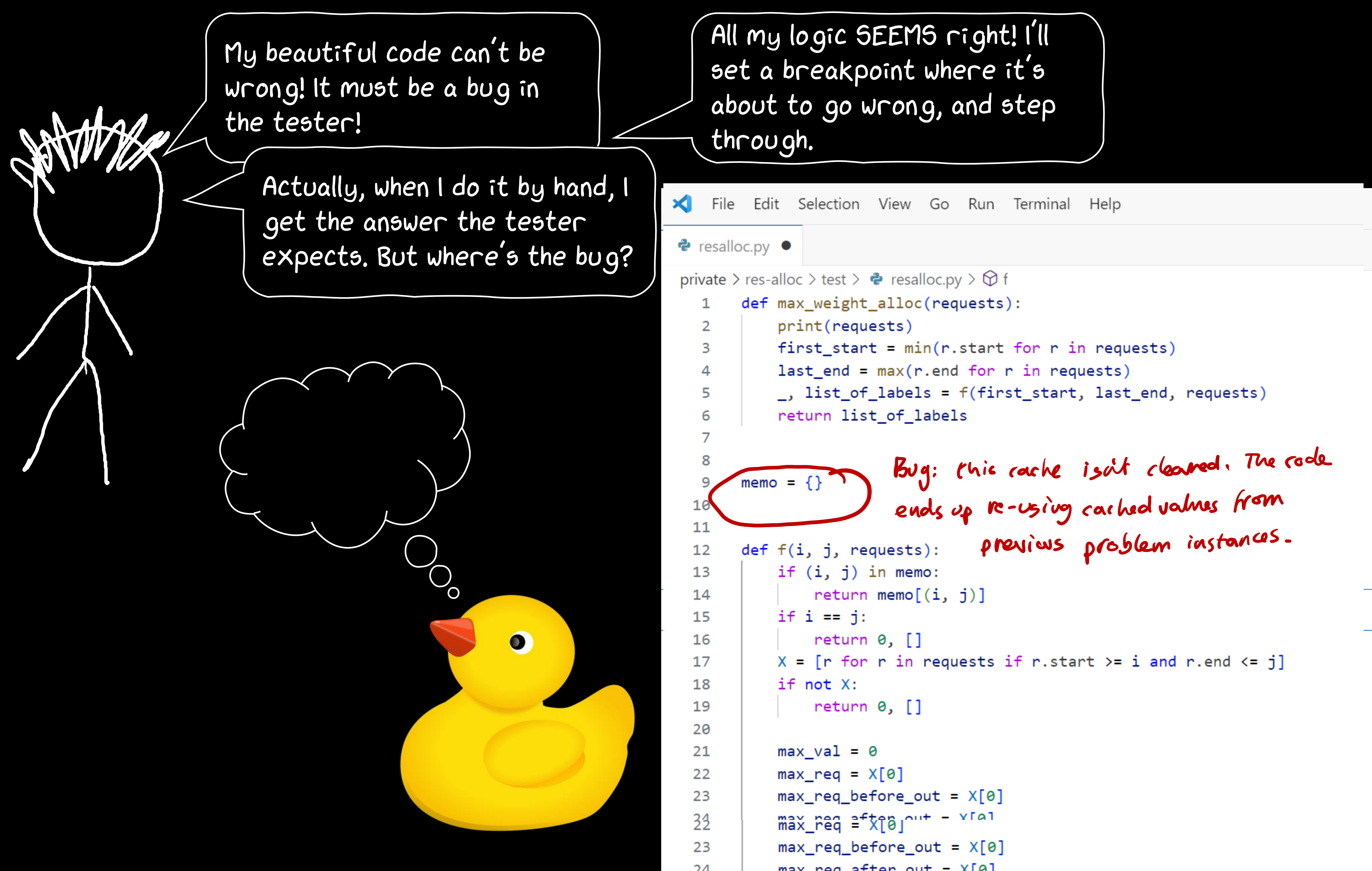

My beautiful code can't be wrong! It must be a bug in the tester!
All my logic SEEMS right! I'll set a breakpoint where it's about to go wrong, and step through.
Actually, when I do it by hand, I get the answer the tester expects. But where's the bug?
File Edit Selection View Go Run Terminal Help
resalloc.py
private > res-alloc > test > resalloc.py > f
def max_weight_alloc(requests):
print(requests)
first_start = min(r.start for r in requests)
last_end = max(r.end for r in requests)
_, list_of_labels = f(first_start, last_end, requests)
return list_of_labels
memo = {}
def f(i, j, requests):
if (i, j) in memo:
return memo[(i, j)]
if i == j:
return 0, []
X = [r for r in requests if r.start >= i and r.end <= j]
if not X:
return 0, []
max_val = 0
max_req = X[0]
max_req_before_out = X[0]
max_req = X[0]
max_req_before_out = X[0]
Bug: this cache isn't cleared. The code ends up re-using cached values from previous problem instances.

Assertion (line 9).

Just after a vertex v is popped, $v.\mathtt{distance} = \text{distance}(s \text{ to } v)$

CLAIM: This assertion never fails.

PROOF: Suppose it does fail. Consider the instant T at which it first fails, and let v be the vertex for which it fails.

By assumption, our assertion succeeded at every point prior to T. We can use this to reason about the events leading up to the failure at T.

We obtain a contradiction. Therefore the supposition is false, i.e. the claim is true.

I call this the “breakpoint” proof strategy.

It’s a proof by induction on program execution ...

1. Decide on a property we want to be true at all times
2. Assume it’s true up to time $T - 1$
3. Show that it must therefore be true at time T

SECTION 5.5

Algorithms and proofs

Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians.

Edsger Dijkstra,
How do we tell truths that might hurt?

How to learn a proof

PASSIVE LEARNING	ACTIVE LEARNING	REFLECTIVE LEARNING
▪ read it / watch it	▪ copy it out ▪ hide part of the proof, and try to fill it in ▪ identify the “beats” of the argument	▪ refactor it to be more elegant ▪ see if it still works when we tweak the problem statement

Problem statement. Given a directed graph in which each edge is labelled with a cost ≥ 0, and a start vertex s, compute the distance from s to every other vertex.

CLAIM. The assertion on line 9 never fails.

PROOF. By induction on program execution. Suppose it first fails at some vertex v. Then,

$$
\begin{aligned}
&\text{distance}(s \text{ to } v\,) \\
&\quad < v.\mathtt{distance} && \text{from our induction supposition} \\
&\quad \leq u_i.\mathtt{distance} && \text{by the nature of the priority queue} \\
&\quad \leq u_{i-1}.\mathtt{distance} + \text{cost}(u_{i-1} \to u_i) && \text{edge relaxation logic} \\
&\quad = \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i) && \text{induction hypothesis} \\
&\quad \leq \text{distance}(s \text{ to } v)
\end{aligned}
$$

Every "beat" of the proof relies on some crucial part of code, or on an assumption.

(EXERCISE. Where is the assumption "cost ≥ 0" used in this proof?)

```
1  def dijkstra(g, s):
2      for v in g.vertices:
3          v.distance = ∞
4      s.distance = 0
5      toexplore = PriorityQueue([s],
                          sortkey = λv: v.distance)
6
7      while not toexplore.is_empty():
8          v = toexplore.popmin()
9          # Assert: v.distance is distance(s to v)
10         # Assert: v is never put back into toexplore
11         for (w,edgecost) in v.neighbours:
12             dist_w = v.distance + edgecost
13             if dist_w < w.distance:
14                 w.distance = dist_w
15                 if w in toexplore:
16                     toexplore.decreasekey(w)
17                 else:
18                     toexplore.push(w)
```

Example sheet 4

Graphs and path finding

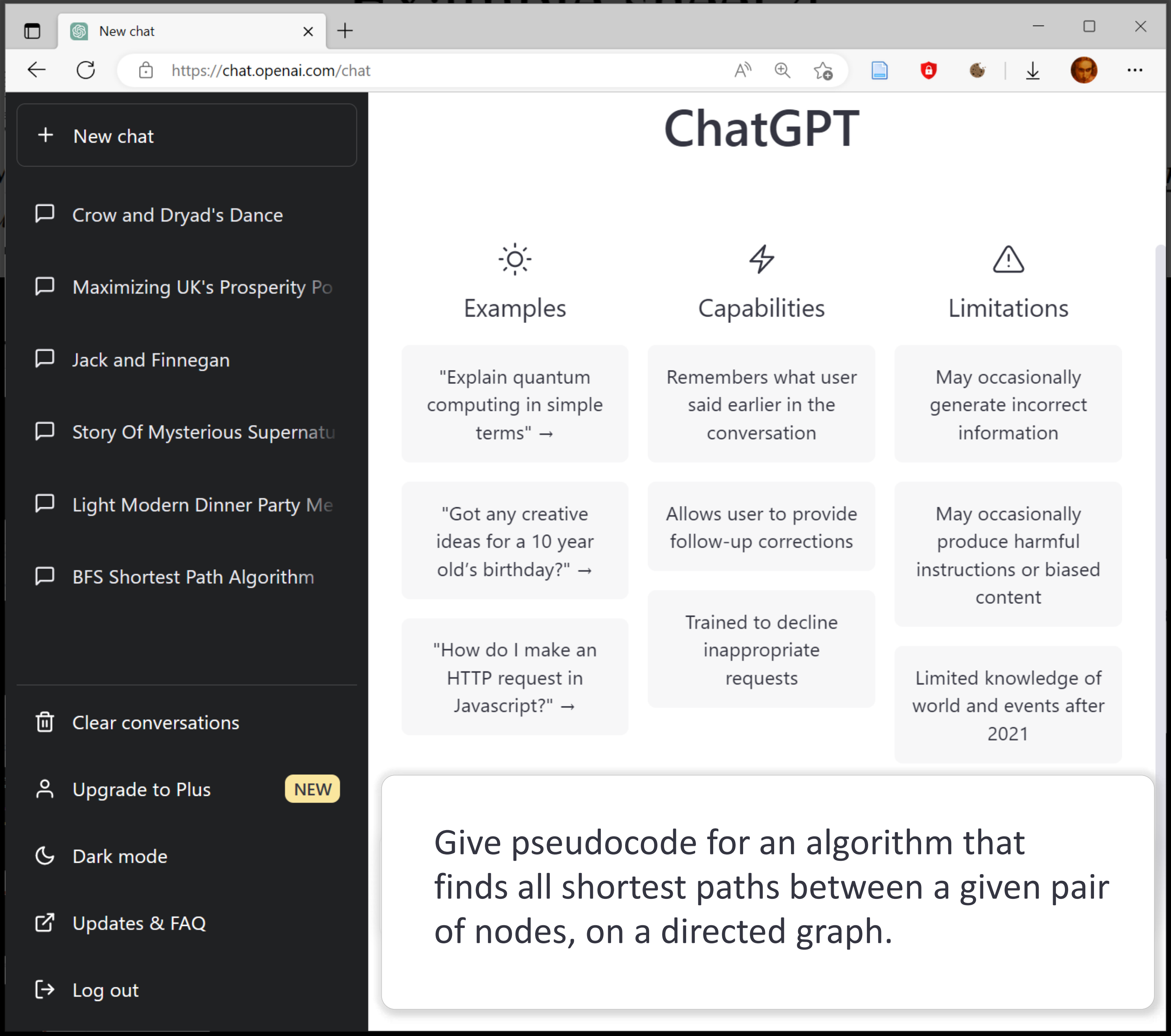
Algorithms—DJW*—2022/2023

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.]*

[redacted]

Example sheet 4

Question 6. M ... r, *linked to on the course website, for you* ...



Here is the pseudocode for finding all shortest paths between a given pair of nodes using Breadth-First Search (BFS) algorithm:

scss

Copy code

```
BFS_shortest_paths(graph, start, end):
  queue = []
  paths = []
  visited = set()
  queue.append((start, [start]))
  visited.add(start)

  while queue:
    node, path = queue.pop(0)
    if node == end:
      paths.append(path)
    for neighbor in graph[node]:
      if neighbor not in visited:
        queue.append((neighbor, path + [neighbor]))
        visited.add(neighbor)

  return paths
```

Note: The `graph` variable is a dictionary where each key is a node and its values are a list of neighbors of that node. The `start` and `end` variables are the nodes between which the shortest paths are to be found.

DW

This code is meant to find all shortest paths between a pair of nodes in a graph. However, I'm not sure it's correct. Is there a problem with this code?

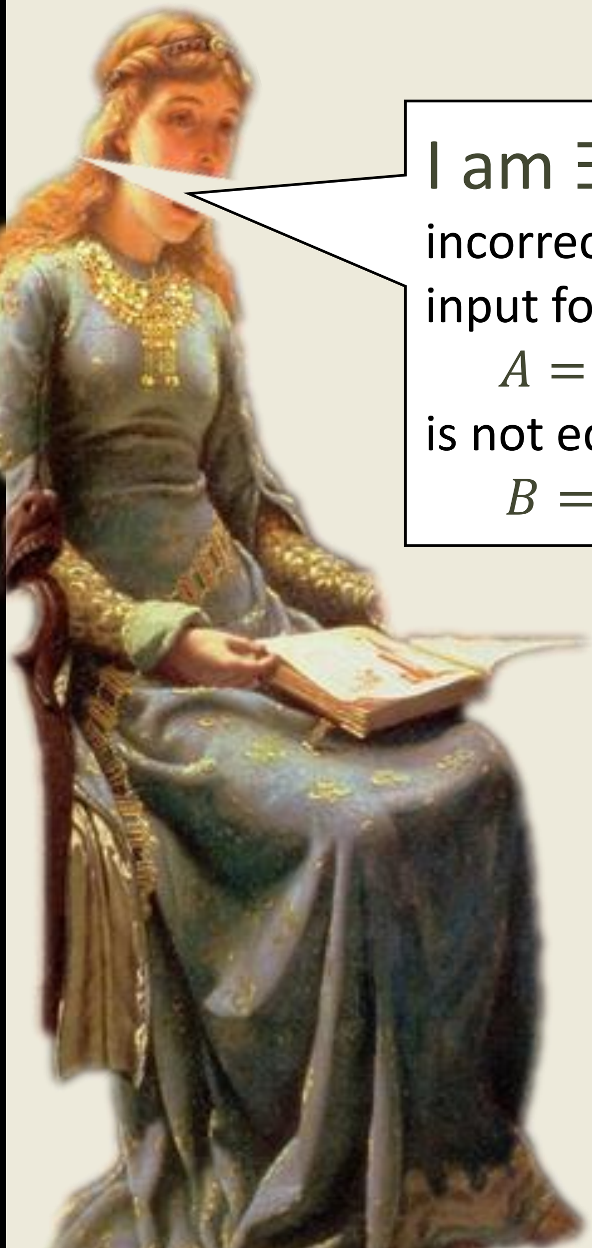
The problem with this code is that it does not necessarily find all shortest paths between the given pair of nodes in the graph.

The code uses Breadth-First Search (BFS) to explore the graph and finds the shortest path from the start node to the end node. However, it only adds the first path that it finds to the **`paths`** list, and does not continue searching for other shortest paths.

To find all shortest paths between the given pair of nodes, the code needs to be modified to continue searching for other paths after finding the first shortest path. One way to do this is to use a **`visited`** set to keep track of the visited nodes and add all paths with the same length as the shortest path found so far to the **`paths`** list.

DW

This code is meant to find all shortest paths between a pair of nodes in a graph. However, I'm not sure it's correct. Is there a problem with this code?



I am ∃loise. The code is incorrect: there **exists** an input for which the set

$$A = \{\text{paths returned}\}$$

is not equal to the set

$$B = \{\text{shortest paths}\}$$

I am ∀belard. The code is correct: for **all** inputs, $A = B$.

How do we prove this sort of result? Typically by proving $A \subseteq B$ and $B \subseteq A$.

I'm expecting these as the "beats" of the proof.

Algorithms challenge chatgpt-b

cl.cam.ac.uk/teaching/2324/Algorithm1/ticks/chatgpt-bfs.html

Algorithms challenge: chatgpt-bfs

Ask ChatGPT to prove a graph algorithm correct

Find prompts that instruct ChatGPT to produce a valid algorithm for solving tick bfs-all.

Then find prompts that instruct ChatGPT to give a valid proof that its algorithm is correct.

Submit a text document (.txt, .rtf, .docx, .odt) containing both sides of your dialogue, including the finished algorithm, on Moodle.

> You should run the algorithm through the tester for bfs-all. You may make syntactical tweaks if necessary to turn the code into valid Python. If you can't get it to produce a valid algorithm, you may use your own algorithm instead.

Types of answer to a question

Right

Wrong

Not even wrong

Wolfgang Pauli (1900–1958)

"Das ist nicht nur nicht richtig;
es ist nicht einmal falsch"

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.

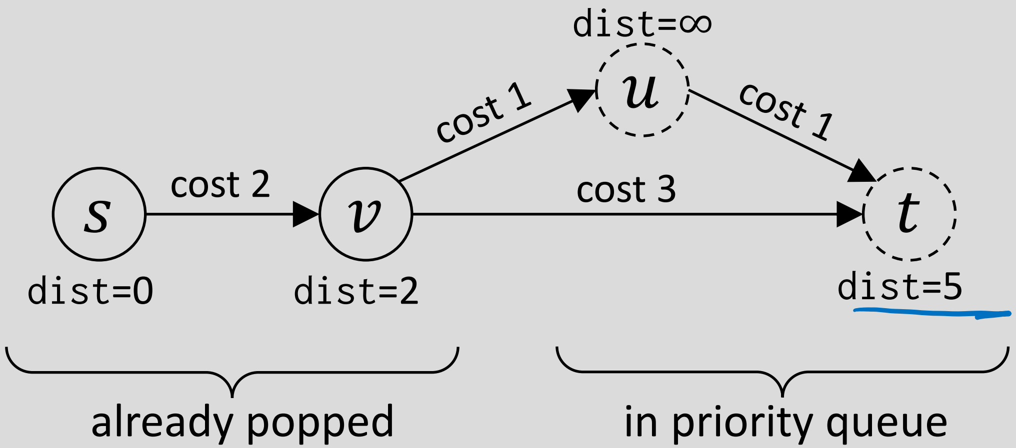
$\underbrace{\text{v.distance}}_{\text{computed}} = \underbrace{\text{distance (s to v)}}_{\text{mathematical}}$

PROPOSED ANSWER.

At the moment when the vertex t is popped from the priority queue, it has to be the vertex in the priority queue with the least distance from s. computed This means that any other vertex in the priority queue has distance $\geq$ computed that for t. Since all edge weights in the graph are ≥ 0, any path from s to t via anything still in the priority queue will have distance $\geq$ that of the probably mathematical distance from s to t when it is popped, thus the distance to t is correct when t is popped.

The $s \to v \to u \to t$ path has dist 4.

Fundamental problem: This question demands a proof by induction.



"This algorithm is correct."

When we evaluate a claim, our answer is either ∀ or ∃

- ∀ inputs, the algorithm's output is correct
- ∃ input for which the algorithm's output is incorrect

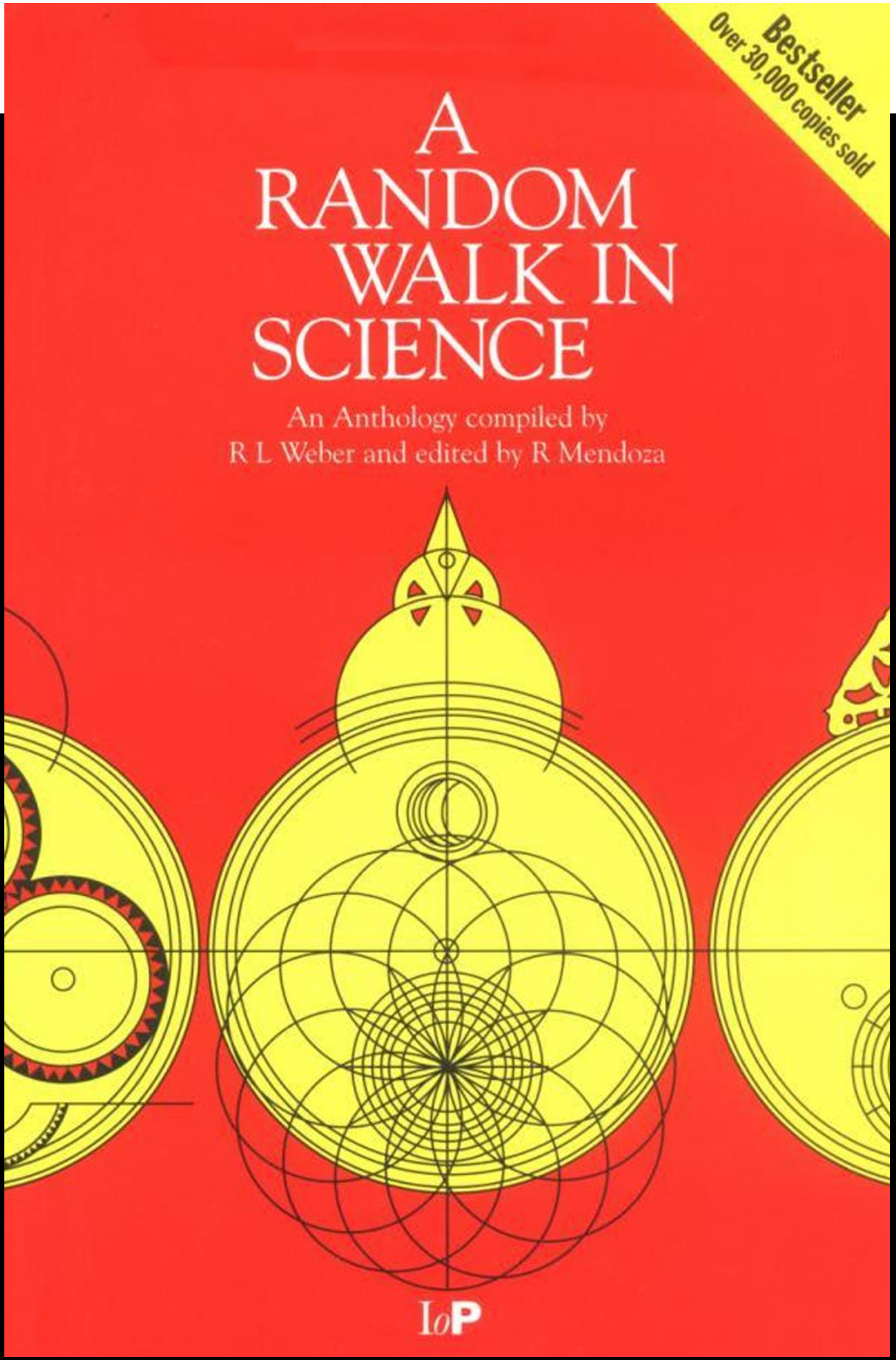
"This proof is correct."

When the conclusion is true and we're evaluating a *proof*, our answer is again either ∀ or ∃

- ∀ steps of the proof, ∀ cases that satisfy the step's premise, the step's conclusion is correct
- ∃ a step of the proof, ∃ a case satisfying the step's premise, for which its conclusion is false

Proof strategies

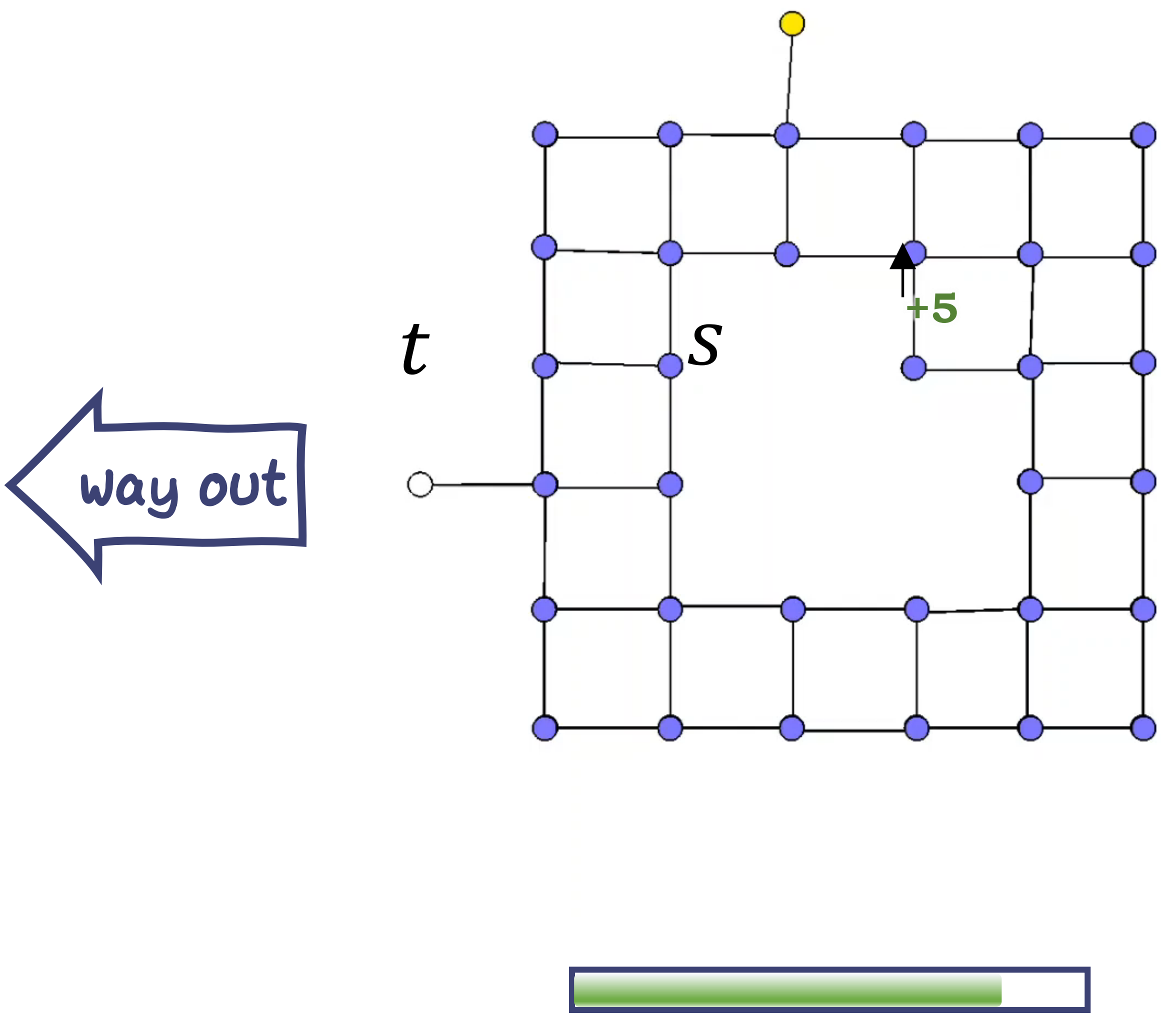
- Breakpoint strategy
  (a type of proof by induction)
- Reductio ad absurdum
  (proof by contradiction)
- Reductio ad nauseam
- Proof by assignment
- Proof by sleight of timetable



SECTION 5.6

Graphs with negative edge weights

Goal: reach the way out before your health runs out.
You can move one step per tick, and your health runs out one unit per tick.
There is a health potion — but is it worth the detour?



game states where we've drunk the potion

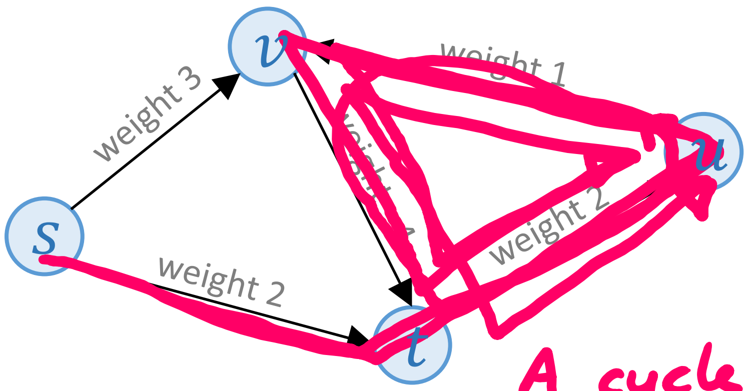
game states where we've not drunk the potion

- the "drink potion" edge has cost -5
- all other edges have cost 1

What is the minimum cost path from s to t?

Let's use terms "edge weight" and "minimum weight path".

What's the issue with negative edge weights?



$$\text{weight}(s \to t \to u) = 4$$

$$\text{weight}(s \to t \to (u \to v \to t) \to u) = 3$$

$$\text{weight}(s \to t \to (u \to v \to t) \to (u \to v \to t) \to u) = 2$$

$$\text{minweight}(s \to u) = -\infty$$

A cycle of weight −1

On this graph, Dijkstra's algorithm will get stuck in an infinite loop.

Clearly, in graphs with negative edge weights, the proof of correctness of Dijkstra's algorithm is invalid. What goes wrong?

Problem statement. Given a directed graph in which each edge is labelled with a cost ≥ 0, and a start vertex s, compute the distance from s to every other vertex.

CLAIM. The assertion on line 9 never fails.
PROOF. By induction on program execution. Suppose it first fails at some vertex v. Then,

distance(s to v)

$< v$.distance   by supposition (part of our proof by induction)

$\leq u_i$.distance   by the nature of Priority Queue

$\leq u_{i-1}$.distance $+ \text{cost}(u_{i-1} \to u_i)$ by Edge Relaxation logic

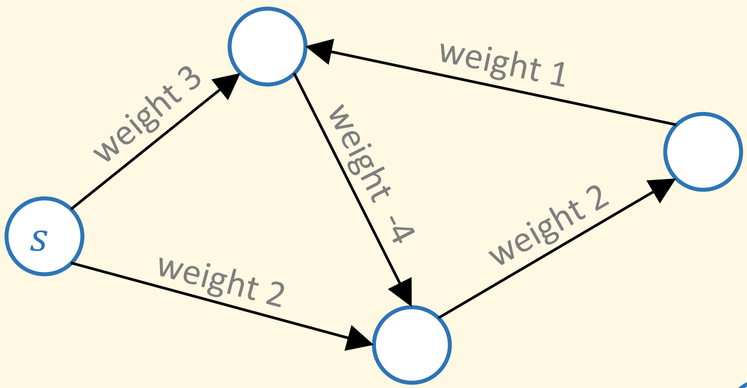
$= \text{distance}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i)$ by induction hypothesis

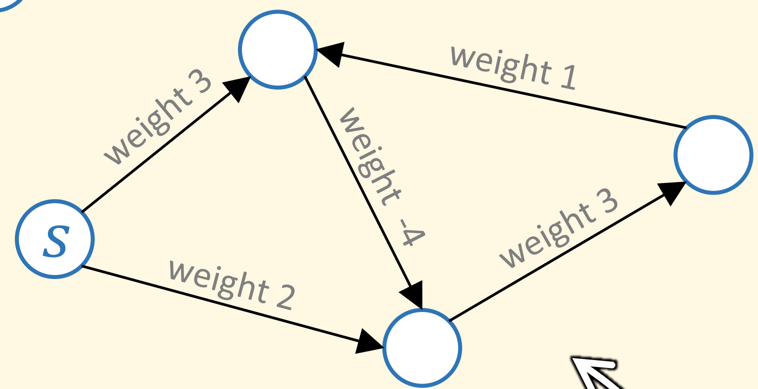
$\leq \text{distance}(s \text{ to } v)$ since all edge costs are ≥ 0.

Proof strategy: check if your proof does indeed use all your assumptions.

EXERCISE (ex4 q13)

Run Dijkstra's algorithm by hand on these two graphs. What happens?





- Could we add a check to Dijkstra's algorithm, so that we can run it safely on any graph?
- For graphs where it terminates, is it always correct?
- If so, is it a good algorithm, or are there better algorithms?

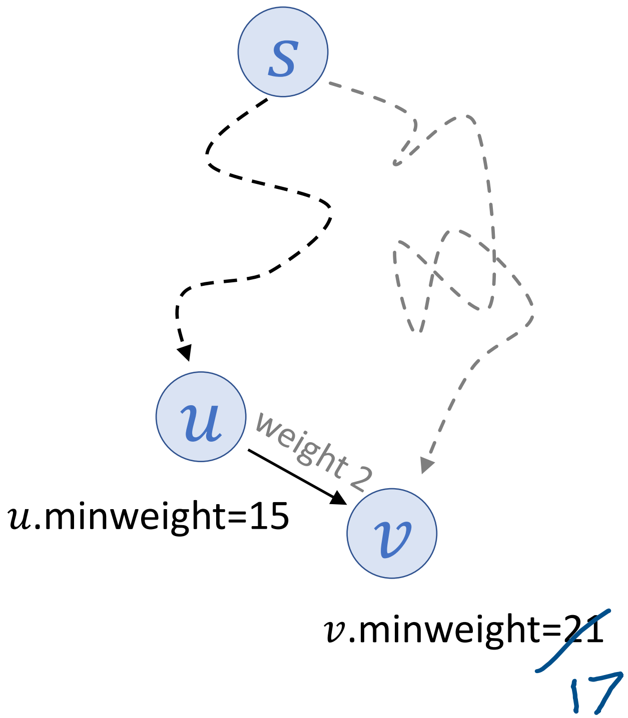
Dijkstra's algorithm gets stuck in an infinite loop

Dijkstra's algorithm terminates, with the correct distances

SECTION 5.6

Bellman-Ford

How can we find minimum-cost paths in graphs where some edge costs may be negative?



Edge relaxation

We're looking for minimum-weight paths from s

For each vertex w, let's store the minimum weight that we've found so far. Call it w.minweight.

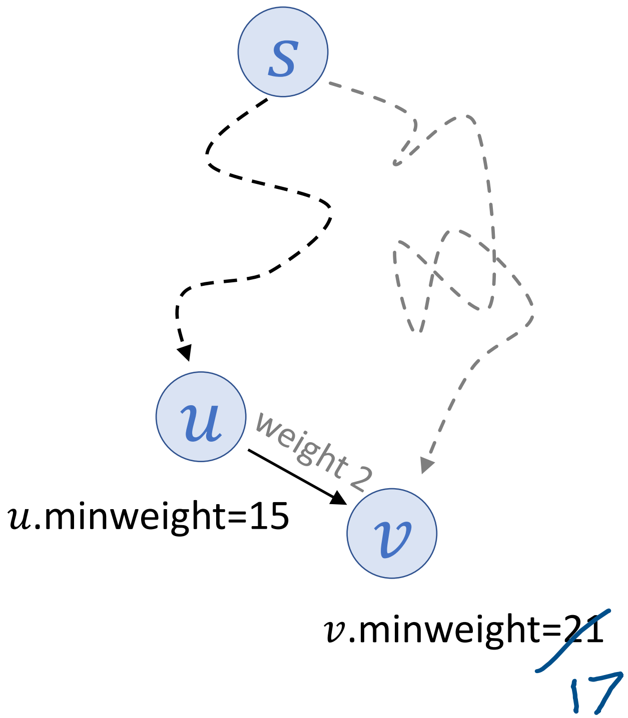
If there's an edge $u \to v$, we may be able to improve v.minweight:

```
if u.minweight + weight(u → v) < v.minweight:
        set v.minweight = u.minweight + weight(u → v)
```

<u>Lemma</u>

If there is an edge $u \to v$, then

minweight (s to v) $\le$ minweight(s to u) + weight $(u \to v)$



Bellman-Ford algorithm

Just keep on relaxing all the edges in the graph, over and over again! (It only takes V rounds.)

```
def bf(g, s):
    for every vertex v:
        v.minweight = ∞
    s.minweight = 0

    repeat |V| − 1 times:
        for every edge e in the graph:
            relax e

    for every edge e in the graph:
        relax e

    if this final pass results in a change:
        raise Exception("negative-weight cycle detected")
    else:
        return the v.minweight values
```

Proof: by Assignment.

Theorem

Given a directed graph g where each edge is labelled with a weight, and given a start vertex s,

- if g has no -ve weight cycles reachable from s, this algorithm finds the true minimum weight from s to every other vertex
- otherwise, it throws an exception