1.Introduction

Conflict-Based Search (CBS) is a leading algorithm for optimal Multi-Agent Path Finding (MAPF). Solving pathfinding problems optimally is commonly done with conflict search algorithms based on A* search. Normally, such algorithm perform a best-first search and find the optimal path solution guided by $f(n) = g(n) + h(n)$, where g(n) is the cost of the shortest path from the start state n and h(n) is a heuristic function estimating the cost from n to the nearest goal state.

In this final project, we will explore the feasibility and performance of iterative deepening conflict based search(IDCBS) iterative linear conflict based search (ILCBS). Through comparing with the basic conflict based search (CBS), their advantages and disadvantages will be figured out and discussed. In detail, based on the characteristics of the specific problem, cases that ILCBS and IDCBS are more efficient than IDCBS and CBS will be described, and circumstances that ILCBS is inferior will also be discussed. Although ILCBS is less efficient than IDCBS and CBS, it has superior performance in other scenarios. After theoretical methodologies are presented, experiments and results will be presented to support theoretical understandings.

2.Implementation

The input to the multi-agent pathfinding problem can be defined as follows.

The *input* to the *multi-agent pathfinding problem* (MAPF) is:

(1) A directed graph $G(V, E)$. The vertices of the graph are possible locations for the agents, and the edges are the possible transitions between locations.

(2) k agents labeled $a_1, a_2 \ldots a_k$. Every agent a_i has a start vertex, $start_i \in V$ and a goal vertex, $goal_i \in V$.

Each agent in the problem can perform a move action to a neighboring vertex or a wait action to stay at the current vertex between success time points. Actions chains can designed for all agents in a specific time period and a combination of action chains can be decided as the solution for MAPF the problem without conflicting and achieving the minimal value for the cost functions.

The main constraint in MAPF is that each vertex can only be occupied by at most one agent at a specific time point. If this constraint is violated, the conflict will occur. Besides, when two agents switch their position with each other at the same timestamp (which cannot happen physically), it is also considered a violation of the constraint.

The solutions can be optimized by minimizing the cost functions. Moreover, the cost function can also be regarded as a method to select the optimal models (model selection). The most commonly used cost function is the sum of time costs. To be specific, the total number of times that agents cost will be recorded, and the solution

with the minimal time counts will be selected as the optimal one. Another common cost function is applied to minimize the total time until the last agent reaches its destination or the total amount of distance traveled by all agents.

Such cost function works as the global cost function, where the total costs are measured in all agents together. However, the global cost function cannot be used sometimes, and then a set of individual cost functions can be applied. This type of cost function is known as mulit-objective optimization.

3.Methodology

Basic conflict-based search can solve the MAPF problem by decomposing it into a large number of constrained single-agent pathfinding problems. The solutions of these problems can be solved in time proportional to the size of the map and length of the solution.

In the CBS, the path is targeting to a single agent, and the solution is the clustering of all paths for all agents in this problem. A constraint is a tuple (a,v, t) where agent a is prohibited from occupying vertex v at time step t. During the course of the algorithm, agents will be associated with constraints. A consistent path for agent is a path that satisfies all its constraints. Likewise, a consistent solution is a solution that is made up from paths, such that the path for any agent is consistent with the constraints of agent. A conflict is a tuple (ai,aj,v,t)where agent ai and agent aj occupy vertex v at time point t. A solution (of k paths) is valid if all its paths have no conflicts. A consistent solution can be invalid if, despite the fact that the individual paths are consistent with the constraints associated with their agents, these paths still have conflicts.

The key idea of CBS is to grow a set of constraints and find paths that are consistent with these constraints. If these paths have conflicts, and are thus invalid, the conflicts are resolved by adding new constraints.

CBS works in two levels. At the high level, conflicts are found and constraints are added. CBS searches a constraint tree and a CT a binary tree. Each node in the CT contains a set of constraints, a solution and the total cost. Node N in the CT is a goal node when N solution is valid, i.e., the set of paths for all agents have no conflicts. The high-level performs a best-first search on the CT where nodes are ordered by their costs

The low level finds paths for individual agents that are consistent with the new constraints.

Algorithm 1: High-level of CBS

```
1  Main(MAPF problem instance)
2      R ← new CT node
3      R.constraints ← {}
4      R.solution ← {a shortest path for each agent}
5      R.cost ← SIC(R.solution)
6      R.conflicts ← Find Conflicts(R.solution)
7      Insert R into OPEN
8      while OPEN not empty do
9          N ← node with lowest f from OPEN
10         Delete N from OPEN
11         if N has no conflict then
12             return N.solution // N is a goal node
13         Classify N.conflicts into types
14         Compute N.h if it has not yet been computed
15         if N.f > lowest f in OPEN then
16             Insert N back into OPEN
17             continue
18         C ← Choose Conflict(N)
19         Children ← []
20         foreach agent a in C = ⟨a_i, a_j, v or e, t⟩ do
21             N' ← Generate Child(N, ⟨a, v or e, t⟩)
22             if N'.cost = N.cost and
                |N'.conflicts| < |N.conflicts| then
23                 N.solution ← N'.solution
24                 N.conflicts ← N'.conflicts
25                 Children ← [N]
26                 break
27             Insert N' into Children
28         Insert Children into OPEN
29     return "No solution"
30 Generate Child(Node N, Constraint C on a_i)
31     N' ← new CT node
32     N'.constraints ← N.constraints ⋃ {C}
33     N'.solution ← N.solution
34     CAT ← Build CAT(N.solution, i)
35     N'.solution_i ← Low Level(a_i, CAT)
36     N'.cost ← SIC(N'.solution)
37     N'.conflicts ← Find Conflicts(N'.solution)
38     return N'
```

IDCBS

Since both-level of conflict search is just implementation of A* search, like A*, it is limited by the memory complexity. Although it can quickly find the optimal solution if memory is enough, once too many nodes are stored in the open list and use up all the memory, the efficiency of CBS will drop significantly. Therefore, we use the idea of iterative deepening A*(IDA*) to change this situation.

The implementation of IDCBS is to integrate IDA* algorithm into the high-level conflict tree. To be specific, depth-first search will be implemented on the constraint tree; after that, the nodes whose costs exceed the initial threshold will be pruned. The threshold

Algorithm 1: IDCBS with standard splitting

```
Result: An Optimal Solution or Exception (No Solution)
Compute the heuristics;
Initialized an open_list with stack;
Get the root node;
threshold = the cost of root node;
// Implement DFS on the conflict tree with standard splitting
solution = DFS(standard_splitting);
while solution == False do
    reset(); // Reset the open_list
    Push root into open_list;
    solution = DFS(standard_splitting);
    if threshold doesn't change and solution == False then
        raise Exception (No solution);
    end
end
return solution;
```

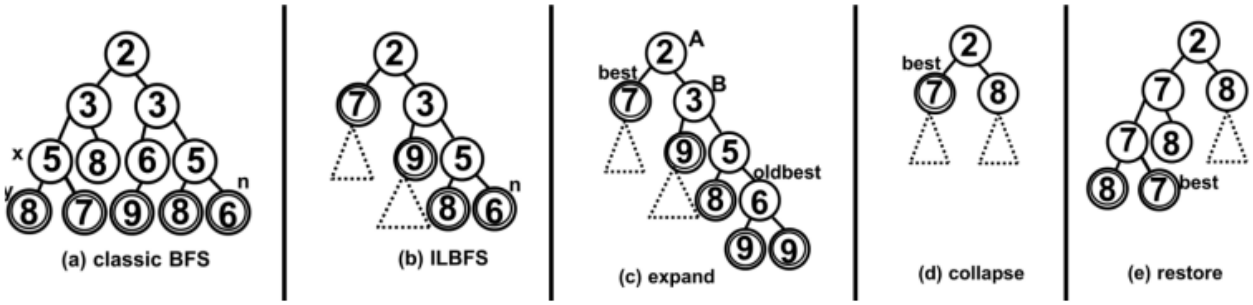
will be updated based on the smallest discarded cost in prior iteration. By using IDCBS, we can reduces the number of nodes in the open list over all time, which saves memory space. Although IDCBS will re-expand the expanded nodes for every iteration, it avoids the drop of efficiency caused by running out of memory.

ILCBS

The iterative linear conflict-based search (ILCBS) also reduces the number of nodes in the open list, at the same time, it re-expand fewer nodes than IDCBS does. It uses the idea of iterative linear best first search (ILBFS).

```
   Input: Root R
1  Insert R into OPEN and TREE
2  oldbest=NULL
3  while OPEN not empty do
4      best=extract_min(OPEN)
5      if goal(best) then
6          exit
7      if oldbest ≠ best.parent then
8          B ← sibling of oldbest that is ancestor of best
9          collapse(B)
10     if best.C=True then
11         best ← restore(best)
12     foreach child C of best do
13         Insert C to OPEN and TREE
14     oldbest ← best
```

As IDCBS, ILCBS is also improved by altering the higher-level of algorithm. In each iteration, it will check that if the parent of the node best from open list is the old best node. If it is, add best's children to the open list as normal and set it to be old best. On the other hand, if best's parent is not the old best, the algorithm will find the nod B that is the sibling of the best and also is the ancestor of old best. And them collapse the branch below B to B, and then if the new best is a node that is collapsed before, we need to restore the branch and set the leaf node with smallest f-value as best node.



The idea of ILBFS is to keep only the current best's sibling and its ancestors' sibling in open list. However, the process of collapse and restore is depend on the algorithm, and the restore is very hard to implement correctly. In this project, due to time constraints, we failed to use the idea of ILBFS to implement ILCBS successfully. Instead, we will introduce the methods we tried, the problems we encountered, and the possible solutions in the Result section.

4.Experimental Setup

In the experiments, we will use python 3.9, with Matplotlib installed for plotting. The environment for program to run is Windows with i7-9700k @ 3.60GHz and 32GB memory.

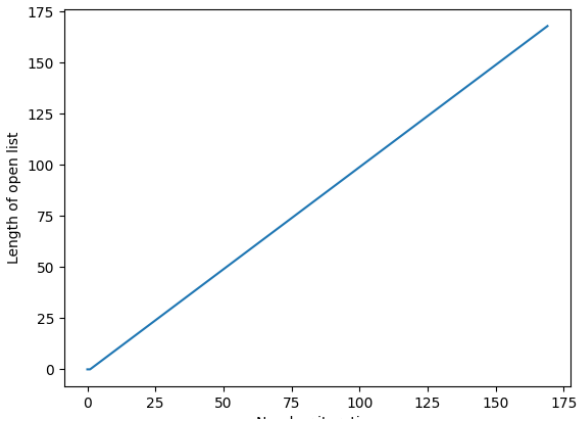
After the problem is specified and setup, the running time and memories used in the experiments will be counted and compared. During the experiments, we will try to build and use huge test cases to take up a lot of memory. To determine the efficiency difference between CBS and IDCBS, we will monitor how much time the algorithm takes to resolve each instances and the length change of open list during these processes.

5.Results

5.1 CBS and IDCBS

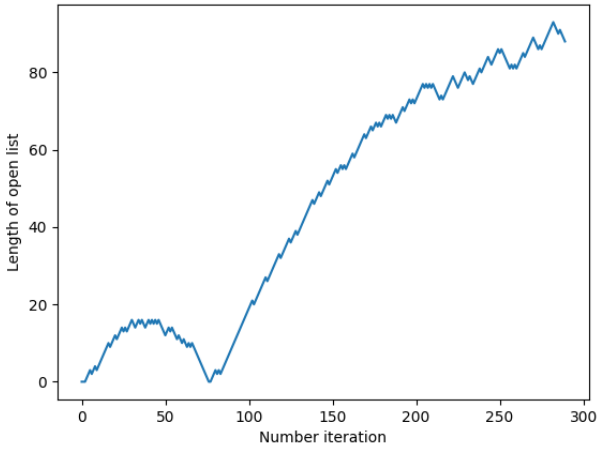
Due to time constraints, we used some test cases to test CBS and IDCBS, but we did not integrate their data. In this section, we will use the results of two test cases to illustrate the performance difference between CBS and IDCBS. We can't guarantee that what happens in all test cases is consistent with the conclusions we have drawn from these two test cases, but from our observations, the results of most of the test cases are consistent with the conclusions we have obtained from these two test cases

Case1: test_1.txt
CBS:
CPU time: 0.11
Sum of costs: 41
Generated nodes: 337
Average length of open list: 83.50588235294117



IDCBS:

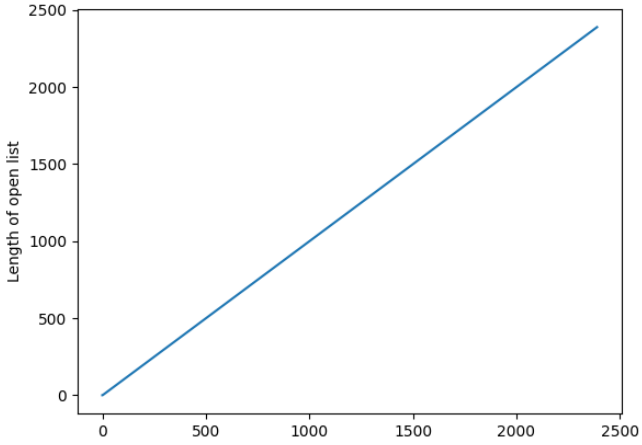
CPU time (s): 0.12
Sum of costs: 41
Generated nodes: 377
Average length of open list: 45.94137931034483



Case2: test_100.txt

CBS:

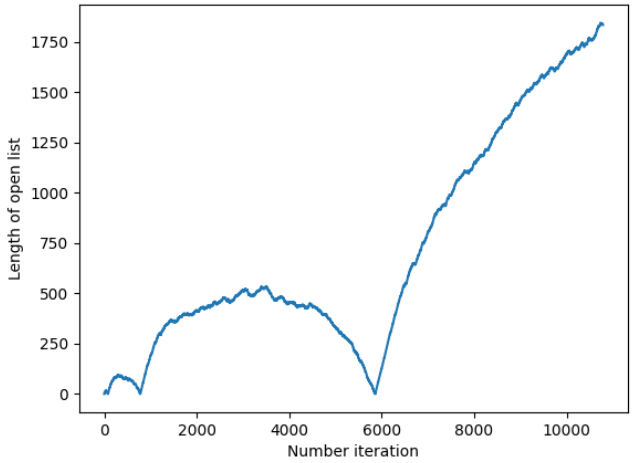
CPU time (s): 36.00
Sum of costs: 323
Generated nodes: 4779
Average length of open list: 1194.000418235048



IDCBS:

CPU time (s): 81.89

Sum of costs: 323
Generated nodes: 12621
Average length of open list: 715.2556781310838



Open list length:

By observing the graph, we can see that the length of the CBS's open list grows linearly as the number of iteration increase. Which make since, since CBS has a binary search tree, and after each expansion of node, we will remove one node and add two child nodes to the frontier. On the other hand, for both the maximum and average Open List length, IDCBS is smaller than CBS. Which showed that IDCBS does use less memory than CBS. The reason for this is, unlike CBS, IDCBS does not expand and store many deep nodes at the same time. Instead, the iterative deepening strategy of it makes it only store a relatively small number of nodes in the open list.

Time and node generation:

In both test cases, IDCBS generated more nodes and took more time than CBS. The reason for this phenomenon should be, in IDCBS, nodes are constantly re-expanded and generated in each iteration. As a result, IDCBS spends more time on these two test cases.

Side product:

When implementing CBS and IDCBS, we have tried using heuristic function to help our program run faster. However, we only implemented a heuristic function which violates the admissibility. The heuristic is easy to implement. Compare to real conflict graph heurist, our heuristic only finds the minimum vertex cover over all constraints (but not cardinal constraints only).

When using CBS with this naive heuristic to resolve test_100.txt, the CPU time is reduced from 36.00 seconds to 0.52 second. Because of the violation of admissibility, the cost of solution was increased from 323 to 324. In most of the test cases, with this heuristic, the efficiency in space and time has been significantly improved. Although in over 20% test cases, CBS with

this naïve heuristic failed to find the optimal solution, the costs of sub-optional solutions are usually just slightly higher than the optimal solution.

Although we failed to implement an admissible heuristic in this project, the improvement over time and space efficiency brought by this simple heuristic is far beyond our expectation. Which overturns our initial deduction that the a relatively small heuristic cannot bring great efficiency improvement.

5.3 ILCBS

Since we didn't implement ILCBS successfully, we will introduce the two different method we tried to implement collapse and restore, the problems we encountered and a potential solution we thought about but didn't have to try.

Method 1: Use independent collapse and restore macro functions.
In the beginning, we chose to try and write Collapse and Restore ourselves. As mentioned earlier, in each iteration, after goal test and before expanding of node, use collapse and restore macro.

- **Collapse**: With the node B that is the ancestor of old best and sibling of current best node, we find all of it's children in the open list. Mark B as a collapsed node and set its big F-value to the minimum f-value over its child nodes in the open list.
- **Restore**: To restore the collapsed branch, we use the depth first search to regenerate the branch and use the big F-value of the collapsed node as the threshold of DFS. To be more precise, for each node that popped out from DFS's open list, if it's f-value is smaller than the threshold, expand it and add its children to DFS's open list. If its f-value is greater than the threshold, do not expand the node. If the f-value is equal to the threshold, set the node as target node. After expand the branch, we also need to collapse the branch under the sibling of target's ancestors to keep the tree's shape stay properly. After the restore, we add all the collapsed nodes, target node and its siblings back to high-level CBS's open list.

Method 2: Low-level ILBFS's way
The other method we try is based on the low-level ILBFS's pseudo code from paper "The Collapse Macro in Best-First Search Algorithms and an Iterative Variant of RBFS" by Ariel Felner.

```
   Input: Root R
1  Insert R into OPEN and TREE
2  oldbest=NULL
3  while OPEN not empty do
4      best=extract_min(OPEN)
5      if goal(best) then
6          exit
7      while (oldbest ≠ best.parent) do
8          oldbest.val ← min(values of oldbest children)
9          Insert oldbest to OPEN
10         Delete all children of oldbest from OPEN and TREE
11         oldbest ← oldbest.parent
12     foreach child C of best do
13         F(C) ← f(C)
14         if F(best) > f(best) and F(best) > F(C) then
15             F(C) ← F(best)
16         Insert C to OPEN and TREE
17     oldbest ← best.
```

- **Collapse**: Felner suggest to use an iterative way to collapse the node and pass the old best's f-value layer by layer, until to the node B that is the sibling of current best node and also be then ancestor of old best. After collapse, B will store old best's f-value as it's big F value.
- **Restore**: To restore, we let the algorithm generate children of current best node as normal BFS search, but check if the F-value of best is greater than its f-value (which means current best node is a collapsed node) and if F-value of best is also greater than the F-value of its child. If both of the conditions are fulfilled, we pass the F-value of current best to this child. In next iteration, we will pick a node from these children with the same F-value as previous best node, which actually doing a DFS search with these nodes. Until all of the nodes have greater or equal F-value than the collapsed node's F-value, then we know the restore is done and ILCBS can keep run as usual.

Ideally, both methods should make ILCBS work. However, after restore, search tree became deformed and it appears we cannot find node B that is the ancestor of old best and also be the sibling of current best.

Potential solution: Instead of using F-value to store the information and rebuild the collapsed tree, we can use a compromised way. If we let the node B to store the constraints lists of its every child leaf node, that means we can easily reproduce the collapsed frontier. Although we were unable to complete this algorithm due to time constraints, we hypothesized that its performance would be between the ILCBS and the base CBS. Which means the time complicity is greater than basic CBS and smaller than ILCBS, and its memory complexity is greater than ILCBS and smaller than basic CBS. However, we're not sure if this algorithm will work with nested collapse situation.

Side product: a non-admissible heuristic that help CBS run 10 times faster to find a sub-optimal solution that the cost is usually slightly at most 5% greater than the optimal solution.

6.Conclusion

In this project, we test the performance between CBS and IDCBS with different kind of test cases, and tried to implement ILCBS. For the previous two algorithms, we find depict IDCBS used less memory, the time it cost to solve small test cases is actually longer than CBS. Which means, if the memory space sufficient, CBS would still be our best choice of MAPF problems. However, we can see from the change in the length of the open list that IDCBS is much more efficient in handling large test cases than CBS. When processing large test cases, all memory space may be quickly consumed by CBS and additional data will be stored in the storage. Once this happen, the disk's relatively slow read and write speeds will slow down the efficiency of the CBS. And IDCBS, by taking less memory space, can avoid this situation and maintain its efficiency.

Ideally, ILCBS would be even better than IDCBS. It will store a small number of nodes in the open list as IDCBS does, but ILCBS will do fewer repeated expansion over nodes than IDCBS does. However, to successfully implement ILCBS, we must ensure that we find a stable way to collapse and restore branch.

7.Reference

[1] Boyarski, E, Felner, A, Harabor, D, Stuckey, PJ, Cohen, L, Li, J & Koenig, S 2020, Iterative-deepening Conflict-Based Search. in C Bessiere (ed.), *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. IJCAI International Joint Conference on Artificial Intelligence, vol. 2021-January, Association for the Advancement of Artificial Intelligence (AAAI), Marina del Rey CA USA, pp. 4084-4090, International Joint Conference on Artificial Intelligence-Pacific Rim International Conference on Artificial Intelligence 2020, Yokohama, Japan, 7/01/21. https://doi.org/10.24963/ijcai.2020/565

[2] Sharon, Guni & Stern, Roni & Felner, Ariel & Sturtevant, Nathan. (2015). Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence. 219. 40-66. 10.1016/j.artint.2014.11.006.
[3] Hönig, W., Kiesel, S., Tinka, A., Durham, J.W., & Ayanian, N. (2018). Conflict-Based Search with Optimal Task Assignment. *AAMAS*.