), 600 (brown) and 800 (gray). For each C value (color) and number of agents N (x-axis) the doted line gives the average distance (y-axis, measured by Manhattan Distance) of the closest unassigned agent that was then assigned to the popped task. The dotted curves keep reducing as with more agents the distance to the closest agent is likely to reduce. Even for the AAA case (gray line, $C = 800$) when N gets larger more agents may arrive at their target and become unassigned at the current time step. The allocation function spreads them efficiently to the new tasks and the dotted line drops. For the other dotted curves, when N becomes larger than C, then $N - C$ unassigned agent start to be present. There are significantly more potential unassigned agents to assign

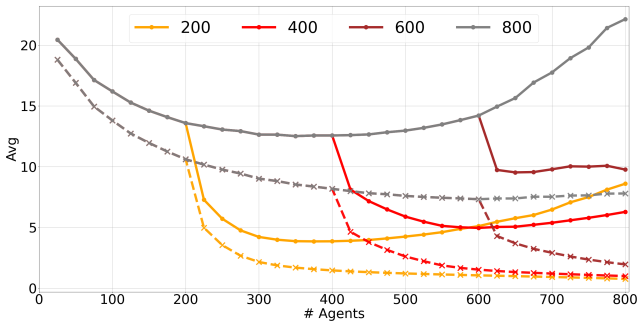

Figure 5: Path traveled vs. heuristic estimation

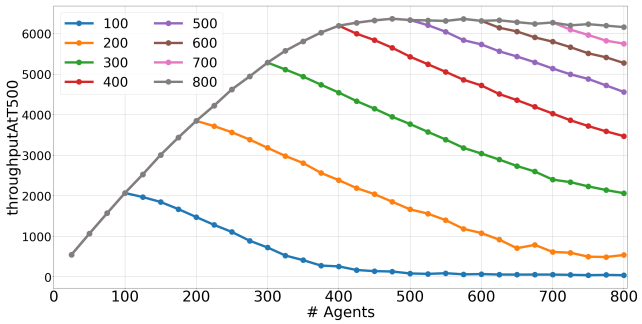

Figure 6: AT results 32×32 Random allocation.

from and the dotted line dramatically decreases (with a diminishing return later). By contrast, the solid curves report the average distance traveled by the assigned agents for the same scenario. In fact, the dotted lines can be seen as an admissible heuristic for the solid lines. Importantly, as more agent are present, the environment becomes dense, more conflicts occur and the average path traveled starts to increase. Thus, even though agents are closer to their goals (dotted curves) their path to the goals are much longer and the total throughput increases (solid curves). Therefor, while the dotted curves always decrease the solid curves start to increase after the sweet spot.

Random Allocation Figure 6 shows the same results but for the Random allocation policy (simulates external unknown allocation algorithms). Even the AAA case (gray curve for $C = 800$) has much worse throughput than that of the former experiment (Figure 3, Closest allocation). The reason is that at each time step several agents fulfilled their tasks. Then, they were all assigned to new tasks. Randomizing these assignments along them provide much longer distances to travel than assigning the closest agents to the goals. Furthermore, with Random once more unassigned agents appear (along the x-axis) overall throughput decreases. Under random allocation, these additional agents do not contribute to reduce the paths traveled. To the contrary, they do contribute to the density of agents and to adding more conflicts. Overall, the results indicate that under Random allocation it is preferable to have all existing agents as assigned agents while transferring some of the agents to be unassigned agents is harmful.

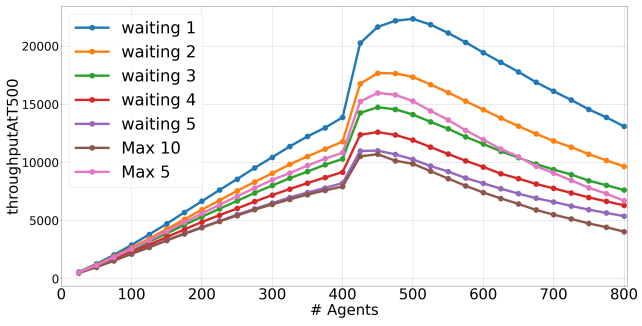

Figure 7: AW results at $C = 400$. Adjusted closest heuristic.

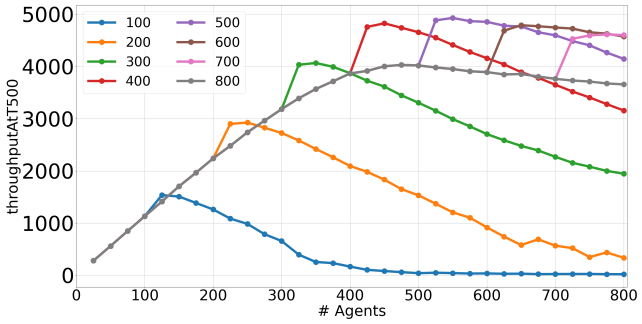

Figure 8: PD results using LaCAM Heuristic UA strategy.

4.2 The AW and PD tasks

Figure 7 deals with the case of *arrive and wait* task (AW). It illustrates the impact of increasing the required waiting time at the goal (k) on the throughput for the case of $C = 400$. There is a curve for each of $k = 1 \ldots 5$. There is also a curve for k distributed uniformly in the range of $1 \ldots 5$ and $1 \ldots 10$. One can observe that larger values of k decrease the throughput as more waiting needs to be done. Here too, adding more unassigned agents is beneficial up to a certain point. Note, however, that longer waiting times lead to earlier peaks because agents remain occupied at target locations for longer, reducing the effective rate of task completion and increasing congestion near targets.

Figure 8 shows results for the *pickup and delivery* tasks (PD). Here, the trends are similar to the AT case. However. here, the peaks appear earlier. For example, the best throughput is achieved for $N = 550$ and $C = 500$. The agent allocation is only relevant for the pickup tasks and the existence of nearby unassigned agents is less influential than for AT.

4.3 Comparing best variants

An important question is: if one wants to attain the maximum throughput in a given domain, how many agents should be used and of these agents what is the best split between assigned and unassigned agents. We performed the same experiment of Figure 3 for different $n \times n$ grids (with the closest allocation policy and the AT task). Table 1 provides the mixture of assigned and unassigned agents that produced the maximum throughput for the given domain. It reports the total number of agents (N), their split to assigned agents (AA) and unassigned agents (UA) and the throughput achieved

Size		Mixed AA and UA					All AA		Imp
n	n^2	N	AA	UA	alr	THR	AA	THR	Imp
8	64	50	45	5	0.78	3,271	45	2,610	25%
10	100	75	65	10	0.75	4,999	75	3,617	38%
12	144	105	85	20	0.72	7,004	100	4,740	48%
14	196	135	110	25	0.68	9,327	135	5,988	56%
16	256	180	140	40	0.70	11,831	165	7,335	61%
20	400	265	190	75	0.66	17,814	255	10,281	73 %
24	576	320	240	80	0.55	23,563	320	13,199	78%
28	784	500	345	155	0.63	32,369	475	17,143	89%
32	1,024	670	460	210	0.65	41,285	600	21,121	95%

Table 1: Maximum throughput for different empty grids

Task	Mixed AA and UA					All AA		Imp
Task	TA	AA	UA	alr	THR	AA	THR	Imp
32×32 empty grid. 1024 cells								
AT	670	460	210	0.65	41,285	600	21,121	95%
AW5	725	700	25	0.70	14,070	675	10,756	31%
PD	525	475	50	0.51	4,955	475	4,030	23%
32×32 Maze. 666 cells								
AT	250	170	80	0.37	2,472	150	781	217%
AW5	190	180	10	0.28	1,118	170	715	56%
PD	160	140	20	0.24	397	130	304	31%
Warehouse (3-9-8-2-1). 1,024 cells								
AT	400	300	100	0.39	6,532	300	2,404	172%
AW5	325	300	25	0.31	3,713	300	2,260	64%
PD	275	250	25	0.26	1,016	250	784	30%
Room (32-32-4). 682 cells								
AT	200	175	25	0.29	3,944	225	2,091	89%
AW5	325	300	25	0.47	2,139	225	1,655	29%
PD	200	175	25	0.29	969	175	709	36%

Table 2: Maximum throughput for different environments

(THR). Felner and Stern (2026) defined the notion of *agent location ratio* (alr) which is the ratio between the number of agents and the number of empty locations (in our table this corresponds to $alr = N/n^2$) which is also presented in the table. The best results are achieved with $alr = 0.78$ for the 8×8 grid which constantly decreases slightly to $alr = 0.65$ in the 32×32 grid. Also, one can observe that the best balance between unassigned and assigned agents starts at 10% for the relative number of unassigned agents out of all agents (for the 8×8 grid), and this and goes up to 31% for the 32×32 grid. The next two columns are for the case where *all agents* must be assigned agents (AAA, similar to the gray curve above). The *Imp* column shows the improvement of the *mixed AA and UA* case over the *all agents assigned* AAA case. For example, for 12×12 the best number of agents in the AAA case is 100 while for the *Mixed AA and UA* case the best number is to have 105 agents (85 assigned and 20 unassigned). This yields a 48% increase in the throughput (from 4,740 to 7,004). Importantly, as the size of the domain increases this improvement also increases. It starts at 25% for the 8×8 grid and goes up to 95% for the 32×32 grid.

Other domains We have also applied LaCAM$_{UA}$ with CFET on other domains from the Moving AI repository (Sturtevant 2012). Table 2 shows representative results

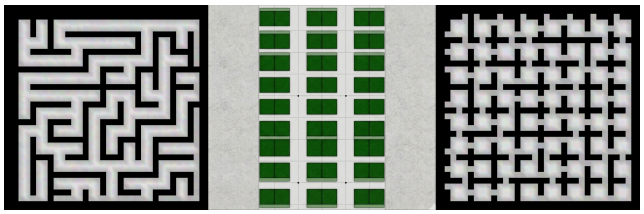

Figure 9: The Maze, Warehouse and the Room

on a maze, a warehouse, and a rooms map (all are shown in Figure 9), together with the results on our 32×32 grid. We again compare the best mixture of assigned and unassigned agents to the best number of agents of the AAA case for the three tasks: AT, AW and PD. Similar trends are seen. The best variant with mixed assigned and unassigned agents always significantly improves on the best variant of the AAA case. The improvement is more significant for the maze, warehouse and the room map than for the empty grid because the former domains had more corridors and the presence of unassigned agents greatly improved the lengths of the paths traveled. For all domains, the improvement is larger for AT than for AW and for PD. The reason is again that the allocation function has a major influence on arriving to the target location in AT but its influence decreases when other tasks need to be done at the target location such as waiting or going away to deliver.

Summary of the Experiments Our experimental results show the following clear trends: **(1)** It is always beneficial to limit the total number of agents to avoid congestion. **(2)** It is better to have a mixture of assigned and unassigned agents over having only assigned agents. **(3)** The improvement of intentionally adding unassigned agents is most significant for tasks such as AT but is also evident in AW and PD.

5 Summary and Conclusions

We presented a general framework for LMAPFUA and introduced a new variant of LaCAM$_{UA}$ and a new stopping condition for LMAPF algorithms. We then studied how the throughput is affected by the total number of agents and by their division into assigned and unassigned agents. Clearly, the advantage of limiting the total number of agents as well as adding unassigned agents was demonstrated and justified.

This was just the first paper on this topic. There are yet many open questions and many directions to continue in the future as we now list. **(1)** Given an environment, what is the exact optimal number of assigned and unassigned agents. While we showed some trends, a much deeper research is warranted to fully answer this question. For example, machine learning techniques can be applied to learn such a model from a massive number of examples. **(2)** As side topics, we have introduced a few algorithmic variants such as the new low-level for LaCAM$_{UA}$ as well as the new stopping condition of CFET. Deeper treatment of these algorithms deserve their own future papers. **(3)** Finally, the LMAPFUA setting can be extended to include more possible tasks, more complex allocation functions, and a more sophisticated mechanism to pop tasks from the stream of tasks.

Acknowledgments

This research was supported by a grant from the Israeli Ministry of Science and Technology (MOST) to Ariel Felner and Roni Stern. The research reported in this paper was also supported by an Amazon Research Award, Fall 2025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of Amazon.

References

Felner, A.; and Stern, R. 2026. Blue-Sky: Multi-Agent Path Finding with Unassigned Agents (MAPFUA). In *AAAI Conference on Artificial Intelligence*.

Fu, B.; Chen, Z.; Chandan, R.; Barbosa, A.; Caldara, M.; Durham, J.; and Pecora, F. 2025. Symbolic Planning and Multi-Agent Path Finding in Extremely Dense Environments with Movable Obstacles. *arXiv preprint arXiv:2509.01022*.

Gabay, N.; Morag, J.; Felner, A.; and Stern, R. 2026. The Reachability Objective in Multi-Agent Path Finding. In *AAMAS*.

He, J.; Zhang, Y.; Veerapaneni, R.; and Li, J. 2024. Scaling Lifelong Multi-Agent Path Finding to More Realistic Settings: Research Challenges and Opportunities. In Felner, A.; and Li, J., eds., *Seventeenth International Symposium on Combinatorial Search, SOCS 2024, Kananaskis, Alberta, Canada, June 6-8, 2024*, 234–242. AAAI Press.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021. Lifelong multi-agent path finding in large-scale warehouses. In *AAAI Conference on Artificial Intelligence*, volume 35, 11272–11281.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In Larson, K.; Winikoff, M.; Das, S.; and Durfee, E. H., eds., *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, 837–845. ACM.

Morag, J.; Gabay, N.; Koyfman, D.; and Stern, R. 2025. Should Multi-Agent Path Finding Algorithms Coordinate Target Arrival Times? In *International Symposium on Combinatorial Search (SoCS)*, volume 18, 231–235.

Okoso, A.; Otaki, K.; Koide, S.; and Nishi, T. 2022. High Density Automated Valet Parking via Multi-agent Path Finding. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, 2146–2153.

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

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

Pertzovsky, A.; Stern, R.; and Zivan, R. 2024. CGA: Corridor Generating Algorithm for Multi-Agent Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3455–3462.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Symposium on Combinatorial Search (SoCS)*.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Trans. Comput. Intell. AI Games*, 4(2): 144–148.

Surynek, P. 2010. An optimization variant of multi-robot path planning is intractable. In *AAAI conference on artificial intelligence*, 1261–1263.

Veerapaneni, R.; Saleem, M. S.; Li, J.; and Likhachev, M. 2025. Windowed MAPF with completeness guarantees. In *AAAI Conference on Artificial Intelligence*, 23323–23332.

Yu, J.; and LaValle, S. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference on Artificial Intelligence*, 1443–1449.