Section B4 Searching Graphs

Depth First Search Algorithm

depth first search. It is characterised by:

- a list of visited vertices necessary to avoid cyclic traversal, and
- the ability to backtrack from dead ends.

A natural implementation of a depth first search uses a stack of visited vertices to allow backtracking by pushing the visited path onto a stack and then backtracking by popping dead ends off the stack to search for a more fruitful route.

1. 从起始点开始。
2. 将当前节点标记为已访问。
3. 访问当前节点的所有未访问邻居。
4. 对于每一个邻居，递归地执行DFS。
5. 当当前节点没有未访问的邻居时，回溯。

```
def iterative_dfs (self, start, goal):
        '''
        Returns true when an iterative depth first search finds a path from the
        Vertex 'start' to the Vertex 'goal'.
        '''

        # We use a list as a stack with methods append(), pop() and len(stack)
        stack = list()
        # A set of visited vertices.
        visited = set()

        # The first operation of the iterative approach is to push the start vertex
        # onto the stack.
        stack.append(start)

        while len(stack) > 0:

            current = stack.pop()
            visited.add (current)

            if current == goal:
                return True

            for v in current.get_connections():
                if not v in visited:
                    stack.append(v)

        return False"
```

we can show that this iterative implementation has time complexity O(n) - since, in the worst case, we will visit each vertex just once. Similarly, space complexity follows the worst case where the stack holds (almost) all the vertices in the graph - thus also O(n).

Breadth First Search with Queues

BFS uses a queue to track child vertices that were encountered but not yet explored. Precautions must of course be taken to avoid visiting the same vertex twice. BFS does this by maintaining separate state for visited and unvisited vertices.
Breadth first search may be used to determine the levels of each and every vertex, to find the shortest path between two vertices or to find the presence of a vertex in the graph. Breadth first search is an exhaustive search, meaning that it looks at all paths at the same time, but will also find the shortest path, with the least number of edges, between any two vertices in a graph. Performing breadth first search on large graphs may take too long to be of practical use.

1. 从起始点开始。
2. 将当前节点标记为已访问，并将其添加到一个队列中。
3. 当队列不为空时，取出队列的第一个元素。
4. 访问当前节点的所有未访问邻居，将这些邻居标记为已访问并添加到队列中。
5. 返回第3步。

The BFS algorithm can most simply be described by the following steps:

```
# create a queue
#
# from a starting vertex, mark as visited and place it into the queue
#
# while the queue is not empty and target not found
#
#    pop the head of the queue
#
#    if head of queue == target, return
#
#    else
#       mark as visited and queue all unvisited neighbors of the vertex
```

This iterative approach will perform a breadth first search but does not record the path to the target.

This sample implementation uses the Python collections.deque for queuing. Vertices are appended to the 'right' tail of the queue and popped from the 'left' head of the queue.

```
def iterative_bfs (self, start, goal):
        '''
        Returns true when the goal is found in the graph from a breadth first
        search commencing at the vertex 'start'
        '''
        visited = set()
        queue = deque()
        found = False

        # Place the start vertex on the queue.
        visited.add (start)
        queue.append (start)

        while queue and not found:

            vertex = queue.popleft()

            # Pop the head of the queue and return True if it is the goal.
            if vertex == goal:
                found = True

            else:
                # Queue and mark as visited all non visited neighbours
                for neighbour in vertex.get_connections():
                    if neighbour not in visited:
                        visited.add (neighbour)
                        queue.append(neighbour)

        return found
```

Complexity

For a graph G = (V, E), time complexity can be expressed as O(|V| + |E|) where |V| is the number of vertices and |E| is the number of edges - since every vertex and edge must be explored. In a heavily interconnected graph, the set of edges E, may approach $|V|^2$ in number making this exhaustive search impractical in some scenarios.

By tracking visited vertices, space complexity is O(|V|).

Dijkstra's Algorithm

Dijkstra's algorithm maintains two sets of vertices:

1. The *unvisited set* is a set of vertices that are yet to be considered while looking for minimum cost paths. The unvisited set serves the same purpose as the stack when performing depth first search on a graph.
2. The *visited set* contains all vertices which already have their minimum cost and path computed. The visited set serves the same purpose as the visited set in depth first search of a graph.

算法步骤：

1. 初始化：从起点开始，设置起点的最短路径长度为0（它自己到自己的距离），其他所有点的最短路径长度为无穷大。
2. 对于起点，考虑所有的未处理的邻居，并计算从起点到这些邻居的距离。
3. 从未处理的节点中选择距离最短的节点，将其标记为“已处理”。
4. 对于当前选择的节点，考虑其所有未处理的邻居，并更新其从起点到邻居的距离。
5. 重复第3和4步，直到处理了所有节点。

we wish to use a *priority queue* - in this case one that queues elements by order such that the shortest edge distance to a vertex is at the head of the queue and that this ordering is alway retained as elements (weighted edges to the adjacent vertices) are added. The notion of the queue as an ordered *first in, first out* collection is here modified as *shortest in, first out*.

In this sample implementation, we will build an Object Oriented representation of *DijkstraVertex* and *DijkstraGraph* which, in practice, could easily extend our existing Vertex and Graph classes.
In the case of *DijkstraVertex*, we will instrument the class such that it:

- maintains a dictionary of adjacent vertices with the edge-weight to that vertex as the value in the dict,
- records the distance from the source to this vertex - initially set at infinity,
- records whether the vertex has been visited - initially False,
- records the previous vertex in the path.

```
class DijkstraVertex:

    def __init__(self, node):
        self._id = node
        self._adjacent = dict()
        # Set distance to infinity for all nodes
        self._distance = math.inf
        # Mark all nodes unvisited
        self._visited = False
        # Predecessor
        self._previous = None
```

The DijkstraGraph class is as before except the function add_edge which uses DijkstraVertex.add_neighbour to set the edge relationship and weight:

```
class DijkstraGraph:

    def add_edge(self, frm, to, cost = 0):
        if frm not in self._vertices:
            self.add_vertex(frm)
        if to not in self._vertices:
            self.add_vertex(to)

        self._vertices[frm].add_neighbour(self._vertices[to], cost)
        self._vertices[to].add_neighbour(self._vertices[frm], cost)
```

Which brings us to the implementation of Dijkstra's algorithm using a priority queue to set the shortest path from a defined source vertex:

```
def dijkstra_spf (self, start):

    # Set the distance for the start node to zero
    start.set_distance(0)

    # Put the vertices into the priority queue
    unvisited_queue = list(self._vertices.values())
    heapq.heapify(unvisited_queue)

    while unvisited_queue:
        # Pops a vertex with the smallest distance
        current = heapq.heappop(unvisited_queue)
        current.set_visited()

        #for next in v.adjacent:
        for next in current.get_adjacent():
            # if visited, skip
            if next.is_visited():
                continue
            new_dist = current.get_distance() + current.get_weight(next)

            if new_dist < next.get_distance():
                next.set_distance(new_dist)
                next.set_previous(current)
                #print ('updated : current = %s next = %s new_dist = %s' \
                #    %(current.get_id(), next.get_id(), next.get_distance()))
            else:
                #print ('not updated : current = %s next = %s new_dist = %s' \
                #    %(current.get_id(), next.get_id(), next.get_distance()))
                pass

        # Rebuild heap
        # 1. Pop every item
        while unvisited_queue:
            heapq.heappop(unvisited_queue)
        # 2. Put all vertices not visited into the queue
        unvisited_queue = [v for v in list(self._vertices.values()) if not v.is_visited()]
        heapq.heapify(unvisited_queue)
```

Note that heapq needs to compare instances of DijkstraVertex so that it can arrange by distance and thus enforce ordering. Hence, we must implement a comparison function in DijkstraVertex for "less than" like:

```
def __lt__(self, other):
    return self._distance < other.get_distance()
```

This is an example of a *dunder method* (double underscore) or *magic method*. Finally, having run the method dijkstra_spf, the shortest distance to each vertex in the graph is set. Also the shortest path from the target back to the origin is

set. Thus we add a final convenience method to extract the path as a list of vertex ids:

```
def get_shortest_path (self, target, path):

    if len(path) == 0:
        path.append(target.get_id())

    previous = target.get_previous()
    if previous:
        path.append(previous.get_id())
        self.get_shortest_path(previous, path)
    else:
        # The path is traversed from target to source. So we reverse the
        # path once we have reached the source (i.e.: no previous vertex)
        path.reverse()
```

Note the step to reverse the list ordering - since the path is set originally as 'target' to 'origin'.

Complexity

Using a priority queue, Dijkstra's Algorithm will run in O(n log n) time - where n is the size of the set of vertices. Space complexity in this case is O(n).