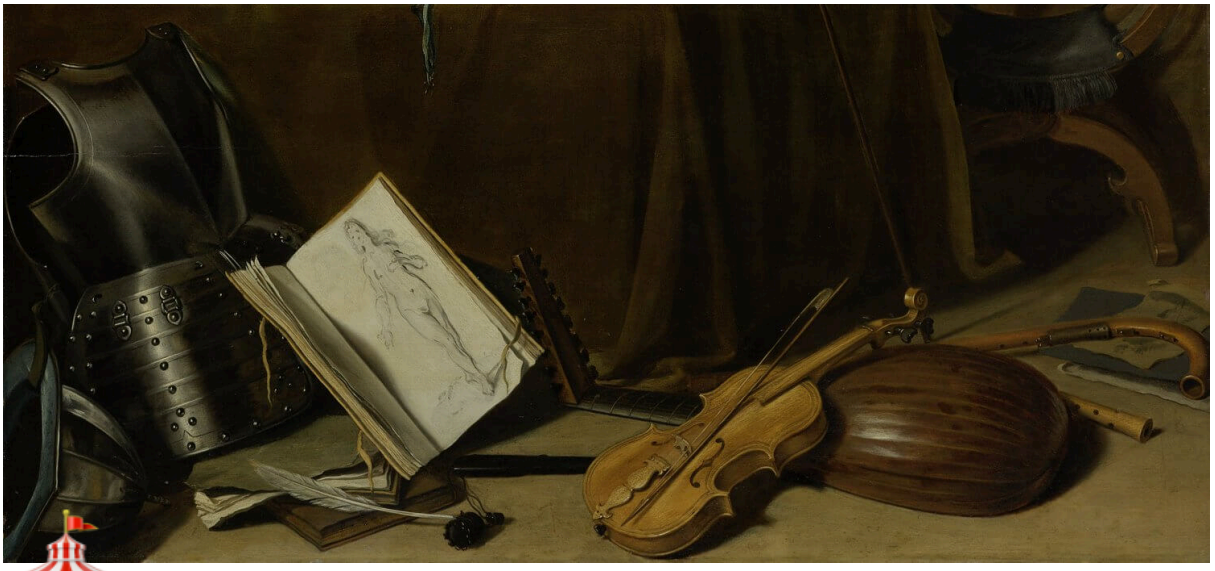


Graphs and Distances

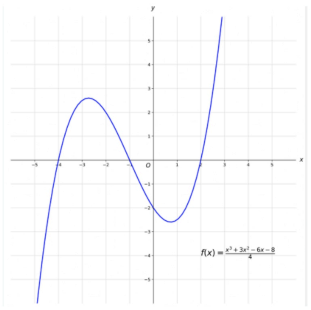
Administrivia (Week 7 – 11)

- **Office hours**: Every Wednesday 1500—1600.
  - CS Sloman Lounge Meeting Room 1
- **Email policy**:
  - All discussions regarding lecture material and related **MUST** be in Slack General channel.
  - Want to discuss privately? Meet me during office hours.
  - *Private communication will only be answered if urgent and not related to course material.*
- **Classroom policy**:
  - **Ask questions**. No question is stupid!
  - **No need to take notes**. Video and (non-annotated) slides will be uploaded.

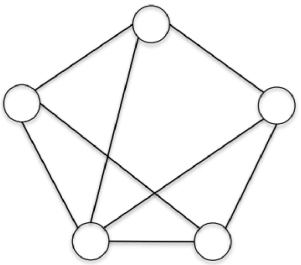
Aims of this week

- We've seen a lot of trees already
  - Binary (search) tree to quickly search.
  - Heaps to quickly insert an element and find a priority element.
  - Decision trees in the lower bound for comparison sorts
  - DFS and BFS on trees.
- Now: **Graphs!**
- Expect a change in **design approach**:
  - Not worried about low-level implementation of DSs.
  - Focus on how to use them cleverly.
  - Using **pseudocode** instead of Java.

What is a graph?



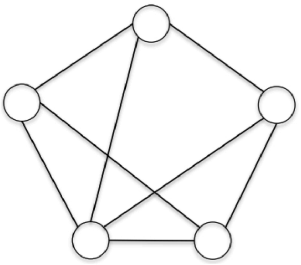
A graph in most of the world



A graph for a computer scientist

What is a graph?

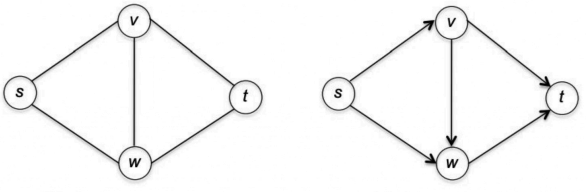
- Objects and their pairwise relationships
- **Node/Vertex**: An object
- **Edge**: Pairwise relationship
- V = set of vertices
- E = set of edges
- $G = (V, E)$ is a graph.



A graph for a computer scientist

Direction?

- **Undirected graph**: Edge corresponds to ***unordered pair*** of vertices.
  - Edge $e = (u, v) = (v, u) = \{u, v\}$.
  - Vertices u and v are endpoints.
- **Directed graph**: Each edge is an **ordered pair.**
  - $(u, v) \neq (v, u)$.
  - For (u, v), u is the tail and v is the head of the **arc**.

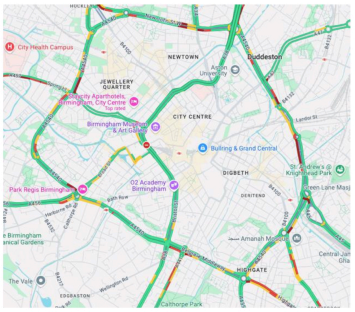


(a) An undirected graph (b) A directed graph

Why should we care?

- **Social network**:
  - Friendship network (undirected) or follower network (directed).
- **Road network**:
  - Google map
- **The Web**:
  - Google search (Pagerank, directed)
- **Precedence constraint:**
  - How to schedule lectures?

A friendship network



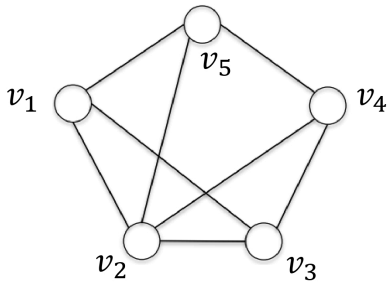
A road network

Sparse vs Dense graph

- **Quiz:** Undirected graph $G = (V, E)$ with no parallel edges.
  - Maximum number of edges?
  - If the graph is connected, minimum number of edges?
- Every pair of vertices (u, v) can have an edge between them.
  - How many such distinct pairs? $\frac{n(n-1)}{2} = O(n^2)$.
- Every vertex must have an edge incident on it.
  - # edges = $\Omega(n)$.
- Sparsest connected graph on n vertices?
  - Tree.

Graph representation

- **Adjacency list**: Ingredients
  - An array/linked list V containing vertices.
  - An array/linked list E containing edges.
  - Each edge e has a pointer to two of its endpoints.
  - Each vertex v has a pointer to each of incident edges.

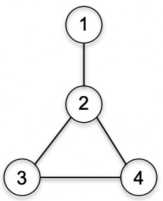


Graph representation

- **Adjacency list**: Ingredients
  - An array/linked list V containing vertices.
  - An array/linked list E containing edges.
  - Each edge e has a pointer to two of its endpoints.
  - Each vertex v has a pointer to each of incident edges.
- **Quiz**: How much space is required for n vertices and m edges?
  - $O(m+n)$

Graph representation

- **Adjacency matrix**: Ingredients
  - A 2D-array/matrix A of dimension $n \times n$.
  - $A_{i,j} = 1$ if and only if there is an edge between vertices i and j.



(a) A graph...

	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	0

(b) ...and its adjacency matrix

- **Quiz**: How much space is required for n vertices and m edges?
  - $O(n^2)$

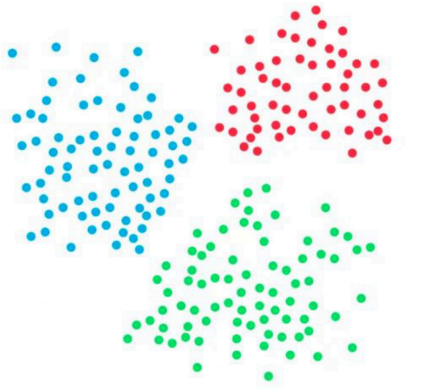
Graph representation: Which one?

It depends...

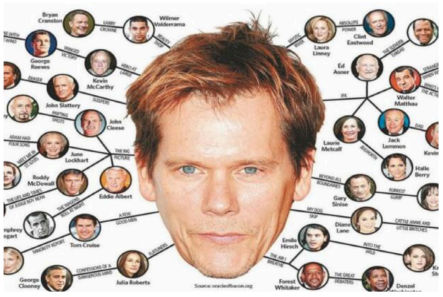
- **Graph exploration**: Usually adjacency list shines.
  - We will see them soon.
- **Sparse graph** $m \approx O(n)$: Usually adjacency list shines.
  - Why?
- **Graph properties using linear algebra**: Usually adjacency matrix shines.
  - We will not see them in this module.

Graph Search and Exploration

When is it useful?



- **Connectivity**:
  - Network reachability
  - Sudoku/Chess/Puzzle solving



- **Connected components**:
  - Clustering

- **Shortest path**:
  - Erdös number/Bacon number

Generic Graph Exploration

Input: An undirected or directed graph $G = (V, E)$ and a starting vertex v.

Goal: Identify all vertices reachable from s.

- First, if G is connected, s can reach every vertex.
  - G is disconnected: How to check? How to find reachable vertices?
- Strategy:
  - Start from s and explore edges connected to what already explored.
  - Avoid exploring anything twice!

Generic Graph Exploration

Input: An undirected or directed graph $G = (V, E)$ and a starting vertex v.

Goal: Identify all vertices reachable from s.

1. Create sets Explored and Unexplored.
2. Put s in Explored, every other vertex in Unexplored.
3. **while** ∃ edge (u, v) with $u \in$ Explored and $v \in$ Unexplored:
   I. Put $v \in$ Explored.

Quiz: Suppose you can do (3) in $O(1)$ time:

- What is the runtime of generic graph exploration?

Generic Graph Exploration

How to do:
3. while ∃ edge (u, v) with $u \in$ Explored and $v \in$ Unexplored:
I. Put $v \in$ Explored.

Suppose we explore in layers!

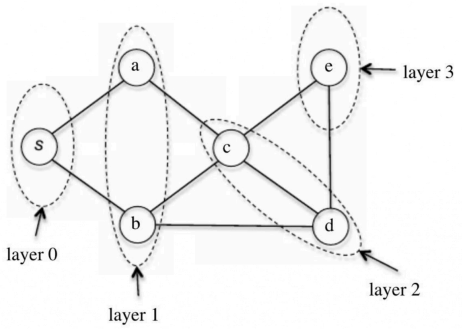
1. Layer 0 contains just s.
2. Layer 1 contains vertices 1-hop away from s.
3. …
4. Layer $k+1$ contains vertices 1-hop away from vertices in Layer k.

Generic Graph Exploration

How to do:
3. while ∃ edge (u, v) with $u \in$ Explored and $v \in$ Unexplored:
I. Put $v \in$ Explored.

Suppose we explore in layers!

1. Layer 0 contains just s.
2. Layer $k+1$ contains vertices 1-hop away from vertices in Layer k.



You have seen this before!

Recall: BFS Exploration

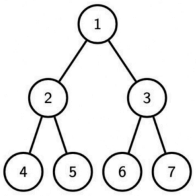
How to do:

3. while ∃ edge (u, v) with $u \in$ Explored and $v \in$ Unexplored:

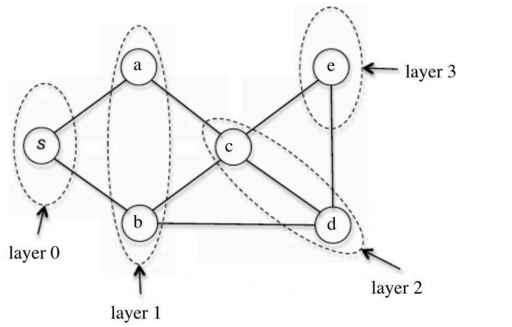
I. Put $v \in$ Explored.

BFS = Breadth First Search

Traversing the tree by levels, i.e., all children will be visited before any node on level 2.



This is called **breadth first search (BFS)**.



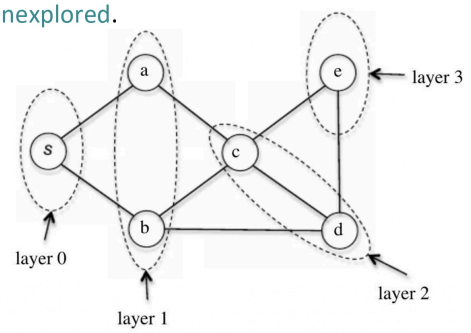
Data Structure & Algorithms 23

BFS Exploration

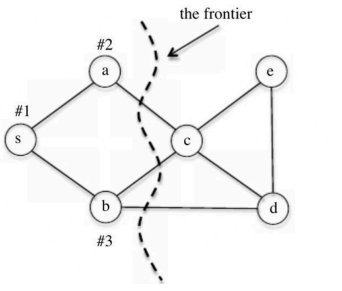
Input: An undirected or directed graph $G = (V, E)$ and a starting vertex v.

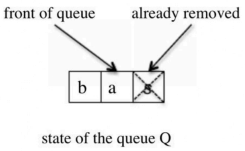
Goal: Identify all vertices reachable from s.

1. Create sets Explored and Unexplored, and Q = a queue initialised with s.
2. Put s in Explored, every other vertex in Unexplored.
3. **while** Q is not empty
   - *a)* v = Dequeue(Q).
   - **b)** **for each** edge (v, w):
     - If $w \in$ Unexplored:
       - $w \in$ Explored
       - Enqueue(w)

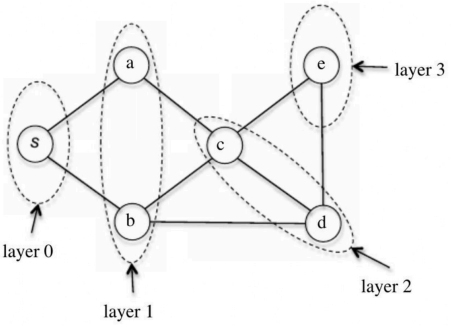


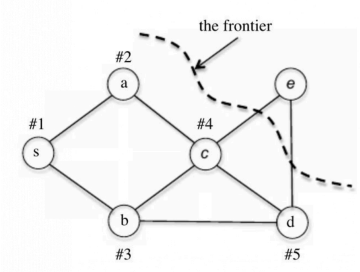
BFS Exploration in action

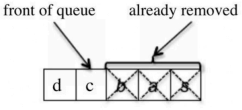




state of the queue Q







state of the queue Q

BFS Exploration Analysis

1. Create sets Explored and Unexplored, and Q = a queue initialised with s.
2. Put s in Explored, every other vertex in Unexplored.
3. **while** Q is not empty
   - *a)* v = Dequeue(Q).
   - **b)** **for each** edge (v, w):
     - If $w \in$ Unexplored:
       - $w \in$ Explored
       - Enqueue(w)

Runtime of init: $O(n)$

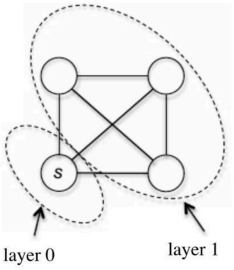
How many times an edge (u, v) can be used?

- Once when u is dequeued.
- Once when v is dequeued.

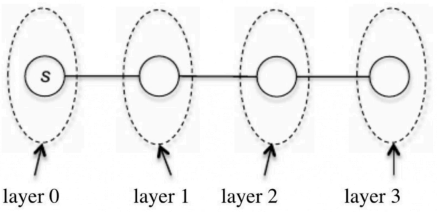
Runtime of for loop: $O(m)$

BFS Exploration: Quiz

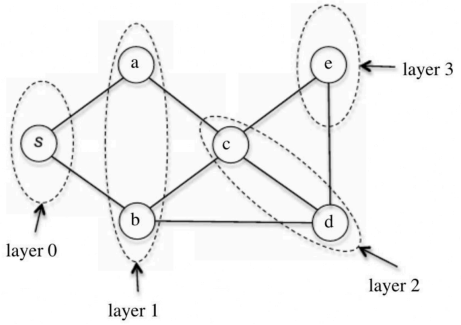
- Consider a complete graph G. How many layers?



- Consider a path graph G with s in one end. How many layers?



BFS Exploration: Hop-distance



What does Layer k mean?

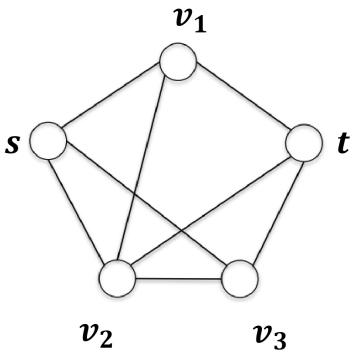
- $v \in$ Layer $k \Rightarrow$ You can reach v from s by visiting k vertices.
- In other words: v is hop-distance k away from s.

A BFS tree is hidden in BFS exploration: Broadcast tree.
Can you find it?
Can you find more than one?

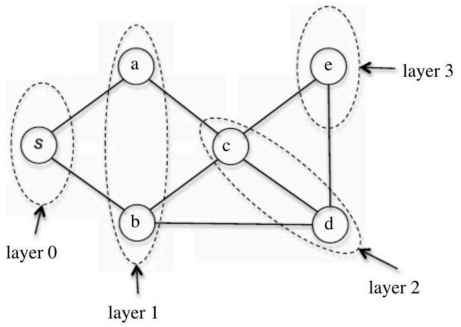
Graph Distance

Distance in graphs

- Consider two vertices s, t.
- Multiple paths connecting s and t.
  - s, v_1, t.
  - s, v_2, v_3, t
  - s, v_1, v_2, v_3, t, etc.
- $dist(s, t)$ = length of the shortest path.
  - Length of s, v_1, t = 2.
- Same as hop-distance for **unweighted graphs**!



Single-source shortest path (SSSP)



Observe:

- BFS exploration from s ensures that:
  - $v \in$ Layer $k \Rightarrow dist(s, v) = k$.
- BFS correctly computes shortest path from s to every other vertex.
- **Think:** The graph can be directed as well! BFS does not care.

BFS on directed graph: What changes?

1. Create sets Explored and Unexplored, and Q = a queue initialised with s.
2. Put s in Explored, every other vertex in Unexplored.
3. **while** Q is not empty
   - *a)* v = Dequeue(Q).
   - **b)** **for each** ~~edge~~ arc (v, w):
     - If $w \in$ Unexplored:
       - $w \in$ Explored
       - Enqueue(w)

Runtime of init: $O(n)$

How many times an edge (u, v) can be used?

- Once when u is dequeued.
- ~~Once when v is dequeued.~~

Runtime of for loop: $O(m)$

The Waterloo of BFS: Weighted graphs

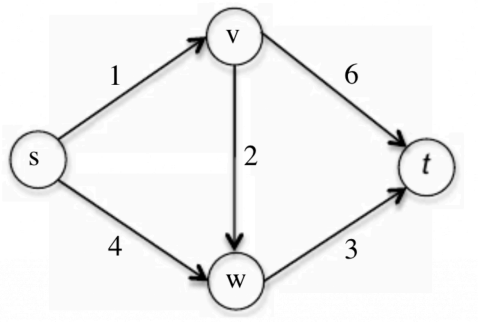
What is a weighted graph?

- $G = (V, E)$ with n vertices and m edges/arcs
- Each edge e has a weight/length ℓ_e.

What is distance now?

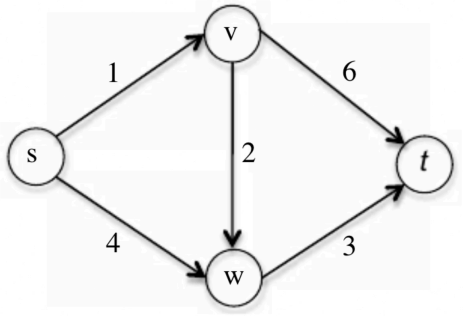
- Length of a path P = ~~number of edges~~ Total weight of the edges in P

Run BFS from s and see what happens!

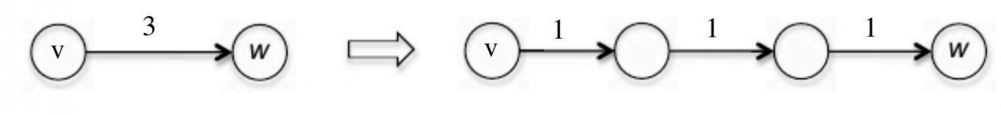


The Waterloo of BFS: Weighted graphs

Run BFS from s and see what happens!



Possible fix: Replace an edge e with ℓ_e unweighted edges in series and run BFS.



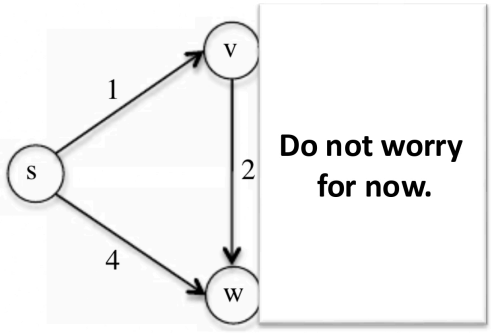
What goes wrong?

SSSP on weighted graph

Notation: $len(v) = dist(s, v)$.

- Start with $len(s) = 0$ and $len(v), len(w) = \infty$.

- Look at outgoing arcs from s.
  - Set $len(v) = 1, len(w) = 4$.
- Look at outgoing arcs from v.
  - Shorter path to w.
  - Change $len(w) = 3$.
- …



How do we formalise this?

Dijkstra's algorithm for SSSP

Input: A weighted directed graph $G = (V, E)$ and a starting vertex s.

Goal: Identify distance $dist(s, v) = len(v)$ all vertices v reachable from s.

- **Set** $X = \{s\}, len(s) = 0, len(v) = \infty$ for all $v \neq s$.
- **while** ∃ arc (v, w) with $v \in X, w \notin X$, **do**: **Loop runs $O(n)$ times**
  - **Find the arc (a, b) with minimum $len(a) + \ell_{ab}$.**
  - **Add** b to X.
  - **Set** $len(b) = len(a) + \ell_{ab}$.

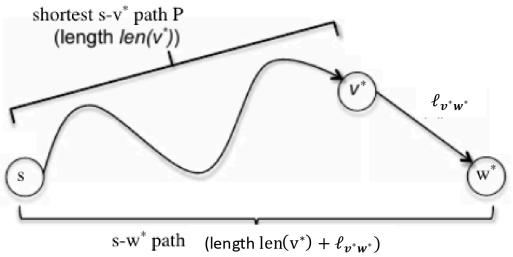
May take $O(m)$ time

Total runtime: $O(mn)$

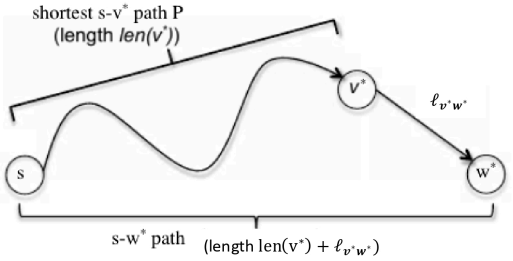
Dijkstra's algorithm: Why does it work?

- **Set** $X = \{s\}, \text{len}(s) = 0, \text{len}(v) = \infty$ for all $v \neq s$.
- **while** ∃ arc (v, w) with $v \in X, w \notin X$, **do**:
  - **Find the arc (a, b) with minimum** $\text{len}(a) + \ell_{ab}$.
  - **Add** b to X.
  - **Set** $\text{len}(b) = \text{len}(a) + \ell_{ab}$.

- Consider a path P from s to w. Say this is the shortest path.
- Now consider any v in that path.
- **Observe**: $P(s, v)$ is the shortest path from s to v.



Dijkstra's algorithm: Induction



Induction Hypothesis: $len(v) = dist(s, v)$ for the first $k - 1$ vertices added to X.

To Show: $len(w) = dist(s, w)$. ⇨ $len(w) \geq dist(s, w)$ and $len(w) \leq dist(s, w)$.

Easy! Show a path **Non-trivial!**

$len(w) \geq dist(s, w)$

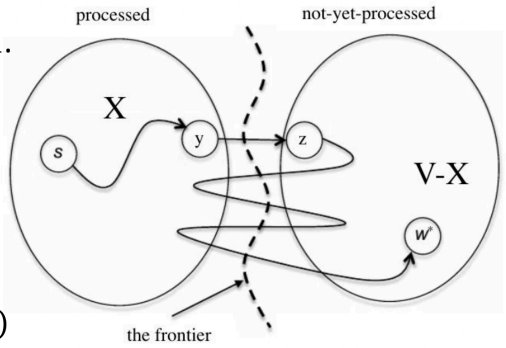
- When arc (v, w) was chosen, $v \in X$.
- Take path $P(s, v)$ and add arc (v, w). This is a path from s to w.
- By definition, length of this path $\geq dist(s, w)$.

Dijkstra's algorithm: Induction

Induction Hypothesis: $len(v) = dist(s, v)$ for the first $k - 1$ vertices added to X.

To Show: $len(w) \leq dist(s, w)$.

- Recall: X = all processed vertices till $k - 1$.
  - This means, $v \in X$.
- Consider the shortest path Q from s to w.
- (y, z): First time Q crosses frontier.



$$\text{Length}(Q) = dist(s, y) + \ell_{yz} + \text{Length}(Q_{zw})$$

$$\geq \mathbf{0}$$

$$\geq dist(s, y) + \ell_{yz} \geq dist(s, v) + \ell_{vw} = len(w).$$

Dijkstra's algorithm: Faster!

Input: A weighted directed graph $G = (V, E)$ and a starting vertex s.

Goal: Identify distance $dist(s, v) = len(v)$ all vertices v reachable from s.

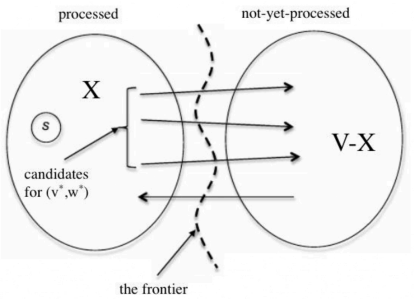
- **Set** $X = \{s\}, d(s) = 0, d(v) = \infty$ for all $v \neq s$.
- **while** ∃ arc (v, w) with $v \in X, w \notin X$, **do**: **Loop runs $O(n)$ times**
  - **Find the arc (a, b) with minimum $len(a) + \ell_{ab}$.**
  - **Add** b to X.
  - **Set** $len(b) = len(a) + \ell_{ab}$.

Can we use any nice data-structure?

Dijkstra's algorithm: $O((m + n)\log m)$

- **Set** $X = \{s\}, d(s) = 0, d(v) = \infty$ for all $v \neq s$.
- **while** ∃ arc (v, w) with $v \in X, w \notin X$, **do**:
  - **Find the arc (a, b) with minimum $len(a) + \ell_{ab}$.**
  - **Add** b to X.
  - **Set** $len(b) = len(a) + \ell_{ab}$.

What are we doing?



Attempt 1: Store arcs+scores in a Min-heap

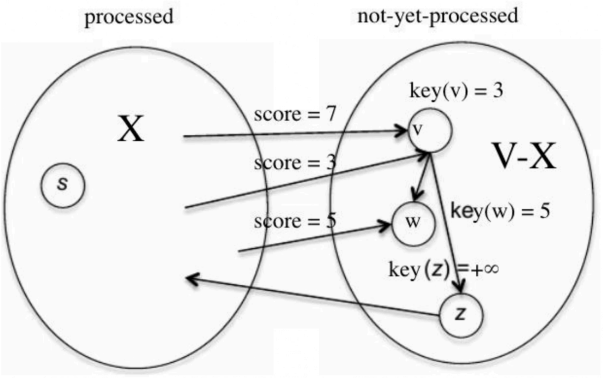
- When $b \in X$:
  - Remove arcs with head b.
  - Add arcs with tail b.
- Remove more than just the min.

- Each arc is inserted once and deleted once. Run time = $O(m \log m)$.

Dijkstra's algorithm: Min-heap of vertices

- Consider *key* of a vertex $v \in V - X$:

$$key(v) = \min_{(u,v)\in E: u\in X} len(u) + \ell_{uv}.$$



Dijkstra's algorithm: Min-heap of vertices

- Consider *key* of a vertex $v \in V - X$:

$$key(v) = \min_{(u,v)\in E: u\in X} len(u) + \ell_{uv}.$$

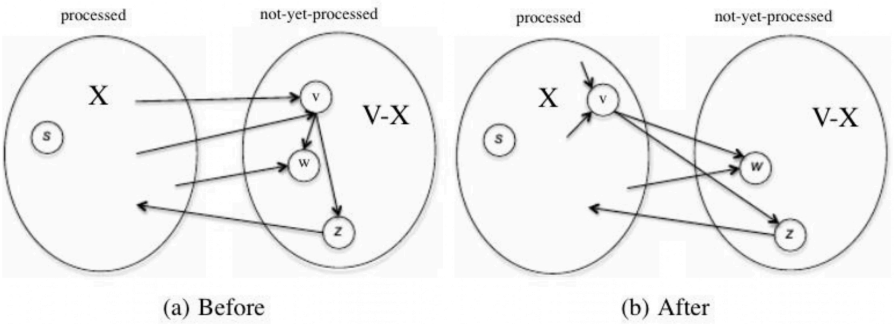
- Suppose we consider $v^* \in V - X$ with minimum key and

$$key(v^*) = len(u^*) + \ell_{u^*v^*}.$$

- Do you see?
  - (u^*, v^*) is the arc with lowest score crossing the frontier.
  - Absorb v^* in X.
- But now the *key* values are altered! How to fix?
  - Which *key* values are altered?

Dijkstra's algorithm: Min-heap of vertices

- But now the *key* values are altered! How to fix?
  - Which *key* values are altered?



- Only update *key* for out-neighbours $w \in V - X$ of $v \in X$.

Dijkstra's algorithm: $O((m+n)\log n)$

- **Set** $X = \emptyset, H =$ empty min-heap.
- **Set** $key(s) = 0$, and for all $v \neq s$, set $key(v) = \infty$.
- **Insert** all vertices in H.

Initialisation

- **while** H is not empty, **do**:
  - $w =$ ExtractMin(H).
  - **Add** w to X. Set $len(w) = key(w)$.

Total # ExtractMin = $O(n)$

Distance calculation

  - **for** every edge (w, y) with $y \in V - X$, **do**:
    - **Delete** y from H.
    - **Update** $key(y) = \min\{key(y), len(w) + \ell_{wy}\}$.
    - **Insert** y to H.

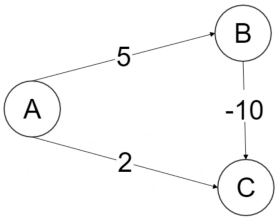
Total # delete and updates: $O(m)$

Key fixing

Dijkstra's algorithm: Negative weight?

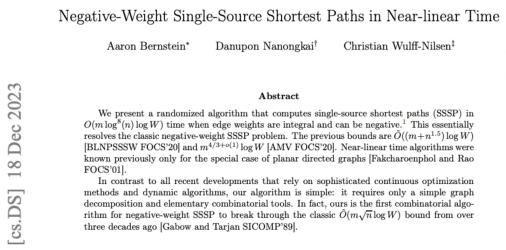
- **Quiz**: G has some negative edge weights. G has no cycles.
  - Dijkstra might loop forever.
  - Impossible to run Dijkstra.
  - Dijkstra will always halt, but can be incorrect for some vertices.
  - Dijkstra will always halt, and will be correct for all vertices.

Bellman-Ford $O(mn)$



Dijkstra's algorithm: Negative cycle?

- **Quiz**: G has a negative cycle C. G has a path from $s \to C$.
  - Dijkstra might loop forever.
  - Impossible to run Dijkstra.
  - Dijkstra will always halt, but can be incorrect for some vertices.
  - Dijkstra will always halt, and will be correct for all vertices.



Negative-Weight Single-Source Shortest Paths in Near-linear Time

Aaron Bernstein* Danupon Nanongkai† Christian Wulff-Nilsen‡

[cs.DS] 18 Dec 2023

Abstract

We present a randomized algorithm that computes single-source shortest paths (SSSP) in $O(m \log^8(n) \log W)$ time when edge weights are integral and can be negative.[1] This essentially resolves the classic negative-weight SSSP problem. The previous bounds are $\tilde{O}((m+n^{1.5}) \log W)$ [BLNPSSSW FOCS'20] and $m^{4/3+o(1)} \log W$ [AMV FOCS'20]. Near-linear time algorithms were known previously only for the special case of planar directed graphs [Fakcharoenphol and Rao FOCS'01].

In contrast to all recent developments that rely on sophisticated continuous optimization methods and dynamic algorithms, our algorithm is simple: it requires only a simple graph decomposition and elementary combinatorial tools. In fact, ours is the first combinatorial algorithm for negative-weight SSSP to break through the classic $\tilde{O}(m\sqrt{n} \log W)$ bound from over three decades ago [Gabow and Tarjan SICOMP'89].

FOCS 2022 Best Paper
ICBS Frontier of Science Award