Introduction to Algorithms and Data Structures

Lecture 16: Dijkstra's Algorithm (for shortest paths)

Mary Cryan

School of Informatics
University of Edinburgh

Welcome back!

Directed and Undirected Graphs

We return to the world of graphs and directed graphs.

- A *graph* is a mathematical structure consisting of a set of *vertices* and a set of *edges* connecting the vertices.

  Formally: $G = (V, E)$, where V is a set and $E \subseteq V \times V$.

- $G = (V, E)$ *undirected* if for all $v, w \in V$:

  $$(v, w) \in E \iff (w, v) \in E.$$

  Otherwise *directed*.

  Directed $\sim$ *arrows* (one-way)
  Undirected $\sim$ *lines* (two-way)

Road Networks

The weighted case is a very natural graph model - eg, road network where vertices represent intersections, edges represent road segments, and the weight of an edge is the distance of that road segment.

Shortest paths in graphs

In this lecture we will consider *weighted* graphs (and digraphs) $G = (V, E)$ where there is a weight function $w : E \to \mathbb{R}$ defining weights for all arcs/edges.

We are interested in evaluating the cost of shortest paths (from specific node u to specific node v) in the given weighted graph.

> *We will focus on single-source shortest paths, where we want to find the minimum path from node s to node v, for every v.*

Single-source shortest paths

unweighted graphs and digraphs

We can use breadth-first search to explore a graph $G = (V, E)$ from a specific vertex $s \in V$. $\Theta(|V| + |E|)$ running-time.

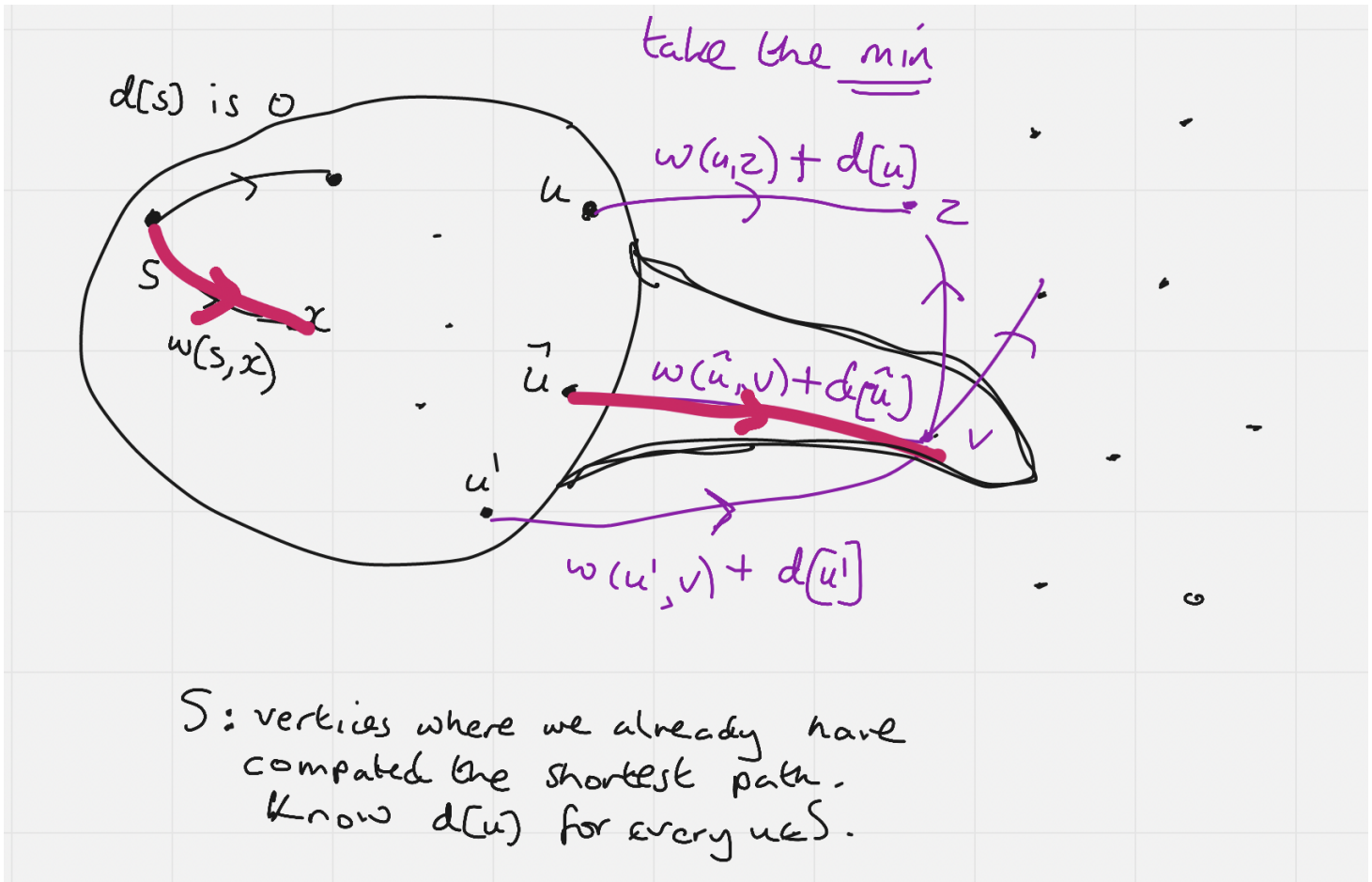
(in the unweighted case, the shortest path is the one with fewest edges)

weighted graphs and digraphs

- Dijkstra's algorithm, will compute the single source shortest paths (and their values) for any graph or directed graph without negative weights.
- Dijkstra's Algorithm is a greedy algorithm.
- With the use of a (Min) Heap to deliver the Priority Queue, Dijkstra's can achieve running-time $O((|V| + |E|) \lg(|V|))$, or $O((m + n) \lg(n))$.

Dijkstra's Algorithm

A Greedy algorithm which starts with the source s ($S \leftarrow \{s\}$) and at each step, adds the fringe vertex $v \in V \setminus S$ with the shortest candidate path into S.



Dijkstra's Algorithm

Input: Graph $G = (V, E), w : E \to \mathbb{R}^+$ is a weighted graph/digraph (no negative weights), $s \in V$ a distinguished source vertex.

Output: Arrays d and π of length $n = |V|$ each:
$d[v]$ to hold shortest-path distance $\delta(s, v)$ from s to v,
$\pi[v]$ to be v's predecessor along that shortest path.

1. Initialise $d[v] \leftarrow \infty, \pi[v] \leftarrow \text{NIL}$ for every $v \in V$.
2. Initialise $S = \{s\}, d[s] \leftarrow 0$.
3. while S still has fringe edges, we update as follows:

   - Consider the current fringe edges $(u, v) \in E$ with $u \in S, v \in V \setminus S$
   - Let $v^* \in V \setminus S$ be the fringe vertex with minimum $d[u^*] + w(u^*, v^*)$ (minimum over all fringe edges).
   - Assign $d[v^*] \leftarrow d[u^*] + w(u^*, v^*)$, $\pi[v^*] \leftarrow u^*$, update $S \leftarrow S \cup \{v^*\}$.

until we arrive at the point where S has no fringe edges, and we terminate. (happens after $n - 1$ or fewer iterations.)

Dijkstra's Algorithm: proof of correctness

claim: Before any iteration of 3., for every $u \in S$, $d[u]$ contains the shortest path value $\delta(s, u)$ in G, and $\pi[u]$ is the predecessor vertex to u along a shortest path.

proof: (by induction)

Base case: After the initialisation in 1. and 2, we have $S = \{s\}$ and $d[s] = 0$, as it should be ($\delta(s, s) = 0$ always). $\pi[s] = \text{NIL}$ is correct for the source s.

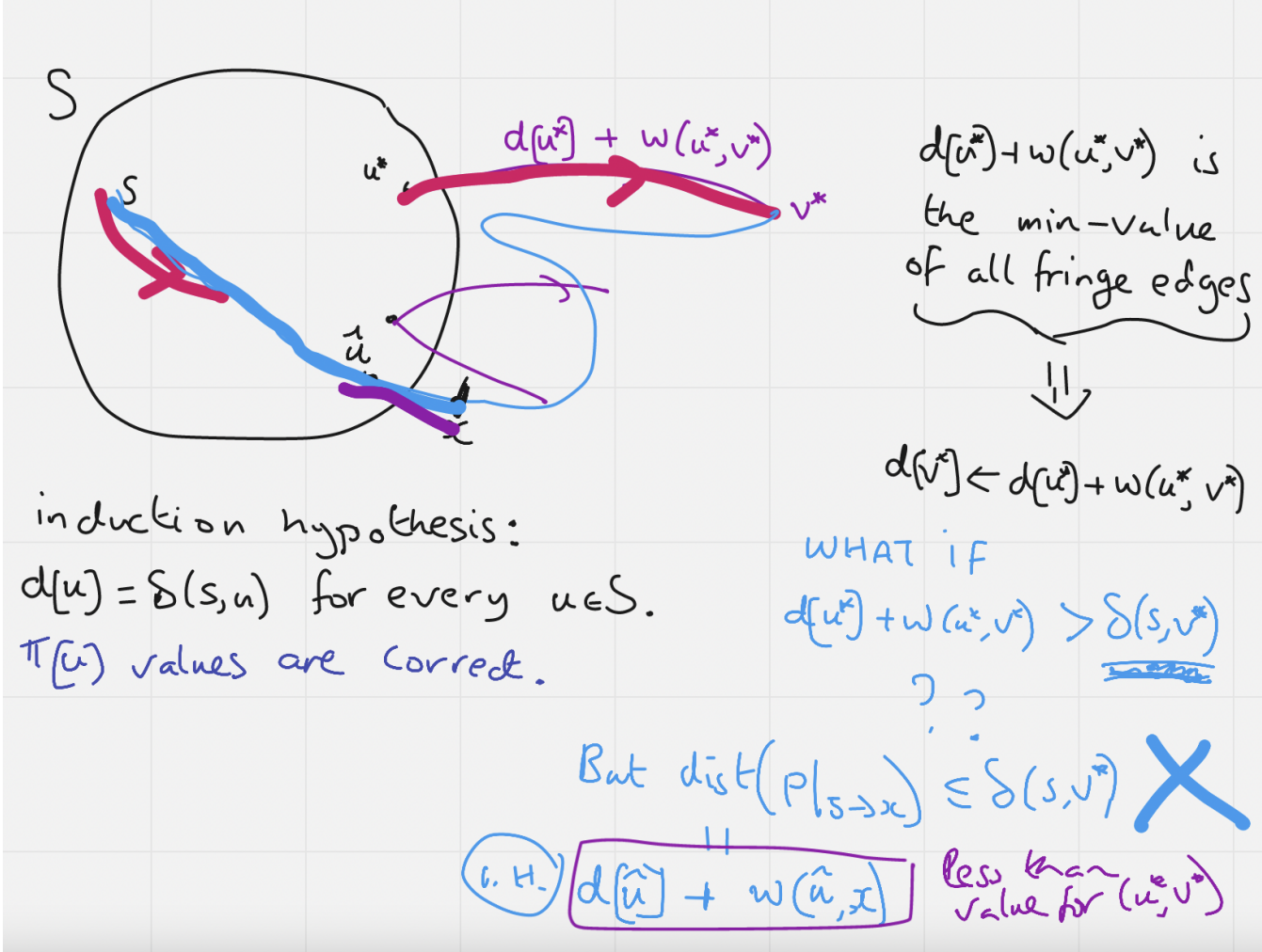
Induction step: Suppose that the claim holds for S, and that S has fringe edges. Let (u^*, v^*) be the fringe edge generating the minimum $d[u^*] + w(u^*, v^*)$ in 3.. Now suppose that $d[u^*] + w(u^*, v^*) > \delta(s, v^*)$ for v^*? (if not, we are done) *(we will derive a contradiction).* Let p be a path of distance $\delta(s, v^*)$ from s to v^* in G, and let $(\widehat{u}, x)$ be the first fringe edge wrt S on p. The prefix path $p_{|s \to x}$, ending in $(\widehat{u}, x)$, has total distance *at most* $\delta(s, v^*)$ (G has no negative weights). Hence $(\widehat{u}, x)$ is a fringe edge wrt S with

$$d[\widehat{u}] + w(\widehat{u}, x) < d[u^*] + w(u^*, v^*).$$

CONTRADICTION to our choice of (u^*, v^*)!! Hence $d[u^*] + w(u^*, v^*) = \delta(s, v^*)$ must be true, and $\pi[v^*] \leftarrow u^*$ is also correct.

Dijkstra's Algorithm: proof of correctness

Picture showing what we mean for the inductive step of the proof.



Recovering the shortest paths

(in a graph/digraph with non-negative weights)
In practice, we will want the short paths themselves, not just the values.

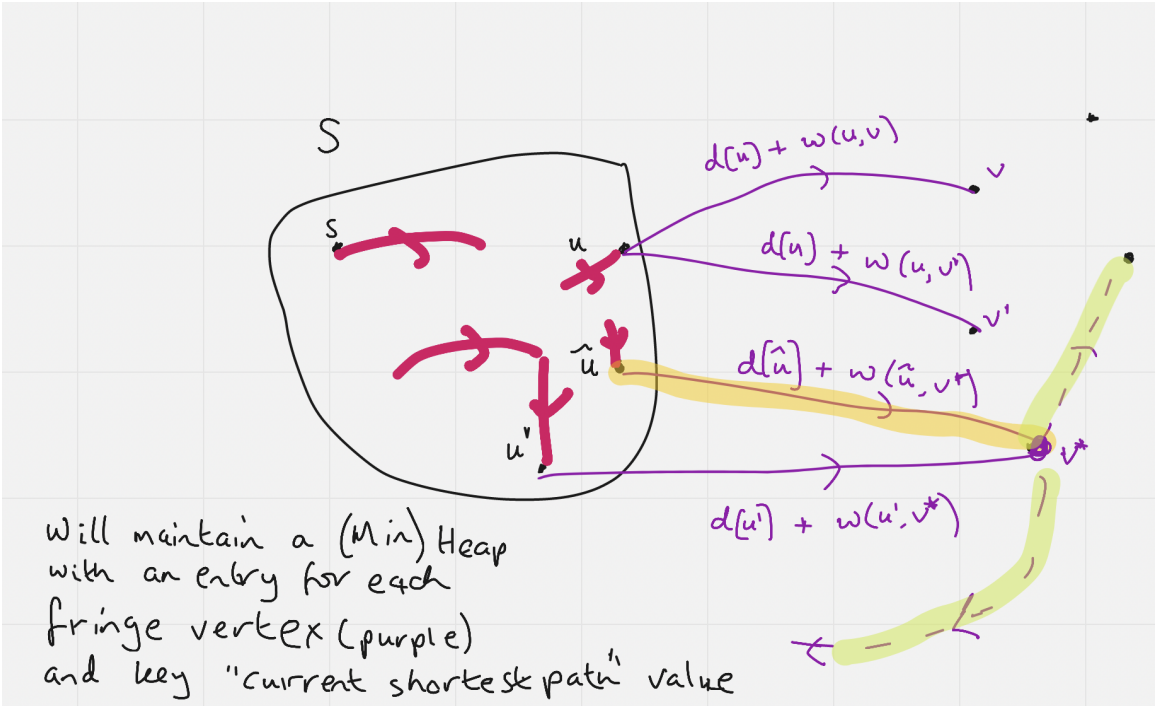
Some facts that help us:

- No shortest path from s to any v need contain a cycle.
  why?: If a path p contains a cycle, cycle's weight is ≥ 0, we could delete it to get another $s \rightarrow v$ with fewer edges, and distance no greater.
- Every shortest path has at most $n-1$ edges.
  why?: no cycles, so can visit any node at most once.
- If $s = v_0, v_1, \ldots, v_k$ is a shortest path to v_k, then every prefix $s = v_0, v_1, \ldots, v_i$ is a shortest path to v_i.

  why?: If we had a shorter path for one of the v_i, we could replace section $s = v_0, v_1, \ldots, v_i$ to get a shorter path for v_k too.

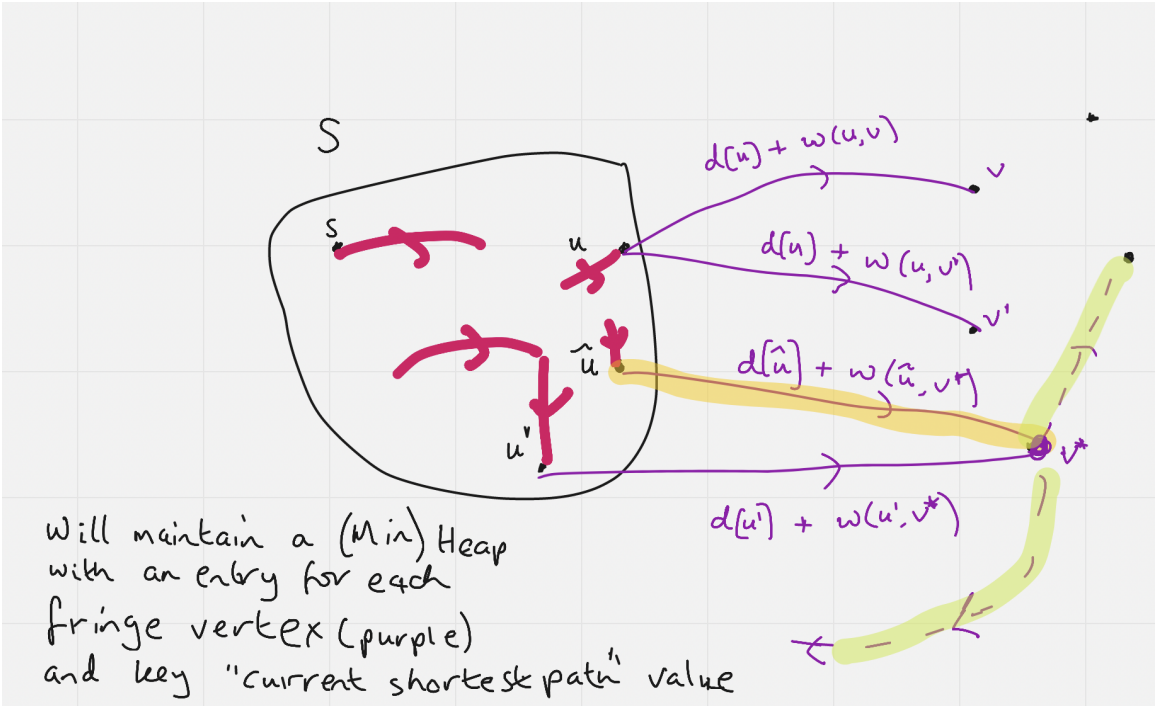
The third point allows us to use the π array to recursively build the short path for any $v \in S$ (lookup $\pi[v]$ to get last edge $(\pi[v], v)$, lookup $\pi[\pi[v]]$, ...)

Towards an efficient implementation



- "All about" the ranking/management of the fringe edges/vertices.
- Will store each fringe vertex v with its "shortest so far" $d[\cdot] + w(\cdot, v)$ value.
- Change (when $S \leftarrow S \cup \{v^*\}$) is limited to the edges round v: $(\widehat{u}, v^*)$ will "drop out" and the $(v^*, \cdot)$ edges next considered.

Towards an efficient implementation



- Adjacency list rep. for G - visit "outgoing edges from v" in $O(outdeg(v))$.
- (Min) Heap of current fringe vertices, with their current shortest path value (so far) as key. (we will need to be able to *update/reduce* keys, after a successful Relax operation).
- Q.removeMin() $\Leftrightarrow$ "add the best fringe vertex v" to S.

Implementation using (Min) Heap

Algorithm InitializeSingleSource(G, s)

1. **for** each vertex $v \in V[G]$
2. **do** $d[v] \leftarrow \infty$
3. $\pi[v] \leftarrow \text{NIL}$

Algorithm Relax$(G, (u, v))$

1. **if** $d[v] = \infty$
2. **then** $d[v] \leftarrow d[u] + w(u, v)$
3. $\pi[v] \leftarrow u$
4. Q.insertItem$(d[v], v)$
5. **if**$(d[v] > d[u] + w(u, v))$
6. **then** $d[v] \leftarrow d[u] + w(u, v)$
7. $\pi[v] \leftarrow u$
8. Q.reduceKey$(d[v], v)$

Implementation using (Min) Heap

Edsger Dijkstra

Algorithm Dijkstra(G, s)

1. InitializeSingleSource$(G,\ s)$
2. Q.insertItem$(0, s)$
3. $d[s] \leftarrow 0$
4. **while** $\neg$(Q.isEmpty())
5. **do** $(d^*, u) \leftarrow$ Q.removeMin()
6. **for** $x \in Out(u)$
7. Relax$(G, (u, x))$

(Min) Heaps

In Lectures 11 we saw how we can use a Heap to implement a Priority Queue with n items, so operations have the following worst-case running-times:

Q.isEmpty()	$\Theta(1)$
Q.minElement()	$\Theta(1)$
Q.removeMin()	$O(\lg(n))$
Q.insertItem(d,v)	$O(\lg(n))$
Q.reduceKey(d',v)	$O(\lg(n))$

Strictly speaking, we demonstrated this for a Max Heap - however, by exchanging $>$ and $<$ we can transform a Max Heap implementation into a Min Heap structure, same running-times.

updates: We can also add the operation Q.reduceKey(d', v) (to replace v's current key by a smaller d') to operate in $O(\lg(n))$ worst-case time.

- Use the "bubble up" of insertItem, but may start higher than a leaf.
- (we assume we have an index supporting jumps to v's cell of the Heap)

Running-time analysis

- InitializeSingleSource takes $O(n)$ time at most.
- lines 2.-3. take $O(1)$.
- *Take an "aggregated" approach to bounding run-time of the* **while**
- A vertex v can be *added* to the Heap only once (need $d[v] = \infty$ in Relax) and hence, only removed once

  $\Rightarrow O(n.\lg(n))$ covers *all* the Q.removeMin() and Q.insertItem(d, v) calls.
- *Apart* from the insertItem calls, a call to Relax takes

  $O(1) + T_{\text{reduceKey}}(n) = O(1) + O(\lg(n))$ time.
- We might call Relax at most *twice* for every edge $e \in E$ ... as we only call Relax$(G, (u, v))$ immediately after an endpoint has joined S.

  Hence total for *all* Relax calls is $O(m + m \cdot \lg(n))$ time.
- Other work done by the **while** is at most $O(n)$.

$O((n + m) \lg(n))$ time overall

Reading

- Graph representations in Section 22.1
- Chapter 24: shortest-paths and their representation (pages 580-586)
- Section 24.3 on "Dijkstra's Algorithm" and Section 24.5 with some proofs.
  (our version of Dijkstra is *slightly* different to [CLRS], as we only add vertices to the Queue when they are adjacent to S)