Suboptimal Variants based on CBS Algorithm in MAPF problem

Group WOW

Sibei Zhou (301416917)

ChenKun Zhang (301401736)

Honghao Yu (301385669)

Abstract:

The multi-agent path finding (MAPF) problem is finding a solution of paths for multiple agents in a 2D map with no collision at all. At present, CBS is a more effective method to find the optimal solution. However, CBS algorithm has the disadvantages of a long time and high memory consumption. By modifying the algorithm in the high level and low level of CBS, we control the loss of the cost to a small degree, and greatly reduce the CPU time and memory required for operation (which is shown by the number of expanded nodes).

1. Introduction:

The task of the multi-agent Path finding problem (MAPF) is to find paths for multiple agents, each problem with a different starting point (in a set of start locations s1, s2, ..., sn) and target location (a set of goal locations g1, g2, ..., gn) and multiple (2 or more) number of agents, and the paths of the solution should make sure that all agents do not collide. The conflict-based search (CBS) algorithm is a successful optimal MAPF solver, it is a two-level algorithm, and its special conditions ensure that it returns an optimal solution. In this report, we give up the traditional method using the CBS algorithm to find the optimal solution, and try to find some sub-optimal solution with the fastest speed within a small range of loss in cost, and there we use 2 kinds of CBS-based suboptimal MAPF solvers which relax the high-level and/or the low-level searches, which are Bounded Suboptimal CBS (BCBS) and Enhanced CBS (ECBS). And we also do some experiments for those MAPF solvers using various parameters to better compare them.

The Experimental results show the superiority of these methods, they all speed up over the traditional CBS while using less memory space.

2. Implementation:

2.1. CBS

CBS provide a way to find a obtimal result by adding constraint to a constrain tree. When CBS find a new paths for each agent, if the path has new conflicts, the first conflict will be added the constraint tree. And the CBS will find a result when there is no conflict. Hence it would be possible that the CBS will fall into the infinitely loop if there is no solution.

The CBS has 2 level, one is high and another is low. The high level will search the constraint tree and find the node with the lowest cost. As the low level, it will find the minimal value of g_val and h_val in a node, where the g_val is the distance where the current position to the original position and the h_val is the shortest distance from current position to the goal position which is calculated by dijkstra algorithm.

Pseudo-code for CBS is shown in Figure 1 below.

Algorithm 1: High-level search of CBS.

Input: Representation of the environment, start cells, and goal cells
Result: optimal collision-free solution

```
R.constraints ← ∅
R.paths ← find independent paths for all agents using a_star()
R.collisions ← detect_collisions(R.paths)
R.cost ← get_sum_of_cost(R.paths)
insert R into OPEN
while OPEN is not empty do
    P ← node from OPEN with the smallest cost
    if P.collisions = ∅ then
        return P.paths // P is a goal node
    collision ← one collision in P.collisions
    constraints ← standard_splitting(collision)
    for constraint in constraints do
        Q ← new node
        Q.constraints ← P.constraints ∪ {constraint}
        Q.paths ← P.paths
        a_i ← the agent in constraint
        path ← a_star(a_i, Q.constraints)
        if path is not empty then
            Replace the path of agent a_i in Q.paths by path
            Q.collisions ← detect_collisions(Q.paths)
            Q.cost ← get_sum_of_cost(Q.paths)
            Insert Q into OPEN
return 'No solutions'
```

Figure 1: Pseudo-code for CBS

2.2. BCBS

Since the CBS would abandoning some node that is very close to the solution, only because they do have a high cost. Moreover, the high level of CBS does not use the heuristic guidance WA*, so we consider using the Focal Search, which can give us a bound suboptimal search. This would be BCBS algorithm based on the CBS by adding the focal search which is combained the A* and A_ϵ (Barer, Max,2014). Focal search not only contain the open list which is same as the A* but

also adding a focal list. As the focal list it contain nodes that are from the open list. As the equation is showed below f_1 is the min value from the open list and the w is a parameter that we are choosing by ourself normally is bigger then 1. The nodes that are from Open list satisfy the following equation will be added to the focal list.

$$f_1(n) < w \cdot f_{1min}$$ (Barer, Max,2014)

By adding the focal search to the high level, we apply this function focal_search(g(n),h_c) to get node from the constrain tree. To be more specific, the h_c is the heuristic function related to the number of collisions of the node n, and the g(n) is cost from the node. To be clearly, the all the node are from the open list.

As the low level the function will be changed to a different function: focal_search(f(n),h_c), where the h_c still same as the h_c in the high level function but the f(n) is same as the f(n) in the A* search, where f(n) = g(n) +h(n).

Pseudo-code for BCBS is shown in Figure 2 below.

2.3. ECBS

However, while we testing the w_h and w_l we find that it is hard distribute them. Hence it probably be better if we are not fix the w_h and w_l in the algotithm. So we decide to using the ECBS which might can improve the performance. In ECBS, the weight used in focal search in low level is same as the high level, and its low-level search is the same as BCBS(1,w) which only uses only low-level focal search, but it will return the cost of node t which a node from constrain tree and LB(n).

$$LB(n) = \sum_{i=1}^{k} f_{min}(i)$$ (Barer, Max,2014)

As the high level the focal list will be different than the BCBS.

$$Focal\ list = \{n | n \in OPEN, n.cost \leq LB \cdot w\}$$

And the Pseudo-code for ECBS is shown in Figure 3 below.

Algorithm 2: BCBS

Input: Representation of the environment, start cells, and goal cells
Output: optimal collision-free solution

```
1   R.constraints ← ∅
2   R.paths ← apply focal search to find relative shorter paths with minimal collision
                for all agents using a_star()
3   R.collisions ← detect_collisions(R.paths)
4   R.cost ← get_sum_of_cost(R.paths)
5   insert R into OPEN
6   while OPEN is not empty do
7       min_cost ← the smallest cost of nodes from OPEN
8       for N ← node form OPEN with N.cost <= w * min_cost
9           Insert N into FOCAL
10      P ←node in FOCAL with the smallest number of collisions
11      if P.collisions = ∅ then
12          return P.paths // P is a goal node
13  collision ← one collision in P.collisions
14  constraints ← standard_splitting(collision)
15  for constraint in constraints do
16      Q ← new node
17      Q.constraints ← P.constraints ∪ {constraint}
18      Q.paths ← P.paths
19      ai ← the agent in constraint
20      path ← a_star(ai, Q.constraints)
21      if path is not empty then
22          Replace the path of agent ai in Q.paths by path
23          Q.collisions ← detect_collisions(Q.paths)
24          Q.cost ← get_sum_of_cost(Q.paths)
25          Insert Q into OPEN
26  return 'No solutions'
```

Figure 2: Pseudo-code for BCBS

Algorithm 3: ECBS

Input: Representation of the environment, start cells, and goal cells
Output: optimal collision-free solution

```
1   R.constraints ← ∅
2   R.paths ← apply focal search to find relative shorter paths with minimal collision
              for all agents using a_star()
3   R.collisions ← detect_collisions(R.paths)
4   R.cost ← get_sum_of_cost(R.paths)
5   insert R into OPEN
6   while OPEN is not empty do
7       for node n in OPEN
8           LB(n) ← the sum of the smallest cost of nodes from OPEN in low-level
9       LB ← min(LB(n)) for all node n in OPEN
10      for N ← node form OPEN with N.cost <= w * LB
11          Insert N into FOCAL
12      P ←node in FOCAL with the smallest number of collisions
13      if P.collisions = ∅ then
14          return P.paths // P is a goal node
15  collision ← one collision in P.collisions
16  constraints ← standard_splitting(collision)
17  for constraint in constraints do
18      Q ← new node
19      Q.constraints ← P.constraints ∪ {constraint}
20      Q.paths ← P.paths
21      ai ← the agent in constraint
22      path ← a_star(ai, Q.constraints)
23      if path is not empty then
24          Replace the path of agent ai in Q.paths by path
25          Q.collisions ← detect_collisions(Q.paths)
26          Q.cost ← get_sum_of_cost(Q.paths)
27          Insert Q into OPEN
28  return 'No solutions'
```

Figure 3: Pseudo-code for ECBS

3. Methodology:

We aim at improving CBS in space or time consumption in this project, to achieve this, we propose BCBS and ECBS based on traditional CBS. Therefore, the first two questions we want to solve is whether these two algorithms take effects saving space or time. Besides, we want to see if the different values of weights

multiplied on high- and low-level searching and the different choice of the heuristic function will make difference in the experiments.

3.1. Experiments

We design five experiments to find the answer for our questions.

The first experiment is to compare different choices of heuristic functions in BCBS and ECBS (separately), and we want to check which heuristic function will improve the algorithm to find the suboptimal solution faster and generates/expand fewer extra nodes. There are 2 heuristic functions that will be used: The number of conflicts heuristic (h1) and the number of agents with conflicts heuristic (h2). And the result shows that in most cases, h1 runs slightly faster than h2 on average but h2 is more robust across different instances. Therefore, whenever we refer to hc we relate to h2.

The second experiment is to compare the three different BCBSs using focal search in only high-level search and in only low-level search on their running time and the sum of generated and expanded nodes to see in which level the focal search used is more efficient.

The third experiment is to compare different values of weights using in the focal search of BCBS and ECBS with their baseline CBS, we want to improve the efficiency by using different weights. There are 3 different values of weights we used: 1, 1.2, 1.5. And easy to see that when the BCBS and ECBS algorithms' both weights using in high- and low-level equal to 1, it is actually traditional CBS, so we choose traditional CBS as the baseline and check the ratio of ECBS and BCBS got.

The fourth experiment is to compare different number of agents, percentage of obstacles in instances, and the size of the instances in BCBS and ECBS algorithm, we want to know how does the performance of ECBS vary as a function of those 3 varibles.

3.2. Instances

We compared the three methods for our test on 50 MAPF instances. In order to better show the differences in the three dimensions of each algorithm (the algorithm used, weights used for high- and low-level focal search, different heuristic functions), except for the map tested in the individual assignment (especially test_40.txt - test_50.txt), we use the generate_map.py

to generate our own maps instances. The map's size is fixed by 16*16 and agent location is generated randomly.

4. Experimental setup:

The experiment was run on Macbook Pro using the M1Pro chip with 16G RAM and an 8 core CPU. The entire experiment was coded in version 3.9.12 of Python using PyCharm (2022.1.3).

Run "python run_experiments.py --instance test/* --solver CBS --batch" in terminal will run and generate the output of all the test instances.

Run "python run_experiments.py --instance test/largeTest* --solver CBS --batch" in terminal will run and generate the output in 5.4 (Compare different number of agents case for BCBS and ECBS.).

5. Experimental result:

In different test case, the ratio can better reflect the improvement of the two methods in terms of time and number of generating nodes compared with traditional cbs, instead of using actual running time and node number directly. The lower ratio, the better performance for reducing time and node number.

5.1. Compare different heuristic functions in BCBS and ECBS

We choose the instance from individual assignment's instances which is test_47.txt to do this experimental, because it is the test which spends the longest CPU time with most generated/expanded nodes in our individual assignment, so it should show a clear difference between efficiency. From figure 4 that shows the relationship of method with different weights and CPU time, and figure 5 that shows the relationship of method with different weights and number of node generated and expanded, we can see that for the time cost, when we test in test_47.txt instance, the number of agents with conflicts heuristic (h2) performs better than the number of conflicts heuristic (h1).

And then we use all instances in the individual assignment, and got similar result like that. This result is consistent with what we know that the number of conflicts heuristic (h1) used will truly lead to spending more time and space in local search.

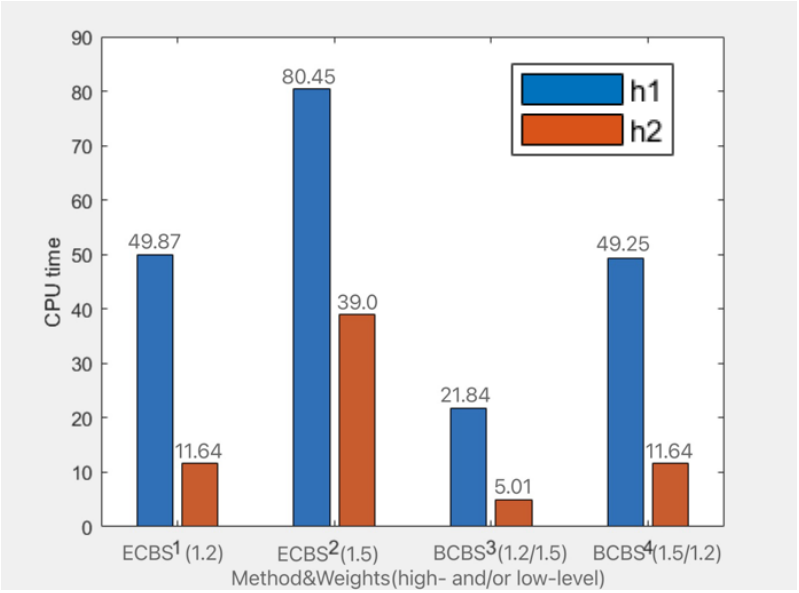

Figure 4: Method&Weights(high- and/or low-level) VS. CPU time

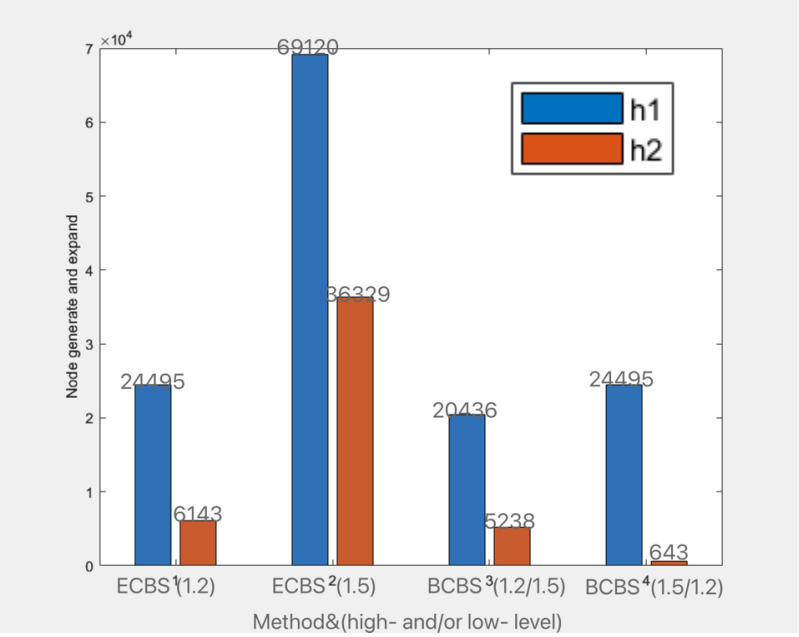

Figure 5: Method&Weights(high- and/or low-level) VS. Nodes generated/expanded

5.2. Compare focal search in low-level and high-level in BCBS

In the figure 6 and figure 7 below, we can see that the BCBS with only low-level perform better than BCBS with the only high-level if we focus on space efficiency, because it always get the fewer number of the sum of generated and expanded node, while the result of CPU running time shows that BCBS with the only high-level focal search is faster when weight larger

than 1.2. And this situation happened in most case for the all instance we used. So it is the evidence to show that the effect of the low-level focal search is larger than the high-level focal search in BCBS algorithm.

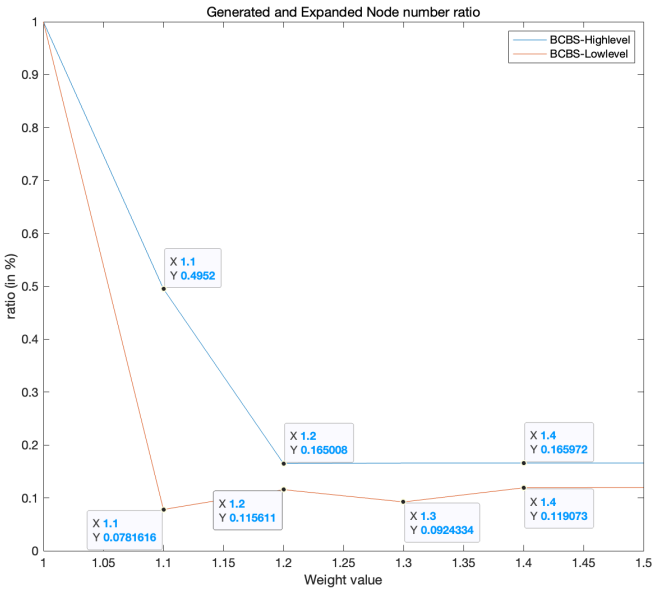

Figure 6: Weight value in focal search VS. Number of nodes generated/expanded ratio

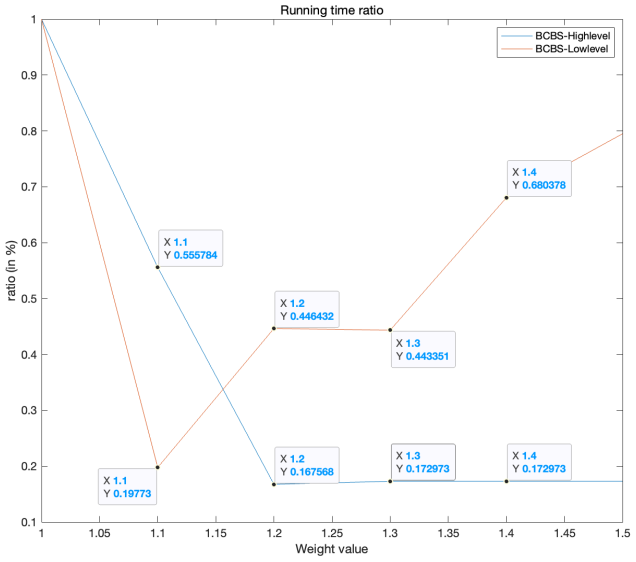

Figure 7: Weight value in focal search VS. Running time ratio

5.3. Compare different weights in the focal search of BCBS and ECBS

If any of the high and low level is 1, that means we did not modify this layer and keep it be the traditional CBS method. For example, BCBS(1,1.2) means it only use focal search in high-level with weight 1.2, and BCBS(1, 1) is actually traditional CBS without any focal search used.

In the tables below show the relationship between weights and number of node generated and expanded, and also the relationship between weights and CPU running time for both BCBS and ECBS. We found that in this case, the algorithm get the highest time efficiency (lowest running time) at BCBS(1.2, 1.5), and by comparing the two algorithm's data with the same weights used in focal search (like BCBS(1.2, 1.2) and ECBS(1.2)), we can see that BCBS always get higher efficiency and in fewer cases lower efficiency, but this situation is reversed in other instances, so we think the weight chosen to get a better perform should depend on the instances.

High Low	1	1.2	1.5
1	592286 (100%)	98282 (16.6%)	98303 (16.6%)
1.2	68475 (11.56%)	5238 (0.8%)	5238 (0.8%)
1.5	70784 (12%)	6143 (1.03%)	6143 (1.03%)

Table 1: BCBS(h2) Node Number data table

High Low	1	1.2	1.5
1	205.68 (100%)	31.73 (15.427%)	31.74 (15.432%)
1.2	81.6 (29.673%)	5.12 (2.489%)	5.01 (2.436%)
1.5	144.59 (70.299)	11.64 (5.659%)	11.67 (5.674%)

Table 2: BCBS Running time table

Weight	1	1.2	1.5
Node Number	592286 (100%)	36329 (6.134%)	6143 (1.037%)

Table 3: ECBS Node Number data table

Weight	1	1.2	1.5
Running time	205.68 (100%)	39 (18.961%)	11.46 (5.572%)

Table 4: ECBS Running time table

5.4. Compare different numbers of agents case for BCBS and ECBS.

We have get the fittest w-value from previous test. In this case, we increase the size of the map and the number of the agent to test the ECBS and BCBS methods. In Figure 8 and Figure 9, the map's size is fixed by 16*16 and generated randomly by generate_map.py.

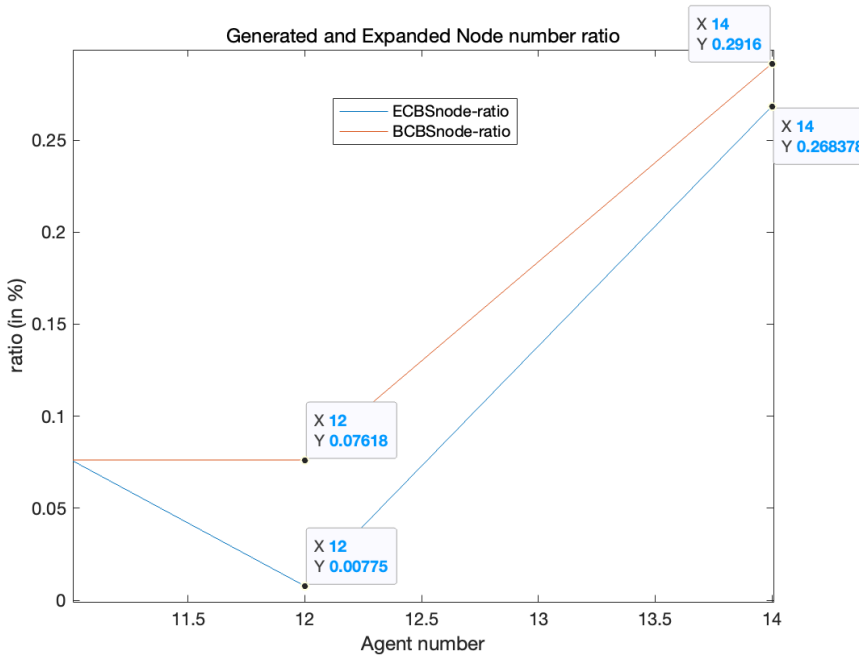

Figure 8: Different number of agents VS. Number of nodes generated/expanded ratio

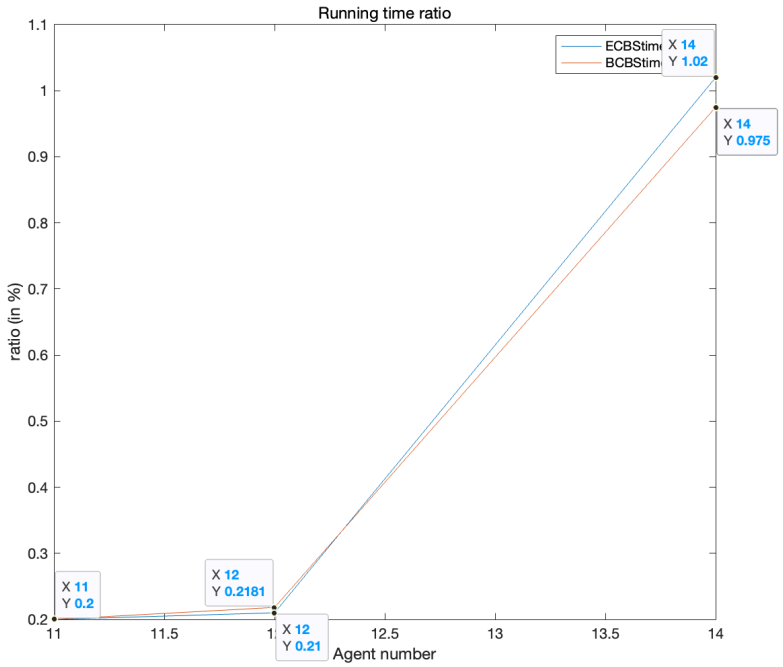

Figure 9: Different number of agents VS. CPU running time ratio

6. Conclusion section:

As the Herustic cost in BCBS and ECBS, we found that the h2 would have better performance than the BCBS. And we believe the h2 which is the number of agents with conflicts could bring more improvement to the BCBS because the number of conflicts heuristic (h1) used will always lead to spending more time and space in local search.

As the weights in BCBS use only high-level focal search and only low-level focal search, the former would be stable after w >= 1.15, while the latter would be very unstable, it got the lowest point at w = 1.1 and the performance would gradually decrease when w >= 1.3, showing an increase in CPU running time and more expanded nodes.

As the difference between weights value of both high- and/or low-level, CBS, ECBS and BCBS perform differently. Hence, when we consider the distribution of w_h and w_l we believe w_h = 1.5 and w_l =1.2 would have a better performance on the BCBS in the case that we test. As ECBS would be easy to conclude, since the w_h = w_l for ECBS. We believe when w_h = 1.2 is the best choice for ECBS in this case.

From our testing, we found that the BCBS owns almost the same performance as ECBS, for some tests, the BCBS even has a better performance than the ECBS. To be more specific, ECBS would have more advantages in generating fewer nodes to get the solution, but the BCBS could get the result in a short time. Moreover from the testing, we found that when the number of agent increase, the distance between ECBS and BCBS becomes larger by comparing their Running time. However, the gap between the BCBS and ECBS becomes smaller while the number of agent increase.

7. Reference:

1. Barer, Max, Guni Sharon, Roni Stern and Ariel Felner. “Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem.” *Symposium on Combinatorial Search* (2014).
2. Hatem, M.; Stern, R.; and Ruml, W. 2013. Bounded suboptimal heuristic search in linear space. In SOCS.

3. Sharon, G., R. Stern, A. Felner, and N. Sturtevant. “Conflict-Based Search For Optimal Multi-Agent Path Finding”. *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 26, no. 1, Sept. 2021, pp. 563-9, doi:10.1609/aaai.v26i1.8140.