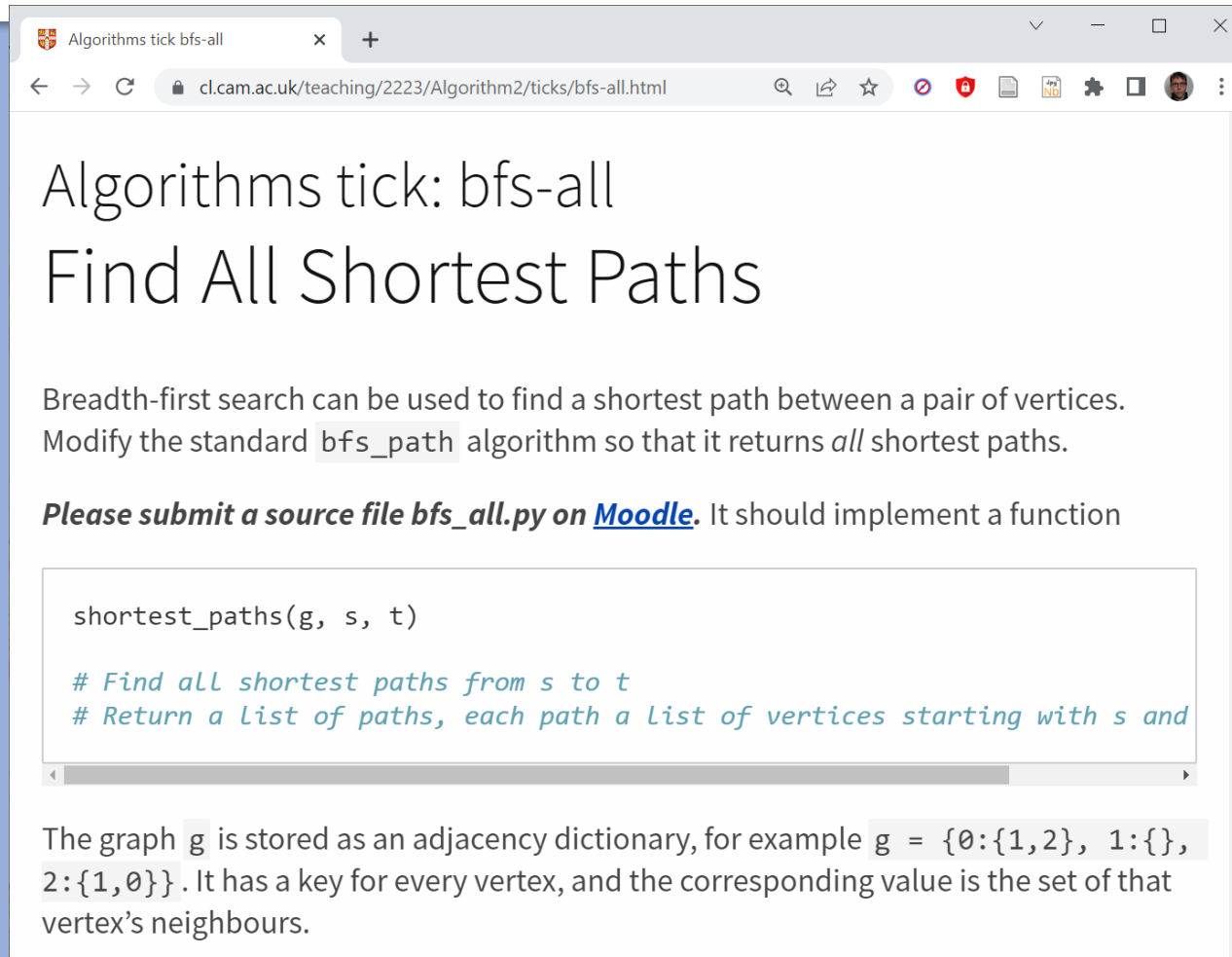


Example sheet 4

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

```
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.

Euan Worth (Queens')

The screenshot shows the ChatGPT web interface. The browser address bar displays <https://chat.openai.com/chat>. The page title is "ChatGPT". On the left sidebar, there is a "New chat" button and a list of previous conversations: "Crow and Dryad's Dance", "Maximizing UK's Prosperity Po", "Jack and Finnegan", "Story Of Mysterious Supernatu", "Light Modern Dinner Party Me", and "BFS Shortest Path Algorithm". At the bottom of the sidebar are options for "Clear conversations", "Upgrade to Plus" (with a "NEW" badge), "Dark mode", "Updates & FAQ", and "Log out".

The main content area features the "ChatGPT" logo and three columns: "Examples" (with a sun icon), "Capabilities" (with a lightning bolt icon), and "Limitations" (with a warning triangle icon). Below these are three boxes: "Examples" contains the text "Explain quantum computing in simple terms" followed by a right arrow; "Capabilities" contains "Remembers what user said earlier in the conversation"; "Limitations" contains "May occasionally generate incorrect information".

A large text input box contains the following text: "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 you 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." A paper plane icon is visible at the bottom right of the input box.

At the bottom of the page, there is a footer: "ChatGPT Feb 13 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve."



To modify the classic Breadth First Search on a graph where all paths have equal weight, we use a version of Dijkstra's algorithm. This is often called "Shortest Path Algorithm," and it is a modification of Dijkstra's algorithm for graphs with cycles.

The basic idea behind the All Shortest Paths algorithm is to find all possible shortest paths from the source vertex to every other vertex in the graph. We can achieve this by starting from the source vertex and building a list of all possible shortest paths to each vertex.

Here is the Python implementation:

```
1 def all_shortest_paths(g, s, t):
2     distance = {v: float('inf') for v in g}
3     distance[s] = 0
4     visited = {v: False for v in g}
5     paths = {v: [] for v in g}
6     paths[s] = [[s]]
7     queue = deque([s])
8
9     while queue:
10        v = queue.popleft()
11        if v == t:
12            return paths[t]
13
14        for w in g[v]:
15            if not visited[w]:
16                new_dist = distance[v] + 1
17                if new_dist <= distance[w]:
18                    distance[w] = new_dist
19                    queue.append(w)
20                    if new_dist == distance[w]:
21                        for path in paths[v]:
22                            paths[w].append(path + [w])
23
24        visited[v] = True
25
26    return []
```

EW

Using thorough mathematical language and sound reasoning, prove both that your algorithm terminates, and produces the correct result.



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 `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.

EW

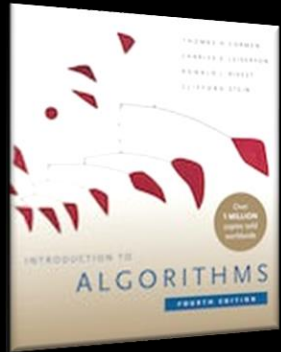
Please finish the proof

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' , 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 ,

$$s = v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k = t.$$

Suppose we initialize the data structure by

$$v.\text{dist} = \infty \text{ for all vertices other than } s$$

$$s.\text{dist} = 0$$

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 ... then $v_{k-1} \rightarrow v_k$. 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: ...

Jack Peck (Jesus)

The screenshot shows the ChatGPT web interface in a browser window. The address bar shows the URL <https://chat.openai.com/chat>. The page title is "ChatGPT". On the left, there is a sidebar with a "New chat" button and a list of previous conversations: "Crow and Dryad's Dance", "Maximizing UK's Prosperity Po", "Jack and Finnegan", "Story Of Mysterious Supernatu", "Light Modern Dinner Party Me", and "BFS Shortest Path Algorithm". At the bottom of the sidebar are options for "Clear conversations", "Upgrade to Plus" (with a "NEW" badge), "Dark mode", "Updates & FAQ", and "Log out".

The main content area features the "ChatGPT" logo and three columns: "Examples" (with a sun icon), "Capabilities" (with a lightning bolt icon), and "Limitations" (with a warning icon). Below these are three boxes: "Examples" contains the text "Explain quantum computing in simple terms" →; "Capabilities" contains "Remembers what user said earlier in the conversation"; "Limitations" contains "May occasionally generate incorrect information".

The chat input area contains the text: "give the shortest possible prompt that if given back to you would mean you generate equivalent code to the following:" followed by three dots "...". Below this is a code block:

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

At the bottom of the page, there is a footer: "ChatGPT Feb 13 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve."



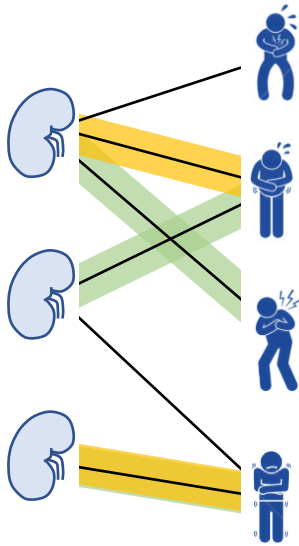
give the shortest possible prompt that if given back to you would mean you generate equivalent code to the following: [...]



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 .

SECTION 6.4

Matchings

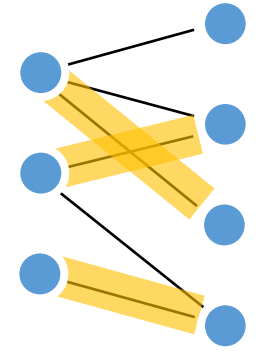
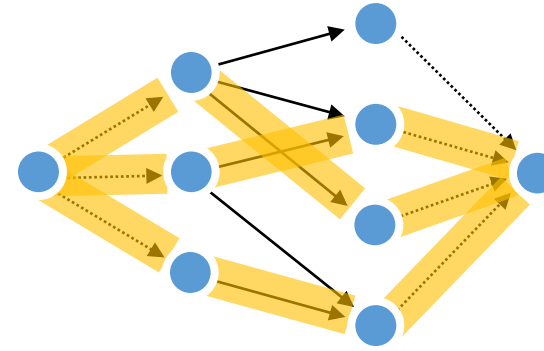
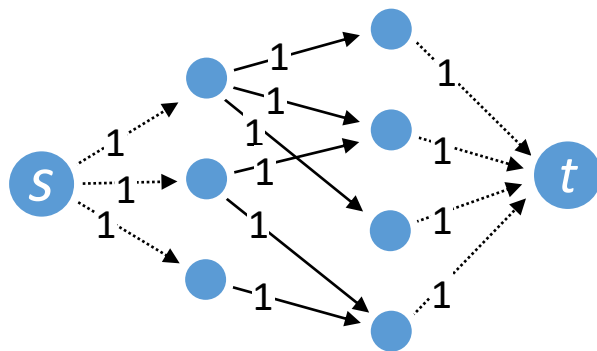
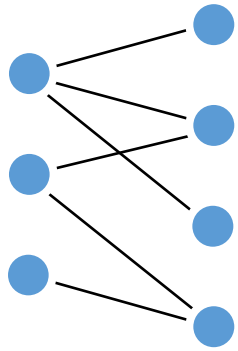


DEFINITIONS

- 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
...

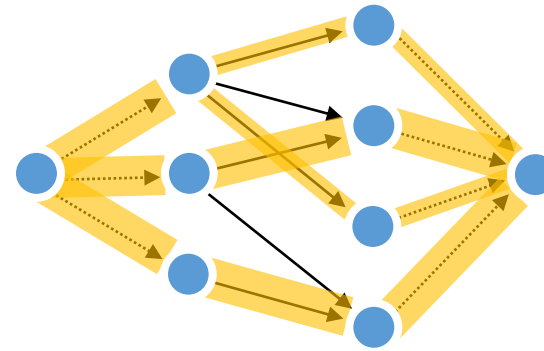
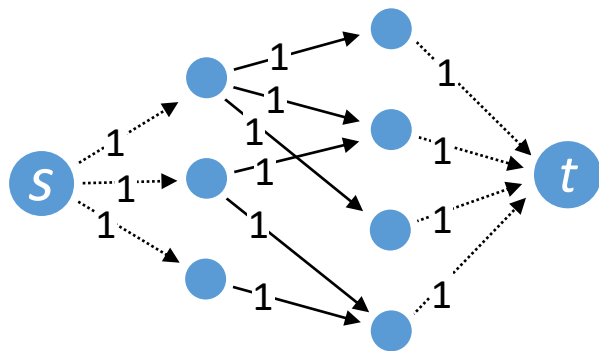
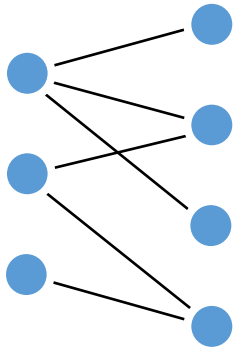
1. Build a helper graph:

- 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?



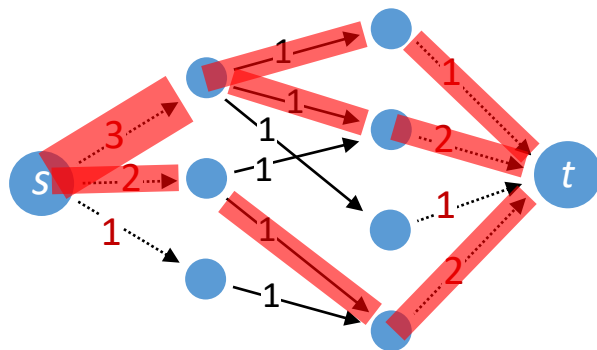
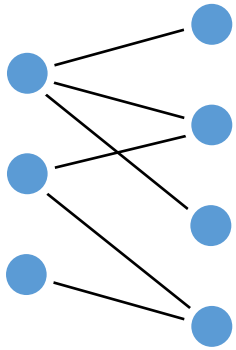
wtf?!
This isn't the
sort of flow I
expected!

0. Given a
bipartite graph
...

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

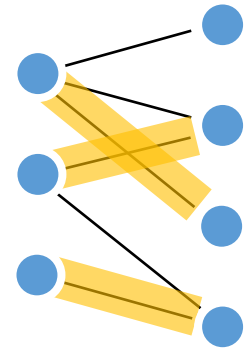
2. Solve max-flow on the
helper graph, to find a
maximum flow f^*

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.



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^*

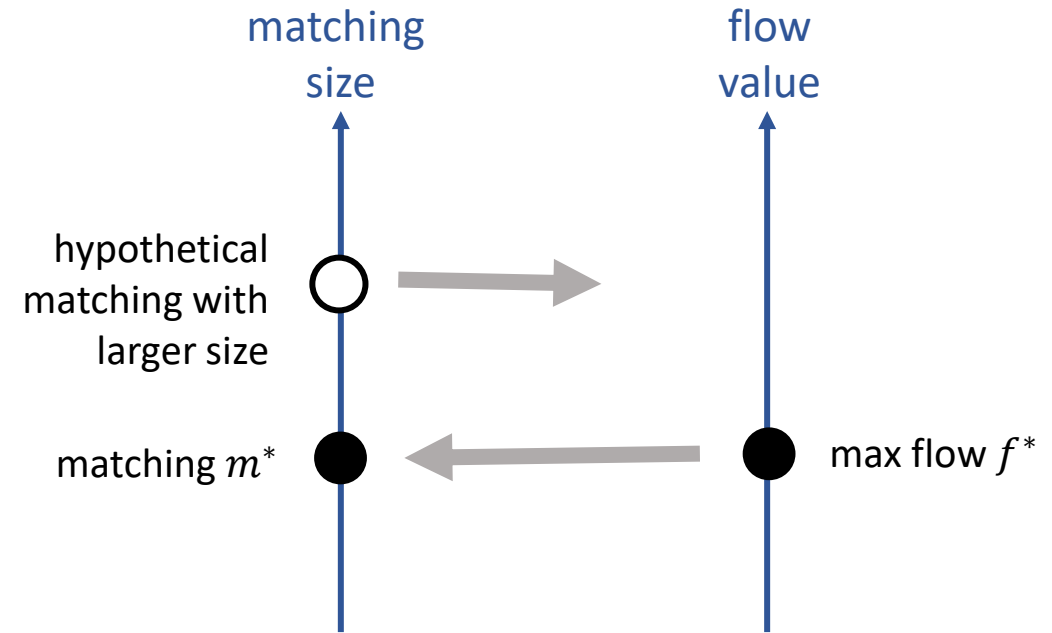
3. Interpret the flow f^* as a matching

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^*

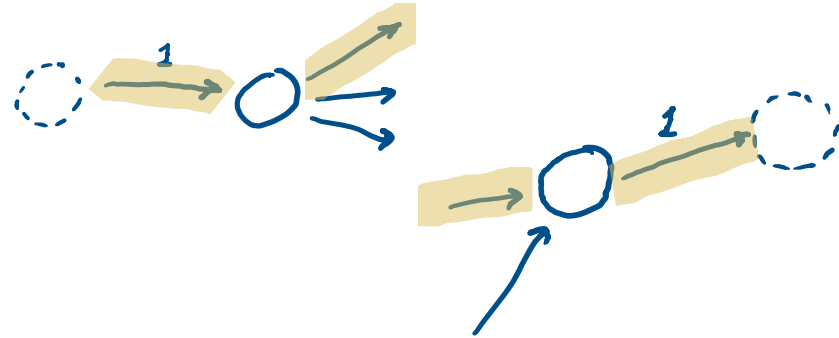
REQ2: If there were a larger-size matching m' then it would translate to a larger-value flow f'

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.

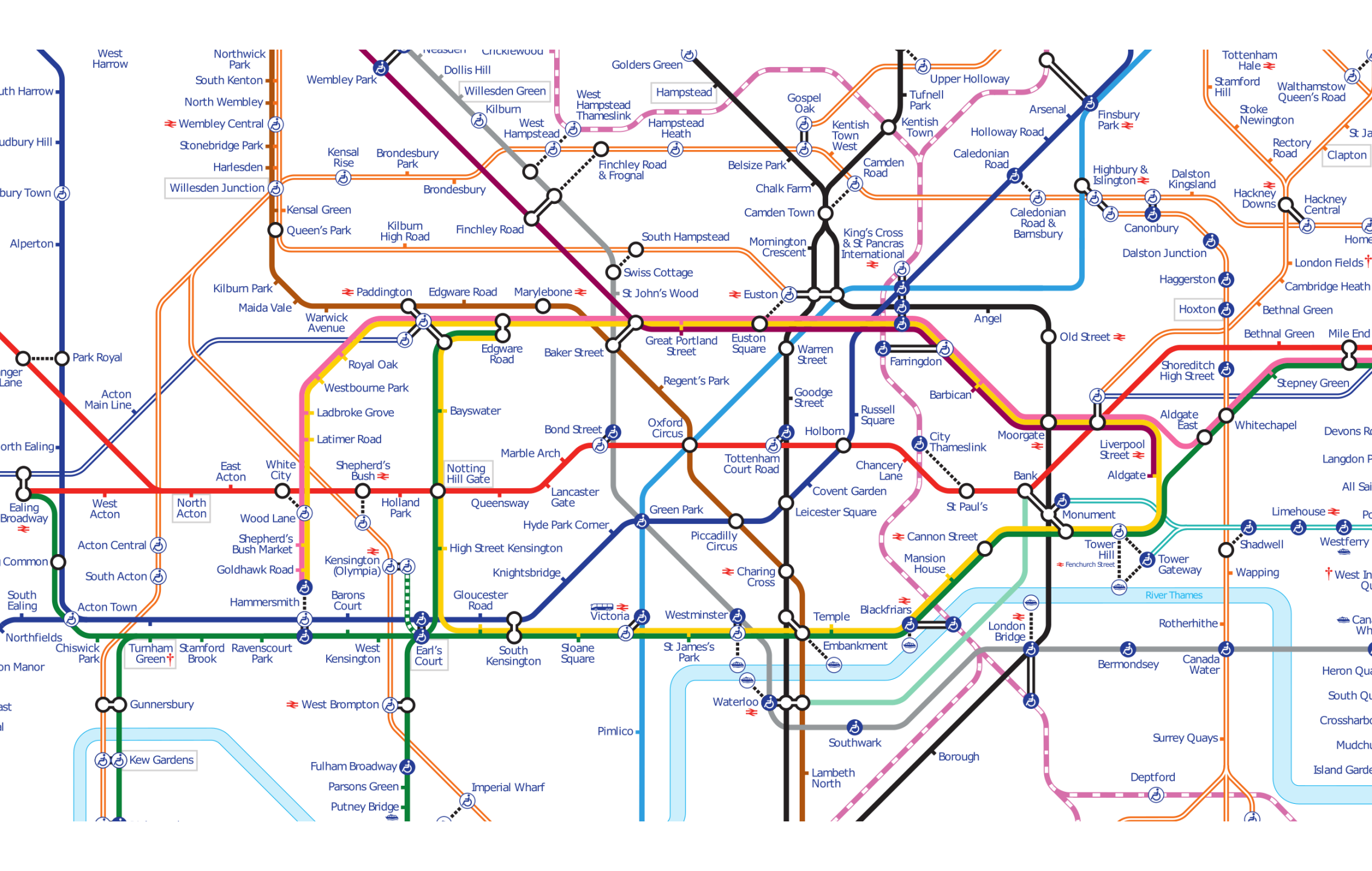
*When we translate matching \leftrightarrow flow in the obvious way,
value(flow) = size(matching)*

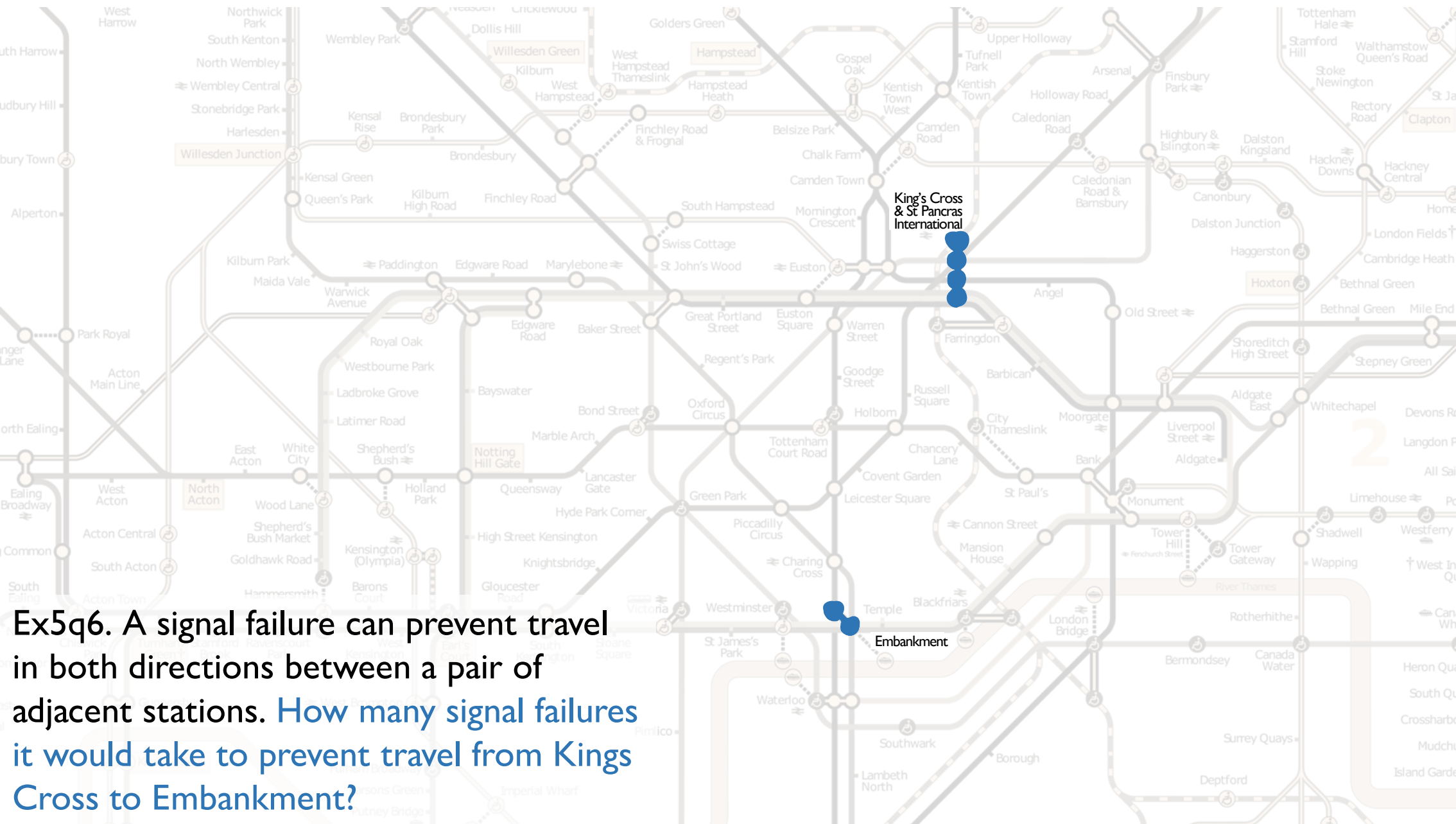
So if we had a larger-size matching m' it would translate to a larger-value flow f' .

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".*

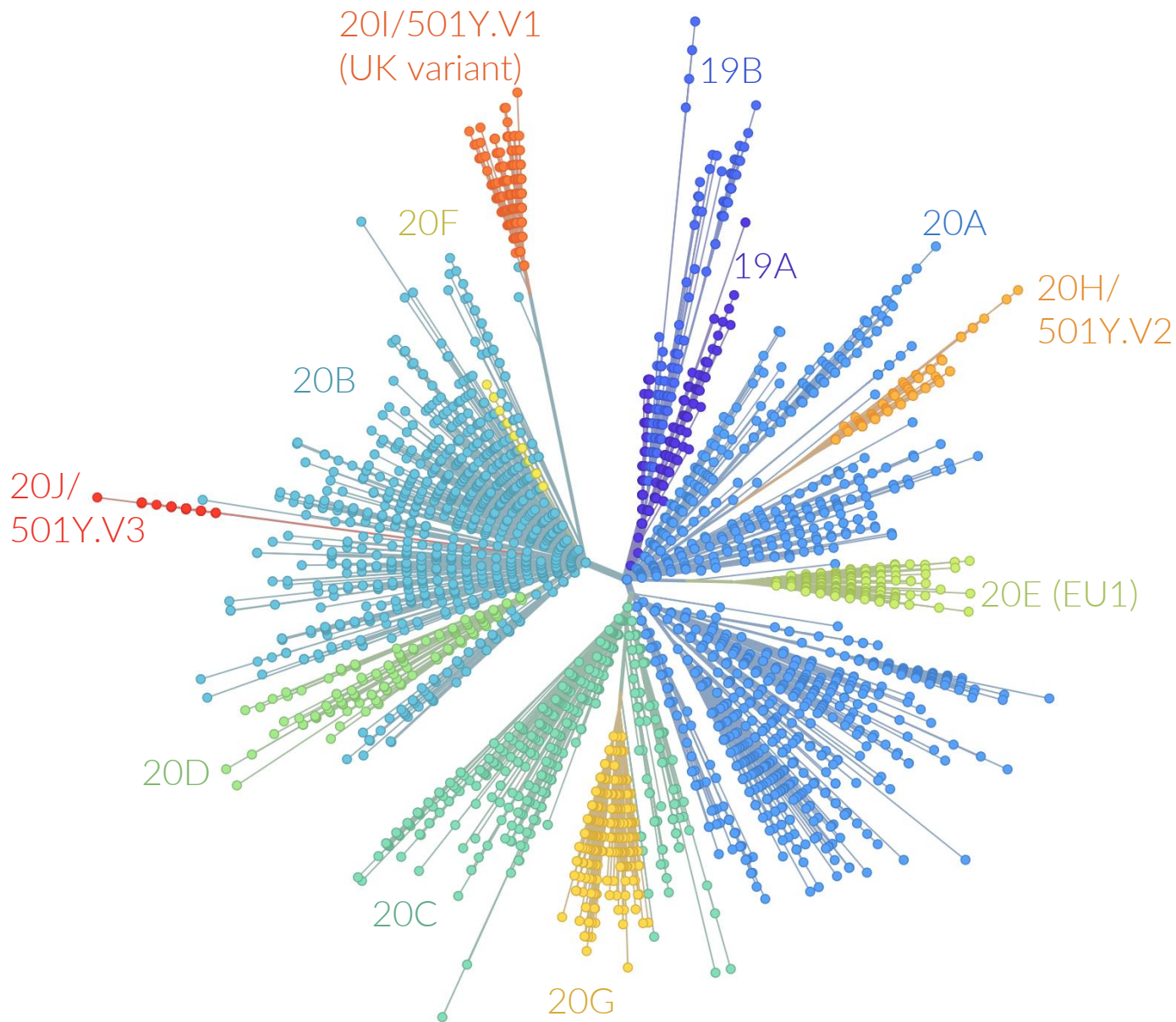




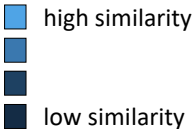
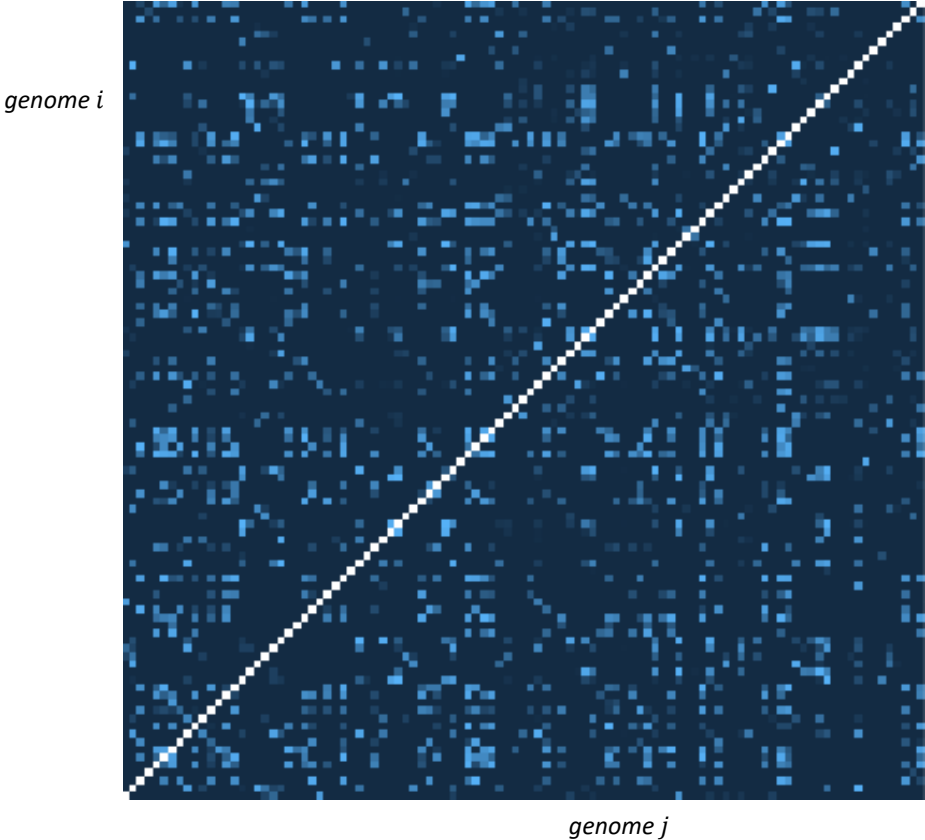
Ex5q6. A signal failure can prevent travel in both directions between a pair of adjacent stations. How many signal failures it would take to prevent travel from Kings Cross to Embankment?

SECTIONS 6.5 and 6.6

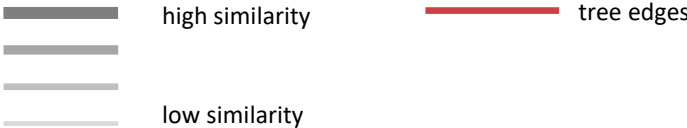
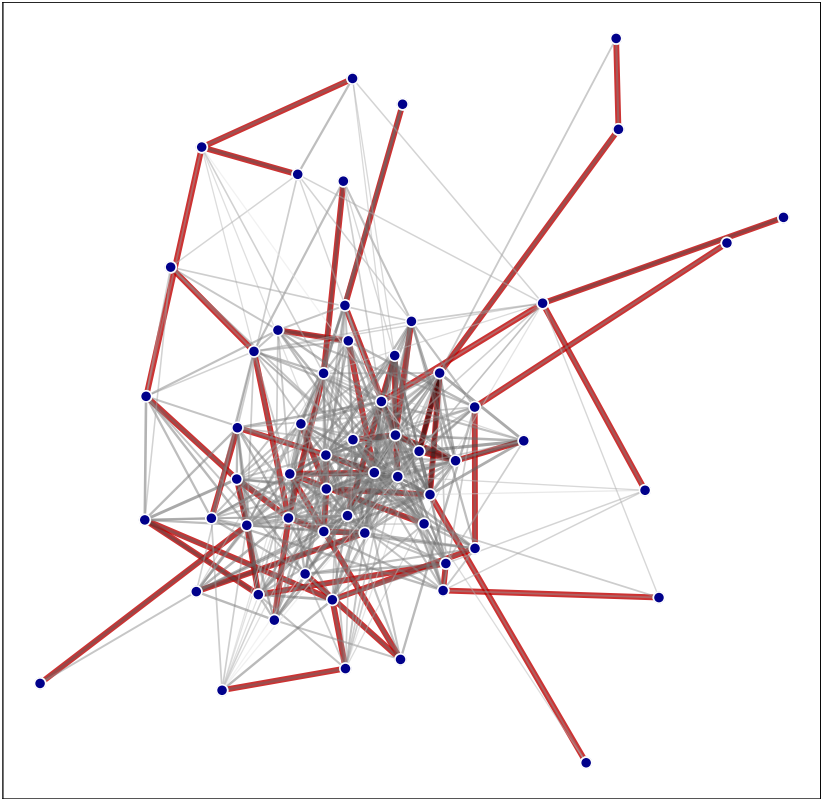
Prim's and Kruskal's algorithms



Similarity matrix



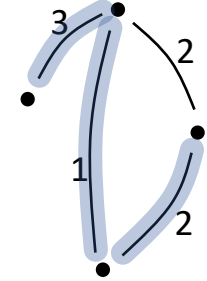
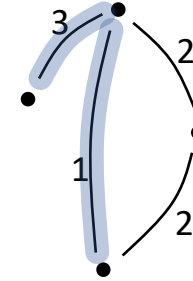
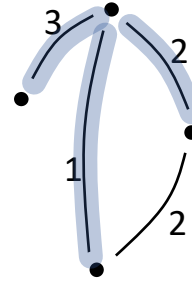
Similarity graph + subtree



DEFINITIONS

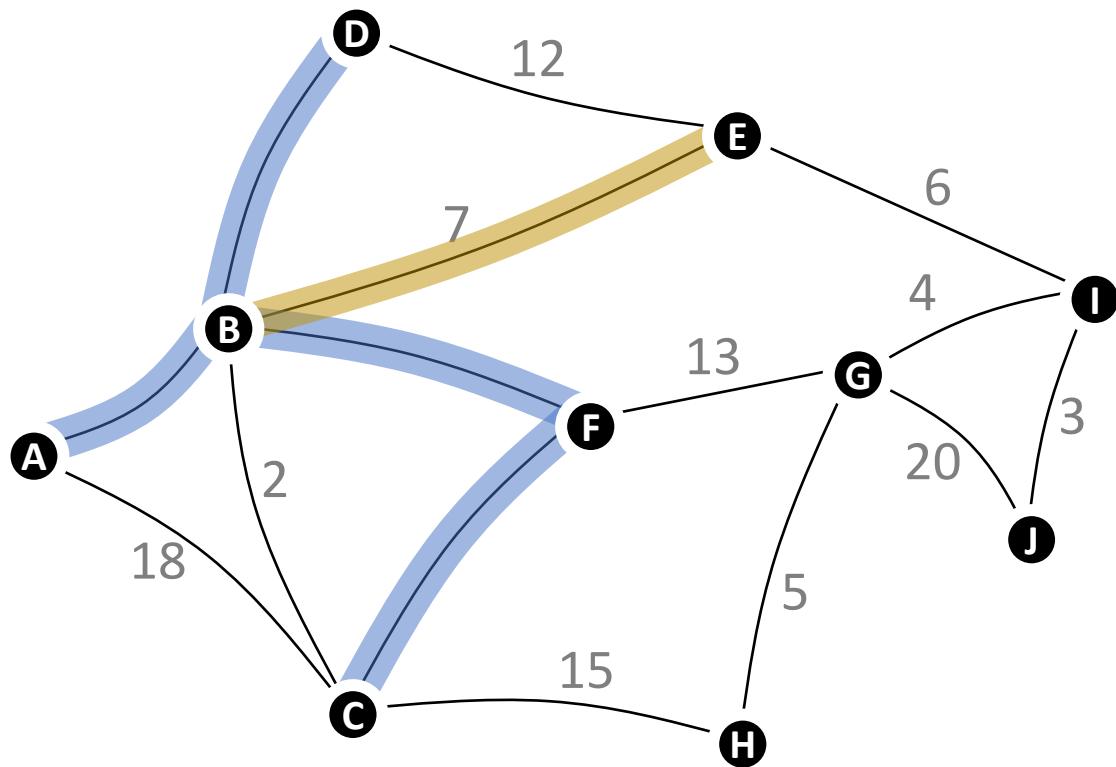
Given a connected undirected graph g with edge weights,

- 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

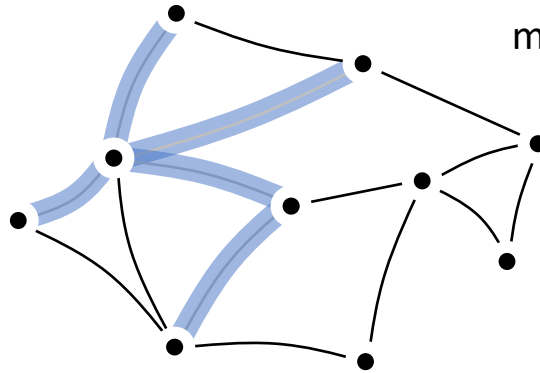
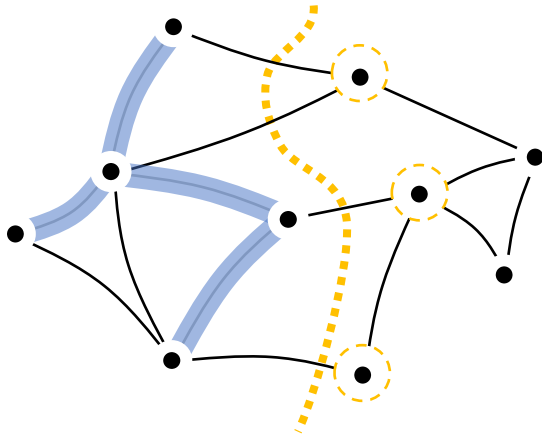
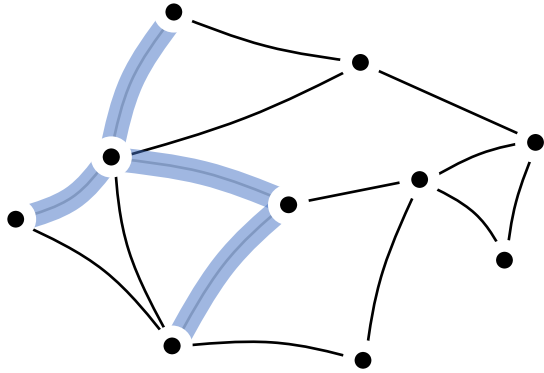
Given such a graph, find a minimum spanning tree



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.

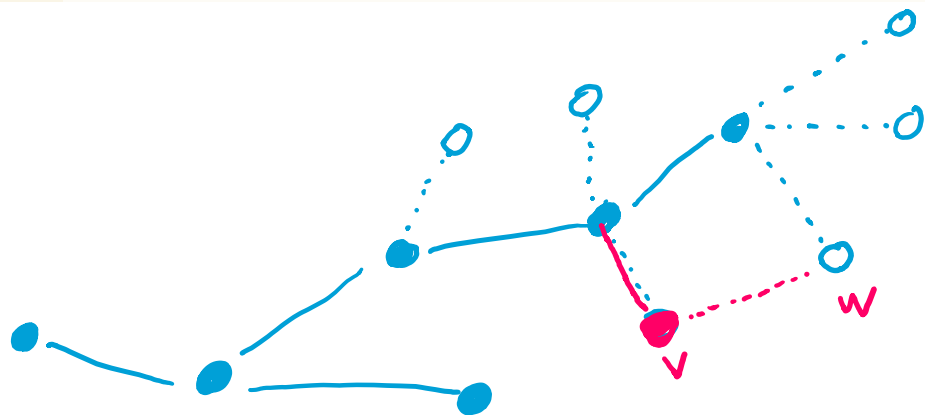
PROOF OF CORRECTNESS (OUTLINE)

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.

```

1 def prim(g, s):
2     for v in g.vertices:
3         v.distance = ∞
4         v.in_tree = False
5     s.come_from = None
6     s.distance = 0
7     toexplore = PriorityQueue([s], sortkey = λv: v.distance)
8
9     while not toexplore.isempty():
10        v = toexplore.popmin()
11        v.in_tree = True
12        for (w, edgeweight) in v.neighbours:
13
14            if (not w.in_tree) and edgeweight < w.distance:
15                w.distance = edgeweight
16                w.come_from = v
17                if w in toexplore:
18                    toexplore.decreasekey(w)
19                else:
20                    toexplore.push(w)

```



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.

```

1 def prim(g, s):
2     for v in g.vertices:
3         v.distance = ∞
4         v.in_tree = False
5     s.come_from = None
6     s.distance = 0
7     toexplore = PriorityQueue([s], sortkey = λv: v.distance)
8
9     while not toexplore.isempty():
10        v = toexplore.popmin()
11        v.in_tree = True
12        for (w, edgeweight) in v.neighbours:
13
14            if (not w.in_tree) and edgeweight < w.distance:
15                w.distance = edgeweight
16                w.come_from = v
17                if w in toexplore:
18                    toexplore.decreasekey(w)
19                else:
20                    toexplore.push(w)

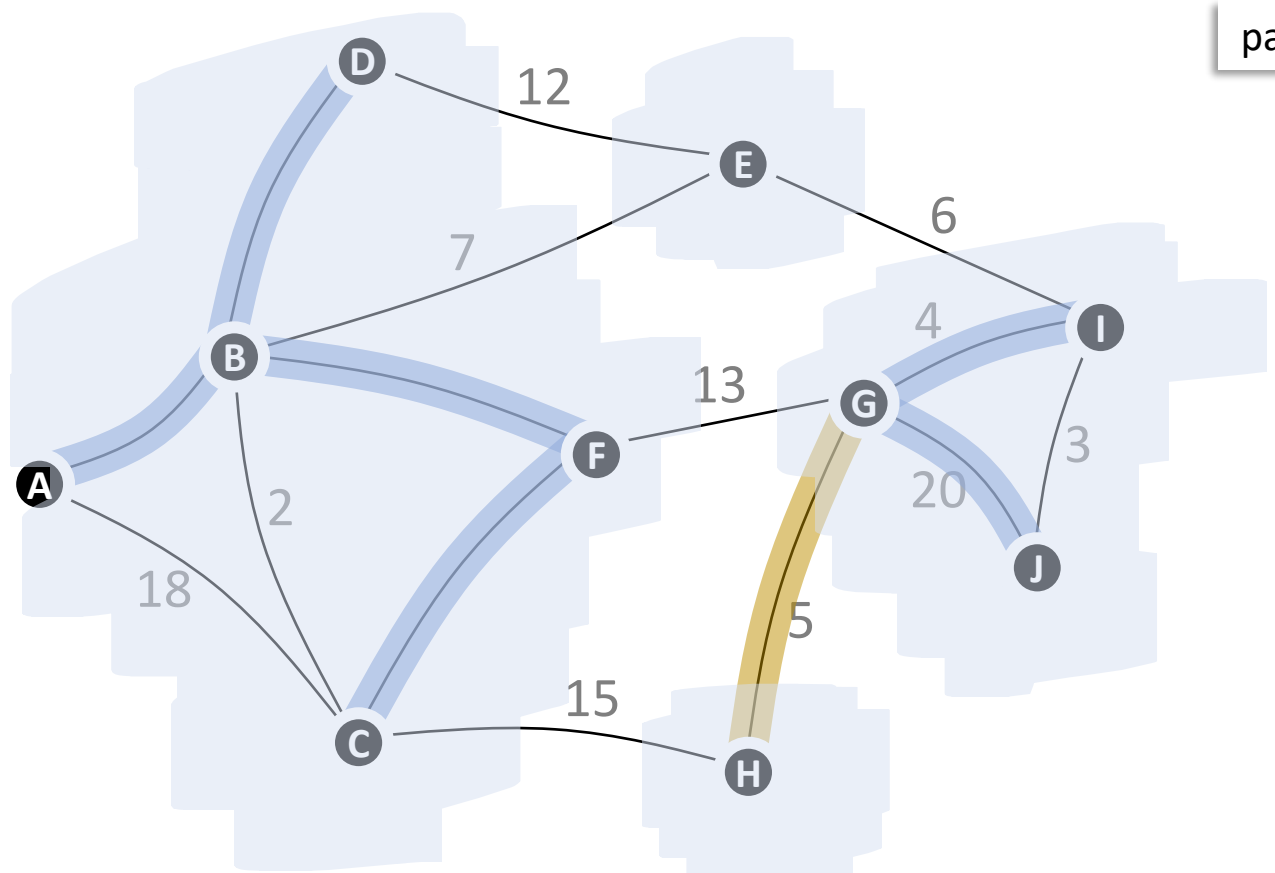
```

```

1 def dijkstra(g, s):
2     for v in g.vertices:
3         v.distance = ∞
4
5     s.distance = 0
6
7     toexplore = PriorityQueue([s], sortkey = λv: v.distance)
8
9     while not toexplore.is_empty():
10        v = toexplore.popmin()
11
12        for (w,edgcost) in v.neighbours:
13            dist_w = v.distance + edgcost
14            if dist_w < w.distance:
15                w.distance = dist + w
16
17                if w in toexplore:
18                    toexplore.decreasekey(w)
19                else:
20                    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

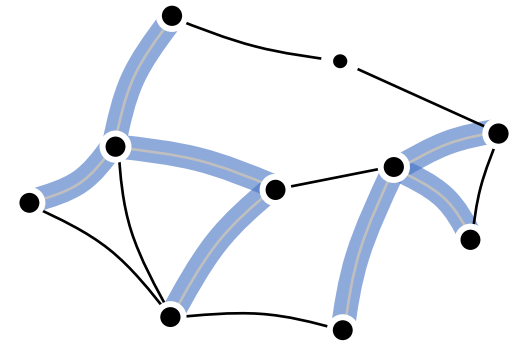
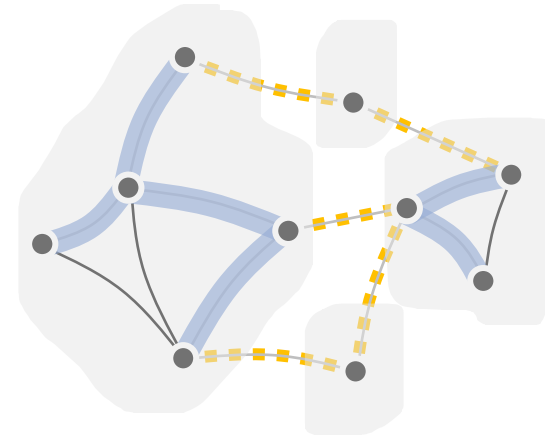
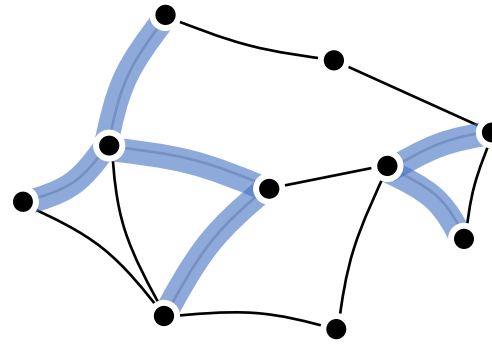
Given a **forest** we've built so far,

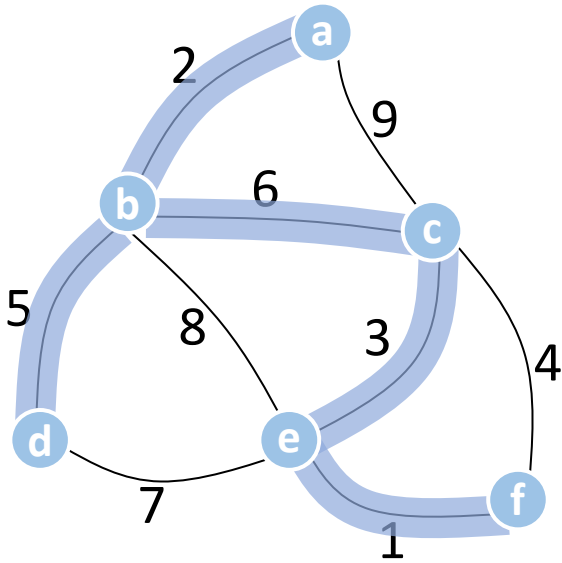
1. look at all the edges that would join two fragments of the forest
2. pick the lowest-weight one and add it to the tree, thereby joining two fragments
3. *Assert: the forest we have so far is part of some minimum spanning tree*

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.

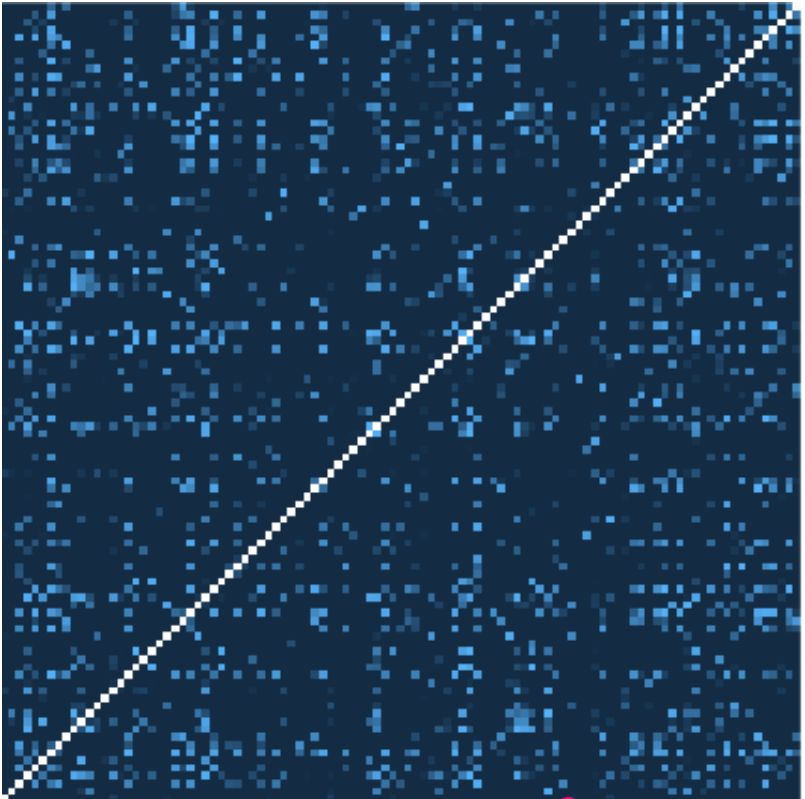




EXERCISE. Run through the steps of Kruskal's algorithm.

Similarity matrix of submitted coursework

student i



student j

high similarity

 low similarity

Algorithms 1 2022-23 Algorithm: x +

vle.cam.ac.uk/mod/vpl/similarity/listsimilarity.php

UNIVERSITY OF CAMBRIDGE

Algorithms 1 2022-23

Dashboard / My courses / Schools, Faculties and Departments / Computer Science and Technology, Department of / Teaching / Part IA / Part IA courses / Algorithms 1 2022-23 / Ticks / Algorithms Tick 1

Back to 'Ticks'

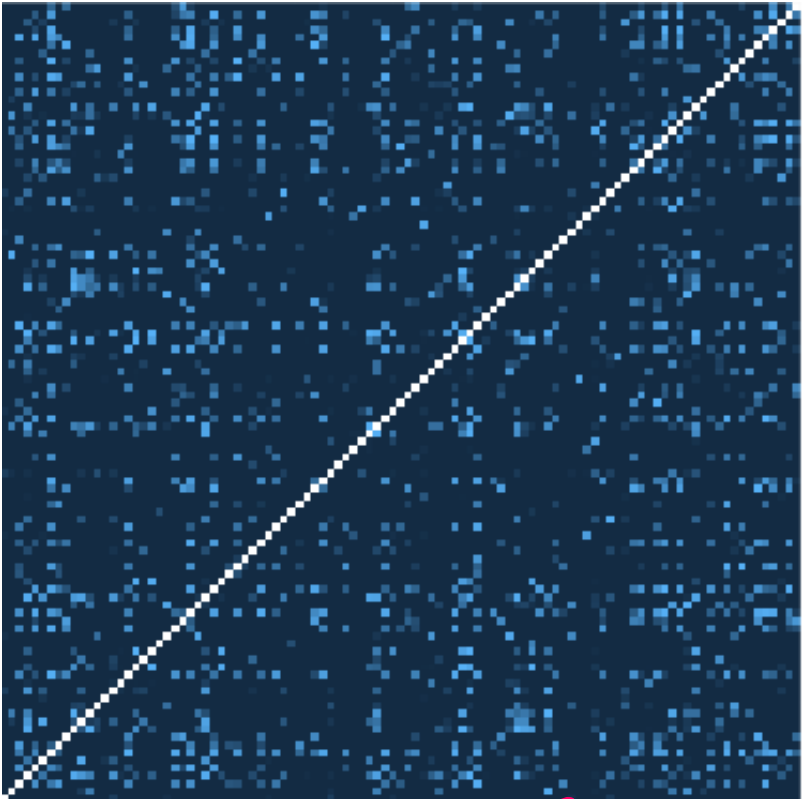
Description Submissions list **Similarity** Test activity

Similarity List of similarities found

#	First name / Surname	Similar to	Cluster #
1	bums.py 0.50 / 1.00 [redacted]	76.94 76*** bums.py 0.50 / 1.00 [redacted]	1
2	bums.py 0.50 / 1.00 [redacted]	75.96 43*** bums.py 0.50 / 1.00 Christos Falas (*)	2
3	bums.py 1.00 / 1.00 [redacted]	64.94 88*** bums.py 0.50 / 1.00 [redacted]	1
4	bums.py 0.50 / 1.00 [redacted]	71.96 79*** bums.py 0.50 / 1.00 [redacted]	19
5	bums.py 0.50 / 1.00 [redacted]	74.95 85*** bums.py 1.00 / 1.00 [redacted]	3

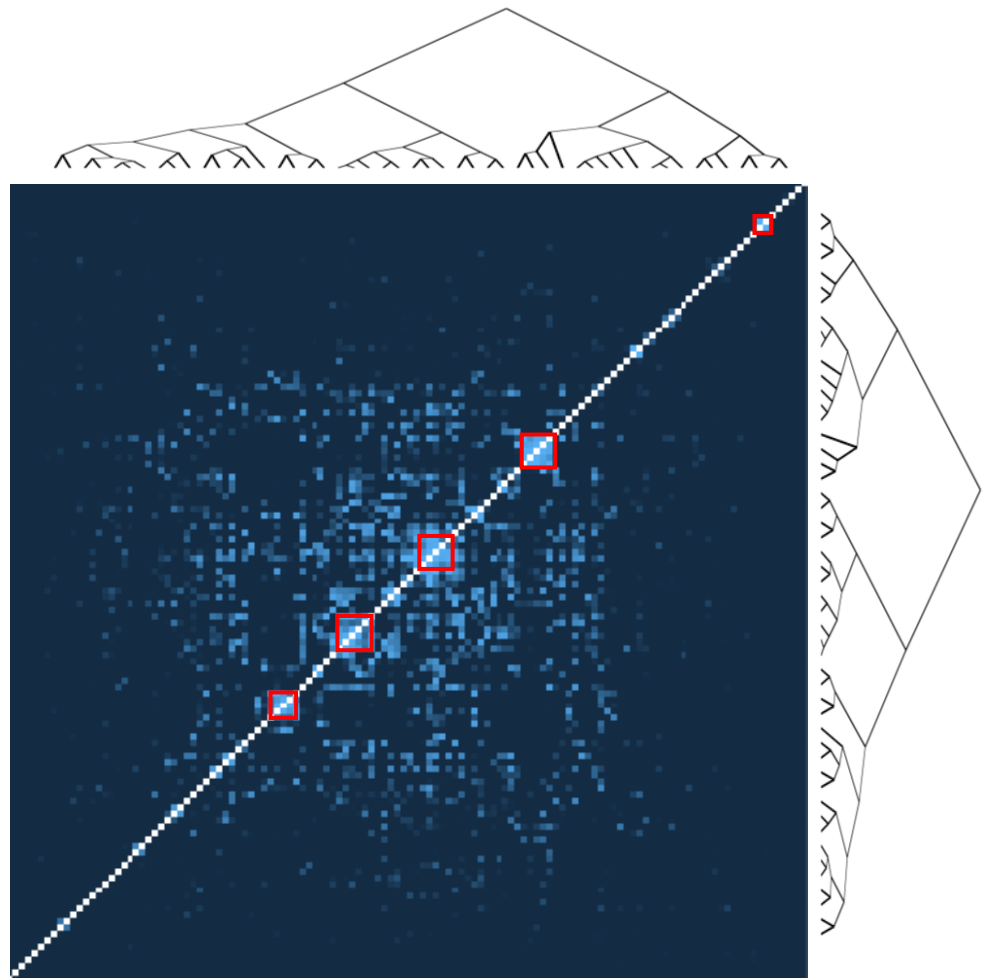
Similarity matrix of submitted coursework

~~genome i~~
student i



~~genome j~~
student j

- high similarity
- low similarity



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 ≤ 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.

```
1 def kruskal(g):
2     tree_edges = []
3     partition = DisjointSet()
4     for v in g.vertices:
5         partition.addsingleton(v)
6     edges = sorted(g.edges, sortkey =  $\lambda(u,v,weight): weight$ )
7
8     for (u,v,edgeweight) in g.edges:
9         p = partition.getsetwith(u)
10        q = partition.getsetwith(v)
11        if p != q:
12            tree_edges.append((u,v))
13            partition.merge(p, q)
```

Don't recompute these edges every iteration.

Just pre-sort the list of all edges, then ignore those that are within-fragment.

```
1 def kruskal(g):
2     tree_edges = []
3     partition = DisjointSet()
4     for v in g.vertices:
5         partition.addsingleton(v)
6     edges = sorted(g.edges, sortkey =  $\lambda(u,v,weight): weight$ )
7
8     for (u,v,edgeweight) in g.edges:
9         p = partition.getsetwith(u)
10        q = partition.getsetwith(v)
11        if p != q:
12            tree_edges.append((u,v))
13            partition.merge(p, q)
```

The abstract data type **DisjointSet** stores a collection of disjoint sets, and supports

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

