Divide-and-Conquer CBS Design for Multi-Agent Path Finding in Large-scale Scenario

Mengxia Chen, Yiming Liu and Hejiao Huang*
Harbin Institute of Technology, Shenzhen 518055, China

*E-mail: huanghejiao@hit.edu.cn

Abstract. Multi-Agent Path Finding (MAPF) problem is intensively studied in theoretical computer science, robotics and so on. The key for MAPF problem is to plan a conflict-free path for each agent with different start and goal positions, and to minimize the cost of the paths. Conflict-Based Search (CBS) is one of the optimal algorithms, which can ensure that the optimal solution is obtained, in the case of small-scale maps and low number of agents. However, in many real-world multi-agent systems, the scale of the map is very large and the number of agents is more. In most cases, CBS is not applicable any more. Therefore, we develop Divide and Conquer CBS, called DC-CBS algorithm which divides large-scale map and the original problem into several smaller ones. For each subproblem, we use the CBS to get the optimal solution. The experimental results show that for the large-scale scenario, the DC-CBS algorithm can get the result more quickly and plan the paths for agents successfully. As for small-scale situation, the performance of DC-CBS is almost as good as that for CBS.

1. Introduction
The Multi-Agent Path Finding (MAPF) problem includes a undirected graph $G = (V,E)$ and a team of k agents $a_1 ... a_k$ where each agent $a_i$ has a start position $s_i \in V$ and goal position $g_i \in V$. Solving MAPF optimally is NP-hard [1], as the state space grows exponentially with the number of agents. The common objective of MAPF is to find the valid path for every agent without conflicts, and minimize the total cost of the paths. MAPF problem has practical applications including robotics, traffic control, warehouse applications and assembly planning [2]. Recently, many researches have paid attention to the MAPF problem of small maps. However, the results of researches cannot perform well in large-scale maps with a great many agents. For example, the Kiva is a large-scale automatic warehouse which has hundreds of agents [3], it is a challenge to plan paths for massive agents rapidly.

In this paper, we propose a Divide-and-Conquer CBS (DC-CBS) algorithm to solve the large-scale map situation. DC-CBS algorithm is an improvement based on CBS algorithm [4] which is one of the optimal algorithms to solve the MAPF problem. CBS can get the best solution, but it runs slowly. The key of DC-CBS algorithm is to combine divide-and-conquer idea with CBS algorithm, divide large-scale map and the original problem into several smaller ones. The experimental results show that the DC-CBS algorithm can get the result more quickly than CBS, and the quality of solution is applicable although it is somewhat worse than the optimal solution.

2. Related work
Algorithms for solving MAPF can be divided into three classes: optimal solvers, suboptimal solvers, bounded suboptimal solvers. When the number of agents is relatively small and the aim is to find the
minimum solution, optimal solvers are adopted. While suboptimal solvers are usually used when the number of agents is large.

A number of optimal solvers are based on the A* algorithm. A relevant variant is Enhanced Partial Expansion A* (EPEA*) [5], which sorts all successors of a given state to their f value and only the successors that \(f \leq C^*\) are generated. Standley presented the independence detection (ID) and operation decomposition (OD) technique [6]. OD introduces the intermediate states by considering a single agent at a time which reduces the branching factors of A* algorithm successfully. Furthermore, M* can dynamically change the branching factor according to the encountered conflicts. Recursive M* (RM*) is an enhanced version which divides the conflicting agents into several small groups with independent conflicts. Recently, the increasing cost tree search (ICTS) [7] and the conflict based search (CBS) algorithms are one type of search-based optimal solver. CBS is a two-level search algorithm that the low-level plans the path and the high-level resolves the conflicts between the agents.

For suboptimal solvers, most of them are unbounded and not provide any guarantee on the quality of the planned path. A prominent suboptimal MAPF algorithm is the Cooperative A*(CA*) [8] and its variants HCA* and WHCA*. CA* and its variants plan paths for every agent separately. If the path is returned successfully, it will be written a global reservation table and not have conflicts with the following agents. Importantly, these algorithms have a good performance in practice but not complete. Another type of suboptimal solvers is rule based, that formulate specific rules for different scenarios.

The suboptimal algorithm cannot guarantee the quality of solution, but the bounded suboptimal search algorithm can accept a parameter \(w\) and ensure that the solution is less than or equal to \(w \times C^*(C^*\) is the cost of the optimal solution). The bounded suboptimal search algorithm achieves a balance between the quality and running time of the solution. WA*[9] and others [10] are well-known bounded suboptimal algorithm. Optimal and suboptimal algorithms have their own advantages and disadvantages, and they behave differently in different scenarios. In this paper, we combine the search algorithm with the actual situation and introduce a variant of CBS.

3. Problem description

3.1. MAPF

MAPF is defined as an undirected graph \(G = (V, E)\) and a set of agents labelled \(a_1, a_2 ... a_k\), each agent \(a_i\) is allocated a task with a unique start position \(s_i \in V\) and goal position \(g_i \in V\). At each time step, an agent takes an action to move to adjacent or wait in the current position. For each action move or wait, the cost is one. The cost of an agent is equal to the number of time steps required from the start position to goal position. For the whole problem, the total cost is the sum of the cost for all agents.

We use the term path only in the context of a single agent and use the solution to denote all paths for the given agents. A path for agent \(a_i\) is a sequence of its move and wait actions, which can be recorded as a sequence \(P = v_1 v_2 ... v_k ...\) where \(V \in V\) and \(i = 0, 1, 2 ... k ...\). If both agents \(a_i\) and \(a_j\) occupy vertex \(v\) at time point \(t\), we record this point conflict as a tuple \((a_i, a_j, v, t)\). In order to solve it, constraints \((a_i, v, t)\) and \((a_j, v, t)\) are added to agent \(a_i\) and \(a_j\) respectively. A constraint \((a_i, v, t)\) means \(a_i\) cannot occupy vertex \(v\) at time step \(t\).

In MAPF problem, the agents \(a_i\) and \(a_j\) are not allowed to cross the same edge \((v_1, v_2)\) at opposite direction simultaneously. If happens, we call it edge conflict \((a_i, a_j, v_1, v_2, t)\) and add edge constraint \((a_i, v_1, v_2, t)\) which means agent \(a_i\) is forbidden to move from \(v_1\) to \(v_2\) at the time \(t\) to solve it.

3.2. The Conflict Based Search algorithm (CBS)

CBS is a two-level algorithm, including high and low levels. The high-level of CBS searches the constraint tree (CT), which is a binary tree. The pseudo-code for the high level is shown in Algorithm 1. Each node \(N\epsilon CT\) contains three types of information.

1. a set of new constraints for agents \(N.constrains\)
2. a current solution for MAPF problem \(N.solution\)
3. The cost of N.solution (N.cost)

The root node $R$ of CT has no constraints (line 1). The low-level search is invoked and return one shortest path for every agent $a_i$ (line 2). After calculating the cost of R.solution, the root node $R$ is inserted into open table. The high-level search performs a best-first search on the CT while the nodes are ordered by their costs. Then get a node $N$ from open table, we validate the N.solution (line 7). If there is no conflict between any two agents, the CT node $N$ would be regarded as a goal node and the N.solution would be returned (line 9). Otherwise, the first conflict $C_n = (a_i, v_1, v_2, t)$ and the node $N$ is declared as non-goal. To solve the conflict and guarantee getting the optimal solution, node $N$ generates two child CT nodes that inherit the constraints from its parent and are added a new constraint respectively. For example, a new $(a_i, v_1, t)$ constraint is added for agent $a_i$ (line 14). Then the low-level is invoked to replan, which is responsible for planning an optimal path for a single agent that satisfies all its constraints while completely ignoring the other agents (line 16).

4. Algorithm Design

4.1. Divide and Conquer

Divide and conquer is to divide a complex problem into two or more identical subproblems, then divide these subproblems into smaller ones until the smallest subproblems can be solved directly. And the solution of the original problem is the combination of the solutions of the subproblems. In this paper, the divide and conquer idea is applied to CBS algorithm. By transforming the problem of large-scale map into small-scale subproblems, the large-scale problem can be solved by finding solutions for subproblems. Experimental results indicate that the solving time using this method can be well shortened.

4.2. Divide and Conquer Conflict-Based Search algorithm (DC-CBS)

The core idea of DC-CBS is to divide the actual map into blocks and abstract into smaller regional map, namely upper grid. Correspondingly, we can call actual map lower grid. Each upper point corresponds to a region in the lower grid. As shown in Figure 1(a) and Figure 1(b), the size of lower grid and upper grid is $12 \times 12$ and $3 \times 3$ respectively. The lower grid is divided into 9 regions and

The time complexity of CBS is discussed as follows. In the worst case, the branching factor of CT is 2 and $2^{2d_p}$ nodes must be expanded where $d_p$ is the depth of CT. When generating a CT node, a new constraint is added. In the worst case, every agent will be constraint to avoid every other vertex except one at every time step. We use $C^*$ denotes the total steps of all agents. Thus, the upper bound of CT nodes is $2^{|V| \times C^*}$. For each node, the low level is invoked and expands at most $|V| \times C^*$ (single-agent) states. The number of states expanded in the low-level is $(2^{|V| \times C^*} \times |V| \times C^*)$. 

---

Algorithm 1 High-level of CBS

```plaintext
Input: MAFInstance
1. R.constraints = δ
2. R.solution = find paths involving the low-level()
3. R.root = SC(R(solution))
4. insert R to Open
5. while Open not empty do
6. N = last node from Open
7. Validate the paths in N until a conflict occurs
8. if N has no conflict then
9. return N.solution
10. end if
11. C = first conflict(a_i, a_j, v_t) in N
12. for each agent a_i in C do
13. A = new node
14. A.constraints = N.constraints + (a_i, a_j, v_t)
15. A.solution = N.solution
16. Update A.solution by invoking low-level(a_i)
17. A.cost = SC(A.solution)
18. insert A to Open
19. end for
20. end while
```

Algorithm 2 DC-CBS

```plaintext
Input: MAFInstance
1. upperProblem = change to upper()
2. upperSolution = find individual paths by Is
3. for each path p in upperSolution do
4. subProblem. startNode = p.front()
5. if start the first nodes of subProblem
6. p, pop, front()
7. end for
8. while point exists in upperSolution do
9. for path p in upperSolution do
10. tempNode = SG-Art(p, startNode, endSet)
11. subProblem.endNode = tempNode
12. if the end nodes of subProblem
13. p, pop, front()
14. Update subSolution by invoking CBS(subProblem)
15. subProc.startNodes = subProc.endNodes
16. end for
17. end while
```

Algorithm 3 SG-A*

```plaintext
Input: (startNode, endSet)
1. openList.push_back(startNode)
2. while (openList is not empty()) do
3. curPoint = node with lowest F value
4. openList, remove(curPoint)
5. closelist.push_back(curPoint)
6. if curPoint in endSet then
7. openList, close() closelist, close()
8. return curPoint
9. end if
10. get surroundPoints of curPoint
11. for point p in surroundPoints do
12. if P in closelist then
13. do nothing
14. end if
15. if P in openList then
16. update F value
17. end if
18. if P not in openList then
19. add p into openList
20. end if
21. end if
22. end for
23. end while
```
each one shrinks to an upper point in the upper grid. Points and regions with the same color correspond to each other. For example, the red region in Figure 1(a) corresponds to the red point in Figure 1(b) and so on.

Figure 1. (a) an example of lower grid. The solid circles are the start points, while the dashed circles mean the temporary end points. The stars mean the actual end points. The dashed arrows represent the paths for agents. (b) the corresponding upper grid for (a). The solid circles are the start points and the the dashed circles are end points. The solid arrows represent the rough paths for agents

In the process of solving the MAPF, the start and goal points of the agents in the lower grid are mapped to the upper grid firstly. Then the rough paths are obtained by A* algorithm. That means, the direction of all agents to perform tasks are decided on the upper grid. Next, the two adjacent upper points from rough paths are used to search the temporary start and end points in the lower grid. Actually, a edge in the rough path corresponds to a task of lower grid. After traversing all the edges into tasks and solving all the subproblems, the paths of the whole MAPF are planned completely.

In terms of the upper grid, it is mainly used to plan the rough path of all agents and guide the direction of agents. Rough path refers to the sequence of directions that need to be changed from the start to the goal point, which is composed by a series of continuous coordinate points. For example, as shown in Figure 1(a), assuming there are three tasks, start points from (1,2), (2,1), (9,11) and end points are (2,10), (10,6), (7,0) respectively. Three tasks are projected into the upper grid, as shown in the Figure 1(b). After mapping, the rough paths can be planned as follows:

\[\begin{align*}
A_1 & \quad (0,0) \rightarrow (0,1) \rightarrow (0,2) \\
A_2 & \quad (0,0) \rightarrow (0,1) \rightarrow (1,1) \rightarrow (2,1) \\
A_3 & \quad (2,2) \rightarrow (1,2) \rightarrow (1,1) \rightarrow (1,0)
\end{align*}\]

Pseudo-code for DC-CBS is shown in Algorithm 2. Because an upper point corresponds to many points of the lower, there is no conflict between rough paths. We adopt A* algorithm without considering conflicts to plan the rough path for each agent (line 2).

The main function of the lower grid is to generate small MAPF subproblems and call CBS to solve them. Suppose the rough path has been planned in the upper grid, we use two adjacent upper points to plan actual path for every agent. The key of planning the actual path is how to determine the temporary end points that must be passed in the process of executing the task. We propose the Search Goal A* (SG-A*) algorithm to search the temporary endpoints and it can ensure the end point is closest one. The biggest difference between A* and SG-A* is that the latter uses a set of endpoints instead of a fixed one. The endpoint set is composed of allowable nodes in a lower region corresponding to a single upper point. The pseudo-code for SG-A* is shown in Algorithm 3. SG-A* will stop search when an allowable point \( p \) in the endpoint set. Point \( p \) will be taken as a temporary endpoint and returned (line 6 - line 9).

For example, as shown in Figure 1(a) and Figure 1(b), in the upper grid, \( A_1 \) and \( A_2 \) need to go from \((0,0)\) to \((0,1)\). The actual start point of \( A_1 \) is \((1,2)\) (Figure 1(a)) and its endpoint set contains all allowable nodes corresponding to upper nodes \((0,1)\) (Figure 1(b)). The SG-A* algorithm finds the nearest temporary endpoint \((1,4)\) (Figure 1(a)). Next, \( A_1 \) directly plans the path to the actual end point \((2,10)\) (Figure 1(a)) to complete the planning task, because the actual endpoint is in the adjacent
region. Similarly, the actual start point of $A_2$ is (2,1) (Figure 1(a)). Due to (2,4) position is an obstacle and (1,4) position has been occupied by $A_1$, (3,4) is chosen as the current optimal position.

When the temporary endpoints of all agents are determined, they are combined into a small-scale MAPF problem and then CBS algorithm is invoked to solve it (line 14). If the original MAPF problem is not planned completely, the latest temporary endpoints of these agents will be taken as the new start points (line 15). And the next temporary endpoints will be searched again. The combination of all subsolution is the solution of the original MAPF problem.

4.3. Algorithm Analysis
The time complexity analysis of CBS is also applied for DC-CBS. In the worst case, for example, the subproblems adopt the whole map and expand all points, the time complexity of DC-CBS is $O(2^{|V|} \times C^* \times |V| \times C^*)$. But in practice, DC-CBS algorithm runs faster and takes less time to find the solution. The original problem is divided into several subproblems, in which the start point and the end point of each task are very close and it takes little time to plan the path. Hence, the running time of DC-CBS is shorter than that of CBS.

5. Experimental Results
We have done extensive experiments to compare the algorithms. Our experiments were conducted on an Inter(R) Core i5 and processor operates at 3.10GHz and has 4GB of RAM. All code for experiments was written in C++ and experiments were conducted on Visual Studio 2015.

![Figure 2. Comparison between CBS and DC-CBS in terms of run time (a) and cost (b). The influence of different upper grid size is shown as (c).](image)

Firstly, we compare the DC-CBS algorithm with the original CBS algorithm. We use 80 $\times$ 80 4-connected grid with generating 10% obstacles randomly. At the same time, each agent is arranged a task with unique start and goal points. For DC-CBS algorithm, the size of upper grid is 5 $\times$ 5.

Figure 2(a) shows the running time(s) for CBS and DC-CBS. Compared with CBS algorithm (labeled as the red line), the DC-CBS algorithm (labeled as the blue line) clearly outperforms it. It should be noted that CBS algorithm cannot find a solution within four minutes while the DC-CBS algorithm could obtain the solution with half a minute when the number of agents is over 70. When the number of agents reaches 45, as shown in Figure 2(a), the running time is greatly reduced. The reason is that agent tasks are generated randomly and it is perhaps that the distance of the random start and goal points is short. All in all, DC-CBS is much better than CBS algorithm in running time. Figure 2(b) presents the relationship between the number of agents and the total cost of paths generated by CBS and DC-CBS respectively. It is well known that CBS is the optimal algorithm that can obtain the lowest cost. And it can be seen that the gap between the two algorithms is quite small when the number of agents is no more than 70. As for large-scale cases (e.g. 100 agents), DC-CBS still perform well while CBS is not available any more.

Furthermore, the size of upper grid also has a significant effect on running time. Figure 2 show the running time of 20 random tasks on a 60 $\times$ 60 4-connected grid with 10% obstacle rate. As shown in the figure, the size of upper grid has a little impact on the running time when the grid is between 5 $\times$ 5 and 15 $\times$ 15. Too large or too small size of upper grid would increase running time. When the
size of grid is too small, such as $1 \times 1$, DC-CBS is the same as the original algorithm. And when the size of the upper grid is larger than $15 \times 15$, the running time is also increase. The main reason is that the running time spent on dividing subproblems has increased. Consequently, DC-CBS can greatly reduce the running time with the appropriate size of upper grid.

From the above experiments, it is obvious that DC-CBS takes less time to solve large-scale problem combined with moderate size of upper grid, and the quality of the solution is almost as good as the optimal solution.

6. Conclusion
We studied the MAPF problem in large-scale scenarios. In most cases, CBS cannot obtain the solution in three minutes, which is not acceptable in practical application. Therefore, we propose the DC-CBS algorithm which combines Divide-and-Conquer technique with CBS algorithm. DC-CBS divides the actual map into upper and lower grids, where a region of lower grid shrinks to a point of upper grid. Then the original problem is decomposed into several subproblems. In the upper grid, we introduce the rough path which is planned by A* without considering conflicts to guide directions for the agents. In the lower grid, SG-A* is proposed to search the temporary endpoint. It guarantees the temporary endpoint is the best one which is closest to the start point. Experimental results show that DC-CBS is more efficient to get the solution than CBS and the quality is almost as good as the optimal solution.

Acknowledgments
This work is financially supported by National Key R&D Program of China under Grant No. 2017YFB0803002 and No. 2016YFB0800804, National Natural Science Foundation of China under Grant No. 61672195 and No. 61732022.

References
[1] Jingjin Yu and Steven M. Lavalle. Structure and intractability of optimal multi-robot path planning on graphs. In Twenty-seventh Aaai Conference on Artificial Intelligence, 2013.
[2] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. Journal of Artificial Intelligence Research, 31(3):591–656, 2008.
[3] Peter R. Wurman, Raffaello D’Aposandrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. In National Conference on Innovative Applications of Artificial Intelligence, 2007.
[4] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. In Twenty-sixth Aaai Conference on Artificial Intelligence, pages 40–66, 2012.
[5] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan Sturtevant, Jonathan Schaeffer, and Robert C Holte. Partial-expansion a* with selective node generation. In Twenty-sixth Aaai Conference on Artificial Intelligence, 2012.
[6] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.
[7] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. Artificial Intelligence, 195(195):470–495, 2013.
[8] David Silver. Cooperative pathfinding. In Aaai Conference on Artificial Intelligence & Interactive Digital Entertainment, 2005.
[9] Ira Pohl. Heuristic search viewed as path finding in a graph. Artificial Intelligence, 1(3):193–204, 1970.
[10] Jordan Tyler Thayer and Wheeler Ruml. Faster than weighted a*: An optimistic approach to bounded suboptimal search. In ICAPS, pages 355–362, 2008.