SSN College of Engineering

Department of Computer Science and Engineering

CS1504 — Artificial Intelligence

2021 – 2022

Assignment — 04 (Additional)
(State Space Search — Mobile Robot)

August 25, 2021

Problem Statement

Consider an autonomous mobile robot in a crowded environment that needs to find an efficient path from its current location S to a desired location G. As an idealization of the situation, assume that the obstacles (whatever they may be) are abstracted by polygons. The problem now reduces to finding the shortest path between two points in a plane that has convex polygonal obstacles

a. How do we formulate the state-space? How many states are there? How many paths are there to the goal? Think carefully to define a good state-space. Justify your decisions.

b. Formulate this problem in Python by subclassing the Problem class in "search.py" of the reference implementation

c. Define your evaluation function to evaluate the goodness or badness of a state

d. Create several instances (at least 100) of this problem by randomly generating planes with random start and goal points and random polygons as obstacles

e. Solve all the instances using the following search strategies
   - Any basic strategy of your choice (DFS/BFS/IDS)
   - Best-first greedy search
   - A* search

f. Perform an empirical analysis in terms of number of nodes generated, expanded, actual time taken, completeness, optimality, etc. Which algorithm performs better, in general, on all the instances?

Responses

1. Formulated below (State Space Formulation)

2. Implemented in python (Python Program Code)

3. Evaluation function for each search strategy defined in respective modules (BestFirstSearch.py - Script to implement Best First Greedy Search, AStarSearch.py - Script to implement A* Search)

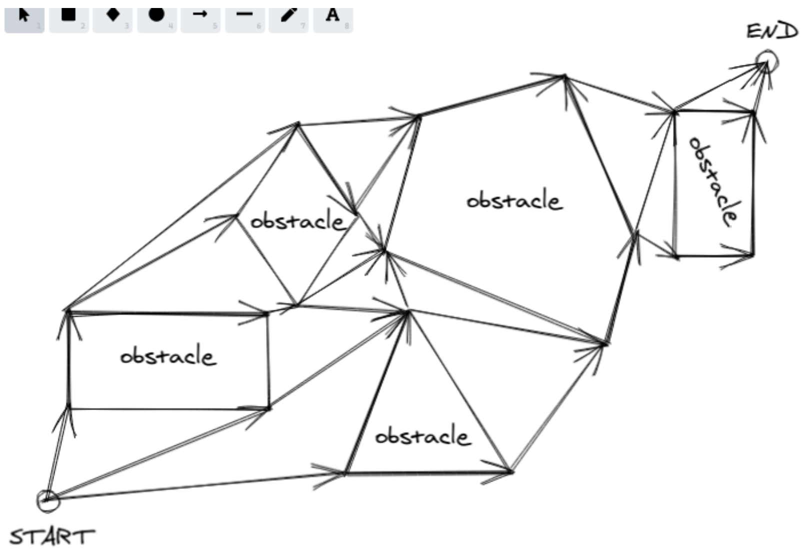
4. Random Instances Generated using script (InstanceGenerator.py - Script to generate problem instances and retur)

5. Search strategies implemented (BreadthFirstSearch.py - Script to perform BFS, BestFirstSearch.py - Script to implement Best First Greedy Search, AStarSearch.py - Script to implement A* Search)

6. Empirical Analysis performed in script (main.py - Driver program to implement and summarize the search). Outputs attached (Sample Output)

(*Link to* ***repl.it*** *implementation repository*)

State Space Formulation



State: Being at a vertex
(or) start/finish point

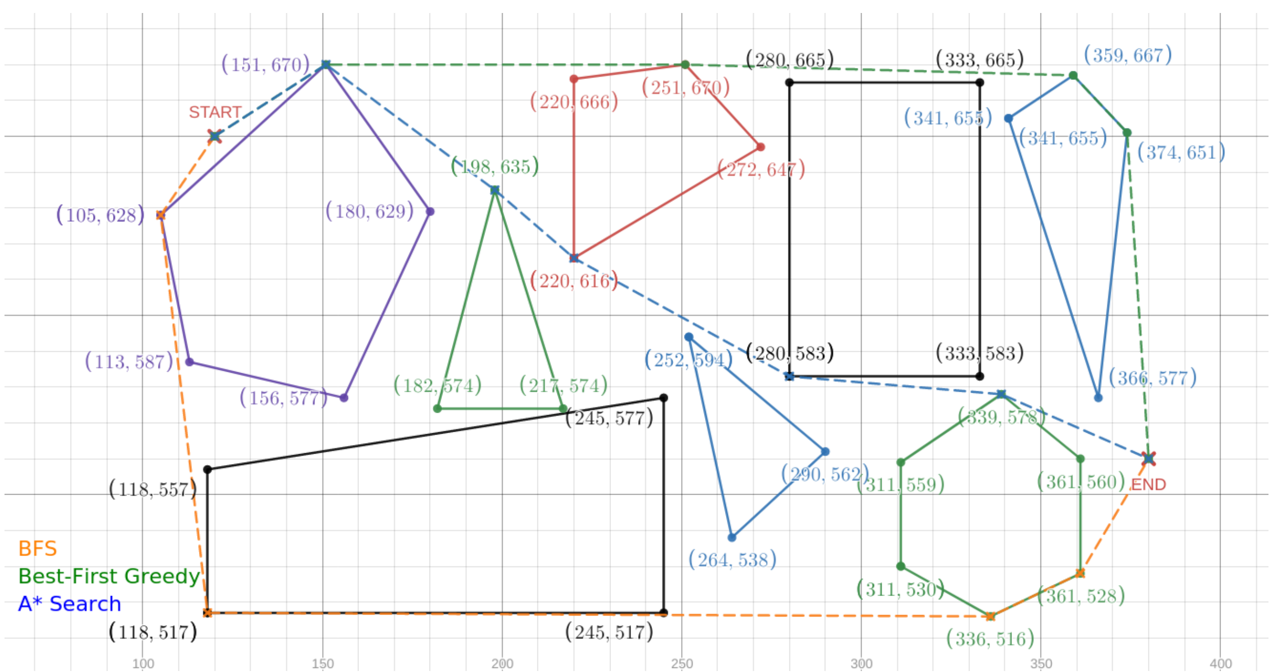
Action: Moving to a non-obstructed
vertex/pt

Heuristic: SLD to finish point

Path Cost: Actual length of each
state transition

Representation
Start and Finish : (x,y)
Polygons : Vector < (x1,y1),...(xn,yn) >
sequentially specified

Sample Test Case (with Solution found using Code)



Python Program Code

1. Queue.py - Script for Queue ADT

```
class Queue:
   # [HEAD, ....... , TAIL]
   def __init__(self, data_list=None):
       self.data = list()
       self.size = 0
       if data_list is not None:
           self.data.extend(data_list)
           self.size = len(data_list)

   def enqueue(self, data_list):
       self.data.extend(data_list)
       self.size += len(data_list)

   def dequeue(self):
       if self.size == 0:
           return None
       self.size -= 1
       return self.data.pop(0)

   def get_front(self):
       # Show first element of the queue
       return self.data[0]

   def set_back(self, value):
       # Modify first elemnent of the queue
       self.data[0] = value

   def get_back(self):
       # Show last element of the queue
       return self.data[-1]

   def set_back(self, value):
```

```
        # Modify last elemnent of the queue
        self.data[-1] = value

    def get_contents(self):
        return self.data

    def is_empty(self):
        return self.size==0
```

2. <u>PriorityQueue.py</u> - Script for PriorityQueue ADT (with custom ordering for each search)

```
class PriorityQueue:
    # Stores elements (states) along with some payload associated
with each of them

    def __init__(self, priority_func):
        # 'contexts' contains the data required to resolve the
priority of elements
        # 'priority' is a function that returns the sort key ->
priority(elem, its_context)
        self.data = []
        self.contexts = []
        self.payloads = []
        self.priority = priority_func
        self.size = 0

    def sort_elements(self, elems, contexts, payloads):
        for i in range(1, len(elems)):
            j = i-1
            while j>=0 and
self.priority(elems[j],contexts[j])>self.priority(elems[i],contexts[
i]):
                elems[j] = elems[j-1]
                contexts[j] = contexts[j-1]
                payloads[j] = payloads[j-1]
                j -= 1
            elems[j+1] = elems[i]
            contexts[j+1] = contexts[i]
            payloads[j+1] = payloads[i]
        return elems, contexts, payloads
```

```python
    def enqueue(self, elems, contexts, payloads):
        # 'contexts' contains the data required to resolve the
priority of elements
        # The resolution is done using the 'priority_func' function
        # Sort the insertion elements
        elems, contexts,payloads = self.sort_elements(elems,
contexts, payloads)
        # Merge with current data
        data_i = 0
        elem_j = 0
        merged_data = []
        merged_contexts = []
        merged_payloads = []
        while data_i<self.size and elem_j<len(elems):
            if self.priority(self.data[data_i],
self.contexts[data_i]) <= self.priority(elems[elem_j],
contexts[elem_j]):
                merged_data.append(self.data[data_i])
                merged_contexts.append(self.contexts[data_i])
                merged_payloads.append(self.payloads[data_i])
                data_i += 1
            else:
                merged_data.append(elems[elem_j])
                merged_contexts.append(contexts[elem_j])
                merged_payloads.append(payloads[elem_j])
                elem_j += 1
        if data_i<self.size:
            merged_data.extend(self.data[data_i:])
            merged_contexts.extend(self.contexts[data_i:])
            merged_payloads.extend(self.payloads[data_i:])
        else:
            merged_data.extend(elems[elem_j:])
            merged_contexts.extend(contexts[elem_j:])
            merged_payloads.extend(payloads[elem_j:])
        # Update data and contexts
        self.data = merged_data
        self.contexts = merged_contexts
        self.payloads = merged_payloads
        self.size += len(elems)
```

```
    def dequeue(self):
        if self.is_empty():
            return None
        self.size -= 1
        return self.data.pop(0), self.payloads.pop(0)

    def is_empty(self):
        return self.size==0
```

3. geometry.py - Classes and Functions to represent and manipulate 2D space elements

```
from numpy import inf
from math import pi, atan, degrees

class Point:

    def __init__(self, coords):
        self.x = coords[0]
        self.y = coords[1]

    def distance_to(self, other_pt):
        horizontal_sq = (self.x-other_pt.x)**2
        vertical_sq = (self.y-other_pt.y)**2
        return (horizontal_sq+vertical_sq)**0.5

    def __eq__(self, other_pt):
        return self.x == other_pt.x and self.y==other_pt.y

    def __str__(self):
        return '({x},{y})'.format(x=self.x, y=self.y)

    def __hash__(self):
        # Represent Point as a 'tuple' and use tuple's hash function
        return hash((self.x, self.y))
```

```python
class Vector:

   def __init__(self, src, destn):
       # src and destn are Point instances
       self.x = destn.x - src.x
       self.y = destn.y - src.y
       self.quadrant = self.get_quadrant()
       self.direction = self.get_direction()

   def get_direction(self):
       # By defn, theta of vector wrt x-axis
       # Set offset as per quadrant
       offset = pi if self.quadrant in (2,3) else 0
       if self.x==0:
           # Attach sign of y to infinity
           tan_theta = inf*(self.y)
       else:
           tan_theta = atan(self.y/self.x)
       direction = offset + atan(tan_theta)
       return Vector.normalize_angle(direction)

   def get_quadrant(self):
       if self.x>=0:
           if self.y>=0:
               return 1
           else:
               return 4
       else:
           if self.y>=0:
               return 2
           else:
               return 3

   def get_relative_direction(self, other_vector):
       return Vector.normalize_angle(self.direction -
other_vector.direction)

   def is_zero_vector(self):
       return self.x == 0 and self.y==0
```

```
    @staticmethod
    def normalize_angle(angle):
        # Return angle in rannge [0,360]
        while angle<0:
            angle += 2*pi
        return angle


class Polygon:

    def __init__(self, vertices):
        self.vertices = [ Point(vertex) for vertex in vertices ]
        self.edges = self.find_edges()

    def find_edges(self):
        # Polygon Edge is an ordered set (tuple) of two adjacent
points
        edges = []
        for i in range(len(self.vertices)-1):
            edges.append((self.vertices[i], self.vertices[i+1]))
        edges.append((self.vertices[-1], self.vertices[0]))
        return edges

    def __str__(self):
        display_string = '{sides}-sided Polygon : {vertices}'.format(
            sides=len(self.vertices),
            vertices='{'+', '.join(map(str, self.vertices))+'}'
        )
        return display_string


def has_duplicates(points):
    return len(set(points))!=len(points)


def segments_intersect(seg_1, seg_2):
    # If any of the points are repeated, they touch => No obstruction
    if has_duplicates(seg_1+seg_2):
```

```
        return False
    # Check if seg_1 intersects/touches seg_2
    # If the orientation of seg_1 wrt either ends of seg_2 are
different (or one is collinear), they intersect
    orient12_1 = get_orientation((seg_1[0], seg_1[1], seg_2[0]))
    orient12_2 = get_orientation((seg_1[0], seg_1[1], seg_2[1]))
    # (not any([orient12_1, orient12_2])) ==> Check if one of them is
0
    if (not any([orient12_1, orient12_2])) or orient12_1==orient12_2:
        return False
    # Check if seg_2 intersects/touches seg_1
    orient21_1 = get_orientation((seg_2[0], seg_2[1], seg_1[0]))
    orient21_2 = get_orientation((seg_2[0], seg_2[1], seg_1[1]))
    if (not any([orient21_1, orient21_2])) or orient21_1==orient21_2:
        return False
    return True

def get_orientation(three_pt_sequence):
    # Returns the orientation of a sequence of three points
    # 0: Collinear, -1: Clockwise, +1: Anti-Clockwise
    pt1, pt2, pt3 = three_pt_sequence
    vector_1 = Vector(pt1, pt2)
    vector_2 = Vector(pt2, pt3)
    # Check if adjacent points are same - Direction can't be found
    if vector_1.is_zero_vector() or vector_2.is_zero_vector():
        return 0
    # Find angle between vectors
    relative_direction =  vector_2.get_relative_direction(vector_1)
    # Determine orientation
    if relative_direction < pi:
        return -1
    elif relative_direction > pi:
        return 1
    else:
        return 0
```

4. StateFormulation.py - Script to formulate the state space and instantiate a problem case

```
from geometry import *

class StateSpace:

   def __init__(self, start, end, obstacles):
       # State Space Constants
       self.start = Point(start)
       self.end = Point(end)
       self.obstacles = obstacles
       self.poly_edges = self.get_poly_edges()
       self.states = [self.start, self.end] +
self.get_poly_vertices()
       self.visited = set()
       # Current State Representation
       self.curr_state = None
       self.curr_polygon = None
       # Store the polygon of current state

   def move_to_state(self, new_state):
       prev_state = self.curr_state
       self.curr_state = new_state
       self.curr_polygon = self.get_curr_polygon()
       if prev_state is not None:
           self.visited.add(prev_state)

   def get_poly_vertices(self):
       # Poly_Edges is the set of all edges of the obstacles in the
state space
       vertices = []
       for polygon in self.obstacles:
           vertices.extend(polygon.vertices)
       return vertices

   def get_poly_edges(self):
       # Poly_Edges is the set of all edges of the obstacles in the
state space
       edges = []
       for polygon in self.obstacles:
```

```
            edges.extend(polygon.edges)
        return edges

    def get_curr_polygon(self):
        # Check if move was on same polygon
        if self.curr_polygon and self.curr_state in
self.curr_polygon.vertices:
            return self.curr_polygon
        # Find polygon
        for polygon in self.obstacles:
            if self.curr_state in polygon.vertices:
                return polygon
        # Set to None for terminal points
        return None

    def at_goal_state(self):
        return self.curr_state == self.end

    def is_visited(self, state):
        return state in self.visited

    def is_reachable(self, destn, src):
        # Already checked that destn and src are not in the same
polygon
        # Check if the path from src to destn intersects any other
edge
        path = (src, destn)
        for edge in self.poly_edges:
            if segments_intersect(edge, path):
                return False
        return True

    def get_next_states(self, include_visited=False):
        # Excludes visited states
        # Either: 1. Adjacent vertex in same polygon
        #     Or: 2. Non-obstructing path to vertex of other polygon
        next_states = []
        for state in self.states:
            # Check if state is same as current
            if state == self.curr_state:
```

```
                continue
            # Skip if state is visited
            if not include_visited and self.is_visited(state):
                continue
            # Case: State is part of same polygon (grazing allowed)
            if self.curr_polygon and (state in
self.curr_polygon.vertices):
                # Allow if it is an adjacent vertex
                num_vertices = len(self.curr_polygon.vertices)
                for i in range(num_vertices):
                    if self.curr_polygon.vertices[i] ==
self.curr_state:
                        # Check if state is previous or next vertex
                        if state in (
self.curr_polygon.vertices[(i+1)%num_vertices],
self.curr_polygon.vertices[i-1] ):
                            next_states.append(state)
                continue
            # Case: State is not part of same polygon
            if self.is_reachable(state, self.curr_state):
                next_states.append(state)
        return next_states

    def __str__(self):
        display_string = "Start: {start}\nGoal:
{goal}\nPolygons:\n{polygon}".format(
            start=self.start,
            goal=self.end,
            polygon='\n'.join(map(str, self.obstacles))
        )
        return display_string
```

5. search_utils.py - Utilities for search methods

```
class Path:

    def __init__(self, sequence=[], cost=0):
        self.sequence = sequence
        self.cost = cost
```

```
    def add_state(self, state):
        self.cost += self.sequence[-1].distance_to(state)
        self.sequence.append(state)

    def __str__(self):
        return " -> ".join(map(str, self.sequence))+'\nCost:
{cost}'.format(cost=self.cost)
```

6. BreadthFirstSearch.py - Script to perform BFS

```
from Queue import Queue
from copy import deepcopy

from StateFormulation import *
from search_utils import *

def search(state_space):

    # Queue of fringe states and their respective paths
    state_queue = Queue([state_space.start])
    path_queue = Queue([Path([state_space.start])])
    # Node Counters
    visited_cnt = 0
    generated_cnt = 0
    while(not state_queue.is_empty()):
        # PRE-VISIT
        state = state_queue.dequeue()
        path_to_state = path_queue.dequeue()
        if state_space.is_visited(state):
            continue
        # VISIT
        # Move to state and do goal test
        state_space.move_to_state(state)
        visited_cnt += 1
        if state_space.at_goal_state():
            return path_to_state, generated_cnt, visited_cnt
        # POST-VISIT
        # Find fringe and add to queue
        fringe = state_space.get_next_states(include_visited=False)
```

```
        state_queue.enqueue(fringe)
        # Store paths to each fringe
        for next_state in fringe:
            next_path = deepcopy(path_to_state)
            next_path.add_state(next_state)
            path_queue.enqueue([next_path])
        generated_cnt += len(fringe)

    return False

# Solution Found
# (120,650), (105,628), (118,517), (336,516), (361,528), (380,560)
# Cost: 421.3344534244741
```

7. BestFirstSearch.py - Script to implement Best First Greedy Search

```
from PriorityQueue import PriorityQueue
from copy import deepcopy

from StateFormulation import *
from search_utils import *

def heuristic(state, goal_state):
    # SLD heuristic is used
    return state.distance_to(goal_state)

def evaluate_priority(state, context):
    # 'context' contains data reqd to evaluate priority of state
    # Here, it is the goal_state
    (goal_state,) = context
    return heuristic(state, goal_state)

def search(state_space):

    # Priority Queue of fringe states and their respective paths as
payloads
    state_queue = PriorityQueue(evaluate_priority)
    state_queue.enqueue(
        elems=[state_space.start],
```

```
        contexts=[(state_space.end,)],
        payloads=[Path([state_space.start])]
    )
    # Node Counters
    visited_cnt = 0
    generated_cnt = 0
    while(not state_queue.is_empty()):
        # PRE-VISIT
        state, path_to_state = state_queue.dequeue()
        if state_space.is_visited(state):
            continue
        # VISIT
        # Move to state and do goal test
        state_space.move_to_state(state)
        visited_cnt += 1
        if state_space.at_goal_state():
            return path_to_state, generated_cnt, visited_cnt
        # POST-VISIT
        # Find fringe and add to queue
        # Store paths to each fringe as its payload
        fringe = state_space.get_next_states(include_visited=False)
        for next_state in fringe:
            next_path = deepcopy(path_to_state)
            next_path.add_state(next_state)
            state_queue.enqueue(
                elems=[next_state],
                contexts=[(state_space.end,)],
                payloads=[next_path]
            )
        generated_cnt += len(fringe)

    return False

# Solution Found
# (120,650), (151,670), (251,670), (359,667), (374,651), (380,560)
# Cost: 358.0626920104638
```

8. AStarSearch.py - Script to implement A* Search

```
from PriorityQueue import PriorityQueue
from copy import deepcopy

from StateFormulation import *
from search_utils import *

def heuristic(state, goal_state):
   # SLD heuristic is used
   return state.distance_to(goal_state)

def evaluate_priority(state, context):
   # 'context' contains data reqd to evaluate priority of state
   # Here, it is the path_cost_so_far and goal_state
   (path_cost, goal_state) = context
   return path_cost + heuristic(state, goal_state)

def search(state_space):

   # Priority Queue of fringe states and their respective paths as
payloads
   state_queue = PriorityQueue(evaluate_priority)
   path_to_start = Path([state_space.start])
   state_queue.enqueue(
       elems=[state_space.start],
       contexts=[(path_to_start.cost, state_space.end)],
       payloads=[path_to_start]
   )
   # Node Counters
   visited_cnt = 0
   generated_cnt = 0
   while(not state_queue.is_empty()):
       # PRE-VISIT
       state, path_to_state = state_queue.dequeue()
       if state_space.is_visited(state):
           continue
       # VISIT
       # Move to state and do goal test
```

```
        state_space.move_to_state(state)
        visited_cnt += 1
        if state_space.at_goal_state():
            return path_to_state, generated_cnt, visited_cnt
        # POST-VISIT
        # Find fringe and add to queue
        # Store paths to each fringe as its payload
        fringe = state_space.get_next_states(include_visited=False)
        for next_state in fringe:
            next_path = deepcopy(path_to_state)
            next_path.add_state(next_state)
            state_queue.enqueue(
                elems=[next_state],
                contexts=[(next_path.cost, state_space.end)],
                payloads=[next_path]
            )
        generated_cnt += len(fringe)

    return False

# Solution Found
# (120,650), (151,670), (198,635), (220,616), (280,583), (339,578),
(380,560)
# Cost: 297.02594348473116
```

9. InstanceGenerator.py - Script to generate problem instances and return state spaces to run

```

from StateFormulation import *
from geometry import *
import numpy.random as random

# CONSTANTS (modify as required)
MIN_OBSTACLES = 10
MAX_OBSTACLES = 20
MIN_VERTICES_POLY = 3
MAX_VERTICES_POLY = 10
MIN_COORDINATE_VALUE = -200
```

```
MAX_COORDINATE_VALUE = +200



def generate_coordinates():
   # Return a random 2D point within the coordinate limits
   return (random.randint(MIN_COORDINATE_VALUE,
MAX_COORDINATE_VALUE+1),
           random.randint(MIN_COORDINATE_VALUE,
MAX_COORDINATE_VALUE+1))



def generate_polygon():
   # Randomly determine number of vertices and generate points
   num_vertices = random.randint(MIN_VERTICES_POLY,
MAX_VERTICES_POLY+1)
   vertices = [ generate_coordinates() for k in range(num_vertices)
]
   return Polygon(vertices)

def generate_state_space():
   # Randomly determine number of obstacles and generate polygons
   num_obstacles = random.randint(MIN_OBSTACLES, MAX_OBSTACLES+1)
   polygons = []
   for i in range(num_obstacles):
       polygons.append(generate_polygon())
   # Make State-Space
   state_space = StateSpace(
       start=generate_coordinates(),
       end=generate_coordinates(),
       obstacles=polygons
   )
   return state_space
```

10. main.py - Driver program to implement and summarize the search techniques

```
import time
from copy import deepcopy
```

```python
from numpy import inf

from StateFormulation import *
from InstanceGenerator import generate_state_space
import BreadthFirstSearch as BFS
import BestFirstSearch as Best_Greedy
import AStarSearch as AStar


def run_with_timer(function, args):
   start = time.time()
   results = function(*args)
   return time.time()-start, results


def display_summary(exec_time, results):
   if results:
       print("Path Found")
       print("Cost of Solution:", results[0].cost)
       print("No. of Nodes Generated:", results[1])
       print("No. of Node Expanded: ", results[2])
   else:
       print("Path NOT Found")
   print("Time Taken: {time}s".format(time=exec_time))


def display_overall_summary(completes, found, optimal, gen, visited,
time, N_INSTANCES):
   print("No. of Completions:", completes)
   print("No. of cases where Path Found: ", found)
   print("No. of Optimal Paths: ", optimal)
   print("No. of Nodes Generated: ", gen)
   print("No. of Nodes Visited: ", visited)
   print("Avg. Time Taken: {}s".format(time/N_INSTANCES))


line = '-'*50

# SIMPLE MANUAL TEST CASE (diagrams attached in 'Documentation')
print(line)
```

```
print("MANUAL TEST CASE")
print(line)

# Make Polygonal Obstacles
polygons = [
   Polygon([(220,616), (220,666), (251,670), (272,647)]),
   Polygon([(341,655), (359,667), (374,651), (366,577)]),
   Polygon([(311,530), (311,559), (339,578), (361,560), (361, 528),
(336, 516)]),
   Polygon([(105,628), (151,670), (180,629), (156,577), (113,
587)]),
   Polygon([(118,517), (245,517), (245,577), (118,557)]),
   Polygon([(280,583), (333,583), (333,665), (280,665)]),
   Polygon([(252,594), (290,562), (264,538)]),
   Polygon([(198,635), (217,574), (182,574)]),
]
# Define State Space for Problem
state_space = StateSpace(
   start = (120,650),
   end = (380,560),
   obstacles = polygons
)

print(state_space)

print("\n\nBREADTH FIRST SEARCH")
print(line)
result = BFS.search(deepcopy(state_space))
if result:
   print("Path Found")
   print(result[0])
# Solution Found
# (120,650), (105,628), (118,517), (336,516), (361,528), (380,560)
# Cost: 421.3344534244741

print("\n\nBEST FIRST GREEDY SEARCH")
print(line)
result = Best_Greedy.search(deepcopy(state_space))
if result:
   print("Path Found")
```

```
    print(result[0])
# Solution Found
# (120,650), (151,670), (251,670), (359,667), (374,651), (380,560)
# Cost: 358.0626920104638

print("\n\nA* SEARCH")
print(line)
result = AStar.search(deepcopy(state_space))
if result:
    print("Path Found")
    print(result[0])
# Solution Found
# (120,650), (151,670), (198,635), (220,616), (280,583), (339,578),
(380,560)
# Cost: 297.02594348473116

input("\n\nPress any key to start Empirical Analysis...\n\n")

N_INSTANCES = 10
# Empirical Analysis using N_INSTANCES problems. Modify this
constant as required
# Parameters analysed:
#   Number of nodes generated
#   Number of nodes expanded,
#   Actual time taken,
#   Completeness,
#   Optimality

print(line)
print("EMPIRICAL ANALYSIS")
print(line)

# Overall Times
bfs_time = 0
bestfs_time = 0
astar_time = 0
# Overall Solution Found count
bfs_found = 0
bestfs_found = 0
astar_found = 0
```

```python
# Overall Completions
bfs_completes = 0
bestfs_completes = 0
astar_completes = 0
# Overall Optimal Solution
bfs_optimal = 0
bestfs_optimal = 0
astar_optimal = 0
# Overall Generated Nodes
bfs_gen = 0
bestfs_gen = 0
astar_gen = 0
# Overall Visited Nodes
bfs_visited = 0
bestfs_visited = 0
astar_visited = 0
# Generate and Run instances
for i in range(N_INSTANCES):
   state_space = generate_state_space()
   # Run BFS
   bfs_exec_time, bfs_results = run_with_timer(
       function=BFS.search,
       args=(deepcopy(state_space),)
   )
   # Run Best-First
   bestfs_exec_time, bestfs_results = run_with_timer(
       function=Best_Greedy.search,
       args=(deepcopy(state_space),)
   )
   # Run A*
   astar_exec_time, astar_results = run_with_timer(
       function=AStar.search,
       args=(deepcopy(state_space),)
   )
   # Collect Summaries
   # Completions
   bfs_completes += 1
   bestfs_completes += 1
   astar_completes += 1
```

```
# Running time
bfs_time += bfs_exec_time
bestfs_time += bestfs_exec_time
astar_time += astar_exec_time
# Completion
if bfs_results:
    bfs_found += 1
    bfs_cost = bfs_results[0].cost
    bfs_gen += bfs_results[1]
    bfs_visited += bfs_results[2]
else:
    bfs_cost = inf
if bestfs_results:
    bestfs_found += 1
    bestfs_cost = bestfs_results[0].cost
    bestfs_gen += bestfs_results[1]
    bestfs_visited += bestfs_results[2]
else:
    bestfs_cost = inf
if astar_results:
    astar_found += 1
    astar_cost = astar_results[0].cost
    astar_gen += astar_results[1]
    astar_visited += astar_results[2]
else:
    astar_cost = inf
# Optimality
opt_cost = min([bfs_cost, bestfs_cost, astar_cost])
if opt_cost!=inf:
    if bfs_cost == opt_cost:
        bfs_optimal += 1
    if bestfs_cost == opt_cost:
        bestfs_optimal += 1
    if astar_cost == opt_cost:
        astar_optimal += 1
# Display Instance-Wise Summary
print("INSTANCE", i+1)
print(line)
print(state_space)
```

```
    # For BFS
    print("\nBREADTH FIRST SEARCH")
    print(line)
    display_summary(bfs_exec_time, bfs_results)
    # For Best First
    print("\nBEST FIRST GREEDY SEARCH")
    print(line)
    display_summary(bestfs_exec_time, bestfs_results)
    # For A*
    print("\nA* SEARCH")
    print(line)
    display_summary(astar_exec_time, astar_results)
    print("\n\n")

print(line)
print("OVERALL SUMMARY (for {num}
instances)".format(num=N_INSTANCES))
print(line)
print("\nBREADTH FIRST SEARCH")
print(line)
display_overall_summary(bfs_completes, bfs_found, bfs_optimal,
bfs_gen, bfs_visited, bfs_time, N_INSTANCES)
print("\nBEST FIRST GREEDY SEARCH")
print(line)
display_overall_summary(bestfs_completes, bestfs_found,
bestfs_optimal, bestfs_gen, bestfs_visited, bestfs_time,
N_INSTANCES)
print("\nA* SEARCH")
print(line)
display_overall_summary(astar_completes, astar_found, astar_optimal,
astar_gen, astar_visited, astar_time, N_INSTANCES)
```

(Output on next page)

Sample Output

1. Manual Test Case

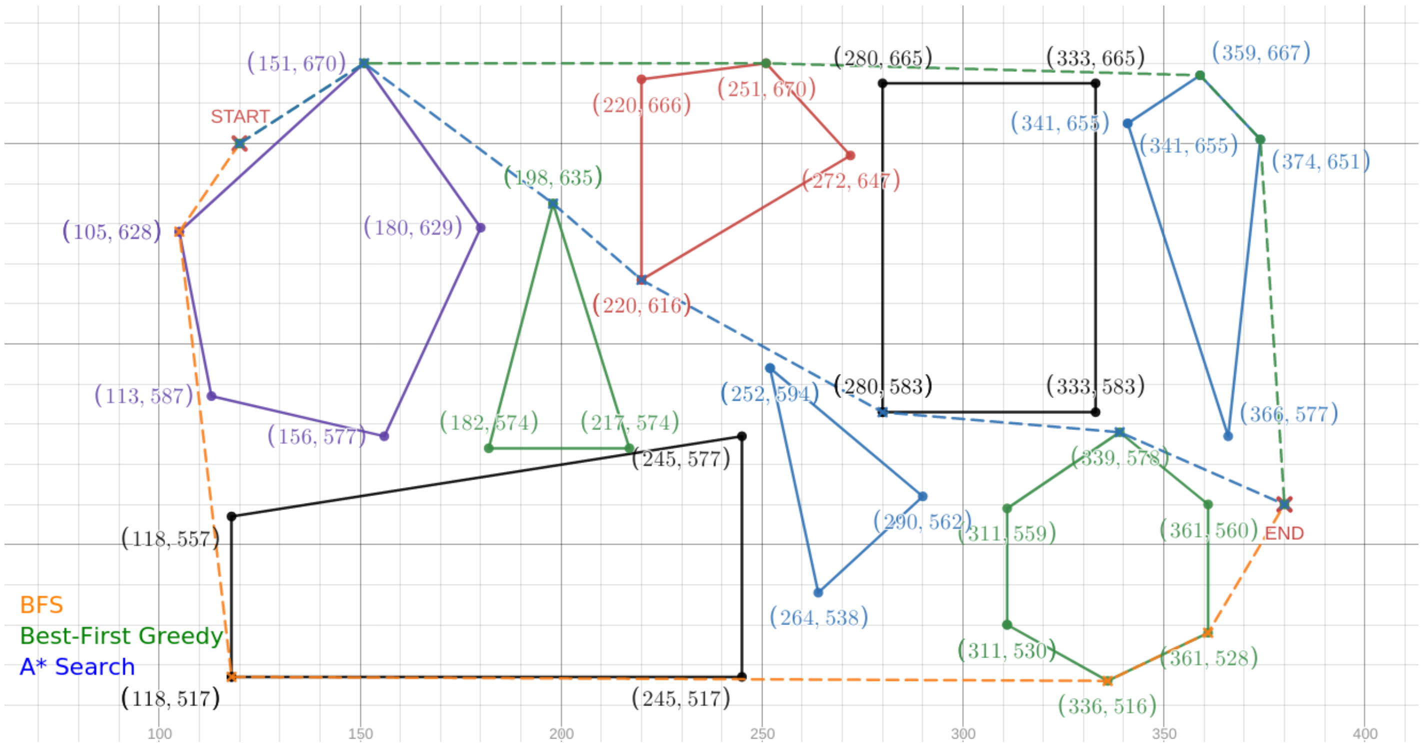
```
--------------------------------------------------
MANUAL TEST CASE
--------------------------------------------------
Start: (120,650)
Goal: (380,560)
Polygons:
4-sided Polygon : {(220,616), (220,666), (251,670), (272,647)}
4-sided Polygon : {(341,655), (359,667), (374,651), (366,577)}
6-sided Polygon : {(311,530), (311,559), (339,578), (361,560), (361,528), (336,516)}
5-sided Polygon : {(105,628), (151,670), (180,629), (156,577), (113,587)}
4-sided Polygon : {(118,517), (245,517), (245,577), (118,557)}
4-sided Polygon : {(280,583), (333,583), (333,665), (280,665)}
3-sided Polygon : {(252,594), (290,562), (264,538)}
3-sided Polygon : {(198,635), (217,574), (182,574)}


BREADTH FIRST SEARCH
--------------------------------------------------
Path Found
(120,650) -> (105,628) -> (118,517) -> (336,516) -> (361,528) -> (380,560)
Cost: 421.3344534244741


BEST FIRST GREEDY SEARCH
--------------------------------------------------
Path Found
(120,650) -> (151,670) -> (251,670) -> (359,667) -> (374,651) -> (380,560)
Cost: 358.0626920104638


A* SEARCH
--------------------------------------------------
Path Found
(120,650) -> (151,670) -> (198,635) -> (220,616) -> (280,583) -> (339,578) -> (380,560)
Cost: 297.02594348473116
```

2. Output Visualization of Manual Test Case



(contd.)

3. **Empirical Summary of Search Methods (over 10 random instances)**

```
--------------------------------------------------
OVERALL SUMMARY (for 10 instances)
--------------------------------------------------

BREADTH FIRST SEARCH
--------------------------------------------------
No. of Completions: 10
No. of cases where Path Found:  6
No. of Optimal Paths:  3
No. of Nodes Generated:  605
No. of Nodes Visited:  238
Avg. Time Taken: 0.29303114414215087s

BEST FIRST GREEDY SEARCH
--------------------------------------------------
No. of Completions: 10
No. of cases where Path Found:  6
No. of Optimal Paths:  5
No. of Nodes Generated:  135
No. of Nodes Visited:  43
Avg. Time Taken: 0.15320100784301757s

A* SEARCH
--------------------------------------------------
No. of Completions: 10
No. of cases where Path Found:  6
No. of Optimal Paths:  5
No. of Nodes Generated:  220
No. of Nodes Visited:  69
Avg. Time Taken: 0.16350274085998534s
```