Improved Conflict-based Search (ICBS) with Disjoint Splitting

Gloria Yoon
gya13@sfu.ca

Yizhong Wang
yizhongw@sfu.ca

Ash Peng
rpa47@sfu.ca

1 Introduction

Conflict-based Search (CBS) is an optimal solver for the canonical robotics task of multi-agent path finding (MAPF), and research on it has been flourishing in recent years. This project is an experimental implementation and comparative study on combining the disjoint splitting approach for conflict resolution in CBS with Improved CBS (ICBS), which incorporates four extensions made to the standard CBS algorithm. The four extensions are as follows: Prioritizing Conflicts, Bypassing Conflicts, Meta-agent, and Merge & Restart.

The basis of this project is a solid foundation of previous knowledge on MAPF, CBS, and disjoint splitting. In a MAPF problem, the input includes an undirected graph $G = (V, E)$, a set of heuristics for each vertex of G, and two sets of vertices $s : [1, ..., k] \rightarrow V$ and $t : [1, ..., k] \rightarrow V$, that represent the start and goal locations for each of the k agents; the output is a set of sequences of actions $\pi = (a_1, ..., a_n)$, one sequence for each agent i, that maps each time step to an action of either staying in its current location, or move one step along an edge to an adjacent vertex in the graph, till it reaches its goal location without in the meantime colliding with other agents (forming vertex or edge conflicts) or the environment. [1] As a standard two-level MAPF solver, CBS focuses on finding an optimal solution by resolving agent conflicts through splitting them into constraints, each of which prohibits an agent from occupying a certain vertex or traveling from a vertex to another at a certain time step. CBS performs a best-first search on a binary constraint tree (CT), where ties are broken in favor of nodes whose solutions have fewer conflicts, until it finds a node in the CT where there is no conflict. CBS then returns the solution.

It is worth noting that at each level of the CT, standard CBS randomly chooses a conflict to split, which may result in poor choices that significantly impact algorithm performance. ICBS was introduced to improve these choices by four extensions to standard CBS, and it has been shown to introduce "a significant speedup"[2] when all four are combined.

On the other hand, disjoint splitting is a modification to remedy the inefficient splitting mechanism of standard CBS. Instead of separating each conflict into two negative constraints, CBS with disjoint splitting splits each conflict into a negative constraint and a positive constraint. A positive constraint (a_i, v, t) forces a_i to be at v at time step t, which implicitly prohibits any other agent $a_j (j \neq i)$ from being at v at time step t, thereby eliminating duplicate plans in the two child nodes generated. Disjoint splitting is observed to be at least as good as standard splitting and can speed up the CBS search in many cases.

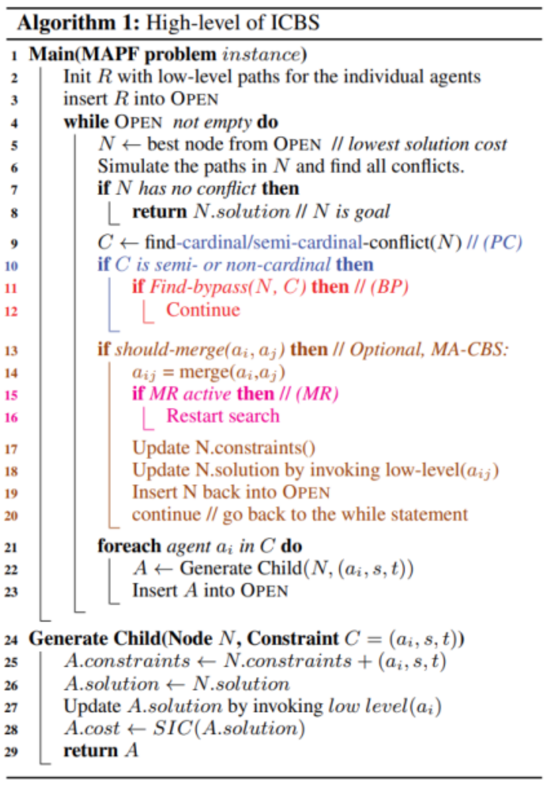
Research has shown that both ICBS and CBS with disjoint splitting can empirically induce better performance than the standard CBS baseline. However, up to the time we write this report, there is yet to be any research done on ICBS with disjoint splitting. Since the four extensions in ICBS only modify the process of conflict selection while disjoint splitting only modifies the process of splitting the chosen conflict, the two approaches can be at least combined orthogonally. Furthermore, while implementing ICBS, we also noticed that ICBS makes use of conflict splitting numerous times outside the main CBS loop, and it is intuitive that disjoint splitting may improve the efficiency of such procedures.

Through our implementation and analysis of ICBS with disjoint splitting, we hope to offer insights on the node-generation and -expansion mechanism of the standard CBS algorithm, how well the four extensions of ICBS improves CBS performance, and how much disjoint splitting can further enhance ICBS performance for MAPF problems.

In the context of this report, a better performance for CBS refers to a decreased number of generated and/or expanded nodes, hence resulting in a smaller-sized CT which allows for a fewer number of iterations before CBS finds a solution.

2 Implementation

At the highest level, we attempt to implement the following ICBS algorithm, using disjoint splitting (as opposed to standard splitting from the standard CBS algorithm) whenever possible.



```
Algorithm 1: High-level of ICBS
Main(MAPF problem $instance$)
    Init $R$ with low-level paths for the individual agents
    insert $R$ into OPEN
    while OPEN not empty do
        $N \leftarrow$ best node from OPEN // lowest solution cost
        Simulate the paths in $N$ and find all conflicts.
        if $N$ has no conflict then
            return $N.solution$ // $N$ is goal
        $C \leftarrow$ find-cardinal/semi-cardinal-conflict($N$) // (PC)
        if $C$ is semi- or non-cardinal then
            if Find-bypass($N$, $C$) then // (BP)
                Continue
        if should-merge($a_i, a_j$) then // Optional, MA-CBS:
            $a_{ij}$ = merge($a_i$,$a_j$)
            if MR active then // (MR)
                Restart search
            Update N.constraints()
            Update N.solution by invoking low-level($a_{ij}$)
            Insert N back into OPEN
            continue // go back to the while statement
        foreach agent $a_i$ in $C$ do
            $A \leftarrow$ Generate Child($N, (a_i, s, t)$)
            Insert $A$ into OPEN
Generate Child(Node $N$, Constraint $C = (a_i, s, t)$)
    $A.constraints \leftarrow N.constraints + (a_i, s, t)$
    $A.solution \leftarrow N.solution$
    Update $A.solution$ by invoking $low\ level(a_i)$
    $A.cost \leftarrow SIC(A.solution)$
    return $A$
```

Figure 1: ICBS pseudocode[1] we are trying to implement

We divide the four extensions into two pairs, as Prioritizing and Bypassing Conflicts work closely together, and same for Meta-agent plus Merge & Restart. In sub-section 2.1 and 2.2, we will give a more detailed analysis of the two pairs.

[1]Source: *Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges*. Available: https://people.engr.tamu.edu/guni/Papers/SOCS17-MAPF.pdf

2.1 Prioritizing Conflicts, and Bypassing Conflicts

The first and the second extensions to CBS($Algorithm 1$), namely prioritizing conflicts and bypassing conflicts, are tightly coupled so we discuss them as one single algorithm.

Conceptually, a conflict C is classified as one of the three categories: *cardinal*, *semi-cardinal*, or *non-cardinal*. C is cardinal for a CT node N if adding any of the two constraints derived from C to N and invoking the low-level A* search on the constrained agent, increases $N.cost$, which is the sum of costs for the two paths travelled by the corresponding agents. C is defined as semi-cardinal if adding one of the two constraints derived from C increases $N.cost$, but adding the other keeps $N.cost$ unchanged. Finally, C is non-cardinal if adding either of the two constraints derived from C and generating the new paths with the lower-level search does not lead to an increase of $N.cost$. [3]

As a brief description of the algorithm, we first iterate over all conflicts in an attempt to find a *cardinal* conflict to split; we stop at the first cardinal conflict we can find. If the search is unsuccessful, we try to find a *semi-cardinal* conflict with the same method we used for *cardinal*; if both searches fail, we resort to the original implementation of standard CBS, which is choosing a conflict randomly (in this case it is out of all *non-cardinal* conflicts, as we may now ensure all conflicts are *non-cardinal*). Then we use disjoint splitting to split the chosen conflict into two constraints and proceed with the subsequent steps in CBS.

Picking a cardinal conflict can improve CBS performance by decreasing the size of the CT, since doing so generates two child nodes with both costs larger than C; and if another node M in *OPEN* exists with cost C, M will be chosen for expansion next without further developing nodes below the current node. As most of the increase in cost while traveling down the CT is due to the expansion of cardinal conflicts, prioritizing conflicts marked as cardinal at earlier stages would decrease the overall number of nodes generated and expanded, resulting in an improved performance in the entire search. If there is no cardinal conflict, picking a semi-cardinal conflict would achieve a similar (while less pronounced) improvement.

Furthermore, if we have a semi- or non-cardinal conflict, as semi- or non-cardinal conflicts guarantee that there will be a path for at least one of the involved agents that avoids this conflict and has the same cost as the path the current agent is following (i.e. the path that causes the current conflict we are investigating), we can try to *bypass* it by essentially doing a one-step look ahead. [4] More specifically, we can check the child nodes of the current node in the CT to find a child with the same cost and potentially a fewer number of agent conflicts, so we can insert the child into *OPEN* to *bypass* the current conflict instead of going through the standard child node generation process which is generating one child node for each constraint that comes from disjoint splitting. If a *bypass* solution is found by the lower-level single-agent search algorithm, we no longer have to split on the current node; this also effectively decreases the size of the CT. For more details of CBS with these two extensions please refer to Algorithm 1.

Algorithm 1 High-level CBS with Prioritizing & Bypassing Conflicts

```
function CBS-SOLVER(MAPF $instance$)
    initialize root node $R$ for each agent
    insert $R$ into OPEN
    while OPEN is not empty do
        $p \leftarrow$ best node in OPEN
        if no conflicts in $p$ then
            return p.paths as solution
        end if
        if $\exists$ cardinal conflict in p.conflicts then
            conf. = first cardinal conflict
        else if $\exists$ semi-cardinal conflict in p.conflicts then
            conf. = first semi-cardinal conflict
        else
            conf. = arbitrary conflict
        end if
        $constraints \leftarrow$ Disjoint-Split($conf.$)
        for $c$ in $constraints$ do
            $q \leftarrow$ Get-Child($c+$ p.constraints)
            if $conf.$ is semi- or non-cardinal then
                $bp \leftarrow$ Find-Bypass($p$)
                if $bp$ is not $None$ then
                    insert $bp$ into OPEN
                    break
                end if
            end if
            insert $q$ into OPEN
        end for
    end while
end function
```

To implement the low-level algorithm of prioritizing conflicts (i.e. detecting cardinality of conflict) and *Find-Bypass*, we generate two child nodes from the two resulting constraints, and for each node, we use the A* algorithm to plan a new path for each conflicting agent, and then we compare the costs of the child nodes with the current node for the two extensions.

It is worth noting that standard splitting is used for detecting cardinality in the low-level instead of disjoint splitting, because by the definition of cardinality, finding the category of a certain collision requires creating two negative constraints and generating two new sets of paths to obtain the new costs.

2.2 Meta-agent, and Merge & Restart

In Meta-agent CBS (MA-CBS) ($Algorithm 2$ at the end of the report), frequently conflicting agents are merged into a meta-agent. Two agents a_i and a_i are merged into a meta-agent a_ij if the number of conflicts between the agents exceeds B for MA-CBS(B). (Sharon et al. 2015) CBS and MA-CBS as high-level search algorithms may use A* as its low-level solver. In CBS, a decoupled version of A* is used in which A* finds optimal paths for each agent individually. In MA-CBS, meta-agents are treated as a single agent and the path's of the meta agents are solved by the coupled A*. As an extension of MA-CBS, ICBS also may use decoupled A* as the lower-level solver.

To prevent repeated merging of agents in the CT, when a merge occurs at node N, the current CT is discarded and is restarted with node N as the root node. This procedure is called merge and restart (MR). (Sharon et al. 2015)

The coupled A* algorithm constitutes the lower-level implementation and thus the foundation for MA-CBS and has accounted for the most work in the whole project. For the pseudo code of coupled A* please refer to Algorithm 2 (on the last page on the report).

3 Methodology

In order to investigate the amount of speedup obtained from ICBS with disjoint splitting, we compare the following four implementations and evaluate them based on the number of nodes generated and the number of nodes expanded in the main CBS loop, which is equivalent to number of nodes generated/expanded in the CT.

For the algorithms we have implemented, we are testing all six combinations of three CBS variants with two methods of splitting conflicts. Specifically, the CBS variants include standard CBS, CBS with prioritizing conflicts and bypassing conflicts, and ICBS, which is CBS with prioritizing conflicts, bypassing conflicts, meta-agents, and merge & restart. Each of the three variants are paired with two different ways of splitting conflicts (namely, standard splitting and disjoint splitting) on our MAPF instances. The reason for this arrangement is that first we would like to compare the two pairs of extensions on CBS; secondly, we would like to separately inspect the effects of ICBS and those of disjoint splitting on standard CBS; and finally, how much improvement can be obtained when everything is combined.

For test instances, we first use the batch instances taken from the individual project to ensure the optimality of our implementations (i.e. resulting in identical sums of costs as in our individual project, as they had been provided beforehand). Secondly, we will test our implementations on seven custom instances we designed, which are designed with variations in the ratios of map size versus number of agents, how "maze-like" the map is (i.e., how much open space there is that allows the agents to travel freely), and variations of bottlenecks (i.e. tight space that connects two open spaces). The instances include, as examples, a primarily open-space map with a small number of agents, a primarily tight-spaced map (i.e. a maze) with a large number of agents, and other variants of this kind.

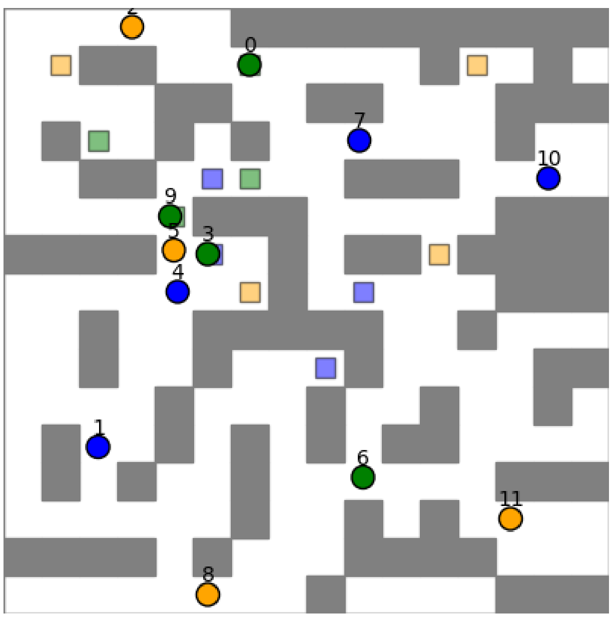


Figure 2: Customized instance of 15 agents (test$_5$3)

Figure 2 shows a customised map with 12 agents, which joins together a moderate amount of open and tight spaces with a fair number of bottlenecks. We use it as our exemplary instance to compare performance in the *Results* section.

Finally, for the evaluation metric, we use the number of nodes generated/expanded because the size of the CT is known to be a good indicator of CBS performance as explained in the previous sections. It is also used as the main evaluation metric for a large body of research in this field as for our individual project.

4 Experimental Setup

The following table describes the environment we use for developing the aforementioned algorithms, and testing the algorithms on all of our MAPF instances:

Environment	**Specification**
Language	Python
Version	3.9.5
OS	Ubuntu 20.0.4
Processor	Intel i7-9700k 3.6gHz
Memory Avail.	16GB

Table 1: Our running and testing environment

5 Results

Firstly, through testing with the 50 batch instances, we have found that both disjoint splitting and the four extensions to CBS have tremendously decreased both the number of nodes expanded and number of nodes generated, as is shown in Figure 3 and 4 (for the line-plot, results have been sorted for a clearer idea of the degree of improvement across different instances).

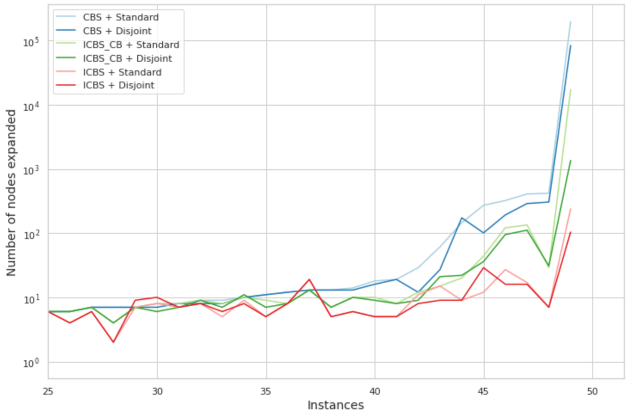


Figure 3: Comparison of different combinations of CBS variants with splitting methods, on 50 test cases

Secondly, through testing with the five customized instances we designed with a variety of map size, agent number, proportion of open space, and number of bottlenecks, we have confirmed the same trend.

CBS Variant	Method of Splitting	# of Nodes Generated	# of Nodes Expanded
Standard	Standard	2787	1394
Standard	Disjoint	2761	1336
ICBS	Standard	94	51
ICBS	Disjoint	89	50

Table 2: Performance of four different CBS implementations on MAPF instance “test_53.txt”

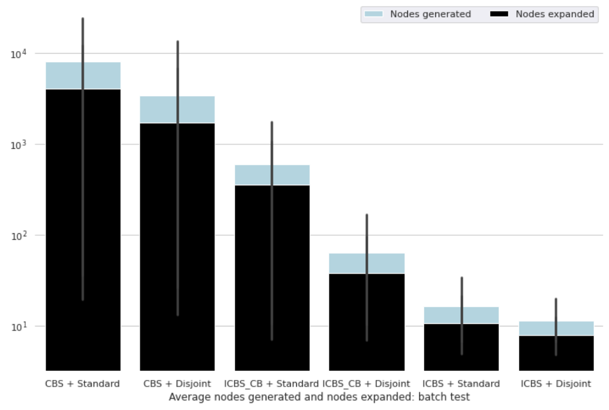


Figure 4: The number of nodes generated and expanded across different combinations, with logarithmic y-axis

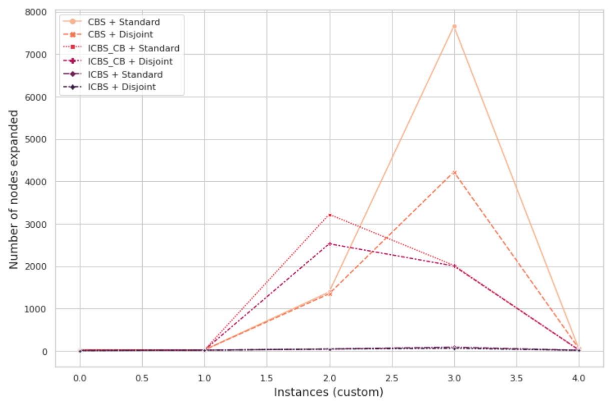


Figure 5: The number of nodes generated and expanded in our customized test instances

Particularly, for our exemplary instance - “test_53”, which is a maze-like map with a moderate ratio of open to tight spaces and a relatively large number of bottleneck, Table 2 shows the exact number of nodes expanded and generated.

For this particular instance, we assume that the stark improvement of ICBS over standard CBS is primarily resulted from the Prioritizing Conflicts extension, which splits cardinal conflicts earlier than the rest of the conflicts, effectively solving the bottlenecks first in the map forming a shallow CT and a quicker path to the solution.

Also, by comparing the two pairs of extensions (namely prioritizing conflicts bypassing conflicts, versus meta-agent plus merge and restart) while observing the characteristics of the instances we have, we are also able to discover interesting details about what pair of extensions work better for what kinds of instances.

Specifically, we find that the performance of CBS with prioritizing conflicts plus bypassing conflicts, which we have named CBS_CB (for *Cardinal* and *Bypass*) is positively correlated with the proportion of tight spaces and the number of bottlenecks in the map, whereas the performance of ICBS (with meta-agents included) achieves the most improvement in open spaces or wide passages where agents have a lesser chance of collision. This reflects our expectations because A* works best in open spaces, and A* has been the low-level algorithm that constitutes the foundation of meta-agent CBS.

6 Conclusion

From the analyses of the six different combinations of the CBS variants with standard/disjoint splitting, we found that the four extensions included in ICBS improves CBS performance tremendously, namely in the number of nodes expanded and number of nodes generated (i.e. size of the constraint tree), and disjoint splitting further enhances the improvements brought about by ICBS, though disjoint splitting alone brings less of an improvement to any CBS variant. Also, we have found that maps with a greater proportion of open spaces see the most improvements from ICBS (where meta-agents are included), whereas tighter spaces see the most improvements from the resolution of bottlenecks (i.e. areas most prone to cardinal conflicts), so the CBS_CB variant, which is CBS with prioritizing and bypassing conflicts.

7 Future Work

While trying to simulate real world MAPF problems (e.g. “test-56.txt”), we found situations that CBS cannot solve in a reasonable amount of time but our ICBS implementation can. Future research can focus on more real-world problems such as traveling through a tight passage into a parking lot, forking paths on railways, and analyze the difference of performance for different algorithms on these problems. Also, future research can focus on comparing and contrasting the difference of improvement of the bypassing conflicts algorithm versus the meta-agent algorithm without bypassing conflicts, since meta-agent builds upon bypassing conflicts.

References

[1] Felner, A., Stern, R., Shimony, S.E., Boyarski, E., Goldenberg, M., Sharon, G., Sturtevant, N.R., Wagner, G., & Surynek, P., *Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges*, May 2017. Available: https://arxiv.org/pdf/1810.04805v2.pdf

[2] Boyarski, E., Felner, A., Stern, R., Sharon, G., Betzalel, O., Tolpin, D., & Shimony, S.E., *ICBS: The Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding*, 2015. Available: https://www.ijcai.org/Proceedings/15/Papers/110.pdf

[3] Boyarski, E., Felner, A., Sharon, G., & Stern, R., *Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding*, 2015. Available: https://ojs.aaai.org/index.php/ICAPS/article/view/13725/13574

[4] Sharon, G., Stern, R., Felner, A., & Sturtevant, N.R., *Conflict-based search for optimal multi-agent pathfinding*, 2015. Available: https://www.aaai.org/ocs/index.php/AAAI/AAAI12/paper/viewFile/5062/5239

Algorithm 2 Low-level coupled A*

```
function COUPLED A*(Map $M$, agents $A$, constraints $C$)
    initialize *OPEN* and *CLOSED*
    $h - value \leftarrow 0$
    $constraint - table \leftarrow$ constraints for each agent in $A$
    for agent in $A$ do
        initialize root node
        push root to *OPEN*
        *CLOSED*(loc,timestep)←root
        while *OPEN* not empty do
            $curr \leftarrow$ pop node from *OPEN*
            for  agent $a$ in meta-agents do
                if $a$ in it's goal then
                    $a.reach - goal \leftarrow$ True
                end if
            end for
            if all agents reaches its goal then
                return paths
            end if
            for each agent that is searching do
                for each direction do
                    if curr[loc] is goal or invalid move then
                        Continue
                    end if
                    check external constraints
                    if constrained then
                        Continue
                    end if
                    Compute h-value for each agent
                    generate $Child$ node
                    if $Child$ in *CLOSED* then
                        if $Child$ is better than $curr$ then
                            insert $Child$ to *CLOSED*
                            push into *OPEN*
                        end if
                    else
                        insert $Child$ to *CLOSED*
                        push into *OPEN*
                    end if
                end for
            end for
        end while
        print "NO SOLUTION"
```