Anytime MAPF via Large Neighbourhood Search

Clare Dang
pdang@sfu.ca

Jasleen Phangara
jphangar@sfu.ca

Lei Gong
gongleig@sfu.ca

***Abstract*—Multi-Agent Path Finding (MAPF) is the problem of determining collision-free paths for moving multiple agents from given start locations to their respective goal locations, in a grid-like environment. There are two main categorizations of algorithms for solving MAPF: (bounded sub)optimal algorithms that find high-quality solutions but are inefficient for large problems, and unbounded suboptimal algorithms that find solutions for large problems efficiently but usually find low quality solutions. In this project, we will be studying an anytime approach for solving MAPF using Large Neighborhood Search (LNS), which attempts to combine the best aspects of the two different categories; that is, MAPF-LNS is able to compute initial solutions fast, find near-optimal solutions eventually, and scale to very large numbers of agents**

I. Introduction

Multi-Agent Path Finding (MAPF) is the problem of determining collision-free paths for moving multiple agents from given start locations to their respective goal locations, in a grid-like environment. Since MAPF is an NP-Hard problem, finding an optimal solution for a large enough MAPF problem can take an extensive amount of time and memory. Whereas, finding an optimal solution for a small enough MAPF problem can be done with a reasonable amount of time and memory.

Hence, algorithms for solving MAPF problems can be divided into two categories: (bounded sub)optimal algorithms and unbounded suboptimal algorithms. Bounded suboptimal optimal algorithms ensure that the cost of the solution returned is at most as large as $k \cdot \textit{opt-soln}$, where $k \geq 1$ and *opt-soln* denotes the cost of the optimal solution. However, such algorithms do not scale well for large numbers of agents as an intractable amount of resources (i.e., time and memory) are needed. In contrast, unbounded suboptimal algorithms scale well for large numbers of agents as planning can be done fast using predefined movement or agent ordering rules. But, the solution returned from such algorithms can be of low quality (i.e., far from the optimal solution cost) as there is no upper bound on the cost of the returned solution.

In this project, we explore an anytime approach for solving MAPF using Large Neighborhood Search (LNS), called MAPF-LNS[2], which attempts to combine the best aspects of the two different categories. That is, MAPF-LNS is fast, scalable, and near optimal since it is able to compute initial solutions quickly, scale to very large numbers of agents, and find near optimal solutions eventually (with enough time).

MAPF-LNS starts by finding an initial solution using an existing non-optimal MAPF algorithm. Next, LNS[4] is used to iteratively improve the quality of the solution. That is, for each iteration while time is available, a subset of the agents will be chosen to be replanned, holding the planned paths of the remaining agents constant. LNS uses a neighborhood selection heuristic to select a subset of agents with a cardinality equal to a given neighborhood size. A modified version of a MAPF algorithm is used to replan the subset of agents such that they do not collide with the constant paths, as well as each other. Once the paths for the selected subset of agents has been replanned, if the new paths minimize the cost compared to the old paths, then they will be added to the solution. The algorithm will repeat this process until it runs out of time.

We evaluate MAPF-LNS by assessing performance of the algorithm over various MAPF problem instances for increasing numbers of agents. In particular, we experiment with different neighborhood selection heuristics, neighborhood sizes, and time limits to determine how different factors impact performance of the algorithm.

II. Implementation

In this section, we outline the MAPF-LNS algorithm, as well as, the algorithms used for the three key phases of MAPF-LNS which are initial planning, neighborhood selection, and replanning.

A. MAPF-LNS

Given a MAPF instance, we first find an initial solution P using a MAPF algorithm. The algorithm used for finding the initial solution can be any suboptimal MAPF algorithm which is considered to be efficient. While time is available, we select a subset of k agents, say $A_k \subset A$, where A is a set containing all agents in the problem instance. Large neighborhood search is used to select this subset of k agents. Then, we remove the paths of the agents in A_k from P, and replan new paths for them by calling a modified MAPF algorithm. The modified algorithm will treat the remaining paths in P as constraints when planning the paths for the agents in A_k, such that the new paths for agents in A_k do not collide with the remaining paths in P as well as each other. We then compare the old paths to the new paths for the agents in A_k, and add the paths which minimize the solution cost to P. Lastly, we repeat this procedure until we run out of time. The resulting algorithm is called MAPF-LNS, and the pseudocode for MAPF-LNS can be found in algorithm 1.

B. Initial Planning: EECBS

For finding an initial solution we use Explicit Estimation Conflict-Based Search (EECBS)[1] which is a bounded sub-optimal search for MAPF. We choose to use EECBS as it

Algorithm 1 High-level MAPF-LNS

Input: MAPF Instance I, Set of Agents A, Neighborhood size k, Time limit t

```
1: $timer \leftarrow startTimer()$
2: $P \leftarrow findInitSoln(I, A)$
3: while $timer.timeElapsed() \leq t$ do
4:     $A_k \leftarrow selectNeighborhood(I, A, k)$
5:     $P_k^- \leftarrow \{P_{a_i} \in P \mid a_i \in A_k\}$
6:     $P \leftarrow P \setminus P_k^-$
7:     $P_k^+ \leftarrow replan(I, P, A_k)$
8:     if $cost(P_k^+) \leq cost(P_k^-)$ then
9:         $P \leftarrow P \cup P_k^+$
10:    else
11:        $P \leftarrow P \cup P_k^-$
12:    end if
13: end while
```

can find suboptimal solutions reasonably fast which is the ideal behavior we desire for initial planning in MAPF-LNS. Pseudocode for EECBS can be found in algorithm 2 which is taken from [1] as we will be using the author's existing implementation of EECBS. Since we could have chosen any suboptimal MAPF algorithm for initial planning, we omit the specific details of EECBS though we highlight a few noteworthy features.

EECBS is a variant of CBS which uses an inadmissible heuristic to estimate the cost of the solution of each unexpanded node, and uses Explicit Estimation Search (EES) to choose which unexpanded node to expand next. The heuristic is computed using an online learning method for learning the cost-to-go during search using the error experienced during node expansion [3]. An advantage of this heuristic is that it does not require preprocessing and allows for instance specific learning. In addition, [1] includes CBS improvements such as adding bypassing conflicts, prioritizing conflicts, symmetry reasoning, and using the Weighted Dependency Graph (WDG) heuristic to EECBS such that EECBS is able to significantly outperform comparable algorithms.

Algorithm 2 High-level search for EECBS

```
1: Generate root CT node $R$            ▷ CT = constraint tree
2: $computeWDGHeuristic(R)$
3: $pushNode(R)$
4: while $open$ is not empty do
5:     $P \leftarrow selectNode()$
6:     if $P.conflicts = \emptyset$ then
7:         return $P.paths$
8:     end if
9:     if $P$ is selected from $cleanUp$ and the WDG heuristic
       of $P$ has not been computed yet then
10:        $computeWDGHeuristic(P)$
11:        $pushNode(P)$
12:        continue
13:    end if
14:    $conflictPrioritization(P.conflicts)$
15:    $symmetryReasoning(P.conflicts)$
16:    $conflict \leftarrow chooseConflict(P.conflicts)$
17:    $constraints \leftarrow resolveConflict(conflict)$
18:    $children \leftarrow \emptyset$
19:    for $constraint \in constraints$ do
20:        $Q \leftarrow generateChild(P, constraint)$
21:        if $P$ is not selected from $cleanUp$ and
           $\forall i\, |Q.paths[i]| \leq w \cdot f^i_{min}(P)$ and $cost(Q) \leq w \cdot lb(best_{lb}$
           and $h_c(Q) < h_c(P)$ then
22:            $P.paths \leftarrow Q.paths$
23:            $P.conflicts \leftarrow Q.conflicts$
24:            Go to line 6
25:        end if
26:        Add $Q$ to $children$
27:    end for
28:    for $Q \in children$ do
29:        $pushNode(Q)$
30:    end for
31:    $updateOneStepErrors(P, children)$
32: end while
33: return ”No Solution”
```

C. Large Neighborhood Search

Large Neighborhood Search (LNS)[4] is a metaheuristic used for finding better solutions when solving an optimization problem by exploring neighborhoods to find subproblems to solve which can improve the solution to the original problem. In the context of MAPF-LNS, a neighborhood is defined to be a set of agents whose paths are removed from a given solution. So, each neighborhood corresponds to the subproblem of planning the paths for a subset of agents while holding the paths of the remaining agents constant. In algorithm 1, lines 4-12 implement LNS in MAPF-LNS.

More generally, LNS begins by selecting a neighborhood of the problem to solve, assuming that there is an existing solution to the problem (not necessarily optimal). Then, the subproblem corresponding to the chosen neighborhood is solved, producing a new solution to the problem. If the new solution is better than the old solution (i.e., maximizes/minimizes the solution), then the new solution is kept, otherwise the old solution is kept. This process can be repeated several times as the solution is guaranteed to be no worse than the initial solution we started with. Moreover, the quality of LNS is dependent on the neighborhood selection method used. Hence, we discuss the neighborhood selection methods considered in our project next.

D. Agent-Based Neighborhood Selection

Agent-based neighborhood selection[2] selects an agent whose path could be shorter if some other agents were not blocking its way, as replanning them together has a chance to reduce the overall costs of their paths. In algorithm 3, we have described the pseudocode for this selection method. First, an agent which is not in the tabu list is chosen, say a_j, such that the agent has the largest delay. The tabu list is a globally maintained set which is initially empty but is populated as the iterations of LNS proceed such that the same agent won't be repeatedly chosen for the initial agent in agent based neighborhood selection. In addition, the delay is defined to be the difference between the length of an agent's path and the distance between the agent's start and goal locations. So, a large delay indicates that an agent waits or is blocked by many other agents when traversing its path.

Next, the neighborhood of agents, A_k, is initialized to the selected agent a_j. Then, while A_k contains less than k agents, agent a_j will perform a restricted random walk to find agents that prevent it from reaching its target location at an earlier point. The agents found in this walk will be added to A_k, until the walk has terminated or A_k has k agents. Then, a new agent will randomly be chosen from A_k to be a_j and the process repeats. The details of the random walk procedure are omitted as it is beyond the scope of this project.

Algorithm 3 Agent-Based Neighborhood Selection

```
Input: MAPF Instance $I$, Paths of Agents $P = \{P_1, \ldots, P_n\}$, Set of Agents $A$, Neighborhood Size $k$, Tabu List $tabuList$
1: $a_j \leftarrow max\{delay(P_i) \mid a_i \in A \setminus tabuList\}$
2: $tabuList \leftarrow tabuList \cup \{a_j\}$
3: if $tabuList.size() = n$ or $delay(P_j) = 0$ then
4:     $tabuList \leftarrow \emptyset$
5: end if
6: $A_k \leftarrow \{a_j\}$
7: while $A_k.size() < k$ do
8:     $randomWalk(I, P, a_j, A_k)$
9:     $a_j \leftarrow$ random vertex in $A_k$
10: end while
11: return $A_k$
```

E. Map-Based Neighborhood Selection

Map-based neighborhood selection[2] selects agents that visit the same intersection vertex, i.e., a location on the map which is traversed by more than one agent, as changing the order in which the agents traverse an intersection vertex could lead to different solution costs. In algorithm 4, we show the pseudocode for this selection method. First, we determine all the intersection vertices using the paths of the agents, and randomly select one of the intersection vertices, say v. Then, a queue is initialized containing v and the neighborhood of agents, A_k, is initialized to the empty set.

Next, while the queue is not empty and A_k contains less than k agents, we pop an element from the queue and set v to this element. Then, if v is an intersection vertex, we add the agents which visit v to A_k. Lastly, the vertices adjacent to v are added to the queue, if the location exists in the map and it has not previously been in the queue. This procedure repeats until A_k contains k agents or every location in the map has been visited (i.e., the queue is empty).

Algorithm 4 Map-Based Neighborhood Selection

```
Input: MAPF Instance $I$, Paths of Agents $P = \{P_1, \ldots, P_n\}$, Set of Agents $A$, Neighborhood Size $k$
1: $V \leftarrow findIntersections(P)$
2: $v \leftarrow$ random vertex in $V$
3: $Q \leftarrow initQueue()$
4: $Q.push(v)$
5: $A_k \leftarrow \emptyset$
6: while $!Q.isEmpty()$ and $A_k.size() < k$ do
7:     $v \leftarrow Q.pop()$
8:     if $isIntersection(v)$ then
9:         $getIntersectionAgents(v, P, A_k)$
10:    end if
11:    for $d \in \{$right, left, up, down$\}$ do
12:        $u \leftarrow move(v, d)$
13:        if $u \in I.map$ and $u$ hasn't been visited before then
14:            $Q.push(u)$
15:        end if
16:    end for
17: end while
18: return $A_k$
```

F. Random Neighborhood Selection

Random neighborhood selection[2] selects k agents uniformly at random, where k is the neighborhood size. Algorithm 5 contains pseudocode for this selection method. Note that the randomizer used in algorithm 5 reorders the agents such that each possible permutation of the agents has equal probability of appearance.

Algorithm 5 Random Neighborhood Selection

```
Input: Set of Agents $A$, Neighborhood Size $k$
1: $L \leftarrow A.toList()$
2: $randomize(L)$
3: return $L[0 : k - 1]$
```

G. Adaptive LNS

Adaptive LNS[6] is a stronger variant of LNS which makes use of multiple neighborhood selection heuristics by recording their relative success in improving the current solution and choosing the next neighborhood guided by the most promising heuristic. In our context, adaptive LNS will be referred to as adaptive neighborhood selection as the adaptive method will choose the most promising neighborhood selection method between agent-based, map-based, and random selection for a given subproblem. For instance, the adaptive method may choose agent-based selection for one iteration of LNS and random selection for another.

Algorithm 6 contains pseudocode for the adaptive method which is a modification of algorithm 1. The adaptive method maintains a list of weights where each weight corresponds to a neighborhood selection method, and the list is initialized to 1 for all entries. Then, in each iteration of LNS, a neighborhood selection method will be chosen with probability

$$P(\textit{method i chosen}) = \frac{\textit{weight of method i}}{\textit{sum of all weights}}.$$

So, the larger the weight, the higher the probability that a method will be chosen. Once the method is chosen and the new paths have been found, the weight for the chosen method is updated according to how much the new paths improve the solution quality (i.e., reduce the cost of the solution).

Algorithm 6 MAPF-LNS with Adaptive LNS

```
Input: MAPF Instance $I$, Set of Agents $A$, Neighborhood size $k$, Time limit $t$
 1: $timer \leftarrow startTimer()$
 2: $P \leftarrow findInitSoln(I, A)$
 3: $W \leftarrow initWeights()$
 4: while $timer.timeElapsed() \leq t$ do
 5:     $method \leftarrow selectMethod(W)$
 6:     $A_k \leftarrow selectNeighborhood(I, A, k, method)$
 7:     $P_k^- \leftarrow \{P_{a_i} \in P \mid a_i \in A_k\}$
 8:     $P \leftarrow P \setminus P_k^-$
 9:     $P_k^+ \leftarrow replan(I, P, A_k)$
10:     if $cost(P_k^+) \leq cost(P_k^-)$ then
11:         $P \leftarrow P \cup P_k^+$
12:     else
13:         $P \leftarrow P \cup P_k^-$
14:     end if
15:     $updateWeights(W, P_k^-, P_k^+, method)$
16: end while
```

H. Naive Neighborhood Selection

In addition to the prior methods, we include two naive methods for neighborhood selection. The first method is to select a consecutive subset of k agents from a circular array of the agents, ordered lexicographically. For instance, suppose the agents are labeled from $1, \ldots, n$, then on iteration j we pick agents $a_j, \ldots, a_{j+k}$ to be in our neighborhood, where the indices are taken modulo n. The pseudocode for this naive method is shown in algorithm 7 which is a modified version of algorithm 1.

The second naive method is to select the k agents which have the highest cost paths, i.e., the k agents with the longest paths. The pseudocode for this naive method is shown in algorithm 8.

I. Replanning: Prioritized Planning

Lastly, we discuss the modified MAPF algorithm used for replanning the paths of the agents in the neighborhood. We use a modified version of prioritized planning with a random priority ordering for replanning paths. That is, the agent ordering is randomized, and the existing paths P are added to the constraint set before prioritized planning proceeds as expected. The pseudocode for replanning using prioritized planning is shown in algorithm 9, where we assume the reader is familiar with prioritized planning.

Algorithm 7 MAPF-LNS with Naive Selection

```
Input: MAPF Instance $I$, Circular Array of Agents $A$, Neighborhood size $k$, Time limit $t$, Number of Agents $n$
 1: $timer \leftarrow startTimer()$
 2: $P \leftarrow findInitSoln(I, A)$
 3: $j \leftarrow 0$
 4: while $timer.timeElapsed() \leq t$ do
 5:     $A_k \leftarrow A.getSubarray(j, j + k)$
 6:     $P_k^- \leftarrow \{P_{a_i} \in P \mid a_i \in A_k\}$
 7:     $P \leftarrow P \setminus P_k^-$
 8:     $P_k^+ \leftarrow replan(I, P, A_k)$
 9:     if $cost(P_k^+) \leq cost(P_k^-)$ then
10:         $P \leftarrow P \cup P_k^+$
11:     else
12:         $P \leftarrow P \cup P_k^-$
13:     end if
14:     $j = j + 1$
15: end while
```

Algorithm 8 High Cost Neighborhood Selection

```
Input: Set of Agents $A$, Neighborhood Size $k$, Paths of Agents $P = \{P_1, \ldots, P_n\}$
 1: $P' \leftarrow findHighCostPaths(P, k)$
 2: $A_k \leftarrow \{a_i \in A \mid P_i \in P'\}$
 3: return $A_k$
```

III. Methodology

We use five problem instances from the MAPF benchmark[5] instances, namely, random-32-32-20 of size 32x32, room-32-32-4 of size 32x32, den312d of size 65x81, ht_mansion_n of size 133x270, and Boston_0_256 of size 256x256. These instances are chosen as they provide a variety of map difficulty ranging from easy to hard for solving problem instances. In addition, we use one random scenario per problem instance for generating results as using all 25 random scenarios (or some non-trivial subset) per problem instance for generating results in each experiment causes the runtime of the experiments to be in the multitude of hours. Note that the random scenario picked per problem instance is consistent through the entire experiment. In addition, we use 50, 100, and 150 agents for varying the number of agents in each experiment; that is, each experiment executes once for each number of agents, totalling to three runs.

Since MAPF-LNS is an anytime algorithm, performance cannot be measured in terms of runtime as the runtime is set

Algorithm 9 High-level Modified Prioritized Planning

```
Input: MAPF Instance $I$, List of Agents in Neighborhood $A_k$, Paths of Agents $P = \{P_1, \ldots, P_{n-k}\}$
 1: $randomize(A_k)$
 2: $constraints \leftarrow buildConstraints(P)$
 3: return $prioritizedPlanning(I, A_k, constraints)$
```

by the user. Instead, we measure performance of MAPF-LNS as the improvement of the solution quality defined to be the percent of cost decreased from the initial solution to the final solution. That is, the amount the solution cost decreased from the initial solution to the final solution as a percentage. We evaluate MAPF-LNS through the following four experiments.

A. *Experiment 1: Neighborhood Selection*

In experiment 1, we determine how the choice of the neighborhood selection heuristic impacts performance of MAPF-LNS. We compare performance of MAPF-LNS using agent-based, map-based, random, and adaptive neighborhood selection over the selected problem instances. In particular, for each problem instance, for each neighborhood selection heuristic we use a time limit of 30 seconds and a neighborhood size of 5, over varying numbers of agents. Hence, we can determine which neighborhood selection heuristic results in the highest overall performance of MAPF-LNS, as well as, determining if certain problem instances are better suited for a different neighborhood selection heuristic.

B. *Experiment 2: Neighborhood Size*

In experiment 2, we evaluate how the neighborhood size impacts performance of MAPF-LNS. We compare performance of MAPF-LNS using neighborhood sizes of 2, 4, 8, and 16 over the selected problem instances. In particular, for each problem instance, for each neighborhood size we use a time limit of 30 seconds and adaptive neighborhood selection, over varying numbers of agents. Thus, we can determine if there is an optimal neighborhood size which results in the highest overall performance of MAPF-LNS, as well as, determining if certain problem instances are better suited for a different neighborhood size.

C. *Experiment 3: Time Limit*

In experiment 3, we assess how increasing the time limit impacts the performance of MAPF-LNS. We compare performance of MAPF-LNS using time limits of 15, 30, 45, and 60 seconds over the selected problem instances. In particular, for each problem instance, for each time limit we use a neighborhood size of 5 and adaptive neighborhood selection, over varying numbers of agents. Therefore, we can determine whether increasing the time limit improves performance of MAPF-LNS by a significant amount. In addition, we can determine which problem instances showed the most improvement, meaning that MAPF-LNS is well suited for these instances.

D. *Experiment 4: Naive Neighborhood Selection*

In experiment 4, we assess how naive neighborhood selection heuristics compare to sophisticated neighborhood selection heuristics using performance impact on MAPF-LNS. We compare performance of MAPF-LNS using the first naive approach, high cost naive approach, and adaptive neighborhood selection over the selected problem instances. In particular, for each problem instance, for each neighborhood selection heuristic we use a time limit of 30 seconds and neighborhood size 16, over varying numbers of agents. Hence, we can determine if naive heuristics improve performance of MAPF-LNS by an amount comparable to a sophisticated heuristic. In addition, we can determine if there are problem instances which are better suited for a naive heuristic than a sophisticated heuristic.

IV. Experimental Setup

We use the existing MAPF-LNS implementation which is available at https://github.com/Jiaoyang-Li/MAPF-LNS and we modify the implementation to include the naive neighborhood selection heuristics. We use the existing implementation because implementing MAPF-LNS requires multiple complicated algorithms to be implemented as subroutines, which all take significant amounts of time to implement. All parts of MAPF-LNS are implemented in C++11. In addition, we implement launch scripts using Python 3.8 for invoking each of the previously described experiments. The experiments are run on macOS 10.15.2 with Intel i5-8210Y (1.6 GHz) and 16GB memory available.

V. Results

Note that in the following plots and discussion, the random walk method is the agent-based selection method and the intersection method is the map-based selection method.

A. *Experiment 1: Neighborhood Selection*

In figure 1, we have plotted the performance improvement (%) of each problem instance for each neighborhood selection heuristic over different numbers of agents. First, we notice that for larger problem instances, such as Boston and ht_mansion, the magnitude of improvement is quite small. This is because we used a 30 second time limit and neighborhood size of 5 which might not be well suited for larger instances as they could require larger neighborhoods and more time for finding a significantly better solution. In addition, we notice that the performance improvement seems to increase as the number of agents increases, which indicates that MAPF-LNS is able to perform well as the number of agents grows.

Overall, we observe that the adaptive method appears to be the best as, on average, this method produces the largest performance improvement. Also, we notice that there are instances where the performance improvement from the adaptive method is less than one of the other methods. This occurs when the performance improvement from the random, random walk, and intersection methods are equivalent or close together in value. For instance, in figure 1, for the random instance with 100 agents, we see that the adaptive method improves performance slightly less than the intersection method, but we also see that all four methods improve performance by similar amounts as the values are relatively close together.

B. *Experiment 2: Neighborhood Size*

In figure 2, we have plotted the performance improvement (%) of each problem instance for each neighborhood size over different numbers of agents. We observe that on average

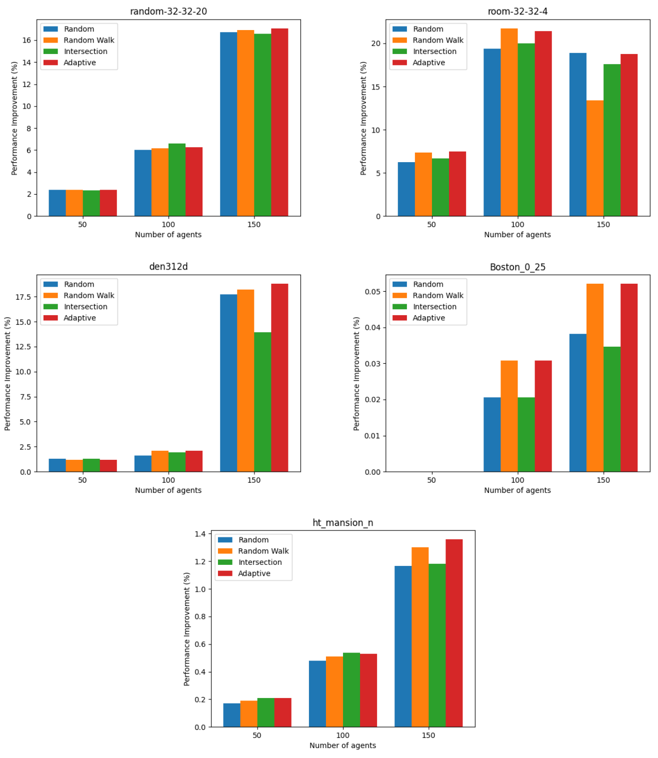

Fig. 1. Performance improvement for varying neighborhood selection methods.

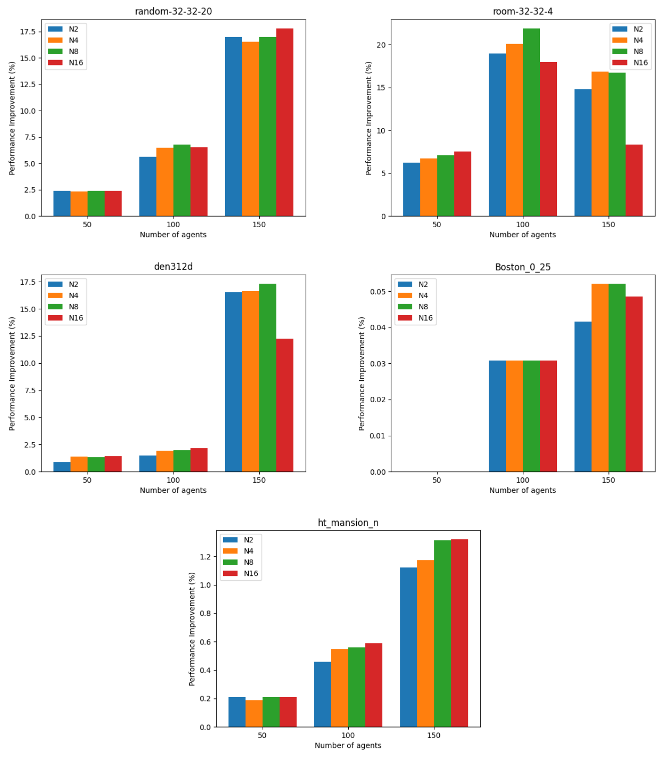

Fig. 2. Performance improvement for varying neighborhood sizes.

it appears that a neighborhood size of 8 seems to be the best from our results, however there does not appear to be an overall trend over all problem instances for neighborhood size. In addition, we should not regard 8 as an optimal neighborhood size as we see from the results that each problem instance has specific trends for neighborhood size indicating that neighborhood size should be determined on a case by case basis.

In particular, we see that ht_mansion and random have higher performance improvement on average for neighborhood size of 16. In addition, we observe that there are instances where the performance improvement is relatively similar for all neighborhood sizes, meaning that for such instances we should choose the smallest neighborhood size as the heuristic will compute faster and more iterations will be able to compute within the time limit. For instance, in figure 2 for the Boston instance with 100 agents, we see that the neighborhood sizes all produce similar performance improvement, and so we should use a neighborhood size of 2 for this configuration of the problem.

C. *Experiment 3: Time Limit*

In figure 3, we have plotted the reduction of the solution cost of each problem instance for each time limit over different numbers of agents. That is, we have plotted the difference between the initial and final solution cost. We observe that for large numbers of agents, increasing the time limit does appear to improve the quality of the solution as the solution cost is reduced by a larger amount. However, for small numbers of agents, increasing the time limit does not appear to improve the solution quality by a significant amount. Hence, we conclude that the time limit required for finding a good solution depends on the number of agents in the problem. In addition, we conclude that on average increasing the time limit does improve the quality of the solution.

D. *Experiment 4: Naive Neighborhood Selection*

In figure 4, we have plotted the performance improvement (%) of each problem instance for each selection method (First, High Cost, and Adaptive) over different numbers of agents. We observe that the naive methods do not perform comparably to the adaptive method, as figure 4 depicts that the performance improvement from using a naive method never exceeds the performance improvement from using the adaptive method. So, naive methods do not outperform sophisticated methods nor is their performance relatively close to a sophisticated method. In addition, it does not appear that any of our problem instances are better suited for a naive method as the performance improvement from the adaptive method is larger.

VI. Conclusion

In this project, we discussed MAPF-LNS and experimented with different factors which impact the performance of the algorithm over various problem instances. We concluded that the adaptive neighborhood selection method led to the most performance improvement of MAPF-LNS over varying numbers of agents. In particular, the adaptive method appears to

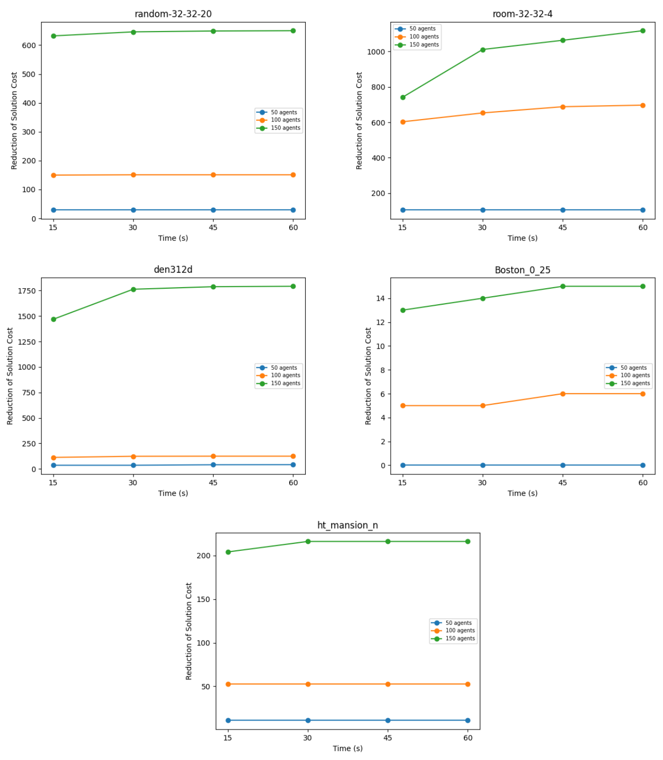

Fig. 3. Reduction of solution cost for varying time limits.

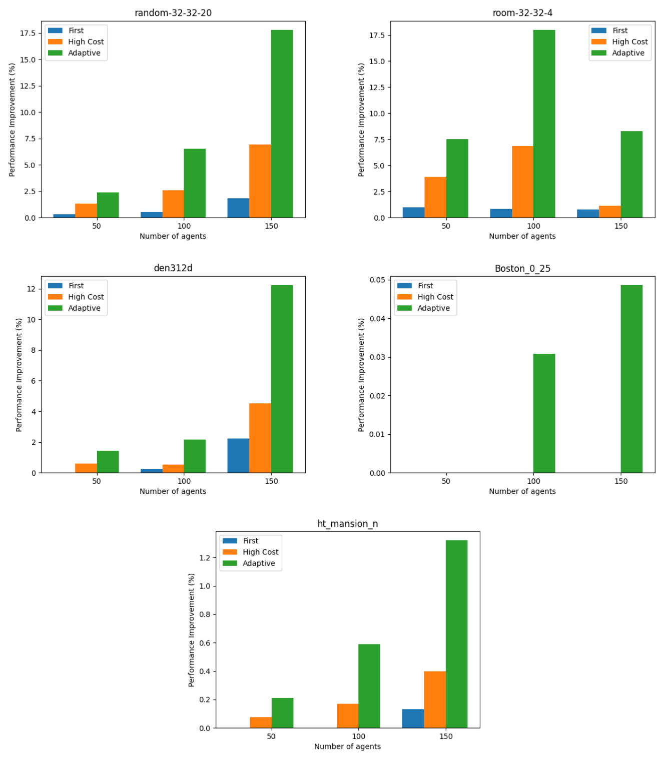

Fig. 4. Performance improvement for naive and adaptive selection methods.

always be the best for large numbers of agents. In addition, we concluded that a neighborhood size of about 8 appears to lead to the most performance improvement of MAPF-LNS over varying numbers of agents. Notably, the fluctuation in performance caused by varying the neighborhood size is small relative to the fluctuation in performance caused by varying the neighborhood selection method. Hence, we conclude that the choice of the neighborhood selection method has a greater impact on performance compared to the choice of the neighborhood size.

Additionally, we concluded that longer time limits improve the quality of the solution found by MAPF-LNS. Hence, it is recommended that a longer time limit be used when possible, though the solution found by MAPF-LNS with a short time limit could still be sufficient (depending on the difficulty of the problem instance and number of agents). Lastly, we conclude that naive methods for neighborhood selection are not comparable to sophisticated methods as the naive methods perform very poorly compared to the sophisticated methods.

References

[1] J. Li, W. Ruml, S. Koenig. EECBS: Bounded-suboptimal search for multi-agent path finding. In AAAI, pp. 12353–12362, 2021.

[2] J. Li, Z. Chen, D. Harabor, P. J. Stuckey, S. Koenig. Anytime Multi-Agent Path Finding via Large Neighborhood Search. In IJCAI, pp. 4127-4135, 2021.

[3] J. T. Thayer, A. J. Dionne, W. Ruml. Learning Inadmissible Heuristics During Search. In ICAPS, pp. 250–257, 2011.

[4] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In CP, pp. 417–431, 1998.

[5] R. Stern, N. R. Sturtevant, A. Felner, S. Koenig, H. Ma, T. Walker, J. Li, D. Atzmon, L. Cohen, T. K. S. Kumar, E. Boyarski, R. Bartak. Multi-agent pathfinding: Definitions, variants, and benchmarks. In SoCS, pp. 151–159, 2019.

[6] S. Ropke D. Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Science, 40(4), pp. 455–472, 2006.