A phylogenetic tree is a tree representation of the evolutionary history of a set of gene sequences e.g. COVID variants.
SECTIONS 6.5 and 6.6

Prim’s and Kruskal’s algorithms
Given the similarity score between every pair of genomes, can we reconstruct a likely phylogenetic tree? In other words, can we find a high-similarity tree embedded in the similarity graph?
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
SIMPLE GREEDY ALGORITHM: Which edge would you add next, to grow the tree?

Let’s build up a tree, edge by edge.
PRIM’S ALGORITHM

Choose an arbitrary start vertex as our initial tree. Then, 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

Choose an arbitrary start vertex as our initial tree. Then, 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,
2. And at the cut between our tree and those vertices
3. Pick the lowest-weight edge across this cut,
4. And add it to the tree
5. Assert: the tree we have so far is part of some minimum spanning tree

Repeat until we have a spanning tree.

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)

    return g

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)

    return g

Cost \( O(E + V \log V) \)
because it's basically the same as Dijkstra.
Alternatively ...
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 forest, 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

<table>
<thead>
<tr>
<th>Student i</th>
<th>Student j</th>
</tr>
</thead>
<tbody>
<tr>
<td>genome i</td>
<td>genome j</td>
</tr>
</tbody>
</table>

- high similarity
- moderate similarity
- low similarity

Cluster

<table>
<thead>
<tr>
<th>#</th>
<th>First name / Last name</th>
<th>Similar to</th>
<th>Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>(*)</td>
</tr>
<tr>
<td>4</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>resalloc.py 1.00 / 1.00</td>
<td>3</td>
</tr>
</tbody>
</table>
Similarity matrix of submitted coursework

- genome i
- genome j

- Student i
- Student j

- high similarity
- moderate similarity
- low similarity
**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.

```python
def kruskal(g):
    tree_edges = []
    partition = DisjointSet()
    for v in g.vertices:
        partition.addsingleton(v)
    edges = sorted(g.edges, sortkey = λ(u,v,weight): weight)
    for (u,v,edgeweight) in g.edges:
        p = partition.getsetwith(u)
        q = partition.getsetwith(v)
        if p != q:
            tree_edges.append((u,v))
            partition.merge(p, q)
```

Don’t recompute these edges every iteration.

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

- \(\mathcal{O}(1)\) \* addsingleton(v)
- \(\mathcal{O}(1)\) \* p = getsetwith(v)
- \(\mathcal{O}(1)\) \* merge(p,q)
QUESTION. How might we segment this image into “handsome stoat” and “background”?

dynamic programming
greedy algorithms
translation strategy

depth-first search
breadth-first search
Dijkstra’s algorithm
Bellman-Ford algorithm
Johnson’s algorithm

Ford-Fulkerson algorithm
matchings
topological sort
Prim, Kruskal
1. define a grid
2. measure dissimilarity along edges
3. run Kruskal’s algorithm, and stop when the forest it’s building has just a few trees
1. define a grid
Alternatively ...

2. measure dissimilarity along edges
Alternatively...

3. ask the user to label some “stoat” points and some “background” points
4. set up a flow network
Alternatively...

5. find a minimum-capacity cut