Replacing space-time A* with Safe Interval Path Planning (SIPP) for Cooperative A* and CBS

CMPT 417

Group 10:

Yifan Zuo, 301354273

Yilun Huang, 301280711

Lizhou Ding, 301326630

1. Introduction

In this project, we implement the Safe Interval Path Planning for Dynamic Environment(SIPP) algorithm. For dynamic obstacles, the position of obstacles is changing constantly instead of static obstacles. If we treat dynamic obstacles as static obstacles in each timestep, we need to add an additional timestep variable to replan. Instead of adding time as an additional dimensionality, the SIPP algorithm introduces the idea of safe intervals. A safe interval is a time period with collision-free configuration. It is surrounded by collisions which pre-timestep and the next timestep are collisions. A configuration usually has limited states or nodes of safe intervals. In this case, it is more efficient compared to adding the time variable and replanning when obstacles move. The SIPP algorithm guarantees the same optimality and completeness as adding the time variable, but much faster.

2. Implementation

2.1 Overview:

For the implementation, we basically use **A* with safe intervals(SIPP)** and **CBS** algorithm. The CBS implementation is the same as the individual project. Basically, CBS detects collisions in paths and creates constraints. CBS inputs constraints to A* with safe intervals.
In the A* with safe intervals, a node has {loc, g_value, h_value, parent node, timstep, safe_interval, wait_time} parameters. Most of them are intuitive. The **safe_interval** is the tuple which contains the start time and end time of the safe interval. The **wait_time** parameter means how much unit times the robot needs to wait to move to the location. For example, the robot needs to wait 2 unit times and then move to the next location, then the wait_time is 2. We first initialize the root node, push it to open_list. Secondly, we call **get_sucessors** function to generate its successor nodes. We return successor nodes back to A* with safe interval function. If the successor is in closed_list, then update and push. If not, we push the successor directly. Doing this process, we either find a path that satisfies the given constraints or do not find a path which means no solution. A* with safe intervals return the path back to CBS. CBS

continues detecting collisions and creating constraint tables. CBS and A* keep doing the process until we find the optimal collision-free path for all agents.

2.2 Functions and Decisions:

Algorithm 1: **a_star_safe_interval()**

Explanation:
Given a map, start location, goal location, h values, agent number and constraints, find the path that satisfies given constraints for the agent.

```
open_list, closed_list = empty, empty
if the goal is not connect to start return no solution
constraint_table = build_constraint_table()
build root node and push root node
while open_list is not empty and time_limit > 0:
        curr = pop_node()
        if find solution and curr_time > max_time return path
        successors = get_successors(curr)
        for successor in successors do:
                if successor in closed_list and with smaller f-val:
                        update the successor
                push(successor)
        time_limit -= 1
return no solution
```

Decision:

- Node structure:
  root = {'loc': start_loc, 'g_val': 0, 'h_val': h_value, 'parent': None, 'timestep': 0, 'safe_interval': root_safe_interval, 'wait_time': 0}
  loc: location of this node
  g_val, h_val: g value and h_value of this node
  parent: parent of this node

timestep: the timestep when this node is createdsafe_interval: the interval which the agent is save in this location
wait_time: how much unit time that the parent node needs to wait in its location. Note here the wait time is not for the current node, instead it's for the parent node. More detail see get_path().

- Max_time:
  When finding the solution, we need to guarantee that the current time is greater than the max_time which is the maximum timestep in the constraint table. This covers the case where the agent needs to leave its goal location to avoid colliding with other agents that pass its goal location.
- time_limit:
  the time_limit is the upper bound of the time that this a_star_safe_interval can run. By limiting the running time, we can detect the case where there is no solution at all and there is an infinite loop.

Algorithm 2: **Build_constraint_table()**

Explanation:
Given constraints and agent number, generate the table that contains the list of constraints of the given agent for each time step.

```
constraint_table, temp = {}, []
deep copy constraints into temp
for constraint in temp do:
    if constraint['agent'] = agent:
        if time_step not in constraint_table.keys():
            create a table with new timestep
        append constraint into constraint_table['timestep']
```

Algorithm 3: **get_safe_interval()**

Explanation:
Given current location, next location, current time and constraint table, calculate the set of safe intervals that satisfies the given constraint table.

```
safe_interval, constraint_set, last_time = [], [], None
for constraint in constraint_table:
    if constraint is vertex constraint and constraint location match:
        append constraint time into constraint_set
    else constraint is edge constraint and constraint location match:
        append constraint time and time - 1 into constraint_set
if constraint_set is empty return [(curr_t, -1)]
for each time in constrant_set:
    if first constraint point is not on start location and time not past:
        append (curr_t, constraint_t -1) into safe_interval
    if not the first constraint point and collide agent not stay in the location:
        if time not past append (curr_t, constraint_t-1) in safe_interval
        else append (pre_constraint_t+1, constraint_t-1) in safe_interval
    if constraint is the last one:
        if time not past append (curr_t, -1) in safe_interval
        else append (constraint_t+1, -1) in safe_interval
return safe_interval
```

Decision:

- Corner case 1, if the constraint location is in the start location, this means the agent is safe until next time in this location.
- Corner case 2, if the other collide agents stay in this location, this means the agent is safe until the other agents leave in this location.
- Corner case 3, when the collide time is past, the safe interval starts with the current time instead of the collide time.
- The algorithm uses -1 to indicate infinity. For example, (0,-1) means the agent is safe from time 0 till forever.

Algorithm 4: **get_successors()**

Explanation:
Given current node, map, h values and constraint table, expand the current node and return all its valid successors.

for each movable next location:
    detect the boundary and obstacles constraints
    start_t = curr_node[‘timestep’] + 1
    end_t = curr_node['safe_interval'][1]
    cfg_safe_intervals = **get_safe_interval** of next location
    for each interval **in** cfg_safe_intervals:
        if interval start > end_t **or** interval end < start_t **continue**
        t = **find_earliest_arrival** for this interval with no collisions
        if t is None **continue**
        generate the successor node
        if t > 1 successor wait_time += t - 1
        append successor into successors
return successors

Decision:

- if t is greater than 1, this means that the agent needs to wait some time in the current location and then go to the next location.
- The movable motion doesn’t include wait action, because we already include this case in the parameter t which makes wait action irrelevant. In other words, if the current location is not the goal location, then the agent eventually will move to the next location and won’t stay in this location.

Algorithm 5: **find_earliest_arrival()**

Explanation:
Given a safe interval and current time, find the earliest arrival time at the next location during the interval with no collisions.

if the safe interval is already past, **return** None
wait_time = start of safe_interval - curr_time - 1
if wait_time > 0 **return** wait_time + 1
else return 1

Decision:

- Given a safe interval, to determine the time to reach the next location. The agent either immediately moves to the next location, or waits, then goes to the next location.

3. Methodology

We want to find out what will be the issue casing the A* search with save interval always has better performance on any instance compare to normal A* search.
Q1: Does the percentage of blocks in the map affect the speed of SIPP with CBS and how is that compare to the normal A* with CBS?
Q2: Does the number of agents affect the speed of SIPP with CBS and how is that compare to the normal A* with CBS?

We generated 50 test instances for each question, and ran algorithms on all map instances. For our analysis, we used all the results which can be solved under a 5 minutes time limit. If it exceeds 5 minutes, we regenerate the instance.

3.1 Experiments Design

We create a map generator that can automatically generate different sizes of instances for running benchmarks. For Q1, we fix the other variables and only change the percentage of blocks in the map. For our experiment, we use a 20*20 map with 5 agents. And we create 50 instances generated by our map generator that the percentage of blocks increases from 1% to 50%. We increase 1%(400*1%=4) of blocks each instance, and record each CPU time of the instance.
For Q2, we fix the other variables and only change the number of agents in the map. For our experiment, we use a 20*20 map with 10% of blocks. And we create 50 instances

generated by our map generator that the number of agents increases from 1 to 20. We increase 1 agent for each 2 instances, and record each CPU time of the instance.

3.2 Example

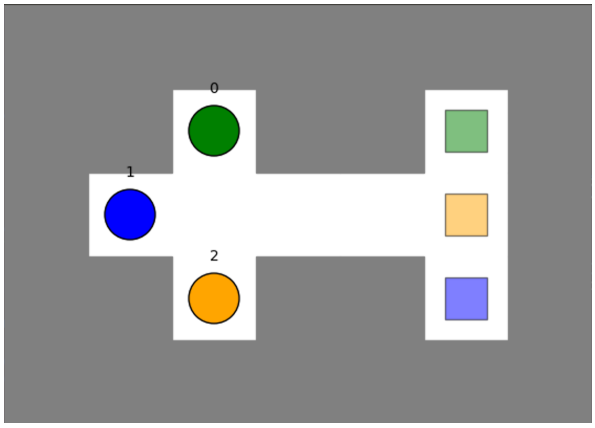

Fig.0. 3 agents simple example show why SIPP is fast

In Fig.0, we show why theoretically SIPP is fast in this instance. The path of all agents are: a0: [(1, 2), (2, 2), (2, 3), (2, 4), (2, 5), (1, 5)], a1: [(2, 1), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 5)] , a2:[(3, 2), (3, 2), (3, 2), (2, 2), (2, 3), (2, 4), (2, 5)]. As you can see, a1 and a2 need to wait a0 pass and a2 need to wait a1 pass. There are lots of wait-then-move actions. SIPP consider wait-then-move action as one node where normal A* consider it as more than one node. Therefore, SIPP is faster than normal A* in this case.

4. Experimental Setup

Detail information about our experimental environment:

- **Language**: Python 3.7.3
- **External Libraries**: matplotlib, numpy
- Operating System: MacOS Big Sur Version 11.2.3
- CPU: 5nm 8 core M1, 2020
- Memory: 16 GB

Our project is based on Model AI Assignments 2020: A Project on Multi-Agent Path Finding (MAPF) by Wolfgang Hönig, Jiaoyang Li, Sven Koenig at the University of

Southern California. We implemented new save time interval A* based on the existing project, used the same visualize method. We created a program that can accept commands to generate default 50 of random M x M size instances for benchmarking. Because the way to generate walls, start positions and goal positions for each agent are totally random, there is a chance that an agent will never find the path(e.g. an agent's start position is surrounded by walls). To avoid this kind of behavior, we find a path for each agent independently to ensure that everyone has a valid path, if no, re-generate a new instance and run the test again.

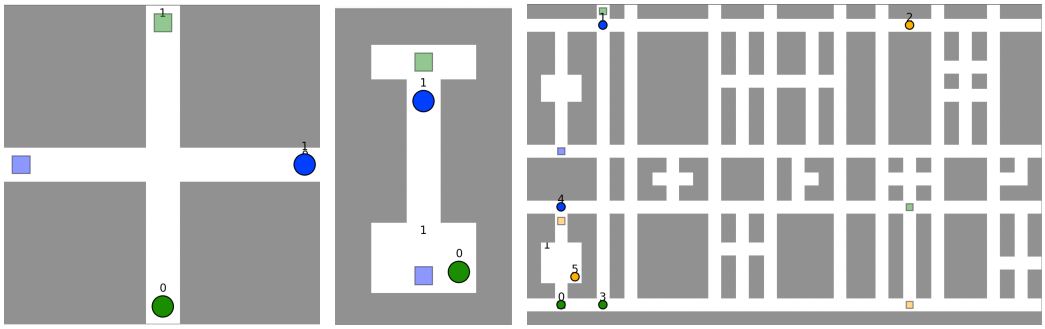

Fig.1. Some maps for Multi-Agent Path-Finding (MAPF) testing.

5. Experimental Results

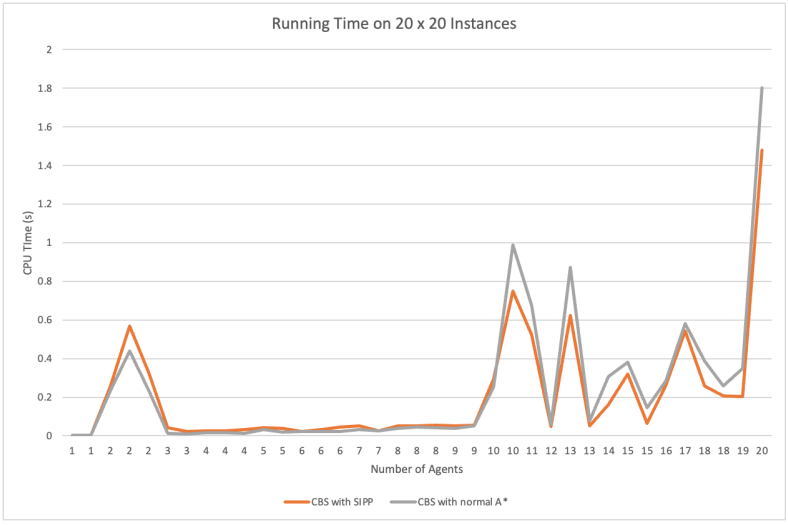

Fig.2. Relation of CPU time and number of agents

In the same 20*20 map, we fix all other variables and only increase the number of agents from 1 to 20 in the map(increase 1 agent for every 2 instances). Figure 2 indicates that running time of normal A* with CBS is slightly less than that of SIPP with CBS before 9 agents, and running time of SIPP with CBS is significantly less than that of normal A* with CBS after 9 agents.

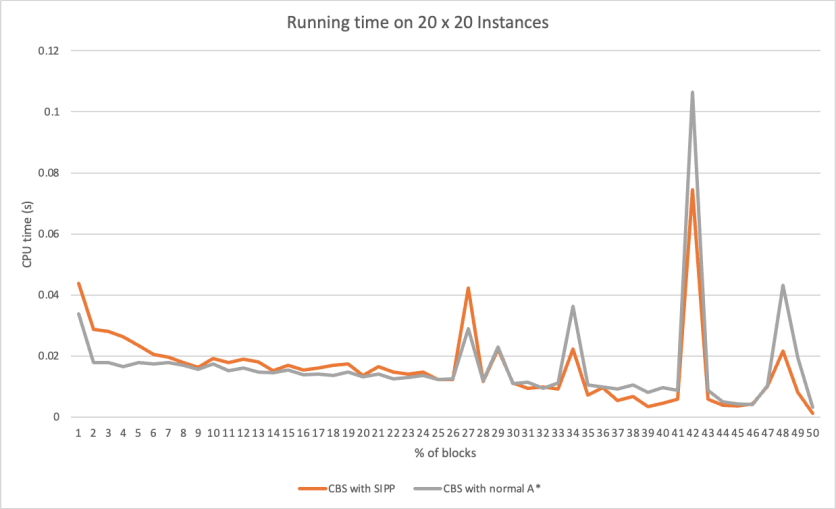

Fig.3. Relation of CPU time and percentage of blocks

In the same 20*20 map, we fix all other variables and only increase the percentage of blocks from 1% to 50% in the map(increase 1% of blocks each instance). Figure 3 shows that running time of normal A* with CBS is less than that of SIPP with CBS before 26% of blocks, running time of SIPP with CBS is less than that of normal A* with CBS after 26% of blocks.

6. Conclusions

Our experimental result shows the same result as our expectation. As Fig 3 shown, assuming other variables are fixed besides the number of blocks, the SIPP with CBS is more time-consuming than normal A* with CBS in the beginning. The reason is that SIPP has the overhead of calculating the safe interval. If the percentage of blocks is small, there will be less wait-then-move actions. In this case, normal A* with CBS performs better than SIPP with CBS. On the other hand, if the percentage of blocks exceeds certain value, there will be more wait-then-move actions, then SIPP with CBS performs better than normal A* with CBS in this case.

As Fig 2 shown, in the 20*20 maps, assuming other variables are fixed besides the number of agents, the number of wait-then-move actions will increase as the number of agents increases. The same reason as above. The performance of SIPP with CBS is better than normal A* with CBS after a certain number of agents.

Overall, we should use SIPP with CBS when CBS generated collisions are large. Otherwise, we should use normal A* with CBS. In other words, when the percentage of blocks in the map is large or the number of agents is large, we should use SIPP with CBS. Otherwise, we should use use normal A* with CBS.

We could create an automatic choosing process in the future. We could calculate the value **v** using the number of agents and percentages of the blocks before running CBS. By doing so, we can tune a certain threshold **th.** If v > th, we call SIPP with CBS. Otherwise, we call normal A* with CBS.

7. References

Mike Phillips∗ and Maxim Likhachev Robotics Institute, Carnegie Mellon University, Pittsburgh, PA 15213. SIPP: Safe Interval Path Planning for Dynamic Environments