Homework 3

May 18, 2021

0.1 Algorithmic Design Homework 3

0.1.1 Exercise 1.

Implement the binary heap-based version of the Dijkstra's algorithm.

0.1.2 Solution

In the cell below we implement the heap-based version of the Dijkstra's algorithm on the Graph data structure written in the file `Graph.py` and using the binary heap defined in `Heap.py`.

The heap implements all the methods we saw during the lectures and some more which we found useful during the implementation of the Dijkstra's algorithm, in particular it implements a way to decrease a value without needing the index of it in the heap, obviously to accomplish this it needs more space because it needs to store a dictionary that maps a value to a specific index, the key of this mapping is defined by the user. We used this feature in the Dijkstra's algorithm to set as a key the name of the vertex, which is unique in the graph

Designing the graph was more complicated because it can be implemented in different ways, from a high level perspective we chosed to define the graph as a set of vertexes and a vertex as a set of edges, in the implementation however we don't use a set but a dictionary because it allows finding a specific vertex or edge of the graph in $\Theta(1)$ time on average. Defining the edge as an object is very useful if we want to store additional information about the connection, we didn't use this feature in the code below, however we used the fact that the vertex is an object to add the members `v.d`, `v.prev` and `v.importance` when they were needed. Obviously this implementation requires more space than simply storing a dictionary of {source, destination} pairs, however it allows for a very high generalizability

[1]:
```
from numpy import inf
from src import *
from warnings import warn
from random import randint
from typing import TypeVar, Union
from copy import deepcopy

n = TypeVar("n")
```

```
def init_sssp(G: Graph, s: n):
    """Initializes Dijkstra algorithm

    Args:
        G (Graph): Graph to initialize
        s (n): source

    Raises:
        RuntimeWarning: If s is not present in the graph
    """
    u = G.get_vertex(s)
    if u is None:
        message = f"The source: {s} is not present in the graph, exiting"
        warn(message, RuntimeWarning)
        return

    for v in G.V():
        v.d = inf
        v.pred = None

    u.d = 0.


def relax(Q: Binary_heap, u: Vertex, v: Vertex):
    """Relaxes a node

    Args:
        Q (Heap): Queue implemented by a binary heap
        u (Vertex): Vertex already relaxed
        v (Vertex): Vertex to be relaxed
    """
    w = u.get_weight(v.name)
    if v.d > u.d + w:
        v.d = u.d + w
        v.pred = u
        Q.decrease_value((v.d, v.name), (v.d, v.name))


def dijkstra(G: Graph, s: n):
    """Implements the dijkstra algorithm for finding the single source shortest
    path from the source s

    Args:
        G (Graph): Graph to complute the distance
        s (n): Name of the source in the graph

    Raises:
```

```
        RuntimeWarning: If s is not present in the graph
    """
    init_sssp(G, s)
    # Initialize Q with tuples (v.d, v.name), this is much lighter than storing␣
↪v itself
    Q = Binary_heap([(v.d, v.name) for v in G.V()], total_order=lambda x, y:␣
↪x[0] < y[0], dict_key=lambda x: x[1])
    while not Q.is_empty():
        u = Q.remove_min()
        u = G.get_vertex(u[1])
        # Relax all the childs of v
        for v in G.get_childs(u.name):
            relax(Q, u, v)


# Create a simple graph for testing
G = Graph(directed=True)
G.add_edge("A", "B", 1000)
G.add_edge("A", "C", 1)
G.add_edge("C", "D", 5)
G.add_edge("D", "B", 8)
G.add_vertex("E")

dijkstra(G, "A")
[(v.name, v.d) for v in G.V()]
```

[1]:
```
[('B', 14.0), ('A', 0.0), ('C', 1.0), ('D', 6.0), ('E', inf)]
```

[2]:
```
# Creates the graph from the slides for testing
G_slides = Graph()

edges = [(1,6,1), (1,5,1), (2,3,2), (3,2,1), (3,4,3), (4,7,1),
         (4,8,3), (5,6,1), (5,1,3), (6,8,1), (7,8,1), (8,1,1), (8,7,1)]

[G_slides.add_edge(s, d, w) for s, d, w in edges]
[setattr(v, "importance", v.name) for v in G_slides.V()]

dijkstra(G_slides, 1)
[(v.name, v.d) for v in G_slides.V()]
```

[2]:
```
[(6, 1.0),
 (1, 0.0),
 (5, 1.0),
 (3, inf),
 (2, inf),
 (4, inf),
 (7, 3.0),
```

```
(8, 2.0)]
```

0.1.3 Exercise 2.

0.1.4 Solution

The implementation of the function to add the shortcuts to a graph can be found in the file `Graph.py` at line 90, below we will implement the bidirectional version of the Dijkstra algorithm

[3]:
```python
def dijkstra_iteration(G: Graph, Q: Binary_heap) -> n:
    """Performs a single iteration of the bidirectional dijkstra algorithm

    Args:
        G (Graph): Graph to iterate
        Q (Binary_heap): Corresponding queue

    Returns:
        n: name of the vertex that has been completed
    """
    u = Q.remove_min()
    u = G.get_vertex(u[1])
    for v in G.Adj(u.name):
        v = G.get_vertex(v)
        # Only relax if the importance is greater
        if v.importance < u.importance:
            continue
        Q.insert((v.d, v.name))
        relax(Q, u, v)

    return u.name


def get_path(G: Graph, v: n, d: n) -> "tuple(list[n], float)":
    """Returns the path to be travelled to reach d from v and its associated␣
 ↪weight

    Args:
        v (n): Source of the path
        d (n): Destination of the path

    Returns:
        tuple(list[n], float): the first element is the path to travel to reach
        d from v, the second is it's associated weight
    """
    v = deepcopy(v)
    path = [v]
```

```
    w = 0.

    while v != d:
        v = G.get_vertex(v) # Get the vertex corresponding to v
        edge = v.pred[v.name] # Get the edge from v.pred to v
        path.extend(edge.decompose()[::-1]) # Add to the path the vertexes the␣
↪edge passes by in reverse order
        w += edge.weight # Update the overall weight
        v = v.pred.name # Update v

    return path, w


def remove_duplicates(a: list) -> list:
    """Simple function that removes subsequent duplicate values from a list

    Args:
        a (list): list to remove the duplicates

    Returns:
        list: list with the duplicates removed
    """
    return [a[i] for i in range(0, len(a)) if a[i-1]!=a[i]]


def Bidirectional_dijkstra(G: Graph, s: n, d: n) -> "tuple(list[n], float)":
    """Implements the bidirectional Dijkstra algorithm to find the
    shortest path from s to d of G, note that if the node importance
    is not set properly this algorithm can find sub-optimal paths
    or

    Args:
        G (Graph): Graph to compute the shortest path
        s (n): Source or starting point of the path
        d (n): Destination or ending of the path

    Returns:
        tuple(list[n], float): the first element is the path to travel to reach
        d from s, the second is it's associated weight

    Raises:
        RuntimeWarning:
            If a path between s and d does not exist, in this case returns None
    """
    # Create G_up and G_down
    G_down = G.T()
    G_up = G
```

```
    # Add the shortcuts
    G_up.add_shortcuts()
    G_down.add_shortcuts()
    # Initialize the search
    init_sssp(G_up, s)
    init_sssp(G_down, d)

    # Create the binary heaps
    u = G_up.get_vertex(s)
    Q_up = Binary_heap([(u.d, u.name)], total_order=lambda x, y: x[0] < y[0],␣
↪dict_key=lambda x: x[1])
    u = G_up.get_vertex(d)
    Q_down = Binary_heap([(u.d, u.name)], total_order=lambda x, y: x[0] < y[0],␣
↪dict_key=lambda x: x[1])

    # Create two sets to keep track of the vertexes encountered so far
    vertices_up = set()
    vertices_down = set()
    common_vertex = ""
    # I iterate on G_up and G_down iteratively until I find a common vertex
    # or until both queues are empty, if this is the case then it means that
    # I haven't found a common vertex and so there is no path from s to d
    while not Q_up.is_empty() or not Q_down.is_empty():
        if not Q_up.is_empty():
            up = dijkstra_iteration(G_up, Q_up)
            if up in vertices_down:
                common_vertex = up
                break

        if not Q_down.is_empty():
            down = dijkstra_iteration(G_down, Q_down)
            if down in vertices_up:
                common_vertex = down
                break
        vertices_up.add(up)
        vertices_down.add(down)

    if common_vertex == "":
        warn(f"A path between {s} and {d} does not exist", RuntimeWarning)
        return

    # Get the path to be travelled to reach s from d and its weight
    p1, w1 = get_path(G_up, common_vertex, s)
    p2, w2 = get_path(G_down, common_vertex, d)
    p1.reverse()
    return remove_duplicates([*p1, *p2]), w1+w2
```

[4]:
```
# Test on the simple graph defined above
for v in G.V():
    v.importance=randint(0, 10)

Bidirectional_dijkstra(G, "A", "B")
```

[4]:
```
(['A', 'C', 'D', 'B'], 14.0)
```

[5]:
```
# Test on the graph on the slides
Bidirectional_dijkstra(G_slides, 1, 8), Bidirectional_dijkstra(G_slides, 3, 6)
```

[5]:
```
(([1, 6, 8], 2.0), ([3, 4, 7, 8, 1, 6], 7.0))
```

0.2 Conclusion

The code of the bidirectional Dijkstra algorithm can be improved by creating two ad-hoc procedures to build G_up and G_down, in particular in these graphs the connections that go from a vertex with higher importance to one with lower importance are never used, so we could create G_up and G_down with these connections removed by default, this saves both time (we don't have to check if v.importance < u.importance) and space (because we store less connections). We sticked to this implementation because the homework says that the bidirectional Dijkstra has to operate on the graphs decorated by the algorithm at Point 2a, for completeness, below I implement the two procedures to create G_up and G_down like we described above

[ ]:
```
def remove_useless_edges(G: Graph) -> Graph:
    """Removes connections that go from a higher importance to a lower␣
 ↪importance,
    be aware that the removal is done in place!

    Args:
        G (Graph): Graph to remove the connections

    Returns:
        Graph: The graph with the connections removed

    Raises:
        RuntimeWarning: If a vertex does not have an importance
    """
    try:
        for edge in G.E():
            s = edge.get_source()
            d = edge.get_destination()
            if G.get_vertex(s).importance > G.get_vertex(d).importance:
                G.remove_edge(s, d)

    except AttributeError:
```

```
        warn(f"The graph contains vertexes without an importance attribute,␣
↪exiting", RuntimeWarning)
        return

    return G


def get_G_up(G: Graph) -> Graph:
    """Computes G_up, useful for the bidirectional Dijkstra algorithm

    Args:
        G (Graph): Graph to compute G_up

    Returns:
        Graph: The G_up graph
    """
    G_up = deepcopy(G)

    G_up.add_shortcuts()
    return remove_useless_edges(G_up)


def get_G_down(G: Graph) -> Graph:
    """Computes G_down, useful for the bidirectional Dijkstra algorithm

    Args:
        G (Graph): Graph to compute G_down

    Returns:
        Graph: The G_down graph
    """
    G_down = G.T()

    G_down.add_shortcuts()
    return remove_useless_edges(G_down)
```

Now we can use `get_G_up` and `get_G_down` in the `Bidirectional_dijkstra` function, we can also update `dijkstra_iteration` to avoid the checking of the importances