CS532 Homework 8 Critique

Archana Machireddy

Question 1

Adjacency matrix will have node2 entries. Therefore here it will use 10242 bits or 10242/8 bytes. (131072 bytes)

For adjacency list we will have a pointer for each node, so 1024 \* 8 bytes are used for pointers. Next we have a pointer to each of the edges. If it is a directed graph 2048 \* 8 bytes. If it is an undirected graph each edge will be stored twice in the list so this will require 2048 \* 2 \*8 bytes. Each dge weight requires a bit to store, therefore it uses 2048/8 bytes. For a directed graph total space will be

(1024 \* 2) + (2048 \* 8) + 2048/8 bytes. (18688 bytes)

For this graph adjacency list required less space as compared to adjacency matrix.

Question 2

Part 1

with open(fpath, mode='r') as csv\_file:

csv\_reader = csv.DictReader(csv\_file)

for row in csv\_reader:

system\_mapping[row['system\_id']] = row['stargates']

name\_to\_id[row['solarsystem\_name']] = row['system\_id']

for index, name in enumerate(name\_to\_id):

id\_to\_index[name\_to\_id[name]] = index

# empty list of the right size to put your adjacency lists into

graph: List[List[int]] = [None] \* len(system\_mapping)

for system, adjacents in system\_mapping.items():

# here is where you populate the adjacency lists representation

#raise NotImplementedError

adj\_list = []

adjacents = adjacents.strip('[')

adjacents = adjacents.strip(']')

adjacents = adjacents.replace(' ', '')

adjacents = adjacents.split(',')

for stars in adjacents:

if stars != '':

adj\_list.append(id\_to\_index[str(stars)])

graph[id\_to\_index[system]] = adj\_list

Part 2

Jita -> Niyabainen -> Tunttaras -> Nourvukaiken -> Tama -> Sujarento -> Onatoh -> Tannolen -> Tierijev -> Chantrousse -> Ourapheh -> Botane -> Dodixie

Part 2 Critique

Instead of searching the entire graph, I should have stopped when I hit the destination node. Instead I backtrack after searching the entire graph.

def breadth\_first\_search(

graph: List[List[int]], source: int, destination: int, use\_python\_deque=False

) -> List[int]:

# initialization of the nodes

vertices = [Vertex(index) for index, \_ in enumerate(graph)]

vertices[source].color = "gray"

vertices[source].d = 0

if use\_python\_deque:

queue = deque()

else:

queue = Queue()

# Here is where you implement the breadth first search

queue.append(source)

while bool(queue) == True:

u = queue.popleft()

for adj\_star in graph[u]:

if vertices[adj\_star].color == 'white':

vertices[adj\_star].color = 'gray'

vertices[adj\_star].d = vertices[u].d + 1

vertices[adj\_star].pi = vertices[u]

if adj\_star == destination:

return backtrace(vertices[destination])

queue.append(adj\_star)

vertices[u].color = 'black'

Part 3

|  |  |  |
| --- | --- | --- |
|  | Time using custom Queue object (in seconds) | Time using native Queue object (in seconds) |
| Jita 🡪 Dodixie | 0.02444309100974351 | 0.01936052495148033 |
| 313I-B 🡪 ZDYA-G | 0.02269595500547439 | 0.020144721027463675 |

Though the second route is much longer than the first, the adjacency list of each of the solar systems in the path is small. This leads to similar processing times for both routes. The native queue was always faster than the custom implementation of queue.

Part 4

Depth first search keeps searching deeper in the graph and returns to a level higher only when it runs out of nodes to search deeper. Because of this property it might take a convoluted path to reach a node that is actually close by. Depth first search does not guarantee that a closer node is visited before a farther one, so it is problematic for navigation routing purposes.

Question 3

Part 1

graph = {}

with open(fpath) as f:

for line in f.readlines():

line = line.strip()

if line[0] != '-':

first = line

graph[first] = []

else:

graph[first].append(line.strip('- '))

return graph

Part 2

def depth\_first\_search(graph, use\_python\_deque=False):

# initialization of the nodes

vertices1 = []

for key, value in graph.items():

vertices1.append(key)

vertices1 = vertices1 + value

s\_ver = set(vertices1)

name\_to\_id = {}

for index, name in enumerate(s\_ver):

name\_to\_id[name] = index

vertices = [Vertex(index) for index in s\_ver]

if use\_python\_deque:

queue = deque()

else:

queue = Queue()

time = 0

for index, vertex in enumerate(vertices):

if vertices[index].color == 'white':

queue.appendleft(vertex.identifier)

while bool(queue) == True:

u = queue.popleft()

time = time + 1

if vertices[name\_to\_id[u]].color == 'white':

vertices[name\_to\_id[u]].d = time

vertices[name\_to\_id[u]].color = 'gray'

if vertices[name\_to\_id[u]].identifier in list(graph.keys()):

for adj\_vert in graph[u]:

if vertices[name\_to\_id[adj\_vert]].color == 'white':

vertices[name\_to\_id[adj\_vert]].color = 'gray'

time = time + 1

vertices[name\_to\_id[adj\_vert]].d = time

vertices[name\_to\_id[adj\_vert]].pi = vertices[name\_to\_id[u]]

if vertices[name\_to\_id[adj\_vert]].identifier in list(graph.keys()):

processed = []

for vert in graph[adj\_vert]:

processed.append(vertices[name\_to\_id[vert]].color)

if 'white' in processed:

queue.appendleft(adj\_vert)

else:

vertices[name\_to\_id[adj\_vert]].color = 'black'

time = time + 1

vertices[name\_to\_id[adj\_vert]].f = time

else:

vertices[name\_to\_id[adj\_vert]].color = 'black'

time = time + 1

vertices[name\_to\_id[adj\_vert]].f = time

vertices[name\_to\_id[u]].color = 'black'

time = time + 1

vertices[name\_to\_id[u]].f = time

Following an iterative fashion similar to breath-first search, if a node has multiple adjacent nodes appending them into the beginning of the queue and then popping them will not give a correct order of finish timings.

Part 3

class dfs():

def \_\_init\_\_(self,graph):

self.graph = graph

self.time = 0

vertices1 = []

for key, value in self.graph.items():

vertices1.append(key)

vertices1 = vertices1 + value

s\_ver = set(vertices1)

self.name\_to\_id = {}

for index, name in enumerate(s\_ver):

self.name\_to\_id[name] = index

self.vertices = [Vertex(index) for index in s\_ver]

self.final\_list = []

def depth\_first\_search(self):

for index, vertex in enumerate(self.vertices):

if self.vertices[index].color == "white":

self.dfs\_visit(vertex)

return self.final\_list

def dfs\_visit(self,vertex):

self.time = self.time + 1

self.vertices[self.name\_to\_id[vertex.identifier]].d = self.time

self.vertices[self.name\_to\_id[vertex.identifier]].color = "gray"

if vertex.identifier in list(self.graph.keys()):

for adj\_vert in self.graph[vertex.identifier]:

if self.vertices[self.name\_to\_id[adj\_vert]].color == "white":

self.vertices[self.name\_to\_id[adj\_vert]].pi = self.vertices[self.name\_to\_id[vertex.identifier]]

self.dfs\_visit(self.vertices[self.name\_to\_id[adj\_vert]])

self.vertices[self.name\_to\_id[vertex.identifier]].color = "black"

self.final\_list = [vertex.identifier] + self.final\_list

self.time = self.time + 1

self.vertices[self.name\_to\_id[vertex.identifier]].f = self.time

def topological\_sort(graph: Dict[str, List[str]]) -> List[str]:

"""Performs topological sort on the adjacency list generated earlier

Arguments:

graph {Dict[str, List[str]]} -- dictionary containing adjacency lists created by parse\_requirements function

Returns:

List[str] -- Sorted dependencies

"""

#raise NotImplementedError

dfs\_graph = dfs(graph)

sorted\_depandencies = dfs\_graph.depth\_first\_search()

return sorted\_depandencies

Result of the topological sort:

['TimeView', 'flake8-bugbear', 'pytest-qt', 'qtawesome', 'numba', 'black', 'click', 'scipy', 'pyqtgraph', 'qtpy', 'pyqt5', 'flake8-mypy', 'flake8', 'pyflakes', 'pycodestyle', 'sqlalchemy', 'PyQt5-sip', 'appdirs', 'mccabe', 'pytest', 'py', 'more-itertools', 'atomicwrites', 'pre-commit', 'virtualenv', 'toml', 'nodeenv', 'identify', 'cfgv', 'six', 'mypy', 'typed-ast', 'pyedflib', 'numpy', 'setuptools', 'pluggy', 'llvmlite', 'cython', 'cached-property', 'attrs', 'aspy.yaml', 'pyyaml']

Question 3 Critique

I should have given the start node of the sort as ‘TimeView’.

def depth\_first\_search(self):

# for index, vertex in enumerate(self.vertices):

# if self.vertices[index].color == "white":

# self.dfs\_visit(vertex)

self.dfs\_visit(self.vertices[self.name\_to\_id["TimeView"]])

['TimeView', 'flake8-bugbear', 'flake8-mypy', 'mypy', 'typed-ast', 'flake8', 'pyflakes', 'pycodestyle', 'mccabe', 'pre-commit', 'virtualenv', 'nodeenv', 'identify', 'cfgv', 'cached-property', 'aspy.yaml', 'pyyaml', 'pytest-qt', 'pytest', 'setuptools', 'py', 'pluggy', 'more-itertools', 'atomicwrites', 'black', 'toml', 'click', 'attrs', 'appdirs', 'sqlalchemy', 'scipy', 'qtawesome', 'six', 'qtpy', 'pyqtgraph', 'pyqt5', 'PyQt5-sip', 'pyedflib', 'numba', 'numpy', 'llvmlite', 'cython']

According to the book, as each vertex is finished, it is added to the sorted list. So this way TimeView would be added last. So the order of installation will be the reverse of this list.