Project 1 -- Python Warm-up

Instructions

Update your report every week. Let me know you where you succeeded and where you could use more help. Turn in your progress report as a PDF file and your lab0.py file in the assignments folder for the current week. So, during week 1 of the class, go to Worldclass > MSDS688 > Assignments > Week 1 -- Project 1. Please post your questions to the Project 1 discussion thread.

Remember, you get credit by either:

1. Making progress in terms of number of tests passing, *or*

2. Describing the roadblock that has stopped you, the sequence of steps you took to solve it, and the reasons you took them. Think of applying the scientific method to programming.

Getting Help

For help with Markdown, check out the concise and helpful Mastering Markdown that lets you modify demo code and see the resulting changes live.

For help generating a PDF file from your progress report markdown file. First, add the Markdown PDF plugin to VSCode. Then follow these directions.

See the Project 1 video(s) in the discussions folder for help in completing these steps. Please post a question if you get stuck.

Please post your questions to the discussion forum for this project.

What progress did you make?

In week 8, I have the lab2 done and complete passing 64 tests out of 66. the two test are for is_admissable and is_consistent. test 58 is noted as tricky and asking how we are checking the admissibilty of the node.

What code are you most proud of, and why?

```

```

Did you get stuck?

I am stuck in is_admissiable and is_consistent they are working but 2 test I can not pass.

What steps did you take to solve the problem?

I have tried modifying the functions, look it up in github.

Did you succeed? If so, what worked and why?

No, whenever I change the code more tests are being faild.

Code

```
 # MIT 6.034 Lab 2: Search
# Written by 6.034 staff

from subprocess import list2cmdline
from sklearn import neighbors
from search import Edge, UndirectedGraph, do_nothing_fn, make_generic_search
import read_graphs
from functools import reduce

all_graphs = read_graphs.get_graphs()
GRAPH_0 = all_graphs['GRAPH_0']
GRAPH_1 = all_graphs['GRAPH_1']
GRAPH_2 = all_graphs['GRAPH_2']
GRAPH_3 = all_graphs['GRAPH_3']
GRAPH_FOR_HEURISTICS = all_graphs['GRAPH_FOR_HEURISTICS']


# Please see wiki lab page for full description of functions and API.

#### PART 1: Helper Functions ##################################################

def path_length(graph, path):
    """Returns the total length (sum of edge weights) of a path defined by a
    list of nodes coercing an edge-linked traversal through a graph.
    (That is, the list of nodes defines a path through the graph.)
    A path with fewer than 2 nodes should have length of 0.
    You can assume that all edges along the path have a valid numeric weight."""
    if len(path) < 2:
        return 0
    x = 0
    for i in range(1,len(path)):
        current = path[i]
        previous = path[i-1]
        edge = graph.get_edge(previous,current)
        x += edge.length
    return x

def has_loops(path):
    """Returns True if this path has a loop in it, i.e. if it
    visits a node more than once. Returns False otherwise."""

    visitedNodes = {}
    for i in path:
        visitedNodes[i] = 0
    if len(visitedNodes) != len(path):
        return True
    return False


def extensions(graph, path):
    """Returns a list of paths. Each path in the list should be a one-node
    extension of the input path, where an extension is defined as a path formed
    by adding a neighbor node (of the final node in the path) to the path.
    Returned paths should not have loops, i.e. should not visit the same node
    twice. The returned paths should be sorted in lexicographic order."""
    # N for Neighbors

    N = graph.get_neighbors(path[-1])
    listOfPath = []
    for i in N:
        nodePath = path[:]
        if i not in nodePath:
            nodePath.append(i)
            listOfPath.append(nodePath)

    return sorted(listOfPath)

def sort_by_heuristic(graph, goalNode, nodes):
    """Given a list of nodes, sorts them best-to-worst based on the heuristic
    from each node to the goal node. Here, and in general for this lab, we
    consider a smaller heuristic value to be "better" because it represents a
    shorter potential path to the goal. Break ties lexicographically by
    node name."""
    nodesList = []
    for i in nodes:
        nodesList.append((i,graph.get_heuristic_value(i,goalNode)))
    x = sorted(nodesList, key=lambda E: E[0])
    y = sorted(x , key=lambda E: E[1])
    sortedList = []
    for i in y:
        sortedList.append(i[0])
    return sortedList



# You can ignore the following line.  It allows generic_search (PART 3) to
# access the extensions and has_loops functions that you just defined in PART 1.
generic_search = make_generic_search(extensions, has_loops)  # DO NOT CHANGE


#### PART 2: Basic Search ######################################################

def basic_dfs(graph, startNode, goalNode):
    """
    Performs a depth-first search on a graph from a specified start
    node to a specified goal node, returning a path-to-goal if it
    exists, otherwise returning None.
    Uses backtracking, but does not use an extended set.
    """
    '''
    a = extensions(graph, startNode) # starting from the root
    while len(a) > 0:
        x = a.pop(0)
        if x[-1] == goalNode:
            return x
        y = extensions(graph, x) + a
        a = y
    return None
    '''
    return generic_search(*generic_dfs)(graph, startNode, goalNode)


def basic_bfs(graph, startNode, goalNode):
    """
    Performs a breadth-first search on a graph from a specified start
    node to a specified goal node, returning a path-to-goal if it
    exists, otherwise returning None.
    """
    ## with Julio's help

    #using extension function instead of creating a list with startNode
    '''
    a = extensions(graph,startNode)
    while len(a) > 0: # while not at goal
        x = a.pop(0) # removing the first Node of the list
        if x[-1] == goalNode: # if the last node of a is = to goal
            return x # goal acheived
        x += extensions(graph, x) # else: continue
    return None
    '''
    return generic_search(*generic_bfs)(graph, startNode, goalNode)



def a_star(graph, startNode, goalNode):
    ex = set()
    list1 = [[startNode]]
    while list1[0][-1] != goalNode: ## first and last are != goal
        node = list1[0][-1]
        path = list1[0]
        if node not in ex:
            ex.add(node)
            for i in graph.get_neighbors(node):
                if i not in path:
                    list1.append(path+[i])
        list1 = list1[1:] # slicing the list
        list1.sort(key = lambda x: path_length(graph, x) +
graph.get_heuristic_value(x[-1],goalNode))
    return list1[0]



#### PART 3: Generic Search #####################################################
```

```
# Generic search requires four arguments (see wiki for more details):
# sort_new_paths_fn: a function that sorts new paths that are added to the agenda
# add_paths_to_front_of_agenda: True if new paths should be added to the front of
the agenda
# sort_agenda_fn: function to sort the agenda after adding all new paths
# use_extended_set: True if the algorithm should utilize an extended set


# Define your custom path-sorting functions here.
# Each path-sorting function should be in this form:
# def my_sorting_fn(graph, goalNode, paths):
#     # YOUR CODE HERE
#     return sorted_paths

def hillsort(graph,goalNode,paths):
    # best first sort is the same as hill climbing sort
    paths = sorted(paths)
    paths = sorted(paths, key = lambda p: graph.get_heuristic_value(p[len(p)-1],
goalNode))
    return paths

def branch_sort(graph,goalNode,paths):
    a = sorted(paths, key = lambda p: path_length(graph, p))
    return a

def alphabetical_sort(graph, goalNode, paths):
    # python built-in sorted function sort alphabetically
    p = sorted(paths)
    return p

def heuristic_sort(graph, goalNode, paths):
    a = sorted(paths, key=lambda p:
path_length(graph,p)+graph.get_heuristic_value(p[-1],goalNode))
    return a



# do Not sort new paths, add path to the front, do not sort agenda, do not use
extended set
generic_dfs = [do_nothing_fn, True, do_nothing_fn, False]
# do not sort new paths, do not add path to the front, do not sort agenda, do not
use extended set
generic_bfs = [do_nothing_fn, False, do_nothing_fn, False]
# use hillsort to sort new paths, add path to the front, do not sort agenda, do
not use extended set
generic_hill_climbing = [hillsort, True, do_nothing_fn, False]
# do not sort new paths, add to the front, use best sort to sort agenda , do not
use extended set
generic_best_first = [do_nothing_fn, True,hillsort , False]
# sort new paths alphabetically, do not add to the front, use branch and bound to
sort agenda, do not use extended set
generic_branch_and_bound = [alphabetical_sort, False, branch_sort, False]
# hillsort for new paths, add the end, heuristic sort for agenda, no extended set
generic_branch_and_bound_with_heuristic = [hillsort, False,heuristic_sort , False]
```

```
# branch sort for new paths, add to the end, branch and bound sort for agenda, use
extended set
generic_branch_and_bound_with_extended_set = [branch_sort, False, branch_sort,
True]
# hill sort for new paths, add to the end, heuristic sort for agenda, with
extended set
generic_a_star = [hillsort, False, heuristic_sort, True]


# Here is an example of how to call generic_search (uncomment to run):
# my_dfs_fn = generic_search(*generic_dfs)
# my_dfs_path = my_dfs_fn(GRAPH_2, 'S', 'G')
# print(my_dfs_path)

# Or, combining the first two steps:
# my_dfs_path = generic_search(*generic_dfs)(GRAPH_2, 'S', 'G')
# print(my_dfs_path)


### OPTIONAL: Generic Beam Search

# If you want to run local tests for generic_beam, change TEST_GENERIC_BEAM to
True:
TEST_GENERIC_BEAM = False

# The sort_agenda_fn for beam search takes fourth argument, beam_width:
# def my_beam_sorting_fn(graph, goalNode, paths, beam_width):
#     # YOUR CODE HERE
#     return sorted_beam_agenda

generic_beam = [None, None, None, None]


# Uncomment this to test your generic_beam search:
# print(generic_search(*generic_beam)(GRAPH_2, 'S', 'G', beam_width=2))


#### PART 4: Heuristics ########################################################

def is_admissible(graph, goalNode):
    """Returns True if this graph's heuristic is admissible; else False.
    A heuristic is admissible if it is either always exactly correct or overly
    optimistic; it never over-estimates the cost to the goal."""
    for node in graph.nodes:
        if path_length(graph, a_star(graph,node,goalNode)) <
graph.get_heuristic_value(node,goalNode):
            return False
    return True

def is_consistent(graph, goalNode):
    """Returns True if this graph's heuristic is consistent; else False.
    A consistent heuristic satisfies the following property for all
    nodes v in the graph:
        Suppose v is a node in the graph, and N is a neighbor of v,
        then, heuristic(v) <= heuristic(N) + edge_weight(v, N)
    In other words, moving from one node to a neighboring node never unfairly
    decreases the heuristic.
    This is equivalent to the heuristic satisfying the triangle inequality."""
    for edge in graph.edges:
        x =  abs(graph.get_heuristic_value(edge.startNode, goalNode) -
graph.get_heuristic_value(edge.endNode, goalNode))
        if edge.length < x:
            return False
        return True



### OPTIONAL: Picking Heuristics

# If you want to run local tests on your heuristics, change TEST_HEURISTICS to
True.
#  Note that you MUST have completed generic a_star in order to do this:
TEST_HEURISTICS = False


# heuristic_1: admissible and consistent

[h1_S, h1_A, h1_B, h1_C, h1_G] = [None, None, None, None, None]

heuristic_1 = {'G': {}}
heuristic_1['G']['S'] = h1_S
heuristic_1['G']['A'] = h1_A
heuristic_1['G']['B'] = h1_B
heuristic_1['G']['C'] = h1_C
heuristic_1['G']['G'] = h1_G


# heuristic_2: admissible but NOT consistent

[h2_S, h2_A, h2_B, h2_C, h2_G] = [None, None, None, None, None]

heuristic_2 = {'G': {}}
heuristic_2['G']['S'] = h2_S
heuristic_2['G']['A'] = h2_A
heuristic_2['G']['B'] = h2_B
heuristic_2['G']['C'] = h2_C
heuristic_2['G']['G'] = h2_G


# heuristic_3: admissible but A* returns non-optimal path to G

[h3_S, h3_A, h3_B, h3_C, h3_G] = [None, None, None, None, None]

heuristic_3 = {'G': {}}
heuristic_3['G']['S'] = h3_S
heuristic_3['G']['A'] = h3_A
heuristic_3['G']['B'] = h3_B
heuristic_3['G']['C'] = h3_C
```

```
heuristic_3['G']['G'] = h3_G


# heuristic_4: admissible but not consistent, yet A* finds optimal path

[h4_S, h4_A, h4_B, h4_C, h4_G] = [None, None, None, None, None]

heuristic_4 = {'G': {}}
heuristic_4['G']['S'] = h4_S
heuristic_4['G']['A'] = h4_A
heuristic_4['G']['B'] = h4_B
heuristic_4['G']['C'] = h4_C
heuristic_4['G']['G'] = h4_G


##### PART 5: Multiple Choice ##################################################

# We need to explore all nodes (bedrooms). Even though British Museum is not
efficint, it visits all nodes where we need to discover.
ANSWER_1 = '2'
# Branch and bound
ANSWER_2 = '4'
#BFS
ANSWER_3 = '1'
# A-star
ANSWER_4 = '3'


#### SURVEY ####################################################################

NAME = "Ahmad Alqurashi"
COLLABORATORS = "Caroline and Julio + https://github.com/MTSavran/MIT-Artificial-
Intelligence--6.034-/blob/master/lab2/lab2.py#L96 "
HOW_MANY_HOURS_THIS_LAB_TOOK = "Weeks"
WHAT_I_FOUND_INTERESTING = "That DFS and BFS are similar , Search algorarithms
brought back memories from my undergrad where we had a class mainly talks about
searching and sorting graphs"
WHAT_I_FOUND_BORING = "It is not boaring rather than complicated"
SUGGESTIONS = None



###########################################################
### Ignore everything below this line; for testing only ###
###########################################################

# The following lines are used in the online tester. DO NOT CHANGE!

generic_dfs_sort_new_paths_fn = generic_dfs[0]
generic_bfs_sort_new_paths_fn = generic_bfs[0]
generic_hill_climbing_sort_new_paths_fn = generic_hill_climbing[0]
generic_best_first_sort_new_paths_fn = generic_best_first[0]
generic_branch_and_bound_sort_new_paths_fn = generic_branch_and_bound[0]
generic_branch_and_bound_with_heuristic_sort_new_paths_fn =
generic_branch_and_bound_with_heuristic[0]
generic_branch_and_bound_with_extended_set_sort_new_paths_fn =
generic_branch_and_bound_with_extended_set[0]
generic_a_star_sort_new_paths_fn = generic_a_star[0]

generic_dfs_sort_agenda_fn = generic_dfs[2]
generic_bfs_sort_agenda_fn = generic_bfs[2]
generic_hill_climbing_sort_agenda_fn = generic_hill_climbing[2]
generic_best_first_sort_agenda_fn = generic_best_first[2]
generic_branch_and_bound_sort_agenda_fn = generic_branch_and_bound[2]
generic_branch_and_bound_with_heuristic_sort_agenda_fn =
generic_branch_and_bound_with_heuristic[2]
generic_branch_and_bound_with_extended_set_sort_agenda_fn =
generic_branch_and_bound_with_extended_set[2]
generic_a_star_sort_agenda_fn = generic_a_star[2]

# Creates the beam search using generic beam args, for optional beam tests
beam = generic_search(*generic_beam) if TEST_GENERIC_BEAM else None

# Creates the A* algorithm for use in testing the optional heuristics
if TEST_HEURISTICS:
    a_star = generic_search(*generic_a_star)
```