Project Report

CMPT 417

Instructor: Hang Ma

Project Team

Zhonghong Zhang, zkzhang@sfu.ca
Shenyu Gu, shenyug@sfu.ca
Cheng Zhang, cza111@sfu.ca

GitHub URL

https://github.com/gushenyu1998/MAPF-LNS2-CMPT417

Environment

We are using python to implement the project with the version 3.9 on the Linux System. The processor of the System is Intel i7 11th processor, and there are 16 GB Ram is available.

General Architecture

Multi-Agent Path Finding is a problem of searching collision-dress routes for multiple agents. In the project we are going to use MAPF-LSN2, which is based on local search, to solve the problem. The MAPF-LSN2 is started from a collision path, and recreates the partial collision path until there are no more collisions in the path. In the project, we are going to create MAPF-LNS2 with different methods, like SIPPS and neighbourhood selection to improve the Multi-Agent Path Finding problem. In the final part of the project, we will compare the MAPF-LSN2 that we created with the existing method, for example, Collision Base Search. It is necessary to show the efficiency of the MAPF-LNS2.

MAPF-LSN2:

LSN is a solution which starts with a given solution and destroys part of the solution. At this time, the solution will try to find some path which starts and ends at the same position as the destroyed part. We will call the paths neighbour. The destroyed part will be replaced if a neighbour is better than the destroyed part. MAPF-LSN2 will call a MAPF algorithm to solve the problem first and return a path P even if the path contains any collisions. MAPF-LSN2 will reorganize the path P until the P is feasible. In every iteration, MAPF-LSN2 removes a part of the path from P and replans the path for the agent to minimize the number of collisions.

SIPPS:

SIPPS is the Time-space A* with time intervals acting as the time stamp for every node in the A* search. Under SIPPS, we resolve the single-agent pathfinding more efficiently. And SIPPS also provides important information on the Neighborhood selection in the following.

We assign SIPPS node n = {vertex, safe interval, is_goal, safe interval id, g, h, c, parent p}.

There are two types of obstacles in the SIPPS. Hard obstacles Oh means obstacles are occupied since [0,∞]. And the soft obstacles Os means obstacles are occupied since [a,b], where 0<a<b<∞. If there are no hard or soft obstacles in a specific time interval, we call it a safe interval and assign it as [n.low, n.high). The safe interval [n.low, n.high) will be saved in the i th safe interval at the safe interval T[v], [n.low, n.high) $\subseteq$ T[v][i]. The c value in the SIPPS node means the number of soft collisions of the partial path from the root node to node n. c(n) = c'(n) + cv + ce, if n is collided with a soft obstacle, then make cv = 1. ce is also calculated in the same way. SIPPS guarantees to return a path if one exists and "No solution" otherwise. And it also guarantees to return a shortest path with zero soft collisions if one exists.

Algorithm of SIPPS:

Once we start the SIPPS, we need to build a safe interval table to save the safe interval of the MAPF. Only the hard obstacles will be inserted into the table, and the soft obstacle is null for the single agent planner. Line 4 is used to avoid the goal being occupied by hard obstacles. Lines 10 to 15 are created for a special

```
1  T ← buildSafeIntervalTable(V, O^h, O^s);
2  root ← Node(s, T[s][1], 1);                    // 1 is the index
3  T ← 0;                                         // Lower bound on travel time
4  if ∃t : (g, t) ∈ O^h then T ← max{t | (g, t) ∈ O^h} + 1;
5  compute g-, h-, f-, and c-values of root;
6  Q ← {root}, P ← ∅;                             // Initialize open and closed lists
7  while Q is not empty do
8      n ← Q.pop();                               // Node with the smallest c-value
9      if n.is_goal then return extractPath(n);
10     if n.v = g ∧ n.low ≥ T then
11         c_future ← |{(g, t) ∈ O^s | t > n.low}|;
12         if c_future = 0 then return extractPath(n);
13         n' ← a copy of n with is_goal set to true;
14         c(n') ← c(n') + c_future;
15         INSERTNODE(n', Q, P);                  // Algorithm 3
16     EXPANDNODE(n, Q, P, T);                    // Algorithm 2
17     P ← P ∪ {n};
18 return "No Solution";
```

case. The node n was arriving at the goal before the lowest estimated time. It must wait a little while and avoid possible soft collision when the agent stays at the goal. Then it follows the expand and insert node as Skeleton of search.

After we get the node with the smallest c-value from the open list, we will expand the node in the ExpandNode algorithm. In Line2-3, it inserts the move action to the search queue. We will indicate the wait action to queue in lines 4-5. Line 8, the 'low' is in range of the [low, high) which is extracted from the safe interval form. If v is an obstacle, then discard the node. For the lines 11-18, when 'low' exists, it still needs to avoid the soft collision and insert the node into the open list. Also, the wait action still needs to be indicated.

Algorithm 2: EXPANDNODE$(n, Q, P, \mathcal{T})$

1 $\mathcal{I} \leftarrow \emptyset$;
2 **foreach** $v : (n.v, v) \in E$ **do**
3 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{(v, id) \mid \mathcal{T}[v][id] \cap [n.low + 1, n.high + 1) \neq \emptyset, id \in \mathbb{N}\}$;
4 **if** $\exists id : \mathcal{T}[n.v][id].low = n.high$ **then**
5 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{(n.v, id)\}$; *// Indicates wait actions*
6 **foreach** $(v, id) \in \mathcal{I}$ **do**
7 $\quad [low, high) \leftarrow \mathcal{T}[v][id]$;
8 $\quad low \leftarrow$ earliest arrival time at v within $[low, high)$ without colliding with edge obstacles in $\mathcal{O}^h$;
9 $\quad$ **if** *low does not exist* **then continue**;
10 $\quad low' \leftarrow$ earliest arrival time at v within $[low, high)$ without colliding with edge obstacles in $\mathcal{O}^h \cup \mathcal{O}^s$;
11 $\quad$ **if** *low' exists* $\wedge low' > low$ **then**
12 $\quad\quad n_1 \leftarrow Node(v, [low, low'), id)$;
13 $\quad\quad$ INSERTNODE(n_1, Q, P) ; *// Algorithm 3*
14 $\quad\quad n_2 \leftarrow Node(v, [low', high), id)$;
15 $\quad\quad$ INSERTNODE(n_2, Q, P); *// Algorithm 3*
16 $\quad$ **else**
17 $\quad\quad n_1 \leftarrow Node(v, [low, high), id)$;
18 $\quad\quad$ INSERTNODE(n_2, Q, P); *// Algorithm 3*

After expanding the node, we need to insert the node into the open list in the InsertNode algorithm. Before it inserts the node to the open list Q, it has to clean up the q (q has the same vertex as n) in the open list and close list (Relax procession and reopen the node in the close list). The c-value here is used to relax.This algorithm prefers the less c-value node.

Algorithm 3: INSERTNODE(n, Q, P)

1 compute g-, h-, f-, and c-values of n;
2 $\mathcal{N} \leftarrow \{q \in Q \cup P \mid q \sim n\}$; *// Nodes identical to n*
3 **foreach** $q \in \mathcal{N}$ **do**
4 $\quad$ **if** $q.low \leq n.low \wedge c(q) \leq c(n)$ **then** *// $q \succeq n$*
5 $\quad\quad$ **return**; *// No need to generate n*
6 $\quad$ **else if** $n.low \leq q.low \wedge c(n) \leq c(q)$ **then** *// $n \succeq q$*
7 $\quad\quad$ delete q from Q or P; *// Prune q*
8 $\quad$ **else if** $n.low < q.high \wedge q.low < n.high$ **then**
9 $\quad\quad$ **if** $n.low < q.low$ **then** $n.high = q.low$;
10 $\quad\quad$ **else** $q.high = n.low$;
11 insert n to Q;

Neighborhood selection:

A good neighborhood is important for the LNS algorithm. There are two methods for choosing the neighborhood in the MAPF-LNS2 algorithm in the project. They are Collision-based neighborhoods and Failure-based neighborhoods. Different methods have different motivations. Assume the current path as P, and the neighborhood as As. The size of the neighborhood will be limited to N. Assume Gc = (Vc,Ec) is the collision path, Vc = {i | ai ∈ A} and Ec = {(pi,pj) | i collide with j).

Collision-based neighborhoods generate neighborhoods that can potentially reduce CP by selecting a subset whose path collides with each other. It will choose a vertex with v, which is deg(v) > 0, randomly and find the largest component Gc' = (Vc', Ec') with v. The neighborhood will be filled with the agent, which will be collided with the vertex v.

```
Collision-Based Neighborhoods(As, Gc):
V = random(Vc)
Vc', Ec' = get_largest_connected_component(Gc, v)
If |Vc'| < len(A):
        As.append(av, v ∈ Vc')
        while |As| < N:
                agent = random(av)
                path = random_walk(agent)
                If detect_collision(path) is True:
                        As.append(agent)
                else:
                        Path = random_walk
```

Because the collision-based neighborhood selection is not built on the low-level algorithm, the speed of calculating and finding a path can be increased. It uses a random walk instead of the h-value and g-value when looking for an

available neighborhood, so the path result may not be an optimal solution for the MAPF problem.

Besides using collisions to assign constraints to the agent, there is another idea of using failure cases to create neighborhoods. In these cases, we firstly have a node that “has no solution of finding a collision-free path”. The failure-based neighbourhood selection focuses on how the failure and two conditions probably caused this. The first one is an agent’s goal was taken by other agents, or the position surrounding the goal was taken. The other one is an agent has no choice but to go to the position except to swap the position with some agents else. These conditions correspond to the vertex collision and edge collision constraints with a different way to approach that. There are some similar points of how the collision happens. When a collision happens, an agent’s path should come across the start or goal of the other agents.

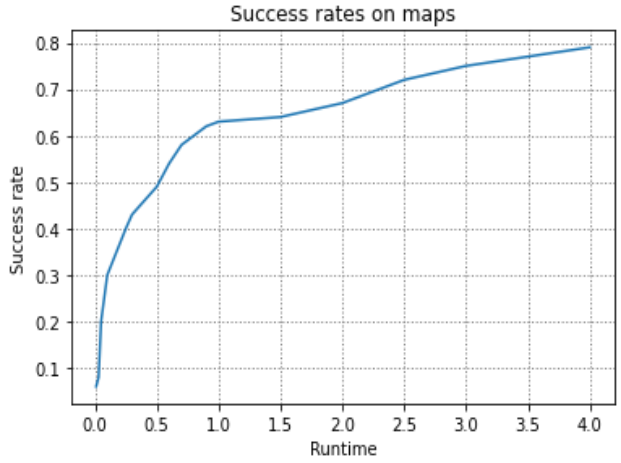
```
Failure-Based Neighborhoods(As, Ag):
//As= agents whose path visit a’ start, Ag = agent whose path visit a’ goal
        if |As ∪ Ag| = 0:
                Let a wait until other agents get their goal then execute moving
        else if |As ∪ Ag| < N-1:
                Add |As ∪ Ag| to Neighborhood (Anei) of the Current plan
                Add agents who visit the start/goal points of ai ∈ |As ∪ Ag| until Anei = N
                When create a neighborhood of Anei, a is extracted from Anei and the agents in
Anei whose target is visit by A
        else:
                If As = 0:
                        Add random agent in Ag to Anei until Anei = N
                Else if Ag > N-1:
                        Add earliest agent in As and  random agents in Ag to Anei until Anei = N
                else:
                        Add all agents Ag and agents As to Anei until Anei = N
```

The neighborhood Search doesn’t need a low-level algorithm to raise the calculation speed. Compared to the priority-based search, the neighbour search

can help to avoid some no-solution issues. However, the neighbor search can not provide an optimized search and make some agents wait for a long time until other agents reach the goal. Also, the neighbor search algorithm highly relies on a random number and a pre-defined parameter N, both Collision-Based Neighbourhood and Failure-Based Neighbourhood. The quality of search results is affected by fortune.

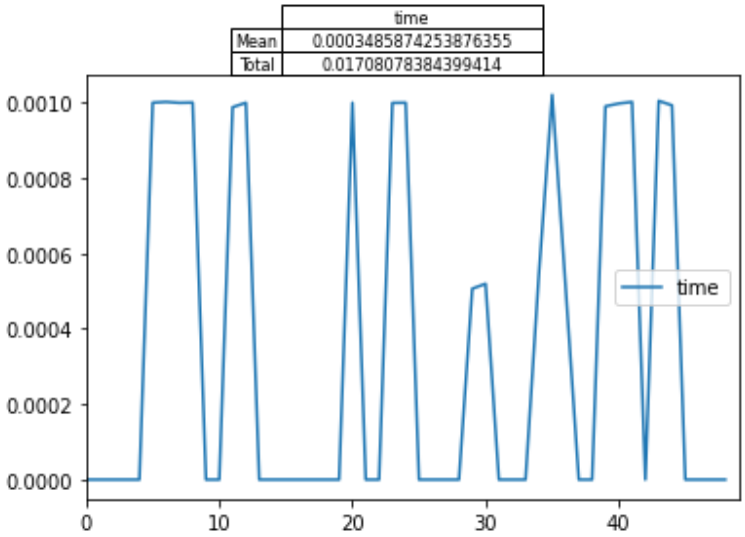
Experiment Result

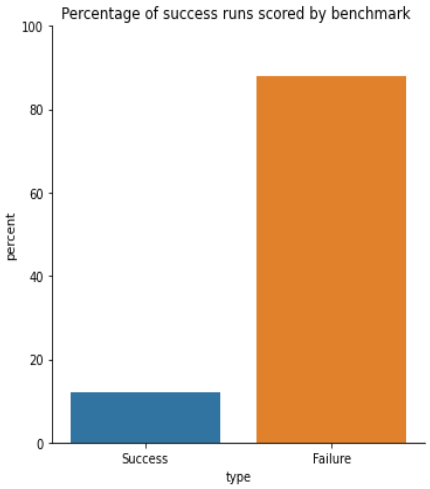
MAPF-LSN2 success rates on maps:



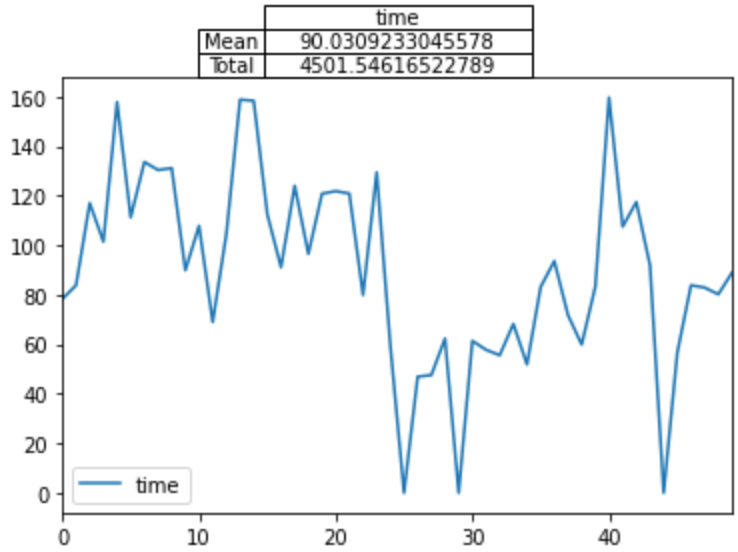
From the figure, we can find that the success rates boost significantly before 1 minute that it solves more than 60% of instances in this time interval. After that, 80% of instances will be solved within 4 minutes. Failure cases of MAPF-LNS2 occur when we encounter highly congested maps, such as room maps. Also, it is interesting that the memory usage of MAPF-LSN2 is relatively small.

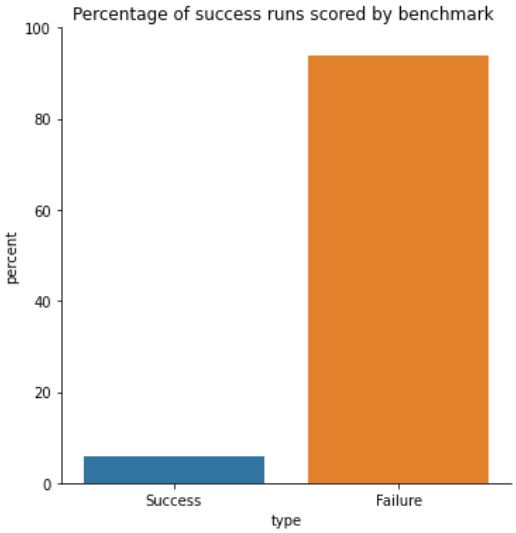
Randomized Conflicted-base search result:





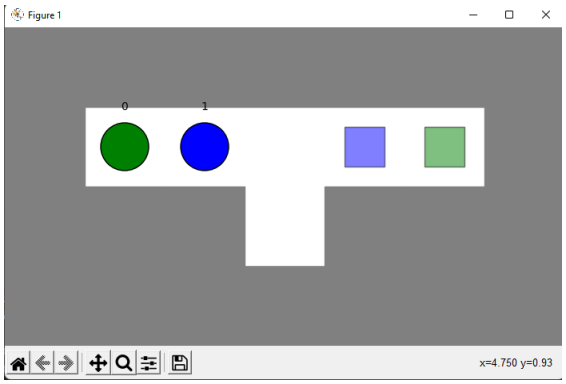
Randomized Priority-base search result:





Discussion:

Collision Based Neighbour Selection:



In the experiment of the Collision based Neighbour selection, we found the success rate is not high. For the benchmark cases, the rate of find an collision path is lower than 20% in one search. So we have a discusion of why the successful rate is low.

In the example, the agent 0 and agent 1 need to exchange there place to get to the goal with out collision. But the Collision-Base Neighbour Select is most

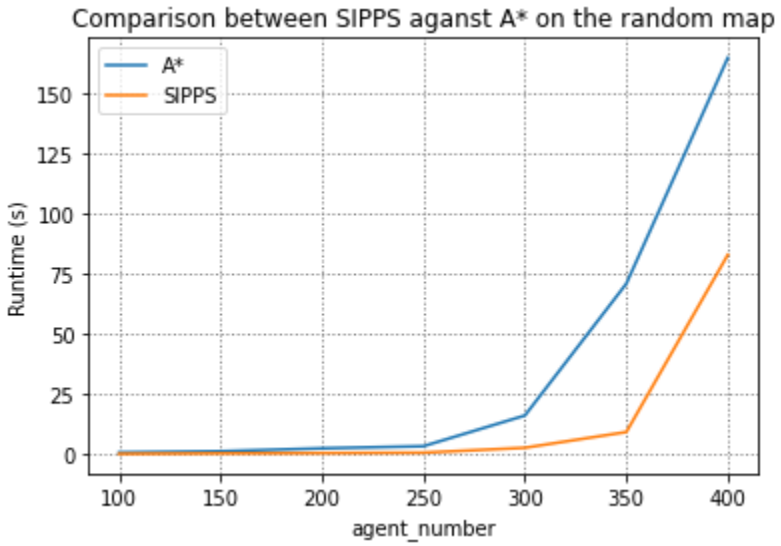
likely a random selected interative algorithm. It means, in order to get to goal, the agent 1 need to move to the “hole” by random walk, and wait at that hole until the green agent by pass to it. The rate of choosing correct path is less than 1/64 for 2 agents by calculation. So in this case, the success rate of the Neigbour Selection is low.

However, in the benchmark case, as the map is much boarder than the simple case, there are more rooms for the agents to walk around randomly to avoid possible collision. So the success rate is much higher than the simple case

Comparing with the Randomized Priority-based Search, the success rate of Collision Base Neighbour Search is most similar to the, but neighbourhood search is much faster. The time Priority-based Search taking is more than hundreds times of Collision Based Neighbour search taking. Even though the neighbour selection took higher cost of the result, it still more fittable to make a draft MAPF collision free path, or judge if the map of the MAPF is solvable.

For the up comming research, consider the the pure random walking can cause trivial walk around the map, the algorithm of more informed Neighbour hood search named “Failure Base Neighbour Selection” will be adapted.

Sipp Single Agent Planner Discussion:



The figure above compares MAPF-LNS2 with SIPPS against MAPF-LNS2 with A* in terms of runtime. When the agent_numbers m is smaller than 250, we cannot see a clear difference in runtime between these two methods. However, when agent_number m is equal to 300, SIPPS dominates A* with 0.64 seconds, which is five times faster than A*. From the calculation of average runtime, SIPPS is also more stable because it has a smaller standard deviation. Due to the evidence I claimed above, I might empirically conclude that SIPP runs significantly faster than space-time A*.

Contribution:

Zhonghong Zhang, zkzhang@sfu.ca 33%
Shenyu Gu, shenyug@sfu.ca 33%
Cheng Zhang, cza111@sfu.ca 33%

Reference

https://www.aaai.org/AAAI22Papers/AAAI-2389.LiJ.pdf
https://www.cs.cmu.edu/~maxim/files/sipp_icra11.pdf

Cohen, L.; Uras, T.; Kumar, T.K.S.; and Koenig, S. 2019. Optimal and Bounded-Suboptimal Multi-Agent Motion Planning. In Proceedings of the International Symposium on Combinatorial Search (SoCS), 44-51

Cohen, L.; Wagner, G.; Chan, D.M.; Choset, H.; Sturtevant, N.R.; Koenig, S.; and Kumar, T.K.S. 2018. Rapid Randomized Restarts for Multi-Agent Path Finding Solvers. In Proceedings of the International Symposium on Combinatorial Search (SoCS), 148-152

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 5628-5635

Instances From the CMPT 417 Individual Project