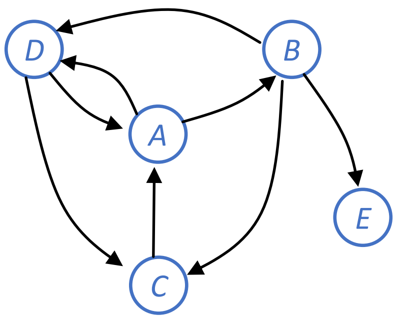
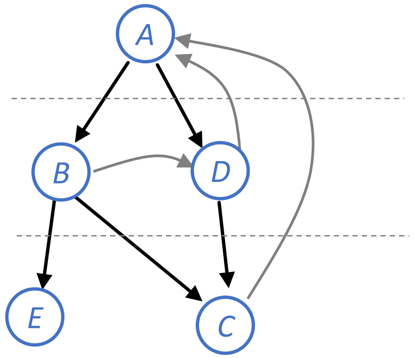
Breadth-first search

Start vertex *A*





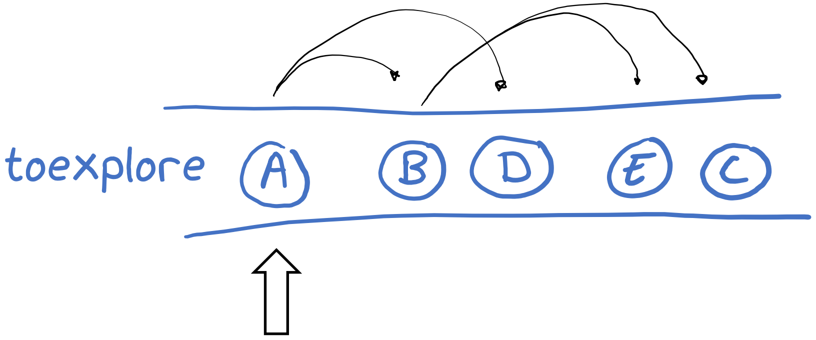
distance from *A* = 0

distance from *A* = 1

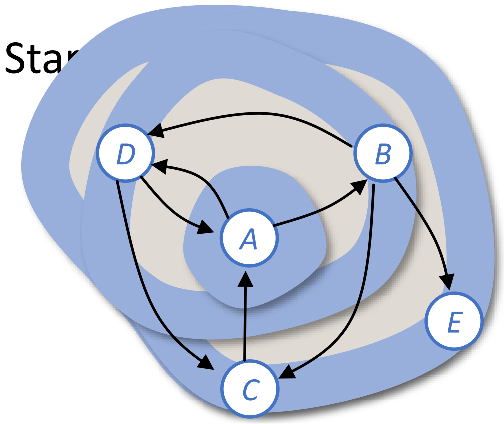
distance from *A* = 2

```
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Queue([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popleft()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```

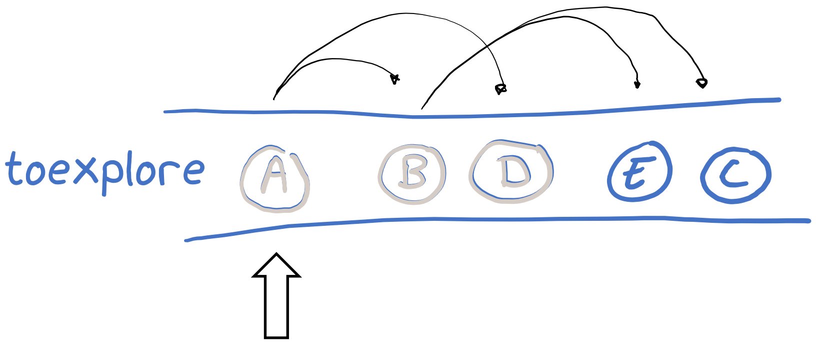


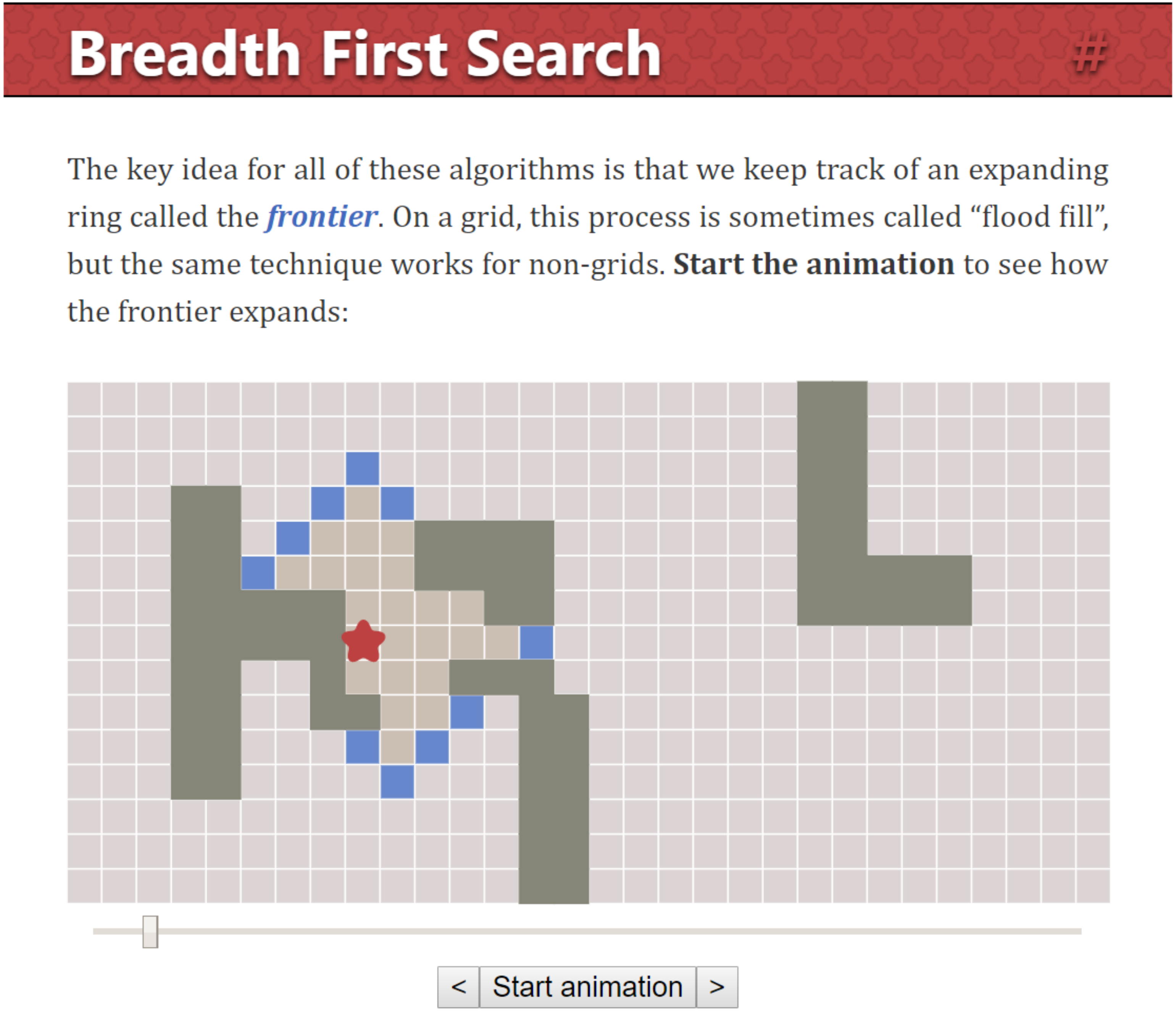
Breadth-first search



```
# Visit all the vertices in g reachable from start vertex s
def bfs(g, s):
    for v in g.vertices:
        v.seen = False
    toexplore = Queue([s])
    s.seen = True

    while not toexplore.is_empty():
        v = toexplore.popleft()
        for w in v.neighbours:
            if not w.seen:
                toexplore.pushright(w)
                w.seen = True
```





Breadth First Search

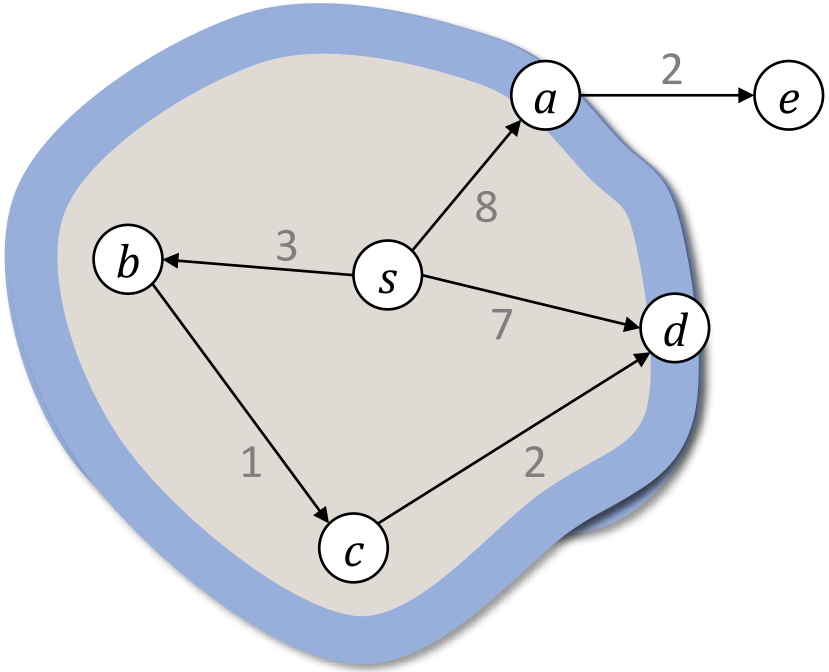
The key idea for all of these algorithms is that we keep track of an expanding ring called the ***frontier***. On a grid, this process is sometimes called “flood fill”, but the same technique works for non-grids. **Start the animation** to see how the frontier expands:

https://www.redblobgames.com/pathfinding/a-star/introduction.html#breadth-first-search

SECTION 5.3

Dijkstra’s algorithm

In a graph where the edges have costs (e.g. travel time), we can find shortest paths by using a similar “grow the frontier” algorithm to bfs.



```
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)
```

"relaxing" the v → w edge.

popped	toexplore
{}	[s]
{s}	[b, d, a]
{s, b}	[c, d, a]
{s, b, c}	[d, a]

s.distance = 0

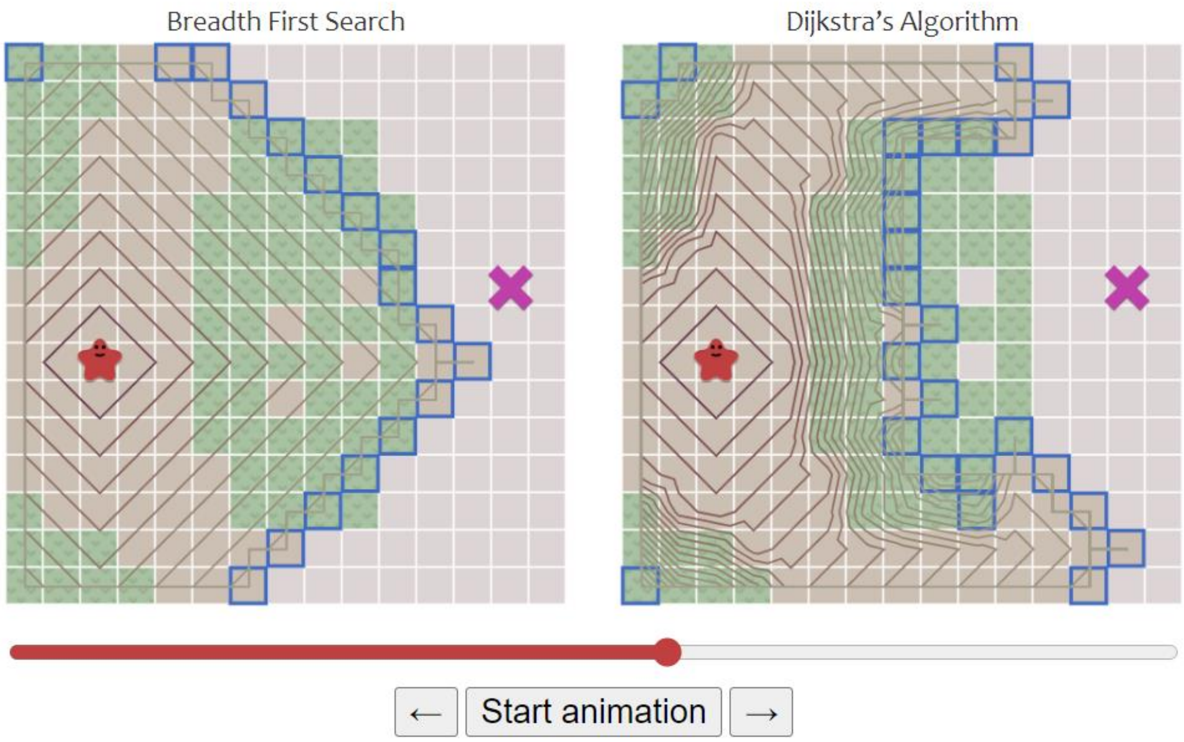
b.distance = 3 d.distance = 7 a.distance = 8

c.distance = 3+1 = 4

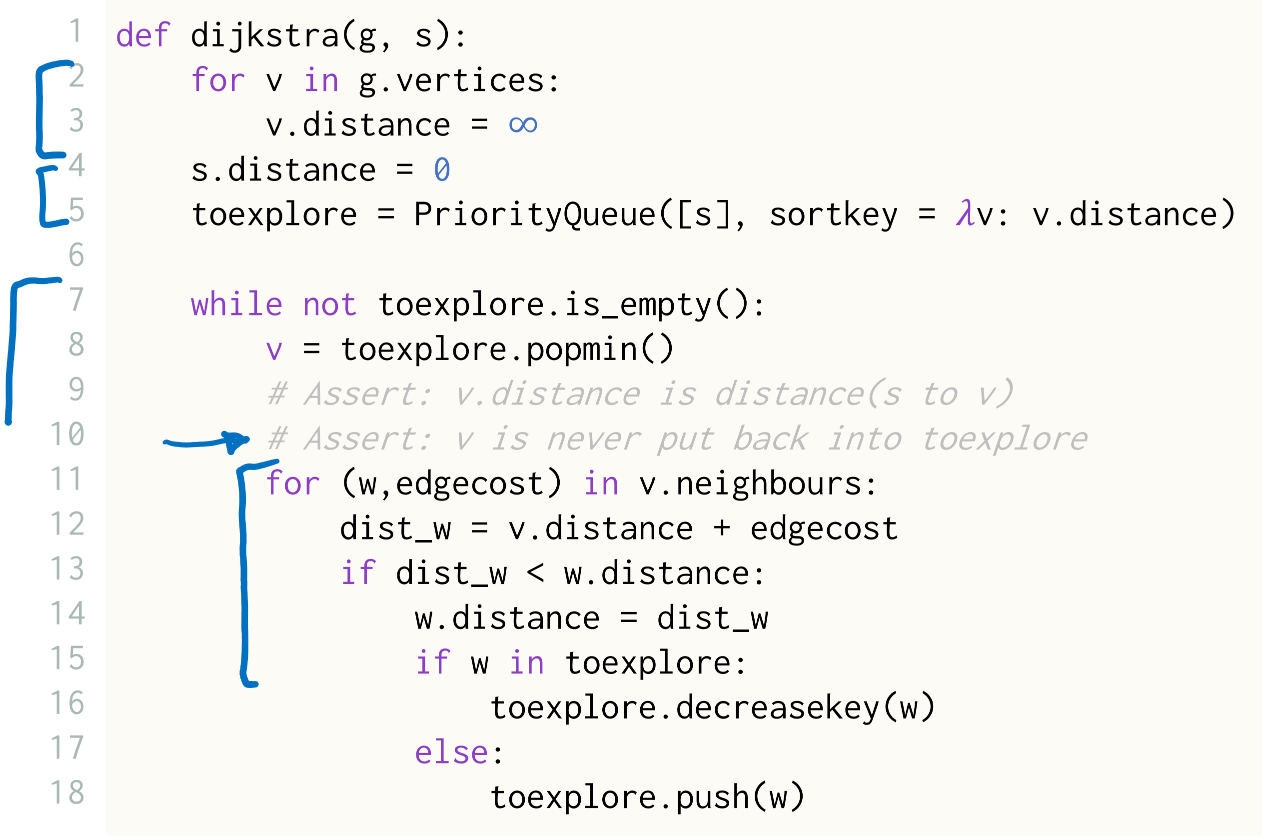
d.distance = 6 I "relaxed" the c → d edge.

Movement costs

So far we’ve made steps have the same “cost”. In some pathfinding scenarios there are different costs for different types of movement. We’d like the pathfinder to take these costs into account. Let’s compare the number of steps from the start with the distance from the start:



https://www.redblobgames.com/pathfinding/a-star/introduction.html#dijkstra



```
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)
```

$O(V)$

$O(1)$

$O(V)$ passes by Ass 10.

$O(E)$ passes, in aggregate

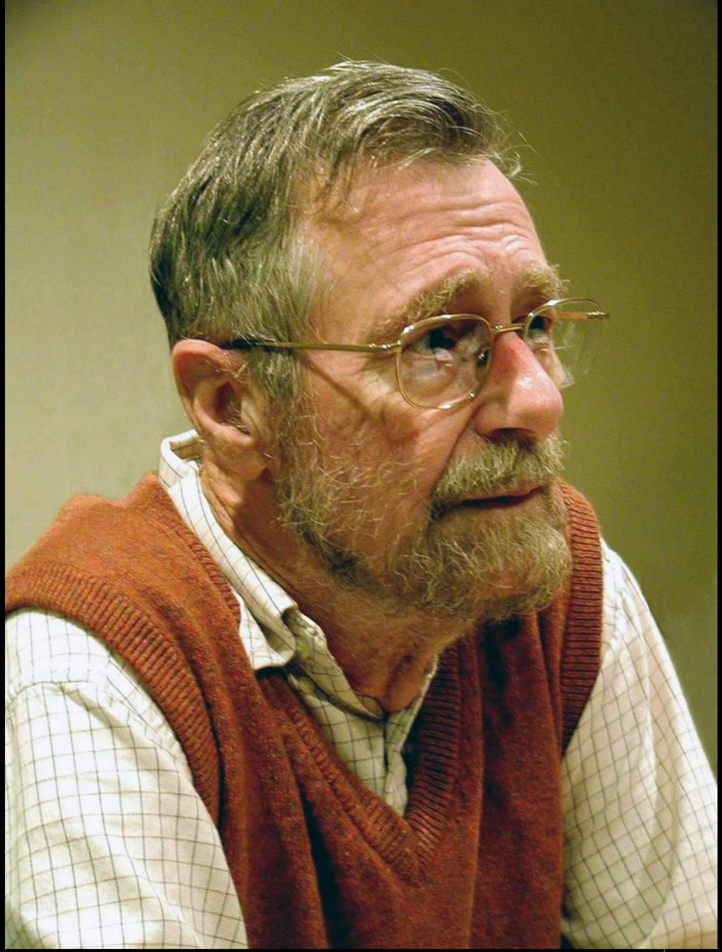
$$\text{Total} = O(V) + O(V) \times \underset{O(\log n)}{C_{\text{popmin}}} + O(E) \times \underset{O(1)}{C_{\text{push/dec.key}}}$$

where $n = \#$ items stored

$n \leq V$ by line 10

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

> 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

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, where …

$\text{cost}(u \to v)$ is the *cost* associated with edge $u \to v$

$\text{cost}(u \to \cdots \to v)$ is the sum of edge costs along the path $u \to \cdots \to v$

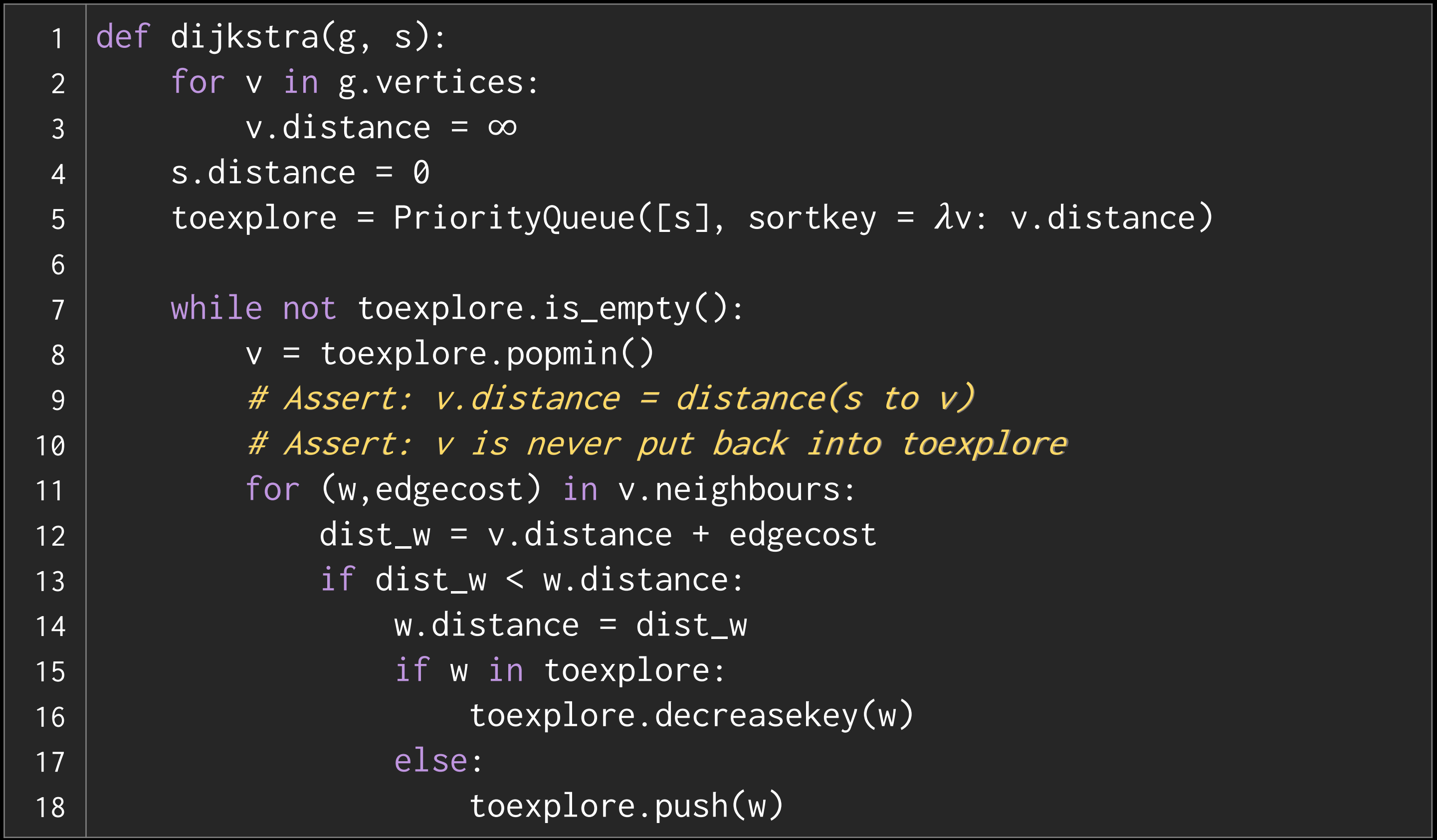
$$\text{distance}(u \text{ to } v) = \begin{cases} \text{min cost of any path } u \to \cdots \to v, & \text{if one exists} \\ 0, & \text{if } u = v \\ \infty, & \text{otherwise} \end{cases}$$

Theorem.

i. On a finite graph, the algorithm terminates

ii. When it does, for every vertex v, $\overbrace{v.\text{distance}}^{\text{computed}} = \overbrace{\text{distance}(s \text{ to } v)}^{\text{Platonic/mathematical}}$

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 = 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)
```

Theorem.

i. On a finite graph, the 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 i.e., just after v is popped,

(9) $v.\mathtt{distance} = \text{distance}(s \text{ to } v)$ and (10) v is never put back into `toexplore`

Proof of (i)

vertices can never be put back into P.Q. (by Ass. 10)

And V is finite (by assumption)

∴ terminates

Proof of (ii)

By Ass. 9, v.distance = dist(s to v) just after v is popped

RTP :
- v.distance doesn't change subsequently
- every vertex reachable from s is eventually popped.
  (and vertices not reachable never have distance set.)

EXERCISE

Assertion (line 9).

Just after a vertex v is popped, v.distance = distance(s to v)

CLAIM: this assertion never fails.

PROOF: suppose it fails at some point in execution. let v be the vertex for which it first fails. let T be the instant it first fails.

Consider a $\overbrace{\text{shortest path}}^{\text{maths object}}$ from s to v:

$$\overset{s}{\overset{\shortparallel}{u_1}} \to u_2 \to \cdots \to \overset{v}{\overset{\shortparallel}{u_k}}$$

CASE1: there is some vertex on this path that hasn't been popped by time $\le T$.

let i be the index of the first such vertex; the path is

$$\underbrace{\overset{s}{\overset{\shortparallel}{u_1}} \to u_1 \to \cdots \to u_{i-1}}_{\text{have been popped}} \to \underset{\text{not yet popped}}{\underset{|}{u_i}} \to \underbrace{\cdots}_{???} \to \overset{v}{\overset{\shortparallel}{u_k}}$$

Then, we obtain a contradiction ⨳
[see next two slides]

CASE2: all vertices on this path have been popped by time $\le T$.

By a similar argument, this leads to ⨳.

So our initial supposition (that the assertion fails at some point in execution) is false.

$\overbrace{\text{dist}(s \text{ to } v)}^{\text{maths object}}$

$< v.\text{distance}$

LEMMA: If the alg. sets $w.\text{distance} = x$ for some vertex w, then $\exists$ path from s to w of cost x.

PROOF: by easy induction.

Thus, $\text{dist}(s \text{ to } v) \overset{\Delta}{=} \min_{\text{paths } p: s \rightsquigarrow v} \text{cost}(\text{path } p) \leq v.\text{distance}$ by the lemma.

But we're supposing that the assertion failed, ie $v.\text{distance} \neq \text{dist}(s \text{ to } v)$.

So the inequality is strict: $\text{dist}(s \text{ to } v) < v.\text{distance}$.

$\leq u_i.\text{distance}$

We're using a priority queue (PQ) we just popped v.

Also, u_i was in the PQ (u_{i-1} ensured it's in there, and by choice of i it hasn't been popped yet)

Thus, by definition of PQ, $v.\text{distance} \leq u_i.\text{distance}$.

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

When we popped u_{i-1}, we relaxed all its edges including $u_{i-1} \to u_i$.

ie we forced $u_i.\text{distance} \leq u_{i-1}.\text{distance} + \text{cost}(u_{i-1} \to u_i)$.

let's check: is this inequality still true at time T?

- For any vertex w, the alg. can only ever decrease $w.\text{distance}$, it can't increase it.
- The RHS can't have changed since u_{i-1} was popped, because $u_{i-1}.\text{distance}$ was correct when we popped it (by induction hypothesis, ie the hypothesis that Assertion 9 didn't fail before T) and it can't decrease any further.
- The LHS might have decreased in the interim,

So this inequality remains true at time T.

PRECEDING SLIDE:

$\text{dist}(s \text{ to } v) < v.\text{distance} \le u_i.\text{distance} \le u_{i-1}.\text{distance} + \text{cost}(u_{i-1} \to u_i)$

Continuing.

$u_{i-1}.\text{distance} + \text{cost}(u_{i-1} \to u_i)$

$= \text{dist}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i)$

by induction hypothesis —
i.e. we're assuming Assertion 9 <u>first</u> failed for v,
so it didn't fail for u_{i-1}. so $u_{i-1}.\text{distance}$ was correct
when u_{i-1} was popped;
and as noted above it can't change thereafter

$\le \text{dist}(s \text{ to } v)$

We chose this path $\overset{s}{\underset{u_1}{\shortparallel}} \to \cdots \to \overset{v}{\underset{u_k}{\shortparallel}}$ to be a shortest path from s to v. Thus

$\text{dist}(s \text{ to } v) = \text{cost}(\text{this path}) \ge \text{cost}(u_1 \to \cdots \to u_{i-1}) + \text{cost}(u_{i-1} \to u_i)$

[by definition of path cost, and the fact that costs are ≥ 0.

Also, by definition of distance, as "min cost over all paths",

$\text{cost}(u_1 \to \cdots \to u_{i-1}) \ge \text{dist}(s \text{ to } u_{i-1})$.

Putting these inequalities together, $\text{dist}(s \text{ to } v) \ge \text{dist}(s \text{ to } u_{i-1}) + \text{cost}(u_{i-1} \to u_i)$.

In summary,

$\text{dist}(s \text{ to } v) < v.\text{distance} \le \cdots\cdots \le \text{dist}(s \text{ to } v)$.

But it's impossible to have $\text{dist}(s \text{ to } v) < \text{dist}(s \text{ to } v)$ — a contradiction ⨳.

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)
```

Not covered in the lecture —
but pretty easy to prove, now that we've seen the proof of Ass. line 9.

PROOF

1. The condition on line 13 ensures that a vertex w is only pushed into the priority queue when we discover a path shorter than $w.\texttt{distance}$
2. Once v is popped, $v.\texttt{distance} = \text{distance}(s \text{ to } v)$ (by the assertion on line 9), so there can be no shorter path, by definition of “distance”.

Hence v is never pushed back.