CMPT 417 - Group Project Report

Anytime Safe-Interval Path Planning for Dynamic Environment

Fitzpatrick Laddaran	Hanra Jeong	Jooyoung Julia Lee
301282987	301449735	301414539

August 20 2022

1 Introduction

Inspired by modern autonomous vehicles, our group's original intention was to simulate a real-life situation where intelligent agents need to navigate their way through a simulated environment where both static and dynamic obstacles are in-place. In this report, we discuss our experimental results after extending the first individual project to include Safe-Interval Path Planning (SIPP), Weighted SIPP with duplicate states (WSIPPd), and Anytime Safe-Interval Path Planning (ASIPP) search algorithms. We simulated our own environment based on the structure of the instances provided in the first project.

All of the algorithms above are compared to A* search to determine their optimality, completeness, and efficiency. It is also important to note that a heuristic is used for all these algorithms, namely: the manhattan distance. The manhattan distance between two locations `x` and `y` is defined as follows:

```
def manhattan_distance(x_start, x_end, y_start, y_end):
    return |x_end - x_start| + |y_end - y_start|
```

2 Implementation

2.1 Common Functions

These are the definitions of common functions presented in the pseudocodes, referenced from [2].

```
g(s) = cost of best optimal known path of reaching a node from initial state s
h(s) = estimated cost from state s to goal state evaluated by a heuristic function
f(s) = estimated cost—minimal path to any goal state = g(s) + h(s)
```

for some state `s`.

2.2 Algorithm and Pseudocode: A*

Referenced from [2], the pseudocode for A* is as follows:

```
1  def a_star():
2      append s_start into OPEN
3      while there are unexpanded fringe nodes in OPEN
4          pick an unexpanded node n with the smallest f(n)
5          append n into CLOSED
6          if s is a goal state
7              return path(s)
8          expand n
9          for every successor of n that is not in CLOSED
10             append child into OPEN
11     return path(None)
```

The actual implementation of both `a_star()` is included in the section: Supporting Information.

2.3 Algorithm and Pseudocode: SIPP

The implementation of SIPP is based on the pseudocode found in [3]. To aid with the implementation, references were made from [3], [4] and [12] when implementing the pseudocodes. We created a class `map` with member `safe_intervals`. `safe_intervals` is simply an interval in which agents can proceed to some location without colliding. Referenced from [3], Figure 1 visualizes this explicitly.

Figure 1: Safe Interval visualization.

Furthermore, we need to accommodate for dynamic obstacles using these safe intervals. Hence, SIPP iterates through the trajectories of every obstacle. The implementation is referenced from [3], [4] and [12].

```
def dynamic_environment():
    obs = {}
    for every obstacle
        generate and append the obstacle's path into obs
        store the obstacle's location, and timestep when an agent lands on the location
        check for duplicates and store the obstacle's path
        for every location in the obstacle's path
            for interval i of location
                find time t where t < length(obstacle(position))
                store time indicating when agent located in position < startTime(i)
                if time >= length(obstacle(position))
                    update safe intervals
                from startTime(i), find t where t < length(obstacle(position))
                    store time indicating when agent located in position <= endTime(i)
                    if found
                        store time of location in obs where it is less than startTime(i)
            updated safe intervals
```

The actual implementation of `dynamic_environment()` is included in the section: Supporting Information.

`SIPP()` is quite straightforward: it uses the same approach as an A* search would, with the exception of how successors are determined (and consequently, their f-values) and how the timesteps are updated. Successors are obtained based on the pseudocode from [3]:

```
1  def getSuccessors(s):
2      successors = {}
3      for all possible movements m in M(s)
4          cfg = configuration of m applied to s
5          m_time = time to execute m
6          start_t = time(s) + m_time
7          end_t = endTime(interval(s)) + m_time
8          for all safe interval i in cfg
9              if startTime(i) > end_t or endTime(i) < start_t
10                 continue
11             t = earliest arrival time at cfg in interval i with no collisions
12             if t does not exist
13                 continue
14             s' = state of configuration cfg with interval i and time t
15             append s' into successors
16     return successors
```

Successors are generated by the earliest possible arrival time to a location which do not have any collisions along some path. `m_time` is generally set to one, as we have defined the movement from one location to another

(or the same location) as one timestep. The actual implementation of `getSuccessors(s)` is included in the section: Supporting Information.

Referenced from [3], the pseudocode for `SIPP()` is as follows:

```
def SIPP():
    append s_start into OPEN
    while there are unexpanded fringe nodes in OPEN
        pick an unexpanded node n with the smallest f(n)
        append n into CLOSED
        if s is a goal state
            return path(s)
        expand n
        for every successor of n that is not in CLOSED
            append child into OPEN
    return path(None)
```

The actual implementation of `SIPP()` is included in the section: Supporting Information.

2.4 Algorithm and Pseudocode: WSIPPd

WSIPPd is based on `SIPP()`; however, WSIPPd creates two child nodes as opposed to one: an optimal and suboptimal copy where f-values are different. Referenced from [6],

```
f(s_optimal) = w * (g(s) + h(s))
f(s_suboptimal) = g + w * h
```

where `w >= 1`. The pseudocode for `WSIPPd()` is as follows:

```
def WSIPPd():
    append s_start into OPEN
    while there are unexpanded fringe nodes in OPEN
        pick an unexpanded node n with the smallest f(n)
        append n into CLOSED
        if s is a goal state
            return path(s)
        expand n, creating two child nodes for every successor
        if optimal child c is not in CLOSED, append c into OPEN
        if suboptimal child c_sub is not in CLOSED, append c_sub into OPEN
    return path(None)
```

2.5 Algorithm and Pseudocode: ASIPP

As mentioned in [4], `ASIPP()` performs repeated weighted A* searches with decreasing values of epsilon E. E inflates the heuristic value, making the algorithm more goal-oriented. Consequently, solutions are found faster; however, the solutions does not guarantee optimality until E has reached the value one, where the algorithm is essentially just SIPP. From the pseudocode, `INCONS` is a list containing sates that a cheaper path was found for but was not expanded (as they have been appended into CLOSED). This list of states will have the chance of being re-expanded in the next iteration.

In our project, we chose epsilon prime E' = E/2, and the starting epsilon E = 15. The pseudocode is referenced from [4]:

```
def ASIPP():
    g(s_goal) = inf
    g(s_start) = 0
    OPEN = CLOSED = INCONS = {}
    append s_start into OPEN with E * h(s_start)
    ImprovePath()
    E' = min(E, g(s_goal) / min(min(g(s)+h(s)) in OPEN, min(g(s)+h(s)) in INCONS))
    publish current E' sub-optimal solution
    while E' > 1
```

```
10          decrease E
11          move states from INCONS into OPEN
12          update priorities for all nodes in OPEN according to f(s)
13          CLOSED = {}
14          ImprovePath()
15          E' = min(E, g(s_goal) / min(min(g(s)+h(s)) in OPEN, min(g(s)+h(s)) in INCONS))
16          publish current E' sub-optimal solution
```

The actual implementation of `ASIPP()` is included in the section: Supporting Information.

To guarantee optimality, `ImprovePath()` is conducted based on the pseudocode from [4]:

```
1  def ImprovePath():
2      while f(s_goal) > min(f(s) in OPEN) do
3          remove s with smallest f(s) from OPEN
4          append s into CLOSED
5          for all possible movements m in M(s)
6              x' = configuration of m applied to x(s)
7              t_m = time to execute m
8              t_start = g(s) + t_m
9              t_end = endTime(interval(s)) + t_m
10             for all safe interval i in x'
11                 if startTime(i) > t_end or endTime(i) < t_start
12                     continue
13                 t = earliest arrival time at x' in interval i with no collisions
14                 if t does not exist
15                     continue
16                 if s is optimal
17                     OPT = {true, false}
18                 else
19                     OPT = {false}
20                 for all o in OPT
21                     s' = {x',i,o}
22                     if s' was not visited before
23                         f(s') = g(s') = inf
24                     if g(s') > t
25                         g(s') = t
26                         if s' is not in CLOSED
27                             if s' is optimal
28                                 f(s') = E*(g(s') + h(s'))
29                             else
30                                 f(s') = g(s') + E * h(s')
31                             append s' into OPEN with f(s')
32                         else
33                             append s' into INCONS
```

`ImprovePath()` conducts a single weighted A* search with SIPP. This is similar to `getSuccessors()` except for the fact that it considers optimality from lines 16 and onwards.
The actual implementation of `getSuccessors()` is included in the section: Supporting Information.

2.6 Special Modifications

In our code implementation, `ImprovePath()` was modified such that lines 6-15 were not repeated by using `getSuccessors()`. Structure implementation was also referenced from [3], [4] and [12] to maximize organization.

3 Methodology

Some of the high-level questions that we wanted to answer in our experiment are as follows:

- Can we experimentally show that SIPP-based algorithms are optimal and therefore produce equivalently optimal solutions as A*?

- When should we pick SIPP-based algorithms over A*, and vice versa in the context of a simulated environment? Can you provide examples?
- Which algorithm shows the minimum total cost? What is the relationship between an algorithm's total cost and the number of obstacles? How does the relationship change as the total number of obstacles increase? Do some algorithms' performance vary less than others when parameters are changed?
- Which algorithm produces the smallest CPU cost? What is the relationship between an algorithm's CPU cost and the number of obstacles? How does the relationship change as the total number of obstacles increase? Do some algorithms' performance vary less than others when parameters are changed?

4 Experimental Setup

The experiment was implemented using Python 3.8.10 (or later). The development platform used were Visual Studio and gedit. The experiment was conducted under multiple operating systems:

- Unix-based (Ubuntu 20.04.3 LTS) virtual environment, 2048 MB of memory, Intel i5-10600K CPU @ 4.10 GHz with 6 cores
- macOS (Monterey Version 12.4) environment, 16 GB of memory, 10-core CPU

To create a .gif file, the ffmpeg version is 5.1. The matplotlib version is 3.5.1.
To run the experiment (under a macOS environment), use the following command:

```
python run_experiments.py [--instance map][--obstacle obstacle][--solver algorithm]
```

The arguments are as follows:

- —instance map: the name of the map instance file
- —obstacle obstacle: **random** to generate the obstacles randomly, **obstacle file name** to run an existing obstacle file
- —solver algorithm: **AStar** to run an A* search, **SIPP** to run a SIPP search, **weightedSIPP** to run a WSIPPd search, **AnytimeSIPP** to run a AnytimeSIPP search

More details can be found in the README file attached in the submission, or with the command (under a macOS environment):

```
python run_experiments.py -help
```

4.1 Experimental Instance

A custom instance was created to test our code. The main agent, coloured black, is assigned with a start location of (17, 11) and goal location (1, 48). Figure 2 shows the implemented instance:

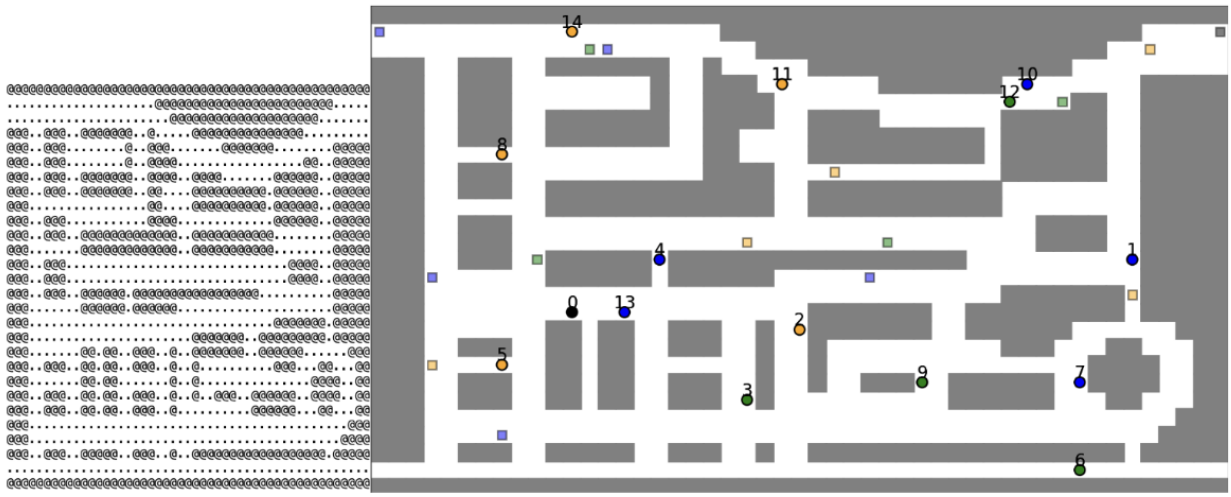

Figure 2: Custom Instance.

The left image is the map instance `map1.map` in the `custominstances` folder, and the right image is the displayed animation.

To randomly generate dynamic obstacles with a search algorithm *, run the following command (under a macOS environment):

```
python run_experiments.py --instance custominstances/map1.map --obstacle random --solver *
```

* represents any of the algorithms that we have implemented and mentioned in the previous section: Experimental Setup. The random dynamic obstacles are generated according to this pseudocode:

```
def random_generator():
    obtain a random number of agents
    make a text file to store agent information
    randomly assign start and goal locations to agents
    if agents match the locations of main agent (coloured black)
        continue
    if locations are duplicated
        continue
    append agent location into text file
```

The actual implementation of `random_generator()` is included in the section: Supporting Information.

5 Experimental Results

The results are based on the random generated file mentioned in the previous section: Experimental Instance. The implementation restricts the total number of agents to be 50; however, we intentionally generated 60 obstacles to determine the answers to our questions.

We ran the code with all four algorithms: A*, SIPP, WSIPPd, and ASIPP. The results are attached as images because attaching a .gif in a .pdf is not doable, and the submission folder on SFU Coursys is limited to 30 MB. We could not submit the .gif files. The .gif files are viewable on this repository.

The experimental results for 14, 21, 40, and 60 agents are as follows:

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

Figure 3: Computed total cost for all search algorithms with different number of agents.

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

Figure 4: CPU time for all search algorithms with different number of agents.

Plotting Figure 3:

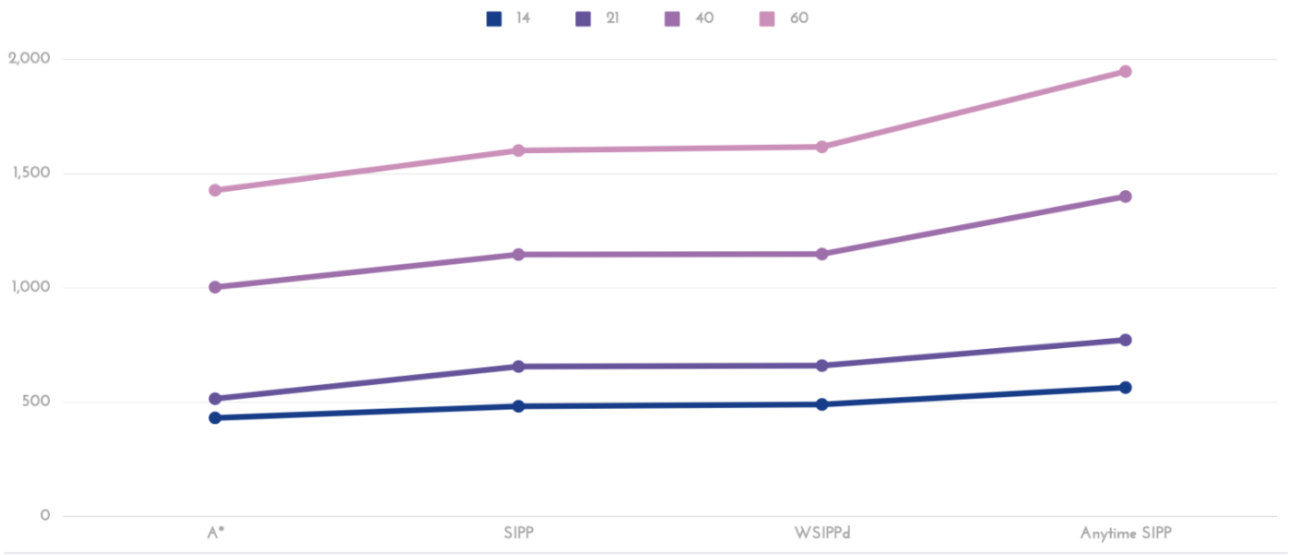

Figure 5: Plotted Figure 3 based on algorithm.

In Figure 5, it is evident that there is an increasing number of computed costs for all of the search algorithms as we increase the number of agents. Intuitively, this makes sense because more paths and obstacles are introduced as we increase the number of agents regardless of the search algorithm. As we know from [2], A* returns an optimal solution. It seems that our implementation may not be as optimal as it could be because there is a slight increase in the computed cost as we use the SIPP-based algorithms. From these four algorithms, ASIPP performs the worse.

An alternative plot for Figure 3 is as follows:

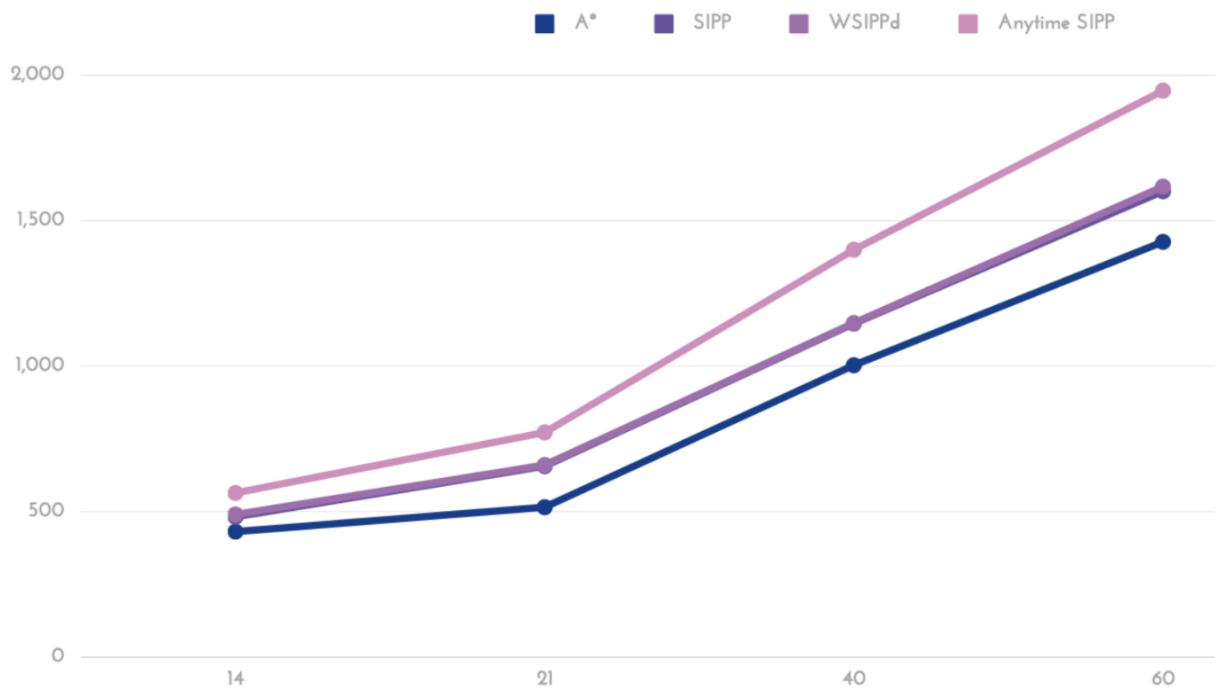

Figure 6: Plotted Figure 3 based on number of agents.

In Figure 6, we see that the total computed costs definitely do increase as we increase the number of agents. Note the uneven intervals on the x-axis.

From these plots, we also get the notion that SIPP and WSIPPd are nearly optimal (based on our implementation) as we decrease the number of obstacles; that is, it almost performs just as optimally as A*. To demonstrate this, Figure 7 shows how each of the algorithm performs in comparison to the optimal cost produced by 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%

Figure 7: Computed costs of SIPP-based algorithms compared to A* computed optimal cost.

From Figure 7, we see that ASIPP performs the worse for the different number of agents.

Finally, we plot Figure 4.

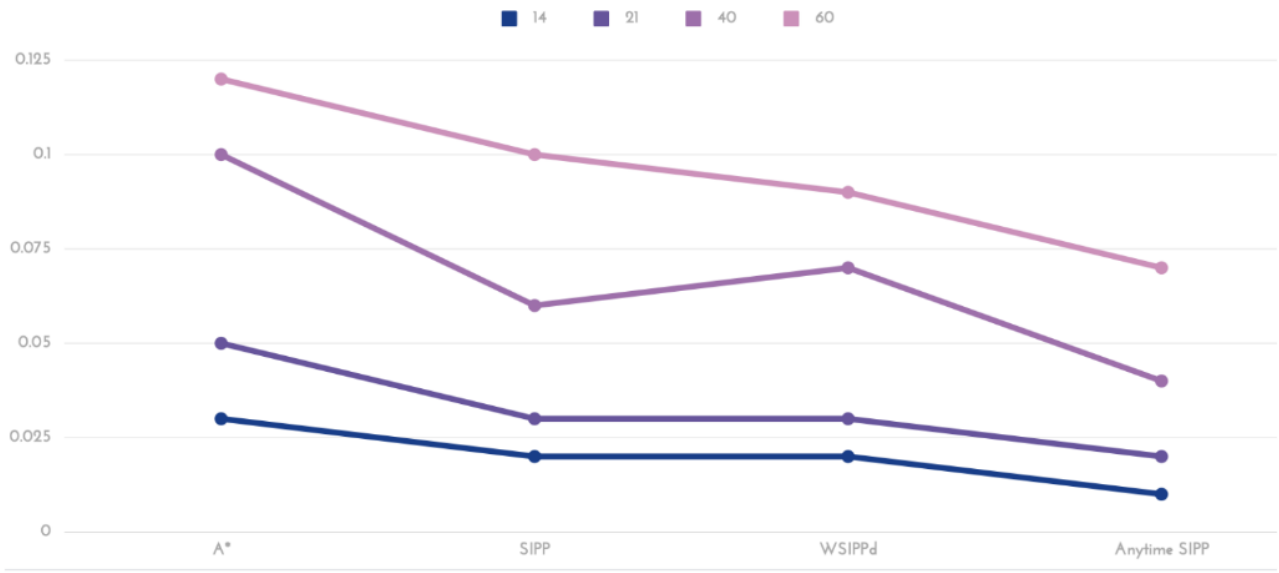

Figure 8: Plotted Figure 4 based on algorithm.

From Figure 8, we see that ASIPP requires the smallest CPU time to find a solution, SIPP and WSIPPd are similar in CPU time, while A* requiring the longest CPU time. This is evident because ASIPP will return a sub-optimal solution ASAP, unlike A* which takes longer because it returns an optimal solution. The anomaly within Figure 8 is the slight decrease in computation time between SIPP and WSIPPd when there are 40 agents. This could be due to the random generator generating easier paths.

Figure 9 presents an intuitive story similar to Figure 6, where the number of agents also increase the CPU runtime.

Figure 10 shows the same comparison as Figure 7 with regards to the CPU time. The percentage ratios are a little difficult to evaluate due to the short runtime of each of the algorithms.

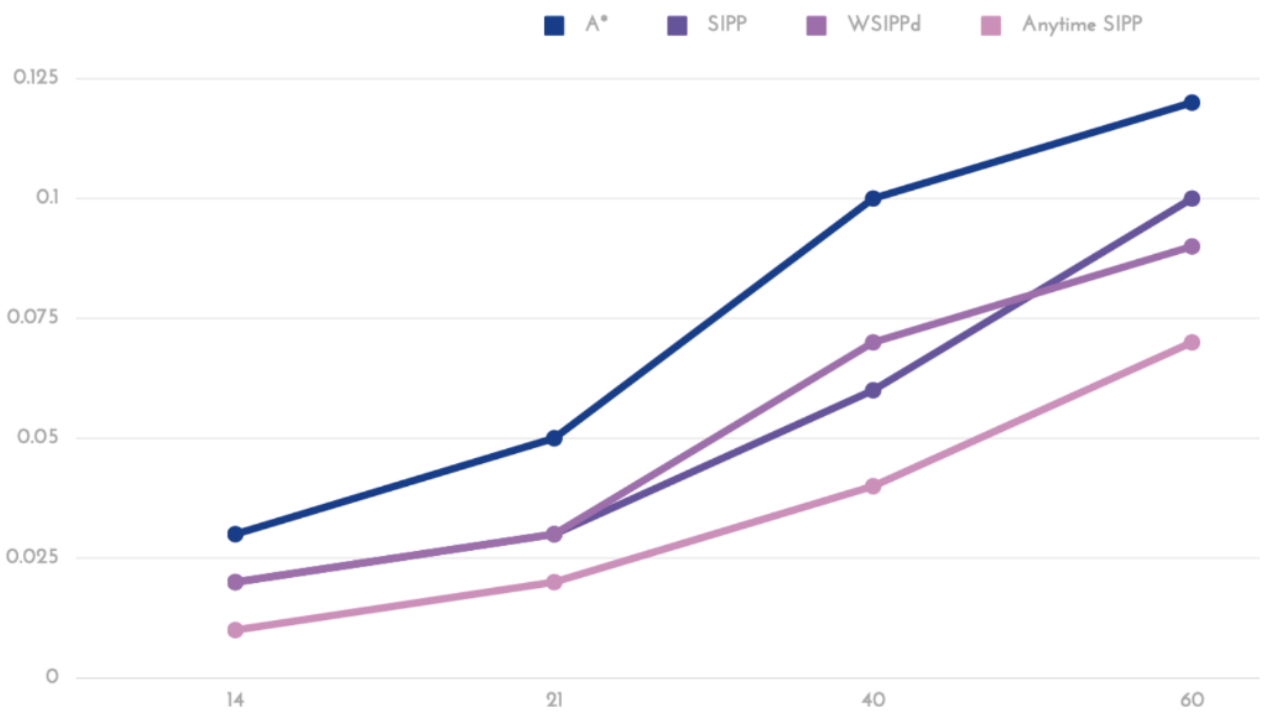

Figure 9: Plotted Figure 4 based on number of agents.

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

Figure 10: CPU runtime of SIPP-based algorithms compared to A* CPU runtime.

6 Conclusion

In this paper, we have implemented four different algorithms: A*, SIPP, WSIPPd, and ASIPP. We obsereved the performance differences between each of the algorithms with regards to the optimal cost and also the CPU runtime. Based on our experimental results, we conclude that:

- A* $<$ SIPP $\approx$ WSIPPd $<$ ASIPP (computed costs)
- A* $>$ SIPP $\approx$ WSIPPd $>$ ASIPP (CPU costs)

These are expected results because A* returns an optimal minimum cost solution at the cost of having a longer CPU runtime. ASIPP produces a suboptimal-solution way faster. As to determine on how we select algorithms, the situation varies depending on the environment. It seems that simulated environments are a lot more controlled; hence, optimal solutions may be ideal; however, ASIPP may be more preferable in dynamic environments (such as in real life) where fast computing time is (arguably) preferred. For example, ASIPP would likely be more ideal with autonomous vehicles.

Furthermore, we think that it would be very cool if we were to simulate the algorithms in a real-life environment with real robots. To do this, we want to ensure that collision-free paths are found between all agents because we found that some collisions still existed when ran on the map with randomly generated obstacles. We could also improve the quality of the code by organizing the structure and minimizing excess code. We could also optimize ASIPP by modifying it to gain the least cost path similar to A*.

7 Supporting Information

7.1 Implementation of A* search algorithm

```
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
```

Figure 11: Implementation of `a_star()`.

7.2 Implementation of the Dynamic Environment function

```python
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
```

Figure 12: Implementation of `dynamic_enviroment()`.

7.3 Implementation of SIPP search algorithm

```
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
```

Figure 13: Implementation of `SIPP()`.

7.4 Implementation of ASIPP search algorithm

```
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
```

Figure 14: Implementation of `ASIPP()`.

7.5 Implementation of the Successor function

```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
```

Figure 15: Implementation of `getSuccessors(s)`.

7.6 Implementation of the Random Generator function

```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
```

Figure 16: Implementation of `random_generator()`.

8 References

[1] Swift, N. (2020, May 29). Easy A* (star) pathfinding. Medium. Retrieved August 20, 2022, from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

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

[3] 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

[4] 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

[5] 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.

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

[7] M. Likhachev, G. Gordon, and S. Thrun. ARA: Anytime A with provable bounds on sub-optimality. In Advances in Neural Information Processing Systems (NIPS) 16. Cambridge, MA: MIT Press, 2003

[8] A. Kushleyev and M. Likhachev. Time-bounded lattice for efficient planning in dynamic environments. Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), 2009.

[9] J. van den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In Proceedings of the IEEE International Conference on Robotics and Automation (ICRA), pages 2366–2371, 2006.

[10] D. Silver. Collaborative pathfinding. In Proceedings of AIIDE, 2005.

[11] [Cohen et al. 2019] Cohen, L.; Uras, T.; Kumar, T. S.; and Koenig, S. 2019. Optimal and bounded-suboptimal multiagent motion planning. In Symposium on Combinatorial Search.

[12] Artser, E. (n.d.). Eartser/Sipp-MAPF: Prioritized multi-agent path finding using safe interval path planning (SIPP) and modifications. GitHub. Retrieved August 20, 2022, from https://github.com/eartser/sipp-mapf

[13] A* Algorithm Pseudocode. Optimisation 2009-2021. (n.d.). Retrieved August 20, 2022, from https://mat.uab.cat/alseda/MasterOpt/AStar-Algorithm.pdf