Right from the beginning, and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.

Edsger Dijkstra (1930—2002)
On the cruelty of really teaching computer science, 1988

The “Selective Attention Test” is a famous experiment by Simons and Chabris (1999). Around half of viewers don’t even notice the unexpected event – someone in a gorilla suit – in the middle of this video.

Challenge 1: fast maximum?

We are given a collection of n items. Common sense (and also CLRS section 9.1) says that to find the maximum we need to make $n-1$ comparisons. The challenge is to find the maximum using only $O(\log n)$ comparisons between items.

To make the problem concrete, let the items be integers in the range $\{1, \ldots, M\}$ for some constant M. We're only counting comparisons, so treat any arithmetic operations as zero-cost.

This question looks like it's about algorithmic complexity ....

but it's actually hiding two gorillas.

Ben Stokes (Pembroke)
Tunan Shi (Sidney Sussex) ×2
Kuba Bachurski (Trinity) ×2
Siddhant Mukherjee (Selwyn)
Michael Lee (Wolfson)
Zhiyi Liu (Trinity)

Challenge 1: fast maximum?

We are given a collection of n items. Common sense (and also CLRS section 9.1) says that to find the maximum we need to make $n-1$ comparisons. The challenge is to find the maximum using only $O(\log n)$ comparisons between items.

To make the problem concrete, let the items be integers in the range $\{1, \ldots, M\}$ for some constant M. We're only counting comparisons, so treat any arithmetic operations as zero-cost.

Silly question! It's O(1), since M is constant.

But ... Is the question still interesting if we replace M by n?

```
def max(x, M):
    a = [0 for _ in range(M)]
    for xi in x:
        a[xi] = 1
    return the last index m such that a[m]=1
```

Challenge 1: fast maximum?

We are given a collection of n items. Common sense (and also CLRS section 9.1) says that to find the maximum we need to make $n-1$ comparisons. The challenge is to find the maximum using only $O(\log n)$ comparisons between items.

To make the problem concrete, let the items be integers in the range $\{1, \dots, M\}$ for some constant M. We're only counting comparisons, so treat any arithmetic operations as zero-cost.

Actually, the whole challenge is silly, since comparisons and arithmetic are basically the same thing!

$$\max(a,b) = \tfrac{1}{2}(a + b + |a-b|)$$

[Oh, but it's cheating to use $|\cdot|$, since $|x| = \max(x,-x)$ is basically a comparison.]

$$|x| = \sqrt{x*x} \qquad \sqrt{x} = e^{\frac{1}{2}\log x}.$$

If we're allowed log and exp, we get max for free!

Challenge 1 is silly, since arithmetic and comparison can’t be disentangled.

And any proof (such as in CLRS3 section 9.1) that doesn’t acknowledge this is “not even wrong”.

Right

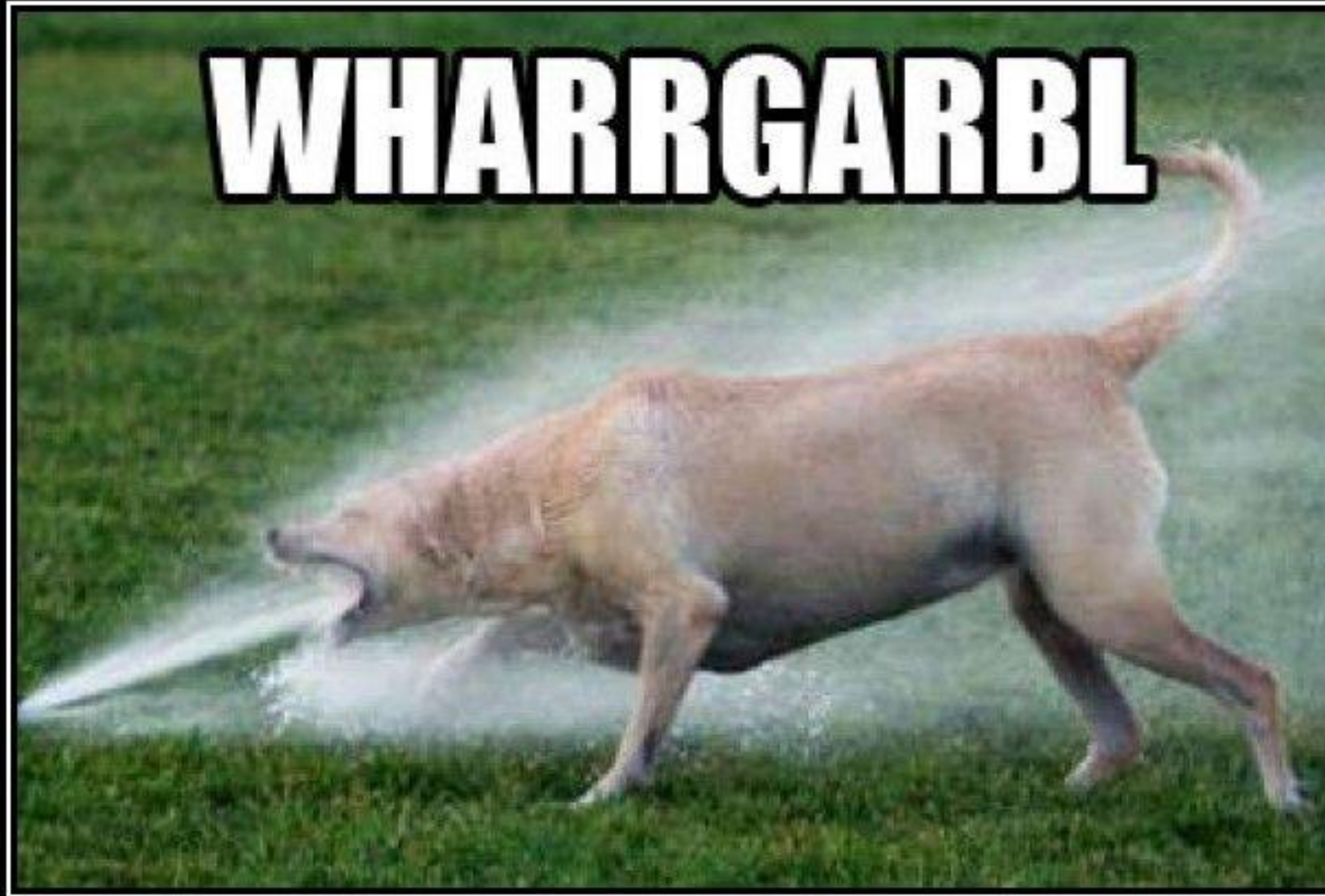
Wrong

Not even wrong

Wolfgang Pauli (1900-1958)

“Das ist nicht nur nicht richtig;
es ist nicht einmal falsch”

another way to say “not even wrong” …



Exam question. Let dijkstra_path(g,s,t) be an implementation of Dijkstra's shortest path algorithm that returns the shortest path from vertex s to vertex t in a graph g. Prove that the implementation can safely terminate when it first encounters vertex t.

Dijsktra's algorithm adds vertices to a min priority queue, sorted by their distance from s. It repeatedly pops the minimum element in the priority queue.

This only makes sense if "distance" means "computed distance".

Suppose that at the time we pop t, there is a shorter path to it than the one we have just found. Consider such a path, and let x be the vertex just before t along this path.

Since x is on a shortest path to t, x would have been popped before t (as the path $s \rightsquigarrow x$ is shorter than the path we just found to t). So we would have considered the path to t via x, and found this shorter path. A contradiction.

This seems to require that we pop in order of true (mathematical) distance. In other words, it's using a different definition of "distance".

This whole answer is "not even wrong" because the reader can't pin down the meaning of the word "distance".

CLRS3 lemma 24.15 (used in Bellman-Ford). Consider a weighted directed graph. Consider any shortest path from s to t,

$$s = v_0 \to v_1 \to \cdots \to v_k = t.$$

Suppose we initialize the data structure by

v.dist = ∞ for all vertices other than s

s.dist = 0

and then we perform a sequence of relaxation steps that includes, in order, relaxing $v_0 \to v_1$, then $v_1 \to v_2$, then … then $v_{k-1} \to v_k$. After these relaxations, and at all times thereafter, v_k.dist = distance(s to v_k).

We'll prove by induction that, after the ith edge has been relaxed,

$$v_i.\texttt{dist} = \text{distance}(s \text{ to } v_i)$$

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

INDUCTION STEP: …

If there's a graph with -ve weight cycle, it's possible that distance (s to s) = -∞!

So, is this proof right, wrong, or not even wrong?

If you want more effective programmers, you will discover that they should not waste their time debugging, they should not introduce the bugs to start with.

Edsger Dijkstra (1930—2002)

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth (1938—)

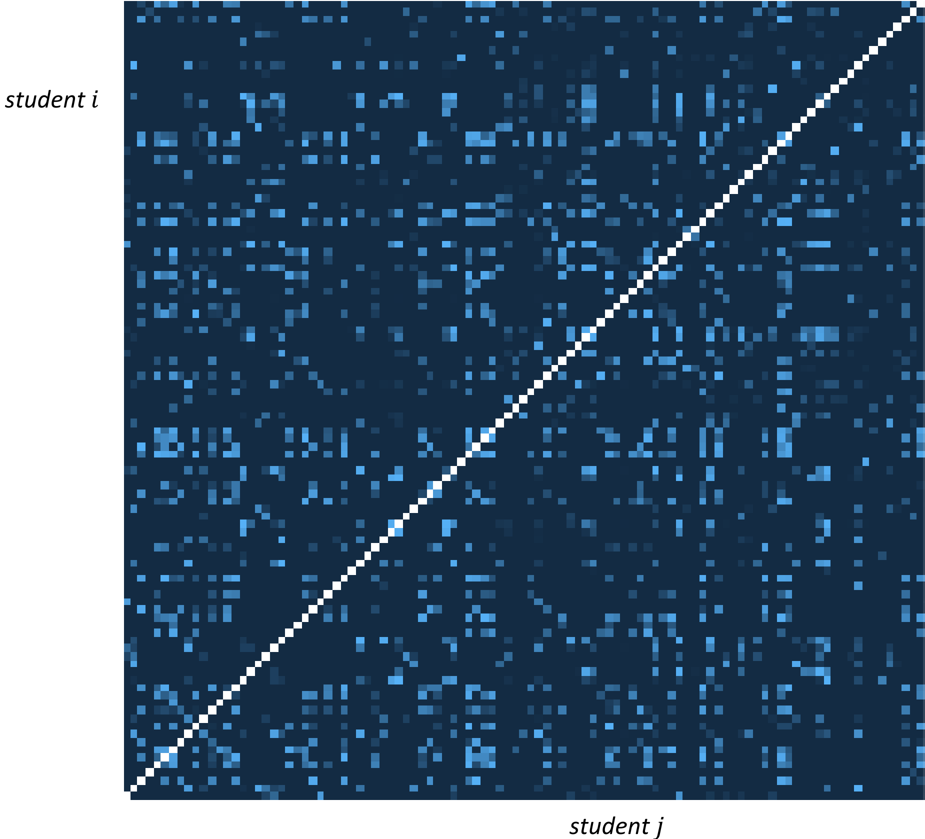
Coming up next week

- Algorithms for finding structures in graphs

Next Friday:

- Graph algorithms as optimization problems
- MaxFlow-MinCut and Generative Adversarial Networks

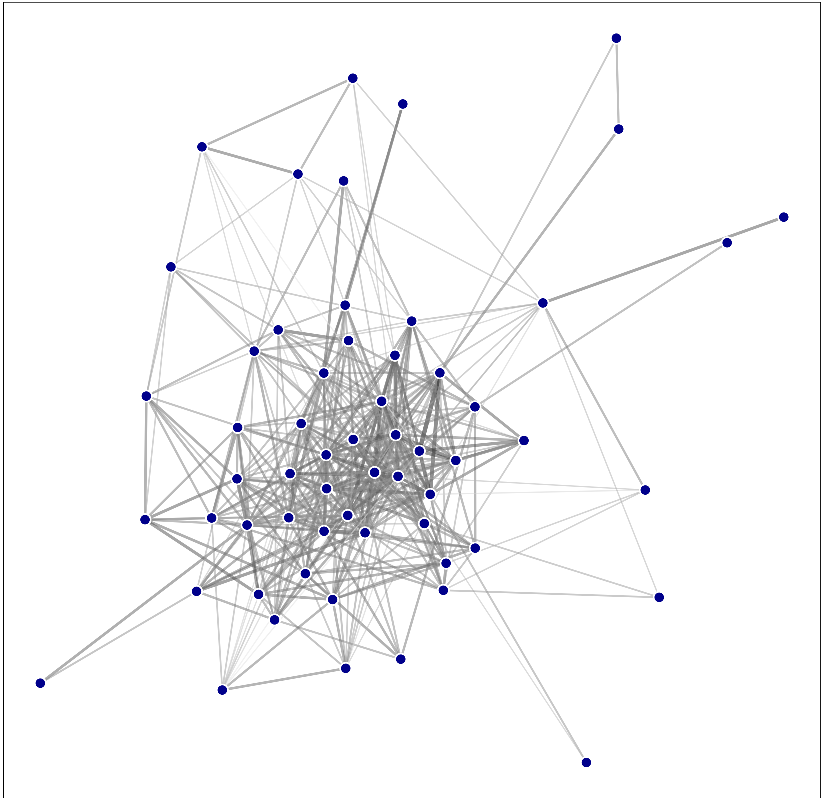
Similarity matrix of Tick 1 code



high similarity

low similarity

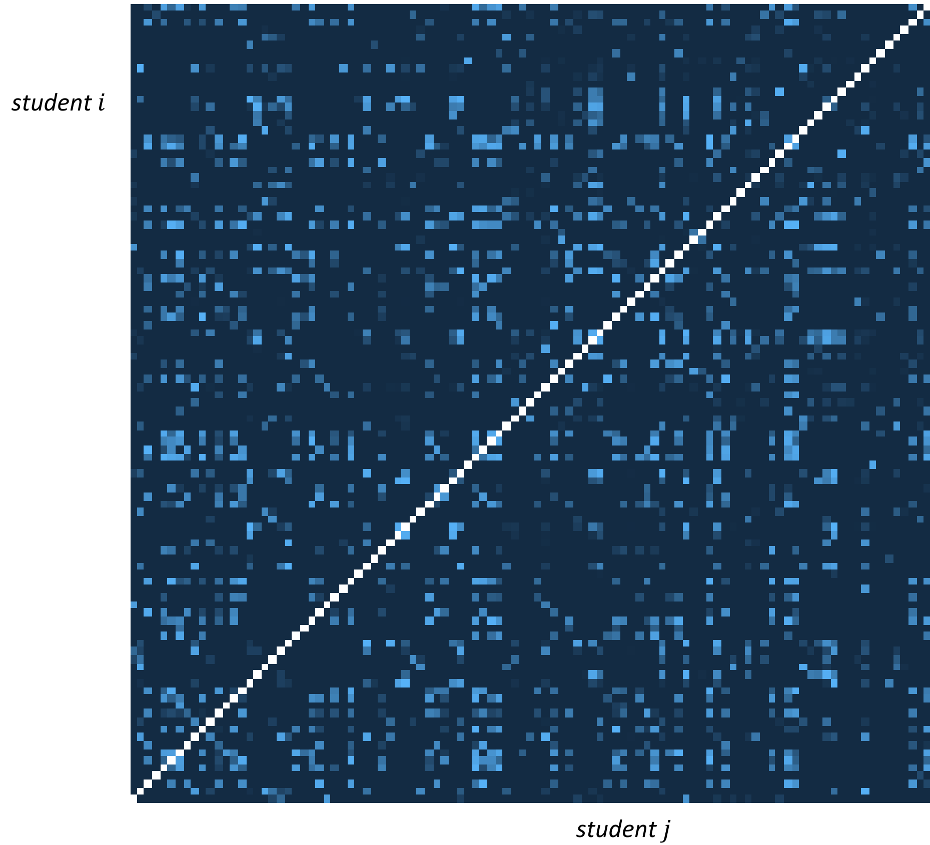
Similarity graph

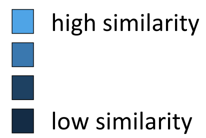


high similarity

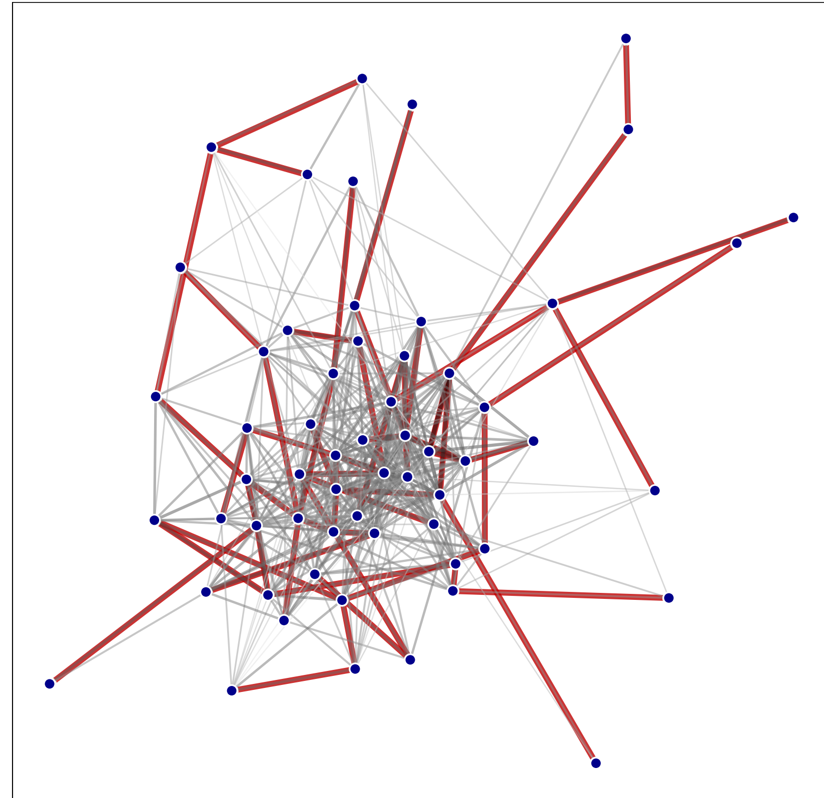
low similarity

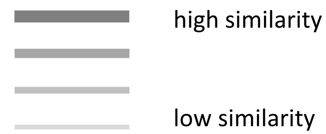
Similarity matrix of Tick 1 code



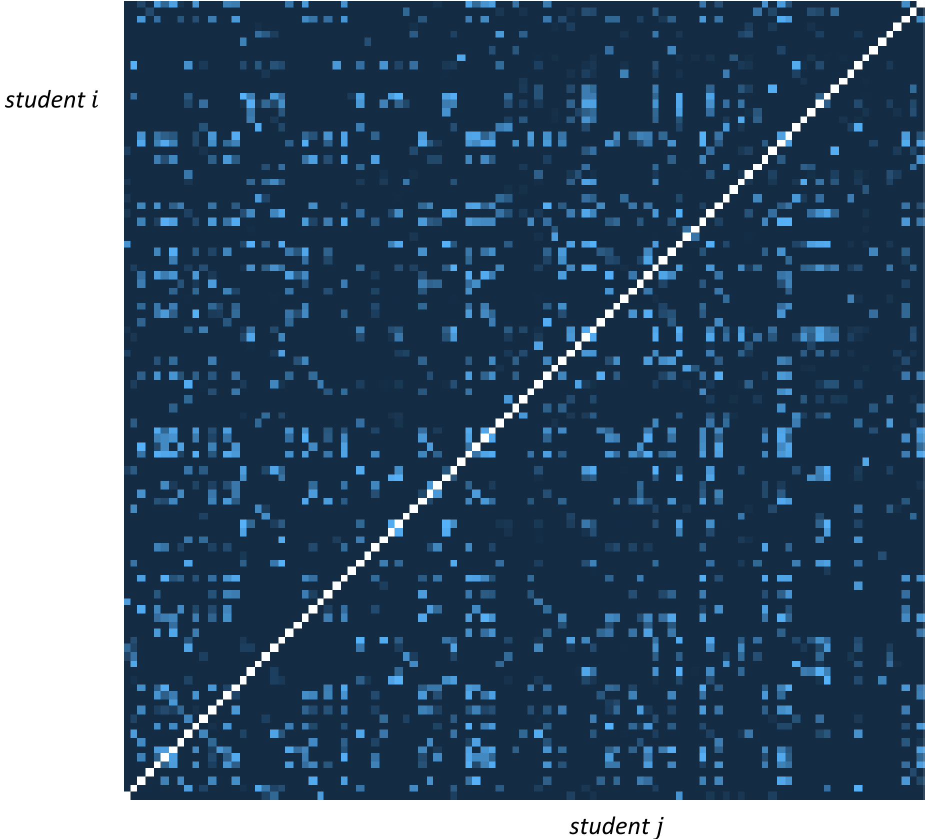


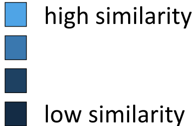
Similarity graph + embedded tree, using high-weight edges



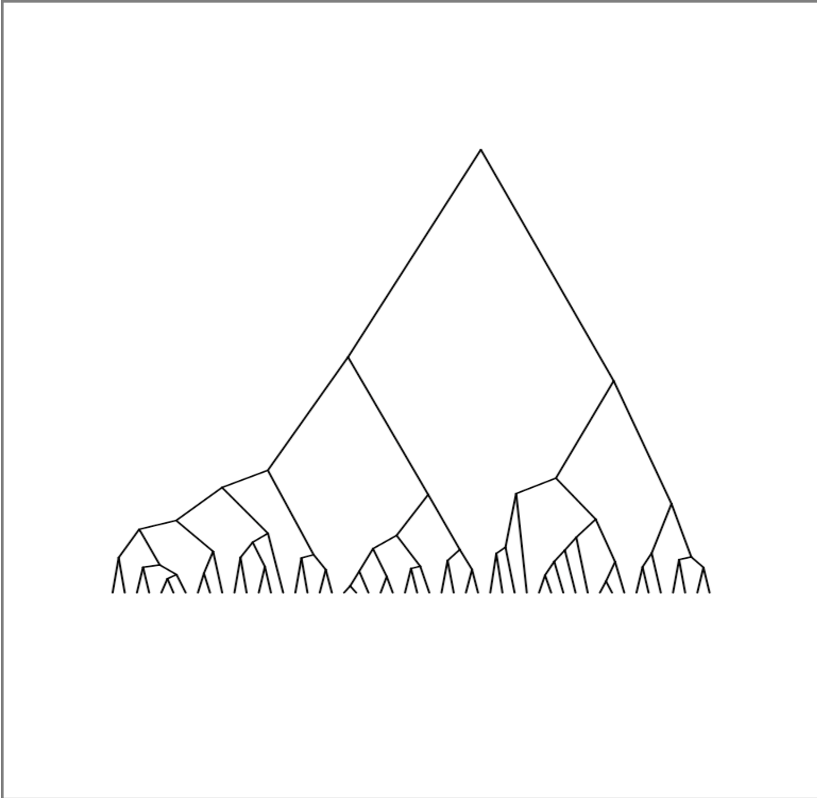


Similarity matrix of Tick 1 code



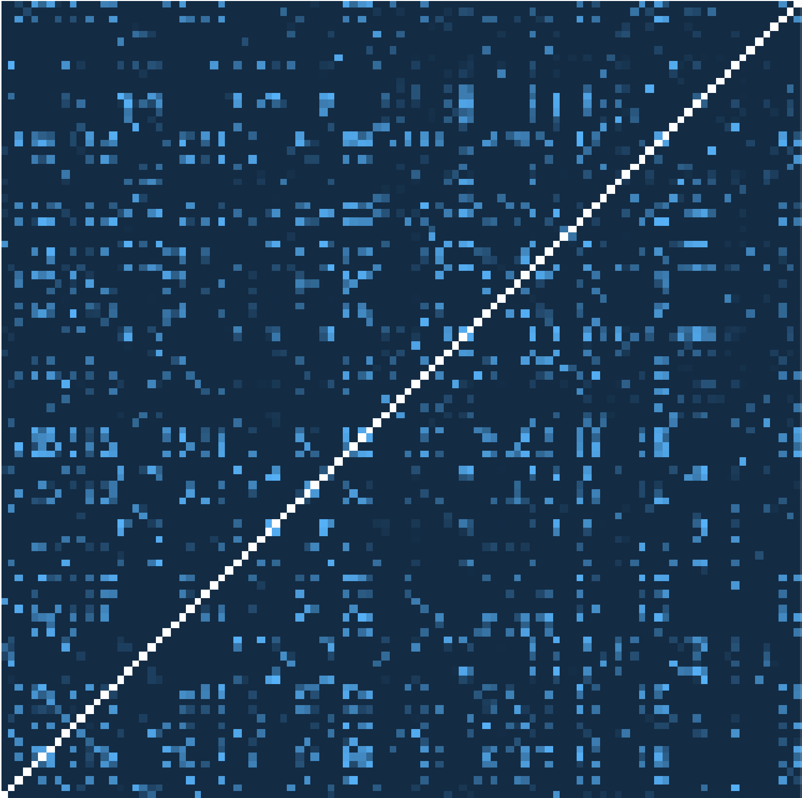


Embedded tree, using high-weight edges

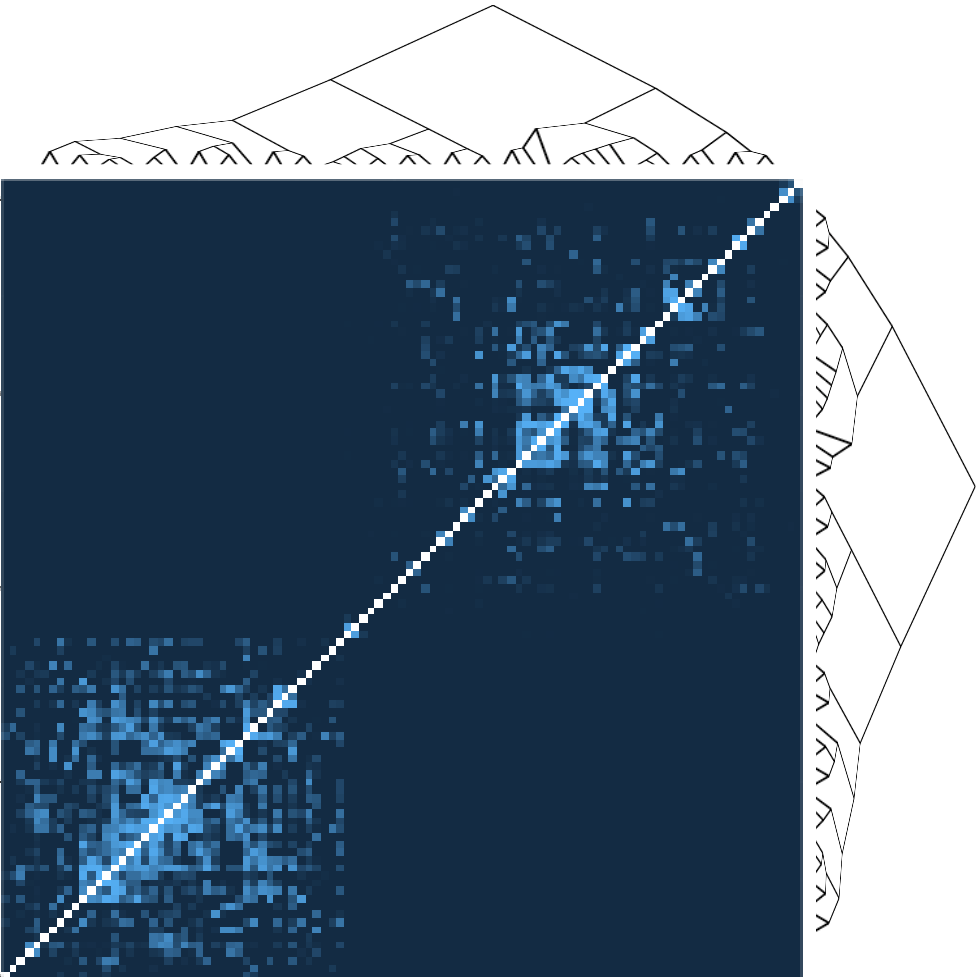


Similarity matrix of Tick 1 code

student i



student j



high similarity

low similarity

Algorithms2 challenge 2

cl.cam.ac.uk/teaching/2122/Algorithm2/ticks/test-challenge2.html

Algorithms2 Challenge 2: Finding Order

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 Tick 1. 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 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,

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

thus any sensible answer will give score > 0.

Hint. Could we improve on the Kruskal clustering method, by choosing which branch of the classification tree is left and which is right? Could we use ideas from toposort? Could we somehow use max-flow to order vertices?