CBS with Heuristics

CMPT 417 Final Project

Alex Nguyen, James Young, Zi Zhou Qu

Introduction

The problem of multi-agent path finding (MAPF) is computationally NP-hard, described as a set of agents < a_1, a_2, a_3, … a_x > where each agent has a start and goal location on an unweighted graph. The MAPF algorithm we employ in this report is Conflict Based Search (CBS) which uses a low level search and a higher search which resolves conflicting nodes for each agent such that each path of agent a_i has a set of constraints that satisfies a collision free path. To improve on CBS there are three heuristics that this paper focuses on, namely the conflict graph (CG), dependency graph (DG), and weighted dependency graph (WDG) heuristics. First, the CG heuristic determines whether a given node in an agent's constraint tree has a cardinal conflict, that is when the cost of the child constraints is greater than the parent constraint. If there are multiple cardinal conflicts, then a conflict graph is built where the minimum vertex cover of that graph is an admissible heuristic for the node. The DG heuristic considers conflicts in future solutions and the conflict graph becomes instead a dependency graph where the edges reflect cost-minimal paths of conflicting agents. Finally, the WDG heuristic considers the cost that each dependent agent contributes to the total cost where the dependency graph becomes a weighted dependency graph and the minimum vertex cover becomes an edge weighted one. With these three heuristics, CBS can be greatly improved on with only a small overhead cost.

Implementation

MDD Pseudo Code

```
Input: map of agents, start locations, goal locations, list of heuristic values, current agent, list of constraints

Output: list of MDD's for each agent

Initialize root to MDDNode with start locations and heuristic value

Initialize levels to empty list of lists

Build constraint table for the agent

Initialize open_list to empty list

Initialize closed_list to dictionary with the MDDNode root

Append root to open_list

While the length of open_list > 0
```

```
        Set curr to open_list.pop(0)
        If the current timestep >= length of levels
            Append empty list to levels
        Append curr to the timestep of the level
        If the current location equals goal location, then goal is reached
        Calculate upper bound of the search
        Check for possible moves using a* search
```

Conflict Graph Pseudo Code

```
    Input: list of MDD's as MDD_List
    Output: list of conflict graphs as graph
    Initialize graph to empty list
    For i = 1 to length of MDD_List
        Append empty list to graph
        For j = 1 to length of MDD_List
            If i >= j
                Append 0 to graph[i]
            Else
                If is_conflicting(MDD_List[i], MDD_List[j])
                    Append 1 to graph[i]
                Else
                    Append 0 to graph[i]
    For k = 1 to length of MDD_List
        For g = 1 to length of MDD_List
            If k < g
                graph[g][k] = graph[k][g]
    Return graph
```

Joint_MDD Code

```
    **Input:** MDD1, MDD2
    **Output:** Boolean, determining the dependency between the two MDDs.
    **Init()**
        Initialize a list for the two MDDs
        Initialize max_level as the max depth between the two MDDs
        Initialize num_of_agents as 2
        Initialize levels as an empty list.
        If one MDD is larger than the other
            Equalize the two MDDs
        For each level in the MDD
            Add all possible combination of the two MDDs for that level to levels
        For each level in levels
            For each combination in level
                Remove any vertex constraint
        Remove all parentless vertex
        For each level after level 0
            For each combination
                Get parents for combination
                If there is only one parent
                    If edge collision
                        Remove combination from level
        Remove all parentless vertex
    **remove_parentless()**
        To_remove =  []
        For levels from bottom-up
            For all combinations in level
                Find parents for that combination
                If parents.isEmpty
                    To_remove.append(combination)
            For combination in To_remove
                Remove from levels
```

Find_parents()

```
    ParentOfChild1 = []
    ParentOfChild2 =  []
    For every possible child movement
        ParentOfChild1.append(child1.move())
        ParentOfChild2.append(child2.move())
    Parents = All possible combination of ParentOfChild1 and ParentOfChild2
    Return Parents
```

Is_edge_collision()

```
    If pos1[0] == pos2[1] and pos1[1] == pos2[0]
        Return True
    Else
        Return False
```

Is_dependent()

```
    For level in levels
        If level is empty
            Return True
    Return False
```

Graph_from_MDD()

```
    Initiulize graph to be empty
    For each MDD:
        For each MDD:
            If outer MDD == inner MDD
                Skip
            Create a joint_MDD for the inner and outer MDD
            If joint_MDD is dependent
```

```
                        Append 1 to graph in position of outer MDD
        Return graph
    Weighted_graph_from_MDD()
        Initiulize graph to be empty
        For each MDD:
            For each MDD:
                If outer MDD == inner MDD
                    Skip
                Create a joint_MDD for the inner and outer MDD
                If joint_MDD is dependent
                    Find weight for that instance
                    Append weight to graph
        Return graph
```

MVC Pseudo Code

We decided to go with a brute force method due to the small number of conflicts or dependencies in our test cases.

```
    Input: List of conflict graphs as graph
    Output: minimum vertex cover
    Init()
        Initialize conflict graph as graph
        Initialize numOfVertex as length of graph
        Initialize cover as list of 0 to numOfVertex – 1
        Initialize minimumCover as length of graph
    GetMVC()
        Initialize subset as all subsets from getAllSubsets()
        For i in subset
            If checkCover(i) and length of i < minimumCover
                Cover = i
                MinimumCover = length of I
        Return minimumCover
    GetAllSubsets()
        Vertex = [0...numOfVertex]
        Subset = []
        For I in range(numOfVertex)
            Set = all combinations of vertex
            For s in set
                Subset.append(s)
        Return subset
    CheckCover(cover)
        For every vertex, I
            If I is in cover
                Continue
            For every vertex j, that is not I
                If j is connected to I and j is not in cover
                    Return False
            Return True
```

EWMVC Pseudo Code

Initially, we did not split the graph into its connected components. After noticing the large runtimes for graphs with agents larger than 5, we implemented the component split in order to reduce the number of agents used when calculating the weights. Although this help for some cases, instances with agents conflicting or dependent with more than 5 agents still suffer from the long run times.

```
    Init()
        Graph = graph
        NumOfVertex = len(graph)
    GetEWMVC()
        Components = component.split()
        Weight = 0
        For component in components
        Assignment = getComponentWeight(component)
        Weight += sum(assignment)
    Return weight
```

Component_split()

```
    Graph = self.graph
    Connect_graph = list of connected vertices to vertex
    While connect_graph isn’t combined into connected components
        Connect_graph = a more combined version of connect_graph
    New_graph  = []
    For component in connect_graph
        componentGraph = graph representation for vertices in component
        New_graph.append(component)
    For component in new_graph
        Delete any vertex that is not connect
        For vertex in component
            Delete any edge that is not connected to that component
    Return new_graph
```

Graph_to_connection(graph)

```
ConnectGraph = []
For every vertex in graph
    Append a list of connected vertices to that vertex to connectGraph
For every vertex in connectGraph
    Delete any non-neighboring vertex of the vertex
For every component in connectGraph sort the component
Return ConnectGraph
```

Combine_connect(graphConnection)

```
For every vertex in graphConnection
    For every vertex in graphConnection that is after the outer loop vertex
        If there is a shared vertex in the outer and inner vertex
```

```
            Combine the two vertex and add it to the list
            Delete the two old vertex.
            Return newgraph that was created.
**Combine_check(graph)**
    For every vertex in graph
        For every vertex in graph that is not the outer vertex
            If there is a shared vertex between the outer and inner vertex
                Return True
    Return False
**GetComponentWeight(component)**
    Initialize recursive arguments
    Call recursive function DFSRecursive(component, index, assign, best, branchBound)
**DFSRecursive(component, index, assign, best, branchBound)**
    If everything has been assigned
        Return assignment
    While x does not exceed bounded limit
        Assign[index] = x
        If assignment is not valid
            Continue
        CurrAssign = DFSRecursive(comp, index+1, Assign, best, branchBound)
        If currAssign is better than best
            BestAssign = currAssign
            Best = sum of currAssign
    Return bestAssign
**Check_assignment(component, assignment, index)**
    For every vertex in component[index]
        If assignment of x not assigned
            Continue
        If fails condition of xi + xj >= weight
```

```
                    Return False
          Return True
```

Methodology

We are investigating whether certain types of heuristics perform better on certain classes of MAPF problems. Particularly, we are looking at the runtimes, nodes expanded, nodes explored. The classes of instances that we used for our test were map density, map size, number of agents.

We noticed for both larger and smaller maps, the run time for the DG and WDG heuristic performs more poorly than CG heuristic and no heuristic. We also see that no heuristic performs slightly better than CG. The nodes expanded and explored were similar for the large maps but for smaller maps they result were reversed. With no heuristic expanding and exploring the largest number of nodes followed by CG, DG and then WDG.

In terms of the number of agents, we see that the number of agents affects WDG and DG the most. The more agents that are in present in the test instance, the worst WDG and DG performs, both in runtime and node expanded. Compared to WDG and DG, the number of agents has a bit of an effect on the CG and no heuristic implementations.

For map density, our results show that only runtime is generally affected. The no heuristic and CG performs better in more sparse maps.

Experiment Setup

Tested the experiments on the Windows operating system with Python 3.7 on an Intel i7 processor and 16 GB RAM.

The python library used for this project are the following:

- argparse
- glob
- itertools
- glob
- copy
- math
- time
- heapq
- random
- numpy
- matplotlib

Results

We ran experiments with the provided MAPF instances on node generation, expansion and general algorithm run time and compared the results of each. Each instance for the run time experiment was run for 5 times to obtain a mean and standard deviation to account for any potential outliers.

We first classify each instance as either an empty graph or dense graph as DG and WDG are affected by map density from our observations in individual testing. In the below graphs are plotted the test instances from test_1 to test_30 indicating sparse and empty graphs, and increasing number of agents in test_1 with 5 agents to test_30 with 8 agents. We see that in Figure 1 the run time of DG and WDG are significantly affected as number of agents increase. This is likely because of calculating the MVC and as such the running time is increased by from less than one second to eight seconds.

Node generation and expansion wise are as expected, since WDG strictly dominates DG, and DG strictly dominates CG. We see in Figure 2 and 3 that there is a significant improvement in expansion and generation as the number of agents increase to 8 as seen in instance test_23.

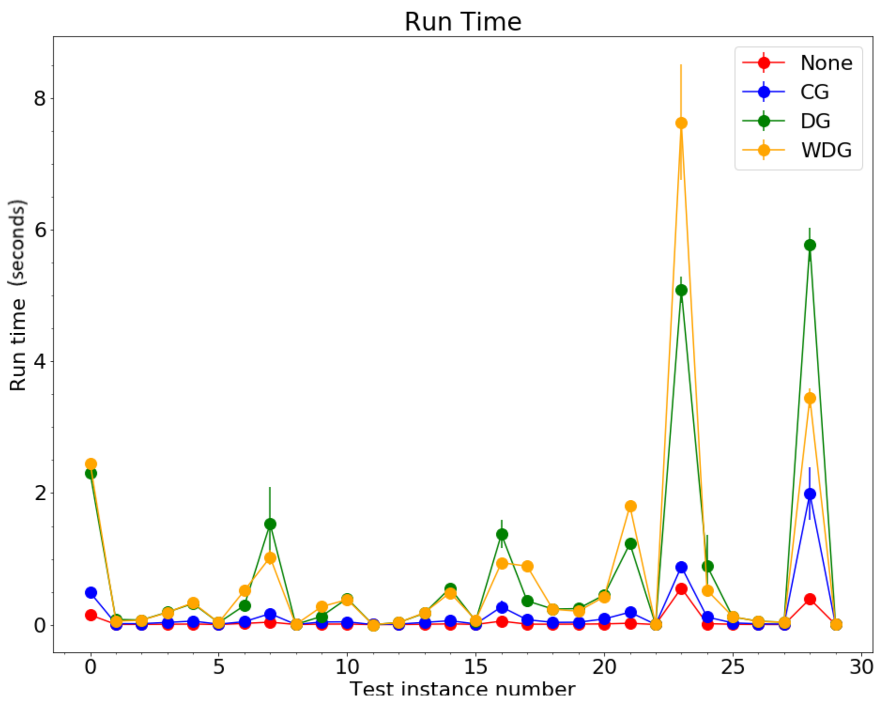

Figure 1: Sparse/Empty Map Running Time

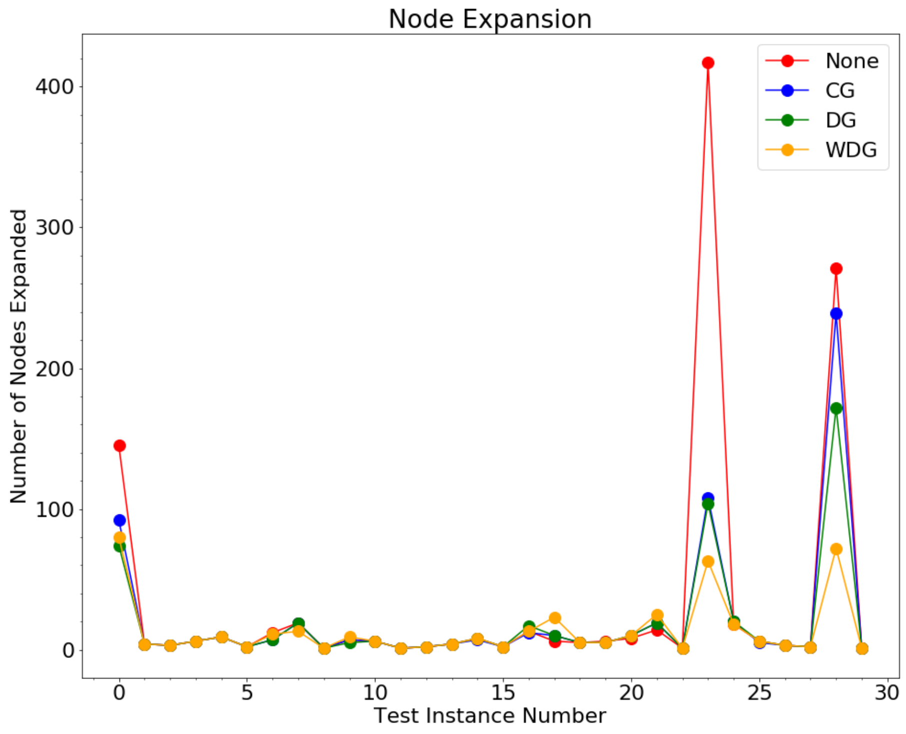

Figure 2: Sparse/Empty Map Node Expansion

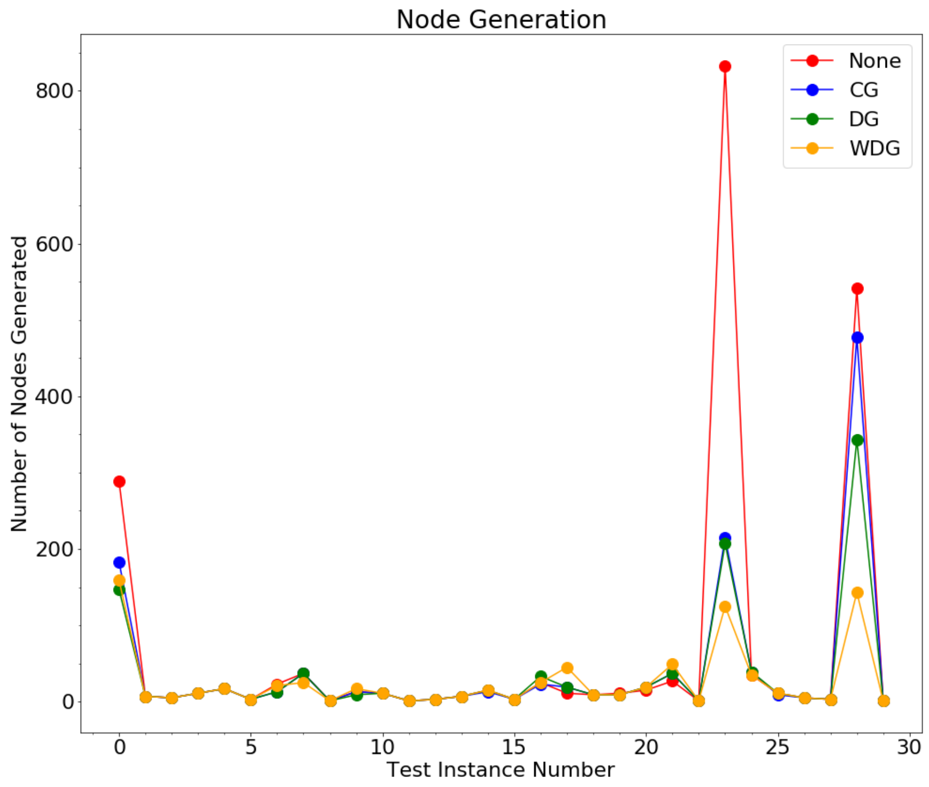

Figure 3: Sparse/Empty Map Node Generation

Applying the heuristics onto a dense graph with few nodes for valid paths, the heuristics have the same running time, except for WDG where in test instance 11 WDG has an extreme outlier running time, we expect this is due to the brute force MVC where seven or more vertices in its graph can exponentially increase the running time of the algorithm as seen in Figure 4.

For node expansion and generation, WDG performs better in almost every test instance as seen in Figure 5 and 6.

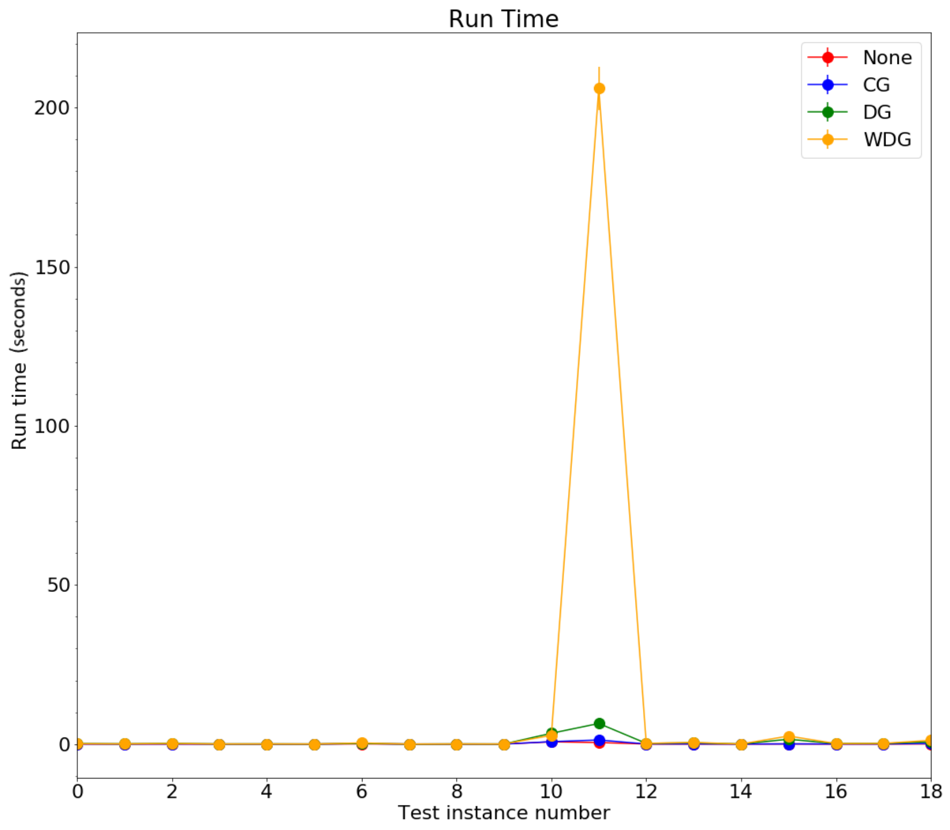

Figure 4: Dense Map Running Time

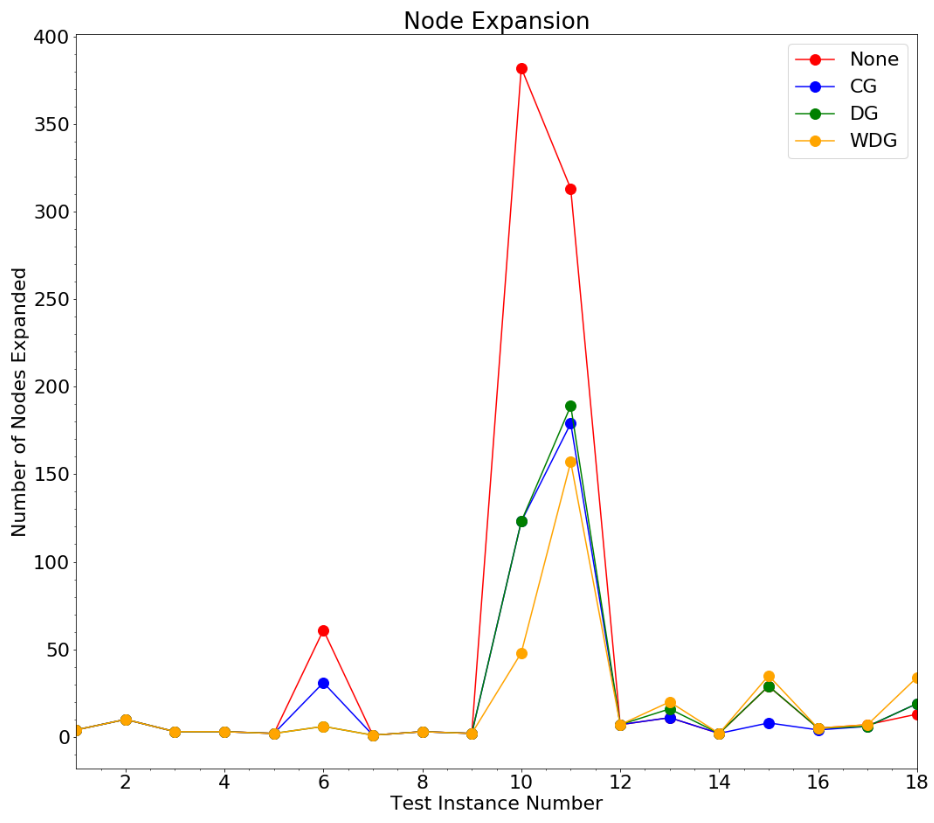

Figure 5: Dense Map Node Expansion

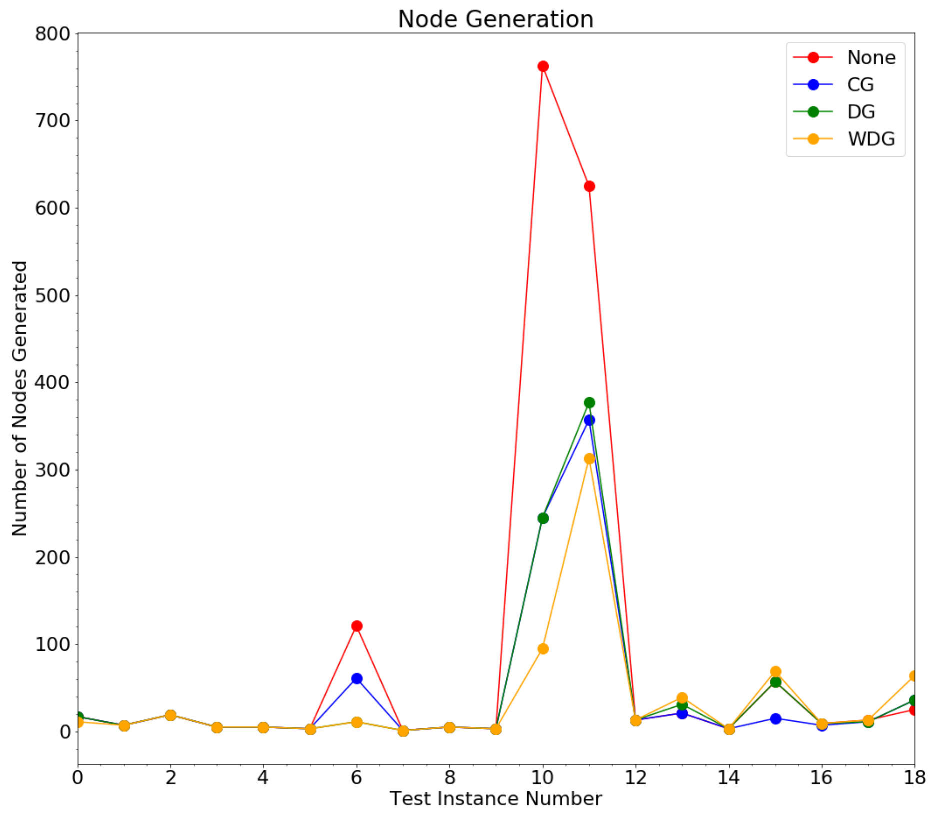

Figure 6: Dense Map Node Generation

When we consider larger 12x12 maps, where test instances 1 to 3 and 7 to 9 being empty graphs, and test instances 4 to 6 and 10 to 12 being dense graphs. In addition, test instances 1 to 6 having 2 agents to test instances 7 to 12 having 5 agents we see that DG and WDG has a significant running time overhead compared to CG and no heuristic. We believe this may be because the larger maps have more vertices to traverse and as such, fewer conflicts exist between agents.

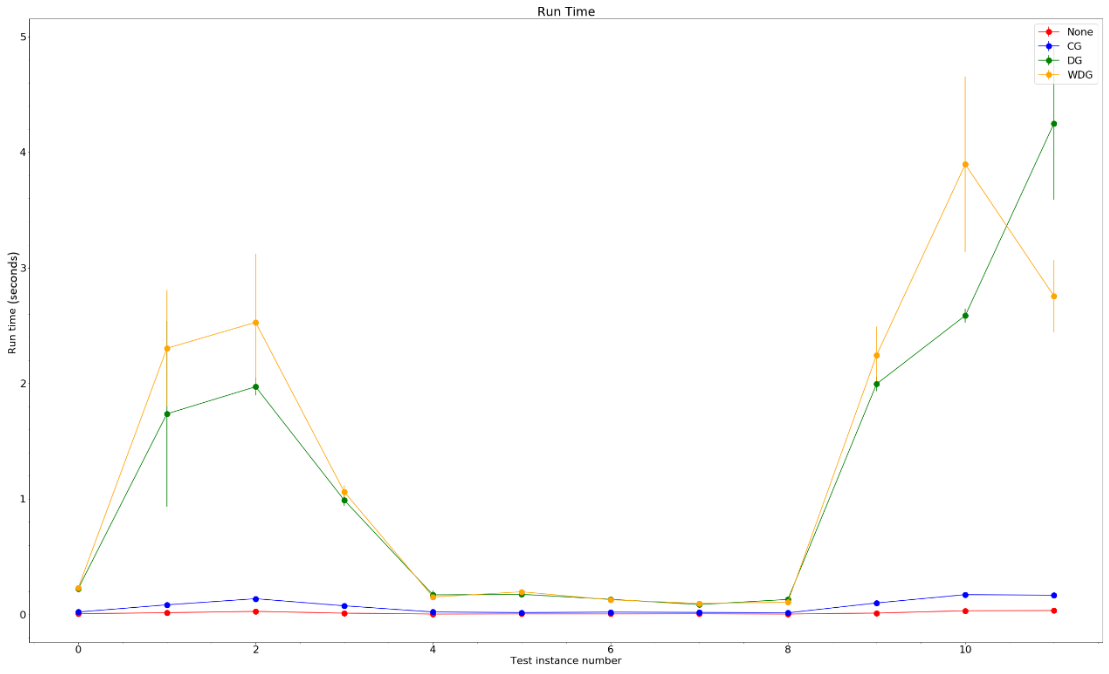

Figure 7: Large Map Run Time

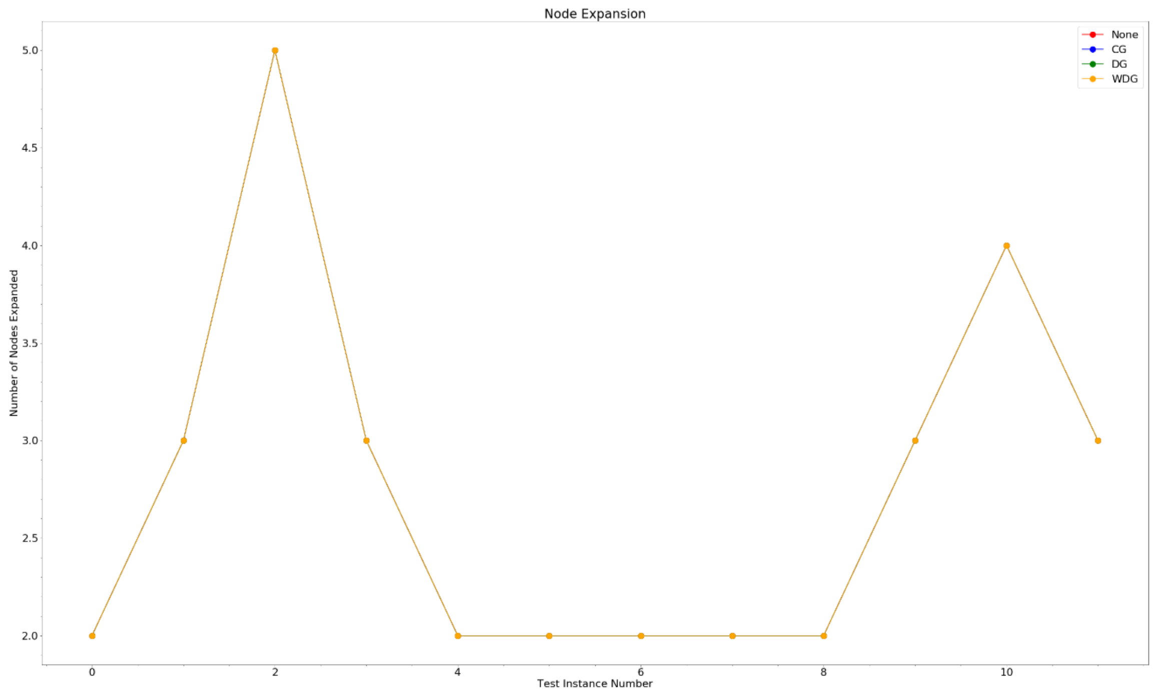

Figure 8: Large Map Node Expansion

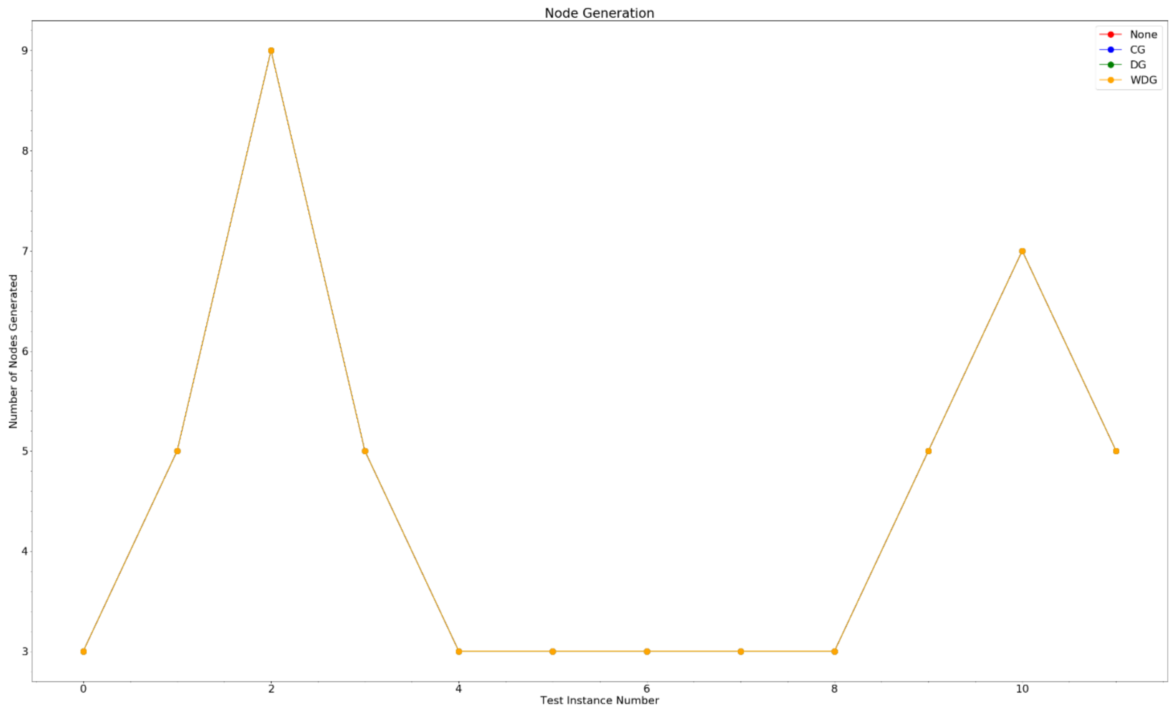

Figure 9: Large Map Node Generation

Conclusion

Based on our results, the more accurate heuristics have better results in the number of nodes expanded and explored. WDG expanding and exploring the least number of nodes, followed by DG, CG, then no heuristic. As for the runtime, we see that the less overhead a heuristic has, the faster a solution is found. We have no heuristic being generally the fastest, then CG, DG and WDG.

The broader take away for this experiment is that the heuristics reduce the number of nodes expanded and explored by the algorithm but when the overhead cost of those heuristics are too large it can increase the running time despite the reduction in nodes explored and expanded.

As for improvements that could be made towards this project, we could try to reduce the runtime of the DG and WDG implementations in some way. One method would be to implement a quicker minimum vertex cover (MVC) and edge weighted minimum vertex cover (EWMVC). The method we used was a brute force method, which greatly affected the runtime. Another extension for this project would be to find a different heuristic with less overhead than DG and WDG. As our results show, CG's runtime was relatively close to the non-heuristic compared to DG and WDG. This is most likely due to the overhead of the two heuristics. Another possible extension to this project would be to use a combination of heuristic. We noticed that the agent number affected the runtimes for DG and WDG. We then could use CG for higher number of agents and DG or WDG for lower number of agents.

References

[1] J. Li, A. Felner, E. Boyarski, H. Ma, and S. Koenig, "Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search," *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 2019. https://www2.cs.sfu.ca/~hangma/pub/ijcai19.pdf

[2] J. Li, A. Felner, E. Boyarski, H. Ma and S. Koenig, "Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search" 2019 http://jiaoyangli.me/files/slides/cbsh2.pdf