CS532 Homework 8
Archana Machireddy

Question 1

Adjacency matrix will have node^2 entries. Therefore here it will use 1024^2 bits or $1024^2/8$ 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

```python
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 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_dependencies = 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']
```