LIMP: Incremental Multi-agent Path Planning with LPA*

Mücahit Alkan Yorgancı[1], Fatih Semiz[1,2] and Faruk Polat[1]
[1]*Middle East Technical University, Computer Engineering Department, Ankara, Turkey*
[2]*Aselsan Inc. Defense Systems Technologies Business Sector, Ankara, Turkey*

Keywords: AI, Multi-agent, Pathfinding, MAPF, Incremental Planning.

Abstract: The multi-agent pathfinding (MAPF) problem is defined as finding conflict-free paths for more than one agent. There exist optimal and suboptimal solvers for MAPF, and most of the solvers focus on the MAPF problem in static environments, but the real world is far away from being static. Motivated by this requirement, in this paper, we introduce an incremental algorithm to solve MAPF. We focused on discrete-time and discrete space environments with the unit cost for all edges. We proposed an algorithm called incremental multi-agent path planning with LPA* (LIMP) and discrete lifelong planning A* (DLPA*) for solving I-MAPF (Incremental MAPF). LIMP is the combination of two algorithms which are the Conflict Based Search D*-lite (CBS-D*-lite) (Semiz and Polat, 2021) and DLPA*. DLPA* is just a tailored version of the lifelong planning A* (Koenig et al., 2004) which is an incremental search algorithm for one agent. We have shown that LIMP outperforms Conflict Based Search replanner (CBS-replanner) and CBS-D*-lite (Semiz and Polat, 2021) in terms of speed. Moreover, in terms of cost, LIMP and CBS-D*-lite perform similarly, and they are close to CBS-replanner.

1 INTRODUCTION

The multi-agent pathfinding (MAPF) problem is the extended version of the classical pathfinding problem for more than one agent. In the MAPF problem, agent paths should not conflict. In other words, agents can not be in the same location at the same time. Semiz and Polat (Semiz and Polat, 2021) extended the definition of MAPF and came up with a new variant of MAPF called incremental multi-agent pathfinding (I-MAPF). In the I-MAPF, the environment can change during the search; in other words, some paths can be blocked. This new approach better covers real-world problems. As the environment is not static in the real world, environmental changes occur when planning a path for agents. For example, if we are planning the routes for cargo delivery trucks, the path costs may change due to the traffic jams, or some of the paths might be entirely blocked because of accidents. In this scenario, classical MAPF algorithms cannot reflect these changes efficiently and generally plan the routes from scratch.

Several algorithms are developed to handle the environmental changes efficiently for both MAPF and classical path planning problems such as CBS-replanner, CBS-D*-lite and LPA*. More algorithms are pointed at the related work and the background sections. Conflict Based Search (CBS) is an algorithm that aims to solve the MAPF problem by solving the conflicts in the agent paths by adding constraints (Sharon et al., 2015). CBS-replanner is the extended version of the CBS that replans the paths after an environmental change occurs. In this paper, we proposed two algorithms named incremental multi-agent path planning with LPA* (LIMP) and DLPA*. DLPA* is a version of a former incremental pathfinding algorithm lifelong planning A* (LPA*). LIMP combines DLPA* with Conflict Based Search D*-lite (CBS-D*-lite). The intuition behind the DLPA* is that environmental changes (path blocking) generally affect only the nodes close to the node that the change occurred. So, the algorithm only re-expands the nodes near the change, and if the cost of the node changes, it re-expands the children of this node.

2 RELATED WORK AND BACKGROUND

2.1 Incremental Pathfinding Algorithms

There are many studies focusing on incremental single-agent pathfinding in the literature. Lumel-

Yorgancı, M., Semiz, F. and Polat, F.
LIMP: Incremental Multi-agent Path Planning with LPA.
DOI: 10.5220/0010824400003116
In *Proceedings of the 14th International Conference on Agents and Artificial Intelligence (ICAART 2022) - Volume 1*, pages 208-215
ISBN: 978-989-758-547-0; ISSN: 2184-433X

sky and Stepanov (Lumelsky and Stepanov, 1986), Pinadeh and Snyder (Pirzadeh and Snyder, 1990), Korf (Korf, 1990) and Zelinsky (Zelinsky, 1992) tried to use existing information to re-plan the path dynamically. Stentz tried to solve the problem by generalizing the A* algorithm by making it responsive to environmental changes to use in dynamic environments. He developed two algorithms D* (Stentz, 1993) which partially changes the f and g values of the vertexes when environmental change occurs, and Focused D*, which is the improved version of the D* (Stentz, 1995) that focuses on updating the costs to reduce the number of expanded states. Later, Ramalingam came up with an incremental algorithm called DynamicSWSF-FP (Ramalingam and Reps, 1996) to solve a similar problem which is a grammar problem. DynamicSWSF-FP is capable of handling the arbitrary number of edge insertion, edge deletion, and cost changes. Koenig (Koenig et al., 2004) combined A* and DynamicSWSF-FP in Lifelong Planning A*. He also proposed D*-Lite, which behaves similarly to D* but algorithmically different. D*-lite is shorter than the former D*. It includes fewer conditional branches, which makes it easier to implement and extend (Koenig and Likhachev, 2002).

Unlike traditional A*, LPA* is capable of handling changes by just expanding the nodes that are affected by the change. It checks whether the shortest path from the start to the node is subject to the edge cost change. The algorithm does that by taking a minimum of the paths adding edge costs to the parents of the node. If the shortest path is changed, LPA* updates the node's g value then marks its children as their shortest path might also be changed. Child nodes are pushed to the open list to be expanded later. LPA* is complete and optimal if the heuristic is consistent and non-negative.

2.2 MAPF Solvers

MAPF solvers might be optimal, sub-optimal, reduction-based that reduces the problem to the well-studied one, centralized or distributed. Solvers may have different cost functions such as sum-of-costs, makespan, etc. (Sharon et al., 2015). Standley (Standley, 2010) proposed an improvement that employs grouping the agents that do not affect other groups of agents to reduce the state-space. Later, M* (Wagner and Choset, 2011) which dynamically changes the branching factor, is released. Sharon et al. developed Conflict Based Search (Sharon et al., 2015), and Boyarski et al. (Boyarski et al., 2020) provided an iterative-deepening variant of CBS that uses LPA* (instead of A*) to find low-level paths of the agents.

Conflict based search (CBS) is an algorithm that aims to find an optimal solution for the multi-agent pathfinding problem. CBS has two levels: low-level search and high-level search. The low-level search is a single agent path planning algorithm that is capable of satisfying some constraints. In the CBS, a constraint is formulated as a tuple that contains agent id, location, time information, which indicates that a given agent cannot be present in the given location at the given time. High-level search constructs a tree called a conflict tree to solve the path conflicts between agents. A CBS node consists of constraints and planned paths. A conflict occurs when more than one agent plans to be at the same position at the same time step. High-level search expands nodes by adding a constraint to one of the conflicting agents at conflict position. As it expands the nodes in the increasing cost order, CBS is known to be complete and optimal.

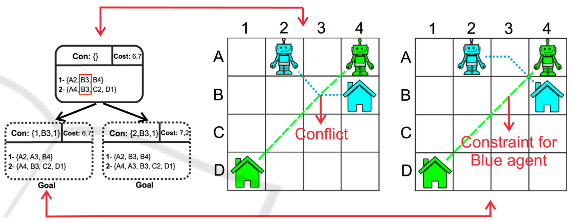


Figure 1: Example run of the CBS with a hypothetical low-level search. Red lines pair the CBS nodes with the corresponding low-level search result. As can be seen, there is a conflict at (B,3) when time equals 1. CBS expands the root by constraining the blue and green agents to present at (B,3) when time is 1 (Semiz and Polat, 2021).

2.3 Dynamic MAPF

There are few studies that solve the dynamic version of the MAPF problem. Murano et al. (Murano et al., 2015) studied multi-agent pathfinding in a dynamic but predictable environment. Atiq et al. (Atiq et al., 2020) studied a dynamic version of MAPF in which existing agents may leave and new agents may join the environment at different times. Wan et al. (Wan et al., 2018) worked on a lifelong dynamic MAPF problem where the dynamism of the problem comes from adding a new agent to the environment. Semiz and Polat (Semiz and Polat, 2021) worked on the I-MAPF problem where nodes in the environment can become temporarily unavailable. They provided CBS-replanner and CBS-D*-lite solutions. Unlike classical CBS, when an environmental change occurs, CBS-D*-lite tries to use a previously constructed conflict tree to get the solution faster.

CBS-D*-lite has three stages of handling environmental changes. After an environmental change occurs, CBS-D*-lite starts with its first stage and tries to find a sub-optimal solution that satisfies some optimization criterion by performing the low-level search on the goal node of the previously constructed conflict tree. If it fails to find such a solution, it moves to the second stage, in which it performs a high-level search on a tree whose root is the previous goal node. If it fails again, it just simply constructs the conflict tree from scratch with the new environment. CBS-D*-lite uses D*-lite in the low-level search. The algorithm is complete but not optimal (Semiz and Polat, 2021).

3 INCREMENTAL MAPF WITH LPA*

The main problem that we are solving is the I-MAPF, an extended version of the MAPF problem. The input of the I-MAPF problem consists of:

- A set of k agents $\{a_0, \cdots, a_k\}$
- An undirected unweighted graph $G = (V, E)$ which represents the environment. In the scope of this paper we used 4-connected grid world.
- A set of starting vertices $\{s_0, \cdots, s_k\}$ where s_i is the starting vertex of a_i.
- A set of goal vertices $\{g_0, \cdots, g_k\}$ where g_i represents the goal vertex of a_i
- A set of environmental changes $E = \{(v_0, t_0, \Delta_0), \cdots, (v_n, t_n, \Delta_n)\}$ where v_i is the vertex which will be blocked. t_i is the time when the vertex is started to be blocked. Lastly, Δ_i is the blocking period of the vertex. To sum up, an agent can not present at v_i between t_i and $t_i + \Delta_i$

The output of the I-MAPF problem is set of paths $\{P_0 ... P_k\}$ where P_i is a sequence of vertexes with time information without conflict or block. Formally, $P_i = \{(v_0, 0), (v_1, 1), \cdots, (v_i, i), (v_{i+1}, t_{i+1}), \cdots, (v_m, m)\}$ where v_0 is the start vertex of a_i and v_m is the goal vertex of a_i (i.e., $v_0 = s_i$ and $v_m = g_i$). Formally, a conflict is vertex, time and set of agents tuple $c = (v, t, A)$ where $|A| > 1 \wedge \forall a_i \in A(v, t) \in P_i$. Blocking occurs when there is a blocked vertex-time tuple in the path. Formally, (v, t) is blocked if $\exists (v', t_0, \Delta_t) \in E$ $v' = v \wedge t_0 < t < t_0 + \Delta_t$. Lastly, a vertex cannot be blocked if there exists an agent.

In the high-level search, LIMP uses CBS-D*-lite as it is. On the other hand, we have developed a modified version of LPA* named discrete lifelong planning A* (DLPA*) for the low-level search of CBS-D*-lite. The new version is simpler, making it easier to implement and modify. In MAPF, an agent might wait or come to the same position at different time-steps to find an optimal solution. Hence, our version is working in the discrete time-space domain. In addition to classical LPA*, our version has the ability to add constraints beside obstacles. The DLPA* has four main procedures, which are replan, addConstraint, and makeChange, advance.

Algorithm 1: DLPA*.

```
Require: I-MAPF simulation {Simulation S}
changes = S.new_environmental_changes
for change c_i in changes do
    makeChange(c_i)
replan()
advance()
```

Replan is the main procedure of the LIMP (Algorithm 2). Whenever an agent path is needed to be replanned, this procedure is called with the former Open and Closed lists. States of the LIMP contain x coordinate, y coordinate and time information. The parent of a state is like the parents of a state in the space domain, except the time of the parent is one less than the time of the child; similarly, the time of a child is one more than the time of the parent. In this way, waiting action can be handled by only advancing the time. The getParents and getChildren functions return the set containing the parents and children, respectively. These functions also check obstacles and constraints and exclude the ineligible states.

Algorithm 2: Replan.

```
Require: Path Planning Instance {OPEN, CLOSED, Goal}
while OPEN.notEmpty() AND getPath() = NULL do
    S ← OPEN.popHighestPriority()   ▷ S : {x,y,t}
    if getParents(S) = { } then
        CLOSED.delete(S)
        for each state s_i in getChildren(S) do
            if CLOSED.get(s_i) then
                OPEN.add(s_i)
    else
        CLOSED.add(S)
        for each state s_i in getChildren(S) do
            if CLOSED.get(s_i) = NULL then
                OPEN.add(s_i)
Return getPath()
```

LIMP maintains two sets in low-level that are Open and Closed. Open set stores the states that are subject to a change and should be examined. Closed set stores the states that have a path from the start state. The popHighestPriority function returns the state with the highest priority with respect to the heuristic. In this paper, we used Manhattan distance as a heuristic. The notEmpty function returns true if the set is not empty, false otherwise. The add function adds the given state to the given set. Both Open and Closed sets are behaving like formal sets, not allowing the duplicates.

The procedure takes states from the Open list until reaching the goal. The goal state is the state that has the exact coordinates as the goal location. The getPath function returns the path performing a backward search if the goal is found. As after adding a constraint or object, the shortest path cannot be shorter than the previous shortest path; we first check whether a path from the goal node to the start node exists; if so, we return this path. The getPath function can be implemented as a constant time operation when there is no path by maintaining the information about the goal state is found or not. If the popped state has no parents, that means the state is unreachable. So, it is deleted from the Closed list. If a state becomes unreachable, that may lead its children to become unreachable, so children of the deleted state are pushed to the open list. If a state becomes reachable, its children are also pushed to the Open list to be expanded further if they are not on the Closed list.

Algorithm 3: addConstraint.

Require: {OPEN, CLOSED, Constraint }
CLOSED.delete(Constraint)
OPEN.delete(Constraint)
for each state s_i in getChildren(Constaint) **do**
  if CLOSED.get(s_i) **then**
    OPEN.add(s_i)

Adding constraint to the LIMP is relatively simple (Algorithm 3). The constrained state is deleted from the closed set and its children are pushed to the open list as they might be unreachable.

Algorithm 4: makeChange.

Require: {OPEN, CLOSED, Position, t, Δ_t } ▷ Position : { x, y}
i← t
while $i \leq t + \Delta_t$ **do**
  S← Position.x, Position.y, i
  addConstraint(S) $i \leftarrow i + 1$

As environmental changes cover a time period, unlike the constraints, all the states having the exact coordinates with the change and being in the given period should be deleted from the closed list, and children of them should be pushed to the open set to be deleted further (See Algorithm 4). Additionally, as stated in the description, the problem only covers environmental changes that constrain states; hence obstacle removal is not implemented, but it can be implemented easily by tailoring the makeChange function.

Algorithm 5: advance.

Require: {OPEN, CLOSED, t }
$S_t \leftarrow getCurrentState()$ ▷ P_t : { x, y, t}
for each state S **do**
  if $S.t = t$ and $S \neq S_T$ **then**
    addConstraint(S)

The advance function simulates one time-step of advancement. As after time-step (t) being a state other than the recently realized state becomes impossible, the function adds constraints to these states. For example, after three advancements, if an agent is at (2,2,3) it cannot be present (2,1,3), so this state should be constrained. The getCurrentState function returns the state which the agent currently is. The advance function does not change the calculated path; however, it affects further replannings. Note that, in implication, we have used a priority queue (prioritizing lower time-steps) storing states in OPEN and CLOSED sets in order not to inspect all states.

3.1 Example Run

In this section we will show an example run of DLPA* with a simple planning case including a constraint addition. The obstacle addition is just consecutive constraint addition we just only considered a constraint addition for simplicity.

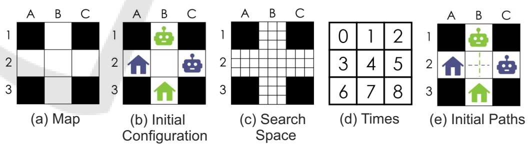


Figure 2: Example run.

Consider the environment in Figure 2 (a). In Figure 2 (b), starting coordinates of agents are shown as a robot figure, and goal coordinates with a house figure having the same colour as the agent. Figure 2 (c) shows the search space of the agents. As we are searching in the time-space domain, we divided every coordinate into nine time spaces. All the squares in the Figure 2 (c) represent a time-space tuple. Every coordinate has nine states starting from the upper left corner, as in Figure 2 (d).

The initial run of the LIMP is indifferent to the classical CBS and the initial run of the DLPA* is indifferent to the A* in three dimensions. After the initial run, agents compute their paths as in Figure 2 (e). After the initial plannings, the search space of the purple agent can be seen in Figure 3 (a) (initial search space). The green states are the states in the closed set, and the yellow states represent the states in the Open list. The path of the purple agent can be easily obtained by performing a backward search from the state (A,2,2). The path is (C,2,0) → (B,2,1) → (A,2,2).

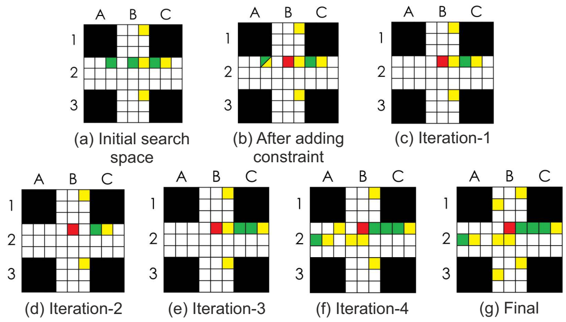


Figure 3: State spaces of the example run.

It can be easily seen that there is a conflict at the state (B,2,1). The high-level search constraints one of the agents. In the example, we assumed that the purple agent is constrained from being at (B,2,1). The addConstraint function deletes the state (B,2,1) from the Closed list and pushes (A,2,3) to the Open list as it is in the Closed list and it is a child of (B,2,1). Figure 3 (b) (search space after adding constraint) represents the search space of the purple agent after the addConstraint function is performed. The red square indicates the constrained state.

The algorithm takes the state (A,2,2) as it has the highest priority. The state has no parents; in other words, (A,2,1) is not in the closed list, (B,2,1) is constrained, and there is an obstacle at both (A,1,1) and (A,3,1). So, the state (A,2,2) is deleted from the closed list. None of the children of the state is in the Closed set, so the algorithm does not add any state to the Open set. Figure 3 (c) (iteration-1) shows the search space after the state (A,2,2) is popped and examined.

At next iteration, (C,2,1) and (B,2,2) have the highest priorities. Assume that (B,2,2) is chosen. The algorithm pops it from the Open set. As it has neither a parent nor a child and it is not in the Closed set, it is basically deleted from the Open set. Figure 3 (d) (iteration-2) represents the search space after the iteration.

Then, the algorithm picks the state (C,2,1). It is added to the Closed set, and its children (B,2,2) and (C,2,2) are pushed to the Open set. Figure 3 (e) (iteration-3) shows the search space after this iteration.

After these iterations, states (B,2,2) and (A,2,3) are popped from the Open set, respectively and their children are added to the Open set. Figure 3 (f) (iteration-4) shows the iterations, respectively.

So the new path of the purple agent can be found easily by performing backward search from (A,2,3). The resulting path is (C,2,0) → (C,2,1) → (B,2,2) → (A,2,3). The final iteration can be seen at Figure 3 (g) (final iteration). As the computed path of green agent was (B,1,0) → (B,2,1) → (B,3,2), there is no conflict in the paths so the LIMP terminates returning given paths.

3.2 Theoretical Analysis

In this section, we will show that DLPA* is optimal. In addition, both DLPA* and LIMP are complete. The DLPA* has the same flow as the LPA*. As in the discrete domain with the constant cost, a vertex is either connected to a vertex, meaning that it has an edge with this vertex with cost one, or disconnected to a vertex, meaning that it has an edge with this vertex with cost infinity. Assigning the cost of an edge to infinity has the same effect as deleting the edge from the map.

In the space-time domain, a state can have an exact g-value (due to the time property) and an exact f value (due to the coordinates). So, changing the cost of a state is only possible when a state is deleted or newly examined. The cost change in the LPA* corresponds to the state deletion or addition in the DLPA*.

Adding a constraint in the DLPA* corresponds to setting all the incoming edges to infinity in the LPA*. Performing environmental change also corresponds to setting proper edges to infinity.

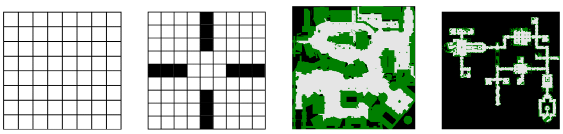
Although advance function in DLPA* have no equivalent in LPA*, adding a constraint to all states in time t except a state s results in the same search tree with a search rooted at (s,t). So, it does not break the optimality or completeness of DLPA*.

As all the operations correspond to an operation in the LPA*, the DLPA* is just a particular instantiation of the LPA*. That implies the DLPA* is also complete, sound, and optimal as the LPA* given that the underlying heuristic is consistent. However, if there exists no solution, the algorithm does not terminate. The termination condition should be checked beforehand.

CBS-D*-lite is not optimal; hence the LIMP is also not optimal but complete.

4 EXPERIMENTAL STUDY

There are no studies except the CBS-replanner and CBS-D*-lite for solving the I-MAPF problem (Semiz and Polat, 2021). Hence, we compared our results with the CBS-replanner and the CBS-D*-lite. We run benchmark tests for all of the three algorithms in 8 configurations. A configuration consists of a map, a number of agents, a number of environmental changes. We also used varying environmental change frequencies (4,6,8,12 and 24 number of environmental changes in one time-step) with different minimum and the maximum manhattan distances between the agent's start and the goal location. Maximum distance is added to the configuration as the distance highly affects memory usage, and our test environment has limited memory. We have used four maps which are named 8x8, cross, den520d and brc202d, given in 4. As all of the algorithms compute different paths at the root of the search, we have chosen different environmental changes for different algorithms. This randomness may make individual test cases biased. To make this not affect the experiment results, we tried to run various test cases so the distribution of the test case results might give better insights. We have run 12200 test cases in total. As some of the cases do not have a solution or some of the environmental changes may lead dramatic increase in the elapsed time, which will create abnormality, we limited one test run to elapse at most 300 seconds and discarded the cases that one of the algorithms is exceeded the limit. In addition to the above, we also run handcrafted experiments to check the completeness of the LIMP and optimality of the DLPA*.



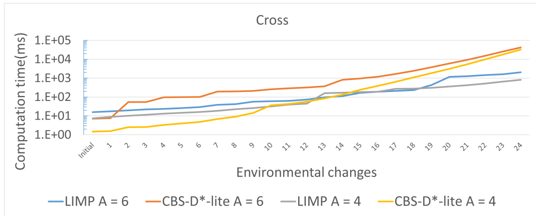
(a) 8x8 (b) Cross (c) Den520d (d) Brc202d

Figure 4: Maps used in the experiments.

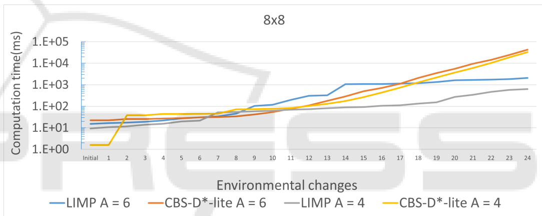
We have tested algorithms in 8 configurations in the maps 8x8, cross, den520d and brc202d maps. Den520d and brc202d are the maps taken from Dragon Age: Origins game. These maps are commonly used in benchmarking MAPF algorithms (Sturtevant, 2012). 8x8 environment has an 8x8 grid without any obstacles. The Cross environment is a 9x9 grid map with a narrow crossing point in the middle making agents congest more. We have experimented with small maps (8x8 and 9x9) with 4 and 6 agents and larger maps (den520d and brc202d) with 10 and 15 agents. Lastly, we applied 24 environmental changes to all configurations.

We have given the results of each configuration in the following subsections. In the graphs, the x dimension is the number of environmental changes, the y dimension is the computation time (ms), and K represents the number of agents. Note that; results are reported on a logarithmic scale to fit in the graph. We have implemented the algorithms using C++. We have run a benchmark test on a test computer having AMD Opteron 6376 1.4 MHz core.

4.1 Small Map Scenarios



(a) cross



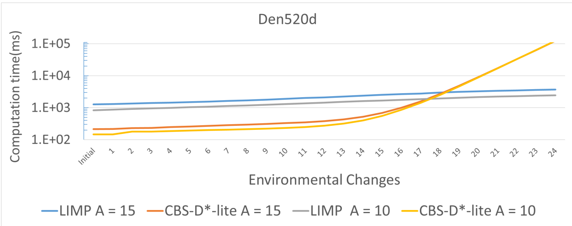
(b) 8x8

Figure 5: Test results of small map scenarios.

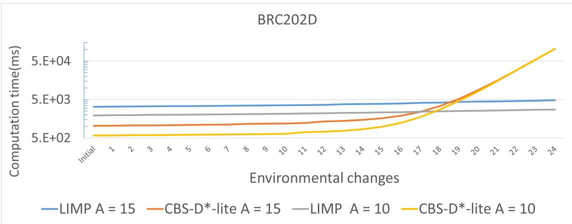
We have run tests on small maps to investigate the performance of LIMP where more congestion happens. 8x8 and cross maps are used for this set of experiments. Since the maps are small, an increase in the agent count results in high computation times, making test cases fail. So we run these cases with a low number of agents (4 and 6). Figure 5 (a) and (b) shows the results of the 8x8 map, and Figure 5 (c) and (d) shows the results of the cross map. For 8x8 map, we have run 1500 and 500 test cases with 4 and 6 agents, respectively. In the cross map (Figure 4b), the walls separate the map into four sub-parts. The centre of the map has no obstacles, so agents can move between the sub-regions, and the crossing region creates congestion. We have run 2000 and 700 test cases for the cross map with 4 and 6 agents.

In both maps, LIMP performed worse in the first environmental changes. On the contrary, it outperformed CBS-D*-lite after several environmental changes.

4.2 Large Map Scenarios



(a) Den520d



(b) Brc202d

Figure 6: Test results of large map scenarios.

Den520d and brc202d (Figure 4c,d) maps are used to test the algorithms, common benchmark maps in the literature. The number of agents is taken as 10 and 15. We have run 2000 and 1000 tests for 10 and 15 agents for each map, respectively. Additionally, we have tested LIMP with 20, 25 and 30 agents, which ended with the same results; however, as eliciting randomness factor requires a high number of test cases, we did not include these in this paper. Figure 6 a show the results of the den520d map, and Figure 6b include the results of the brc202d map.

LIMP, in both maps, performed better than CBS-D*-lite after the 19th environmental change. As a result, LIMP is a better option in larger environments where environmental changes are frequent.

4.3 Comparison with CBS-replanner

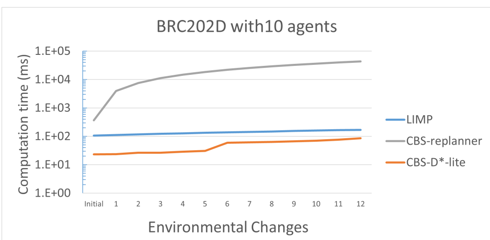


Figure 7: Computation time comparison of I-MAPF solvers.

In this section, we compared computation times of the CBS-replanner, the CBS-D*-lite and LIMP. Because of the high computational time of the CBS-replanner, we did not include it in other configurations. In addition, we lowered the environmental change count to 12 and only tested with ten agent configurations. We have run 1500 tests in the brc202d map. LIMP and CBS-D*-lite strongly outperformed CBS-replanner as expected. Figure 7 shows computation times of the three algorithms tested.

4.4 Comparison of Path Costs

Table 1: Cost analysis of brc202d map with 10 agents.

	LIMP	CBS-D*-lite		LIMP	CBS-D*-lite
0	31.59	31.59	13	32.49	31.93
1	31.65	31.59	14	32.56	31.95
2	31.72	31.65	15	32.62	31.98
3	31.80	31.65	16	32.70	32.01
4	31.86	31.69	17	32.76	32.03
5	31.92	31.73	18	32.82	32.06
6	31.98	31.75	19	32.89	32.07
7	32.06	31.78	20	32.96	32.09
8	32.13	31.80	21	33.04	32.11
9	32.20	31.82	22	33.11	32.14
10	32.27	31.85	23	33.16	32.16
11	32.34	31.87	24	33.24	32.18
12	32.41	31.91			

In this section, we compared the total path cost values of the CBS-D*-lite and LIMP. As CBS-D*-lite has a decent success rate at total path cost, this comparison will give us an important overview of the success rate of LIMP. We took the average total path cost of the resulting paths of the algorithms after each environmental change for each agent in the brc202d map with ten agents. As other results were similar, we just presented one configuration. Table 1 shows the average path costs of each algorithm. Although CBS-D*-lite has better total path costs, there is no significant difference between CBS-D*-lite and LIMP. This shows that we did not deviate from the optimal solution.

5 CONCLUSION AND FUTURE WORK

The existing MAPF algorithms mainly focus on static environments; however, the real-life problems are far away from being static. Recently, an incremental version of MAPF was proposed by Semiz, and Polat (Semiz and Polat, 2021). They also proposed two algorithms to solve the I-MAPF problem with CBS called CBS-replanner and CBS-D*-lite. The CBS-replanner uses A* as a low-level search, and CBS-D*-lite uses D*-lite in the low-level search.

In this paper, we proposed two algorithms called LIMP and DLPA*. LIMP is a combination of the CBS-D*-lite with our new search algorithm DLPA*, which is the relaxed version of LPA*. DLPA* is complete and optimal, while LIMP is just complete as it is built on the top of a sub-optimal I-MAPF solver called CBS-D*-lite. We have seen that, despite falling behind in initial calculation, LIMP catches up and outperforms CBS-D*-lite after several changes, and it shows that LIMP reacts to environmental changes more efficiently. Although D*-lite finds slightly lower path-cost solutions, the differences are at an acceptable level. Hence, LIMP is a good alternative for solving the I-MAPF problem, especially when environmental changes are frequent.

ACKNOWLEDGEMENTS

This work is partially supported by the Scientific and Technological Research Council of Turkey under Grant No 120E504.

REFERENCES

Atiq, B., Patoglu, V., and Erdem, E. (2020). Dynamic multi-agent path finding based on conflict resolution using answer set programming. In *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18-24th September 2020*, volume 325, pages 223–229.

Boyarski, E., Felner, A., Harabor, D., Stuckey, P. J., Cohen, L., Li, J., and Koenig, S. (2020). Iterative-deepening conflict-based search. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4084–4090. Main track.

Koenig, S. and Likhachev, M. (2002). Dlite. In *Eighteenth National Conference on Artificial Intelligence*, page 476–483, USA. American Association for Artificial Intelligence.

Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong planning a*. *Artificial Intelligence*, 155(1):93–146.

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, 42(2):189–211.

Lumelsky, V. and Stepanov, A. (1986). Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE Transactions on Automatic Control*, 31(11):1058–1063.

Murano, A., Perelli, G., and Rubin, S. (2015). Multi-agent path planning in known dynamic environments. In *PRIMA 2015: Principles and Practice of Multi-Agent Systems*, pages 218–231.

Pirzadeh, A. and Snyder, W. (1990). A unified solution to coverage and search in explored and unexplored terrains using indirect control. In *Proceedings., IEEE International Conference on Robotics and Automation*, pages 2113–2119 vol.3.

Ramalingam, G. and Reps, T. (1996). An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305.

Semiz, F. and Polat, F. (2021). Incremental multi-agent path finding. *Future Generation Computer Systems*, 116:220–233.

Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66.

Standley, T. (2010). Finding optimal solutions to cooperative pathfinding problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 24(1).

Stentz, A. (1993). Optimal and efficient path planning for unknown and dynamic environments. *INTERNATIONAL JOURNAL OF ROBOTICS AND AUTOMATION*, 10:89–100.

Stentz, A. (1995). The focussed d* algorithm for real-time replanning. In *In Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1652–1659.

Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148.

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*, pages 3260–3267.

Wan, Q., Gu, C., Sun, S., Chen, M., Huang, H., and Jia, X. (2018). Lifelong multi-agent path finding in a dynamic environment. In *2018 15th International Conference on Control, Automation, Robotics and Vision (ICARCV)*, pages 875–882.

Zelinsky, A. (1992). A mobile robot exploration algorithm. *IEEE Transactions on Robotics and Automation*, 8(6):707–717.