nextstrain.org 20 February 2021

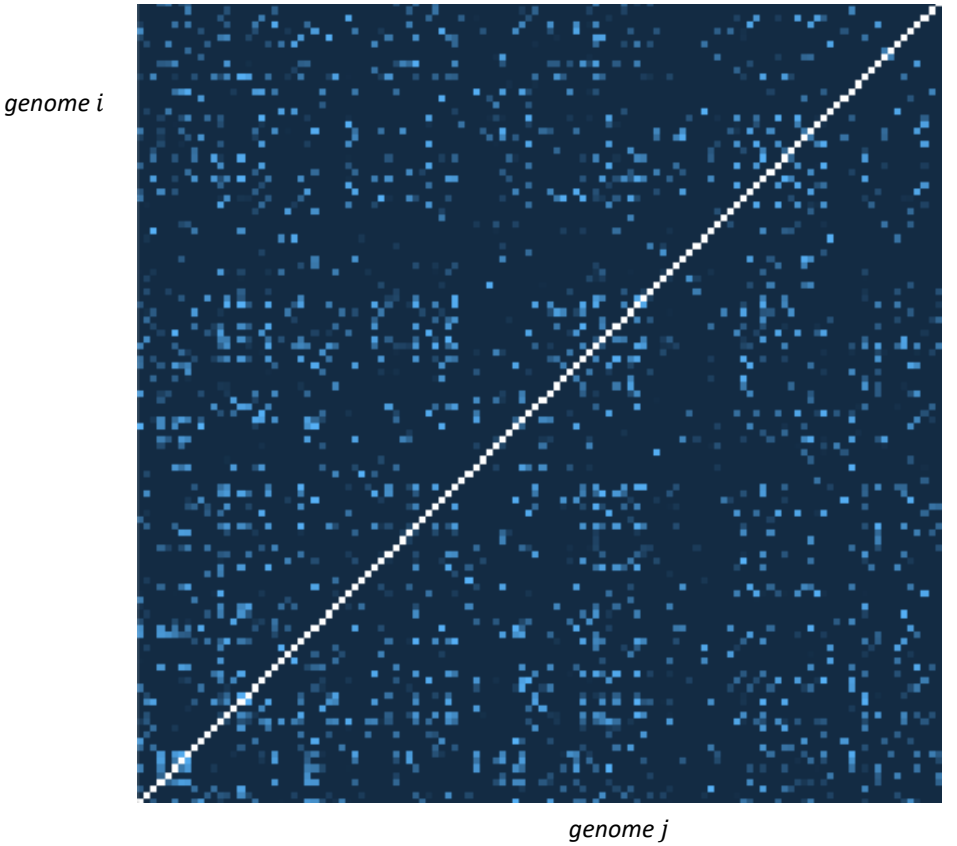
A phylogenetic tree *is a tree representation of the evolutionary history of a set of gene sequences e.g. COVID variants.*

SECTIONS 6.5 and 6.6

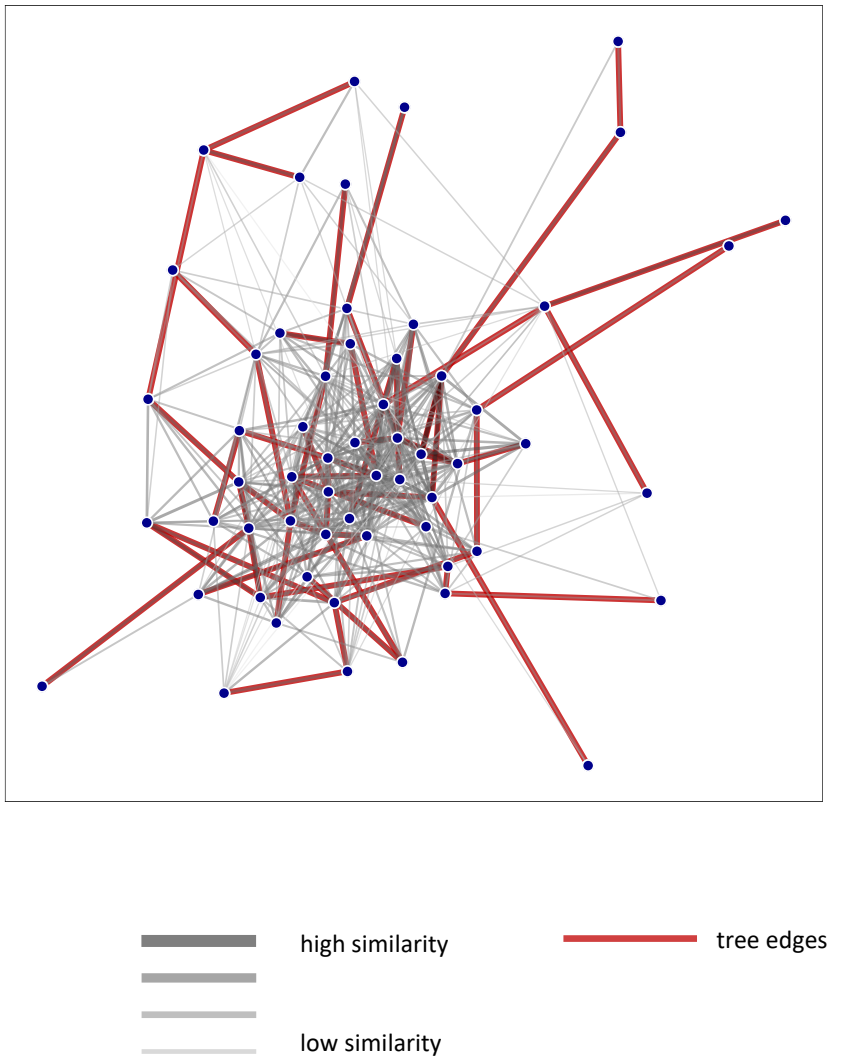
Prim’s and Kruskal’s algorithms

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?

Similarity matrix



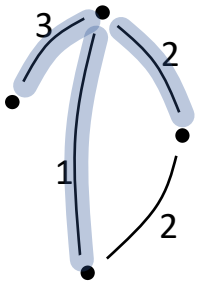
Similarity graph + subtree



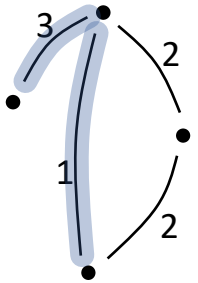
DEFINITIONS

Given a connected undirected graph g with edge weights,

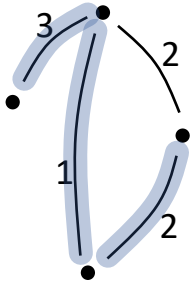
- A **spanning tree** of g is a tree that connects all of g's vertices, using some or all of g's edges
- The **weight** of a spanning tree is the sum of all its edge weights
- A **minimum spanning tree (MST)** is a spanning tree that has minimum weight among all spanning trees



spanning tree, weight 6



not a spanning tree



spanning tree, weight 6

PROBLEM STATEMENT

Given such a graph, find a minimum spanning tree

dynamic programming

greedy algorithms

translation strategy

depth-first search

breadth-first search

Dijkstra's algorithm

Bellman-Ford algorithm

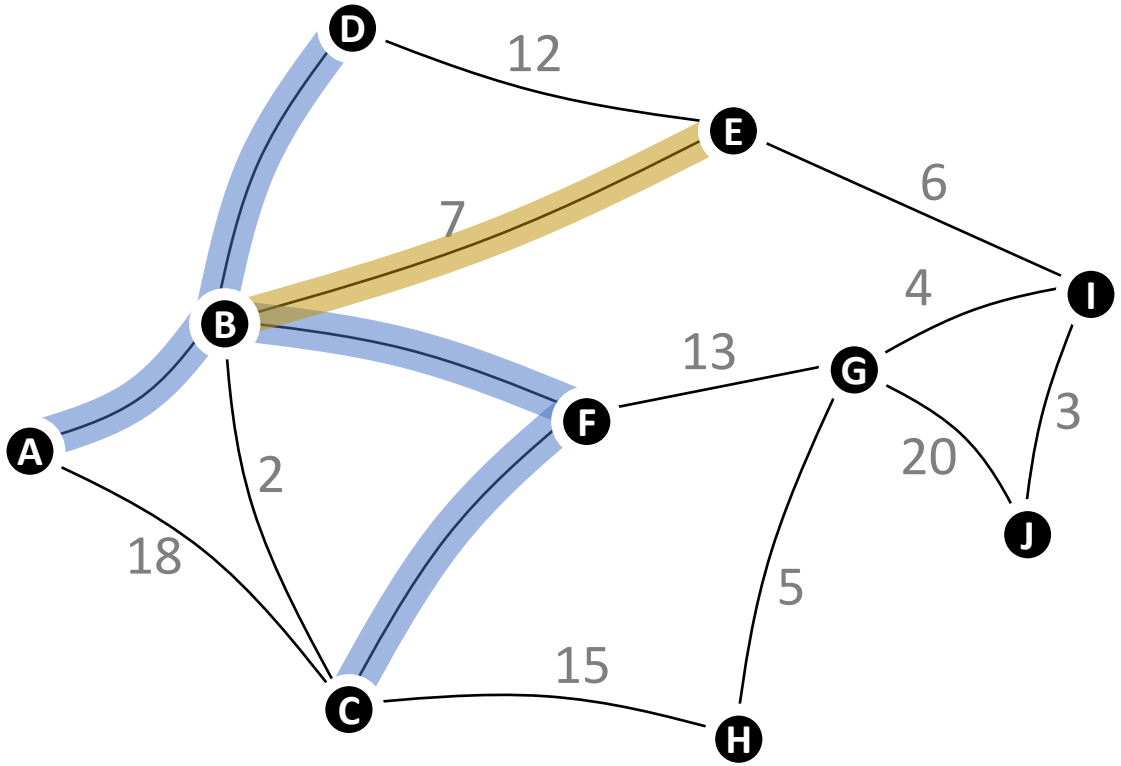
Johnson's algorithm

Ford-Fulkerson algorithm

matchings

topological sort

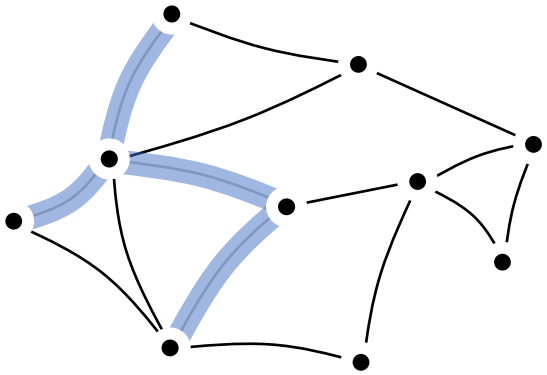
minimum spanning tree

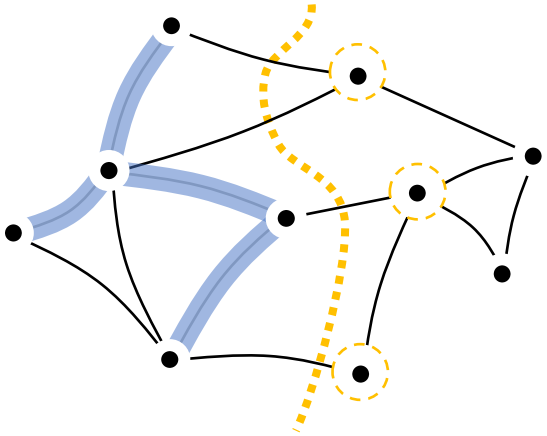


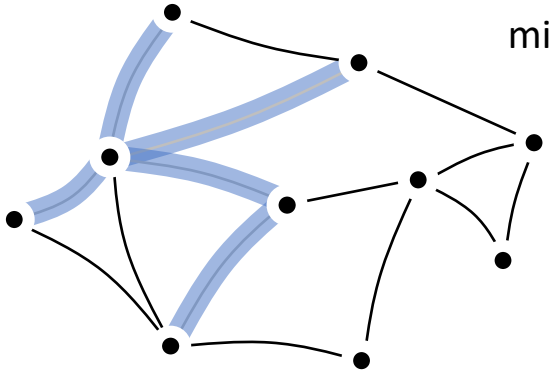
Let's build up a tree, edge by edge.

SIMPLE GREEDY ALGORITHM:

Which edge would you add next, to grow the tree?







PRIM'S ALGORITHM

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.

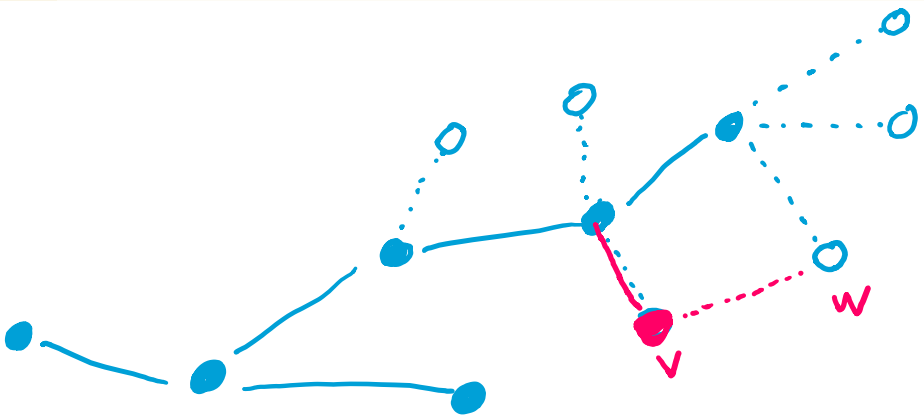
PROOF OF CORRECTNESS (OUTLINE)

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.

```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)
```

distance from tree to v

am I in the tree yet?



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.

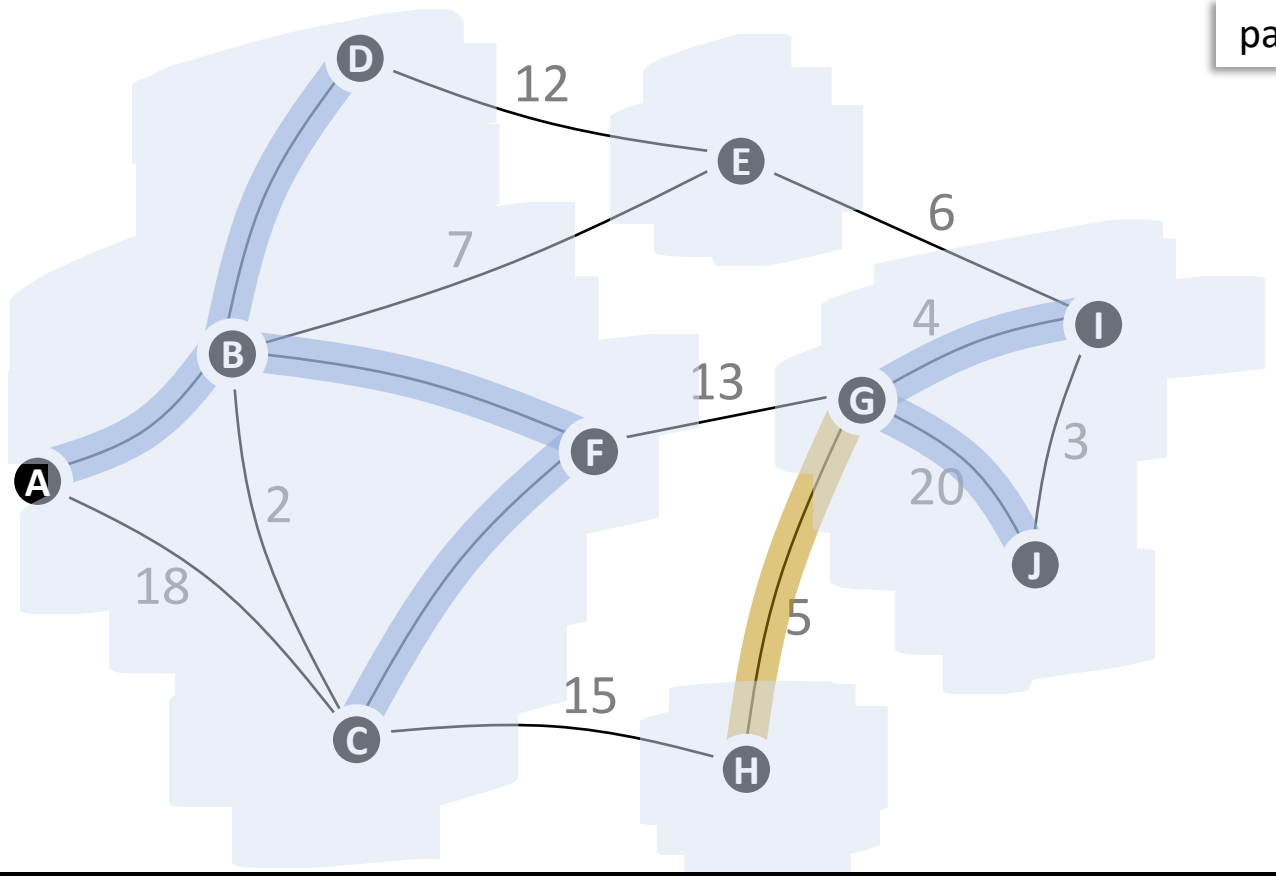
```python
def prim(g, s):
    for v in g.vertices:
        v.distance = ∞
        v.in_tree = False
    s.come_from = None
    s.distance = 0
    toexplore = PriorityQueue([s], sortkey = λv: v.distance)

    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)
```

Cost $O(E + V \log V)$

because it's basically the same as Dijkstra.

```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)
```

Alternatively ...
Let's build up a **forest**, edge by edge.



SIMPLE GREEDY ALGORITHM:

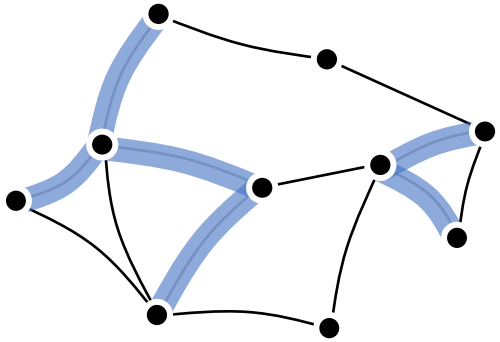
Which edge would you add next, to grow the forest?

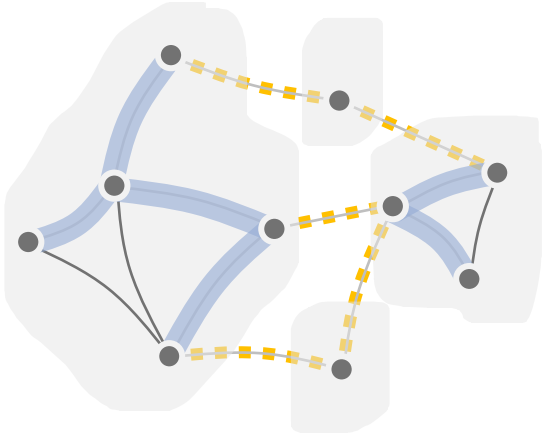
KRUSKAL'S ALGORITHM

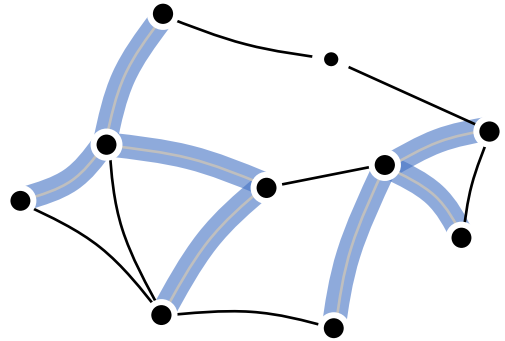
Given a **forest** we've built so far,

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.

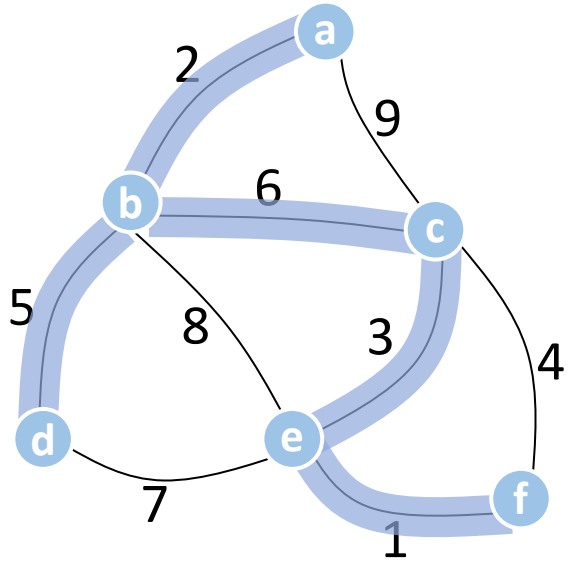


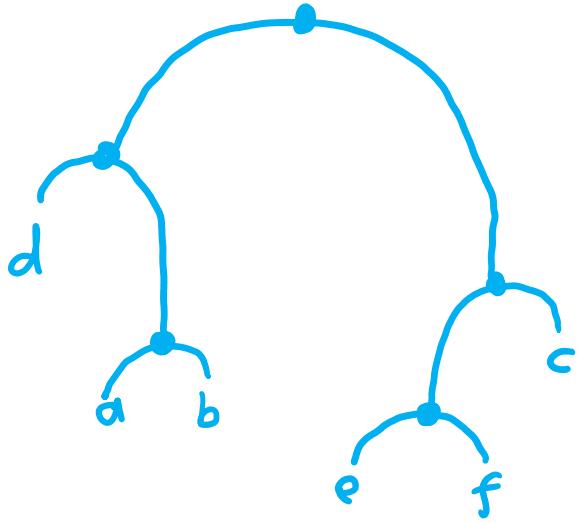




PROOF OF CORRECTNESS (OUTLINE)

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.

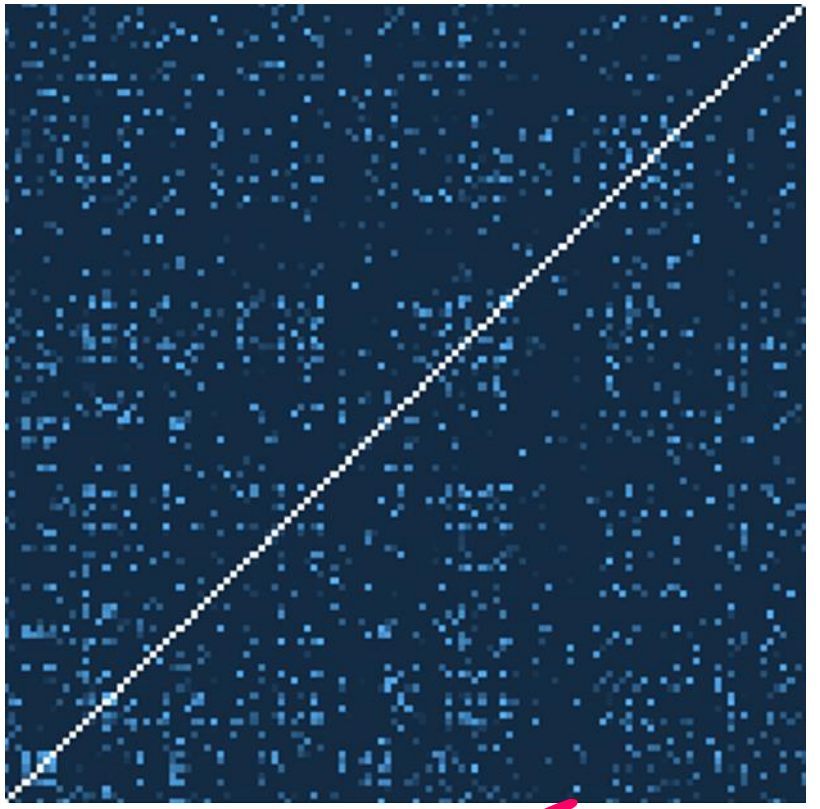




Kruskal's algorithm in effect builds a classification tree*; vertices connected by low-weight edges become nearby leaves of the tree.*

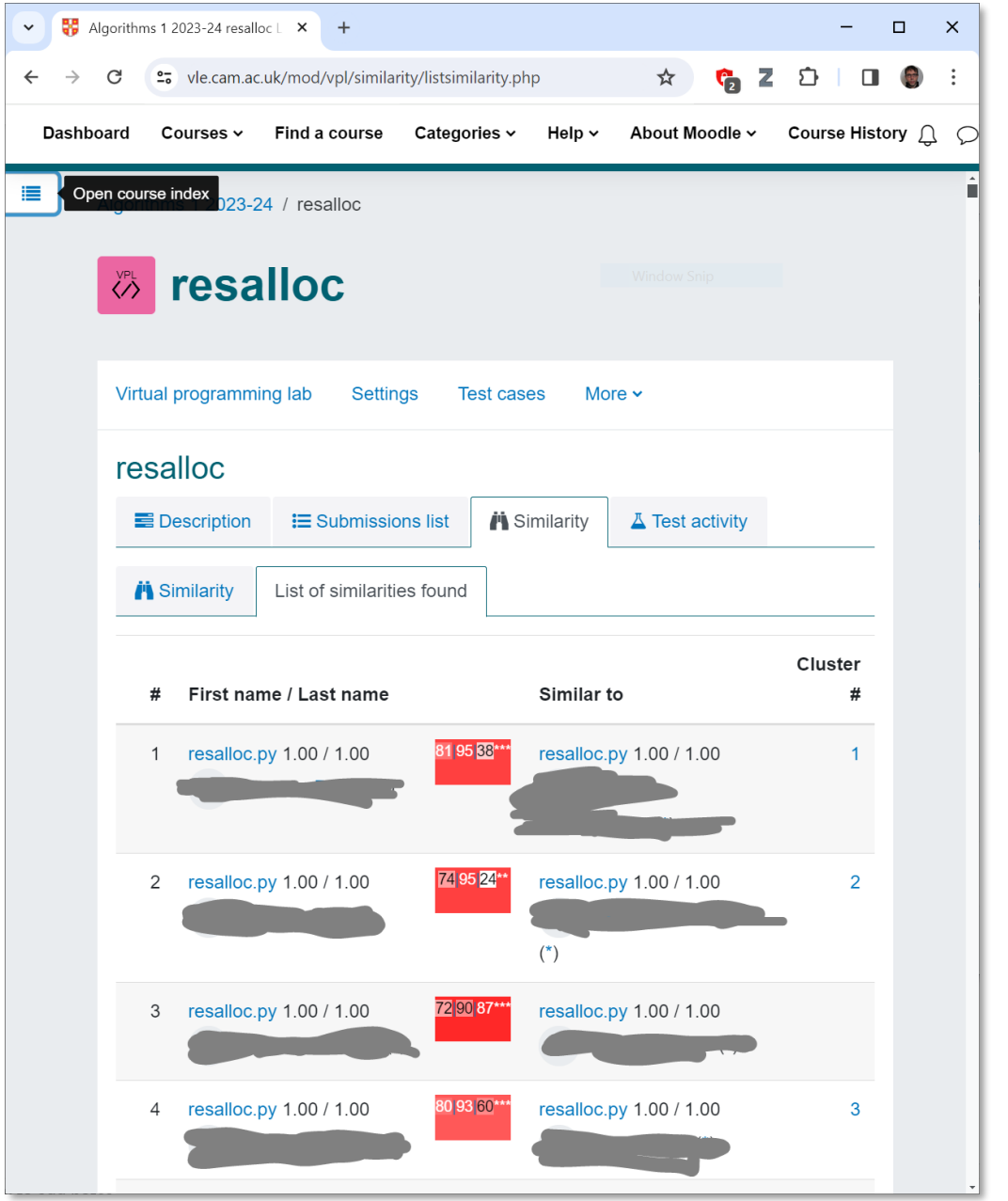
EXERCISE. Run through the steps of Kruskal's algorithm.

Similarity matrix of submitted coursework

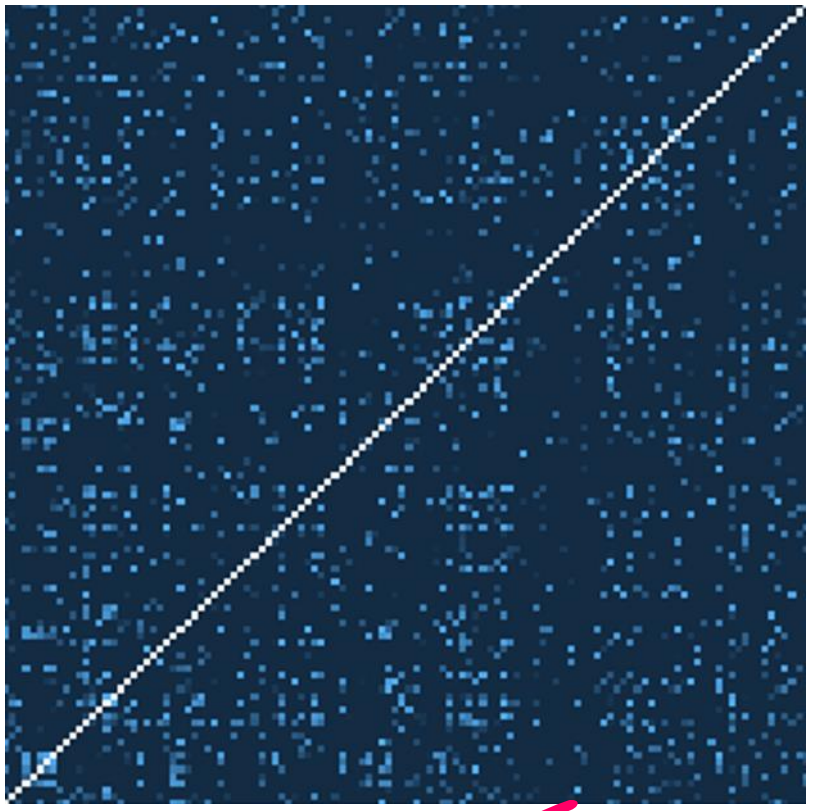


high similarity

low similarity

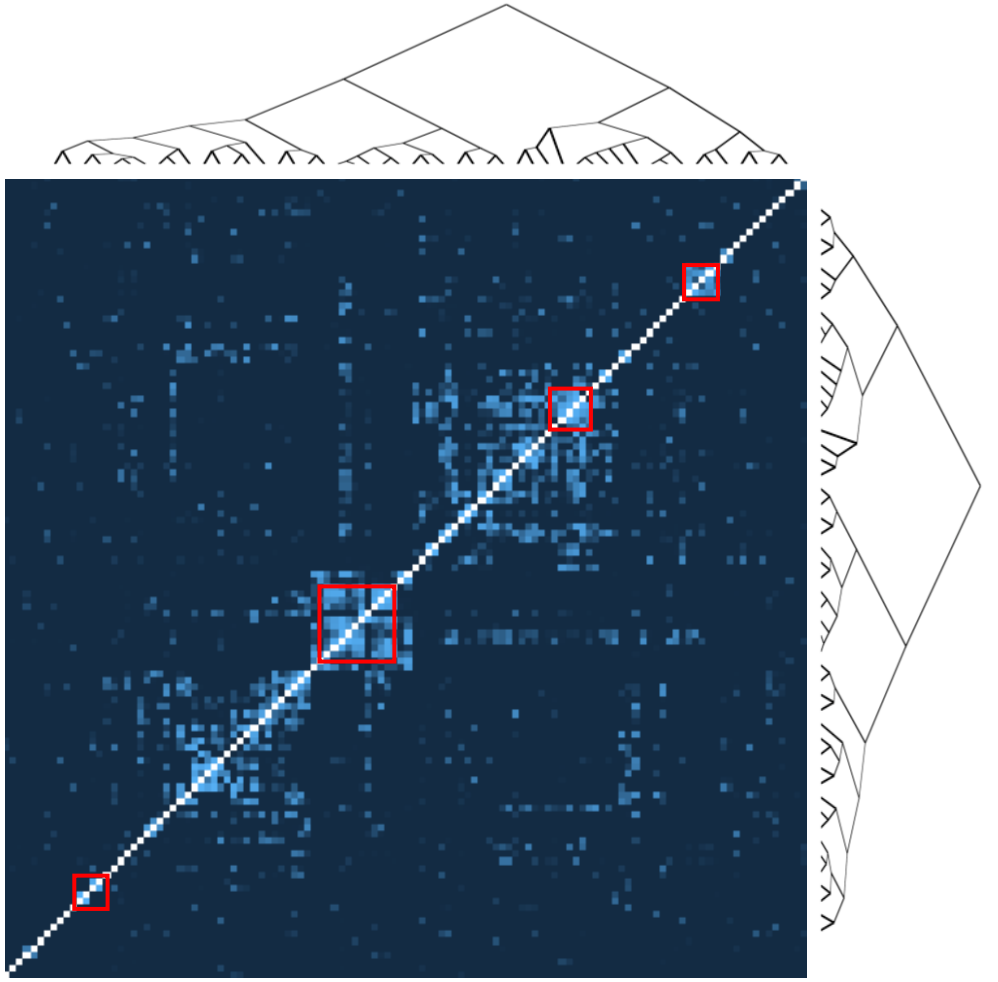


Similarity matrix of submitted coursework



high 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.

```
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 = λ(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 iterate through and ignore those that are within-fragment.

Total cost

$O(V + E + E \log E)$

$= O(V + E \log E)$

We're assuming a connected graph

$\Rightarrow E \geq V-1 \Rightarrow V \leq E+1$

The graph can't have more than $\frac{1}{2}V(V-1)$ edges

$\Rightarrow E \leq \frac{1}{2}V(V-1) \Rightarrow \log E \leq 2 \log V$

So total cost is $O(E \log V)$.

```
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 = λ(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)
```

O(V)

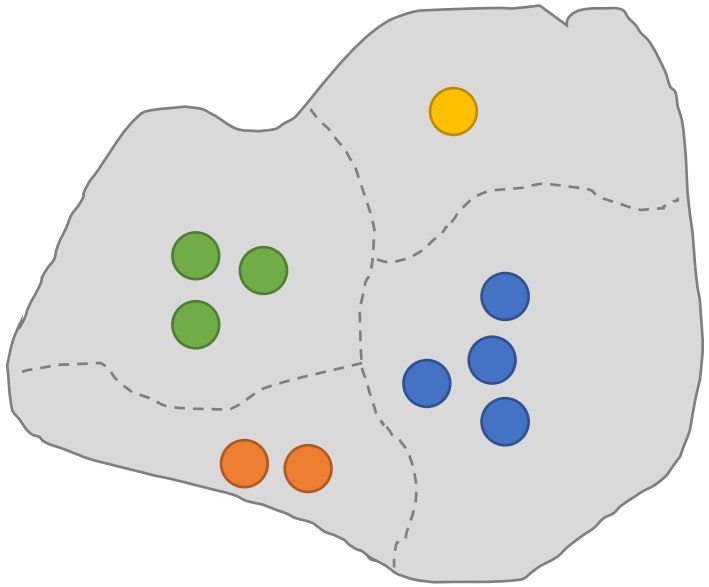
O(E log E)

O(E) iterations

O(1) ish operations

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

- O(1) ish addsingleton(v)
- O(1) ish p = getsetwith(v)
- O(1) ish merge(p,q)

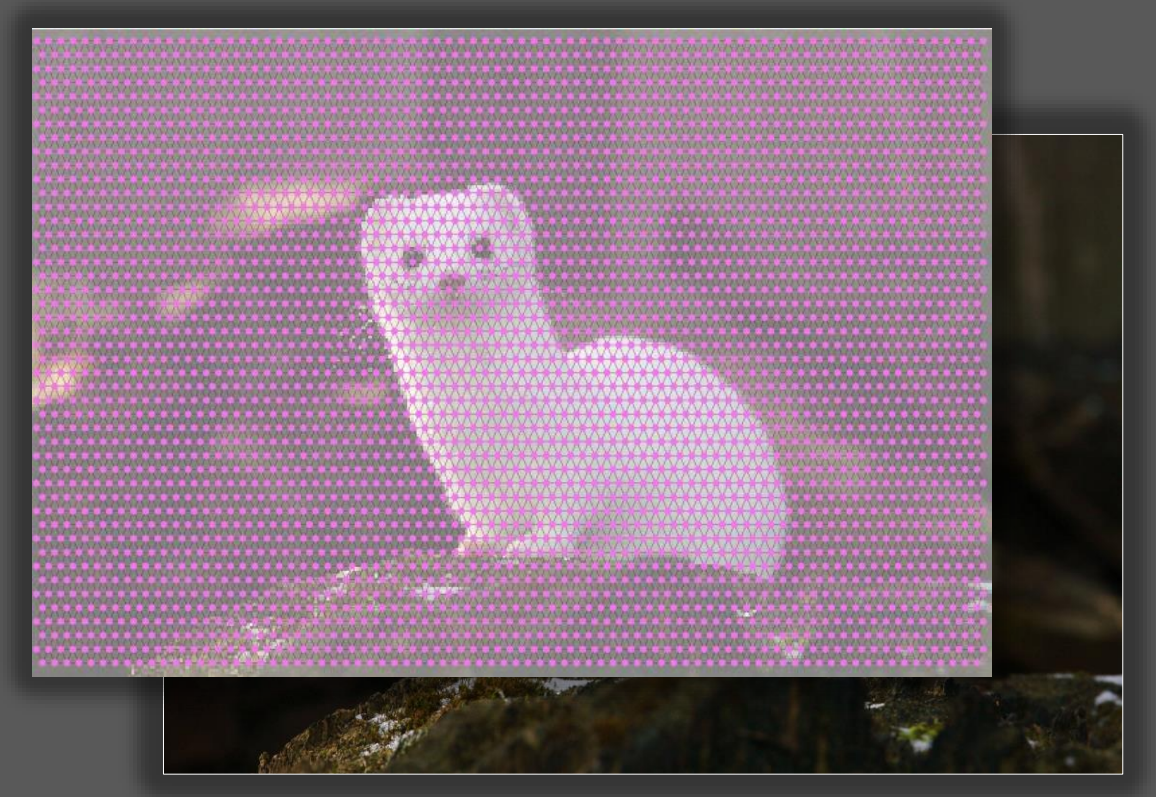


QUESTION. How might we segment this image into “handsome stoat” and “background”?

dynamic programming
greedy algorithms
translation strategy

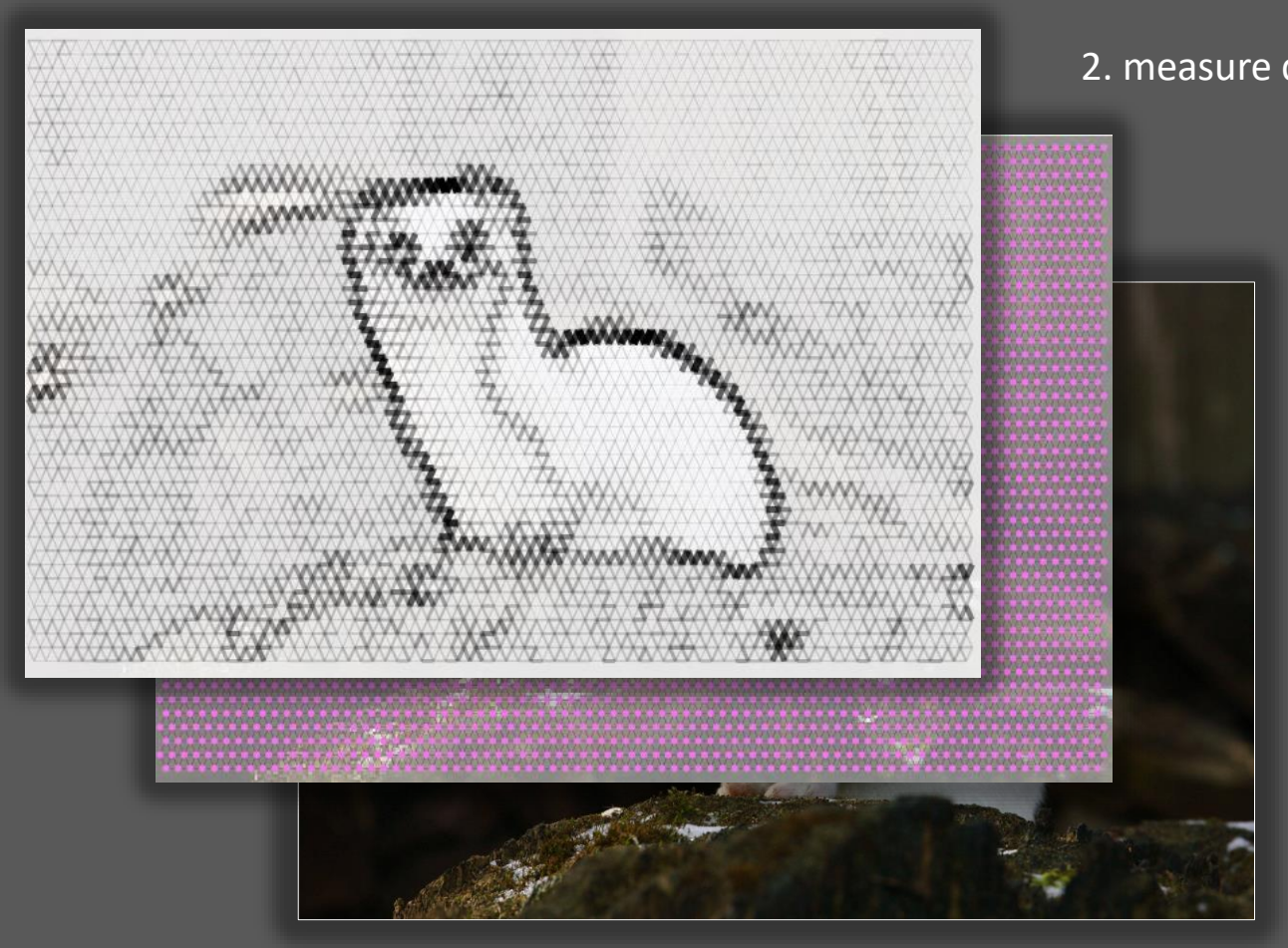
depth-first search
breadth-first search
Dijkstra’s algorithm
Bellman-Ford algorithm
Johnson’s algorithm

Ford-Fulkerson algorithm
matchings
topological sort
Prim, Kruskal

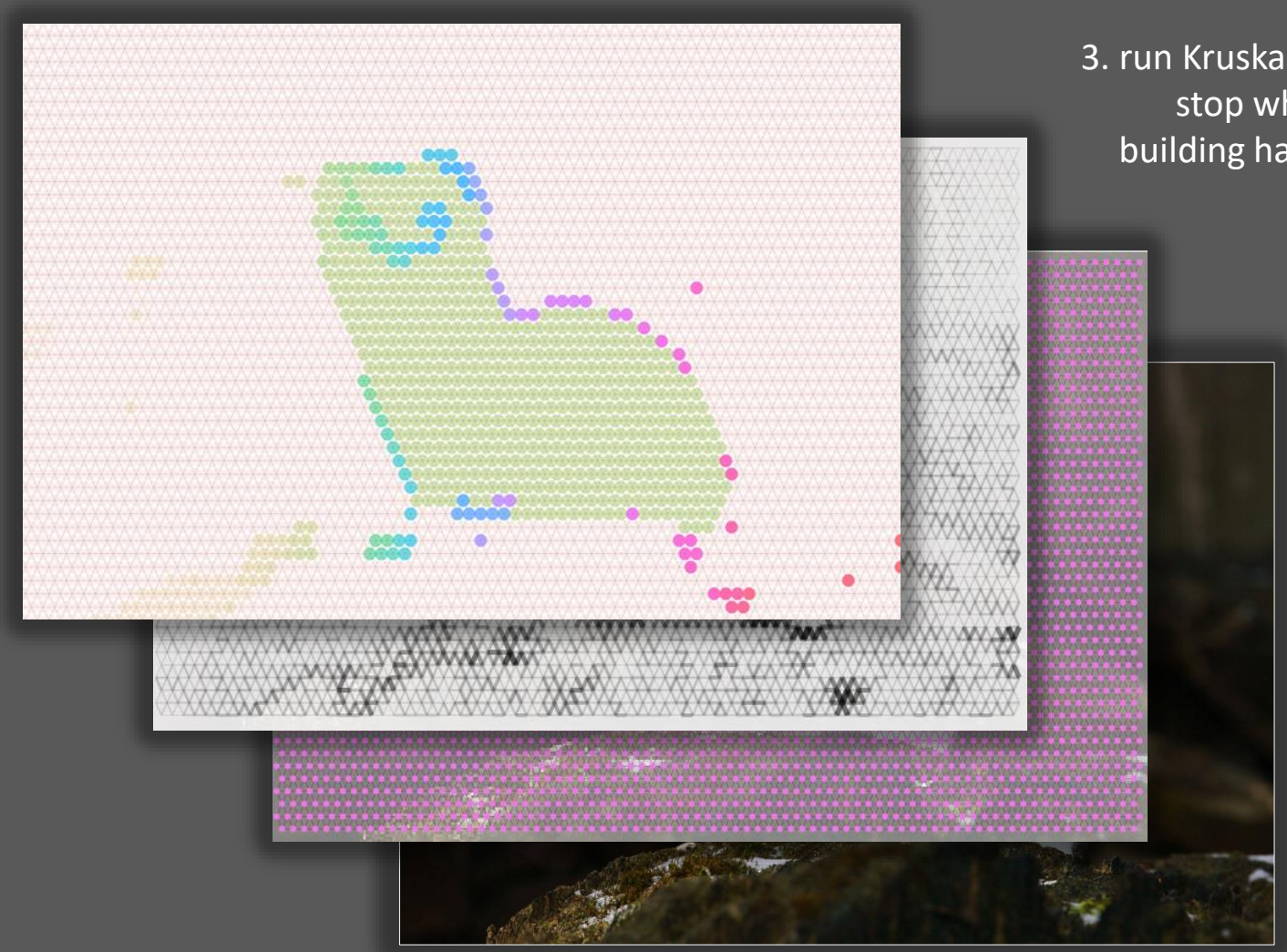


1. define a grid

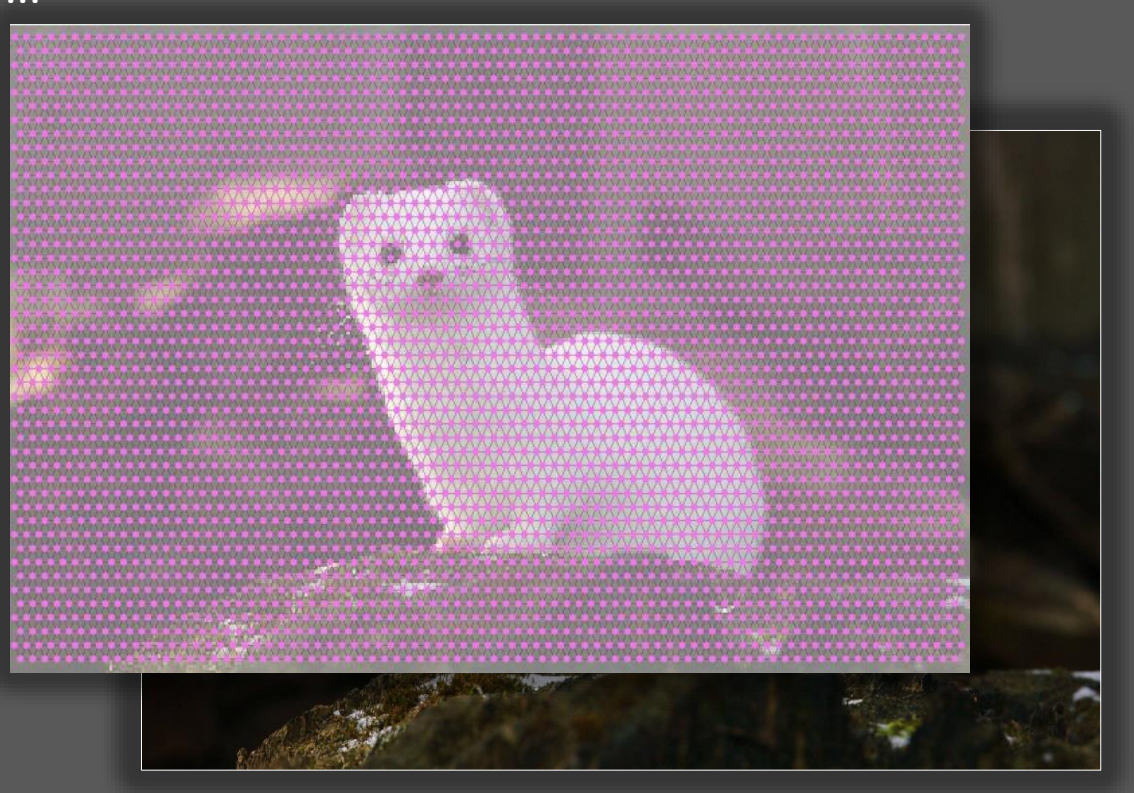
2. measure dissimilarity along edges



3. run Kruskal's algorithm, and stop when the forest it's building has just a few trees

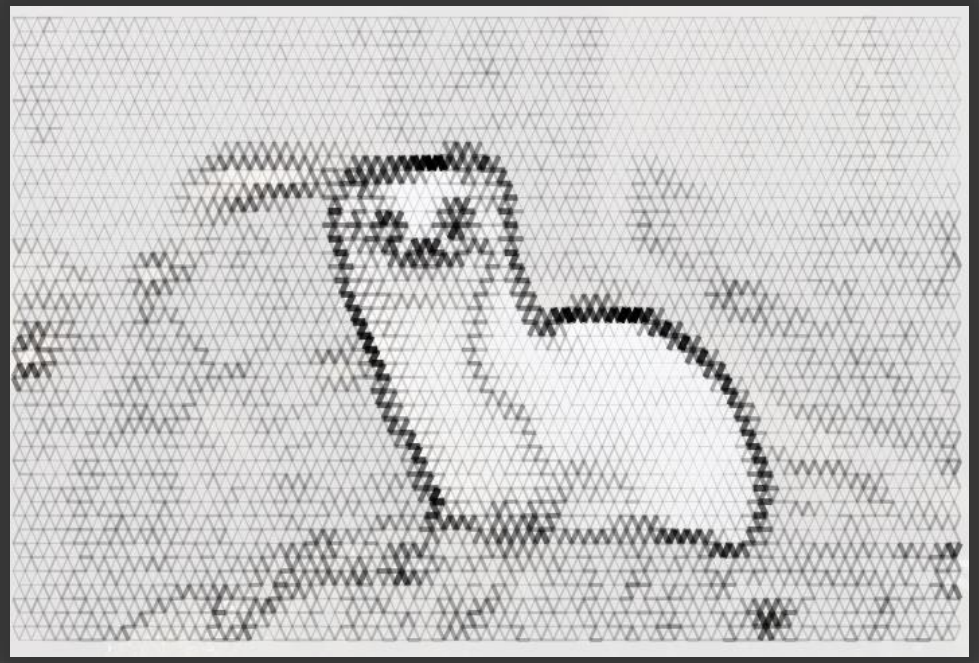


Alternatively ...



1. define a grid

Alternatively ...



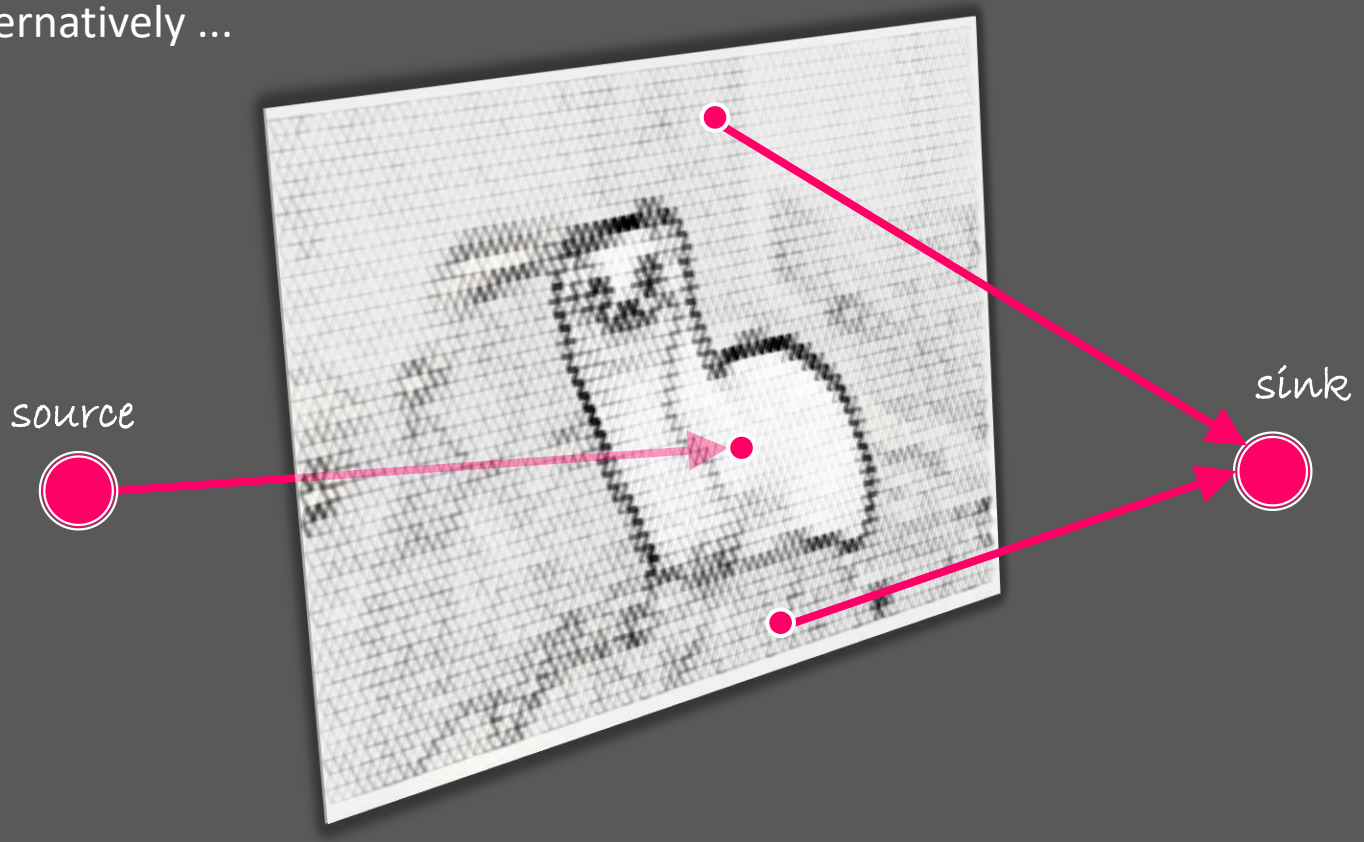
2. measure dissimilarity along edges

Alternatively ...

3. ask the user to label some “stoat” points and some “background” points

Alternatively ...

4. set up a flow network



Alternatively ...

5. find a minimum-capacity cut