**Homework Assignment #5**

**(Informed Search – A\* Search)**

**CS 156 Spring 2021**

This assignment builds on Homework #4 and hnThe objective of this homework assignment is to understand and visualize how an informed search algorithm like A\* works. Specifically, we will implement and compare the performance of A\* search algorithm for two different heuristics:

1. Null heuristic (same as UCS)
2. Simple heuristic - this will compute the manhattan distance between the position of the Spartan and the goal (medal)

We will explore these algorithms in a gaming environment where the Spartan Sammy will try to collect medals in a maze in the shortest possible time.

This assignment re-uses most of the files from Homework #4 namely:

1. data\_structures contains all the Class definitions for data structures to be used by the search algorithms
2. graphics contains Visualization of the maze and solution for the Spartan's quest
3. noway, questA, questB, questC, questD, questE, quewstF, questG, questH, questL and SJSU are txt files containing the description of different mazes with walls, starting position of Sammy and the starting positions of the medals that will be collected by Sammy
4. Sammy is the Spartan (mascot).
5. Spartanquest has the Main program that guides Sammy the Spartan on a quest within a given maze.

What you need to add:

You need to create a file called **informed\_search.py** with the content shown below and **modify your existing spartanquest.py** with one line:

import informed\_search

Then, invoke the program using: spartanquest.py maze\_file search\_algorithm where

The maze\_file is a text file such as SJSU.txt

The search\_algorithm in homework is astar

Example: spartanquest.py SJSU.txt astar

Here is the skeleton of the code for your assignment:

Regarding the heuristics, there are three:

1. single\_heuristic
2. better\_heuristic
3. gen\_heuristic

**You must implement single\_heuristic.**

**For the other two heuristics, enough guidance is provided for you to write the code. Each of these heuristics will be extra credit. 10 + 10 points.**

"""

A\* Algorithm and heuristics implementation

Your task for homework 5 is to implement:

1. astar

2. single\_heuristic

3. better\_heuristic

4. gen\_heuristic

"""

from math import sqrt, floor

import data\_structures

def astar(problem, heuristic):

"""

A\* graph search algorithm

returns a solution for the given search problem.

:param

problem (a Problem object) representing the quest

see Problem class definition in spartanquest.py

heuristic (a function) the heuristic function to be used.

:return: list of actions representing the solution to the quest

or None if there is no solution.

"""

# Enter your code here and remove the pass statement below

closed = set()

fringe = data\_structures.PriorityQueue()

state = problem.start\_state()

root = data\_structures.Node(state)

fringe.push(root, heuristic(state, problem))

while True:

if fringe.is\_empty():

return None

node = fringe.pop()

if problem.is\_goal(node.state):

return node.solution()

if node.state not in closed:

closed.add(node.state)

for child\_state, action, action\_cost in problem.successors(node.state):

# Tie Breaking

h = heuristic(child\_state, problem)

# goal = problem.medals.copy().pop()

# start = problem.start\_state()[0]

# dx1 = state[0][0] - goal[0]

# dy1 = state[0][1] - goal[1]

# dx2 = start[0] - goal[0]

# dy2 = start[1] - goal[1]

# cross = abs(dx1\*dy2 - dx2\*dy1)

# h += cross\*0.001

h \*= (1.0 + (1/100))

# if child\_state not in closed:

child\_node = data\_structures.Node(child\_state, node, action, action\_cost + node.cumulative\_cost)

f = child\_node.cumulative\_cost + h

fringe.push(child\_node, f)

def null\_heuristic(state, problem):

"""

Trivial heuristic to be used with A\*.

Running A\* with this null heuristic, gives us uniform cost search.

:param

state: A state is represented by a tuple containing:

the current position (row, column) of Sammy the Spartan

a tuple containing the positions of the remaining medals.

problem: (a Problem object) representing the quest.

:return: 0

"""

return 0

def single\_heuristic(state, problem):

"""

A simple heuristic for a single medal that finds the manhattan distance.

This heuristic is admissible because the manhattan distance does not account for the cost of the increased cost of the East and West direction thus being optimistic.

:param

state: A state is represented by a tuple containing:

the current position (row, column) of Sammy the Spartan

a tuple containing the positions of the remaining medals.

problem: (a Problem object) representing the quest.

:return:

"""

# Enter your code here and remove the pass statement below

def better\_heuristic(state, problem):

"""

An improved heuristic over the single heuristic that takes into account the cost of each movement. It is admissible because the heuristic calculates at most the true cost to the goal.

:param

state: A state is represented by a tuple containing:

the current position (row, column) of Sammy the Spartan

a tuple containing the positions of the remaining medals.

problem: (a Problem object) representing the quest.

:return:

"""

# Enter your code here and remove the pass statement below

if state[1]:

start = problem.start\_state()[0]

goal = state[1][0]

position = state[0]

D = 1

D2 = 1

# Check where the state is in relation to the start state

if position[1] > goal[1]: # Moved East

D = 1

if position[1] < goal[1]: # Moved West

D = 6

if position[0] > goal[0]: # Moved South

D2 = 2

if position[0] < goal[0]: # Moved North

D2 = 1

#

# if position[0] == goal[1] and (position[0] < 0 or position[0] > 14):

# return manhattan\_distance(position, goal) + 2

# if position[0] == goal[0] and (position[1] < 0 or position[1] > 37):

# return manhattan\_distance(position, goal) + 2

# row = start[0] - position[0]

# column = start[1] - position[1]

# # return D \* (row + column) + (D2- 2 \* D) \* min(row, column)

# if row <= 0 :

# D = 2

# if column <= 0 :

# D2 = 6

# if row > 0:

# D = 1

# if column > 0:

# D2 = 1

# return 2\*7 \* manhattan\_distance(position, goal)

# return (D2 \* abs(position[0] - goal[0])) \* (D + abs(position[1] - goal[1]))

def gen\_heuristic(state, problem):

"""

A heuristic for general problems that uses the manhattan distance. This heuristic works on multiple medals. This heuristic is admissible because the sum to the goal divided by goals is optimistic.

:param

state: A state is represented by a tuple containing:

the current position (row, column) of Sammy the Spartan

a tuple containing the positions of the remaining medals.

problem: (a Problem object) representing the quest.

:return:

"""

if state[1]:

goals = state[1]

start = problem.start\_state()[0]

heru = []

sum = 0

position = state[0]

# for goal in goals:

# D = 1

# D2 = 1

# # Check where the state is in relation to the start state

# if position[1] > goal[1]: # Moved East

# D = 1

# if position[1] < goal[1]: # Moved West

# D = 6

# if position[0] > goal[0]: # Moved South

# D2 = 2

# if position[0] < goal[0]: # Moved North

# D2 = 1

# heru.append((D2 \* abs(position[0] - goal[0])) + (D \* abs(position[1] - goal[1])))

for goal in goals:

D = 1

D2 = 1

# Check where the state is in relation to the start state

if position[1] > goal[1]: # Moved East

D = 1

if position[1] < goal[1]: # Moved West

D = 6

if position[0] > goal[0]: # Moved South

D2 = 2

if position[0] < goal[0]: # Moved North

D2 = 1

# heru.append(((D2 \* abs(position[0] - goal[0])) + (D \* abs(position[1] - goal[1])), goal))

# heru.append(max(((D2 \* abs(position[0] - goal[0])), (D \* abs(position[1] - goal[1]))), goal))

# sum += min((D2 \* abs(position[0] - goal[0])), (D \* abs(position[1] - goal[1])))

# sum += (D2 \* abs(position[0] - goal[0])) + (D \* abs(position[1] - goal[1]))

# heru.sort()

def manhattan\_distance(point1, point2):

"""

Compute the Manhattan distance between two points.

:param point1 (tuple) representing the coordinates of a point in a plane

:param point2 (tuple) representing the coordinates of a point in a plane

:return: (integer) The Manhattan distance between the two points

"""

# Enter your code here and remove the pass statement below

return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

def min\_distance(point1, other\_points):

"""

Find the minimum Manhattan distance from point1 to the other points

:param point1 (tuple) representing the coordinates of a point in a plane

:param other\_points (set of tuples) representing several points in a plane

:return: (integer) maximum Manhattan distance from point1 to the other

points

"""

if other\_points:

return min([manhattan\_distance(point1, point) for point in other\_points])

**Summarize and tabulate your observations in terms of:**

1. Path Length
2. Past Cost
3. Number of nodes expanded and
4. Processing time

for each of the following quests:

questA, questB, questC, questD, questE, questF, questG, questH, questL and noway

using the following heuristics:

1. Null heuristic
2. Simple heuristic
3. Better heuristic (optional)
4. Gen heuristic (optional)

**Bonus points for heuristics (C) and (D).**

**Are there cases where some of the heuristics (B), (C) or (D) performed better than (A) and why do you think it happened?**