1 Dijkstra's and Bellman-Ford

Here, you will implement the two closely-related shortest-path algorithms that we've seen in class.

1.0.1 If you're using Datahub:

- Run the cell below **and restart the kernel if needed**

1.0.2 If you're running locally:

You'll need to perform some extra setup. #### First-time setup * Install Anaconda following the instructions here: https://www.anaconda.com/products/distribution * Create a conda environment: `conda create -n cs170 python=3.8` * Activate the environment: `conda activate cs170` * See for more details on creating conda environments https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html * Install pip: `conda install pip` * Install jupyter: `conda install jupyter`

Every time you want to work

- Make sure you've activated the conda environment: `conda activate cs170`
- Launch jupyter: `jupyter notebook` or `jupyter lab`
- Run the cell below **and restart the kernel if needed**

In [ ]:
```
# Install dependencies
!pip install -r requirements.txt --quiet
```

In [10]:
```
import otter

assert (otter.__version__ >= "4.4.1"), "Please reinstall the requirements and restart your ker

grader = otter.Notebook("shortest-paths.ipynb")
import numpy as np
from numpy.random import randint
from time import time
import heapq
import tqdm
import networkx as nx
import matplotlib.pyplot as plt
import numpy.random as random

rng_seed = 42
```

1.0.3 Representing graphs in code (Part 2!!!)

Unlike last week's assignment, our graphs are now weighted, so we'll need to store weights alongside the edge information. Using an edge list representation, we can represent directed edges (u, v) with weight w by creating a list of tuples `(u, v, w)`.

However, like last week, we'd like to represent our graph using adjacency lists. We can represent the directed edge (u, v) with weight w by storing the tuple `(v, w)` in `adj_list[u]`.

```
In [11]: def generate_adj_list(n, edge_list):
             """
             args:
                 n:int = number of nodes in the graph. The nodes are labelled with integers 0 through n
                 edge_list:List[Tuple[int,int,int]] = edge list where each tuple (u,v,w) represents the
                     and weighted edge (u,v,w) in the graph
             return:
                 A List[List[Tuple[int, int]]] representing the adjacency list
             """
             adj_list = [[] for i in range(n)]
             for u, v, w in edge_list:
                 adj_list[u].append((v, w))
             for nodes in adj_list:
                 nodes.sort()
             return adj_list

         def draw_graph(adj_list):
             """Utility method for visualizing graphs

             args:
                 adj_list (List[List[Tuple[int, int]]]): adjacency list of the graph given by generate_
             """
             G = nx.DiGraph()
             for u in range(len(adj_list)):
                 for v, w in adj_list[u]:
                     G.add_edge(u, v, weight=w)
             nx.draw(G, with_labels=True)
```

1.0.4 Priority Queues in Python

For simplicity, we've given you the following implementation of a priority queue, which uses the `heapq` module under the hood. Our implementation implements `insert` and `deleteMin` as described in DPV, but does not include the `decreaseKey` operation since it's not supported by Python's `heapq` module. However, it turns out that we don't need `decreaseKey` to implement any of the algorithms in this assignment - we'll discuss this more in the relevant sections.

You don't have to understand our implementation, but if you're curious to learn more, the `heapq` module documentation is available here: https://docs.python.org/3/library/heapq.html

You may modify this implementation as you see fit, but if you do so, ensure that your modified implementation is correct and efficient. **A priority queue implementation which is slower than ours may result in your code timing out during grading.**

```
In [12]: class PriorityQueue:
             def __init__(self):
                 self.queue = []

             def insert(self, priority, item):
                 heapq.heappush(self.queue, (priority, item))

             def deleteMin(self):
                 return heapq.heappop(self.queue)[1]

             def __len__(self):
                 return len(self.queue)
```

The operations `insert` and `deleteMin` work as follows:

```
In [54]: pq = PriorityQueue()

         for i in range(10):
             pq.insert(-i, i)

         # should print 9, 8, 7
         print(pq.deleteMin())
         print(pq.deleteMin())
         print(pq.deleteMin())
```

```
9
8
7
```

1.0.5 Q0: The `update` function (Optional)

As described in section 4.6.1. of DPV, we can implement a subroutine `update` to update vertex distances in our graph. Bellman-Ford can be thought of as a applying a sequence of `update` operations, as described in DPV, but it turns out that Djikstra's algorithm can too! In this part, you may implement the `update` function which can be used in both algorithms.

Since later on, we will ask you to reconstruct the actual shortest path, it may be useful to keep track of the predecessor of each vertex when updating an edge.

If you'd like, you can safely skip this part, as it's not worth any points.

Points: 0.0

```
In [14]: def update(u, v, w, dists, prev=None):
             """Updates the distance dists[v] in the dists array based on the
             update procedure described in DPV.

             Args:
                 u (int): starting node of th eedge (u, v)
                 v (int): ending node of the edge (u, v)
                 w (int): weight of the edge (u, v)
                 dists (List[int]): The distance array used in our shortest-path algorithm. The source
                     s actually does not need to be specified here, but it will be needed in the
                     shortest-path algorithm.
                 prev (List[int]): Array keeping track of the previous node along the shortest path
             """
             # BEGIN SOLUTION NO PROMPT
             if dists[u] + w < dists[v]:
                 dists[v] = dists[u] + w
                 prev[v] = u
             # END SOLUTION
             # TODO: your code here!
             pass
```

1.0.6 Q1: Dijkstra's Algorithm

If you need a refresher on how the algorithm works, check out pp.120-121 from DPV: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap4.pdf

Unlike the implementation in DPV, we don't have access to the `decreaseKey` operation, so we'll make the following modification: Instead of calling `decreaseKey`, we'll just insert a new copy of the vertex with the updated distance into the priority queue, and not worry about removing the old copy. This could result in multiple copies of the same vertex in the priority queue, but should not affect the correctness of the algorithm.

> **Task 1:** Compute a shortest path from s to t using Dijkstra's algorithm and return the path as a list of nodes on that path.

For example, the path $s \to a \to b \to c \to t$ corresponds to the list `[s, a, b, c, t]`.

All edge weights are non-negative. If no path exists, return the empty list `[]`. If multiple shortest paths exist, you may return any of them.

Points: 3

```
In [29]: def dijkstra(G, s, t):
```

```
    """Implements Dijkstra's algorithm to find the shortest weighted path from s to t.

    Args:
        G (List[List[Tuple[int, int]]]): The weighted adjacency list of the graph
        s (int): The source node
        t (int): The target node

    Returns:
        List[int]: A list of nodes starting with s and ending with t representing the
        shortest weighted s to t path if it exists. Returns an empty list otherwise.
    """
    # BEGIN SOLUTION
    n = len(G)
    known_region = set()
    dists = [float('inf')]*n
    prev = [None]*n
    dists[s] = 0
    pq = PriorityQueue()

    for v in range(n):
        pq.insert(dists[v], v)

    while len(known_region) < n:
        v = pq.deleteMin()
        if v in known_region:
            continue
        known_region.add(v)

        for z, w in G[v]:
            update(v, z, w, dists, prev)
            pq.insert(dists[z], z)

    # if t not reachable from s, return empty list
    if dists[t] == float('inf'):
        return []

    # if path exists, reconstruct the shortest path
    path = []
    curr = t
    while curr != s:
        path.append(curr)
        curr = prev[curr]
    path.append(curr)
    path.reverse()
    return path
    # END SOLUTION
```

In [ ]:
```
grader.check("q1")
```

1.0.7 Q2: Bellman-Ford

If you need a refresher on how the algorithm works, check out pp.122-124 from DPV: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap4.pdf

> **Task 2:** Compute a shortest path from s to t using Bellman-Ford and return the path as a list of nodes on that path.

For example, the path $s \to a \to b \to c \to t$ corresponds to the list `[s, a, b, c, t]`.

If no $s \to t$ path exists, or if the graph has a negative cycle, return the empty list `[]`. If multiple shortest paths exist, you may return any of them.

Points: 3

```
In [110]: def bellman_ford(G, s, t):
              """Implements the Bellman-Ford algorithm for single-source shortest paths.

              Args:
                  G (List[List[Tuple[int, int]]]): The weighted adjacency list of the graph
                  s (int): The source node
                  t (int): The target node

              Returns:
                  List[int]: A list of nodes starting with s and ending with t representing the
                  shortest weighted s to t path if it exists. Returns an empty list otherwise.
              """
              # BEGIN SOLUTION
              n = len(G)
              dists = [float('inf')]*n
              dists[s] = 0
              prev = [None]*n
              V = len(G)

              for _ in range(V-1):
                  for u in range(V):
                      for v, w in G[u]:
                          update(u, v, w, dists, prev)

              # final iteration to check for negative cycles
              for u in range(V):
                  for v, w in G[u]:
                      if dists[u] + w < dists[v]:
                          return []

              # if t not reachable from s, return empty list
              if dists[t] == float('inf'):
```

```
        return []

    # if path exists, reconstruct the shortest path
    path = []
    curr = t
    while curr != s:
        path.append(curr)
        curr = prev[curr]
    path.append(curr)
    path.reverse()
    return path
    # END SOLUTION
```

```
In [ ]: grader.check("q2")
```

1.1 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit.

```
In [ ]: grader.export(pdf=False, force_save=True, run_tests=True)
```