Bounded Suboptimal and Anytime Safe Interval Path Planning for Dynamic Environment

Hanra Jeong, Jooyoung Julia Lee, Fitz Laddaran

CMPT417 Group Assignment

Abstract

With an interest in the autonomy of modern automobiles and intelligent robots within a dynamic environment, Safe Interval Path Planning (SIPP) is proposed to find the path in such condition. In this paper, we implemented SIPP, Weighted SIPP with Duplicate States (WSIPPd) and Anytime SIPP algorithm. We compare the different results of these and A* algorithm which is known optimal solution.

Introduction

As we implemented in the individual project with A*, Prioritized, CBS algorithms, Multi-agent path finding (MAPF) is important challenge for many applications under static environment. However, in a real-life, for autonomous car, robotic vacuum, or the automated warehousing robots to safely achieve their goals, they need to navigate from the starting to the goal state in the presence of other moving obstacles like human or other robots. For adopting the dynamic obstacles, we need to predict their paths and locations and adopt this to find the optimal path of target agent.

Safe Interval Path Planning (SIPP) algorithm was proposed to solve this [1] which is known as complete and returning optimal solutions. However, for this SIPP algorithm, to get the solution faster, we need to trade off the solution optimality. [2] To minimize this and get the better optimal solution, a bounded-suboptimal SIPP algorithm is introduced. [3]

Mathematical Description and Algorithm

To implement the algorithms, we modified the submitted personal project code in CMPT 417 course. “visualize.py”, “run_experiments.py” files are modified, and modification is mentioned below.

Manhattan Distance

For this paper, Manhattan distance is used as heuristic function.

For the point A(x1, y1) and B(x2, y2), the Manhattan distance is computed as following:

$$\textbf{Manhattan\_distance\_between\_AB} = |x1 - x2| + |y1 - y2|$$

Common description

Variable

g(s) = cost of the best optimal known path, cost of reaching a node from the initial state [4]

h(s) = cost estimation from state s to goal state with the given heuristic function, heuristic estimation from that node to the goal state [4]

* For this paper, we used Manhattan distance for the heuristic function

f(s) = estimate of the cost of a cost-minimal path from there to any goal state [4]

= g(s) + h(s)

A* algorithm

The code is implemented in astar.py file.

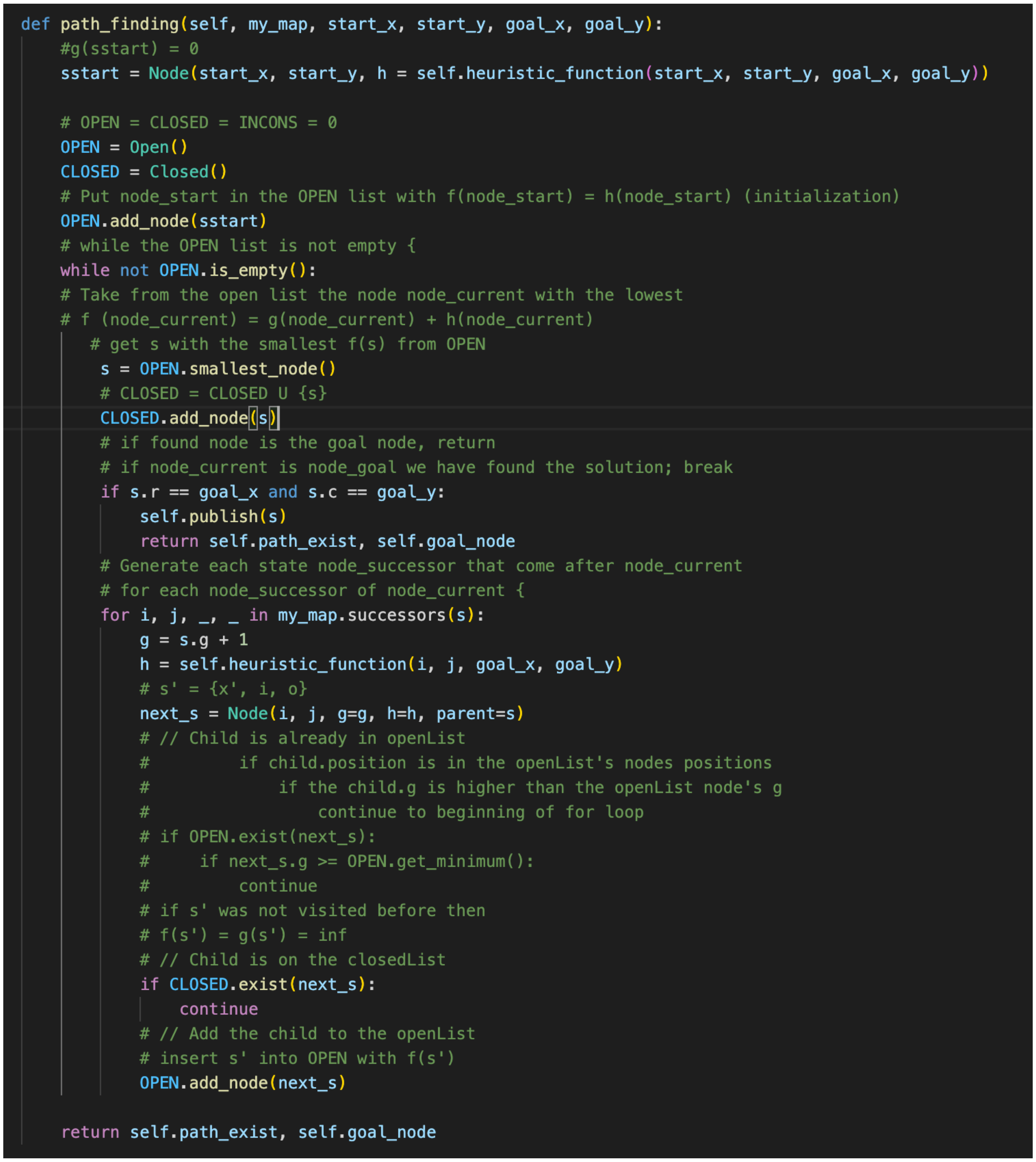
This algorithm is implemented to compare the efficiency with the other implemented SIPP algorithm as it is known optimal solution.

Below pseudocode is the modified code from the given lecture note in the c lass. [4]

A*:

```
1. Push the start node to the OPEN list
2. While there are unexpanded fringe nodes in the OPEN list (If there are no unexpanded
   fringe nodes, stop unsuccessfully.)
3. Pick an unexpanded fringe node n with the smallest $f(n) = g(n) + h(s(n))$
4. Push the node n to the CLOSED list
5. If s is a goal state, stop successfully and return the path from the root node to n
6. Expand n (find the successor states of n) to create a child node of n
```

7. If this child node is not existed in CLOSED list, push the node to the OPEN list
8. Go to 2



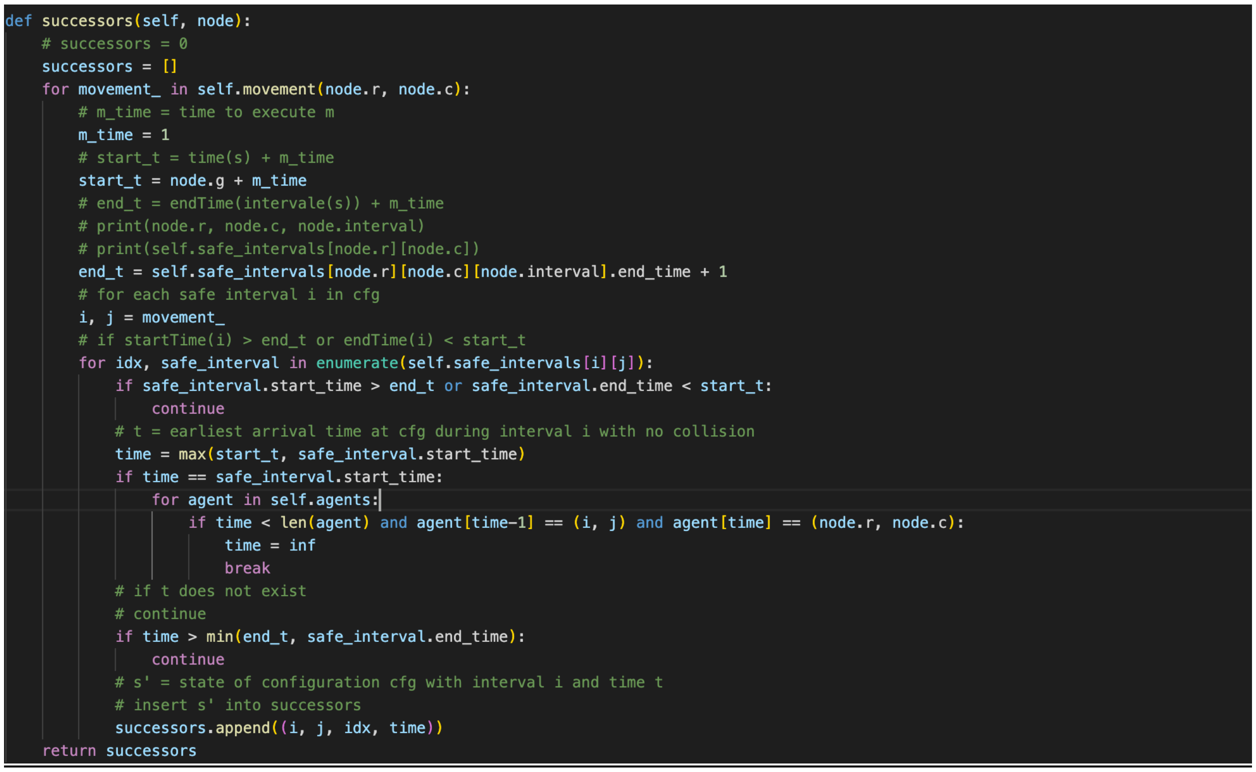
```python
def path_finding(self, my_map, start_x, start_y, goal_x, goal_y):
    #g(sstart) = 0
    sstart = Node(start_x, start_y, h = self.heuristic_function(start_x, start_y, goal_x, goal_y))

    # OPEN = CLOSED = INCONS = 0
    OPEN = Open()
    CLOSED = Closed()
    # Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
    OPEN.add_node(sstart)
    # while the OPEN list is not empty {
    while not OPEN.is_empty():
    # Take from the open list the node node_current with the lowest
    # f (node_current) = g(node_current) + h(node_current)
       # get s with the smallest f(s) from OPEN
        s = OPEN.smallest_node()
        # CLOSED = CLOSED U {s}
        CLOSED.add_node(s)
        # if found node is the goal node, return
        # if node_current is node_goal we have found the solution; break
        if s.r == goal_x and s.c == goal_y:
            self.publish(s)
            return self.path_exist, self.goal_node
        # Generate each state node_successor that come after node_current
        # for each node_successor of node_current {
        for i, j, _, _ in my_map.successors(s):
            g = s.g + 1
            h = self.heuristic_function(i, j, goal_x, goal_y)
            # s' = {x', i, o}
            next_s = Node(i, j, g=g, h=h, parent=s)
            # // Child is already in openList
            #         if child.position is in the openList's nodes positions
            #             if the child.g is higher than the openList node's g
            #                 continue to beginning of for loop
            # if OPEN.exist(next_s):
            #     if next_s.g >= OPEN.get_minimum():
            #         continue
            # if s' was not visited before then
            # f(s') = g(s') = inf
            # // Child is on the closedList
            if CLOSED.exist(next_s):
                continue
            # // Add the child to the openList
            # insert s' into OPEN with f(s')
            OPEN.add_node(next_s)

    return self.path_exist, self.goal_node
```

Successors(s):

# The code to find the successors of the given node s, for the line 6 in A* algorithm

The idea of the pseudocode is from the research paper [1]

```
1. Initialize the successors list
2. For each m from possible movements of node s
3. Construct cfg which is configuration of m applied to s (Here, we made the safe_intervals 2D list, and store the interval list into each position (r, c). Thus, interval(s) can be found by self.safe_intervals[node.r][node.c][node.interval] and each value has the start_time and end_time.)
4. m_time = time to execute m (For one moving step we assign value 1, so, m_time = 1)
5. start_t = time(s) + m_time (Here, for the nodes of the tree, we made the class Node, store the information of g, h, f, interval values and current position, r, c. time(s) is represented with the node.g value)
6. end_t = end_time(interval(s)) + m_time
7. for each safe interval in cfg
8. if startTime(safe intervals[i]) > end_t or endTime(safe intervals[i]) < start_t: continue
9. t = earliest arrival time at cfg during interval i with no collisions
10. if t doesn't exist : continue
11. s' = (I, j, interval, time) and insert s' into successors
12. return successors
```

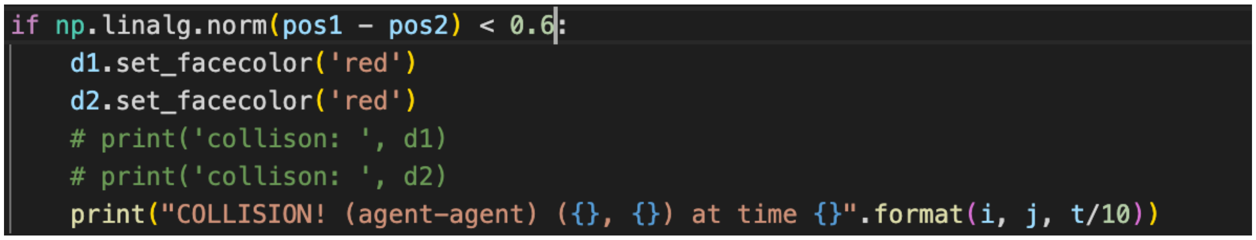


```python
def successors(self, node):
    # successors = 0
    successors = []
    for movement_ in self.movement(node.r, node.c):
        # m_time = time to execute m
        m_time = 1
        # start_t = time(s) + m_time
        start_t = node.g + m_time
        # end_t = endTime(intervale(s)) + m_time
        # print(node.r, node.c, node.interval)
        # print(self.safe_intervals[node.r][node.c])
        end_t = self.safe_intervals[node.r][node.c][node.interval].end_time + 1
        # for each safe interval i in cfg
        i, j = movement_
        # if startTime(i) > end_t or endTime(i) < start_t
        for idx, safe_interval in enumerate(self.safe_intervals[i][j]):
            if safe_interval.start_time > end_t or safe_interval.end_time < start_t:
                continue
            # t = earliest arrival time at cfg during interval i with no collision
            time = max(start_t, safe_interval.start_time)
            if time == safe_interval.start_time:
                for agent in self.agents:
                    if time < len(agent) and agent[time-1] == (i, j) and agent[time] == (node.r, node.c):
                        time = inf
                        break
            # if t does not exist
            # continue
            if time > min(end_t, safe_interval.end_time):
                continue
            # s' = state of configuration cfg with interval i and time t
            # insert s' into successors
            successors.append((i, j, idx, time))
    return successors
```

SIPP algorithm

* General idea for the following algorithm is referred to [1]

As it is mentioned in the paper [1], the collision distance is calculated by adding the agent's radius and obstacles' radius. Since every agent and obstacles are described as circle with the radius 0.3, as the Figure. 2. Shows, the collision distance is fixed with 0.6.



```python
if np.linalg.norm(pos1 - pos2) < 0.6:
    d1.set_facecolor('red')
    d2.set_facecolor('red')
    # print('collison: ', d1)
    # print('collison: ', d2)
    print("COLLISION! (agent-agent) ({}, {}) at time {}".format(i, j, t/10))
```

To descriptively named, rather than using the original run_experiments.py file, we made the Map class and implement the self.safe_intervals variable. The original My_map variable is only used to print the result and as the passing variable to the visualize.py file.

Safe Interval = the time on the timeline where the agents can process without the collision

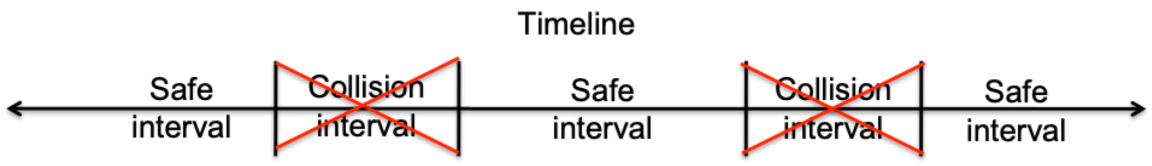
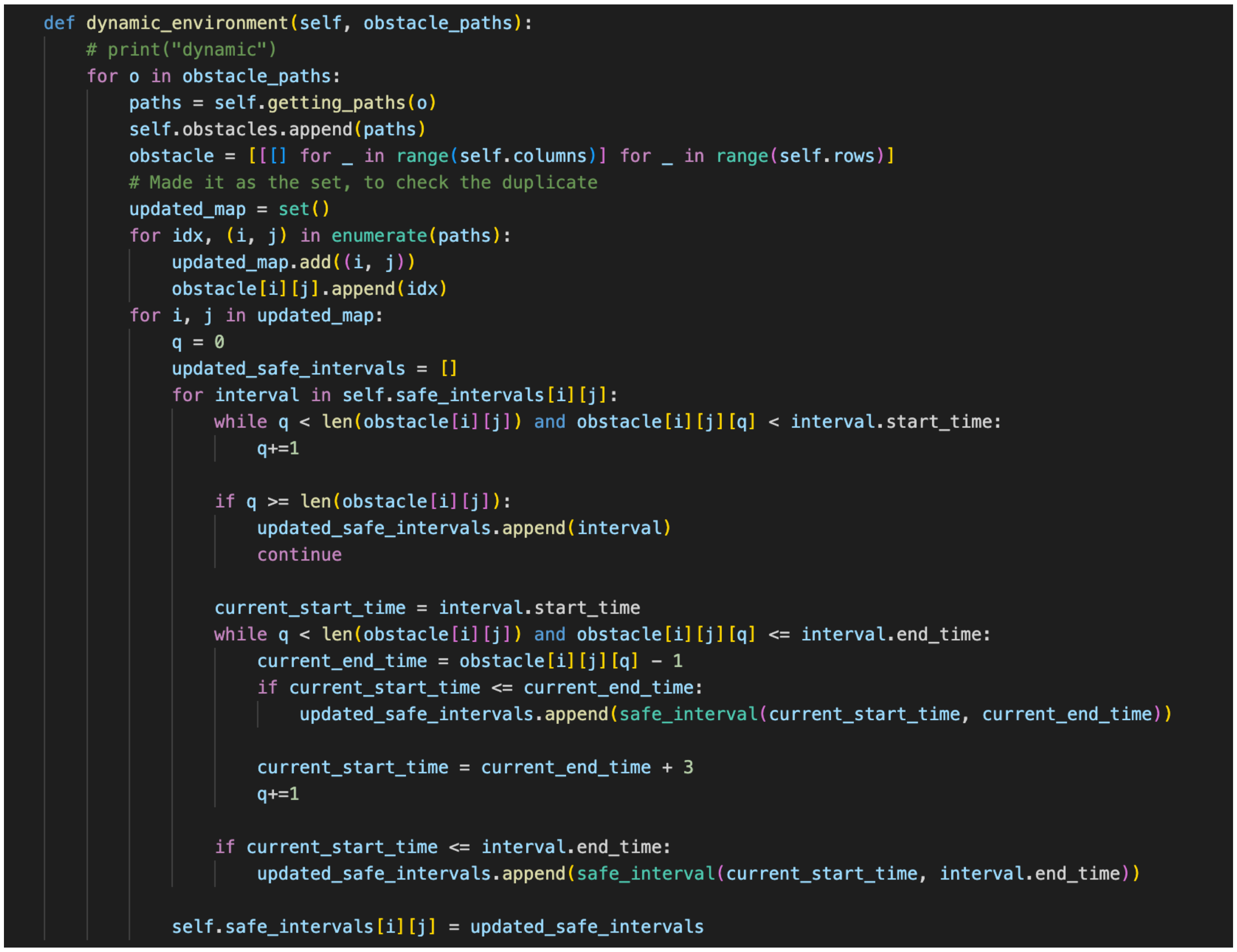


Fig. 1. A timeline showing the safe and collision intervals [1]

To create and update the timelines of configuration by considering the moving obstacles, the algorithm iterates through the trajectory of each obstacle.

Dynamic_environment:

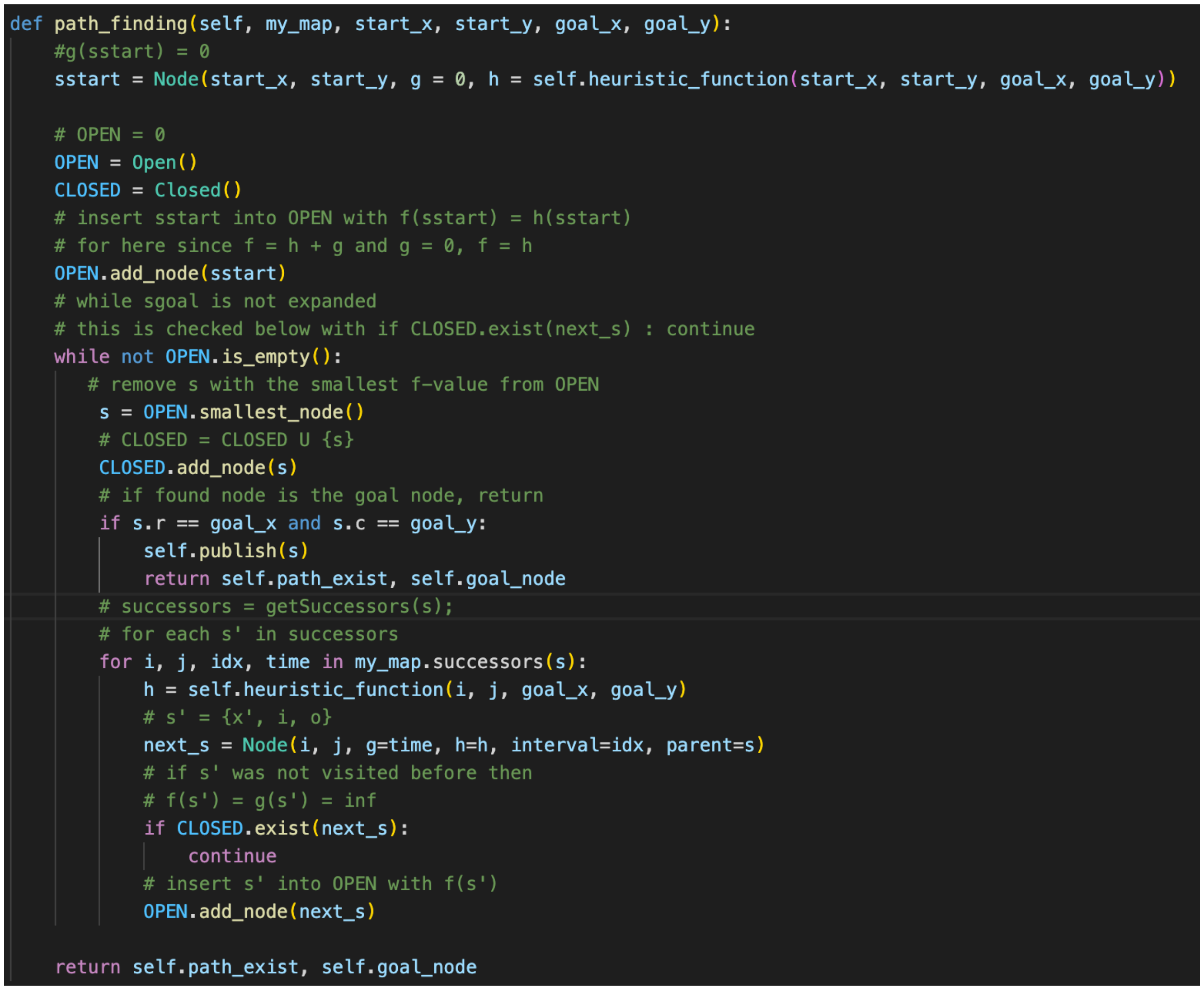
1. iterate through the trajectory of each obstacle
2. generate the path of obstacle and append it to the list obstacles
3. initialize 2D array obstacle to store the existence of obstacle location. Each array element has an array to store the time when the obstacle agent visits along the path
4. initialize a set variable updated_map to check the duplicate and store the path of obstacle as the coordinates.
5. iterate the location of the path of obstacle
6. for the interval, find the time t where time < length(obstacle(position)) and the store time indicating when the obstacle agent located in the given position < startTime(inverval)
7. if the time >= length(obstacle(position)) : update the safe intervals
8. from the start time of the interval, iterate to find the time t where time < length(obstacle(position)) and the store time indicating when the obstacle agent located in the given position <= endTime(inverval)
9. if the founded store time of the location in obstacle array is less than the start time of the interval, update the safe intervals



```
def dynamic_environment(self, obstacle_paths):
    # print("dynamic")
    for o in obstacle_paths:
        paths = self.getting_paths(o)
        self.obstacles.append(paths)
        obstacle = [[[] for _ in range(self.columns)] for _ in range(self.rows)]
        # Made it as the set, to check the duplicate
        updated_map = set()
        for idx, (i, j) in enumerate(paths):
            updated_map.add((i, j))
            obstacle[i][j].append(idx)
        for i, j in updated_map:
            q = 0
            updated_safe_intervals = []
            for interval in self.safe_intervals[i][j]:
                while q < len(obstacle[i][j]) and obstacle[i][j][q] < interval.start_time:
                    q+=1

                if q >= len(obstacle[i][j]):
                    updated_safe_intervals.append(interval)
                    continue

                current_start_time = interval.start_time
                while q < len(obstacle[i][j]) and obstacle[i][j][q] <= interval.end_time:
                    current_end_time = obstacle[i][j][q] - 1
                    if current_start_time <= current_end_time:
                        updated_safe_intervals.append(safe_interval(current_start_time, current_end_time))

                    current_start_time = current_end_time + 3
                    q+=1

                if current_start_time <= interval.end_time:
                    updated_safe_intervals.append(safe_interval(current_start_time, interval.end_time))

            self.safe_intervals[i][j] = updated_safe_intervals
```

SIPP algorithm is slightly modified version of A* search. It runs same with A8 search, except for the information and the way of getting the successors. In A* search, we only used position to get the child node, but for SIPP algorithm, to update the time line, it uses position and interval. The pseudo code is same with the A* written above as the following:

1. Push the start node to the OPEN list
2. While there are unexpanded fringe nodes in the OPEN list (If there are no unexpanded fringe nodes, stop unsuccessfully.)
3. Pick an unexpanded fringe node n with the smallest f(n) = g(n) + h(s(n))
4. Push the node n to the CLOSED list
5. If s is a goal state, stop successfully and return the path from the root node to n
6. Expand n (find the successor states of n) to create a child node of n
7. If this child node is not existed in CLOSED list, push the node to the OPEN list
8. Go to 2



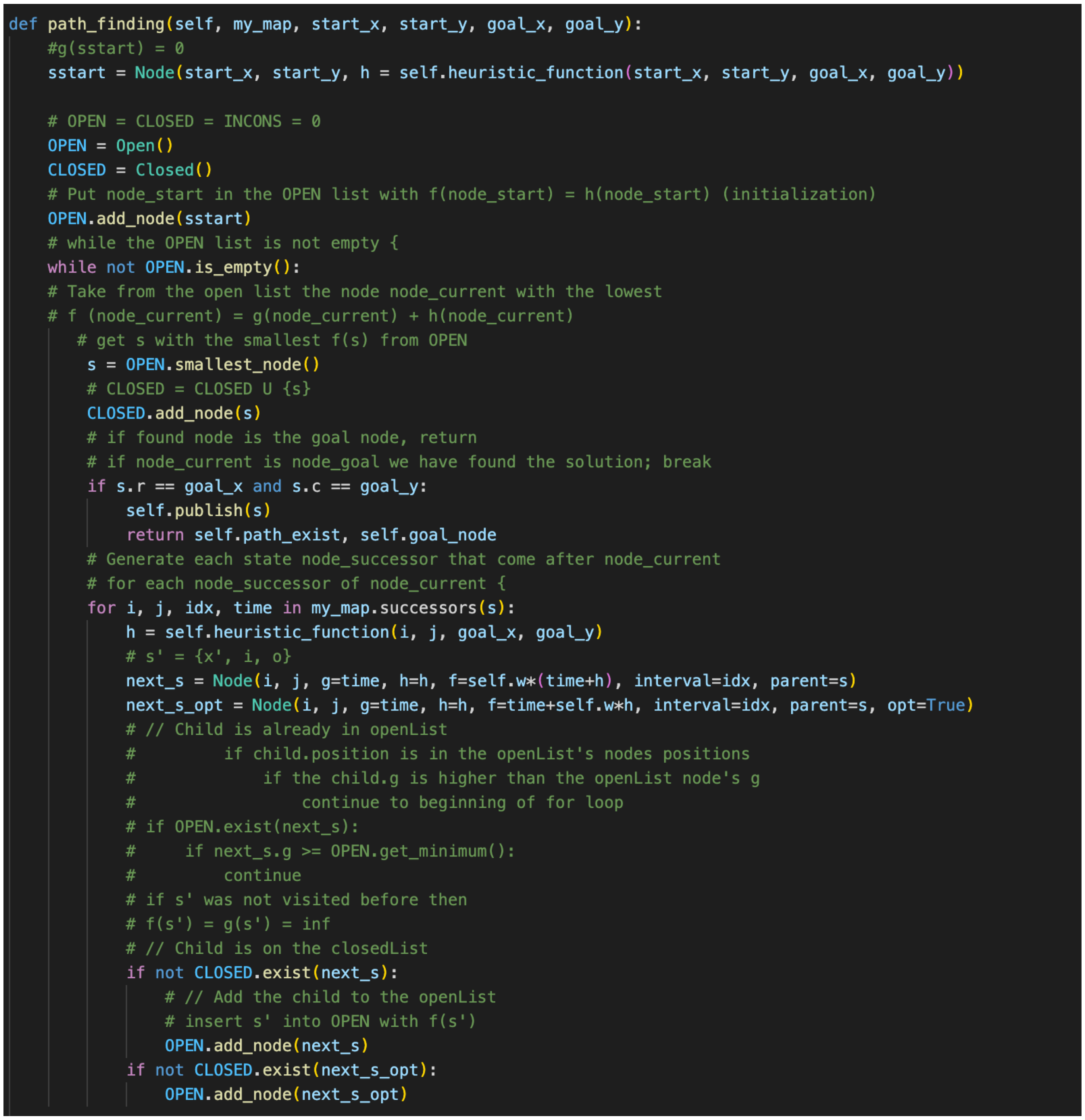
```python
def path_finding(self, my_map, start_x, start_y, goal_x, goal_y):
    #g(sstart) = 0
    sstart = Node(start_x, start_y, g = 0, h = self.heuristic_function(start_x, start_y, goal_x, goal_y))

    # OPEN = 0
    OPEN = Open()
    CLOSED = Closed()
    # insert sstart into OPEN with f(sstart) = h(sstart)
    # for here since f = h + g and g = 0, f = h
    OPEN.add_node(sstart)
    # while sgoal is not expanded
    # this is checked below with if CLOSED.exist(next_s) : continue
    while not OPEN.is_empty():
        # remove s with the smallest f-value from OPEN
        s = OPEN.smallest_node()
        # CLOSED = CLOSED U {s}
        CLOSED.add_node(s)
        # if found node is the goal node, return
        if s.r == goal_x and s.c == goal_y:
            self.publish(s)
            return self.path_exist, self.goal_node
        # successors = getSuccessors(s);
        # for each s' in successors
        for i, j, idx, time in my_map.successors(s):
            h = self.heuristic_function(i, j, goal_x, goal_y)
            # s' = {x', i, o}
            next_s = Node(i, j, g=time, h=h, interval=idx, parent=s)
            # if s' was not visited before then
            # f(s') = g(s') = inf
            if CLOSED.exist(next_s):
                continue
            # insert s' into OPEN with f(s')
            OPEN.add_node(next_s)

    return self.path_exist, self.goal_node
```

Weighted SIPP with Duplicate States (WSIPPd) algorithm

This algorithm is based on the SIPP algorithm, but rather than creating one child as the SIPP algorithm shows : next_s = Node(I, j, g=time, h=h, interval=idx, parent=s), we create two child nodes. An optimal copy with f = w * (g + h) and suboptimal copy with f = g + w * h where w ≥ 1. [2]

1. Push the start node to the OPEN list
2. While there are unexpanded fringe nodes in the OPEN list (If there are no unexpanded fringe nodes, stop unsuccessfully.)
3. Pick an unexpanded fringe node n with the smallest f(n) = g(n) + h(s(n))
4. Push the node n to the CLOSED list

5. If s is a goal state, stop successfully and return the path from the root node to n
6. Expand n (find the successor states of n) to create two child node of n, one is optimal copy and the other is suboptimal copy.
7. If this optimal copy node is not existed in CLOSED list, push the node to the OPEN list
8. If this suboptimal copy node is not existed in CLOSED list, push the node to the OPEN list
9. Go to 2



```python
def path_finding(self, my_map, start_x, start_y, goal_x, goal_y):
    #g(sstart) = 0
    sstart = Node(start_x, start_y, h = self.heuristic_function(start_x, start_y, goal_x, goal_y))

    # OPEN = CLOSED = INCONS = 0
    OPEN = Open()
    CLOSED = Closed()
    # Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
    OPEN.add_node(sstart)
    # while the OPEN list is not empty {
    while not OPEN.is_empty():
    # Take from the open list the node node_current with the lowest
    # f (node_current) = g(node_current) + h(node_current)
       # get s with the smallest f(s) from OPEN
        s = OPEN.smallest_node()
        # CLOSED = CLOSED U {s}
        CLOSED.add_node(s)
        # if found node is the goal node, return
        # if node_current is node_goal we have found the solution; break
        if s.r == goal_x and s.c == goal_y:
            self.publish(s)
            return self.path_exist, self.goal_node
        # Generate each state node_successor that come after node_current
        # for each node_successor of node_current {
        for i, j, idx, time in my_map.successors(s):
            h = self.heuristic_function(i, j, goal_x, goal_y)
            # s' = {x', i, o}
            next_s = Node(i, j, g=time, h=h, f=self.w*(time+h), interval=idx, parent=s)
            next_s_opt = Node(i, j, g=time, h=h, f=time+self.w*h, interval=idx, parent=s, opt=True)
            # // Child is already in openList
            #         if child.position is in the openList's nodes positions
            #             if the child.g is higher than the openList node's g
            #                 continue to beginning of for loop
            # if OPEN.exist(next_s):
            #     if next_s.g >= OPEN.get_minimum():
            #         continue
            # if s' was not visited before then
            # f(s') = g(s') = inf
            # // Child is on the closedList
            if not CLOSED.exist(next_s):
                # // Add the child to the openList
                # insert s' into OPEN with f(s')
                OPEN.add_node(next_s)
            if not CLOSED.exist(next_s_opt):
                OPEN.add_node(next_s_opt)
```

Anytime SIPP algorithm

This algorithm is extended version of above SIPP with anytime planning with decreasing epsilon value. In our project, we decrease the epsilon value by dividing it into 2 while epsilon value is greater than 1.

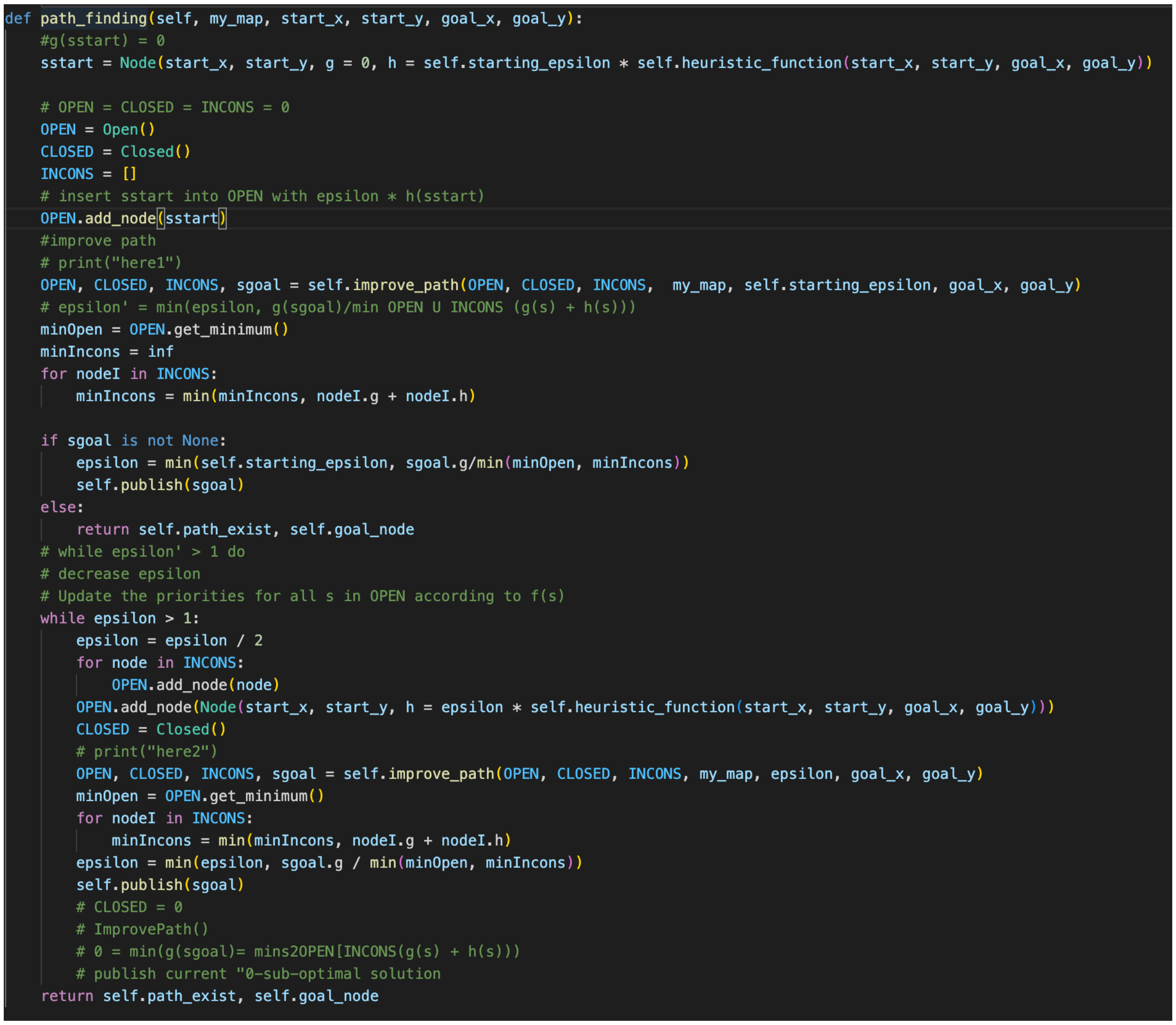
$$\varepsilon' = \varepsilon / 2, (\varepsilon > 1)$$

Newly introduced INCONS variable is used to store the cheaper path's states which is already in CLOSED and not re-expanded. [5]

* Below pseudocode is referred to the paper [5].

AnytimeSIPP:

```
1. Initialize g(sstart) = 0, OPEN = CLOSED = INCONS = 0
2. Insert sstart into OPEN with epsilon * h(sstart)
3. ImprovePath()
4. epsilon' = min ( epsilon, g(sgoal) / $min_{s \in OPEN \cup INCONS}(g(s) + h(s))$
5. publish current epsilon' suboptimal solution, if there is no solution, terminate and return failure
6. while epsilon' > 1, decrease epsilon (in our code, epsilon = epsilon / 2)
7. Move node from INCONS into OPEN
8. Update the OPEN and initialize CLOSED = 0
9. ImprovePath()
10. epsilon' = min ( epsilon, g(sgoal) / $min_{s \in OPEN \cup INCONS}(g(s) + h(s))$
11. publish current epsilon' suboptimal solution
```



```python
def path_finding(self, my_map, start_x, start_y, goal_x, goal_y):
    #g(sstart) = 0
    sstart = Node(start_x, start_y, g = 0, h = self.starting_epsilon * self.heuristic_function(start_x, start_y, goal_x, goal_y))

    # OPEN = CLOSED = INCONS = 0
    OPEN = Open()
    CLOSED = Closed()
    INCONS = []
    # insert sstart into OPEN with epsilon * h(sstart)
    OPEN.add_node(sstart)
    #improve path
    # print("here1")
    OPEN, CLOSED, INCONS, sgoal = self.improve_path(OPEN, CLOSED, INCONS,  my_map, self.starting_epsilon, goal_x, goal_y)
    # epsilon' = min(epsilon, g(sgoal)/min OPEN U INCONS (g(s) + h(s)))
    minOpen = OPEN.get_minimum()
    minIncons = inf
    for nodeI in INCONS:
        minIncons = min(minIncons, nodeI.g + nodeI.h)

    if sgoal is not None:
        epsilon = min(self.starting_epsilon, sgoal.g/min(minOpen, minIncons))
        self.publish(sgoal)
    else:
        return self.path_exist, self.goal_node
    # while epsilon' > 1 do
    # decrease epsilon
    # Update the priorities for all s in OPEN according to f(s)
    while epsilon > 1:
        epsilon = epsilon / 2
        for node in INCONS:
            OPEN.add_node(node)
        OPEN.add_node(Node(start_x, start_y, h = epsilon * self.heuristic_function(start_x, start_y, goal_x, goal_y)))
        CLOSED = Closed()
        # print("here2")
        OPEN, CLOSED, INCONS, sgoal = self.improve_path(OPEN, CLOSED, INCONS, my_map, epsilon, goal_x, goal_y)
        minOpen = OPEN.get_minimum()
        for nodeI in INCONS:
            minIncons = min(minIncons, nodeI.g + nodeI.h)
        epsilon = min(epsilon, sgoal.g / min(minOpen, minIncons))
        self.publish(sgoal)
        # CLOSED = 0
        # ImprovePath()
        # 0 = min(g(sgoal)= mins2OPEN[INCONS(g(s) + h(s)))
        # publish current "0-sub-optimal solution
    return self.path_exist, self.goal_node
```

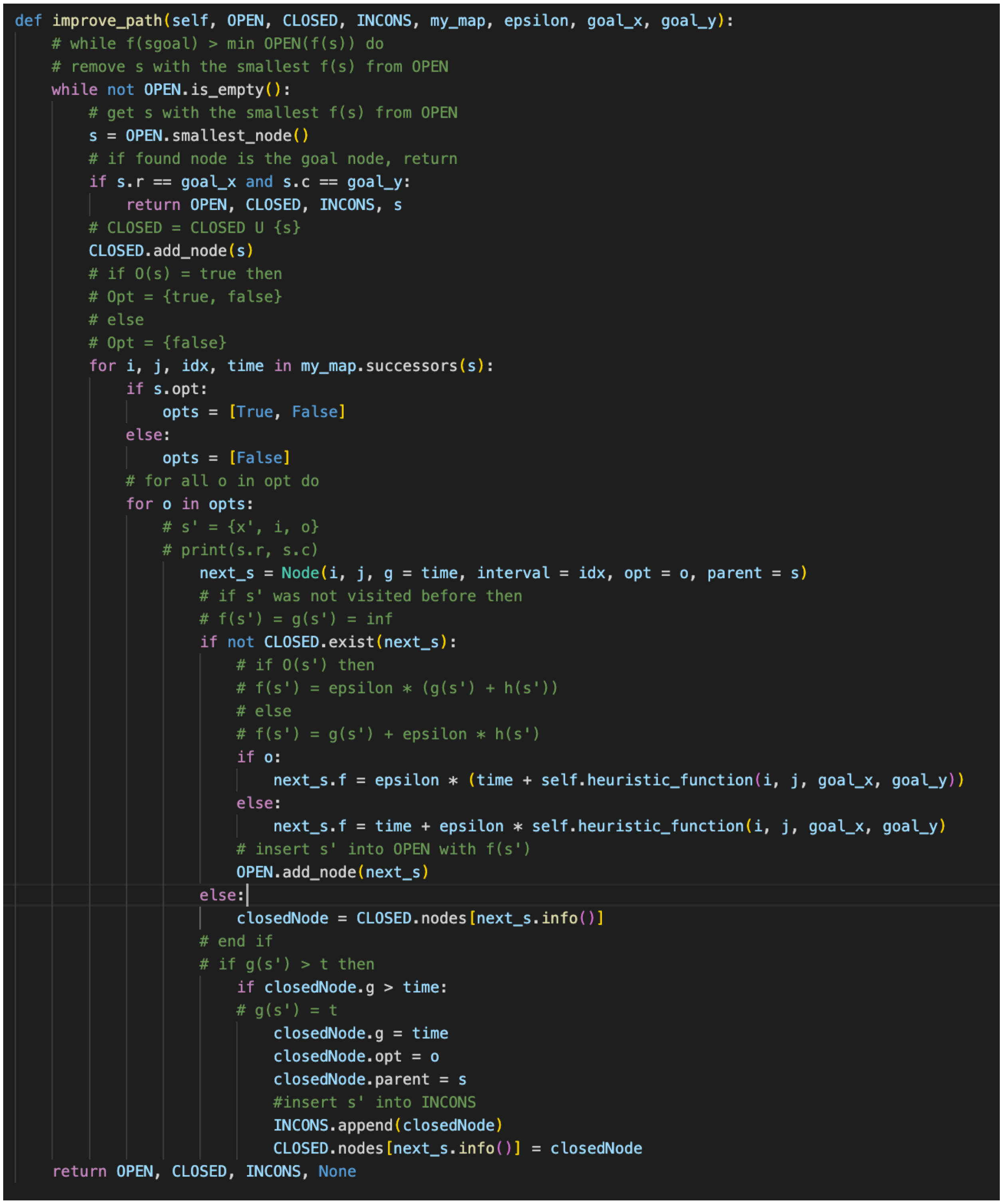
To gain the completeness and sub-optimality bound of algorithm, the improvePath() algorithm used optimal and sub-optimal node. Optimal state has f(n) = g(n) + epsilon * h(s(n)) and suboptimal state has f(n) = epsilon * (g(n) + h(s(n)).

* Below pseudocode is referred to the paper [5].

Improve_path:

1. for the smallest f(s) node s from OPEN, push it to CLOSED
2. for each in successors
3. if O(s) = true, Opt = {true, false}, else, Opt = {false}
4. for all o in Opt,
5. s' = {x', i, o}, if s' is not visited before,
6. if opt : create node with f = epsilon * (g + h), else : create f = g + epsilon * h

7. insert s’ into OPEN
8. if s’ is visited before, insert s’ into INCONS



```
def improve_path(self, OPEN, CLOSED, INCONS, my_map, epsilon, goal_x, goal_y):
    # while f(sgoal) > min OPEN(f(s)) do
    # remove s with the smallest f(s) from OPEN
    while not OPEN.is_empty():
        # get s with the smallest f(s) from OPEN
        s = OPEN.smallest_node()
        # if found node is the goal node, return
        if s.r == goal_x and s.c == goal_y:
            return OPEN, CLOSED, INCONS, s
        # CLOSED = CLOSED U {s}
        CLOSED.add_node(s)
        # if O(s) = true then
        # Opt = {true, false}
        # else
        # Opt = {false}
        for i, j, idx, time in my_map.successors(s):
            if s.opt:
                opts = [True, False]
            else:
                opts = [False]
            # for all o in opt do
            for o in opts:
                # s' = {x', i, o}
                # print(s.r, s.c)
                    next_s = Node(i, j, g = time, interval = idx, opt = o, parent = s)
                    # if s' was not visited before then
                    # f(s') = g(s') = inf
                    if not CLOSED.exist(next_s):
                        # if O(s') then
                        # f(s') = epsilon * (g(s') + h(s'))
                        # else
                        # f(s') = g(s') + epsilon * h(s')
                        if o:
                            next_s.f = epsilon * (time + self.heuristic_function(i, j, goal_x, goal_y))
                        else:
                            next_s.f = time + epsilon * self.heuristic_function(i, j, goal_x, goal_y)
                        # insert s' into OPEN with f(s')
                        OPEN.add_node(next_s)
                    else:
                        closedNode = CLOSED.nodes[next_s.info()]
                    # end if
                    # if g(s') > t then
                        if closedNode.g > time:
                        # g(s') = t
                            closedNode.g = time
                            closedNode.opt = o
                            closedNode.parent = s
                            #insert s' into INCONS
                            INCONS.append(closedNode)
                            CLOSED.nodes[next_s.info()] = closedNode
    return OPEN, CLOSED, INCONS, None
```

Experimental Setup

The code is implemented in Python. Detail of the version for the environmental setting and the main computer that we used is described as below.

Conda : 4.14.0 (for the virtual environment setting)

Python version : Python 3.9.13

ffmpeg version : 5.1 (for creating gif file)

matplotlib version : 3.5.1

used computer : Apple M1 Pro 2021

- M1 macOS Monterey Version 12.4
- 10-core CPU with 8 performance cores and 2 efficiency cores
- 16-core GPU
- 16-core Neural Engine
- Memory 16 GB
- Flash Storage 1TB

With this setting, the code can be run by the below structure of command line.

python run_experiments.py "map_file" "obstacle" "solving algorithm"

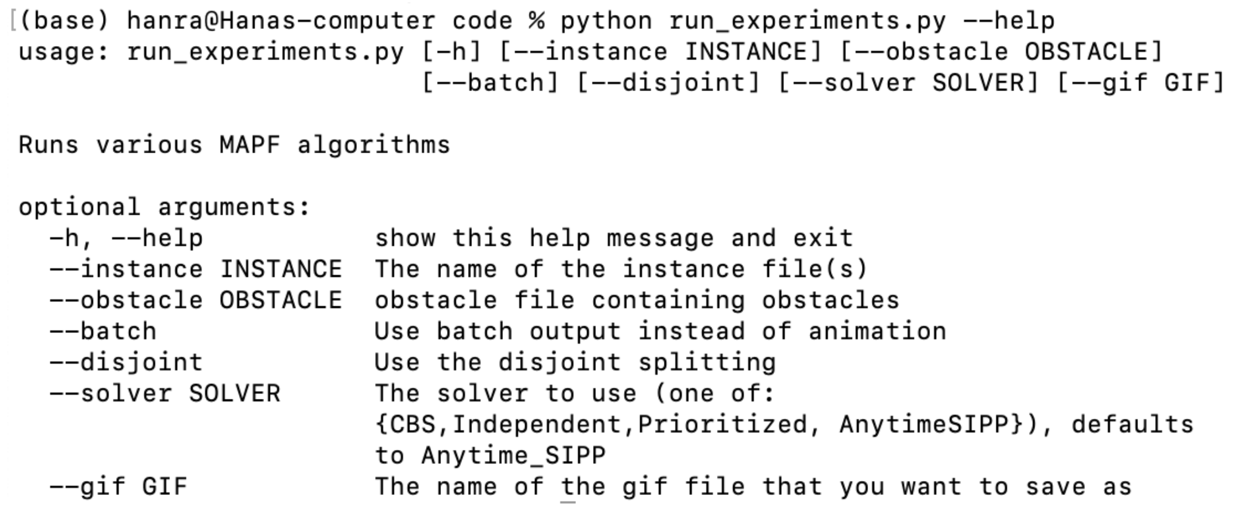
For the arguments:

--instance map_file_name	: The name of the map instance file
--obstacle [input]	: ***random*** – for generating the obstacle randomly
	: ***obstacle_file_name*** – for running the submitted obstacle file
--solver algorithm_name	: AStar – for running A* algorithm
	: SIPP – for running SIPP algorithm
	: weightedSIPP – for running WSIPPd algorithm
	: AnytimeSIPP – for running AnytimeSIPP algorithm

With the command line:

Python run_experiments.py –help

You can find the more detail information as the below:



```
(base) hanra@Hanas-computer code % python run_experiments.py --help
usage: run_experiments.py [-h] [--instance INSTANCE] [--obstacle OBSTACLE]
                          [--batch] [--disjoint] [--solver SOLVER] [--gif GIF]

Runs various MAPF algorithms

optional arguments:
  -h, --help           show this help message and exit
  --instance INSTANCE  The name of the instance file(s)
  --obstacle OBSTACLE  obstacle file containing obstacles
  --batch              Use batch output instead of animation
  --disjoint           Use the disjoint splitting
  --solver SOLVER      The solver to use (one of:
                       {CBS,Independent,Prioritized, AnytimeSIPP}), defaults
                       to Anytime_SIPP
  --gif GIF            The name of the gif file that you want to save as
```

Experimental Instances

Map instance:

We implemented the following map instance. It is hand-crafted by looking at the map from one of the team member's place to SFU. It includes roundabout and the real road. The main agent is assigned with the start location (17, 11) and the goal (1, 48). This main agent is implemented with the black color circle for the agent, and the black color square for the goal location.

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
....................@@@@@@@@@@@@@@@@@@@@@@@@.....
......................@@@@@@@@@@@@@@@@@@@@.......
@@@..@@@..@@@@@@@..@.....@@@@@@@@@@@@@@@.........
@@@..@@@........@..@@@.......@@@@@@@........@@@@@
@@@..@@@........@..@@@@.................@@..@@@@@
@@@..@@@..@@@@@@@..@@@@..@@@@.......@@@@@@..@@@@@
@@@..@@@..@@@@@@@..@@....@@@@@@@@@@.@@@@@@..@@@@@
@@@................@@....@@@@@@@@@@.@@@@@@..@@@@@
@@@..@@@...........@@@@.............@@@@@@..@@@@@
@@@..@@@..@@@@@@@@@@@@@..@@@@@@@@@@@........@@@@@
@@@.......@@@@@@@@@@@@@..@@@@@@@@@@@........@@@@@
@@@..@@@..............................@@@@..@@@@@
@@@..@@@..............................@@@@..@@@@@
@@@..@@@..@@@@@@.@@@@@@@@@@@@@@@@@..........@@@@@
@@@.......@@@@@@.@@@@@@.....................@@@@@
@@@.................................@@@@@@@.@@@@@
@@@......................@@@@@@@..@@@@@@@@@.@@@@@
@@@.......@@.@@..@@@..@..@@@@@@@..@@@@@@......@@@
@@@..@@@..@@.@@..@@@..@..@..........@@@...@@...@@
@@@.......@@.@@.......@..@...............@@@@..@@
@@@..@@@..@@.@@..@@@..@..@..@@@..@@@@@@..@@@@..@@
@@@..@@@..@@.@@..@@@..@..........@@@@@@...@@...@@
@@@...........................................@@@
@@@..........................................@@@@
@@@..@@@..@@@@@..@@@..@..@@@@@@@@@@@@@@@@@@.@@@@@
.................................................
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

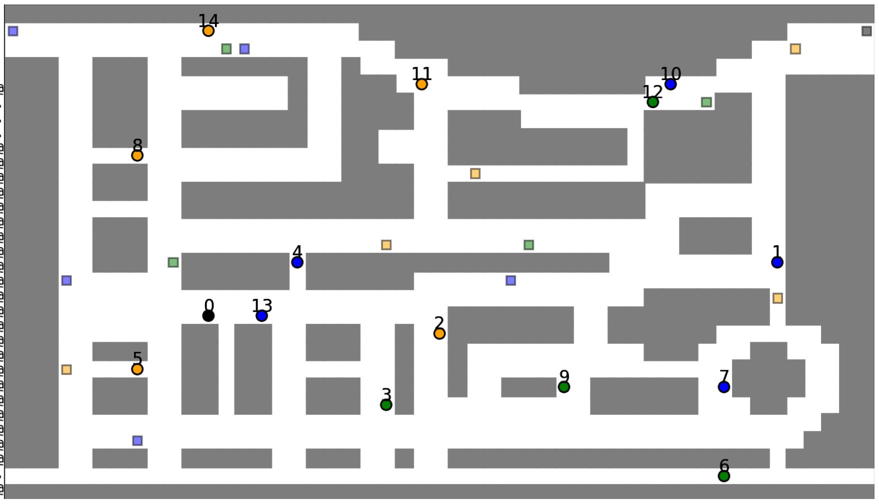


Fig. 2. The implemented map instance in map1.map file in custominstances folder (LEFT) and displayed animation map (RIGHT)

To implement the randomly generate dynamic obstacles:

The command line :

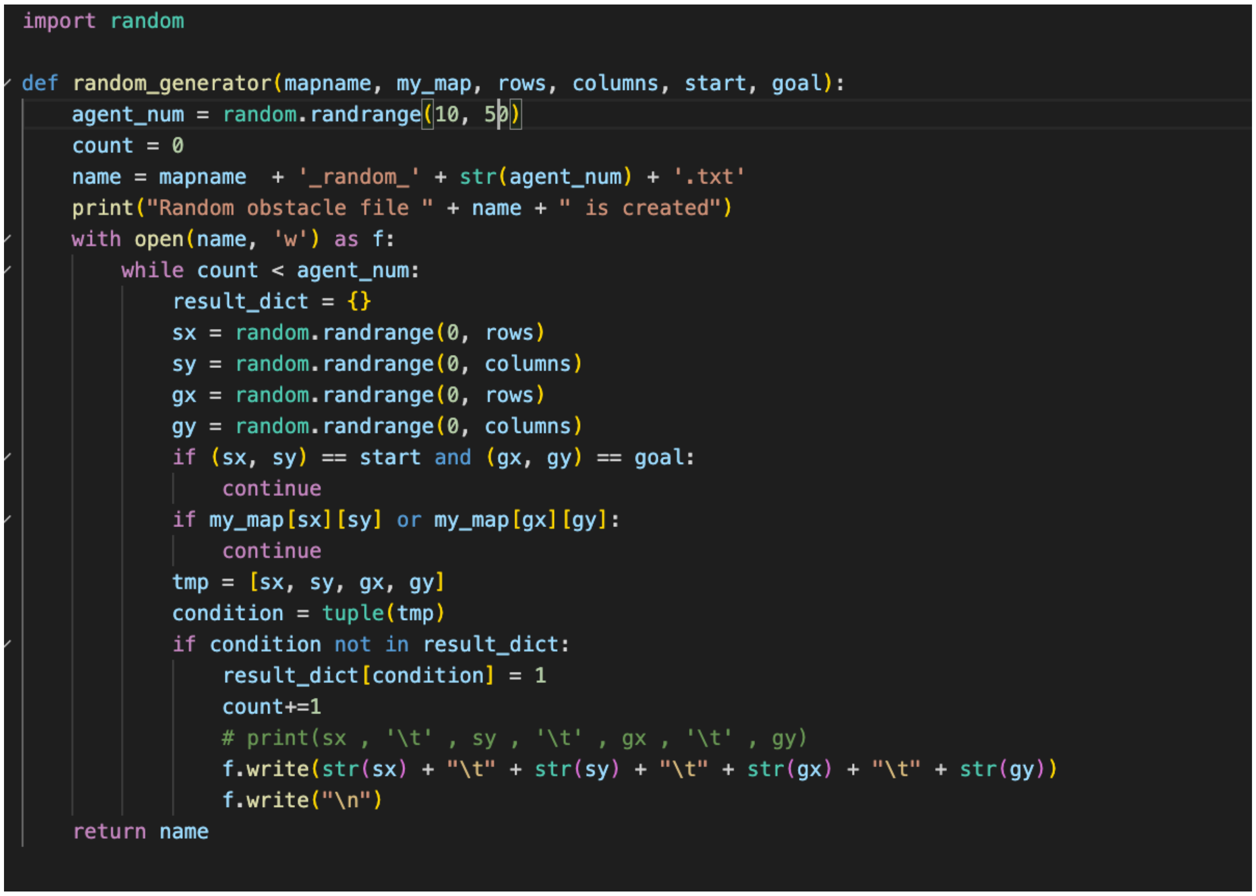
1. Astar

```
python run_experiments.py --instance custominstances/map1.map --obstacle random --solver AStar
python run_experiments.py --instance custominstances/map1.map --obstacle random --solver SIPP
python run_experiments.py --instance custominstances/map1.map --obstacle random --solver weightedSIPP
python run_experiments.py --instance custominstances/map1.map --obstacle random --solver AnytimeSIPP
```

By commanding with the –obstacle random, it runs the function random_generator from random_agents_generator.py file. This generates random number of agents with the random start locations and random goal locations.

Random_generator:

1. get the random number for the number of agents
2. make the name of the text file to store the agents and open it to write
3. randomly assign the values for the starting and goal locations
4. check whether these locations are same with our main target agent, if so, continue
5. check whether these locations are duplicated with the made obstacles' locations, if so, continue
6. otherwise, write the locations into the file



```python
import random

def random_generator(mapname, my_map, rows, columns, start, goal):
    agent_num = random.randrange(10, 50)
    count = 0
    name = mapname  + '_random_' + str(agent_num) + '.txt'
    print("Random obstacle file " + name + " is created")
    with open(name, 'w') as f:
        while count < agent_num:
            result_dict = {}
            sx = random.randrange(0, rows)
            sy = random.randrange(0, columns)
            gx = random.randrange(0, rows)
            gy = random.randrange(0, columns)
            if (sx, sy) == start and (gx, gy) == goal:
                continue
            if my_map[sx][sy] or my_map[gx][gy]:
                continue
            tmp = [sx, sy, gx, gy]
            condition = tuple(tmp)
            if condition not in result_dict:
                result_dict[condition] = 1
                count+=1
                # print(sx , '\t' , sy , '\t' , gx , '\t' , gy)
                f.write(str(sx) + "\t" + str(sy) + "\t" + str(gx) + "\t" + str(gy))
                f.write("\n")
    return name
```

Experimental Results

Methodology questions

Among 4 different algorithms : A*, SIPP, WSIPPd, Anytime SIPP,

1. Which algorithm shows the minimum total cost?
2. What is the relationship between their total cost and the number of obstacles? How they change as the number of dynamic obstacle increases? Are there some algorithms whose performance varies less than others when certain parameters change?
    a. Which algorithms show the best or the worst total cost when the certain number of obstacles is given? How the difference between them changes as the number of obstacles increase?
    b. For each algorithm, how the total cost changes as the number of obstacles increase? Is their tendency similar?
3. Which algorithm shows the minimum CPU cost?

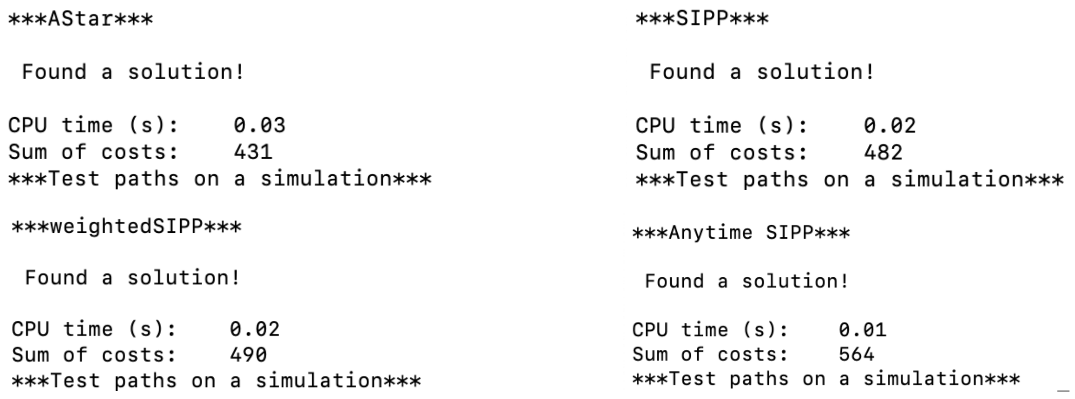
4. What is the relationship between their CPU cost and the number of obstacles? How they change as the number of dynamic obstacle increases? Are there some algorithms whose performance varies less than others when certain parameters change?
    a. Which algorithms show the best or the worst CPU cost when the certain number of obstacles is given? How the difference between them changes as the number of obstacles increase?
    b. For each algorithm, how the CPU cost changes as the number of obstacles increase? Is their tendency similar?
5. By investigating above question, which algorithm is better in which applications? Can you come up with the example of application where Anytime SIPP is better than the other algorithms?

Investigated results

With the written instruction above in "Experimental Setup" to run the code, we used random_generator function to generate random number of obstacles with random positions. On the other hand, even though in the code, the maximum range for the number of obstacles is restricted to 50, we intentionally generate 60 obstacles by assigning 60 to agent_num variable in the code to show the performance tendency. However, by considering the implemented map instance size, we restrict the range to the maximum 50 because if there are too many obstacles, it is highly possible that the program cannot find the optimal path without the collision due to the narrow road and space.
With these generated obstacle files, we ran the code for the 4 implemented algorithms : A*, SIPP, WSIPPd and AnytimeSIPP. ***All the results are submitted in the zip file as the .gif files. However, since pdf doesn't allow us to attach .gif files, we only attached the captured image of generated results in command line in this report.*** For the detail movement of the agents, please run the code with the below given command lines or refer to the submitted .gif files.

With the randomly generated file, map1.map_random_14.txt in custominstances folder, we ran the algorithms, and the results are like the Figure.3.
GIF files are stored in the submitted zip file.

```
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_14.txt --solver AStar
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_14.txt --solver SIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_14.txt --solver weightedSIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_14.txt –solver AnytimeSIPP
```



```
***AStar***

 Found a solution!

CPU time (s):    0.03
Sum of costs:    431
***Test paths on a simulation***
```

```
***SIPP***

 Found a solution!

CPU time (s):    0.02
Sum of costs:    482
***Test paths on a simulation***
```

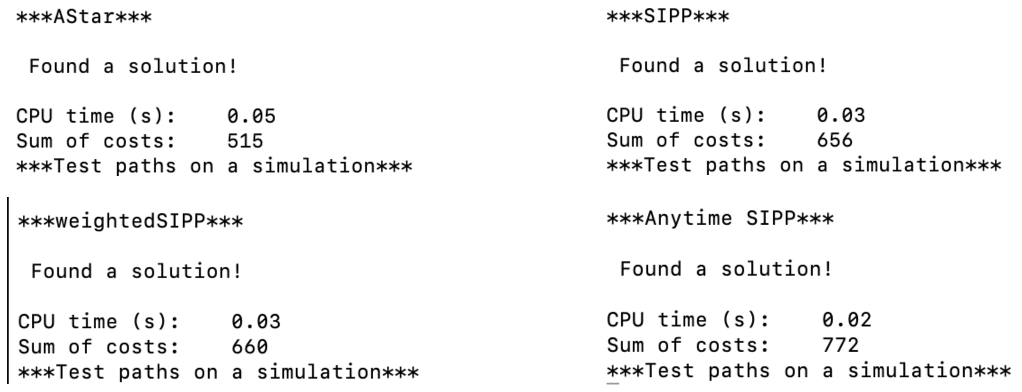
```
***weightedSIPP***

 Found a solution!

CPU time (s):    0.02
Sum of costs:    490
***Test paths on a simulation***
```

```
***Anytime SIPP***

 Found a solution!

CPU time (s):    0.01
Sum of costs:    564
***Test paths on a simulation***
```

Fig. 3. Computed Results for the implemented algorithsm for 14 agents :
A*, SIPP, WSIPPd, Anytime SIPP

With the randomly generated file, map1.map_random_21.txt in custominstances folder, we ran the algorithms, and the results are like the Figure 4.
GIF files are stored in the submitted zip file.

```
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_21.txt --solver AStar
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_21.txt --solver SIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_21.txt --solver weightedSIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/
map1.map_random_21.txt –solver AnytimeSIPP
```



```
***AStar***

 Found a solution!

CPU time (s):    0.05
Sum of costs:    515
***Test paths on a simulation***
```

```
***SIPP***

 Found a solution!

CPU time (s):    0.03
Sum of costs:    656
***Test paths on a simulation***
```

```
***weightedSIPP***

 Found a solution!

CPU time (s):    0.03
Sum of costs:    660
***Test paths on a simulation***
```

```
***Anytime SIPP***

 Found a solution!

CPU time (s):    0.02
Sum of costs:    772
***Test paths on a simulation***
```

Fig. 4. Computed Results for the implemented algorithsm for 21 agents : A*, SIPP, WSIPPd, Anytime SIPP

With the randomly generated file, map1.map_random_40.txt in custominstances folder, we ran the algorithms, and the results are like the Figure 4.

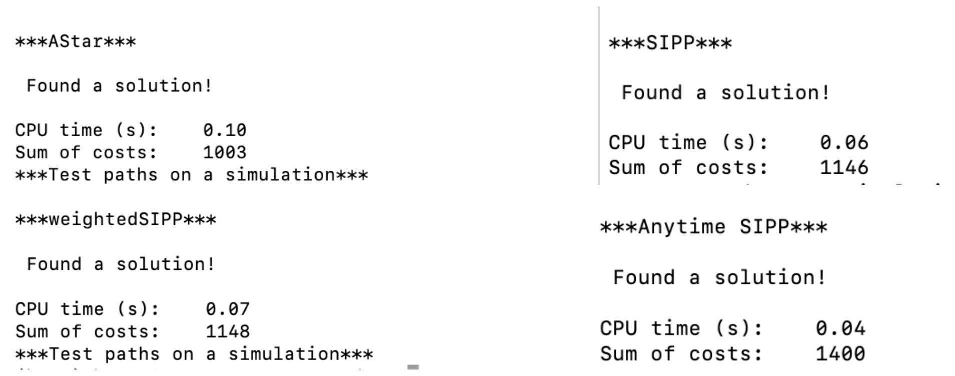
GIF files are stored in the submitted zip file.

python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_40.txt --solver AStar

python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_40.txt --solver SIPP

python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_40.txt --solver weightedSIPP

python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_40.txt –solver AnytimeSIPP



```
***AStar***

 Found a solution!

CPU time (s):    0.10
Sum of costs:    1003
***Test paths on a simulation***
```

```
***SIPP***

 Found a solution!

CPU time (s):    0.06
Sum of costs:    1146
```

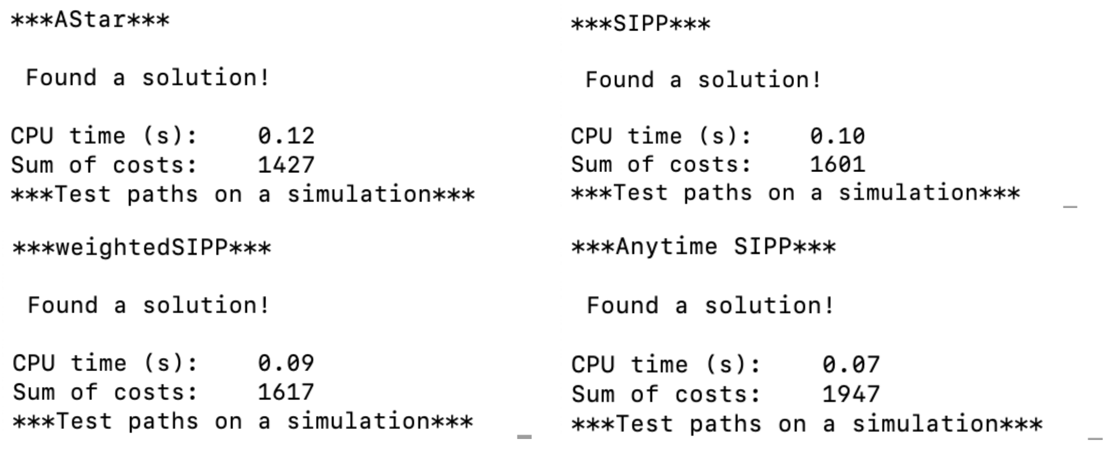
```
***weightedSIPP***

 Found a solution!

CPU time (s):    0.07
Sum of costs:    1148
***Test paths on a simulation***
```

```
***Anytime SIPP***

 Found a solution!

CPU time (s):    0.04
Sum of costs:    1400
```

Fig. 5. Computed Results for the implemented algorithsm for 40 agents : A*, SIPP, WSIPPd, Anytime SIPP

With the randomly generated file, map1.map_random_60.txt in custominstances folder, we ran the algorithms, and the results are like the Figure 4.

```
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_60.txt --solver AStar
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_60.txt --solver SIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_60.txt --solver weightedSIPP
python run_experiments.py --instance custominstances/map1.map --obstacle cusominstances/map1.map_random_60.txt –solver AnytimeSIPP
```



```
***AStar***

 Found a solution!

CPU time (s):    0.12
Sum of costs:    1427
***Test paths on a simulation***
```

```
***SIPP***

 Found a solution!

CPU time (s):    0.10
Sum of costs:    1601
***Test paths on a simulation***
```

```
***weightedSIPP***

 Found a solution!

CPU time (s):    0.09
Sum of costs:    1617
***Test paths on a simulation***
```

```
***Anytime SIPP***

 Found a solution!

CPU time (s):    0.07
Sum of costs:    1947
***Test paths on a simulation***
```

Fig. 5. Computed Results for the implemented algorithsm for 60 agents :

A*, SIPP, WSIPPd, Anytime SIPP

Agent number \ Algorithm	A*	SIPP	WSIPPd	Anytime SIPP
14	431	482	490	564
21	515	656	660	772
40	1003	1146	1148	1400
60	1427	1601	1617	1947

Fig. 6. Computed total cost for the implemented algorithsm for 14, 21, 40, 60 agents :

A*, SIPP, WSIPPd, Anytime SIPP

Agent number \ Algorithm	A*	SIPP	WSIPPd	Anytime SIPP
14	0.03	0.02	0.02	0.01
21	0.05	0.03	0.03	0.02

40	0.1	0.06	0.07	0.04
60	0.12	0.10	0.09	0.07

Fig. 7. CPU time to find the solution for the implemented algorithsm for 14, 21, 40, 60 agents : A*, SIPP, WSIPPd, Anytime SIPP

The computed results for the implemented algorithms : A*, SIPP, WSIPPd, and Anytime SIPP are clearly shown in Figure. 6. For each number of obstacles : 14, 21, 40, 60, I used the random generator function to generate these obstacles randomly.

As the graph in Figure. 8. shows, there is increase relationship between the algorithms and the total cost of movement for the given number of obstacles. As we already read in the paper, A* algorithm shows the minimum cost as it is guaranteed optimal algorithm. After that, SIPP and WSIPPd show the similar performance, but SIPP is slightly better than WSIPPd for every given number of obstacles. Anytime SIPP algorithm shows the worst performance among these 4 algorithms. When the number of obstacles are lower, for this experiment, closer to 14, the performance between 4 different implemented algorithms doesn't vary much and show reliable result. However, as the number of obstacles grows, the performance difference gets bigger, especially for the Anytime SIPP algorithm.

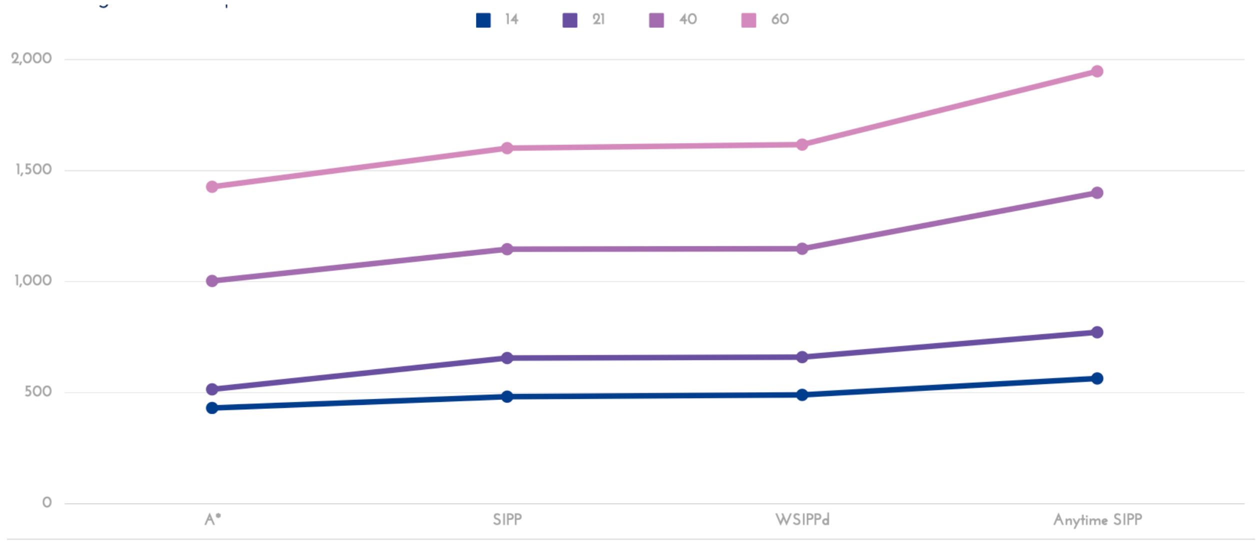


Fig. 8. The graph to show the relationship between the algorithms and the total cost of movement for the given number of obstacles

As the graph in Figure. 9. shows, the total cost increase as the number of obstacles increase for each algorithms : A*, SIPP, WSIPPd, and Anytime SIPP without exception. Especially, as the number of obstacles getting bigger, the increase of the total cost increase rapidly.

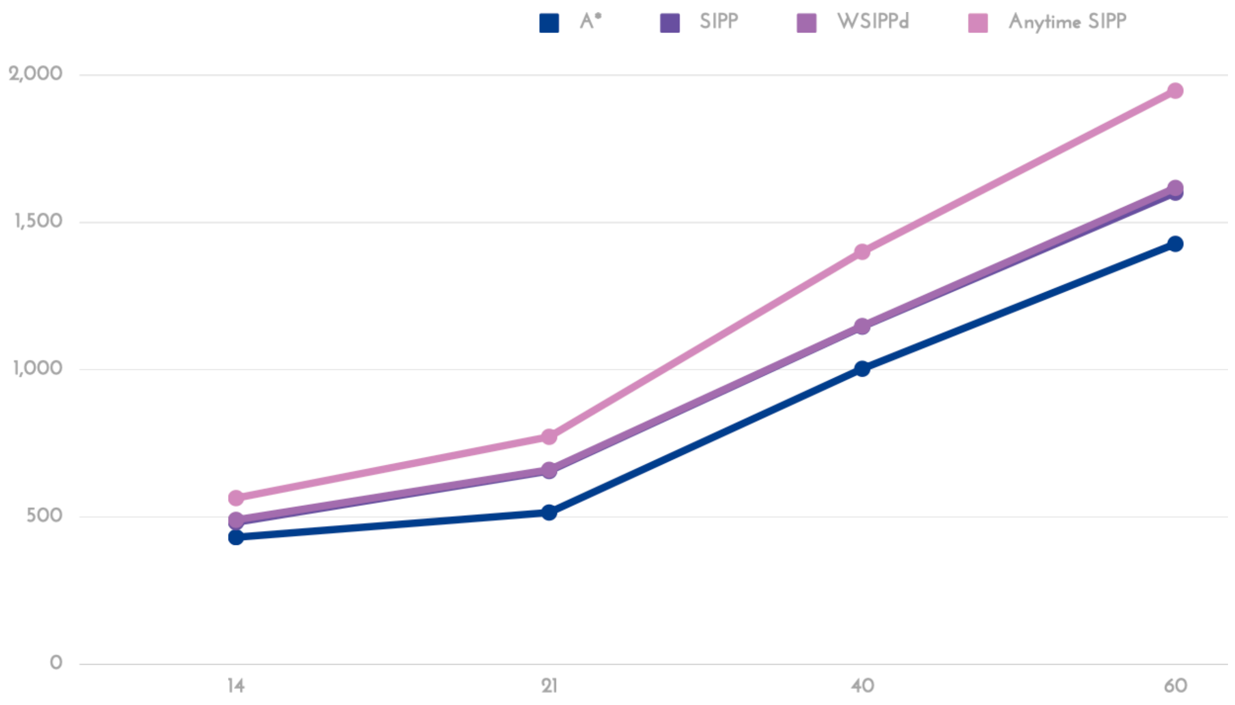


Fig. 9. The graph to show the relationship between the number of obstacles and the total cost of movement for each algorithm

This also demonstrates that when the number of obstacles is small enough, in our experiments, as it gets closer to 14, the total cost of SIPP and WSIPPd algorithms are getting closer to the total cost of A* algorithm. This is because implemented SIPP algorithm is same with the bounded-suboptimal algorithm with w = 1, and WSIPPd algorithm is known Bounded Suboptimal algorithm as the paper [2] shows. As the Figure. 10. shows how close the performance of each algorithms to the optimal solution : A* algorithm. As the number of agents getting smaller, closer to the 14 in this case, the performance varies within about 14% with A*.

Agent number \ Algorithm	A*	SIPP	WSIPPd	Anytime SIPP
14	100%	111.83%	113.69%	130.86%
21	100%	127.38%	128.16%	149.90%
40	100%	114.26%	114.46%	139.58%
60	100%	112.19%	113.31%	136.44%

Fig. 10. The comparisons between each algorithm with A* for the total cost in percentage

For the implemented map instance, and the designed target agent with the start location (17, 11) and the goal (1, 48), Anytime SIPP algorithm shows the lowest performance in the aspect of the total cost among 4 implemented algorithms.
On the other hand, in the aspect of the CPU time requested to compute the solution, the result for 4 algorithms is shown in the Figure. 7. With this data, the graph in Figure. 12. is drawn to show the relationship between the algorithms and the CPU cost for different number of obstacles. As the graph shows, Anytime SIPP shows the smallest CPU time requested to find the solution. SIPP and WSIPPd show almost similar CPU cost. Even though for the low obstacle numbers 14 and 21, they show the same CPU cost up to 2nd decimal point, SIPP shows better performance for 0.01 with 40 obstacles. However, when the number of obstacles get up to 60, WSIPPd show the better performance for 0.01. It seems their performance depend on the implemented number of obstacles, obstacles' locations, etc, but their difference is small enough to say they have almost same CPU time. A* algorithm shows the worst CPU cost among the 4 algorithms. Especially the difference of CPU cost between each algorithm increases as the number of obstacles increase.
When the number of obstacles are lower, in this experiment, closer to 14, the CPU cost of Anytime SIPP is half of the A*. The gap between these two gets smaller as the number of obstacles increase, still there is about 22% of CPU cost difference even with the 60 obstacles. These trends are clearly shown in the Figure. 13. as it shows, the CPU cost increase as the number of obstacles increase.

Agent number \ Algorithm	A*	SIPP	WSIPPd	Anytime SIPP
14	100	66.7	100.0	50.0
21	100	60.0	100.0	66.7
40	100	60.0	116.7	57.1
60	100	83.3	90.0	77.8

Fig. 11. The comparisons between each algorithm with A* for the CPU cost in percentage

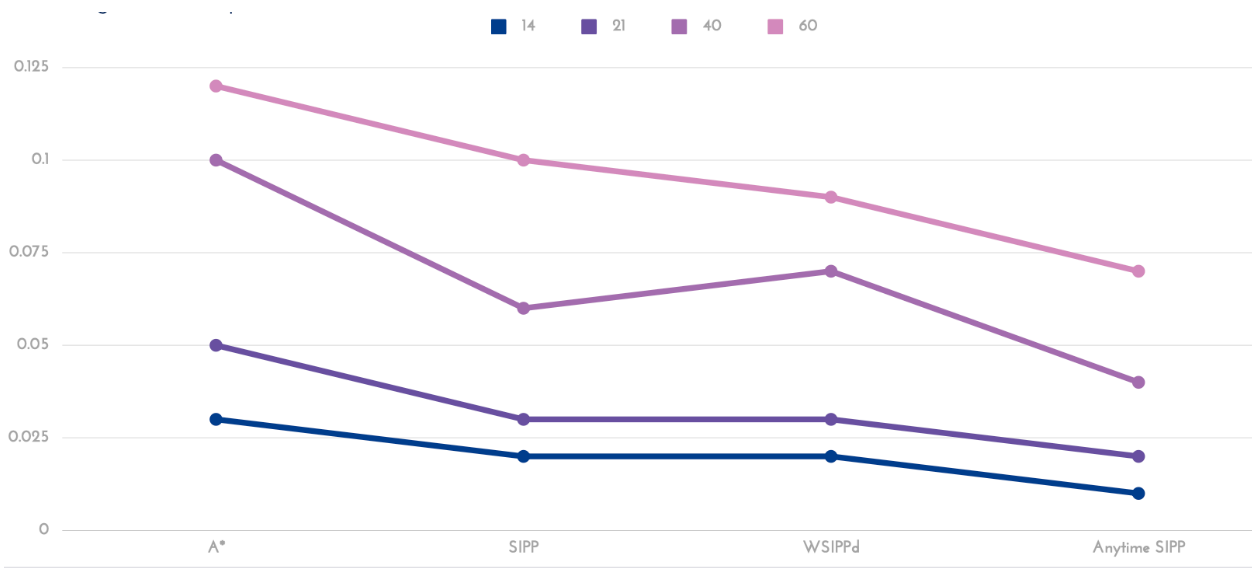


Fig. 12. The graph to show the relationship between the algorithms and the CPU cost for different number of obstacles

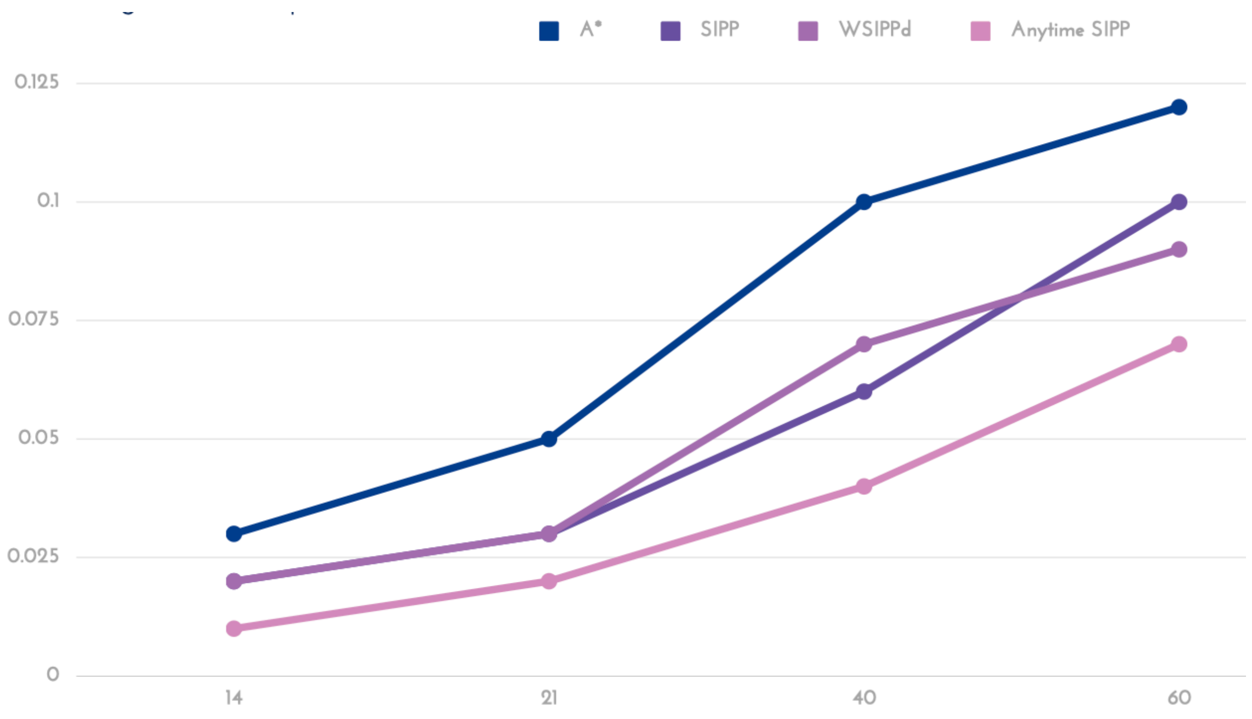


Fig. 13. The graph to show the relationship between the number of obstacles and the CPU cost for each algorithm

Conclusions

In this paper, we have implemented 4 different algorithms: A*, SIPP, WSIPPd and Anytime SIPP with the implementation of dynamic obstacles. By doing this, we could observe the performance difference between these algorithms in the aspect of total cost and CPU cost which represents the time requested to find the solution.

As the above "Experimental Results" shows, the computed results are like the following.

A* < SIPP ≈ WSIPPd < Anytime SIPP (for the total cost)

A* > SIPP ≈ WSIPPd > Anytime SIPP (for the CPU cost)

These results demonstrate that as we expected, A* algorithm returns the optimal minimum cost solution. However, the computing time for A* algorithm is almost double of Anytime SIPP with the smaller number of obstacles, and there is still about 20% difference even with the bigger number of dynamic obstacles (in this experiment, 60).

The implementation of the multi-agent path finding with the dynamic environment in the real life where it requires fast computing time more than the minimum cost path, it is better to implement Anytime SIPP algorithm rather than A* algorithm. This is especially true for the autonomous car implementation, because when the system drives the car, it is required to get the fast result to avoid the collision and make the decision, rather than having the best cost path. However, for the other implementation where computing the minimum cost is more important than fast computing, A* algorithm can be used.

For some cases, we found there are collisions between the agent and obstacles. This could be because our implemented map is small and has narrow roads on it, which cause the algorithm not being able to find the collision free path. However, to be more detail and specific, for the future work, we would like to work on solving the collision problem and implement the collision free path. Moreover, we also want to work on optimization of Anytime SIPP algorithm by modifying it to gain the less cost path or minimum cost path like A*. On the other hand, we would like to have the algorithm simulating in 3D models, so we can include more real-life environment or some other obstacles such as ambulance which cause the pause or slowing down of traffics. Also, we would like to implement these algorithms on an actual robot.

Reference

[1] Phillips, M., & Likhachev, M. (2011, May 1). SIPP: Safe interval path planning for dynamic environments. IEEE Xplore. https://doi.org/10.1109/ICRA.2011.5980306

[2] Yakovlev, Konstantin & Andreychuk, Anton & Stern, Roni. (2020). Revisiting Bounded-Suboptimal Safe Interval Path Planning. https://doi.org/10.48550/arXiv.2006.01195

[3] Sepetnitsky, V.; Felner, A.; and Stern, R. 2016. Repair policies for not reopening nodes in different search settings. In Symposium on Combi- natorial Search (SOCS), 81–88.

[4] Hang Ma.; CMPT417 Intelligent System, Lecture Note : Informed Search

[5] Narayanan, V., Phillips, M., & Likhachev, M. (2012). Anytime Safe Interval Path Planning for dynamic environments. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. https://doi.org/10.1109/iros.2012.6386191