Theorem.

i. Dijkstra's algorithm terminates

ii. When it does, for every vertex v, $v.\mathtt{distance} = \text{distance}(s \text{ to } v)$

iii. The two assertions never fail

```
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()
        # Assert: v.distance is distance(s to v)
        # Assert: v is never put back into toexplore
        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)
```

Assertion line 9.

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

LAST LECTURE

The “breakpoint” proof strategy

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

Assertion line 10.

A vertex v, once popped, is never put back into the priority queue

```
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)
```

Theorem.

i. Dijkstra's algorithm terminates

ii. When it does, for every vertex v, $v.\mathtt{distance} = \mathrm{distance}(s \text{ to } v)$

iii. The two assertions never fail ✓

i. Since vertices can never be pushed back into the priority queue (assertion 10), and there are only finitely many vertices, the algorithm must terminate.

ii. Any vertex v, at the time it is popped, has the correct $v.\mathtt{distance}$ (assertion 9). It remains to prove …

- After v has been popped, $v.\mathtt{distance}$ doesn't change subsequently
- Every vertex that's reachable from s will *eventually* get pushed into the queue, then popped

SECTION 5.5

Algorithms and proofs

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.]*

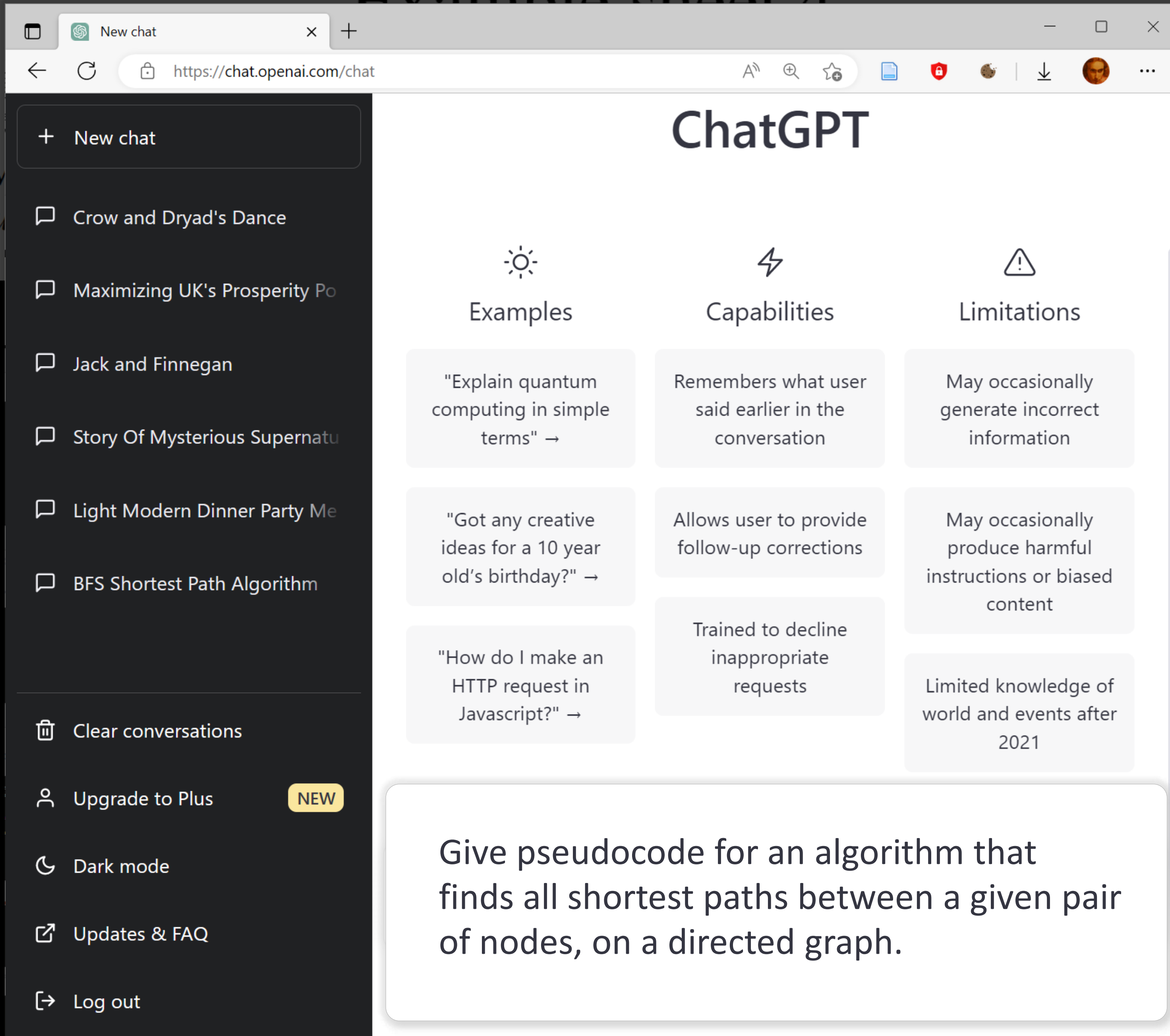
[redacted]

Example sheet 4

Question 6. M

website, for you

r, linked to on the course



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

```scss
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.

Eloise

Abelard

Abelard and Eloise,
Edmund Blair Leighton (1882)

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, the code is incorrect. What is the 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

Modify the code to continue searching for other paths after finding the first shortest path. Do this by using 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.

Here is a modified version of the code that uses a `visited` set to keep track of the visited nodes and adds all paths with the same length as the shortest path found so far to the `paths` list:

lua

Copy code

```
def bfs_shortest_paths(graph, start, end):
    queue = [(start, [start])]
    paths = []
    visited = set()
    visited.add(start)
    shortest_path_length = float('inf')

    while queue:
        node, path = queue.pop(0)

        if node == end:
            if len(path) == shortest_path_length:
                paths.append(path)
            elif len(path) < shortest_path_length:
                shortest_path_length = len(path)
                paths = [path]

        for neighbor in graph[node]:
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
                visited.add(neighbor)

    return paths
```

Algorithms challenge: chatgpt-bfs

Ask ChatGPT to write a graph algorithm

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

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 your algorithm passes the tester, well done! But even if it doesn’t, you are encouraged to submit your attempt.

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

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

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?

DW

Give a proof that this algorithm is correct.

This algorithm uses Breadth-First Search (BFS) to find all shortest paths between a given pair of nodes on a directed graph. BFS works by exploring all nodes at a given distance from the start node before moving on to nodes that are farther away. The distance from the start node to a node is defined as the number of edges in the shortest path from the start node to that node.

Here is the proof that the algorithm is correct:

1. Initialization: The `queue` is initialized with the start node and its corresponding path, and the `distance` dictionary is initialized with the start node and its distance of 0.
2. Termination: The algorithm terminates when the queue is empty. This means that all reachable nodes have been explored and all shortest paths have been found.
3. Correctness: At each step of the algorithm, the node and path popped from the front of the queue correspond to the node and path with the smallest distance from the start node. When a node is popped from the queue and it is the end node, its corresponding path is added to the `paths` list. When a neighbor of the node is explored for the first time, its distance is updated in the `distance` dictionary, and its corresponding path is added to the queue. This ensures that the algorithm only explores the shortest paths and does not revisit any node.

Therefore, the algorithm correctly finds all shortest paths between the given pair of nodes on the directed graph using BFS.

Types of proof

Right

Wrong

Wrong step

incorrect deduction, correct conclusion

Wrong result

incorrect conclusion

Not even wrong

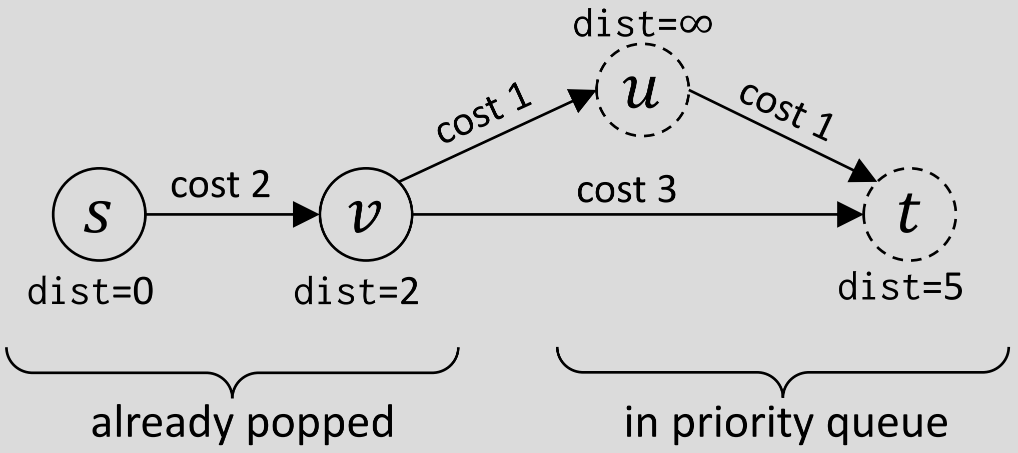
Wolfgang Pauli (1900-1958)

“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.

BAD 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. This means that any other vertex in the priority queue has distance $\geq$ 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 distance from s to t when it is popped, thus the distance to t is correct when t is popped.



EXERCISE

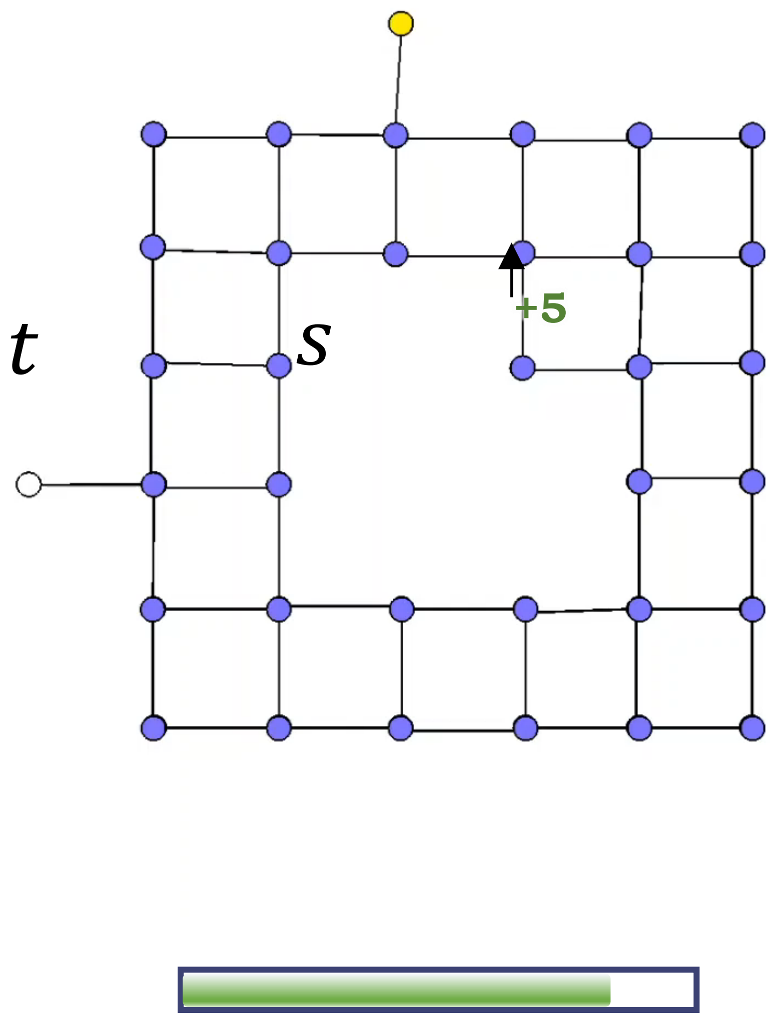
Diagnose the other bad proofs in section 5.5 of notes.

(Remember ∃loise!)

SECTION 5.6

Bellman-Ford

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



game states where we've drunk the potion

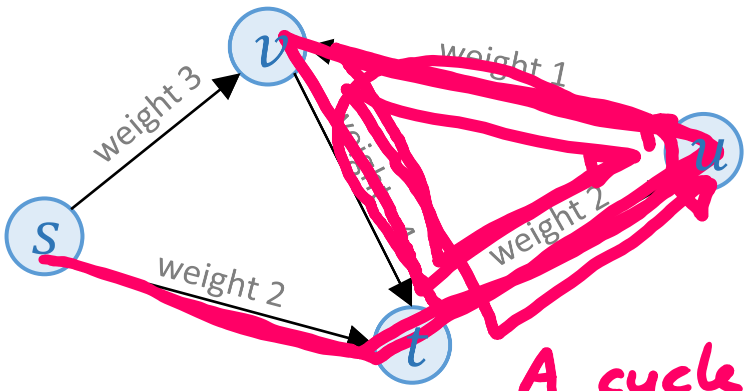
game states where we've not drunk the potion

- 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?



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

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

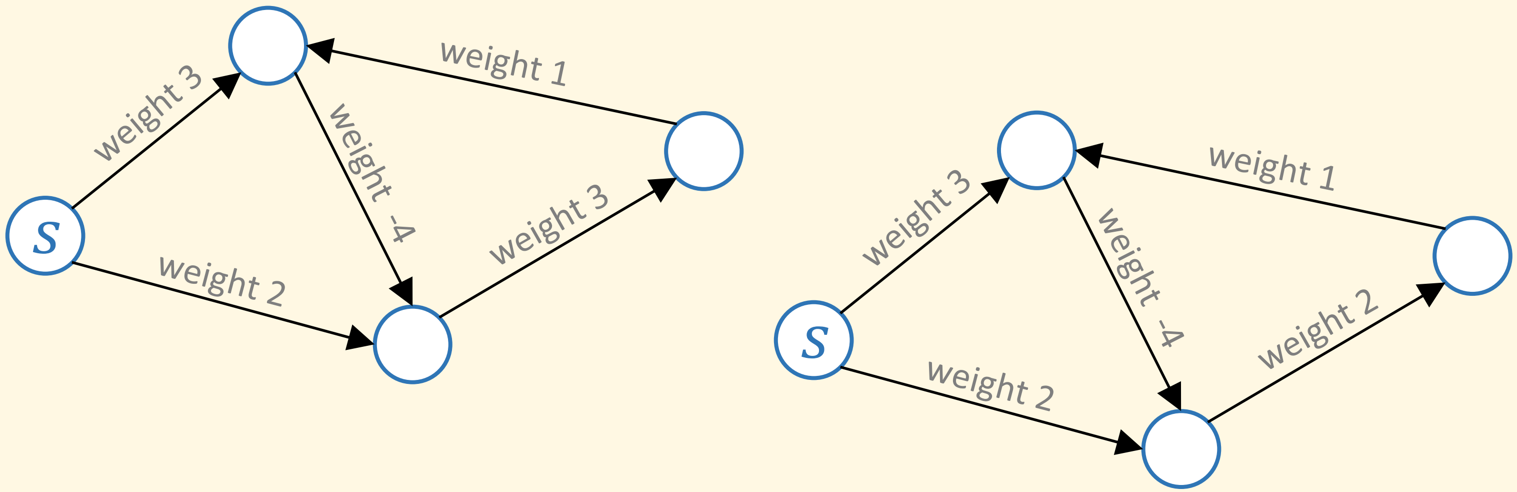
$\text{weight}(s \to t \to (u \to v \to t) \to (u \to v \to t) \to u) = 2$

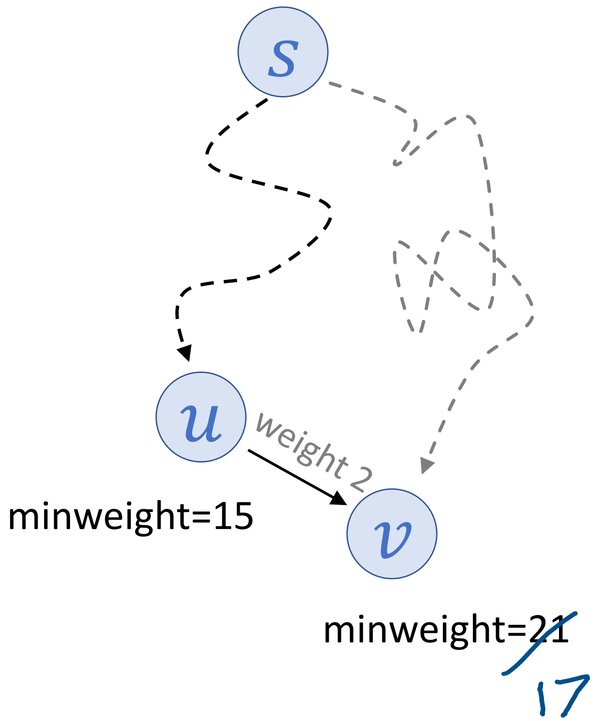
$\text{minweight}(s \to u) = -\infty$

A cycle of weight -1

EXERCISE (ex4 q13)

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





Edge relaxation

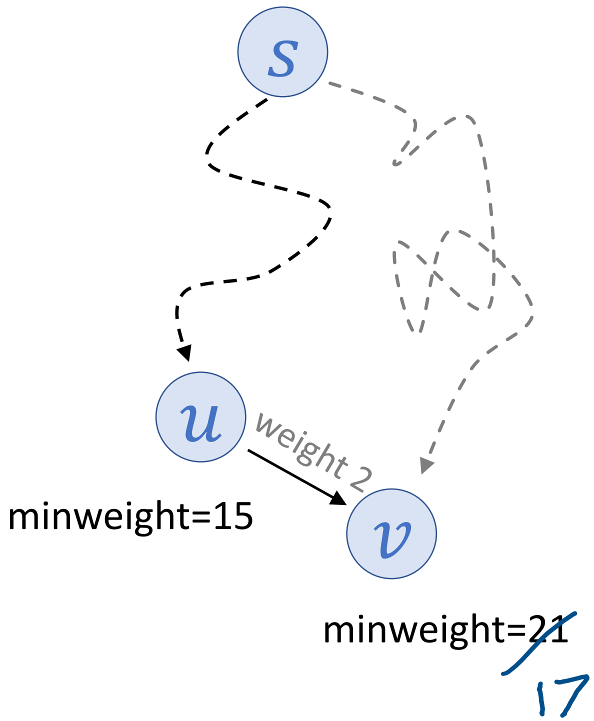
We're looking for minimum-weight paths from s

Let's store the minimum weight of any path we've found so far in the `minweight` variable at each vertex

```
if u.minweight + weight(u → v) < v.minweight:
        let v.minweight = u.minweight + weight(u → 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:
        throw “negative-weight cycle detected”
    else:
        return the v.minweight values
```

Proof: in notes accessible examinable!

Bellman-Ford algorithm

Just keep on relaxing all the edges in the graph, over and over again!

(It only takes V rounds.)

Theorem

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

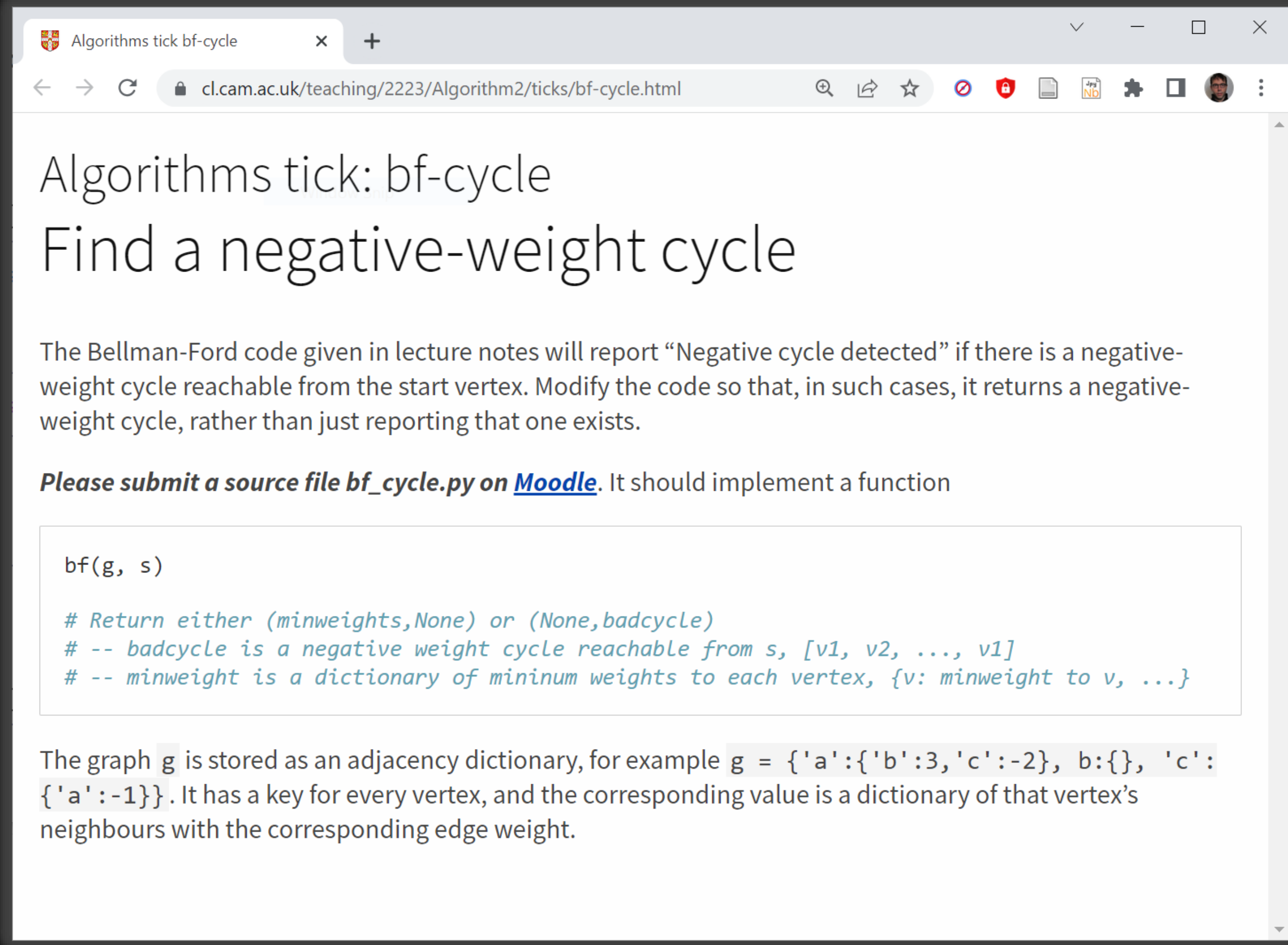
- if g has no -ve weight cycles reachable from s, this algorithm finds the true minimum weight from s to every other vertex
- otherwise, it throws an exception

Example sheet 4

Graphs and path finding

Algorithms—DJW*—2022/2023

Question 19*. The Bellman-Ford code given in lecture notes will report "Negative cycle detected" if there is a negative-weight cycle reachable from the start vertex. Modify the code so that, in such cases, it returns a negative-weight cycle, rather than just reporting that one exists. *[There is a tester, linked to on the course website, for you to check your code.]*



Algorithms tick: bf-cycle

Find a negative-weight cycle

The Bellman-Ford code given in lecture notes will report "Negative cycle detected" if there is a negative-weight cycle reachable from the start vertex. Modify the code so that, in such cases, it returns a negative-weight cycle, rather than just reporting that one exists.

Please submit a source file bf_cycle.py on Moodle. It should implement a function

```
bf(g, s)

# Return either (minweights,None) or (None,badcycle)
# -- badcycle is a negative weight cycle reachable from s, [v1, v2, ..., v1]
# -- minweight is a dictionary of mininum weights to each vertex, {v: minweight to v, ...}
```

The graph `g` is stored as an adjacency dictionary, for example `g = {'a':{'b':3,'c':-2}, b:{}, 'c':{'a':-1}}`. It has a key for every vertex, and the corresponding value is a dictionary of that vertex's neighbours with the corresponding edge weight.