# CS 341: ALGORITHMS

**Lecture 14: graph algorithms V – single source shortest path**

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

# DIJKSTRA'S ALGORITHM

**Single-source** shortest path
in a graph with **non-negative** **edge weights**

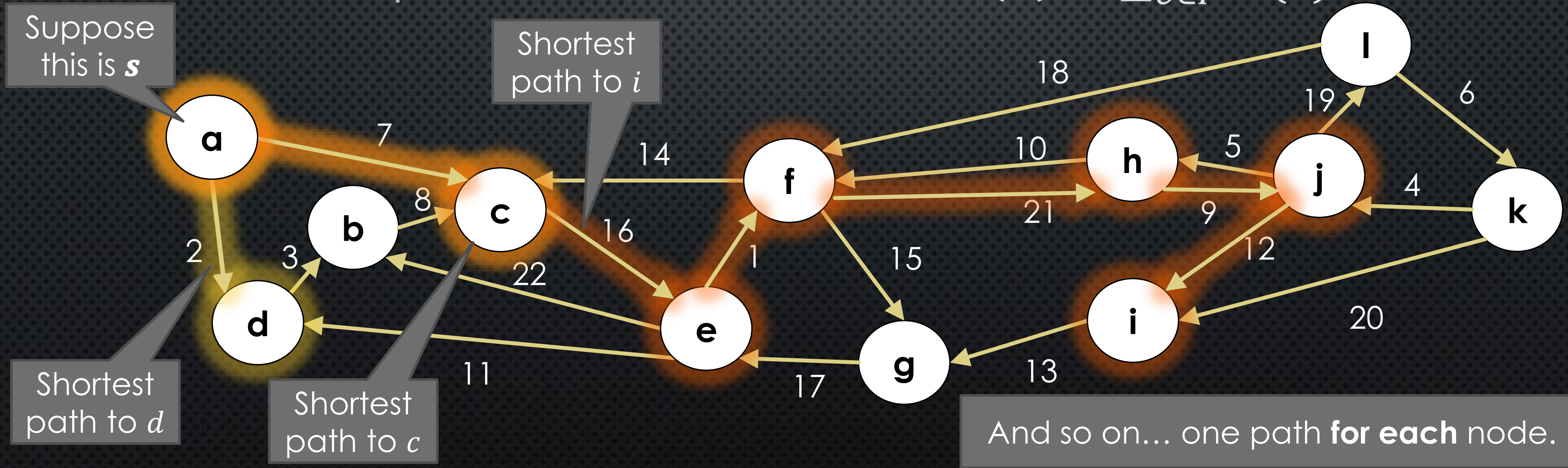
# PROBLEM: **SINGLE SOURCE** SHORTEST PATH**S** (SSSP)

**Let's study directed G.**
Can also be defined for undirected G…

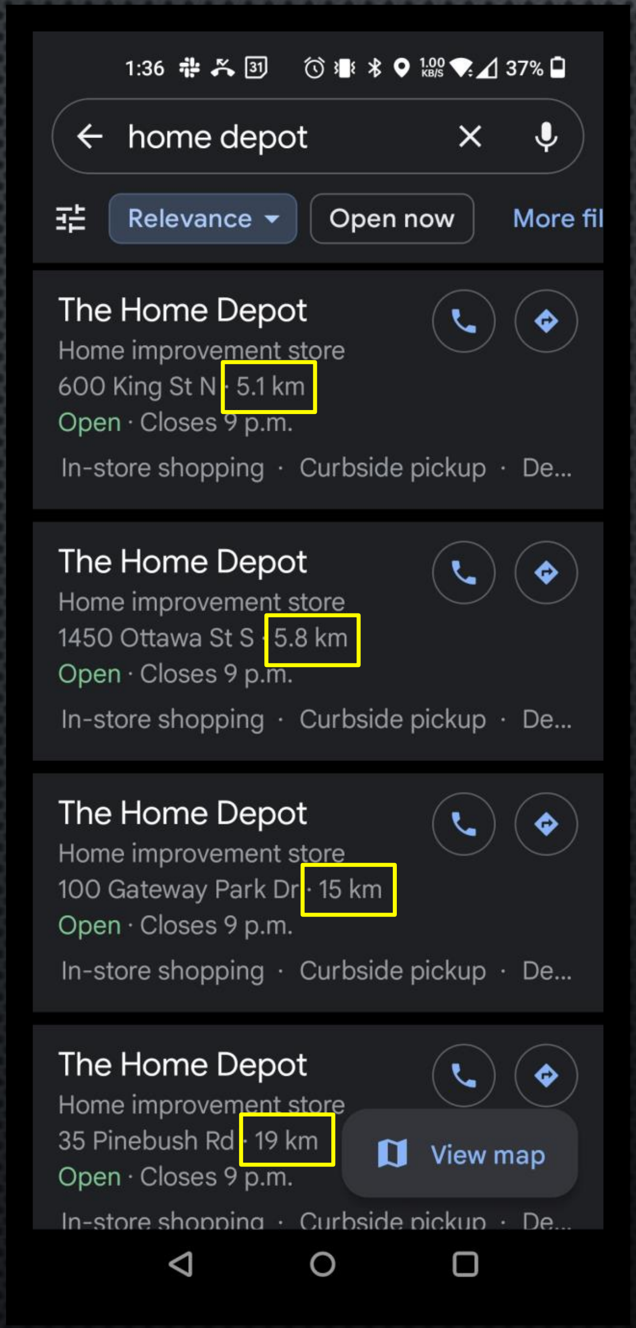
- Input: graph $G = (V, E)$ and a **non-negative** weight function $w(e)$ defined for every edge $e$

"Shortest" means minimum weight

- Problem: **for every node** $v \neq \boldsymbol{s}$, output a path $\boldsymbol{s} \rightsquigarrow v$ with the **smallest total weight** (among all paths $\boldsymbol{s} \rightsquigarrow v$)
  - I.e., **each** path $P$ should minimize $w(P) = \sum_{e \in P} w(e)$




And so on… one path **for each** node.




# APPLICATION: **DRIVING DISTANCE** TO **MANY** POSSIBLE DESTINATIONS

- Single source: from where you are
- Shortest path**s**: to **all** destinations
  - Display a subset of destinations
  - Include the optimal distances computed using SSSP algorithm
- Other heuristics… traffic? Lights?
  - **Weights** can combine many factors

[video clip]

Game AI:
path finding
with waypoints

Divide game world into **linear paths**, then send game characters in **straight lines** between waypoints

If some linear paths are much faster/slower, use **weighted** SSSP

Otherwise use BFS to find shortest sequence of waypoints (with **fewest** waypoints)

# DIJKSTRA'S ALGORITHM

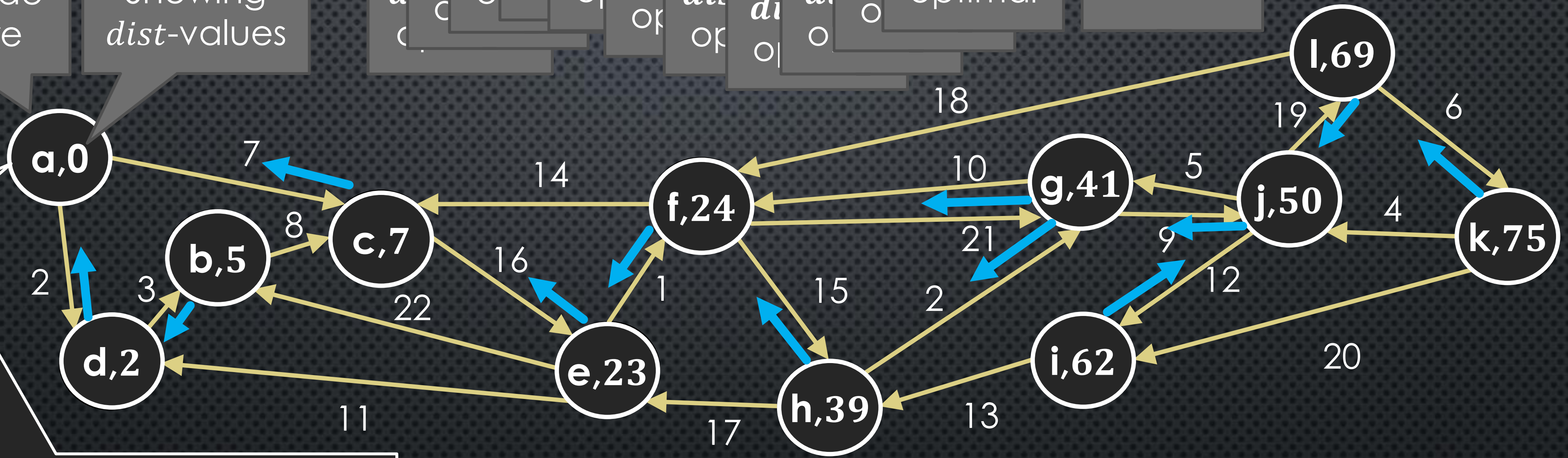
## ILLUSTRATIVE EXAMPLE

Start node $s$ is here

Showing $dist$-values

$dist[k]$ is optimal

Done!




This $dist$ is **optimal**!

Can we use this optimal $dist$ **to improve** the $dist$ of **neighbours**

We call this **relaxing** the neighbours

Key insight: after relaxing all, the smallest $dist$ (that we didn't already know was optimal) is now optimal

```
Dijkstra(adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    pq = new priority queue

    dist[s] = 0
    for u = 1..n
        pq.enqueue(u, dist[u])

    while pq is not empty
        u = pq.dequeueMin()
        for v in adj[u]
            if dist[u] + w(u,v) < dist[v]
                dist[v] = dist[u] + w(u,v)
                pred[v] = u
                pq.changePriority(v, dist[v])

    return pred, dist
```

Maintain nodes in priority order, ordered by smallest distance

Enqueue all nodes with distance $\infty$ except for $s$ with distance 0

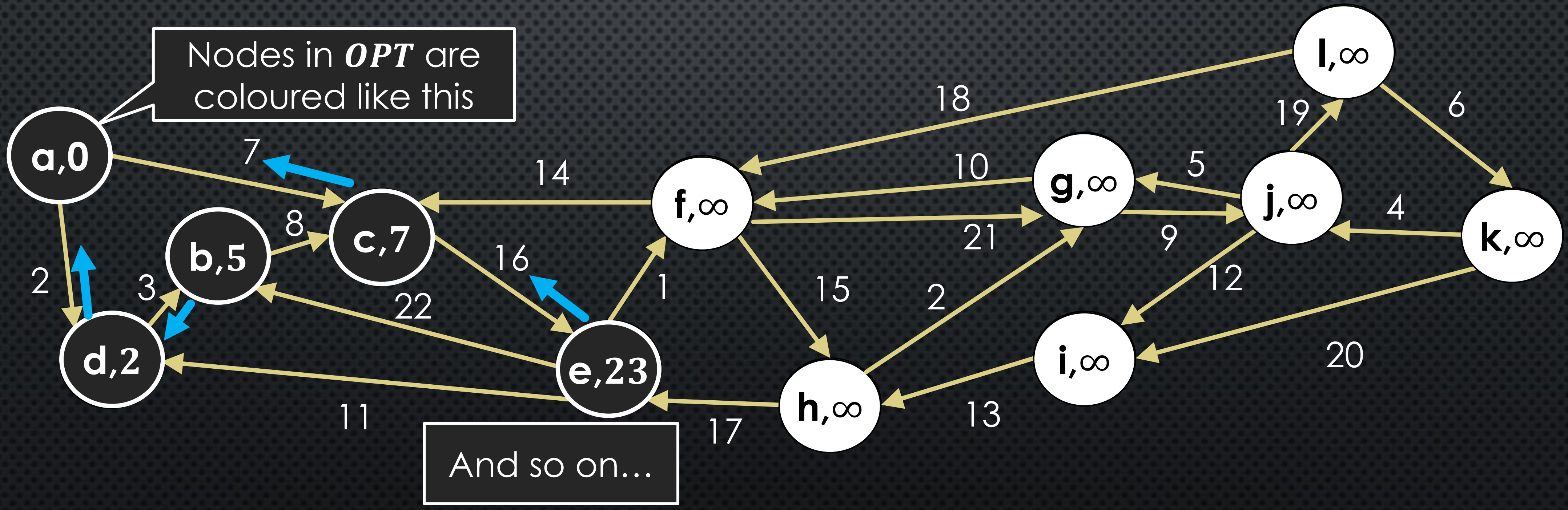
Eventually dequeue all nodes (no more enqueues)

Each dequeued node $\boldsymbol{u}$ has **optimal** $\boldsymbol{dist}$

Relax neighbour $\boldsymbol{v}$

# CORRECTNESS: INTUITION

- Dijkstra's algorithm iteratively construct a set $\boldsymbol{OPT}$ of nodes for which we **know** the shortest path from $\boldsymbol{s}$ (initially $\boldsymbol{OPT} = \{\boldsymbol{s}\}$)
  - After each relaxation step, we grow $OPT$ by adding the node in $\boldsymbol{V \backslash OPT}$ with the smallest $\boldsymbol{dist}$

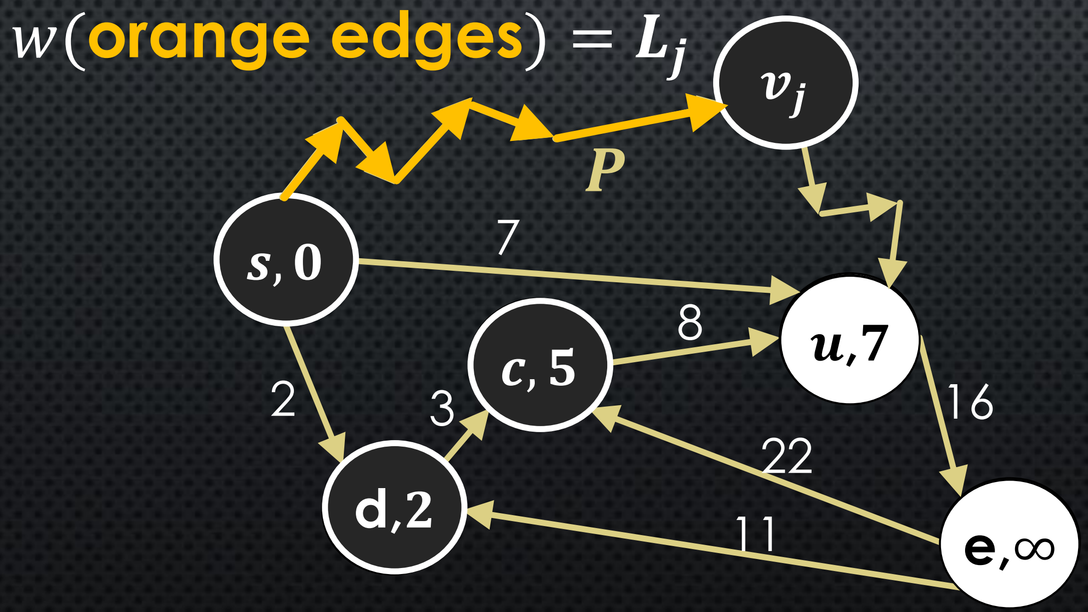


# PROOF

- **Theorem:** At the end of the algorithm, for all $u$, $dist[u]$ is exactly the total weight of the shortest $s \rightsquigarrow u$ path
- We prove this in two parts
  - $dist[u] \leq$ the total weight of the shortest $s \rightsquigarrow u$ path (case $\leq$)
  - $dist[u] \geq$ the total weight of the shortest $s \rightsquigarrow u$ path (case $\geq$)

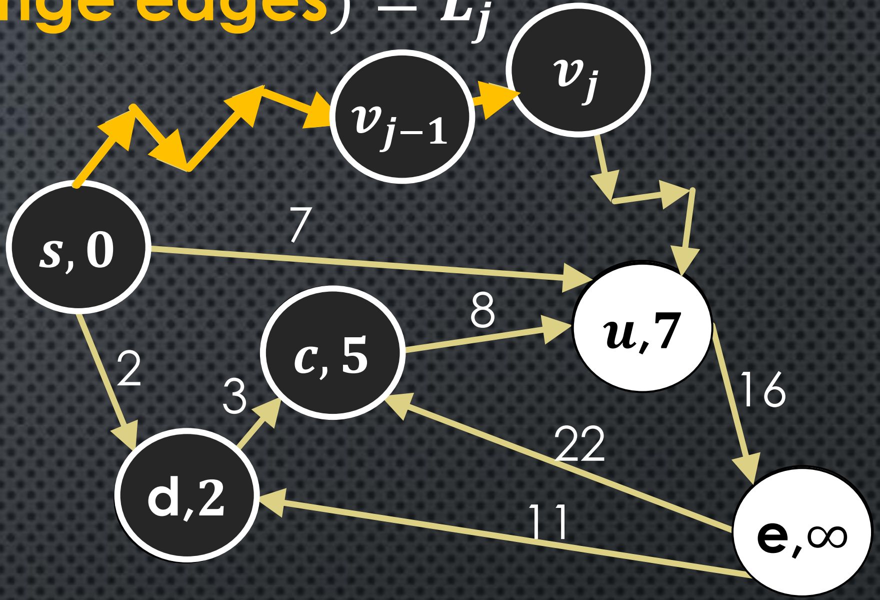
# CASE $\leq$ [ERICKSON THM.8.5]

- Let $P$ be **any arbitrary $s \rightsquigarrow u$ path** $v_0 \to v_1 \to \cdots \to v_\ell$ where $v_0 = s$ and $v_\ell = u$
- For any index $j$ let $L_j$ denote $w(v_0 \to v_1 \to \cdots \to v_j)$
- We prove by induction: $dist[v_j] \leq L_j$ for all $j$



- Prove by induction: $\forall j: \ dist[v_j] \le L_j$
- Base case: $dist[v_0] = dist[s] = 0 = L_0$
- Ind. step: **suppose** $\forall_{j>0}: \boldsymbol{dist[v_{j-1}] \le L_{j-1}}$
  - When dequeueMin() returns $\boldsymbol{v_{j-1}}$: we **check** if $dist[v_{j-1}] + w(v_{j-1}, v_j) < dist[v_j]$
  - If so, we **set** $dist[v_j] = dist[v_{j-1}] + w(v_{j-1}, v_j)$
  - If not, $dist[v_j] \le dist[v_{j-1}] + w(v_{j-1}, v_j)$
  - In **both cases**, $dist[v_j] \le dist[v_{j-1}] + w(v_{j-1}, v_j)$
    - By I.H. $dist[v_{j-1}] \le L_{j-1}$ so $dist[v_j] \le L_{j-1} + w(v_{j-1}, v_j)$
    - And $L_{j-1} + w(v_{j-1}, v_j) = L_j$ by definition
    - **So $dist[v_j] \le L_j$**

$w($**orange edges**$) = \boldsymbol{L_j}$




This proves $dist[u] \le L_u$, the weight of an **arbitrary** $s \rightsquigarrow u$ path.

So $dist[u] \le$ the weight of EVERY $s \rightsquigarrow u$ path.

**Including the shortest** $s \rightsquigarrow u$ path!

# CASE $\geq$

- Let $\boldsymbol{P'}$ be the path $s \to \cdots \to pred\left[pred[u]\right] \to pred[u] \to u$
  - I.e., the reverse of following $pred$ pointers from $u$ back to $s$
- We show $dist[u]$ is as long as **this** path
  (and hence as long as the **shortest** path)
- Denote the nodes in $P'$ by $v_0, v_1, \dots, v_\ell$ where $v_0 = s$ and $v_\ell = u$
- Let $L_j = w\left(v_0 \to v_1 \to \cdots \to v_j\right)$
- **Prove by induction**: $\forall_{j>0} : dist\left[v_j\right] = L_j$
- Base case: $dist[v_0] = dist[s] = 0 = L_0$

# CASE $\geq$

- $\boldsymbol{P' = v_0 \to \cdots \to v_\ell = s} \to \cdots \to pred[pred[u]] \to pred[u] \to u$
- $L_j = w(v_0 \to v_1 \to \cdots \to v_j)$
- **Inductive step**: suppose $\forall_{j>0} : dist[v_{j-1}] = L_{j-1}$
- When we set $pred[v_j] = v_{j-1}$, we set $\boldsymbol{dist[v_j] = dist[v_{j-1}] + w(v_{j-1}, v_j)}$

**Recall:**

```
if dist[u] + w(u,v) < dist[v]
    dist[v] = dist[u] + w(u,v)
    pred[v] = u
```

- By I.H., $dist[v_j] = \boldsymbol{L_{j-1}} + w(v_{j-1}, v_j)$
- By definition $L_j = L_{j-1} + w(v_{j-1}, v_j)$
- So $\boldsymbol{dist[v_j] = L_j}$

So $dist[u]$ = length of a **particular path** $P'$ **in the graph**

And length of $P'$ is $\geq$ length of **shortest path**

So $dist[u] \geq$ length of **shortest** path $s \rightsquigarrow u$

So $dist[u]$ is both $\leq$ and $\geq$ to the length of the shortest $s \rightsquigarrow u$ path!

That means it's **equal** to the length of the shortest path!

# RUNTIME

```
Dijkstra(adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    pq = new priority queue

    dist[s] = 0
    for u = 1..n
        pq.enqueue(u, dist[u])

    while pq is not empty
        u = pq.dequeueMin()
        for v in adj[u]
            if dist[u] + w(u,v) < dist[v]
                dist[v] = dist[u] + w(u,v)
                pred[v] = u
                pq.changePriority(v, dist[v])

    return pred, dist
```

$O(n)$

$O(\log n)$

$O(n \log n)$

$O(\log n)$

$O(\log n)$

- Each node enqueued and dequeueMin'd once
  - $O(n \log n)$
- For each dequeueMin, do $O(\log n)$ per neighbour
  - $O(\log n)$ for **each edge**
  - $O(m \log n)$ w/adjacency lists
- **Total time $O((n+m)\log n)$**

Space complexity?

# OUTPUTTING ACTUAL SHORTEST PATH(S)?

- To compute the actual shortest **path** $s \rightsquigarrow t$
- Inspect $pred[t]$
  - If it is NULL, there is no such path
  - Otherwise, follow $pred$ pointers back to $s$, and return the **reverse** of that path

```
Dijkstra(adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    OPT = [false, false, ..., false]

    dist[s] = 0
    OPT[s] = true
    numOpt = 1

    while numOpt < n
        choose u such that OPT[u] == false
            and dist[u] is minimized
        OPT[u] = true
        numOpt = numOpt + 1
        for v = adj[u]
            if dist[u] + w(u,v) < dist[v]
                dist[v] = dist[u] + w(u,v)
                pred[v] = u

    return pred, dist
```

# AN **ALTERNATIVE** IMPLEMENTATION

- Instead of using a **priority queue**
- Find the minimum dist[] node to add to OPT via **linear search**
- **Runtime?**
  - $O(n^2)$
- **Better or worse than $O((n+m)\log n)$?**

# WEBSITE DEMONSTRATING DIJKSTRA'S ALG

- https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html

# BELLMAN-FORD

**Single-source** shortest path
in a graph with possibly **negative** edge weights
but **no negative cycles**

# Shortest Paths and Negative Weight Cycles

Subsequent algorithms we will be studying will solve shortest path problems as long as there are no cycles having negative weight.

If there is a negative weight cycle, then there is no shortest path (why?).

There is still a shortest simple path, but there are apparently no known efficient algorithms to find the shortest simple paths in in graphs congtaining negative weight cycles.

If there are no negative weight cycles, we can assume WLOG that shortest paths are simple paths (any path can be replaced by a simple path having the same weight).

Negative weight edges in an undirected graph are not allowed, as they would give rise to a negative weight cycle (consisting of two edges) in the associated directed graph.

# BELLMAN-FORD

The *Bellman-Ford algorithm* solves the single source shortest path problem in any directed graph without negative weight cycles.

The algorithm is very simple to describe:

Repeat $n-1$ times: *relax* every edge in the graph (where *relax* is the updating step in Dijkstra's algorithm).

```
BellmanFord(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
    return (D, pred)
```

$O(n)$

$O(n)$

$O(1)$

$O(n)$ outer iterations

$O(m)$ inner iterations **per** outer iteration

$O(1)$ work per inner iteration

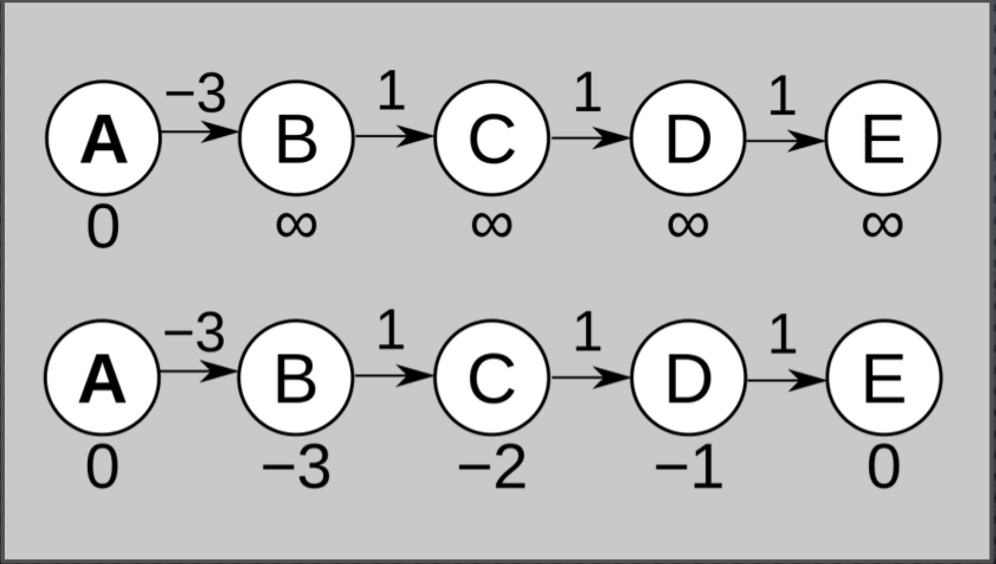
Total $O(nm)$

Could be $O(n^3)$

# BEST CASE EXECUTION

**It technically suffices to do one** iteration of the outer loop

Edges *happen* to be processed **left to right** by the inner loop

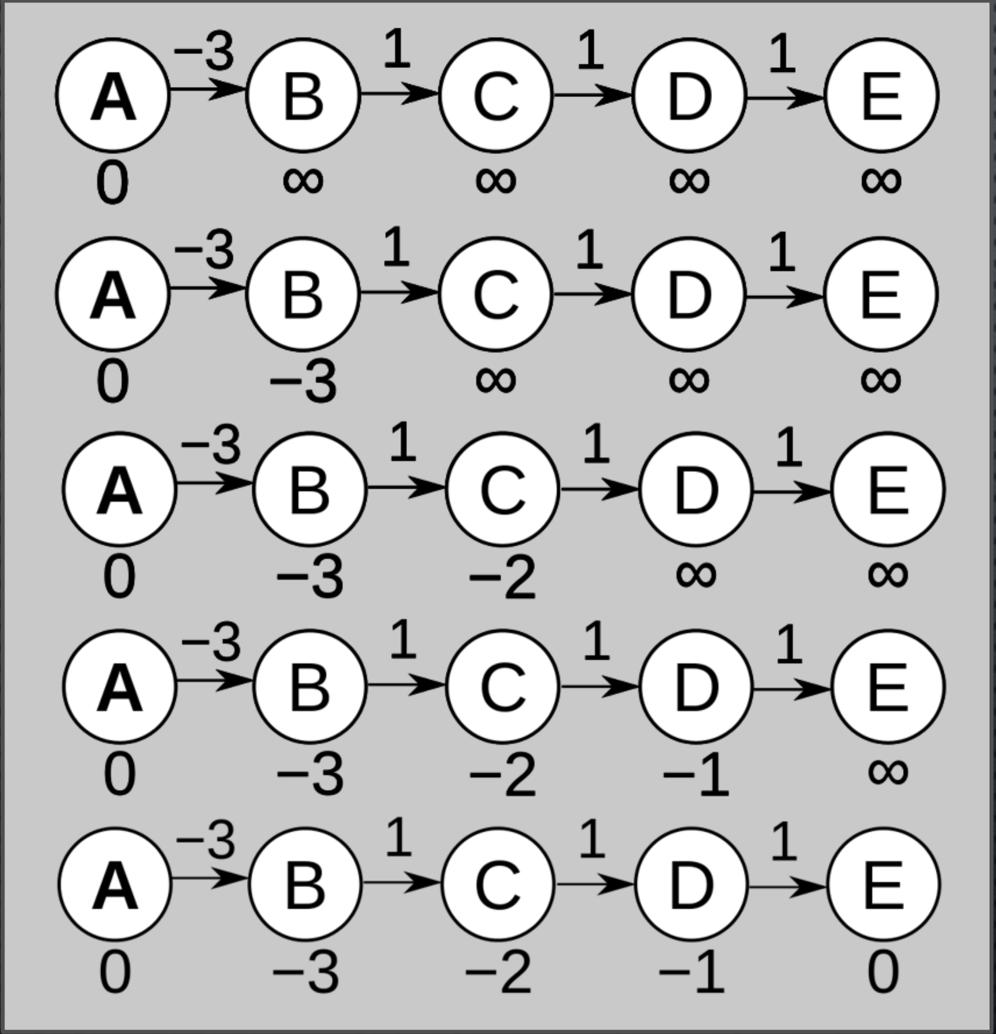



```
BellmanFord(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
    return (D, pred)
```

# WORST CASE EXECUTION

Need $n$ iterations of outer loop

Edges *happen* to be processed **right to left** by the inner loop




```
BellmanFord(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
    return (D, pred)
```

Since the longest possible path without a cycle can be $n-1$ edges, the edges must be scanned $n-1$ times to ensure the shortest path has been found for all nodes.

**Dijkstra's** is similar, but *consistently* achieves good ordering using its priority queue

# WHY BELLMAN-FORD WORKS

- **Not** going to prove this (by induction), but the crucial **lemma** is:
  - After $i$ iterations of the outer *for*-loop,
    - if $D[u] \neq \infty$, it is equal to the weight of some path $s \rightsquigarrow u$; **and**
    - if there is a path $P = (s \rightsquigarrow u)$ with **at most $\boldsymbol{i}$ edges**, then $D[u] \leq w(P)$
- So, after $n - 1$ iterations, if $\exists$ path $P$ with at most $n - 1$ edges, then $D[u] \leq w(P)$. (Note: any *more* edges would create a cycle.)
- So, if $u$ is reachable from $s$, then $D[u]$ is the length of the shortest simple path (no cycles) from $s$ to $u$

Of course every simple path has at most $n - 1$ edges

So what if we do **another iteration**, and some $D[u]$ improves?

There is a negative cycle!

# A MORE DETAILED IMPLEMENTATION

```
BellmanFordCheck(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        changed = false
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
                changed = true
        if not changed
            exit loop
        if i == n // assert: changed == true
            return NEGATIVE_CYCLE
    return (D, pred)
```

- With early stopping
- and checking for negative cycles

# BONUS SLIDE

- Why can't you just modify a graph with negative weights by: finding the minimum edge weight Wmin, and adding that to each edge, so you no longer have negative edges and can run Dijkstra's algorithm?
- **Exercise:** can you find a graph for which this will cause Dijkstra's algorithm to return the **wrong answer**?
- **Solution:**
  - Consider a graph with 5 nodes: s, a, b, c, t
  - And edges s->a with weight -10, b->t with weight 10
    s->b weight -1, b->c weight -1, c->t weight -1
  - What happens if you modify this graph as proposed, then run Dijkstra's to find the shortest path from s to t?