Test and Compare the Performance of CBS With Different Heuristics

Yufei Wang	Xinyue Ma	Qianhe Niu
301338190	301297142	301326166

1 Introduction

This project will focus on comparing the performances of CBS with three heuristics CG, DG, WDG and figuring out which heuristic is bad under some situations. The following definitions are based on Li et al.'s paper [1].

1.1 MAPF

The Multi-Agent Path Finding (MAPF) problem includes an undirected unweighted graph $G = (V, E)$, where has a set of k agents $a_1, ..., a_k$ and each agent a_i has a start vertex $s_i \in V$and a goal vertex $g_i \in V$. Time is recorded using discrete timesteps. From timestep t to timestep $t + 1$, each agent can move to one of the adjacent vertices or wait at the current vertex. Both actions will increase timestep by 1. A path of a_i is a sequence of actions including moves and wait that lead a_i from s_i to g_i. A vertex conflict is that a_i and a_j are in the same vertex v at timestep t, represented by $< a_i, a_j, v, t >$. An edge conflict is that a_i and a_j traverse the same edge (u, v) in opposite directions from timestep t to timestep $t + 1$, represented by $< a_i, a_j, u, v, t >$. The solution to this problem is to find a set of conflict-free paths that lead every agent a_i from its start vertex s_i to its goal vertex g_i while minimizing the sum of the costs of these paths.

1.2 CBS

Conflict-Based Search (CBS) has two levels. The high level of CBS searches the binary constraint tree (CT) using the best-first search according to the costs of the CT nodes. Each CT node N contains:

- A set of constraints $N.constraints$, where a constraint is either a vertex constraint $< a_i, v, t >$ that prohibits a_i from being at vertex v at time step t or an edge constraint $< a_i, u, v, t >$ that prohibits a_i from moving from vertex u to vertex v between timesteps t and $t + 1$,
- A solution $N.solution$ consists of a set of k cost-minimal paths for k agents, which satisfies $N.constraints$, and
- A cost $N.cost$ is the sum of the costs of the paths in $N.solution$

1.3 CBS with heuristics CG, DG, WDG

CBS with heuristics is focusing on reducing the number of CT node being expanded using heuristic values. The high level of CBS searches the binary constrain tree(CT) using the best-first search according to the sum of the costs of the CT nodes and a heuristic value.

1.4 CG

A cardinal conflict is a conflict that when uses CBS to split CT node N, the cost of both children of N is larger than $N.cost$. A conflict graph $G_{CG} = (V', E')$, where V' represents agents and E' represents cardinal conflicts in $N.solution$. The size of a minimum vertex cover (MVC) is an admissible $h - value$ for N.

1.5 DG

Two agents are dependent iff all their cost-minimal paths that satisfy N.constraints have conflicts. Two agents are also dependent if they have cardinal conflicts. A pairwise dependency graph $G_D = (V_D, E_D)$ for each CT node, where V_D represents agents and E_D represents two connected agents are dependent. The size of the MVC of G_D is an admissible heuristic for N.

1.6 WDG

A weighted pairwise dependency graph $G_{WD} = (V_D, E_D, W_D)$ for N is a G_D with weights for each edge. V_D and E_D are the same vertices and edges as those in G_D. For a CT node N and two agents a_i and a_j, the weight for the edge between a_i and a_j in G_{WD} is the difference between the minimum sum of the costs of their conflict-free paths that satisfy $N.constraints$ and the sum of the costs of their paths in $N.solution$, represented by $\triangle_{ij}$. The edge-weighted minimum vertex cover (EWMVC) is an assignment of non-negative integers $x_1, ..., x_k$, one for each vertex, which minimizes the sum of the x_i subject to the constraints that $x_i + x_j \geq \triangle_{ij}$ for all $(v_i, v_j) \in E_D$. The EWMVC is an admissible heuristic for N.

2 Implementation

2.1 CBS

Algorithm 1 shows the pseudo code of the high-level search of CBS. We use standard splitting for CBS and use a priority queue for storing CT nodes, which will pop the CT node with the smallest $f\_value = cost + h\_value$ first. We also store MDDs for agents that satisfying all the constraints in the CT node. In this way, when we expand the children of CT node N, what we need to update MDD according to the new constraint. More details for updating MDD and constructing MDD will be discussed in the 2.6.

2.1.1 Improvements

Multi-valued Decision Diagram (MDD) is an important component for computing h-value. MDD with new depth d needs to be constructed when the length of the path for agent a_i increases. And the MDD with depth d could be constructed more than one time. For example, a CT node N with $N.costC$ has a path with length d for agent a_i. In both children of N, the length of the path for agent a_i increases to d'. It is possible that we expand both children of N in the future, and the MDD with depth d' for agent a_i needs to be constructed two times. In order to reduce the time for constructing MDD with the same depth d more than once, we store the MDD with new depth d' for agent a_i, without any constraints, if the MDD with new depth d' for agent a_i appears for the first time. And if the depth increases and the MDD with the new depth has already been constructed, then the rest is to update the MDD according to the constraints in the CT node.

Algorithm 1: High-level search of CBS

```
Input  : Representation of the environment, start cells, and goal cells
Output: optimal collision-free solution
R.constraints ← ∅;
R.paths ← find independent paths for all agents using A* algorithm;
R.collisions ← detect collisions in the paths;
R.cost ← the sum of the costs of all the paths;
if use any heuristic then
    MDD ← construct MDDs for each agent;
    R.MDD ← MDD;
    MDD_all = [];
    for each agent i do
        mdd = {};
        mdd[depth of MDD[i]] = MDD[i];
        Add mdd to MDD_all ;
    end
end
Insert R into OPEN;
while OPEN is not empty do
    P ←node from OPEN with the smallest f-value = P.cost + h_value;
    if P.collisions = ∅ then
        return P.paths;
    end
    collision ← one collision in P.collisions;
    constraints ← using standard splitting;
    for constraint in constraints do
        Q ← new node;
        Q.constraints ← P.constraints ∪ constraint;
        Q.paths ← P.paths;
        Q.MDD ← P.MDD;
        a_i ← the agent in constraint;
        path ← using A* algorithm for new path satisfying Q.constraints;
        if path is not empty then
            if use any heuristic then
                if length d of path increases then
                    if MDD with d for agent a_i does not exist in MDD_all then
                        Construct MDD with depth d for a_i and store it in MDD_all;
                    end
                    Replace the MDD for a_i in Q with the MDD for a_i in MDD_all;
                end
                Update MDD for a_i according to Q.constraints;
            end
            Replace the path of agent a_i in Q.paths by path;
            Q.collisions ← detect collisions in the paths;
            Q.cost ← the sum of the costs of all the paths;
            Insert Q into OPEN
        end
    end
end
return'No solution'
```

2.2 CG

Algorithm 2: Compute CG heuristic

Input : MDDs for all agents
Output: CG h-value
`// Construct conflict graph according to cardinal conflict`
G ← empty undirected graph;
for *each pair of agents* **do**
  if *there exists a cardinal conflict* **then**
    Add two nodes corresponding to the pair of agents to G;
    Add edge between the two nodes to G;
  end
end
h_value ← the size of MVC of G;
return h_value

Algorithm 2 shows the pseudo code of computing CG heuristic. Two agents have a cardinal conflict iff the contested vertex (edge) is the only vertex (or edge) at level t of the MDDs for both agents. First, we check whether the number of the possible moves for one at timestep t is 1. If so, then we check another agent. If both agents have one possible move at the same timestep t, then we check whether the location is the same. For edge, we check whether there are two consecutive timesteps t and $t+1$ with only one possible move for each timestep for two agents. If so, we check whether it is an edge conflict.

2.3 DG

Algorithm 3: Compute DG heuristic

Input : MDDs for all agents
Output: DG h-value
`// Construct dependency graph`
G ← empty undirected graph;
for *each pair of agents* **do**
  joint_MDD ← merge MDDs for two agents;
  `// Empty means there is no pair (goal[`a_i`], goal[`A_j`]) existing the max depth of joint MDD`
  if *the joint MDD is empty* **then**
    Add two nodes corresponding to the pair of agents to G;
    Add edge between the two nodes to G;
  end
end
h_value ← the size of MVC of G;
return h_value

Algorithm 3 shows the pseudo code of computing DG heuristic. Whether the joint MDD is empty depends on whether there exists a pair of goal locations at the max depth of the joint MDD. During merging

two MDDs, we return the joint MDD when there is no pair exist at timestep t no matter whether timestep t is the max depth or not. This can save some time.

2.4 WDG

Algorithm 4: Compute WDG heuristic

Input : MDDs for all agents, paths in CT node N, constraints in CT node N
Output: WDG h-value
`// Construct dependency graph`
G ← empty undirected graph;
for *each pair of agents* **do**
  joint_MDD ← merge MDDs for two agents;
  `// Empty means there is no pair (goal[`a_i`], goal[`A_j`]) existing the max depth of joint MDD`
  if *the joint MDD is empty* **then**
    Add two nodes corresponding to the pair of agents to G;
    Add edge between the two nodes to G;
  end
end
`// Compute weight for each edge in G`
for *each edge e in G* **do**
  cost ← Use CBS to compute the minimum sum of the costs of conflict-free paths for agents a_i and a_j connected by edge e satisfying N.constraints;
  weight ← cost - the sum of the costs of the N.paths for a_i and a_j;
  Add weight to edge e in G
end
h_value ← use branch and bound to calculate the size of EWMVC of G;
return h_value

Algorithm 4 shows the pseudo code of computing WDG heuristic. For CT node N, We compute the minimum sum of costs of the paths of two dependent agents satisfying N.constraints using original CBS, since we already have this CBS algorithm and CBS is optimal and complete. If we use heuristics for computing weights of edges, we will spend too much on constructing MDD again and again.

2.5 MVC

Algorithm 5 shows the pseudo code of computing MVC. We implement the algorithm for computing minimum vertex cover indicated by Wang et al. [2]. The algorithm is based on the branch-and-bound algorithm. The original algorithm gets good results in an efficient way by using a tight lower bound. The algorithm uses clique-based lower bound ($ClqLB$), degree-based lower bound ($DegLB$) and max-SAT-based lower bound ($SatLB$). We implemented $ClqLB$ and $DegLB$. For $SatLB$, it is difficult to implement since, in order to compute the $SatLB$, we need to convert MVC problem into a $MaxSAT$ problem. And in this project, we want to focus more on the performances of CBS with different heuristics, so we ignore the $SatLB$. Although we ignore $SatLB$, this is the same for every conflict graph, we can still compare the performances of CBS with different heuristics under the same situation.

Algorithm 5: The EMVC algorithm (G, UB, C)

Input : a graph G = (V, E), an upper bound UB = |V|, a growing partial vertex cover C = $\varnothing$
Output: the size of the minimum vertex cover S_{min} of G
if $|C|$ + *max(DegLB(G), ClqLB(G))*$\geq$ *UB* **then**
  return UB;
end
if $G = \varnothing$ **then**
  return |V|;
end
Select a vertex v from V with the maximum degree;
$|C_1| \leftarrow EMVC(G \setminus N^*(v), UB, C \cup N(v))$;
$|C_2| \leftarrow EMVC(G \setminus v, min(UB, |C_1|), C \cup v)$;
return $min(|C_1|, |C_2|)$;

2.6 MDD

2.6.1 Construct MDD for agent

Algorithm 6 shows the pseudo code of constructing MDD for a single agent. We use a directed graph to be the data structure for MDD. In constructing a dependency graph, we need to merge two MDDs, which needs to access the successors of nodes. So, a directed graph is a good choice for MDD.

Improvements

The basic method for constructing MDD is Breadth-First Search(BFS). However, if the depth for MDD is large, then it will take much more time to do the search since in MAPF there are five possible moves for each location. If the map does not have any obstacles, BFS will also add possible locations that the agent moves away from the current location. Hence, we use a heuristic to control the direction of the agent's movement. If the $f\_value(= g\_value + h\_value)$ of the child location is larger than the depth of MDD, then we omit the child. Since an admissible h_value is always smaller than or equal to the actual cost, if the f_value is larger than the depth of MDD, then the path through that child location at that timestep is impossible to be part of the MDD.

2.6.2 Update MDD

Algorithm 7 shows the pseudo code of updating MDD for a single agent. For updating MDD, we only need to delete nodes and edges according to edges. For example, if a constraint prohibits the agent from being at vertex v at timestep t, then we need to delete the edge from nodes on depth $t-1$ to node v. And after deleting all edges that violate constraints, we need to delete internal nodes that do not have parents or children, since there are no paths that are able to arrive at the goal node at timestep t through these internal nodes.

Algorithm 6: Construct MDD for agent a

Input : start and goal locations for agent a, h-values, depth d
Output: Directed graph MDD for agent a
MDD ← empty directed graph;
OPEN ← [];
root ← {};
root.loc ← a.start;
root.g_val ← 0;
root.h_val ← h-value for start location;
root.timestep ← 0 Insert the start locations to OPEN list;
while *OPEN is not empty* **do**
    curr ← node from OPEN ;
    if *curr.timestep = d* **then**
        if *curr is a goal node* **then**
            Add nodes and edges that the path from start location to curr.loc goes through to G;
        end
        continue
    end
    for *each possible move from curr.loc* **do**
        child.loc ← new location;
        child.g_val ← curr.g_val + 1;
        child.h_val ← h-value for child.loc;
        child.parent ← curr;
        child.timestep ← curr.timestep + 1;
        if *child.g_val + child.h_val >d* **then**
            continue
        end
        Add child to OPEN
    end
end
return h_value

Algorithm 7: Update MDD for agent a

```
Input  : MDD to be updated, constraints
Output: Updated MDD
for every edge e in MDD do
    if the move according to edge e violates constraints then
        Remove edge e from MDD;
    end
    for each node n in MDD do
        if n is not the start location or goal location then
            if n does not have parents or children then
                Delete node n and all edges that are incident to n from MDD;
            end
        end
    end
end
return MDD
```

3 Methodology

We will investigate under which condition that some heuristics are better or worse. The number of generated and expanded nodes is the criterion for the performances of CBS with different heuristics. We think the instance from the note (figure 1) is a good instance since we can get different h_values when we apply different heuristics. We decided to divide the large map into three parts. And we combine two of them each time to see how many nodes expanded and generated by different heuristics. Figure 2 shows the basic three parts separated from figure 1. Figure 3 shows different combinations of two of them. Instance 1 indicates that two agents have a cardinal conflict. Instance 2 indicates that two agents are dependent but do not have cardinal conflict. Instance 3 indicates that three agents have cardinal conflicts and the $\triangle_{ij}$ between any two agents is larger than 1.

4 Experimental setup

Our code is written in Python 3.7.4, and our experiments are conducted on a 2.80 GHz Intel Core i7-7700HQ laptop with 12 GB RAM.

5 Experimental results

Table 1 shows the number of generated and expanded nodes using different heuristics.

6 Conclusions

From the results, we can see that CBS with WDG always has the best results among all the instances. The reason is that WDG provides the most information among three heuristics. And from the results, we can clearly see the domination relationship: $WDG \geq DG \geq CG$. When there are only cardinal conflicts,

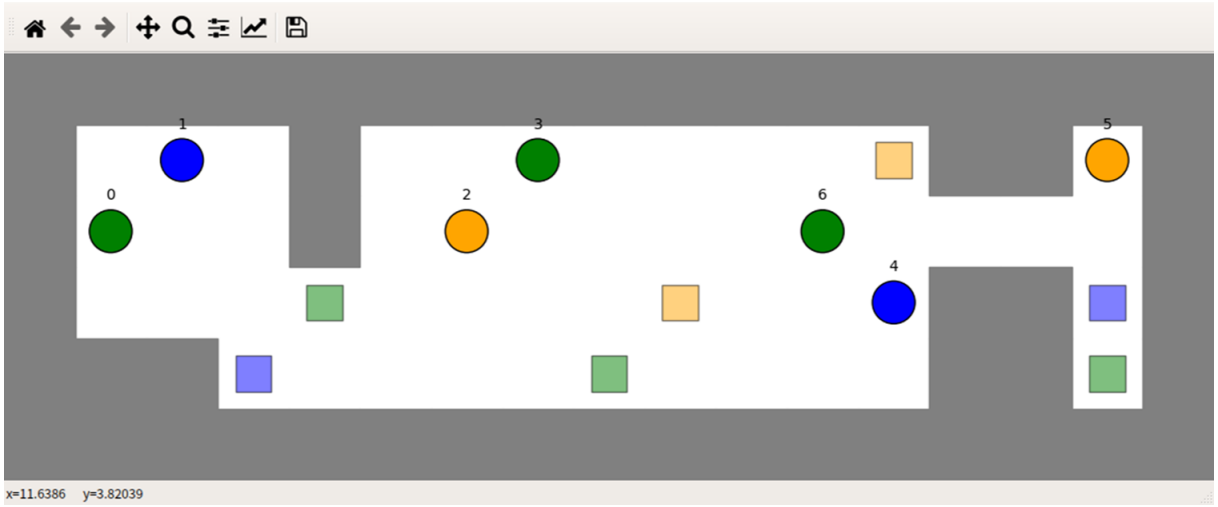


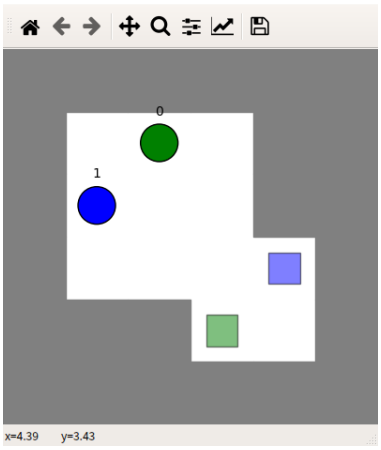
Figure 1: The results of MLP. Left:how well the model learns. Right: how well the predictions are

Table 1: Information about MLP and RNN

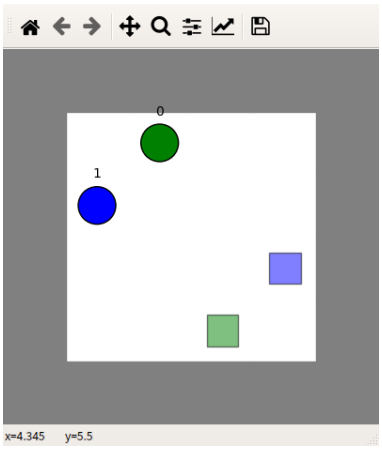
Instances	**No heuristic**		**CG**		**DG**		**WDG**	
	Generated	Expanded	Generated	Expanded	Generated	Expanded	Generated	Expanded
1	9	5	5	3	5	3	5	3
2	13	7	7	4	5	3	5	3
3	139	70	69	35	69	35	45	23
11	49	25	9	5	9	5	9	5
12	69	35	39	20	9	5	9	5
13	699	350	341	171	341	171	181	91
22	97	49	55	28	9	5	9	5
23	979	490	477	239	477	239	253	127
33	56817	28416	4805	2403	4527	2264	169	85
123	4899	2450	2381	1191	2381	1191	1261	631

three heuristics provide the same results. When there are only dependent agents, CG clearly has the worst results among the three heuristics but is still better than CBS with no heuristic. This indicates that during the solving process, constraints may create cardinal conflicts.

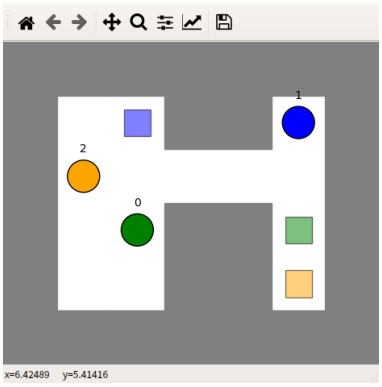
In conclusion, CG is good at those instances with cardinal conflicts but does not have the best performance when there are no cardinal conflicts. WDG has the best results among all the heuristic since it can provide not only the dependent relationship between agents but also the possible increased cost for the possible conflict-free solution. In the future, we may test three heuristics on more challenging instances so that we may see a larger difference between the number of generated and expanded nodes with different heuristics. And the time for constructing MDD could be an obstacle for solving an instance with a large map and a large set of agents. A new method for constructing MDD or a new structure for storing MDD could also be the next project we might focus on.



(a) Instance 1

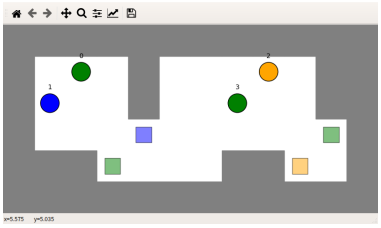


(b) Instance 2

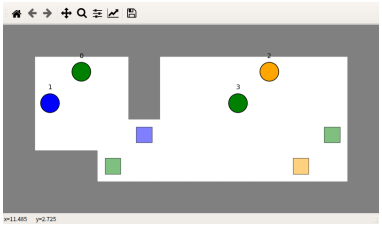


(c) Instance 3

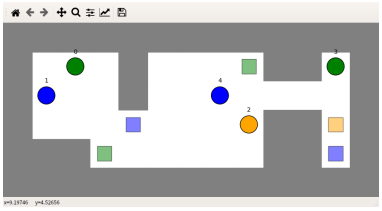
Figure 2: Three basic parts



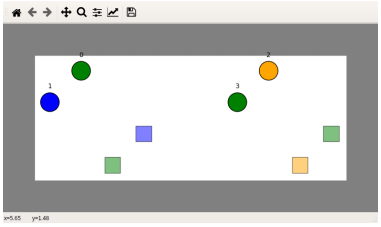
(a) Instance 11



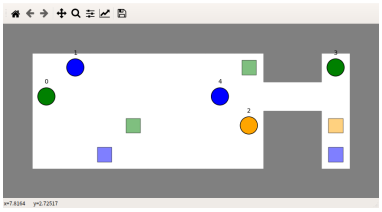
(b) Instance 12



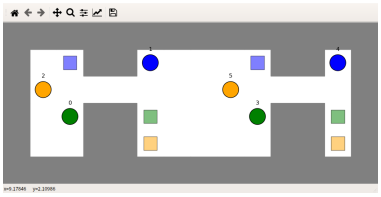
(c) Instance 13



(d) Instance 22



(e) Instance 23



(f) Instance 33

Figure 3: Three simple graphs

7 References

[1] Li, J., Felner, A., Boyarski, E., Ma, H., & Koenig, S. (2019, August). Improved heuristics for multi-agent path finding with conflict-based search. *In Proceedings of the 28th International Joint Conference on Artificial Intelligence (pp. 442-449)*. AAAI Press.
[2] Wang, L., Hu, S., Li, M., & Zhou, J. (2019). An Exact Algorithm for Minimum Vertex Cover Problem. *Mathematics*, 7(7), 603.