2018 15th International Conference on
Control, Automation, Robotics and Vision (ICARCV)
Singapore, November 18-21, 2018

Lifelong Multi-Agent Path Finding in A Dynamic Environment

Qian Wan, Chonglin Gu, Sankui Sun, Mengxia Chen, Hejiao Huang*, and Xiaohua Jia (*IEEE Fellow*)

***Abstract*—In tradition, the problem of Multi-Agent Path Finding is to find paths for the agents without conflicts, and each agent execute one-shot task, a travel from a start position to its destination. However, making just one planning for the agents may not satisfy the requirement in dynamic environments such as logistics sorting center, where the paths of the agents may constantly need to be adjusted according to the incoming tasks. The challenging issue is to dynamically adjust the already planned paths while make planning for the agents ready to execute new incoming tasks. In this paper, we formulate it into Dynamic Multi-Agent Path Finding (DMAPF) problem, the goal of which is to minimize the cumulative cost of paths. To solve this problem, we propose an algorithm called Lifelong Planning Conflict- Based Search (LPCBS), which can efficiently and optimally make planning for the new incoming tasks while adjusting the already planned paths. Experiment results show that the LPCBS performs much better than the existing works in each planning.**

***Index Terms*—Dynamic, Multi-Agent Path Finding, Lifelong Planing**

I. Introduction

Multi-Agent Path Finding (MAPF) can be used to model many real-world problems like video games, traffic control, robotics and etc [1]. Given a grid graph $G(V, E)$, the MAPF needs to find paths for a set of agents $A = \{a_1, a_2, ..., a_m\}$ without conflicts. Each agent $a_i \in A$ executes a task, which is a travel from its start position $s_i \in V$ to its destination $d_i \in V$. The searching space of the problem is increased by b^m, where b is the number of actions that agents could take. For example, in a 4-neighbor grid, b is 5, including 4 *move* actions and a *wait* action. MAPF problems have been shown to be NP-hard [2] [3]. The common optimization objective of the MAPF problem is to minimize the makespan of agents or the cumulative cost (also called sum of individual cost in related researches) of paths.

In recent years, the problem of MAPF has been extensively studied, which focuses on executing one-shot tasks. That is, each agent is assigned a task to move from a start position to its destination, the path of which is planned by a solver. However, in some dynamic environments, making just one planning for the agents may not satisfy the application requirement. Figure 1 is an example of the logistics sorting center (LSC), where agents are repeatedly executing tasks between some specific pickup positions and delivery positions. There are agents busy executing tasks, while there are also free agents in the caching areas waiting for the dynamically arriving tasks. In order to keep the optimality of the paths for all active agents, we need to constantly adjust the already planned paths of activate agents while planning the paths for agents ready to execute new incoming tasks.

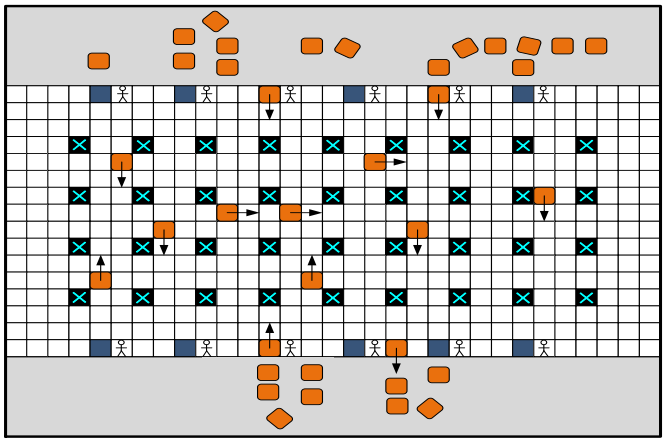

Fig. 1: An example of logistics sorting center

In this paper, we formulate this problem into Dynamic Multi-Agent Path Finding (DMAPF) problem and propose *Lifelong Planning Conflict-Based Search* (LPCBS) algorithm to solve it. LPCBS is an incremental searching version of the previous work Conflict-Based Search (CBS) [4]. Here *lifelong* is inspired from [5], which means saving current planning information for the next planning. The key structure Constraint Tree (CT) used in CBS has been modified in LPCBS, which can greatly reduce the memory used in the searching process. Experiment results show that, rather than treating each planning independently, LPCBS performs better in paths adjusting and paths finding for the new active agents through reusing the previous planning information. We also prove the optimality of LPCBS algorithm in solving the problem of DMAPF.

The rest of this paper are organized as follows. In Section II, we discuss the recent related work of MAPF. Then, we formulate the DMAPF problem in Section III. The CBS algorithm is introduced in Section IV. In Section V, we discuss the details of LPCBS algorithm and give the theoretical analysis of it. We show our experimental results in section VI. Finally Section VII concludes this work.

II. Related Works

The general solvers for MAPF problem can be categorized into *centralized* and *decentralized*. The centralized solvers

All authors are with the Department of Computer Science and Technology, Harbin Institute of Technology (Shenzhen) and Shenzhen Key Laboratory of Internet Information Collaboration, China. Hejiao Huang is the corresponding author, Email: huanghejiao@hit.edu.cn

978-1-5386-9582-1/18/$31.00 ©2018 IEEE

assume that the paths of agents can be calculated by a center computer, so that the agents only need to follow the order of the computer to move along with, while the decentralized solvers assume each agent has its own computing power and calculates the path by its own, and there is a cooperative setting to assist agents to make decisions [1]. Centralized solvers often focus on the solution's quality while decentralized solvers pay more attention to the efficiency.

One type of centralized solvers is to first reduce the MAFP problem into a classical problem, and then solve it using standard solvers. Such solvers are often called reduction-based solvers. Yu et al. in [6] have studied the MAPF problem with different optimization objectives. They reduce the problem into a multi-commdity flow ILP problem, and then propose an efficient heuristic method to solve it. Ma et al. in [7] proposed a variant version of MAPF problem but with package-exchange. They also reduce the problem into a multi-commodity flow problem. Some recent reseaches reduce the MAPF problem into *Satisfiability* problem, and sovle it under different optimization objectives [8] [9] [10]. Another type of centralized solvers is search-based solver. Standley et al. in [11] propose an *operator-decomposition* (OD) method, which introduce intermediate states by assigning actions to agents one by one in a specific order, substantially reducing the braching factor of A* algorithm. *Partial-Expantion A** (PEA*) [12] is an algorithm for solving a single agent path finding problem, based on which *Enhanced Partial-Expantion A** (EPEA*) [13] is proposed aiming at opimizing A* for MAPF problems. *M** [14] is also a searching algorithm that dynamcially changes the searching dimensionality according to the conflicts encountered during the searching. Different from A* based solvers, *Increasing Cost Tree Search* (ICTS) [15] and *Conflict Based Search* (CBS) [4] are both 2-level searching algorithms proposed by Sharon et al. ICTS gives a set optimal values for each agent at the high level and runs a solution test at the low level that tries to find a solution that equals to the set of values at the high level. CBS treats each agent as a single agent at the low-level search, and the conflicts between agents are resolved at the high-level search. Note that, both of the algorithms could find the optimal solution. To achieve better performance, recent researches try to change the CBS to bounded sub-optimal versions [16] [17].

Decentralized solvers are always sub-optimal compared with the centralized solvers. A prominent decentralized solver of MAPF is *Hierarchical Cooperative A** (HCA*) [18], which is shown to have a good performance but not complete. It plans the paths for the agents one by one. If an agent's path is determined successfully, the positions in the path will be distributed as an obstacle in a time-spacial space. Ma et al. propose two methods called *Token Passing* and *Token Passing with Task Swaps*, which perform well in the special enviroment [19]. Some decentrailized solvers are rule-based, Luna et al. propose two rules called *Push* and *Swap*, which could solve any MAPF instance with at least 2 empty nodes in the graph [20]. Wei et al. propose a communication diagram. It reduces the coordinating area to a 2-step sized area instead of the whole team of agents [21].

However, the previous work did not consider the changing number of active agents in a dynamic environment. In this paper, we try to introduce incremental searching into CBS and solve this problem by LPCBS algorithm.

III. Problem Formulation

In this Section, we first give a formal description of MAPF problem, and then the formulation of DMAPF. Here is the problem formulation for MAPF:

Given a grid space $G = (V, E)$, there is an $m - sized$ team of agents $A = \{a_1, a_2, ..., a_m\}$ distributed over the space. For each agent a_i, it will execute a task $\tau_i = (s_i, d_i)$, which is to finish a travel from position s_i to its destination d_i. The objective is to find a set of paths for the agents while satisfying the constraints given in advance. A path is a mapping from a time sequence $T = \{0, 1, 2, ...\}$ to the points in V. Let l_i denote the path of agent a_i, then $l_i(t) \in V$ is the location of agent a_i at time $t \in T$. There are two constraints that must be satisfied by the agents:

1) $\forall a_i, a_j \in A \wedge i \neq j, l_i(t) \neq l_j(t)$.
2) $\forall a_i, a_j \in A \wedge i \neq j, l_i(t) \neq l_j(t+1) \vee l_i(t+1) \neq l_j(t)$.

The first constraint means that each point in V can not be occupied by two agents at the same time. The second means that, for any two agents moving towards each other, they can not arrive at each other's place in the same timestep. The optimization objective of our problem is to minimize the sum of individual cost (SIC).

The cost of agent a_i's path is the total number of time steps moving from its start position to its destination. Let t_s^i and t_d^i denote the starting time step and arriving time step to the destination, respectively. Thus, its path cost is $c_i = t_d^i - t_s^i$. The SIC is $\sum_{a_i \in A} c_i$.

The MAPF focuses only on one-shot problem. The path planning for each agent is made only once. When arriving at their destinations, the agents will no longer move. In dynamic enviroment, however, there may be some new agents put into the environment to execute tasks at any time, and the agents accomplishing the tasks will go back immediately and wait for the next round of planning, which is called DMAPF.

Let Sol_A denote the solution we got for the agents team A, Sol_A^i is the path of agent a_i. The start time step of Sol_A is denoted as t_{SA}. Suppose there is a new team of agents A' put into the environment to execute new incoming tasks at current time step t_{curr}. These agents can be either new added agents or the agents returned back after finishing their tasks. The challenging issue lies in how to keep the optimal property of the new solution $Sol_{A \bigcup A'}$ for agents set $A \bigcup A'$.

Figure 2 is an example showing the dynamic environment. After 5 time steps, agent 1 and agent 2 are on the path to their destinations, while agent 3 and agent 4 are put into the environment. Agent 1 and agent 2 are in team A, agent 3

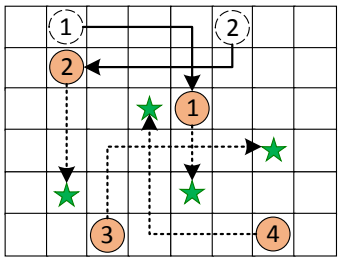

Fig. 2: An example of dynamic environment (Here the dashed arrow means the path to walk, while the solid arrow means the path already walked. The dashed circles means the start positions, while the solid circles are the current positions of the agents).

and agent 4 are in team A'. The paths of agents in A may be adjusted when making path planning for the agents in A'.

IV. The Conflict Based Search (CBS) Algorithm

In this Section, we will give an introduction of CBS, since our algorithm LPCBS is design based on its framework. In general, the searching process of CBS is composed of high-level search and low-level search:

1) At the high level search, the algorithm will process the node in the constraint tree (CT) and resolve its conflict. Note that, the CT is a binary tree in our design. There are 3 types of information in a CT node N:

- A constraints set for agents (*N.Constrains*).
- A solution for MAPF instance (*N.Solution*).
- The SIC for its solution (*N.SIC*).

2) At the low level search, the algorithm will find a path for each agent according to its constraints, ignoring that of others. Thus, there may exist conflicts between agents' paths. We use $C = (a_i, a_j, s, t)$ to represent that agents a_i and a_j encounter a conflict at element s in time step t. The element can be a point or an edge with conflicts corresponding to the two constraints mentioned in Section III.

Algorithm 1 gives the details of CBS [4], in which a root node R is first created with no constraints. Then, find a path for each agent by a time-space version of A* with no consideration of other agents. Thus, all paths of the agents constitute a solution, and we can calculate the SIC for this solution. Next, put the root node to a priority queue *OPEN* according to the SIC value of the nodes. The top of the queue with highest priority is the node with the minimal SIC. The algorithm will proceed by continuously fetching the top node in *OPEN* (lines 5-17). For each fetched node, it involves two procedures: CT node processing and conflict resolving:

- **Processing a CT node**: Given a CT node *N*, there is an optimal solution that can satisfy the constraints of all agents. If there is no conflict happening, return the solution for this MAPF instance. Otherwise, get the first conflict found in this solution and then resolve it (lines 6-8).
- **Resolving a conflict:** Suppose there is a conflict $C = (a_i, a_j, s, t)$ found in this CT node. Create two child nodes for this CT node. The child nodes will initially inherent the solution and constraints from their parent node. For each child node, for example a_i, adding a new constraint (a_i, s, t) to its constraint set. Here constraint (a_i, s, t) means a_i can not occupy the element s in t. The constraint set of a_i is changed, so the path of a_i should be adjusted accordingly. The adjusting process invokes the low-level search such as A* to find a new path for a_i consistent with its new constraint. Finally, update the SIC value for each child node, and put them into the priority queue *OPEN* (lines 9-16).

Algorithm 1: Conflict Based Searching (CBS) Algorithm

Input: MAPF instance
Output: Optimal solution of the instance
1 $R.constraints \leftarrow \varnothing$
2 $R.solution \leftarrow$ Find a path for each agent by invoking low-level search such as A* and so on
3 $R.SIC \leftarrow SIC(R.solution)$
4 Insert R to $OPEN$
5 **while** $OPEN \neq \varnothing$ **do**
6 $\quad N \leftarrow OPEN.top()$
7 $\quad$ **if** $N.solution$ *has no conflict* **then**
8 $\quad\quad$ **return** $N.solution$
9 $\quad C \leftarrow$ Get the first conflict (a_i, a_j, s, t) from $N.solution$
10 $\quad$ **for** *each agent* a_x ***in*** C **do**
11 $\quad\quad P \leftarrow$CreateNode()
12 $\quad\quad P.constraints \leftarrow N.constraints + (a_x, s, t)$
13 $\quad\quad P.solution \leftarrow N.solution$
14 $\quad\quad P.solution[x] \leftarrow$low-level search for a_x
15 $\quad\quad P.SIC \leftarrow SIC(P.solution)$
16 $\quad\quad$ Put P in $OPEN$
17 $\quad OPEN.pop()$

Optimality Discussion: The CBS algorithm returns the optimal solution. It can be summarized to three key points:

- For any valid solution, there is at least one node in OPEN permitting it.
- The SIC value in a node defines a lower bound of cost of all the solution permitted by this node.
- The high-level search always expand the node with the lowest SIC.

V. Lifelong Planning Conflict-Based Search

A. Extended Definitions of CBS

LPCBS inherents some definitions from CBS. We use the term *valid solution* to represent the solution of a MAPF instance with no conflict. Let *CV(N)* denote the solution set in which all solutions are consistent and valid with the constraints of node *N*. $\forall p \in N$, we say *N* permits *p*. For ease of understanding, we introduce some new definitions.

Definition 1. *We say a pair of constraints are mutual related if they share the same element and time step though with*

different agents. For example, the constraints (a_i, s, t) and (a_j, s, t) are mutual related.

Proposition 1. *For any node* N *in* OPEN, *if* $c = (a_i, s, t)$ *is the new added constraint of it, there must exsit another node in the same (binary-tree) level containing a mutual related constraint* $c' = (a_j, s, t)$.

Definition 2. *An* OPEN *is* Constraints-Completed *if for any valid solution* p, *there exist a node* N *in* OPEN *permits it.*

Proposition 2. *Expanding any node in* $OPEN$ *does not break the constraints-complete.*

Proof. See [4].

Definition 3. *We say a node* N *in* OPEN *is time-consistent, if for any path* $N.solution^i$, $N.solution^i(t_{curr}) = l_i(t_{curr})$, *and* $N.solution^i$ *is the optimal path consistent with* $N.constraints$ *for agent* a_i. *Otherwise, we call the node time-inconsistent.*

Proposition 3. *Expanding a time-consistent node* N *in* OPEN *would get two time-consistent nodes. That is, two child nodes of* N *are time-consistent.*

Proof. The low-level search gurantees the optimality of the new path that is consistent with the constraints, and expanding nodes does not change the current position in current time step. Thus, for each child node N_{ch}, we have $N_{ch}.solution^i(t_{curr}) = l_i(t_{curr})$.

B. Lifelong Planning CBS

The basic idea of LPCBS is planning the paths of the new added agents in current time step while adjusting the paths of the running agents based on the previously planned information. In CBS algorithm, the main process is to continuously resolve the conflicts between agents' paths until finding a node with no conflicts in its solution, we call it *CBS process*. When we get a solution Sol_A, it means a set of constraints is found to resolve all the conflicts on the paths of the agents, and Sol_A has the minimal cost in *OPEN*. It is also feasible when adding the paths of new agents A' to *OPEN* through continuing CBS process, which can finally resolve all conflicts of the agents in $A \bigcup A'$. The rationality of this method is that, instead of planning from the scratch, the nodes in *OPEN* have already saved the constraints to deal with the potential conflicts found among the agents in A. If not saved in advance, these conflicts may be encountered again during the replanning process, and expanding *CT* node may be very time-consuming. To achieve this goal, we first need to solve the following two problems:

- **Outdated Node:** If the solution of a node N is time-inconsistent (*See Definition 3*), it is called an outdated node. The correct solution will not be obtained if not eliminating the time-inconsistancy of the outdated nodes. The details for the elimination will be discussed later.
- **Outdated Constraint:** When there is a set of agents A' coming at time step t_{curr}, for any constraint $c = (a_i, s, t)$, c will become an outdated constraint if $t < t_{curr}$. That is to say, they will be useless in the following process of path finding. The outdated constraints will inflate the searching space, lowering the solving efficiency, which will be discussed later.

Sharon et al. in [4] has proved that CBS process can return optimal solution if it starts at *OPEN* within only one root node. But LPCBS starts searching from an intermediate *OPEN*. It does not require the initial condition like CBS algorithm, but requiring the more general contiditions, as defined previously. In each time step, there is a state corresponding to current positions of the agents. The *current problem* is to find the paths for all agents from current state to their destinations, so it will dynamically change with agent set in each time step.

Lemma 1. Given a constraints-completed *OPEN*, if any node in it is time-consistent, then using CBS process to search it will return optimal solution of current problem.

Proof. Assume there is an *OPEN* satisfying the constraints-completed property and all nodes in it satisfy time-consistency, we use the CBS process to search this *OPEN*. According to *Proposition 2* and *Proposition 3*, expanding nodes doesn't change the time-consistency of *OPEN* and constraints-completed property of the nodes in *OPEN*. We simply consider the situation that the algorithm returns a valid solution. Suppose the returned solution is from node N_S, $\forall N \in OPEN$, $N_S.SIC \leq N.SIC$ must set up, because the algorithm runs a best-first search. $\forall N \in OPEN \wedge p \in CV(N), N.SIC \leq SIC(p)$ because $N.Solution$ is obtained by low-level search, and optimal saerch for individual agent. Since *OPEN* is constraints-completed, for any valid solution p, there is a node N permitting it, and $N_S.SIC \leq N.SIC \leq SIC(p)$, so p is the optimal solution. This node is time-consistent, so $\forall p \in N_S.solution, \forall a_i \in A, p^i(t_{curr}) = l_i(t_{curr})$. That means, the agents could follow these paths for next movement.

1) Eliminating Time-inconsistency of Outdated Nodes

Suppose there is a time-inconsistent node $\overline{N}$, $\forall l \in \overline{N}.Solution$, the most direct way to eliminate its inconsistency is to reset the start positions of these paths to $l(t_{curr})$. Figure 3 is an example of time-inconsistency.

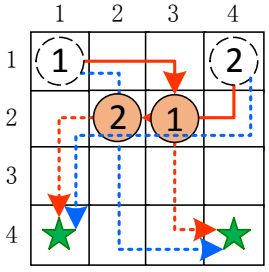

Fig. 3: An example of time-inconsistency (Here the dashed arrow means the path to walk, while the solid arrow means the path already walked. The dashed circles means the start positions, while the solid circles are the current positions of the agents. The paths in red and blue represent the solutions in different nodes).

In this figure, there are two agents a_1 and a_2 moving towards their destinations. There may exist another node in

the *OPEN*. We use red color to denote a solution in node N for a_1 and a_2, whose paths are:

$(1,1) \rightarrow (1,2) \rightarrow (1,3) \rightarrow (2,3) \rightarrow (3,3) \rightarrow (4,3) \rightarrow (4,4)$

$(1,4) \rightarrow (2,4) \rightarrow (2,3) \rightarrow (2,2) \rightarrow (2,1) \rightarrow (3,1) \rightarrow (4,1)$

There may exist another node (denoted as $\overline{N}$) in *OPEN*, which may also contain a valid solution. We use blue color denote its paths. In our example, the path in $\overline{N}$ for a_2 is the same as that in N. The path for a_1 in $\overline{N}$ is:

$(1,1) \rightarrow (1,2) \rightarrow (2,2) \rightarrow (3,2) \rightarrow (4,2) \rightarrow (4,3) \rightarrow (4,4)$

Suppose node N has the smallest SIC in *OPEN*, so all agents will first move based on the solution in N. After 3 time steps, the locations for a_1 and a_2 reach $l_1(3) = (2,3)$ and $l_2(3) = (2,2)$, respectively. In node $\overline{N}$, however, $\overline{N}.Sol^1(3) = (3,2) \neq l_1(3)$ and $\overline{N}.Sol^2(3) = l_2(3)$, so the node $\overline{N}$ is not time-consistent. To handle this time-inconsistent node $\overline{N}$, we adjust the paths in it. For a_2, we only need to cut the traveld path from its start position. It is time-consistent because the remaining path from $\overline{N}.solution^2(3)$ to d_2 is also optimal and consistent with the constraints in $\overline{N}$. For a_1, we need to replan a path from $l_1(3)$ to d_1 by low-level search. *Algorithm* 2 shows how to eliminate time-inconsistency by path adjusting.

Algorithm 2: Time-Inconsistency-Elimination (T-I-Elim)

Input: A LPCBS node N

1 **foreach** *agent* a_x*'s path* p *in* $N.solution$ **do**
2 Cut down the path from start position to $p(t_{curr})$
3 **if** $p(t_{curr}) \neq l_x(t_{curr})$ **then**
4 $p \leftarrow$ Find a path for a_x through low-level search from $l_x(t_{curr})$ to d_x

2) Eliminating Outdated Constraints

Before introducing our algorithm, we first describe the data structure of our *CT*. Different from the nodes in CBS, our *CT* is also a binary tree but storing only the new added constraint in each node. Each leaf node has a corresponding LPCBS node, which stores only two types of information: solution and SIC value. The constraints information can be collected for each node by traversing back to the root. Thus, the memory for storing *CT* can be substantially reduced.

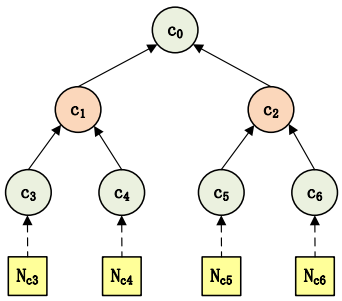

Fig. 4: An example of constraint tree (CT) (Here the circles are the nodes in CT, while the squares are LPCBS nodes).

Figure 4 is an example of our constraint tree *CT*. When creating *CT*, the root node has no constraint, representing as $c_0 = \varnothing$. Here c_0 is also the name of the root node. When there is a conflict found in the LPCBS node corresponding to c_0, it will generate two child nodes with a new constraint stored inside to resolve the conflict. Meanwhile, two LPCBS nodes will be created for each leaf node. The process will continue until there is no conflicts in the leaves. By traversing back to the root node, the constraints in LPCBS nodes N_{c3}, N_{c4} are $\{c_3, c_1, c_0\}$ and $\{c_4, c_1, c_0\}$, respectively. This data structure of *CT* can save the memory space when storing the solutions that have been tested.

We assume c_1 and c_2 in Figure 4 are nodes with outdated constraints. According to *Proposition 1*, the outdated constraints must appear in pair. Let N_{ci} denote LPCBS node i, which will be deleted when its corresponding *CT* node has child nodes. The root node c_0 is created with no constraint, so $CV(N_{c0})$ contains all valid solutions of the MAPF instance, which is formulated by the agents in environment from current positions to their destinations. When there is a conflict found in the $N_{c0}.solution$, $CV(N_{c0})$ will be divided into two subsets $CV(N_{c1})$ and $CV(N_{c2})$, and $CV(N_{c1}) \bigcup CV(N_{c2}) = CV(N_{c0})$. This process will go on until no conflicts can be found any more. Since c_1 and c_2 are outdated, the constraint set N_{c1} and N_{c2} have the same effect on the permitting set, so $CV(N_{c0}) = CV(N_{c1}) = CV(N_{c2})$. It also means $CV(N_{c3}) \bigcup CV(N_{c4}) = CV(N_{c5}) \bigcup CV(N_{c6}) = CV(N_{c0})$. Therefore, we only need to retain one of branches either in child nodes c_1 or in c_2 for constraints-completed. If not removing the outdated nodes, we will make searching in two identical *OPEN*, and some conflicts may even be checked many times. The more outdated stored in the *CT*, the more duplicate conflicts will be checked, which can severely degrade the searching performance.

Algorithm 3 is a recursive algorithm to remove the outdated constraints from the *CT* while still keeping the *OPEN* constraints-completed. Since the root node has no constraint, it will never outdate. In each recursion, we assume that the outdated constraints happen only in the child of current root node, so there are three cases that should be considered:

Case 1: Both child nodes of the root are leaf nodes, corresponding to lines 8-11 to deal with it.

Case 2: Only one child node is a leaf node, corresponding to lines 12-15 to deal with it.

Case 3: Neither of the child nodes are leaf nodes, corresponding to lines 17-24 to deal with it.

Figure 5 is an example showing the three cases and the processed result using *Algorithm 3*. The big node (ellipse shape) in the *CT* represents a branch or a sub-tree, which may contain one or more nodes. In Case 2, we try to save branch from c_2 with more deep levels, because there has been many conflicts found in this branch, and we do not need to make searching from scratch. So it is for Case 3.

Lemma 2. *Algorithm 3 does not break the constraints-completed of the given* OPEN.

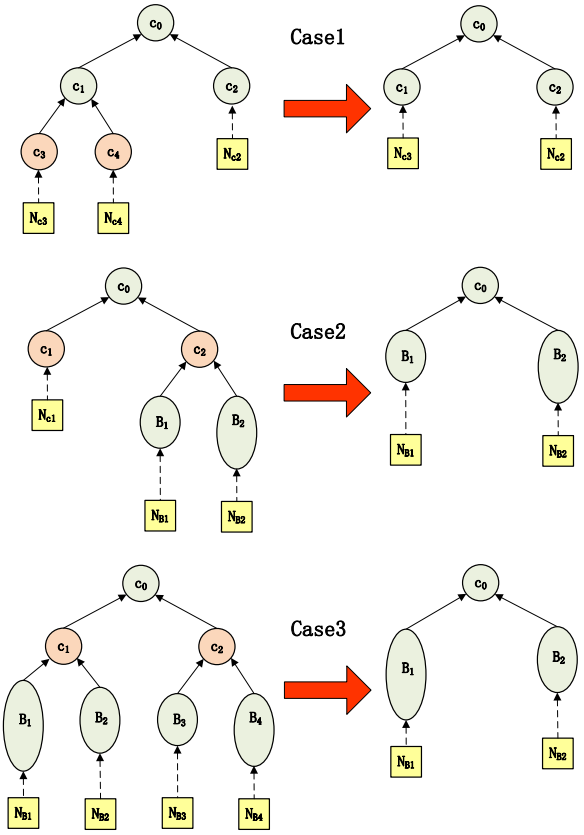

Fig. 5: An example showing the three cases of outdated constraint in *CT* and the processed result using *Algorithm 3* (Here the inconsitent nodes are in light orange).

Proof. In *Algorithm 3*, lines 4-5 guarantee that the branches of the child nodes of root node have no outdated constraints and *OPEN* is constriants-completed, so we only consider its two child nodes. If the child nodes are not outdated, then algorithm returns, and the *CT* is constraints-completed. According to *Proposition 1*, If one child node (denote as child1) is outdated, there is another mutual related child node (denote as child2) outdated. Thus, $CV(N_{child1}) = CV(N_{child2}) = CV(N_{root})$, and the $CV(N_{child1})$ is divided into two sub-branches according to *Proposition 2*. Saving either of the branches of child1 or child2 does not affect the permitting set, so the *OPEN* remains constraints-completed.

Algorithm 4 is our lifelong planning CBS (LPCBS). There are three stages in this algorithm:

1) If there is a new team of agents A' coming to execute tasks, it will first remove the outdated constraints in *CT* according to *Algorithm 3*.

2) For each node in *OPEN*, initialize the paths for A' and add them into the node's solution, and then eliminate the time-incosistency for this node.

3) Invoke the CBS process to search in the new *OPEN*. Thus, we can get the final solution.

Thorem 1. *Lifelong Planning CBS returns optimal solution for the whole agents based on their current positions.*

Algorithm 3: Clean-Constraint-Tree

Input: A Constraint-Tree
1 **if** *root is leaf node* **then**
2 return
3 $CleanConstraintTree(root.child1)$
4 $CleanConstraintTree(root.child2)$
5 **if** *there is no outedated constraint in child nodes* **then**
6 return
7 **if** $root.child1$ *is leaf node* **then**
8 **if** $root.child2$ *is leaf node* **then**
9 $root.LPCBSnode \leftarrow root.child1.LPCBSnode$
10 $ClearConstraintTree(root.child2)$
11 Delete $root.child1$
12 $root.child1 \leftarrow root.child2.child1$
13 $root.child2 \leftarrow root.child2.child2$
14 Delete $root.child2$
15 $ClearConstraintTree(root.child1)$
16 **else**
17 $depth1 \leftarrow Depth(root.child1)$
18 $depth2 \leftarrow Depth(root.child2)$
19 **if** $depth1 < depth2$ **then**
20 $Swap(root.child1, root.child2)$
21 $root.child1 \leftarrow root.child1.child1$
22 $root.child2 \leftarrow root.child1.child2$
23 $ClearConstraintTree(root.child2)$
24 Delete $root.child1$

Algorithm 4: Lifelong Planning CBS

Input: A DMAPF instance A'.
1 **if** $OPEN$ *is empty* **then**
2 Initializing *OPEN* with empty LPCBS node *rn*
3 Create a root node *rc* of *CT*
4 Corelate *rn* with *rc*
5 $Solution_{new} \leftarrow$ Initialize a solution for A' without any constraints
6 Clean-Constraint-Tree(rc)
7 **foreach** *LPCBS node* N *in* OPEN **do**
8 $N.solution \leftarrow N.solution \bigcup Solution_{new}$
9 Eliminate inconsistency through *T-C-Elim(N)*
10 $Solution \leftarrow$ Find the solution through CBS process
11 return $Solution$

VI. Experimental Evaluation

To verify the effectiveness of our method, we simulate the execution of tasks in a LSC-like environment. The layout of the environment is a 32*45 grid, and all agents are staying in the caching area at the beginning. We generate the arrival of tasks with different start positions and destinations every 10 time steps through a *task generator*. After dispatching the tasks to the agents, our solver will formulate each MAPF

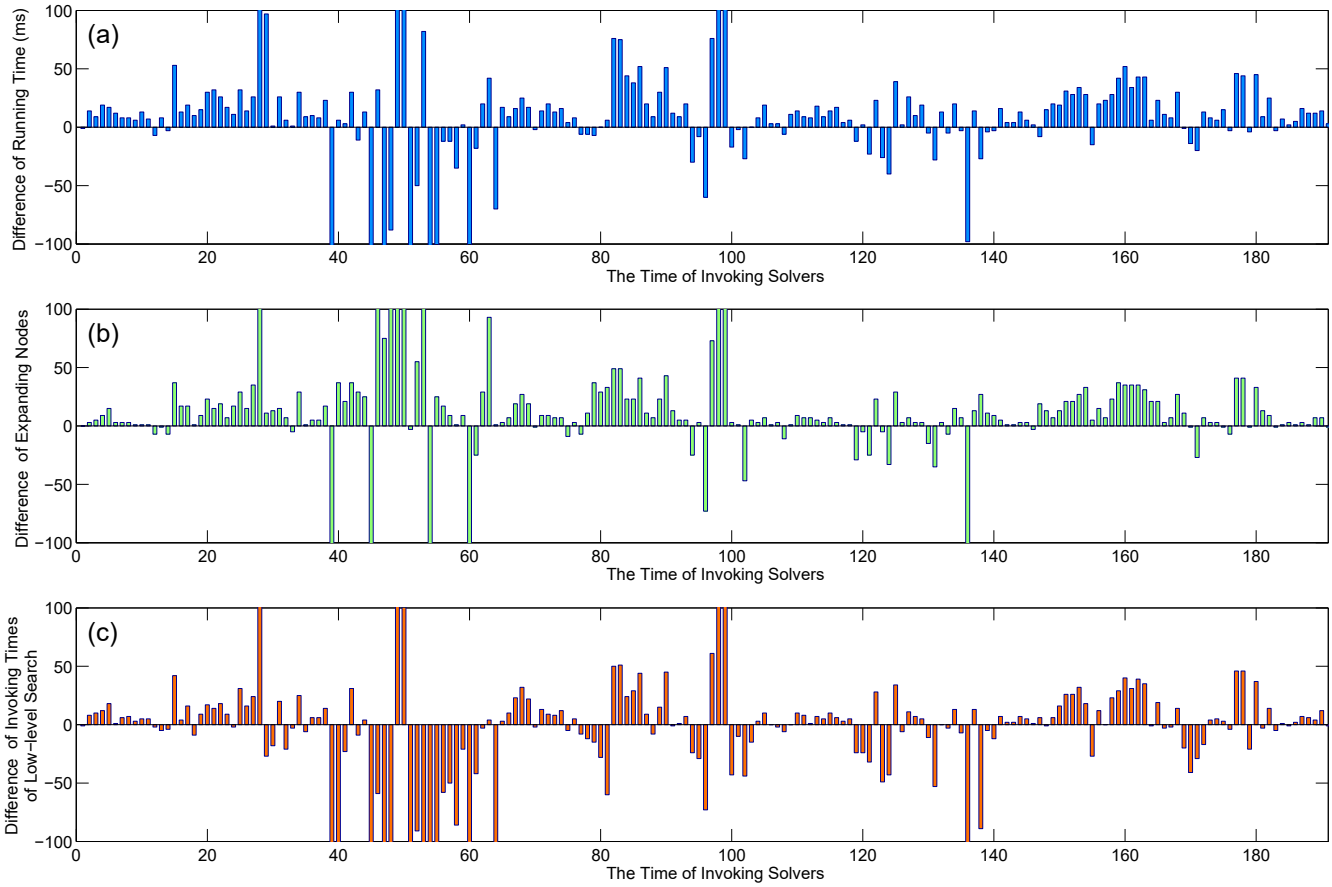

Fig. 6: Comparisons of CBS and LPCBS through the differences in the running time, number of extending nodes, and invoking times of low-level search

instance into a current problem and solve it using LPCBS. All agents will travel according to the paths calculated by the solver. Whenever an agent arrives its destination, it will be assigned a returning-back task to the caching area.

We make comparison with the existing work using 32 agents within a period of 300 time steps. In each current time step when there are new incoming agents, we try different methods to make path planning and compare their results. Figure 6 shows the differences between *CBS* and *LPCBS* in the running time, number of extending nodes, and invoking times of low-level search. As can be seen in Figure 6 (a), it takes less time to get the solution using *LPCBS* than *CBS* in most cases, or we say, *LPCBS* runs faster than *CBS*. It is because the previous planned information has been saved by *LPCBS*, avoiding resolving the same conflicts repeatedly. In fact, more nodes expansion in the high-level means more conflicts have been encountered and resolved. However, it is also a reflection of low efficiency of the searching algorithm in a certain extent. Figure 6 (b) shows the numbers of the expanding nodes in *CBS* are mostly more than that of our method, indicating that *LPCBS* is more efficient in its searching. The merits of *LPCBS* lies in the design of *CT*, which stores only the new added constraint. However, it may invoke low-level search when eliminating inconsistent constraints, which may also be affected by the branch-cut strategy of in *Algorithm 3*. That is why the times of invoking low-level search for *LPCBS* is larger than that of *CBS* in some cases in Figure 6 (c), though not so many. To conclude, *LPCBS* can find the optimal solution with higher efficiency than *CBS* in each planning.

In the existing work, the paths of agents in *HCA** is designed separately in a decentralized way, which is suitable for dynamic environment. In this paper, we use *SIC* to measure the quality of solution. When the *SIC* are the same, we use the running time, as discussed above. Figure 7 shows the SIC difference between *HCA** and *LPCBS*. It can be found that *LPCBS* always get better solution than *HCA**, although *HCA** has a better performance than the centralized optimal solver theoretically. In spite of this, *LPCBS* still can get a good performance with proper agent size.

MAPF problem is NP-hard, so any optimal solver for it can not work well when the agent size is too large. Since there are 16 entrances in the layout of LSC system, we consider the cases when there are 16, 32 and 48 agents and make comparison of the running time under our *LPCBS*. Figure 8 illustrates that using 16 agents runs faster than using 32 agents in most time steps, let alone for 48 agents, because more agents will bring congestions in path finding. Here we did not show the line when using 48 agents due to its too large running time, far exceeding that of 16 and 32.

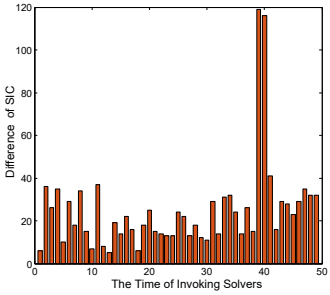

Fig. 7: The SIC Difference between HCA* and LPCBS

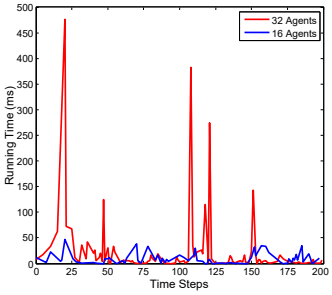

Fig. 8: Running Time for Different Agent Number

VII. Conclusion

In this paper, we studied the problem of DMAPF problem by Lifelong Planning Conflict Based Searching (LPCBS) algorithm, which is an incremental search through storing only the new added constraint in each CT node, which can substantially reduce the memory to be used for this tree. The main idea of *LPCBS* is to reuse the priority queue *OPEN* generated from *CBS* process to adjust the paths planned for existing agents, while make path finding for the new incoming agents. In the searching process, there may exist outdated nodes and time-inconsistent constraints, which are eliminated by our proposed *T-I-Elim* and *Clean-Constraint-Tree* algorithms. Simulation results show that *LPCBS* is more efficient in finding the optimal solution than *CBS*, while with smaller SIC than that of *HCA** in each planning.

Acknowledgment

This work is financially supported by National Science and Technology Major Project under Grant No.2017YFB0803002 and No.2016YFB0800804, National Natural Science Foundation of China under Grant No.61672195 and No.61732022.

References

[1] A. Felner, R. Stern, S. E. Shimony, E. Boyarski, M. Goldenberg, G. Sharon, N. R. Sturtevant, G. Wagner, and P. Surynek, "Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges," in *Proceedings of the Tenth International Symposium on Combinatorial Search, Edited by Alex Fukunaga and Akihiro Kishimoto, 16-17 June 2017, Pittsburgh, Pennsylvania, USA.*, 2017, pp. 29–37.

[2] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, 2013.

[3] P. Surynek, "An optimization variant of multi-robot path planning is intractable," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[4] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent path finding," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[5] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A," *Artif. Intell.*, vol. 155, no. 1-2, pp. 93–146, 2004.

[6] J. Yu and S. M. LaValle, "Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics," *IEEE Trans. Robotics*, vol. 32, no. 5, pp. 1163–1177, 2016.

[7] H. Ma, C. A. Tovey, G. Sharon, T. K. S. Kumar, and S. Koenig, "Multi-agent path finding with payload transfers and the package-exchange robot-routing problem," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 2016, pp. 3166–3173.

[8] P. Surynek, "Towards optimal cooperative path planning in hard setups through satisfiability solving," in *PRICAI 2012: Trends in Artificial Intelligence - 12th Pacific Rim International Conference on Artificial Intelligence, Kuching, Malaysia, September 3-7, 2012. Proceedings*, 2012, pp. 564–576.

[9] P. Surynek, A. Felner, R. Stern, and E. Boyarski, "Efficient SAT approach to multi-agent path finding under the sum of costs objective," in *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August-2 September 2016, The Hague, The Netherlands - Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, 2016, pp. 810–818.

[10] P. Surynek, J. Svancara, A. Felner, and E. Boyarski, "Integration of independence detection into sat-based optimal multi-agent path finding - A novel sat-based optimal MAPF solver," in *Proceedings of the 9th International Conference on Agents and Artificial Intelligence, ICAART 2017, Volume 2, Porto, Portugal, February 24-26, 2017.*, 2017, pp. 85–95.

[11] T. S. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*, 2010.

[12] T. Yoshizumi, T. Miura, and T. Ishida, "A* with partial expansion for large branching factor problems," in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA.*, 2000, pp. 923–929.

[13] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, and R. Holte, "Partial-expansion a* with selective node generation," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[14] G. Wagner and H. Choset, "M*: A complete multirobot path planning algorithm with performance bounds," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2011, San Francisco, CA, USA, September 25-30, 2011*, 2011, pp. 3260–3267.

[15] G. Sharon, R. Stern, M. Goldenberg, and A. Felner, "The increasing cost tree search for optimal multi-agent pathfinding," *Artif. Intell.*, vol. 195, pp. 470–495, 2013.

[16] M. Barer, G. Sharon, R. Stern, and A. Felner, "Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem," in *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, 2014, pp. 961–962.

[17] L. Cohen, T. Uras, and S. Koenig, "Feasibility study: Using highways for bounded-suboptimal multi-agent path finding," in *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, 2015, pp. 2–8.

[18] D. Silver, "Cooperative pathfinding," in *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference, June 1-5, 2005, Marina del Rey, California, USA*, 2005, pp. 117–122.

[19] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems, AAMAS 2017, São Paulo, Brazil, May 8-12, 2017*, 2017, pp. 837–845.

[20] R. Luna and K. E. Bekris, "Push and swap: Fast cooperative path-finding with completeness guarantees," in *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2011, pp. 294–300.

[21] C. Wei, K. V. Hindriks, and C. M. Jonker, "Altruistic coordination for multi-robot cooperative pathfinding," *Appl. Intell.*, vol. 44, no. 2, pp. 269–281, 2016.