Design and Analysis of Algorithms

Part 6

Paths in Graphs

Giulio Chiribella

Hilary Term 2019

How to find the optimal route?

In many applications, like GoogleMaps, it is important to find the shortest route from one location to another.
More generally, going from one location to another could have a *cost* and it is important to find the route that minimizes the cost.

Graph formulation:

- ☐ locations $\rightarrow$ vertices
- ☐ connections between two locations $\rightarrow$ edges
- ☐ cost of going from one location to another $\rightarrow$ weight on the edge
- ☐ route $\rightarrow$ path
- ☐ cost of a route $\rightarrow$ sum of the weights of the edges on the path.

Shortest paths

Consider a directed graph $G = (V, E)$ with ***weight function*** (or *length*) $w : E \longrightarrow \mathbb{R}_{\geq 0}$.

- The ***weight*** (or *length*) ***of a path*** $p = \langle v_0, v_1, \cdots, v_k \rangle$ is

$$w(p) \ := \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

  I.e. $w(p)$ is the sum of edge weights on path p.

- The ***shortest-path weight*** (or *length*) from vertex u to vertex v is

$$\delta(u, v) := \begin{cases} \min\{ w(p) : p \text{ is a path from } u \text{ to } v \} & \text{if } \exists \text{ path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A ***shortest path*** from u to v is a path p such that $w(p) = \delta(u, v)$.

Breadth-first search [CLRS 22.2]

Consider the simplest case: ***all weights equal to 1.***

$\delta(u, v) =$ minimum number of edges on a path from u to v, if such a path exists; otherwise, $\delta(u, v) = \infty$.

Given a source vertex s, ***breadth-first search (BFS)*** finds all the vertices reachable from s and, for each reachable vertex v it finds the shortest-path length $\delta(s, v)$ and a shortest path from s to v.

Input: A **graph** $G = (V, E)$, either directed or undirected, and a **source vertex** $s \in V$.

Output: For each $v \in V$, an integer $d[v]$ and a back pointer $\pi[v]$, such that

1. $d[v] = \delta(s, v)$
2. $\pi[v] = u$ is the predecessor of v on a shortest path from s to v. If there is no path from s to v, then $\pi[v] = \text{NIL}$.

Intuitive idea

Idea: Send a wave out from the source s.

- □ The wave first hits all vertices 1 edge away from s.
- □ From there, the wave will then hit all vertices 2 edges away from s.
- □ When the wave reaches v, we know the distance from s to v.
  The shortest path is constructed by keeping track of the vertices reached by the wave on the way to v.

Data structure implementation: the wave can be implemented by a ***first-in-first-out (FIFO) queue*** Q**.**

A vertex v is put in the queue when it is reached by the wave,
and it is removed by the queue after all its neighbours are reached.

FIFO queues [CLRS 10.1]

A ***FIFO queue*** is an *abstract data structure* with three basic operations:

1. $\textsc{Enqueue}(Q, x)$: Inserts x at the *end* of the queue Q.
2. $\textsc{Dequeue}(Q)$: Returns and removes the item at the *head* of the queue Q.
3. $\textsc{IsEmpty}(Q)$: Returns whether or not the queue is empty. In the pseudo code we will use the notation $Q = \emptyset$ and $Q \neq \emptyset$.

Implementation:

Linked list with an extra "END" pointer to the tail of the list.
The head of the queue corresponds to the head of the linked list.
This makes all operations $O(1)$.

The BFS algorithm [CLRS 22.2]

BFS(V, E, s)

Input: A directed or undirected graph (V, E), and a source $s \in V$.
Output: For each $v \in V$, $d[v]$ and $\pi[v]$

$d[s] = 0; \pi[s] = nil$ // Source can be reached with path of length 0
for each $u \in V - \{ s \}$ // Mark all other nodes
    $d[u] = \infty$ // as not reached yet
    $\pi[u] = nil$
$Q = \emptyset$ // Initialise Q
ENQUEUE(Q, s) // to contain only source s
while $Q \neq \emptyset$
    $u =$ DEQUEUE(Q) // Node u is finished
    for each $v \in Adj[u]$
        if $d[v] = \infty$ // If v is reached for the first time.
            $d[v] = d[u] + 1$ // record shortest distance,
            $\pi[v] = u$ // create backpointer,
            ENQUEUE(Q, v) // and join queue of unfinished nodes

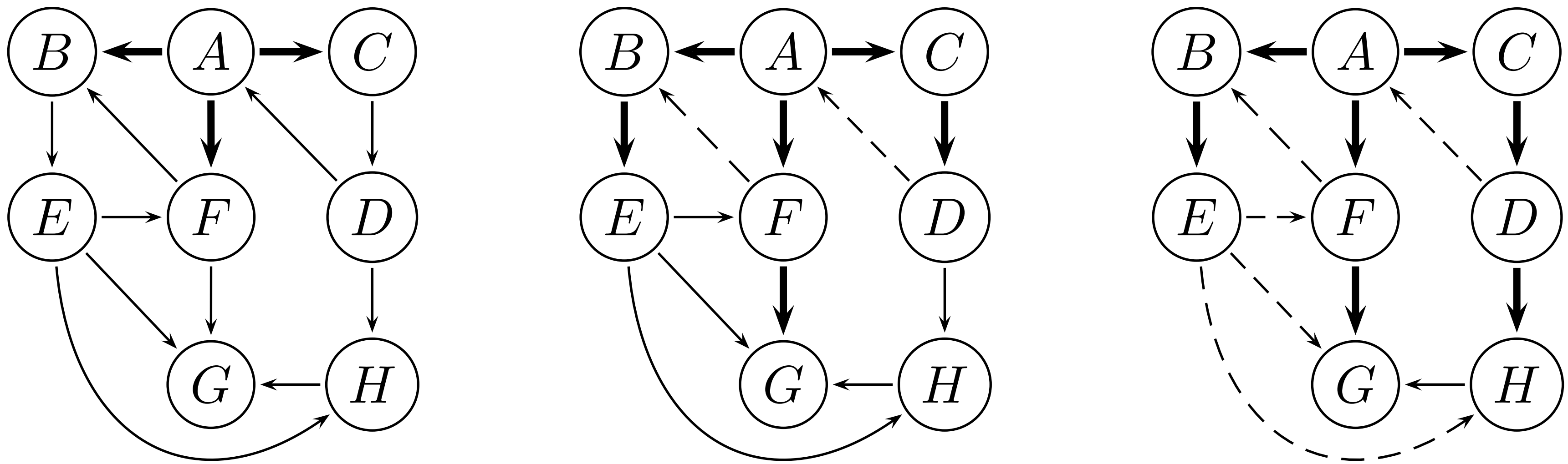
Running time

BFS takes $O(|V| + |E|)$ in total.

- $O(|V|)$ because of the initialization step and of the enqueue/dequeue operations (each vertex is enqueued/dequeued at most once)
- $O(|E|)$ because we examine the edge (u, v) only when u is dequeued. Hence every edge is examined at most once (or at most twice if the graph is undirected and we represent edges as unordered pairs $\{u, v\}$).

Example

Consider the following graph with source $s = A$.



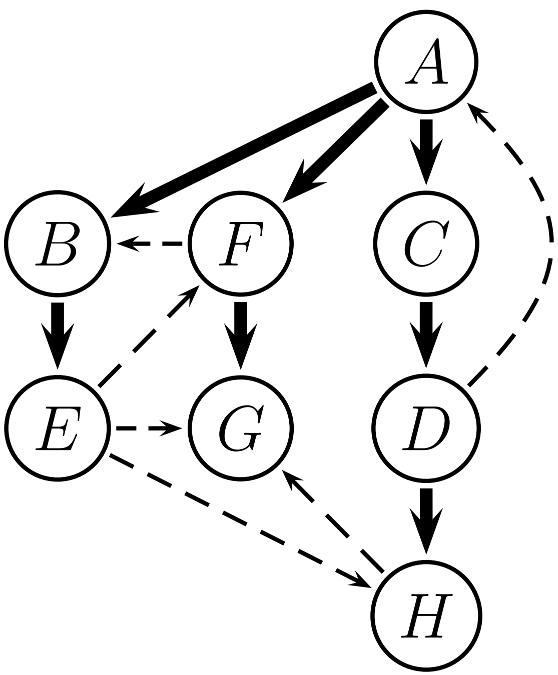
First, the wave hits vertices $\{ B, C, F \}$.
Second, it hits $\{ E, D, G \}$.
Third, it hits $\{ H \}$.
At the fourth step, the wave does not discover any new nodes.

The BFS tree

The ***BSF tree**** consists of the **vertices reachable from** s and has an **edge between** u **and** v **iff** $u = \pi[v]$**.**
Explicitly, the BSF tree is the graph $G_\pi := (V_\pi, E_\pi)$ with
$V_\pi := \{u \in V \ : \pi[u] \neq \text{NIL}\} \cup \{s\}$ and $E_\pi := \{(\pi[u], u) \ : u \in V_\pi\}$.

* note the usual abuse of notation: in math, trees are *undirected graphs*.

Example (cont'd)



Note: $d[v]$ is the level of v in the BFS tree.

Note: the BFS tree depends on the order in which adjacent nodes are explored, for example, if E came before D, the edge (E, H) would be in the tree instead of (D, H).

Lower bound on $d[v]$

Lemma 1 If $d[v] < \infty$, there exists a path of length $d[v]$ from s to v.

Proof. By induction on $d[v]$.
Base case. If $d[v] = 0$, then v must be the source s. Hence, Lemma 1 trivially holds.
Induction step. Suppose that, for every v with $d[v] \leq d_0$, there exists a path of length $d[v]$ from s to v. Suppose that v is such that $d[v] = d_0 + 1$. Then, let $u = \pi[v]$ be the predecessor of v. By construction $d[v] = d[u] + 1$, and therefore $d[u] = d_0$.
Now, the induction hypothesis guarantees that there exists a path of length $d[u]$ from s to u. Adding the edge (u, v) to this path, we obtain a path of length $d[u] + 1 = d[v]$ from s to v. $\square$

Corollary 1 (lower bound on d). $d[v] \geq \delta(s, v)$ for every vertex v.

Properties of the queue [Lemma 22.3, p. 599 of CLRS]

Lemma 2 Let $Q = \langle v_1, \ldots, v_r \rangle$ be the queue at a given step of BFS. Then, $d[v_r] \leq d[v_1] + 1$ and $d[v_i] \leq d[v_j]$ for every i and $j \in \{1, \ldots, r\}$.

Proof. By induction on the total number n of ENQUEUE/DEQUEUE operations. *Base case.* For $n = 1$, only vertex s is been enqueued, and Lemma 2 trivially holds.

Induction step. Suppose that Lemma 2 holds for $n \leq n_0$ queue operations. If a DEQUEUE operation is performed, Lemma 2 still holds. If a ENQUEUE operation is performed, let v be the vertex that is being enqueued, and let $u = \pi[v]$ be its predecessor. Note that $d[u] \leq d[v_1]$, since there are two possibilities: either (1) $\pi[v_1] = u$, in which case $d[u] = d[v_1] - 1$, or (2) $\pi[v_1] \neq u$, in which case u and v_1 must have been both in the queue at earlier time, and induction implies $d[u] \leq d[v_1]$. Now, ENQUEUE sets $v_{r+1} := v$ and $d[v_{r+1}] := d[u] + 1$. Hence, $d[v_{r+1}] = d[u] + 1 \leq d[v_1] + 1$. It remains to show $d[v_r] \leq d[v_{r+1}]$. Again, there are two possibilities: (1) $\pi[v_r] = u$, in which case $d[v_r] = d[u] + 1 = d[v_{r+1}]$, and (2) $\pi[v_r] \neq u$, in which case u and v_r must have been both in Q at an earlier time, and induction implies $d[v_r] \leq d[u] + 1 = d[v_{r+1}]$.

Upper bound on $d[v]$

Corollary 2. If u is enqueued before v, then $d[u] \leq d[v]$.

Proof. Immediate from Lemma 2.

Lemma 3 (upper bound on d). If there exists a path of length l from s to v, then $d[v] \leq l$.

Proof. By induction on l. *Base cases.* For $l = 0$, $v = s$ and $d[s] = 0$. For $l = 1$, the path is an edge (s, v), and therefore $d[v] = 1$.
Induction step. Suppose that Lemma 3 holds for paths of length $l \leq l_0$, and suppose that $\langle\, s, v_1, \ldots, v_{l_0}, v \,\rangle$ is a path of length $l_0 + 1$ from s to v. By the induction hypothesis, $d[v_{l_0}] \leq l_0$. Now, there are three possibilities:
(1) v is enqueued before v_{l_0} is enqueued. Then, Corollary 2 implies $d[v] \leq d[v_{l_0}] \leq l_0$.
(2) v is enqueued before v_{l_0} is dequeued. Then, Lemma 2 implies $d[v] \leq d[v_{l_0}] + 1 \leq l_0 + 1$.
(3) v is enqueued after v_{l_0} is dequeued. Then, $\pi[v] = v_{l_0}$ and $d[v] = d[v_{l_0}] + 1 = l_0 + 1$. $\square$

Correctness of BFS

Theorem 1. $d[v] = \delta(s, v)$ for every vertex $v \in V$.

Proof. Suppose that $\delta(s, v)$ is finite. Then, there exists a path of length $\delta(s, v)$ from s to v. Hence, Lemma 3 yields $d[v] \leq \delta(s, v)$. Since we also have $d[v] \geq \delta(s, v)$ (Corollary 1), we conclude $d[v] = \delta(s, v)$.

Suppose that $\delta(s, v)$ is infinite. Then, Corollary 1 implies $d[v] \geq \delta(s, v) = \infty$. $\square$

Correctness of BFS (cont'd)

Theorem 2. If $0 < d[v] < \infty$, then $\pi[v]$ is the predecessor of v on a shortest path from s to v.

Proof. By induction on $d[v]$. *Base case.* For $d[v] = 1$, (s, v) is an edge in E and, by construction, $\pi[v] = s$.
Induction step. Suppose that Theorem 2 holds for every vertex v such that $d[v] \le l_0$. For a vertex w with $d[w] = l_0 + 1$, define $v_{l_0} := \pi[w]$. By construction, $d[v_{l_0}] = d[w] - 1 = (l_0 + 1) - 1 = l_0$.
Hence, Lemma 1 implies that there exists a path $\langle\, s, v_1, \ldots, v_{l_0} \,\rangle$. Adding the edge (v_{l_0}, w), we obtain the path $p = \langle\, s, v_1, \ldots, v_{l_0}, w \,\rangle$.
The length of the path is $l_0 + 1 = d[w]$. Hence, Theorem 1 implies that p is a shortest path from s to w. The predecessor of w in p is $v_{l_0} = \pi[w]$. $\square$

DFS vs BFS [DPV 4.2]

- DFS goes "as far as possible" (depth), while BSF discovers all the vertices at a certain distance before moving to further vertices.
  In terms of data structures, this is reflected in the fact that DFS uses (implicitly) a last-in-first-out queue (i.e. a stack), while BSF uses a first-in-first-out queue.
- Unlike DFS, BFS *may not reach all vertices*:
  BFS discovers only the vertices that are reachable from the source s.
- Both algorithms have linear running time $O(|V| + |E|)$.

Dijkstra's Algorithm [CLRS 24.3]

- Dijkstra's algorithm solves the ***Single-Source Shortest Path Problem*** for ***non-negative weights***.
- It is essentially a weighted version of breath-first search.
- The algorithm uses a ***min-priority queue*** Q (instead of FIFO queue), with keys given by the *shortest-path weight estimates* $d[v]$.
- At termination,
  - $d[v]$ is equal to the distance from s to v
  - the algorithm provides a back-pointer $\pi[v]$, such that $\pi[v]$ is the predecessor of v on a shortest path from s to v, if such path exists, or $\pi[v] = \text{NIL}$ otherwise.

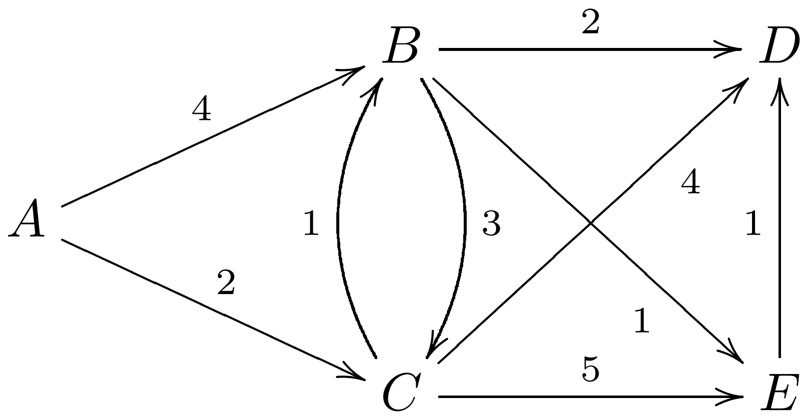
Pseudocode

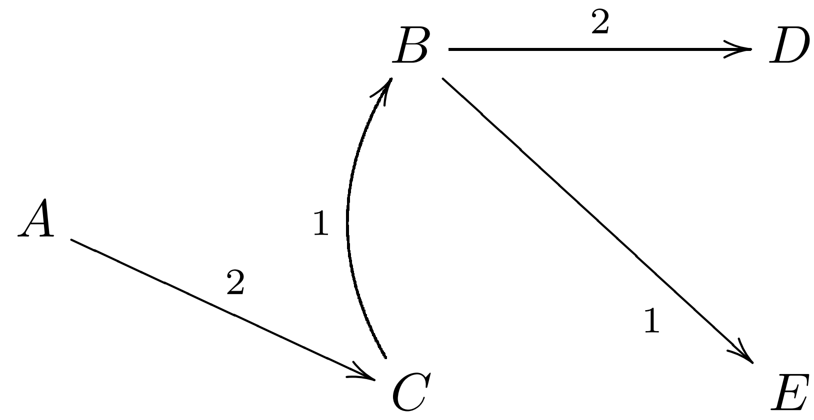
DIJKSTRA(V, E, w, s)

Input: A directed or undirected graph (V, E), $s \in V$, $w : E \to \mathbb{R}_{\geq 0}$.
Output: For each $v \in V$, $d[v]$ and $\pi[v]$

for each $v \in V$
  $d[v] = \infty$
  $\pi[v] = nil$
$d[s] = 0$
$Q =$ MAKE-QUEUE(V) with $d[v]$ as keys
while $Q \neq \emptyset$
  $u =$ EXTRACT-MIN(Q) // Shortest route to u known
  for each vertex $v \in Adj[u]$
    if $d[u] + w(u, v) < d[v]$ // Shorter route to v via u discovered
      $d[v] = d[u] + w(u, v)$
      $\pi[v] = u$
      DECREASE-KEY$(Q, v, d[v])$

Example





n-th iteration	Init	1	2	3	4	5
EXTRACT-MIN(Q)		A	C	B	E	D
(source) $d[A]$	0	0	0	0	0	0
$d[B]$	∞	4	3	3	3	3
$d[C]$	∞	2	2	2	2	2
$d[D]$	∞	∞	6	5	5	5
$d[E]$	∞	∞	7	4	4	4
Q	$\langle A, B, C, D, E \rangle$	$\langle C, B, D, E \rangle$	$\langle B, D, E \rangle$	$\langle E, D \rangle$	$\langle D \rangle$	$\langle\,\rangle$

Correctness

Loop invariants for the while loop:

I1. *For all* $v \in V$*, we have* $d[v] \geq \delta(s, v)$.

I2. *For all* $v \in S := V \setminus Q$*, we have* $d[v] = \delta(s, v)$.

Initialisation:

I1. Just before the start of the **while** loop, $d[s] = 0 = \delta(s, s)$ and $d[v] = \infty$ for every other vertex $v \in V$, $v \neq s$.
Hence I1 holds.

I2. Before the start of the **while** loop $S = \emptyset$, so I2 is vacuously true.

Termination: At the end, $S = V$.
Hence, I2 implies that $d[v] = \delta(s, v)$ for all $v \in V$, as required.

Maintenance of I1: $d[v] \geq \delta(s,v)\ \forall v \in V$

Suppose that I1 holds before an iteration of the **while** loop.
We have to show that I1 still holds after the iteration.

For a generic vertex $v \in V$, there are two possibilities:
either $d[v]$ did not change after the iteration, or $d[v]$ changed.

1. If $d[v]$ did not change, then $d[v] \geq \delta(s,v)$ by I1 before the iteration.
2. If $d[v]$ changed, let u be the vertex selected by EXTRACT-MIN in this iteration of the loop. Then, we have the inequality

$$\begin{aligned} d[v] &= d[u] + w(u,v) \\ &\geq \delta(s,u) + w(u,v) && \text{by I1 before the iteration} \\ &\geq \delta(s,v) && , \end{aligned}$$

where the last inequality holds because $\delta(s,u) + w(u,v)$ is the length of *a* path from s to v. $\square$

Terminology: $\delta(s,v) \leq \delta(s,u) + w(u,v)$ is called the ***triangle inequality***.

Maintenance of I2: $d[v] = \delta(s, v)\ ,\ \forall v \in S$

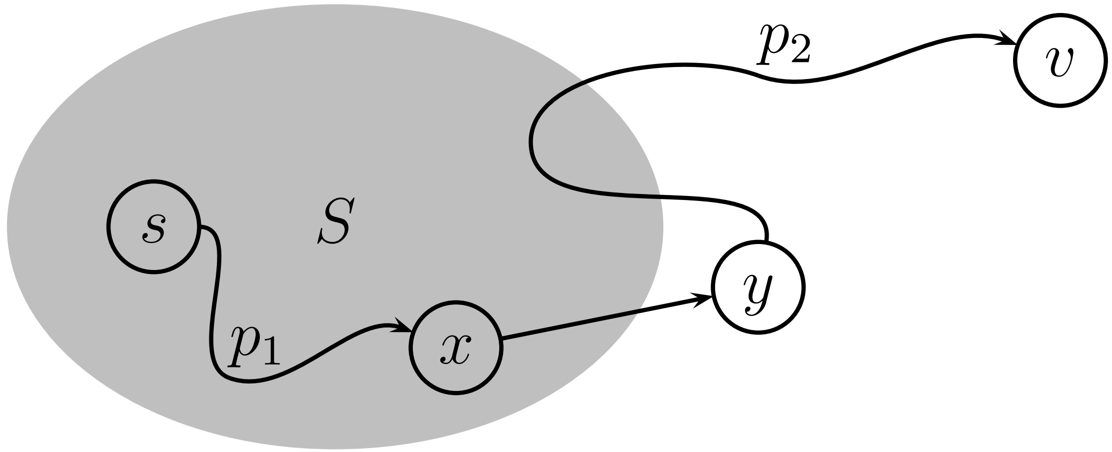
Suppose that I2 holds before an iteration of the **while** loop.
We have to show that I2 still holds after the iteration.

For a generic vertex $v \in S$ there are three possibilities:

1. v was in S before the iteration of the **while** loop.
   Then, $d[v] = \delta(s, v)$ by I2 before the iteration.
2. v has been added to S in this iteration *and* there is no path from s to v.
   Then, the d-value is $d[v] \geq \delta(s, v) = \infty$, namely $d[v] = \infty$.
3. v has been added to S in this iteration *and* there is a path from s to v.
   Then, there is also a *shortest* path $s \overset{p}{\twoheadrightarrow} v$.
   Just before v is added to S, the path p connects a vertex in S
   (the source s) with a vertex outside S (vertex v).
   Let y be the first vertex along p that is outside S,
   and let x be the predecessor of y (by definition, x is in S).

...picture on the next slide.

Maintenance of I2: $d[v] = \delta(s, v)$, $\forall v \in S$ (cont'd)



Optimal substructure: since $s \overset{p_1}{\twoheadrightarrow} x \to y \overset{p_2}{\twoheadrightarrow} v$ is a shortest path to v,
$s \overset{p_1}{\twoheadrightarrow} x \to y$ is a shortest path to y.

Since x was in S before the iteration of the **while** loop,
we have $d[x] = \delta(s, x)$ by I2 before the iteration.
Moreover, when x was extracted from Q,
the d-value of y must have been updated by lines 10-11 of the code.

...continue on the next slide.

The convergence property

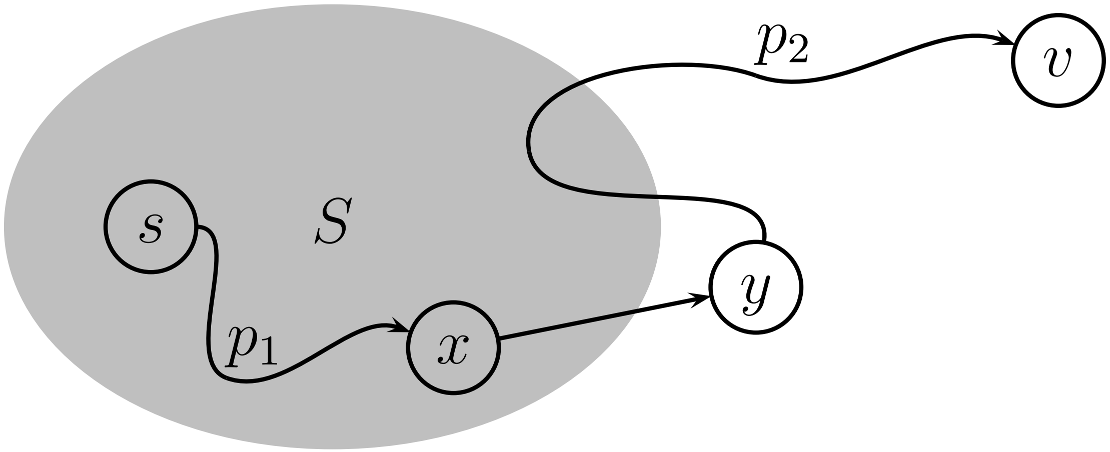
Lemma 1 (Convergence property).
Suppose $s \overset{p_1}{\twoheadrightarrow} x \to y$ *is a shortest path.*
If $d[x] = \delta(s, x)$ *before lines 10-11 are executed on vertex* y*,*
then $d[y] = \delta(s, y)$ *after lines 10-11 are executed on vertex* y*.*

After executing lines 10 and 11, we have

$$\begin{array}{rcll} d[y] & \leq & d[x] + w(x, y) & \text{by lines 10-11} \\ & = & \delta(s, x) + w(x, y) & \text{by hypothesis} \\ & = & \delta(s, y) & \text{because } s \overset{p_1}{\twoheadrightarrow} u \to v \text{ is a shortest path} \end{array}$$

By I1, we have $d[y] \geq \delta(s, y)$.
Hence $d[y] = \delta(s, y)$. $\square$

Maintenance of I2: $d[v] = \delta(s, v)$, $\forall v \in S$ (cont'd)



The Convergence Property implies $d[y] = \delta(s, y)$ before v is extracted from Q.

Since y and v were both in Q when v was chosen by EXTRACT-MIN, we have $d[v] \leq d[y] = \delta(s, y) \leq \delta(s, v)$.

In conclusion, we obtained $d[v] \leq \delta(s, v)$.
Since $d[v] \geq \delta(s, v)$ by I1, we obtained $d[v] = \delta(s, v)$.
This proves the validity of I2 after the iteration of the **while** loop. $\square$

Constructing shortest paths

Theorem. For $v \neq s$ and $d[v] < \infty$, $\pi[v]$ is the predecessor of v on a shortest path from s to v.

Proof. Let $u := \pi[v]$, and let $s \overset{p}{\twoheadrightarrow} u$ be a shortest path from s to u. At termination, we have

$$\begin{aligned} \delta(s,v) &= d[v] && \text{by I2} \\ &= d[u] + w(u,v) && \text{because } u = \pi[v] \\ &= \delta(s,u) + w(u,v) && \text{by I2} \end{aligned} \tag{1}$$

Hence $s \overset{p}{\twoheadrightarrow} u \rightarrow v$ is a shortest path from s to v. $\square$

Running time

The running time of Dijkstra's algorithm depends on the implementation of the min-priority queue. For a min-heap implementation, one has

1. MAKE-QUEUE: $O(|V|)$
2. EXTRACT-MIN: $O(\log |V|)$
3. DECREASE-KEY: $O(\log |V|)$

MAKE-QUEUE is executed once.

EXTRACT-MIN is executed $|V|$ times.

Since each vertex $v \in V$ is added to S exactly once, each edge in $Adj[v]$ is examined in the **for** loop exactly once.

Thus there are a total of $|E|$ iterations of the **for** loop, and hence a total of at most $|E|$ DECREASE-KEY operations.

Total running time:

$$\begin{aligned} & O(|V|) + O(|V| \log |V|) + O(|E|) + O(|E| \log |V|) \\ = \; & O((|V| + |E|) \log |V|) \end{aligned}$$

A variation: Unique Shortest Path

Question: is shortest path to a given vertex is unique or not?

We will now see an algorithm that answers this question.

Input: A directed or undirected graph $G = (V, E)$
with non-negative weights $w : E \to \mathbb{R}_{\geq}$,
and a source vertex $s \in V$.

Output: A Boolean array $usp[v]$ such that, for each vertex v,
$usp[v] = \text{TRUE}$ iff the shortest path from s to v is unique.

Note:

- $usp[s] = \text{TRUE}$
- if $\delta(s, v) = \infty$ we conventionally assign the value $usp[v] = \text{TRUE}$

Pseudocode for the Unique Shortest Path Problem

The problem can be solved by modification of Dijkstra's algorithm.
The array usp[-] is initialized to TRUE.
The **while** loop is modified as follows:

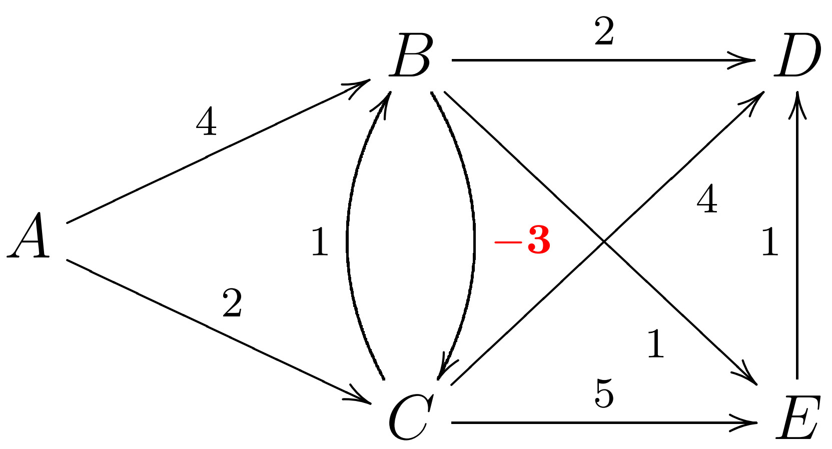
while $Q \neq \emptyset$
    $u = \textsc{Extract-Min}(Q)$
    for each vertex $v \in Adj[u]$
        if $d[u] + w(u, v) < d[v]$
            $d[v] = d[u] + w(u, v)$
            $\pi[v] = u$
            $usp[v] = usp[u]$
            $\textsc{Decrease-Key}(Q, v, d[v])$
        else if $d[v] = d[u] + w(u, v)$
                $usp[v] =$ FALSE

Shortest paths in graphs with negative weights

So far, we assumed that the weight function $w : E \to \mathbb{R}$ was non-negative, namely that $w(u, v) \geq 0$ for every edge $(u, v) \in E$.

How about graphs where some of the weights are negative?

Problem: negative-weight cycles



$\langle\, B, C, B \,\rangle$ is a negative-weight cycle: $w(B, C) + w(C, B) = -2$.
There is *no shortest path* from source A in this graph!

General graphs: Bellman-Ford Algorithm [CLRS 24.1]

BELLMAN-FORD(V, E, w, s)

Input: A directed or undirected graph (V, E), $s \in V$, $w : E \to \mathbb{R}$.
Output: FALSE, if there exist a negative-weight cycle reachable from s, otherwise TRUE, and for each $v \in V$, $d[v]$ and $\pi[v]$.

```
for each v ∈ V
    d[v] = ∞; π[v] = nil
d[s] = 0
for i = 1 to |V| − 1                      // correctly computes the distances
    for each edge (u, v) ∈ E              // if there is no negative-weight
        if d[u] + w(u, v) < d[v]          // cycle reachable from s
            d[v] = d[u] + w(u, v); π[v] = u
for each edge (u, v) ∈ E                  // checks for negative-weight cycles
    if d[u] + w(u, v) < d[v]              // reachable from s
        return FALSE
    else return TRUE
```

Running time

- Initialisation takes $\Theta(|V|)$.
- Each of the $|V| - 1$ iterations of lines 5–7 takes $\Theta(|E|)$
- the **for** loop of lines 8-11 takes $O(|E|)$.

In total, the running time is $\Theta(|V||E|)$.

Correctness: the case of no negative-weight cycles

If no negative-weight cycle is reachable from s,
then the shortest paths are well-defined.

Invariant of the for loop 4-7. Before the i-th iteration of the loop,

I1 $d[v] \geq \delta(s, v)$, $\forall v \in V$

I2 *for every shortest path* $\langle v_0, v_1, \ldots, v_k \rangle$ *from* $v_0 = s$ *to* v_k*,*
$d[v_j] = \delta(s, v_j)$, $\forall j < i$

Initialisation. Before the first iteration, $d[s] = 0 = \delta(s, s)$, and $d[v] = \infty \geq \delta(s, v)$ for every $v \in V$, $v \neq s$. Hence, I1 holds. Moreover, $d[v_0] = d[s] = 0 = \delta(s, s)$. Hence, I2 holds.

Termination. Termination occurs for $i = |V|$.
Then, I2 implies $d[v_j] = \delta(s, v_j)$ for all $j < |V|$. Since there are $|V|$ vertices in the graph, one has $k < |V|$, and therefore $d[v_j] = \delta(s, v_j)$ for all $j < k$. Hence, if there is a path from s to v, $d[v] = \delta(s, v)$. If there is no path, $\delta(s, v) = \infty$ and $d[v] \geq \delta(s, v)$ by I1, whence $d[v] = \delta(s, v)$.

Maintenance of I1: $d[v] \geq \delta(s, v)$, $\forall v \in V$

Fact: I1 is maintained also by the ***internal*** **for** loop (lines 5-7).

Indeed, consider the iteration of the internal **for** loop on the edge (u, v)

- Only the d-value of v can change in this iteration.
  For all the other d-values, I1 still holds.
- If v did not change, then I1 still holds.
- If $d[v]$ changed, then $d[v] = d[u] + w(u, v)$.
  Hence, one has

$$
\begin{array}{rcll}
d[v] & = & d[u] + w(u, v) & \\
 & \geq & \delta(s, u) + w(u, v) & \text{by I1 before the iteration} \\
 & \geq & \delta(s, v)\,. &
\end{array}
$$

Since I1 is maintained by the internal **for** loop 5-7, it is also maintained by the external **for** loop 4-7.

Maintenance of I2: $d[v_j] = \delta(s, v_j)$, $\forall j < i$

Assume that I2 holds before the i-th iteration of the **for** loop 4-7.

Fact: For every $v \in V$, the d-value $d[v]$ does not increase after each iteration of the internal **for** loop 5-7 (and therefore it does not increase after each iteration of the external **for** loop 4-7)

For every $j < i$, the new d-value $d'[v_j]$ is no larger than the old d-value $d[v_j] = \delta(s, v_j)$. By I1, this implies $d'[v_j] = \delta(s, v_j)$.

For $j = i$, consider the iteration of the internal **for** loop on the edge (v_{i-1}, v_i). After this iteration, one has

$$\begin{aligned} d[v_i] &\le d[v_{i-1}] + w(v_{i-1}, v_i) \\ &= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) && \text{because I2 holds for all } j < i \\ &= \delta(s, v_i) && \text{because } \langle\, s, v_1, \ldots, v_k \,\rangle \text{ is a shortest path,} \\ & && \text{whence } \langle\, s, v_1, \ldots, v_i \,\rangle \text{ is a shortest path} \end{aligned}$$

Since the d-values are non-increasing, this bound continues to hold after other iterations of the internal **for** loop. Hence, $d[v_i] = \delta(s, v_i)$.

Constructing shortest paths

Theorem. Suppose that no negative-weight cycle is reachable from s. Then, for $v \neq s$ and $d[v] < \infty$, $\pi[v]$ is the predecessor of v on a shortest path from s to v.

Proof. Same proof as in Dijkstra's algorithm.

Correctness: the case of negative-weight cycles

Theorem. If there is a negative-weight cycle reachable from s, then the **for** loop 5-7 will return FALSE.

Proof. Suppose that there is a negative weight-cycle reachable from s, say $c = (v_1, v_2, \ldots, v_n, v_{n+1})$ with $v_{n+1} = v_1$, and $w(c) := \sum_{i=1}^{n} w(v_i, v_{i+1}) < 0$.

Fact: we have the inequality

$$\begin{aligned} \sum_{i=1}^{n} \left[d[v_i] + w(v_i, v_{i+1}) \right] &= \left(\sum_{i=1}^{n} d[v_i] \right) + w(c) \\ &< \sum_{i=1}^{n} d[v_i] \\ &= \sum_{i=1}^{n} d[v_{i+1}] \,. \end{aligned}$$

Since the l.h.s. is strictly smaller than the r.h.s.,
there must exist a vertex v_i such that $d[v_i] + w(v_i, v_{i+1}) < d[v_{i+1}]$.
Hence, the **for** loop of lines 8-11 will return FALSE upon inspecting the edge (v_i, v_{i+1}). $\square$

Shortest paths in DAGs [CLRS 24.2]

Idea: Find a topological sort of V. The shortest path from s to v can only contain intermediate vertices that come *before* v in the topological sort.

Input: A DAG $G = (V, E)$ with weight $w : E \to \mathbb{R}$
and a source vertex s
Output: For each $v \in V$, length $d[v]$ of shortest-path from s to v

$\textsc{Topological-sort}(V, E)$
for each $v \in V$
    $d[v] = \infty$
    $\pi[v] = nil$
$d[s] = 0$
for each $u \in V$ *in topological order*
    for each $v \in Adj[u]$
        if $d[v] > d[u] + w(u, v)$
        $d[v] = d[u] + w(u, v)$
        $\pi[v] = u$

Running time

1. TOPOLOGICAL-SORT(V, E) takes $O(|V| + |E|)$ time
2. initialization (lines 2-5) takes $\Theta(|V|)$ time
3. line 6 is run one time on each vertex, resulting into a $O(|V|)$ time
4. the **for** loop of line 7 runs $O(|E|)$ time

Total running time: $O(|V| + |E|)$

Correctness

Invariant of the for loop of lines 6-10.

I1 $d[v] \geq \delta(s, v)$ *for all* $v \in V$

I2 *Let* $(v_1, v_2, \ldots, v_n)$ *be the topological sort.*
*Before the **for** loop is executed on vertex* v_k*,*
one has $d[v_i] = \delta(s, v_i)$ *for every* $i \leq k$.

Initialisation:

- $d[s] = \delta(s, s)$. For $v \neq s$, $d[v] = \infty \geq \delta(s, v)$. Hence, I1 holds.
- if $v_1 = s$, $d[v_1] = 0 = \delta(s, s)$ before the loop starts.
  Hence, I2 holds for $v_1 = s$
- if $v_1 \neq s$, then there cannot be a path from s to v_1.
  Hence, $\delta(s, v_1) = \infty = d[v_1]$ and I2 holds for $v_1 \neq s$.

Termination. At termination, I2 yields $d[v] = \delta(s, v)$ for every vertex $v \in V$.

Maintenance of I1: proof as usual (cf. proof in Bellman-Ford algorithm).

Maintenance of I2: $d[v_i] = \delta(s, v_i)$ for every $i \leq k$.

When the **for** loop is executed on vertex v_k, only vertices in $Adj[v_k]$ are affected. That is, only vertices v_j with $j > k$ are affected (because of topological sort). Hence, we only need to consider vertex v_{k+1}.

Let v_j be the predecessor of v_{k+1} on a shortest path from s to v_{k+1}. By construction, $j < k + 1$, and therefore $d[v_j] = \delta(s, v_j)$ by I2 before iteration.

After the **for** loop 6-10 was executed on v_j, one had

$$\begin{aligned} d[v_{k+1}] &\leq d[v_j] + w(v_j, v_{k+1}) \\ &= \delta(s, v_j) + w(v_j, v_{k+1}) \quad \text{by the invariant before iteration} \\ &= \delta(s, v_{k+1})\,. \end{aligned}$$

Since further iterations of the **for** loop can not increase the d-value, and since I1 holds, we conclude that $d[v_{k+1}] = \delta(s, v_{k+1})$ after the **for** loop is executed on v_k.

Constructing shortest paths

Theorem. For $v \neq s$ and $d[v] < \infty$, $\pi[v]$ is the predecessor of v on a shortest path from s to v.

Proof. Same proof as in Dijkstra's algorithm.

All-pairs shortest paths [CLRS 25]

All-pairs shortest paths

Input: A directed graph (V, E) with general weights $w : E \to \mathbb{R}$, and **with no negative-weight cycles.**

Task: For each pair of vertices u and v, find shortest path from u to v.

Using Single-source algorithms

- ☐ Using Bellman-Ford algorithm for each vertex, leads to a $O(|V|^2|E|)$ algorithm, which is $O(|V|^3)$ for a sparse graph ($|E| = O(|V|)$) and $O(|V|^4)$ for a dense graph ($|E| = O(|V|^2)$).
- ☐ If there are no negative-weight edges, using Dijkstra's algorithm leads to a $O((|V| + |E|)\,|V|\log|V|)$ algorithm.

Using dynamic programming

- ☐ $O(|V|^3)$ with the Floyd-Warshall algorithm (next slides).

The Floyd-Warshall algorithm [CLRS 25.2]

Suppose the vertex-set is $\{\, 0, \cdots, n-1 \,\}$ and let

$$d[i,j;k] \;=\; \begin{cases} \text{length of shortest path from } i \text{ to } j\text{, all of whose} \\ \text{intermediate nodes are in the set } \{0,\ldots,k-1\} \end{cases}$$

Initially

$$d[i,j;0] \;=\; \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Floyd-Warshall algorithm, cont'd

Suppose that we have calculated $d[i, j; k]$ for every i and j, and we want to compute $d[i, j; k+1]$.

Optimal substructure: if $s \overset{p_1}{\twoheadrightarrow} u \overset{p_2}{\twoheadrightarrow} v$ is a shortest path, then $s \overset{p_1}{\twoheadrightarrow} u$ and $u \overset{p_2}{\twoheadrightarrow}$ are shortest paths.

Fact: since there are no negative cycles (by assumption), a shortest path from i to j that uses intermediate vertices in $\{0, \ldots, k\}$ goes through k at most once.
Hence, we have

$$d[i, j; k+1] = \min \Big\{ d[i, j; k] \, , \; d[i, k; k] + d[k, j; k] \Big\}$$

Pseudocode

FLOYD-WARSHALL(V, E, w)

```
for i = 0 to |V| − 1
    for j = 0 to |V| − 1
        d[i, j; 0] = ∞
for each edge (i, j) ∈ E
    d[i, j; 0] = w(i, j)
for k = 0 to |V| − 1
    for i = 0 to |V| − 1
        for j = 0 to |V| − 1
            d[i, j; k + 1] = min{ d[i, j; k] , d[i, k; k] + d[k, j; k] }
```

Running time is $O(|V|^3)$.

Summary of Shortest Paths Algorithms

- ***Breadth-First-Search*** solves the **single-source** shortest path problem for directed or undirected graphs with **unit weight**, i.e. where $w(e) = 1$ for all $e \in E$ in $O(|V| + |E|)$.

- ***Dijkstra's Algorithm*** solves the **single-source** shortest-path problem for directed or undirected graphs with **non-negative weights** in $O((|V| + |E|) \log |V|)$.

- The ***Bellman-Ford Algorithm*** solves the **single-source** shortest-path problem for directed or undirected graphs **with arbitrary weights** in $O(|V||E|)$. In addition, it checks **checks for negative-weight cycles**.

- ***Shortest Paths in DAGs*** solves the **single-source** shortest path problem with **arbitrary weights** in the case of a **directed acyclic graph** in $\Theta(|V| + |E|)$.

- The ***Floyd-Warshall Algorithm*** solves the **all-pairs** shortest path problem for directed or undirected graphs with **no negative-weight cycles** in $O(|V|^3)$.