Project Report

Improved Heuristics for CBS and ICTS(partial)

Ashish Kumar Gola
301539266
akg42@sfu.ca

Contents

1 Introduction **1**

2 Algorithm Design and Theoretical Analysis **1**
2.1 Improved Heuristics for CBS algorithm - DG 2
2.2 Improved Heuristics for CBS algorithm - WDG 3
2.3 Improved Heuristics for ICTS algorithm 5

3 Implementation **8**
3.1 Code Base Structure 8
3.2 Building the MDDs 8
3.3 Merging two MDDs 8
3.4 Building the Dependency Graph and Checking Cardinal Conflict 9
3.5 Computing Size of Minimum Vertex Cover 9
3.6 Building the Weighted Dependency Graph 10
3.7 Computing Size of Edge Weighted Minimum Vertex Cover 10
3.7.1 Dividing G_{WD} into multiple Connected Components using DFS 10
3.7.2 Using ILP to compute the EWMVC 10
3.8 Using Lazy Computation Approach 10
3.9 Using Memoization 11
3.10 Merging k-MDDs ($k > 2$) for ICTS 11

4 Methodology **12**

5 Experimental setup **13**

6 Experimental results **13**

7 Conclusions **17**

References **17**

A Appendix **18**
A.1 Source Code - Explanation 18
A.2 Source Code - External Dependencies 19

1 Introduction

In the Single-agent Path Finding problem, we are given a graph along with a pair of source and goal vertex, and our aim is to find a path from source to goal vertex. The **Multi-Agent Path Finding**(MAPF) problem is a generalization of the single-agent path finding problem for more than one agents. Here, in the MAPF problem, we are given a graph with a set of multiple(more than 1) agents along with their respective start and goal positions, and our aim is to find paths for all agents such that the agents do not collide with each other. In addition to this, we also want to minimize the cumulative cost(i.e. the sum of time steps required by every agent to reach its goal). The MAPF is a fundamental problem in the field of AI, having numerous application in robotics, aviation, video games, etc.

Conflict Based Search(CBS) [Sharon et al., 2015] is a popular two level search algorithm to optimally solve the MAPF problem. CBS resolves collisions by adding constraints at higher level and finding paths consistent with those constraints at low level. CBS by default uses collisions arbitrarily at high level. CBS is used in numerous real world applications. There has been a number of research works and enhancements in CBS including ICBS[Boyarski et al., 2015], CBSH [Felner et al., 2018], CG and DG Heuristics [Li et al., 2019] etc.

Increased Cost Tree Search(ICTS)[Sharon et al., 2013] is another two level search algorithm for the MAPF problem. At high level, it assigns cost vectors to each node(representing the cost of each agent), then it visits nodes in a breadth first manner. At the low level, it attempts find a solution by merging the MDDs of all agents.

In this project, I primarily study and implement two heuristics for CBS, namely (1) DG, and (2) WDG, introduced in [Li et al., 2019]. I also attempt to apply these heuristics to ICTS algorithm [Sharon et al., 2013] (partially done).

2 Algorithm Design and Theoretical Analysis

Improved CBS(ICBS)[Boyarski et al., 2015] choses to resolve collisions by prioritizing cardinal conflicts over semi-cardinal and non-cardinal conflicts(a conflict is cardinal iff all shortest paths of the two conflicting agents traverse the conflicting vertex/edge at the conflicting time step). CBSH [Felner et al., 2018] introduces a heuristic where it makes a Conflict Graph(CG) using the cardinal conflicts for each CT node, N. A pair of vertices has an edge between them iff their corresponding agents have a cardinal conflict at N. CBSH then takes the size of the minimum vertex cover in this conflict graph as an admissible h-value for N.

Note that the conflict graph in CBSH does not consider semi-cardinal or non-cardinal conflicts. The DG and WDG heuristics on the other hand consider the semi-cardinal and non-cardinal conflicts also, as they might cause an increment in the total cost in future(in the descendent CT nodes). Thus the conflict graph of the CG Heuristic is always a subgraph of the dependency graph of the DG. It also implies that DG strictly dominates CG.

2.1 Improved Heuristics for CBS algorithm - DG

The idea is to consider semi-cardinal and non-cardinal conflict also if they can cause an increment in the total cost in future[Li et al., 2019].

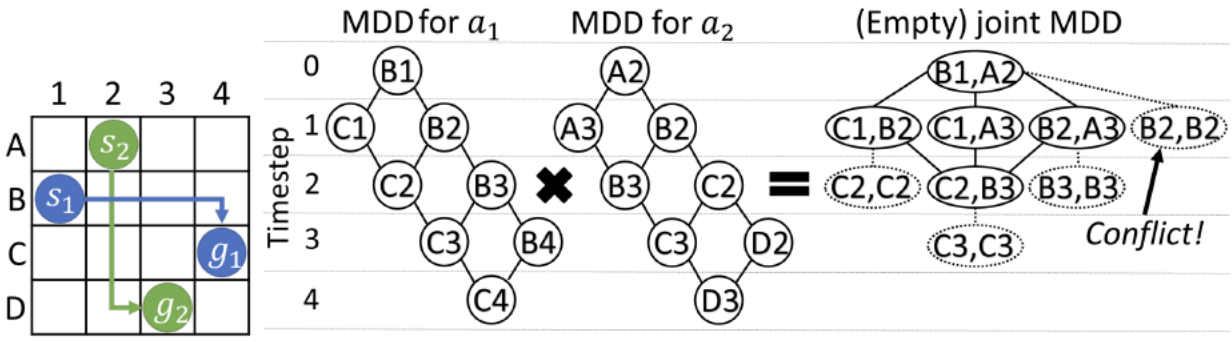

Figure1 : reference - [Li et al., 2019]

In Figure1, note that although the conflict at B2 is not cardinal(because there is another shortest path for each agent), but all those shortest paths have conflicts in future. Thus an h-value of 1 is admissible here.

We build the dependency graph for each CT node. We now briefly describe the Algorithm 1 to build the Dependency graph for a CT node.

Input: a CT Node N
Output: A Dependency Graph G_D
Let $a_1, a_2, .., a_k$ be the all agents.
Initialize a Dependency Graph with all agents and empty edges
for *each pair* a_i, a_j **do**
  if *There is no conflict of any kind between* a_i, a_j *at CT node N* **then**
    No Edge added to the dependency graph between the vertices presenting a_i, a_j
  else if *There is any(at least one) cardinal conflict between* a_i, a_j **then**
    Add an Edge to the dependency graph between the vertices presenting a_i, a_j
  else if *Each conflict between* a_i, a_j *is either semi-cardinal or non-cardinal* **then**
    Build $MDD_i^{c_i}, and MDD_j^{c_j}$.
    Run the merge algorithm to join both MDDs
    if *joint MDD is Empty* **then**
      Add an Edge to the dependency graph between the vertices presenting a_i, a_j
    else
      No Edge added to the dependency graph between the vertices presenting a_i, a_j
return the Dependency Graph

Algorithm 1: Constructing Dependency Graph

In the DG heuristic, we need to build the above graph G_D from scratch at the root, but for subsequent CT nodes, since each CT node has an additional constraint imposed on only one agent, we only need to look at the dependencies between this agent and all other agents and can copy the edges for the other pairs of agents from the GD for the parent CT node. Thus we can modify the

above pseudo code in according to this and save on computation.

We now present the DG Heuristic Algorithm.

Input: map, start locations, goal locations
Output: Cost Optimal Collision-free Solution

```
1  Let root.paths = paths of all agents using single agent a*-search
2  root.collisions = detect-collisions(root.paths)
3  root.cost = get-sum-of-cost(root.paths)
4  Build a dependency graph, G_D for root node using Algorithm 1
5  Compute the size of Minimum Vertex Cover of root.G_D, assign this size to be the h-value
     of root
6  Insert root into open-list queue(the queue is prioritized based on cost+h-value)
7  while open-list is not empty do
8      Let p = open-list.pop() (//Node Expansion)
9      if p.collisions = φ then
10         return p.paths
11     collision = p.collision.pop()
12     constraints = standard-splitting(collision)
13     for constraint in constraints do
14         q = new CT Node() (//Node Generation)
15         q.constraints = p.constraints ∪ constraint
16         q.paths = p.paths
17         let a_i = the agent in constraint
18         path = a*-search(a_i, q.constraints)
19         if path is not Empty then
20             Replace the path of agent a_i in q.paths
21             q.collisions = detect-collisions(q.paths)
22             Build the dependency graph for CT node q
23             Compute the MVC for q from q.G_D, assign this to q.h-value
24             q.cost = get-sum-of-cost(q.paths)
25             insert q into open-list
26 return Null //if reached here, it means no solution found
```

Algorithm 2: CBS with DG Heuristic

Correctness Analysis By construction and design of the dependency graph, for each edge (a_i, a_j), the cost of one of the agents has to increase by at least 1 to resolve this dependency. Hence, the size of the MVC of G_D is an admissible h-value for N. Therefore, the DG heuristic is admissible.

2.2 Improved Heuristics for CBS algorithm - WDG

In the DG Heuristic, we consider the dependency between all pairs of agents, if there is a dependency we assign an edge, but this does not capture the extent or how large value this dependency has. The paper [Li et al., 2019] introduced the WDG heuristic, which captures not only the pair-

wise dependencies between agents but also the extra cost that each pair of dependent agents will contribute to the total cost.

Construction of Weighted Dependency Graph To build G_{WD}, we first build the G_D graph as described in Algorithm 1. Then for each edge (a_i, a_j) in the G_D graph, we run the two agent MAPF solver for these two agents only(assuming there is no other agents). The difference in the total cost returned by the two-agent solver and the sum of both agents total costs at current CT Node is the weight of the edge. Since both agents cost need to be increased by at least this difference, we assign it as the weight of edge. The size of Edge Weighted Minimum Vertex Cover is an admissible heuristic. The EWMVC is an assignment of non-negative integers $x_1, x_2, ..., x_k$ to all agents, such that $x_i + x_j > w_{i,j}$, where $w_{i,j}$ is the weight of edge (i,j) in G_{WD} graph. Note that each x_i can be interpreted as the increase in cost of the agent a_i.

Input: a CT Node N
Output: A Weighted Dependency Graph of all agents
Let $a_1, a_2, .., a_k$ be the all agents.
Build a Dependency Graph G_D using Algorithm 1
for *each edge* (i, j) *in* G_D *graph* **do**
    Run two agent MAPF solver for agents a_i, a_j, Let $\Delta_{i,j}$ be the cost returned by solver
    Assign weight $w_{i,j} = \Delta_{i,j} - (c_i + c_j)$ to this edge in G_{WD}
return Weighted Dependency Graph G_{WD}

Algorithm 3: Constructing Weighted Dependency Graph

Similar to DG Heuristic, in the WDG heuristic also, we need to build the above graph from scratch at the root only. But for subsequent CT nodes, since each CT node has an additional constraint imposed on only one agent, we only need to look at the dependencies between this agent and all other agents and can copy the edges along with their weights for the other pairs of agents from the Weighted Dependency graph for the parent CT node.

We now briefly present the pseudo code of CBS with WDG heuristic.

Input: map, start locations, goal locations
Output: Cost Optimal Collision-free Solution
Let root.paths = paths of all agents using single agent a*-search
root.collisions = detect-collisions(root.paths)
root.cost = get-sum-of-cost(root.paths)
Build a weighted dependency graph, G_{WD} for root node as described above
Compute the size of Edge Weighted Minimum Vertex Cover of root.G_{WD} , assign this size to be the h-value of root
Insert root into open-list queue(the queue is prioritized based on cost+h-value)
while *open-list is not empty* **do**
    Let p = open-list.pop() (//**Node Expansion**)
    if *p.collisions* = ϕ **then**
        return p.paths
    collision = p.collision.pop()
    constraints = standard-splitting(collision)
    for *constraint in constraints* **do**
        q = new CT Node() (//**Node Generation**)
        q.constraints = p.constraints ∪ constraint
        q.paths = p.paths
        let a_i = the agent in constraint
        path = a*-search(a_i, q.constraints)
        if *path is not Empty* **then**
            Replace the path of agent a_i in q.paths
            q.collisions = detect-collisions(q.paths)
            Build the weighted dependency graph for CT node q
            Compute the EWMVC for q from q.G_{WD} , assign this to q.h-value
            q.cost = get-sum-of-cost(q.paths)
            insert q into open-list
return Null //if reached here, it means no solution found

Algorithm 4: CBS with WDG Heuristic

Correctness Analysis By construction and design of the weighted dependency graph, for each edge (a_i, a_j), the total increment in both agents cost has to be at least the weight of this edge. Hence, the value of the EWMVC of G_{WD} graph is an admissible h-value for N. Therefore, the WDG heuristic is admissible.

2.3 Improved Heuristics for ICTS algorithm

The Increased Cost Tree Search algorithm [Sharon et al., 2013] is a two-level search algorithm for optimally solving the MAPF problem. At a high level, it visits nodes in increasing order of cumulative cost in a breadth first manner. In this project I attempt to answer the question whether we can apply the DG and/or WDG heuristics to the ICTS algorithm. We design below Algorithm 5 for ICTS with DG heuristic and analyze its correctness.

Input: $k - agents$ MAPF

Build the root of ICT with cost-vector $[r_1, r_2, .., r_k]$ where $[r_1, r_2, .., r_k]$ are shortest individual costs.

Build an unweighted dependency graph by joining pairwise MDDs of all agents.

Let the h-value of root be equal to the size of MVC of this graph

for *each ICT node in* **Best First manner** *(//High Level Search)* **do**

    Let q be the current node, let it's cost vector be $[c_1, c_2, .., c_k]$

    Build MDDs for each agent : $MDD_1^{c_1}, MDD_2^{c_2}, .., MDD_k^{c_k}$

    Build the Dependency Graph from these MDDs by Performing pairwise searches(i.e. joining the $MDD_i^{c_i}$ and $MDD_j^{c_j}$, there is an edge if joint MDD is empty)

    if *Number of Edges in Graph* > 0 **then**

        Let Δ be the size of Minimum Vertex Cover of above Graph

        Create k children of the node q by increasing the cost of each agent by 1 at a time.

        For each child of the q, Let $h - value = (\Delta - 1)$. (Since the cost of child is already increased by 1, so there is a decrement of 1 from Δ)

        Insert these children nodes of q into open list (**//Node Generation**)

    else

        Perform a k-merge on MDDs of all k-agents(**//Low Level Search**)

        if *k-merge succeeds i.e. goal node was found* **then**

            return Solution

Algorithm 5: ICTS with DG Heuristic

Correctness Analysis For each edge in the dependency graph, We know that to resolve a dependency, the increment in total cost of both agents has to be at least 1. Thus to cover whole graph, the increment in cost need to be at least the size of minimum vertex cover before all of these dependencies can be removed(so that we can merge the MDDs for all pairs). Since each child at high level search in ICTS has a cost higher from its parent by 1, the h-value of ($size\ of\ MVC - 1$) is *admissible.*

Next, We present below the algorithm for ICTS with WDG heuristic.

Input: $k - agents$ MAPF

Build the root of ICT with cost-vector $[r_1, r_2, .., r_k]$ where $[r_1, r_2, .., r_k]$ are shortest individual costs.

Build a dependency graph by joining pairwise MDDs of all agents(in similar manner as ICTS with DG).

For each edge (i, j) in graph, let $OPT_{i,j}$ be the cost returned by two-agent MAPF agent Solver. Let the weight of this edge,$w_{i,j} = OPT_{i,j} - (r_1 + r_2)$

Let the h-value of root be equal to the size of EWMVC of this weighted graph

for *each ICT node in* **Best First manner** *(//High Level Search)* **do**

    Let q be the current node, let it's cost vector be $[c_1, c_2, .., c_k]$

    Build MDDs for each agent : $MDD_1^{c_1}, MDD_2^{c_2}, .., MDD_k^{c_k}$

    Build the Dependency Graph from these MDDs by Performing pairwise searches(i.e. joining the $MDD_i^{c_i}$ and $MDD_j^{c_j}$, there is an edge if joint MDD is empty)

    if *Number of Edges in Graph* > 0 **then**

        For each edge (i, j), compute $w_{i,j} = OPT_{i,j} - (c_i + c_j)$, where $OPT_{i,j}$ are cost returned by two-agent MAPF solver

        Let Δ be the value of Edge Weighted Minimum Vertex Cover of above weighted Graph

        Create k children of the node q by increasing the cost of each agent by 1 at a time.

        For each child of the q, Let $h - value = (\Delta - 1)$. (Since the cost of child is already increased by 1, so there is a decrement of 1 from Δ)

        Insert these children nodes of q into open list (**//Node Generation**)

    else

        Perform a k-merge on MDDs of all k-agents(**//Low Level Search**)

        if *k-merge succeeds i.e. goal node was found* **then**

            return Solution

Algorithm 6: ICTS with WDG Heuristic

Correctness Analysis We now analyze the correctness of Algorithm 6

For each edge in the weighted dependency graph, We know that to resolve a dependency, the increment in total cost of both agents has to be at least the weight of edge between them. Thus to cover whole graph, the increment in cost need to be at least the value of edge weighted minimum vertex cover before all of these dependencies can be removed(so that we can merge the MDDs for all pairs). Since each child at high level search in ICTS has a cost higher from its parent by 1, the h-value of $(value\ of\ EWMVC - 1)$ is *admissible*.

3 Implementation

3.1 Code Base Structure

I have used the same code base from individual project `https://coursys.sfu.ca/2022su-cmpt-827-x1/pages/Individual_Code/download`. I have written all the code in python language.

At most of the places, to save memory, I have assigned every node an id, and then I store this id in a dictionary. Later on I use this id to refer to this node. This way I save some memory(in context of MDDs one node might appear at multiple places, and now I just need to repeat its id and not the whole information about this node). Of course, there might be better methods to achieve this.

3.2 Building the MDDs

We use Breadth First Search to build the MDDs. Since we need to find all paths of a given cost, we need to store a list of parent nodes(since there can be more than one way to arrive at a node). Once I find the goal using BFS, I traverse back to source, storing child information. Again a node can have multiple children, thus I need to store them in a list. We now provide the pseudo code to build MDD. For more details, please refer to the source code(function build_mdd in file mdd.py)

Input: map, agent, constraints, start, goal, cost
Output: a directed acyclic graph containing all paths from start to goal with length = cost

```
1  Build a constraint table indexed by time step
2  Build root with start location and time step = 0
3  Insert root into open-list queue
4  while open list is not empty And goal is not reached do
5      Let s = open-list.size()
6      for i in range(0,s) do
7          Let q = Extract node from open-list
8          if Test for Goal Detection then
9              Generate DAG using Parents pointers.
10             return DAG
11         for each neighbour p of q do
12             Add q to p.parents-list
13             Check for Duplication, if not duplicate , insert p into open-list
14 return None (//If reached here, means no path exist of length = cost )
```

Algorithm 7: Building MDDs

3.3 Merging two MDDs

Given two MDDs, our aim is to merge them. We follow the same merge procedure described in [Sharon et al., 2013]. The idea is to traverse both the MDDs in BFS manner. But there is an

interesting case of handling common locations for both nodes at same level(of time step). The solution is that one node can appear in one of them and not both of them, if we are looking at the same level of BFS. We present the Algorithm 8 to check if joint mdd of two mdds is empty. For more details, please refer to the source code(function is_joint_mdd_empty in file mdd.py)

Input: mdd1,mdd2
Output: return True if join MDD is empty, else return False
Build a joint-list queue
Add (source1, source2) to joint-list
while *joint-list is not empty* **do**
    Let s = joint-list.size()
    for *i in range(0,s)* **do**
        Let (p1,p2) = Extract node from open-list
        if *Test for Both Goal Reached with p1 and p2* **then**
            return False (//joint mdd is not empty, so returning False)
        for *each neighbour q1 of p1 in mdd1, each neighbour q2 of p2 in mdd2* **do**
            if *q1 != q2* **then**
                Insert (q1,q2) into joint-list
return True (//If reached here, that means joint mdd is empty)

Algorithm 8: Is Join MDD Empty

3.4 Building the Dependency Graph and Checking Cardinal Conflict

We have described building the dependency graph in Algorithm 1. For more, implementation details, please see source code(function build_dependency_graph in file dependency_graph.py). To check if a conflict is cardinal [Boyarski et al., 2015], we check the width of both MDDs at the time step(of conflict). Let the conflict be equal to $< a_i, a_j, D, t >$, and if the width of both MDDs at time step t is equal to 1, then the conflict is cardinal .

3.5 Computing Size of Minimum Vertex Cover

In the DG heuristic, we compute the size of minimum vertex cover. This being a NP hard problem, we attempt to solve a parameterized version of this problem. Instead of searching for the minimum vertex cover, we attempt to answer whether there exists a vertex cover of size at most q. Here q is the parameter. This problem can be solved in $O(2^q \times |V|)$ using the algorithm from [Downey and Fellows, 1995]. We briefly describe this Algorithm 9

For more details of this algorithm implementation, please refer to source code(function does_mvc_exist in file minimum_vertex_cover.py)

Input: Graph G = (V, E), parameter q
Output: return True there is a vertex cover of size at most q, otherwise return False

```
1  if E is empty then
2      return True
3  else
4      Pick an edge randomly from E
5      Let u,v = e
6      Recursively Call This procedure on (V − u, E1) and (V − v, E2) with parameter (q-1)
7      if either of them Succeeds then
8          return True
9      else
10         return False
```

Algorithm 9: Does a vertex cover of size at most q exist

3.6 Building the Weighted Dependency Graph

We have described building the weighted dependency graph in Algorithm 3. For more, implementation details, please see source code(function build_wdg in file weighted_d_graph.py)

3.7 Computing Size of Edge Weighted Minimum Vertex Cover

We use reductions to compute the EWMVC. Given an edge weighted graph, we transform it into an instance of Integer Linear Programming problem. Basically, the vertices of graph are converted into variables, the edge weights are converted into constraints. And the objective function is to minimize the sum of variables.

To optimize the running time for computing the EWMVC, we first obtain the connected components of the graph using Depth First Search. Then we run the ILP Solver for EWMVC on each connected component. And the sum of EWMVC for all connected components is the final value of EWMVC for the edge weighted graph.

3.7.1 Dividing G_{WD} into multiple Connected Components using DFS

Source code for obtaining Connected Components of graph based on DFS can be found here (function obtain_components in file ewmvc.py)

3.7.2 Using ILP to compute the EWMVC

Source code for ILP solver can be found here(function edge_weighted_mvc in file ewmvc_ILP_solver.py).

3.8 Using Lazy Computation Approach

In both implementations(DG and WDG), we use a lazy approach, that is we use two heuristics functions, h1 and h2. At the time of node generation, we assign value of h1(We use 0 as h1, it is admissible. One can use any other heuristic as well). Then at the time of node expansion, I check if h2 is applied to this node, if not, h2 is computed and the node is re-inserted in the open-list with

new h-value of h2. Since the computation of h2 is cpu-intensive, we save time with lazy approach. The relevant source code is here(Solver in file dg_heuristic.py and wdg_heuristic.py).

3.9 Using Memoization

We heavily use memoization to store weights of edges in WDG Heuristic and DG Heuristic. The cache key is agent1, agent2, and their constraints. And the value is the weight(this weight is 1 in case of DG Heuristic). We also store MDDs of individual agents. The key in this case is the agent, cost of MDD, and constraints of agent, and the value is MDD itself. Relevant source code can be found here(files dependency_graph.py, and weighted_d_graph.py).

3.10 Merging k-MDDs ($k > 2$) for ICTS

Here, we generalize the Algorithm 8 for k MDDs. The source code for this can be found here(function is_k_joint_mdd_empty in mdd.py). To get the cross product of k lists , we use library function from itertools module.

4 Methodology

By implementing the DG and WDG heuristics I attempt to answer the following

- Out of CBS Standard(without heuristics), CBS with DG, and CBS with WDG, which one perform better in case of sparse environment.
- Out of CBS Standard(without heuristics), CBS with DG, and CBS with WDG, which one perform better in case of dense environment.
- Out of CBS Standard(without heuristics), CBS with DG, and CBS with WDG, which one perform better as the number of agents increases.

To answer these questions, I am running the Standard CBS, DG and, WDG Solvers on below instances.

- All the 50 instances of MAPF included in individual project
- Small dense instances with 20x20 grids, and with random 30%-density [Sturtevant, 2012].
- Small sparse instances with 20x20 grids with empty environment(no obstacles) [Sturtevant, 2012].
- A large instance with 196x196 grids with random 30%-density [Sturtevant, 2012].

For instances taken from above cited reference, I have taken the map environments from their website, and assigned the agents with random start and goal locations using a local random generator.

How does the performance of your algorithm(s) vary as a function of the number of agents, the percentage of obstacles in instances, the size of the instances, etc?
Although, I expected that WDG would run faster in the dense environment, but my experiment does not provide such result. One of the reason that I think is that I used ILP for EWMVC computation(and not the Branch and Bounded algorithm), this might be causing some significant overheads. I need to further investigate this.

If one of your algorithms sometimes returns suboptimal solutions, how close were those solutions to the optimal solutions? Did there exist a class of instances on which that algorithm was always nearly optimal?
Yes, on some(2-3 out of 50) instances from the individual project, my implementation returns suboptimal solutions, but their cost is just differed by 1 unit mostly(2 in one case).

5 Experimental setup

We have performed all experiments on a 3.2 GHz 8-core processor(M1), 8 GB RAM memory laptop. We have used **python 3.8.9** runtime environment. Our code is written in python. The OS used in MacOS(12.1)

6 Experimental results

I observe that DG and WDG both have returned optimal solution for most of the instances. But it had returned suboptimal solution(cost more than 1 unit mostly) for 2-3 instances out of the 50 instances from individual project(It implies there is a minor bug in my code which I am still debugging). Please find below the graphs belonging to various results of these experiments.

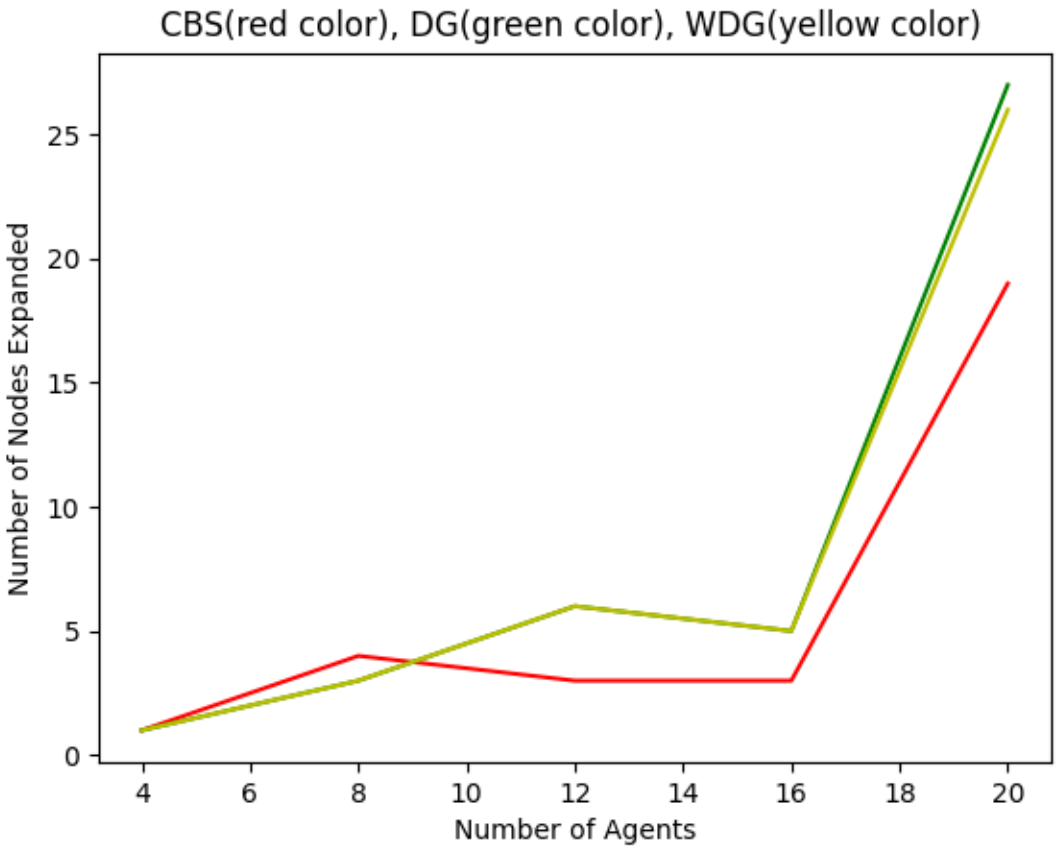

Figure2: Number of Nodes Expanded in Empty Instances of 20×20

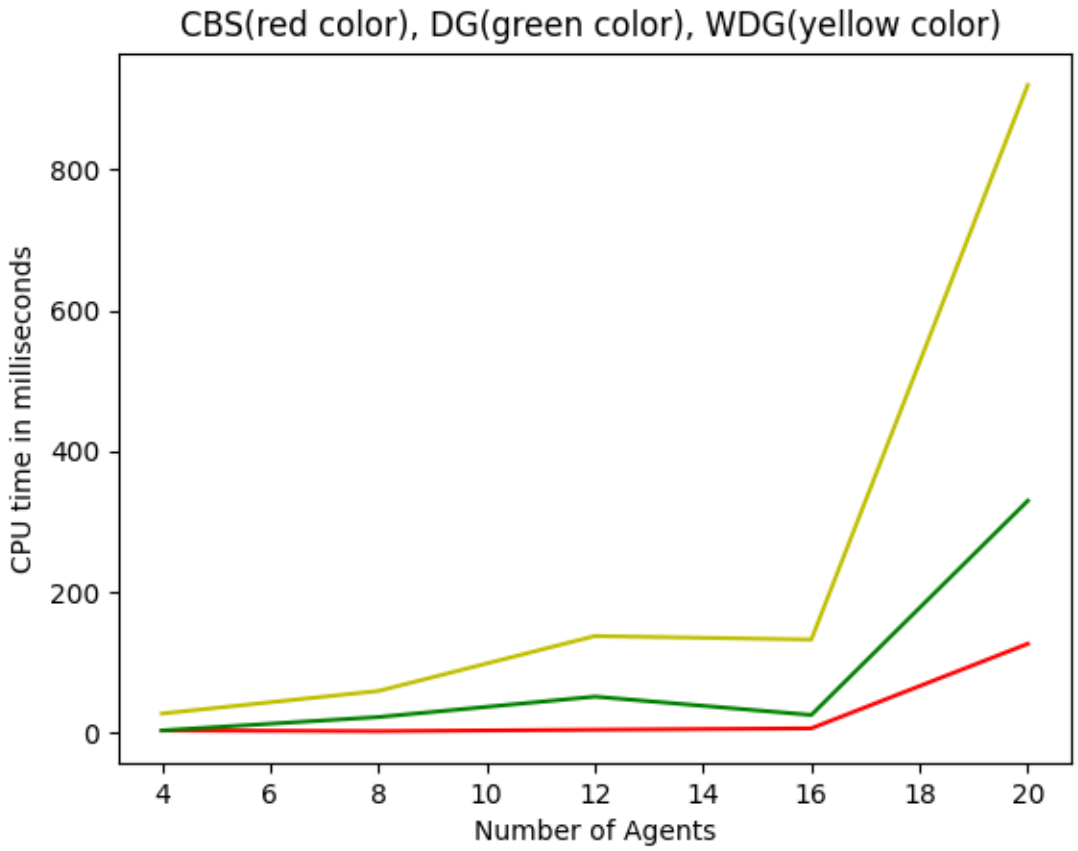

Figure3: CPU time in Empty Instances of 20×20

We observe that for empty instances, DG and WDG takes slightly more cpu time but this is the overhead of constructing these dependency graphs.

Next We present the result of running these algorithms over dense environments.

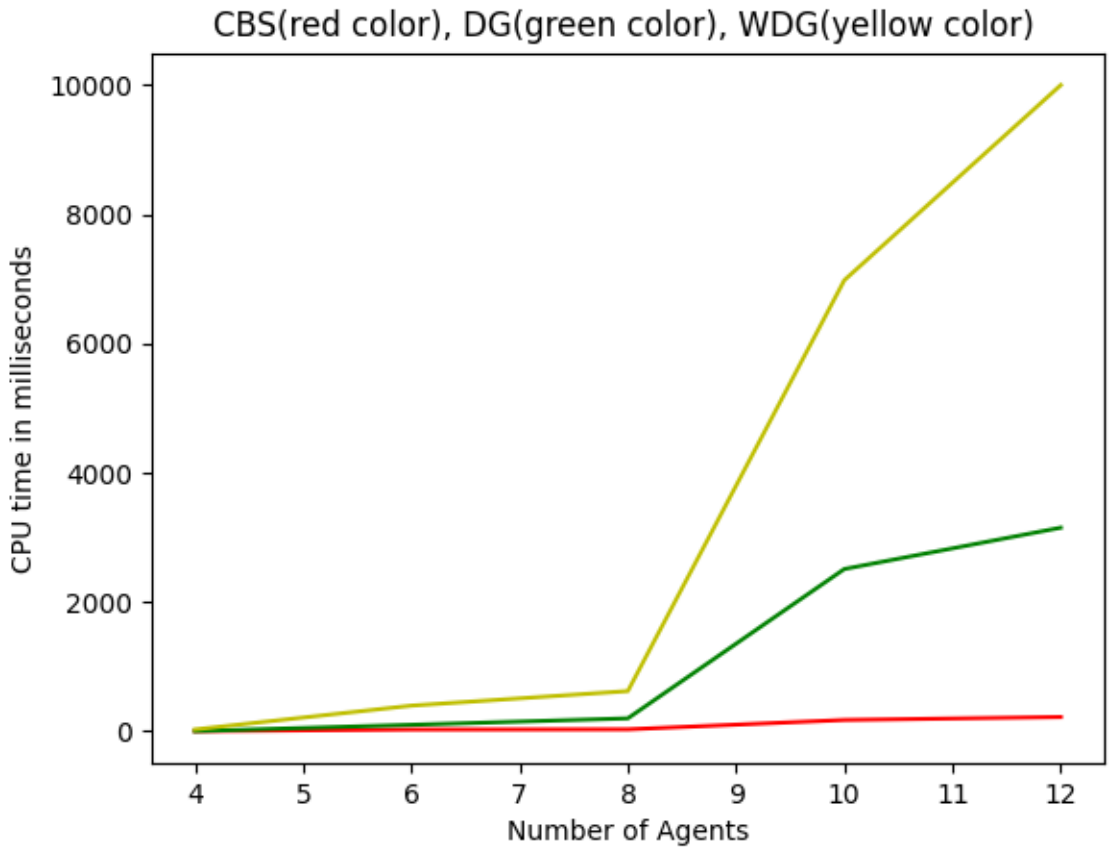

Figure4: CPU time in Dense Instances of 20×20 with 30%*obstacles*

I observe that WDG is taking more time than DG, but that is the overhead mostly due to computing EWMVC. Also I am using a third party package to run an ILP, so that might also effect this time.

We also see the h-values of WDG is on higher from DG which is in agreement of our expectations due to the weights of edges. Here is the graph for this comparison:

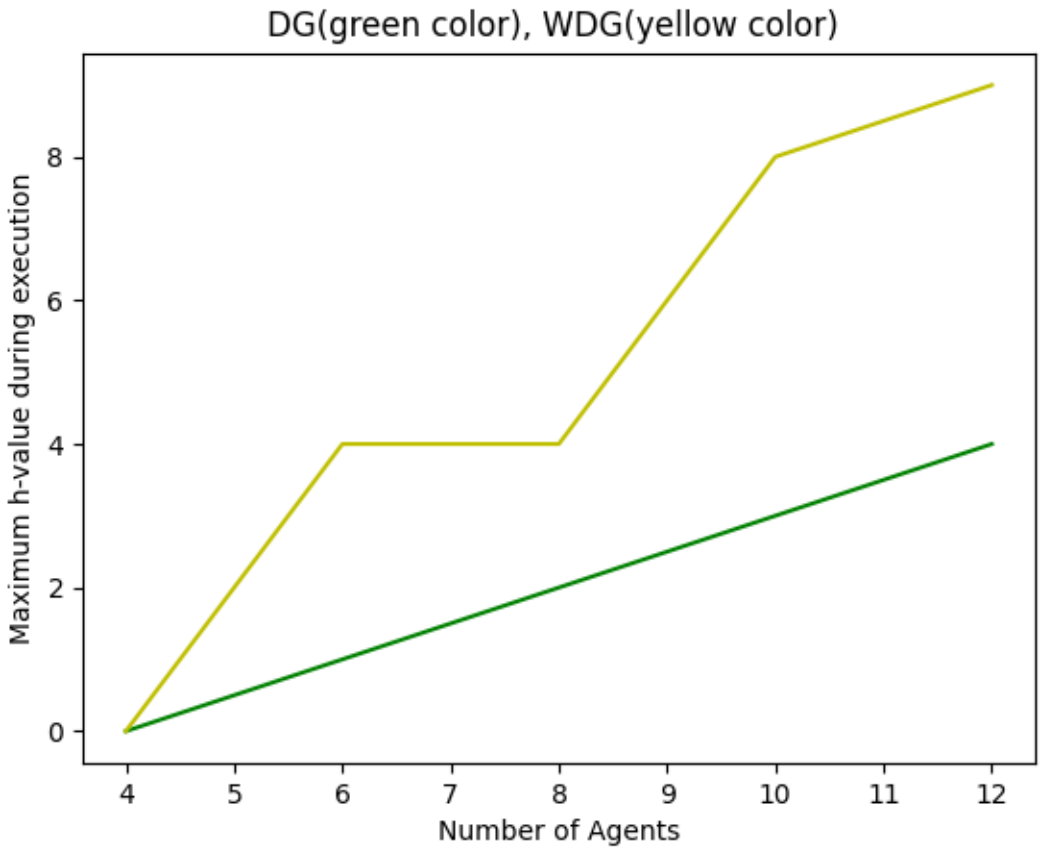

Figure4: Max-h value in Dense Instances of 20×20 with 30%*obstacles*

7 Conclusions

From the experiments, we see that both DG and WDG heuristics are admissible as both returns optimal solution. My broader take aways from the projects are having an insight into what goes into solving problems practically. The project provided an opportunity to implement various basic algorithms such as DFS, BFS, as well as advanced algorithm including the heuristics and CBS. It also provided an opportunity to build solutions for Minimum Vertex Cover and Edge Weighted Minimum Vertex Cover(using reductions to ILP).

If I were to continue to work on this problem, I would like to finish the implementation part of applying the DG and WDG heuristics to ICTS, complete the theoretical proof, and remove errors if there are any(in my reasoning about it). I would also like to make one more enhancement in building of MDDs, instead of a simple Breadth First Search, I would like to use Bidirectional BFS. This will significantly save the cpu time consumed in building MDDs.

On optimizing code - There are parts of the algorithm, which can be made faster by exploiting multithreading. For example, building of multiple MDDs can be parallelized as these are independent computations. That is also something, I would like to implement if I were to continue this work.

References

[Boyarski et al., 2015] Boyarski, E., Felner, A., Stern, R., Sharon, G., Tolpin, D., Betzalel, O., and Shimony, E. (2015). Icbs: Improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 740–746. AAAI Press.

[Downey and Fellows, 1995] Downey, R. G. and Fellows, M. R. (1995). Parameterized computational feasibility. In Clote, P. and Remmel, J. B., editors, *Feasible Mathematics II*, pages 219–244, Boston, MA. Birkhäuser Boston.

[Felner et al., 2018] Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T. K. S., and Koenig, S. (2018). Adding heuristics to conflict-based search for multi-agent path finding. In *ICAPS*.

[Li et al., 2019] Li, J., Felner, A., Boyarski, E., Ma, H., and Koenig, S. (2019). Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 442–449. International Joint Conferences on Artificial Intelligence Organization.

[Sharon et al., 2015] Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. (2015). Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence*, 219:40–66.

[Sharon et al., 2013] Sharon, G., Stern, R., Goldenberg, M., and Felner, A. (2013). The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495.

[Sturtevant, 2012] Sturtevant, N. R. (2012). Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4:144–148.

A Appendix

A.1 Source Code - Explanation

To run the CBS with DG Heuristic Algorithm, run below command

python run_experiments.py –instance "dense-30-20x20-6.txt" –solver DG

To run the CBS with WDG Heuristic Algorithm, run below command

python run_experiments.py –instance "dense-30-20x20-6.txt" –solver WDG

To run the CBS standard (without any heuristic), run below command

python run_experiments.py –instance "dense-30-20x20-6.txt" –solver CBS

dg_heuritics.py and **wdg_heuristics.py** are the main files, which uses dependency_graph.py and weighted_d_graph.py files.

mdd.py, minimum_vertex_cover.py and ewmvc.py files are utility files providing important computations to above mentioned files.

icts_with_dg.py is an attempt to implement Algorithm 5 but it is not complete yet.

A.2 Source Code - External Dependencies

To solve the ILP for edge weighted minimum vertex cover, I use this third-party package *ortools linear solver*. More details about this can be found on this page `https://developers.google.com/optimization/introduction/python` .

It can be installed with this command **pip3 install ortools**