Example sheet 4
Graphs and path finding
Algorithms—DJW—2022/2023

**Question 6.** Modify `bfs_path(g, s, t)` to find all shortest paths from `s` to `t`. *[There is a tester, linked to on the course website, for you to check your code.]*

---

Algorithms tick: bfs-all

Find All Shortest Paths

Breadth-first search can be used to find a shortest path between a pair of vertices. Modify the standard `bfs_path` algorithm so that it returns all shortest paths.

*Please submit a source file `bfs_all.py` on Moodle.* It should implement a function

```python
shortest_paths(g, s, t)

# Find all shortest paths from s to t
# Return a list of paths, each path a list of vertices starting with s and
```

The graph `g` is stored as an adjacency dictionary, for example `g = {0:{1,2}, 1:{}, 2:{1,0}}`. It has a key for every vertex, and the corresponding value is the set of that vertex's neighbours.
Write an amendment to the classic breadth-first search algorithm to return all shortest paths on a graph where all paths have equal weight. Ensure that your algorithm can cope with cycles. Ensure your algorithm can cope with the case where there is no path. Implement this in python. The graph will be stored as a dictionary of sets.
To modify the classic Breadth First Search algorithm to return all shortest paths on a graph where all paths have equal weight, we can make use of a modified version of Dijkstra's algorithm. This algorithm is known as the "All Shortest Paths Algorithm," and it is a modification of Dijkstra's algorithm that can handle graphs with cycles.

The basic idea behind the All Shortest Paths Algorithm is to keep track of all the possible shortest paths from the starting vertex to each of the other vertices in the graph. We can achieve this by maintaining a dictionary that maps each vertex to a list of all possible shortest paths from the starting vertex to that vertex.

Here is the Python implementation of the All Shortest Paths Algorithm:

```python
def all_shortest_paths(g, s, t):
    distance = {v: float('inf') for v in g}
    distance[s] = 0
    visited = {v: False for v in g}
    paths = {v: [] for v in g}
    paths[s] = [[s]]
    queue = deque([[s]])

    while queue:
        v = queue.popleft()
        if v == t:
            return paths[t]

        for w in g[v]:
            if not visited[w]:
                new_dist = distance[v] + 1
                if new_dist <= distance[w]:
                    distance[w] = new_dist
                    queue.append([w])

                if new_dist == distance[w]:
                    for path in paths[v]:
                        paths[w].append(path + [w])

            visited[v] = True

    return []
```

This implementation uses a dictionary to store the shortest path lengths and a queue to keep track of the vertices to be processed. It iterates through the graph, updating the shortest path lengths as it finds shorter paths, and updates the dictionary of paths accordingly.
To prove that the All Shortest Paths Algorithm terminates, we can use the fact that the algorithm is based on Breadth First Search (BFS), which is a well-known algorithm that always terminates on finite graphs. In BFS, each vertex is visited at most once, and the algorithm terminates when all reachable vertices have been visited. Since the All Shortest Paths Algorithm is a modification of BFS, it also terminates on finite graphs.

To prove that the All Shortest Paths Algorithm produces the correct result, we need to show that it computes all possible shortest paths from the starting vertex to each of the other vertices in the graph.

Let \( P(u, v) \) be the set of all possible shortest paths from \( u \) to \( v \). We want to show that the paths computed by the algorithm \( \text{all_shortest_paths}(g, s, t) \) are exactly \( P(s, t) \) for any pair of vertices \( (s, t) \).

We can prove this by induction on the number of vertices in the graph. Let \( g = (V, E) \) be a graph with \( n \) vertices, and let \( s \) and \( t \) be two vertices in \( g \).

**Base case:** If \( n = 1 \), then \( s \) and \( t \) are the same vertex, and the only possible path is the empty path. The algorithm correctly returns the empty path.

**Induction hypothesis:** Assume that the algorithm correctly computes all possible shortest paths from \( s \) to each of the other \( n - 1 \) vertices in \( g \).

**Induction step:** Consider a graph \( g' = (V', E') \) obtained from \( g \) by adding a new vertex \( w \) and edges \( s \rightarrow w \) and \( w \rightarrow t \). We want to show that the algorithm correctly computes all possible shortest paths from \( s \) to each of the other \( n \) vertices in \( g' \).

Now, consider a vertex \( v \) in \( g' \), where \( v \) is not equal to \( s \) or \( t \). We need to show that the algorithm computes all possible shortest paths from \( s \) to \( v \).

**Case 1:** \( v \) is not reachable from \( s \) in \( g \). In this case, the algorithm correctly returns an empty list, since there is no path from \( s \) to \( v \).
CLRS3 lemma 24.15 (used in Bellman-Ford). Consider a weighted directed graph. Consider any shortest path from \( s \) to \( t \),
\[
    s = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k = t
\]
Suppose we initialize the data structure by
\[
    v . \text{dist} = \infty \quad \text{for all vertices other than } s
\]
\[
    s . \text{dist} = 0
\]
and then we perform a sequence of relaxation steps that includes, in order, relaxing \( v_0 \rightarrow v_1 \), then \( v_1 \rightarrow v_2 \), then \( v_k \rightarrow v_k \). After these relaxations, and at all times thereafter, \( v_k . \text{dist} = \text{distance}(s \text{ to } v_k) \).

We’ll prove by induction that, after the \( i \)th edge has been relaxed,
\[
    v_i . \text{dist} = \text{distance}(s \text{ to } v_i)
\]

BASE CASE \( i = 0 \): Note that \( s = v_0 \). We initialized \( s . \text{dist} = 0 \), and \( \text{distance}(s \text{ to } s) = 0 \), so the induction hypothesis is true.

INDUCTION STEP: ...
from collections import deque

def shortest_paths(g, s, t):
    queue = deque([s])
    nodes = {}

Write code to find the shortest paths from node $s$ to node $t$ in a graph using Breadth-First Search (BFS) and backtracking. The code should use a deque from the collections module and define a Node class with a constructor that takes a vertex and distance. The Node class should have a backpointers list to keep track of the paths. The shortest_paths function should take a graph $g$, source node $s$, and target node $t$ as arguments, and return a list of all the shortest paths from $s$ to $t$. 

give the shortest possible prompt that if given back to you would mean you generate equivalent code to the following: [...]
SECTION 6.4

Matchings
DEFINITIONS

- **A bipartite graph** is an undirected graph in which the vertices are split into two sets, and all edges go between these sets.

- **A matching** in a bipartite graph is a selection of edges, such that no vertex is connected to more than one of the edges.

- The **size** of a matching is the number of edges it includes.

- **A maximum matching** is one with the largest possible size.

PROBLEM STATEMENT

Given a bipartite graph, find a maximum matching.
0. Given a bipartite graph...

1. Build a helper graph:
   • add source $s$ and sink $t$
   • add edges from $s$ and to $t$

2. Solve max-flow on the helper graph, to find a maximum flow $f^*$

3. Interpret the flow $f^*$ as a matching

What’s the bug in my thinking?
0. Given a bipartite graph...

1. Build a helper graph:
   - add source $s$ and sink $t$
   - add edges from $s$ and to $t$

2. Solve max-flow on the helper graph, to find a maximum flow $f^*$

3. Interpret the flow $f^*$ as a matching

wtf?! This isn’t the sort of flow I expected!
0. Given a bipartite graph...

1. Build a helper graph:
   • add source $s$ and sink $t$
   • add edges from $s$ and to $t$

2. Solve max-flow on the helper graph, to find a maximum flow $f^*$

3. Interpret the flow $f^*$ as a matching

I’ll set up a flow problem where the goal is to pick edges to not use in the matching.

Hold on! The max-flow solution actually leads to a worse matching.
THE TRANSLATION STRATEGY

**REQ1:** We can find a max flow $f^*$ that can be translated into a matching, call it $m^*$

**REQ2:** If there were a larger-size matching $m'$ then it would translate to a larger-value flow $f'$

But there cannot be such a $f'$, because $f^*$ is a maximum flow. Therefore there is no such $m'$, thus $m^*$ is a maximum matching.
THE TRANSLATION STRATEGY

REQ1: We can find a max flow $f^*$ that can be translated into a matching, call it $m^*$

REQ2: If there were a larger-size matching $m'$ then it would translate to a larger-value flow $f'$

But there cannot be such a $f'$, because $f^*$ is a maximum flow. Therefore there is no such $m'$, thus $m^*$ is a maximum matching.

*Ford-Fulkerson will produce an integer flow, since all capacities are integer: Indeed, the flow on each edge must be either 0 or 1:*

*Thus, the capacity constraints tell us that, when we translate $f^*$ into an edge selection, it meets the definition of “matching”.*

*When we translate matching $\leftrightarrow$ flow in the obvious way, value(flow)=size(matching)*

*So if we had a larger-size matching $m'$ it would translate to a larger-value flow $f'$.*
Ex5q6. A signal failure can prevent travel in both directions between a pair of adjacent stations. How many signal failures it would take to prevent travel from Kings Cross to Embankment?
SECTIONS 6.5 and 6.6

Prim’s and Kruskal’s algorithms
Similarity matrix

Similarity graph + subtree

- high similarity
- low similarity
- tree edges
DEFINITIONS

Given a connected undirected graph \( g \) with edge weights,

- A **spanning tree** of \( g \) is a tree that connects all of \( g \)'s vertices, using some or all of \( g \)'s edges
- The **weight** of a spanning tree is the sum of all its edge weights
- A **minimum spanning tree (MST)** is a spanning tree that has minimum weight among all spanning trees

PROBLEM STATEMENT

Given such a graph, find a minimum spanning tree
Let’s build up a tree, edge by edge.

SIMPLE GREEDY ALGORITHM: Which edge would you add next, to grow the tree?
PRIM’S ALGORITHM

Given a tree we’ve built so far,

1. look at the frontier of vertices we might add next, and at the cut between our tree and those vertices
2. pick the lowest-weight edge across this cut, and add it to the tree
3. Assert: the tree we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

PROOF OF CORRECTNESS (OUTLINE)

We can prove the assertion on line 3, using the “breakpoint” proof strategy plus some fiddly maths about trees. The final output is hence a minimum spanning tree.
PRIM’S ALGORITHM

Given a tree we’ve built so far,

1. look at the frontier of vertices we might add next, and at the cut between our tree and those vertices
2. pick the lowest-weight edge across this cut, and add it to the tree
3. Assert: the tree we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

Don’t recompute the frontier every iteration.
Instead, store it & update it.
PRIM’S ALGORITHM

Given a tree we’ve built so far,
1. look at the frontier of vertices we might add next,
   and at the cut between our tree and those vertices
2. pick the lowest-weight edge across this cut,
   and add it to the tree
3. Assert: the tree we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

```python
def prim(g, s):
    for v in g.vertices:
        v.distance = ∞
        v.in_tree = False
    s.come_from = None
    s.distance = 0
    toexplore = PriorityQueue([s], sortkey = λv: v.distance)

    while not toexplore.isempty():
        v = toexplore.popmin()
        v.in_tree = True
        for (w, edgeweight) in v.neighbours:
            if (not w.in_tree) and edgeweight < w.distance:
                w.distance = edgeweight
                w.come_from = v
                if w in toexplore:
                    toexplore.decreasekey(w)
                else:
                    toexplore.push(w)
```

```python
def dijkstra(g, s):
    for v in g.vertices:
        v.distance = ∞
    s.distance = 0
    toexplore = PriorityQueue([s], sortkey = λv: v.distance)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w, edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            if dist_w < w.distance:
                w.distance = dist + w
                if w in toexplore:
                    toexplore.decreasekey(w)
                else:
                    toexplore.push(w)
```
Let’s build up a **forest**, edge by edge.

**SIMPLE GREEDY ALGORITHM:**
Which edge would you add next, to grow the forest?
**KRUSKAL’S ALGORITHM**

Given a **forest** we’ve built so far,

1. look at all the edges that would join two fragments of the forest
2. pick the lowest-weight one and add it to the tree, thereby joining two fragments

3. **Assert**: the forest we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

**PROOF OF CORRECTNESS (OUTLINE)**

We can prove the assertion on line 3, using the “breakpoint” proof strategy plus some fiddly maths about trees. The final output is hence a minimum spanning tree.
EXERCISE. Run through the steps of Kruskal’s algorithm.
Similarity matrix of submitted coursework.
Algorithms challenge: rank-sim

Order items by similarity

In this tick, your aim is to find a good order for a set of items, given similarity scores between them. You are given a list of pairs of items and their similarity scores (this list doesn't include all pairs). Here is an example:

- ticksim_train.csv

We saw an illustration in the video for section 6.6. We were given a list of students, and also the similarity scores between their submitted code for an Algorithms tick. We used Kruskal's algorithm to find an ordering for the students, such that two students with a high similarity score appeared close to each other in the order.

Your aim is to produce a good ordering of items. To be precise, let $s_{uv} \in (0, 1)$ be the similarity score between items $u$ and $v$. Your score will be

$$\text{score} = 100 \times \frac{x - m}{-m} \quad \text{where} \quad x = \frac{1}{MN} \sum_{\text{pairs } (u,v)} |z_u - z_v| \log(1 - s_{uv}).$$

Here $z_u$ is the index of item $u$ in your ordering, $M$ is the number of pairs, and $N$ is the number of items. The normalization is so that the score is always $\leq 100$ (since $x \leq 0$); and the constant $m$ is the expected score from a random ordering,

$$m = \frac{1}{3M} \sum_{\text{pairs } (u,v)} \log(1 - s_{uv}),$$
KRUSKAL’S ALGORITHM

Given a forest we’ve built so far,

1. look at all the edges that would join two fragments of the forest
2. pick the lowest-weight one and add it to the tree, thereby joining two fragments
3. Assert: the forest we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

Don’t recompute these edges every iteration.

Just pre-sort the list of all edges, then ignore those that are within-fragment.
The abstract data type **DisjointSet** stores a collection of disjoint sets, and supports

- `addsingleton(v)`
- `p = getsetwith(v)`
- `merge(p,q)`