UNIT 6
GRAPHS PART III:
Single-Source Shortest Paths

Iris Hui-Ru Jiang
Spring 2024

Department of Electrical Engineering
National Taiwan University

Outline

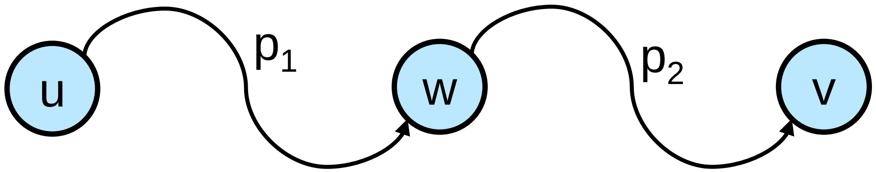
- Content:
  - Optimal substructure: Longest paths? Shortest paths?
  - Single-Source Shortest Paths (SSSP)
  - Dijkstra's algorithm
  - A* search (appendix)
  - SSSP in DAG
  - Bellman-Ford algorithm
  - Difference constraints in linear programming
- Reading:
  - Chapter 22

Optimal Substructure

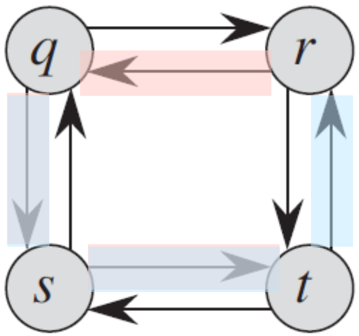
Unweighted shortest path

Unweighted: unit-weight

- **Don't assume optimal substructure can always apply!**
- **Unweighted shortest path:** Given a directed graph $G=(V,E)$, find a ~~simple~~ path from u to v containing the fewest edges
  - Optimal substructure?

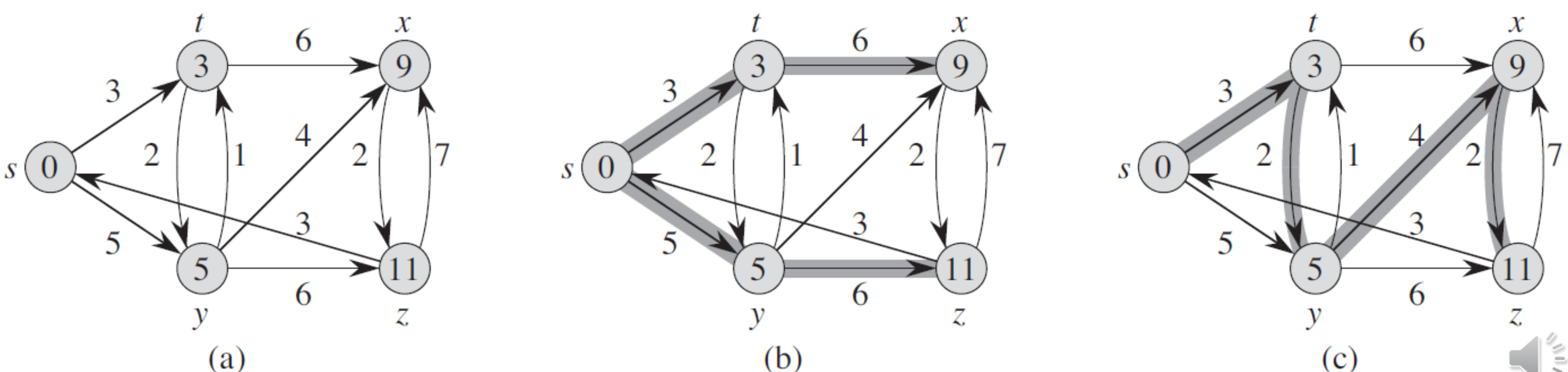


- **Unweighted longest simple path:** Given a directed graph $G=(V,E)$, find a simple path from u to v containing the most edges
  - Optimal substructure?



Single-Source Shortest Paths (SSSP)

- **The Single-Source Shortest Path (SSSP) Problem**
  - **Given:** A **directed** graph *G=(V, E)* with edge weights, and a specific **source node** *s*
  - **Goal:** Find a minimum weight path (or cost) from *s* to every other node in *V*
- Applications: weights can be distances, times, wiring cost, delay, etc.
- **Special case:** BFS finds shortest paths for the case when all edge weights are 1 (the same)

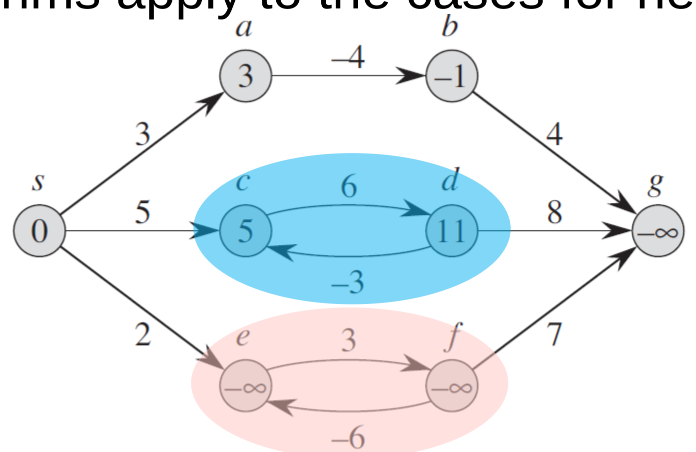


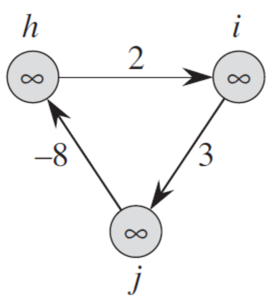
Negative Cycles in Shortest Paths

- A weighted, directed graph $G = (V, E)$ with the weight function $w: E \rightarrow \mathbb{R}$.
  - Weight of path $p = \langle v_0, v_1, \ldots, v_k \rangle$: $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
  - $\delta(u, v)$: **Shortest-path weight** from u to v

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

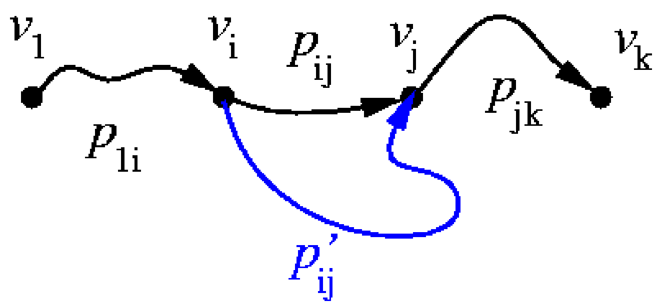
- **Warning!** negative-weight edges/cycles are a problem.
  - Cycle $\langle e, f, e \rangle$ has weight $-3 < 0 \rightarrow$ **$\delta(s, g) = -\infty$.**
  - Vertices h, i, j not reachable from $s \rightarrow$ **$\delta(s, h) = \delta(s, i) = \delta(s, j) = \infty$.**
- Algorithms apply to the cases for negative-weight edges/cycles??



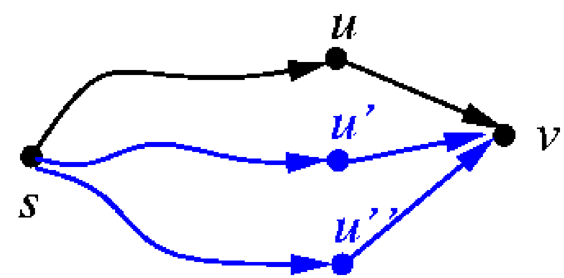


Optimal Substructure of a Shortest Path

- Subpaths of shortest paths are shortest paths.
  - Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from vertex v_1 to vertex v_k, and let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j, $1 \le i \le j \le k$. Then, p_{ij} is a shortest path from v_i to v_j.



subpaths of shortest paths

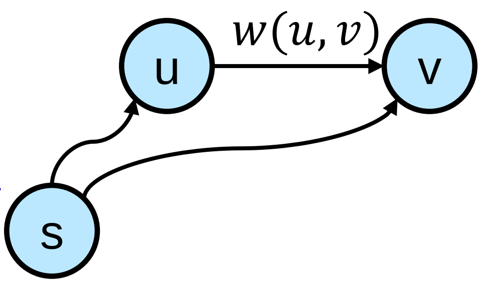


- Suppose that a shortest path p from a source s to a vertex v can be decomposed into $s \overset{p'}{\leadsto} u \to v$. Then, $\delta(s, v) = \delta(s, u) + w(u, v)$.
- For all edges $(u, v) \in E$, $\delta(s, v) \le \delta(s, u) + w(u, v)$.

Triangle Inequality

- **Optimal substructure:**
  - Subpaths of shortest paths are shortest paths
  - $\delta(s,v) = \min_u\{w(p): p: s \rightsquigarrow u \longrightarrow v\}$ if p exists; ∞, otherwise

- **Triangle inequality:**
  - $\forall (u,v) \in E, \delta(s,v) \leq \delta(s,u) + w(u,v)$
  - "=" holds when $u = \text{argmin}\{w(p): p: s \rightsquigarrow u \longrightarrow v\}$

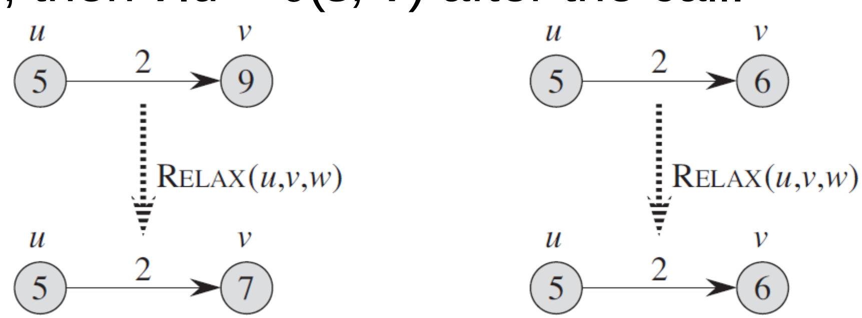


Relaxation ★★★★★

```
Initialize-Single-Source(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    // upper bound on the weight of
    // a shortest path from s to v
    v.π = NIL    // predecessor of v
s.d = 0
```

```
Relax(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

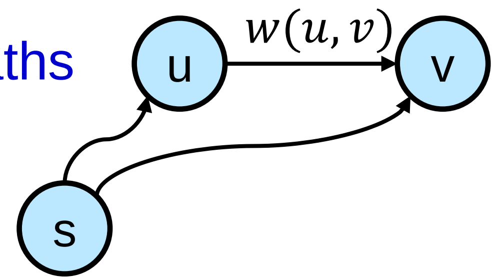
- $v.d \le u.d + w(u, v)$ after calling Relax(u, v, w).
- $v.d \ge \delta(s, v)$ during the relaxation steps; once $v.d$ achieves its lower bound $\delta(s, v)$, it never changes.
- Let $s \rightsquigarrow u \longrightarrow v$ be a shortest path. If $u.d = \delta(s, u)$ prior to the call Relax(u, v, w), then $v.d = \delta(s, v)$ after the call.



Before: $v.d > u.d + w(u, v)$ $\quad$ $v.d <= u.d + w(u, v)$

Quick Summary

- **Optimal substructure:**
  - Subpaths of shortest paths are shortest paths
- **Triangle inequality:**
  - $\forall(u,v) \in E, \delta(s,v) \le \delta(s,u) + w(u,v)$



- Upper-bound property:
  - $\forall v \in V, v.d \ge \delta(s,v)$; once $v.d$ reach $\delta(s,v)$, it never changes
- Convergence property:
  - If $s \sim u \to v$ is a shortest path, $u.d = \delta(s,u)$ prior to relaxing (u,v), $v.d = \delta(s,v)$ afterward
- Path-relaxation property:
  - If $p = \langle s, v_1, \dots, v_k \rangle$ is a shortest path and edges are relaxed in order, $v_k.d = \delta(s,v_k)$ regardless of any other relaxation steps that occur
- Predecessor-subgraph property:
  - Once $v.d = \delta(s,v) \forall v \in V$, the predecessor subgraph is a **shortest-paths tree** rooted at s

Dijkstra’s Algorithm: Greedy

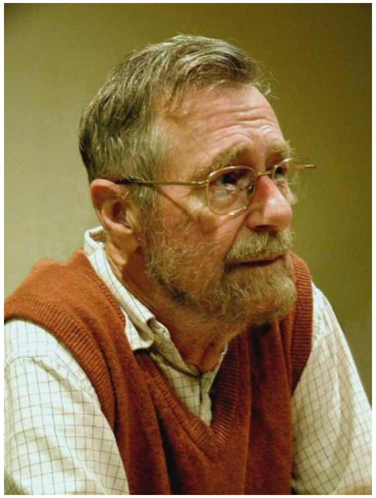
Edsger W. Dijkstra 1959

R. C. Prim: *Shortest connection networks and some generalizations*.
In *Bell System Technical Journal*, 36 (1957), pp. 1389–1401.

E. W. Dijkstra: *A note on two problems in connexion with graphs*.
In *Numerische Mathematik*, 1 (1959), S. pp. 269–271.

Edsger W. Dijkstra (1930--2002)

- 1972 Recipient of the ACM Turing Award



His advice to a promising researcher, who asked how to select a topic for research: "Do only what only you can do"

If you want more effective programmers,
you will discover that they should not waste their time debugging,
they should not introduce the bugs to start with.

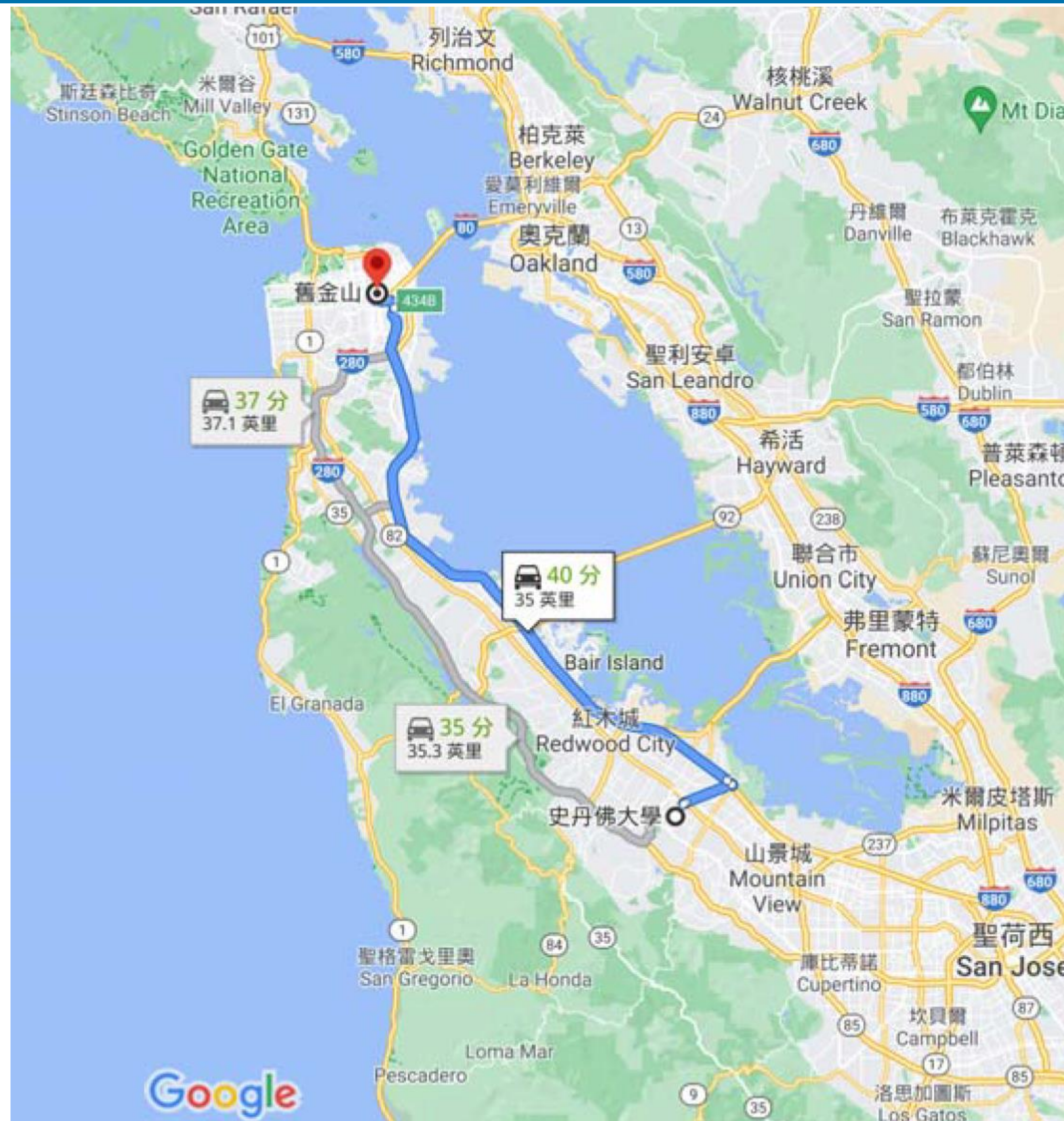
Program testing can be a very effective way to show the presence of bugs,
but it is hopelessly inadequate for showing their absence.

-- Turing Award Lecture 1972, the humble programmer

http://www.cs.utexas.edu/users/EWD/obituary.html

Google Map

Shortest path from Stanford University to San Francisco



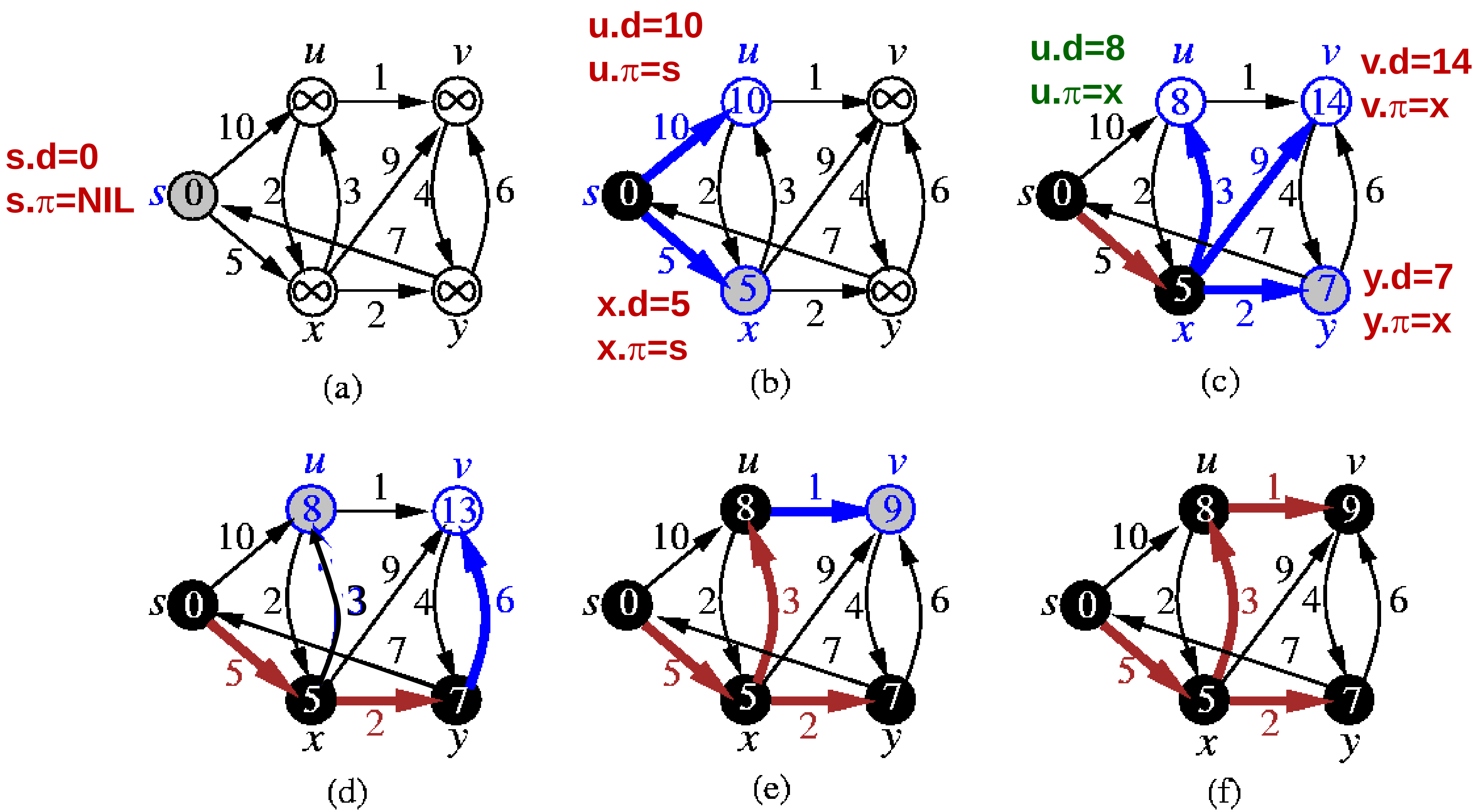
Dijkstra's Shortest-Path Algorithm

Dijkstra(G, w, s)
// S: explored vertices
// key: shortest path estimate $*.d$
1. Initialize-Single-Source(G, s)
2. $S = \emptyset$
3. $Q = G.V$ // priority queue
4. **while** $Q \neq \emptyset$
5. $\quad u$ = Extract-Min(Q)
6. $\quad S = S \cup \{u\}$
7. $\quad$ **for** each vertex $v \in G.Adj[u]$
8. $\qquad$ Relax(u, v, w)

MST-Prim(G,w,r)
1. **for** each vertex $u \in G.V$
2. $\quad u.key = \infty$
3. $\quad u.\pi$ = NIL
4. $r.key = 0$
5. $Q = G.V$ $\qquad O(V \lg V + E \lg V)$
6. **while** $Q \neq \emptyset$
7. $\quad u$ = Extract-Min(Q)
8. $\quad$ **for** each vertex $v \in G.Adj[u]$
9. $\qquad$ **if** $v \in Q$ and $w(u,v) < v.key$
10. $\qquad\quad v.\pi = u$
11. $\qquad\quad v.key = w(u,v)$

- A greedy algorithm
  - Greedy choice: Every time, choose the vertex nearest to the source
- Works only when all **edge weights are nonnegative.**
- Executes essentially the same as Prim's algorithm.
- Naive analysis: $O(V^2)$ time by using adjacency lists.

Example: Dijkstra's Shortest-Path Algorithm



Runtime Analysis of Dijkstra's Algorithm

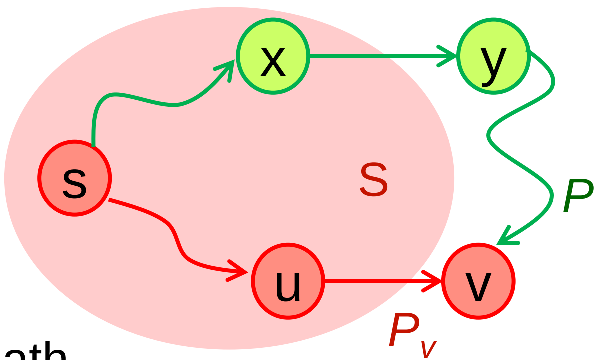
```
Dijkstra(G, w, s)
1. Initialize-Single-Source(G, s)
2. S = ∅
3. Q = G.V
4. while Q ≠ ∅
5.     u = Extract-Min(Q)
6.     S = S ∪ {u}
7.     for each vertex v ∈ G.Adj[u]
8.         Relax(u, v, w)
```

- Q is implemented as a linear array: $O(V^2)$.
  - Line 5: $O(V)$ for Extract-Min, so $O(V^2)$ with the **while** loop.
  - Lines 7--8: $O(E)$ operations, each takes $O(1)$ time.
- Q is implemented as a binary heap: $O(E \lg V)$.
  - Line 5: $O(\lg V)$ for Extract-Min, so $O(V \lg V)$ with the **while** loop.
  - Lines 7--8: $O(E)$ operations, each takes $O(\lg V)$ time for Decrease-Key (maintaining the heap property after changing a node).
- Q is implemented as a Fibonacci heap: amortized $O(E + V \lg V)$.

Correctness

```
while Q ≠ ∅
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        Relax(u, v, w)
```

- Loop invariant: Consider the set S at any point in the algorithm's execution. For each node $v \in S$, $v.d = \delta(s,v)$.
- Pf: Proof by induction on $|S|$
  - Basis step: trivial for $|S| = 1$.
  - Inductive step:
    - Hypothesis: true for iteration $k > 1$.
    - Grow S by adding v;
      let (u, v) be the final edge on our s-v path P_v.
    - By induction hypothesis, P_u is the shortest s-u path.
    - Consider any other s-v path P; P must leave S somewhere; let y be the first node on P that is not in S, and $x \in S$ be the node just before y.
    - P cannot be shorter than P_v because it is already at least as long as P_v by the time it has left the set S.
    - At iteration k+1, $v.d = u.d + w(u, v) \leq x.d + w(x, y) \leq w(P)$



Q: Why $w(u,v) \geq 0$?

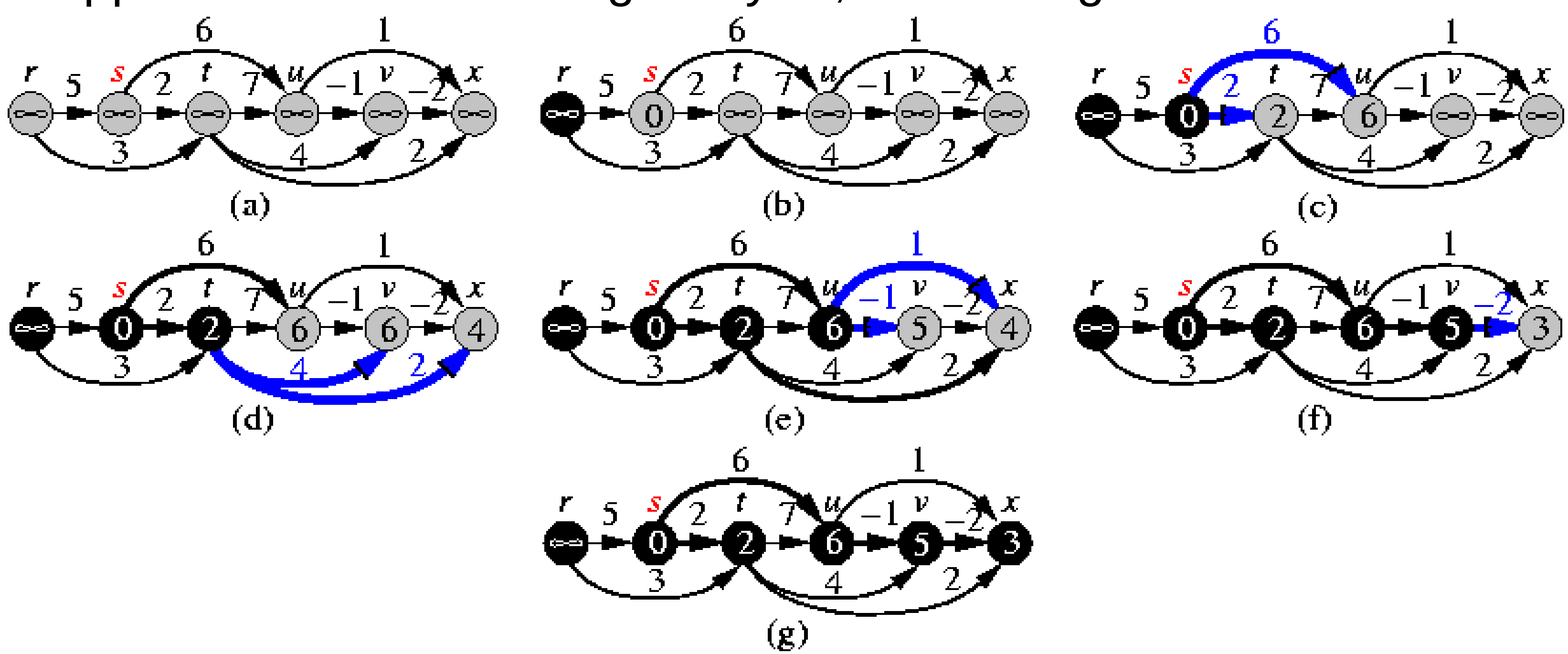
SSSPs in Directed Acyclic Graphs

Special case

SSSPs in Directed Acyclic Graphs (DAGs)

```
DAG-Shortest-Paths(G, w, s)
1. topologically sort the vertices of G
2. Initialize-Single-Source(G, s)
3. for each vertex u taken in topologically sorted order
4.       for each vertex v ∈ G.Adj[u]
5.             Relax(u, v, w)
```

- Time complexity: $O(V+E)$ (adjacency-list representation).
- Applications: circuit timing analysis, scheduling



Bellman-Ford Algorithm: DP

Richard E. Bellman
Lester R. Ford, Jr.

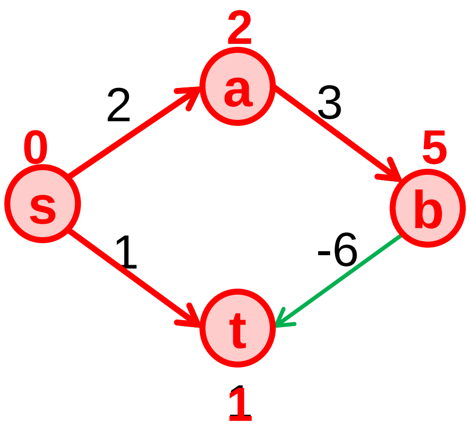
R. E. Bellman 1920—1984
Inventor of DP, 1953

Recap: Dijkstra's Algorithm

```
Dijkstra(G, w, s)
Initialize-Single-Source(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        Relax(u, v, w)
```

$w(u,v) \geq 0$

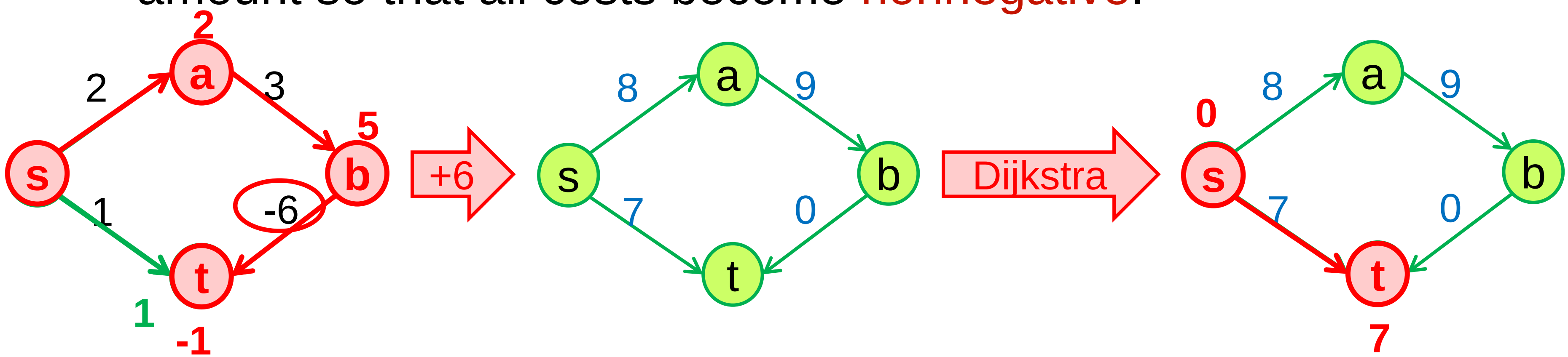
- Q: What if negative edge costs?



Q: What's wrong with s-a-b-t path?

Modifying Dijkstra's Algorithm?

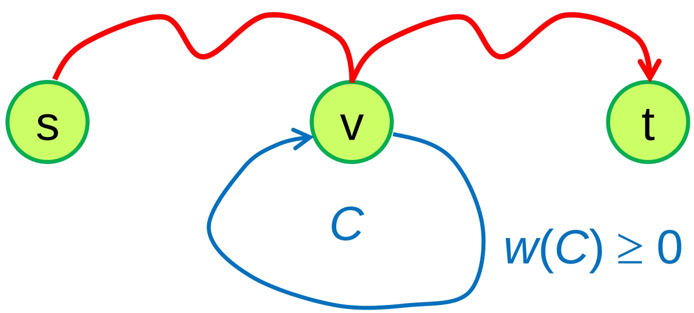
- Observation: A path that starts on a cheap edge may cost more than a path that starts on an expensive edge, but then compensates with subsequent edges of negative cost.
- Reweighting: Increase the costs of all the edges by the same amount so that all costs become nonnegative.



- Q: What's wrong?!
- A: Adapting the costs changes the minimum-cost path

Simple Path or Not?

- If G has no negative cycles, then there is a shortest path from s to t that is **simple** (i.e., does not repeat nodes), and hence has at most $|V|$-1 edges.
- Pf:
  - Suppose the shortest path P from s to t repeats a node v.



  - Since every cycle has nonnegative cost, we could remove the portion of P between consecutive visits to v resulting in a simple path Q of no greater cost and fewer edges.
    - $w(Q) = w(P) – w(C) \leq w(P)$

The Bellman-Ford Algorithm for SSSP

```
Bellman-Ford(G,w,s)
Initialize-Single-Source(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        Relax(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

```
Initialize-Single-Source(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

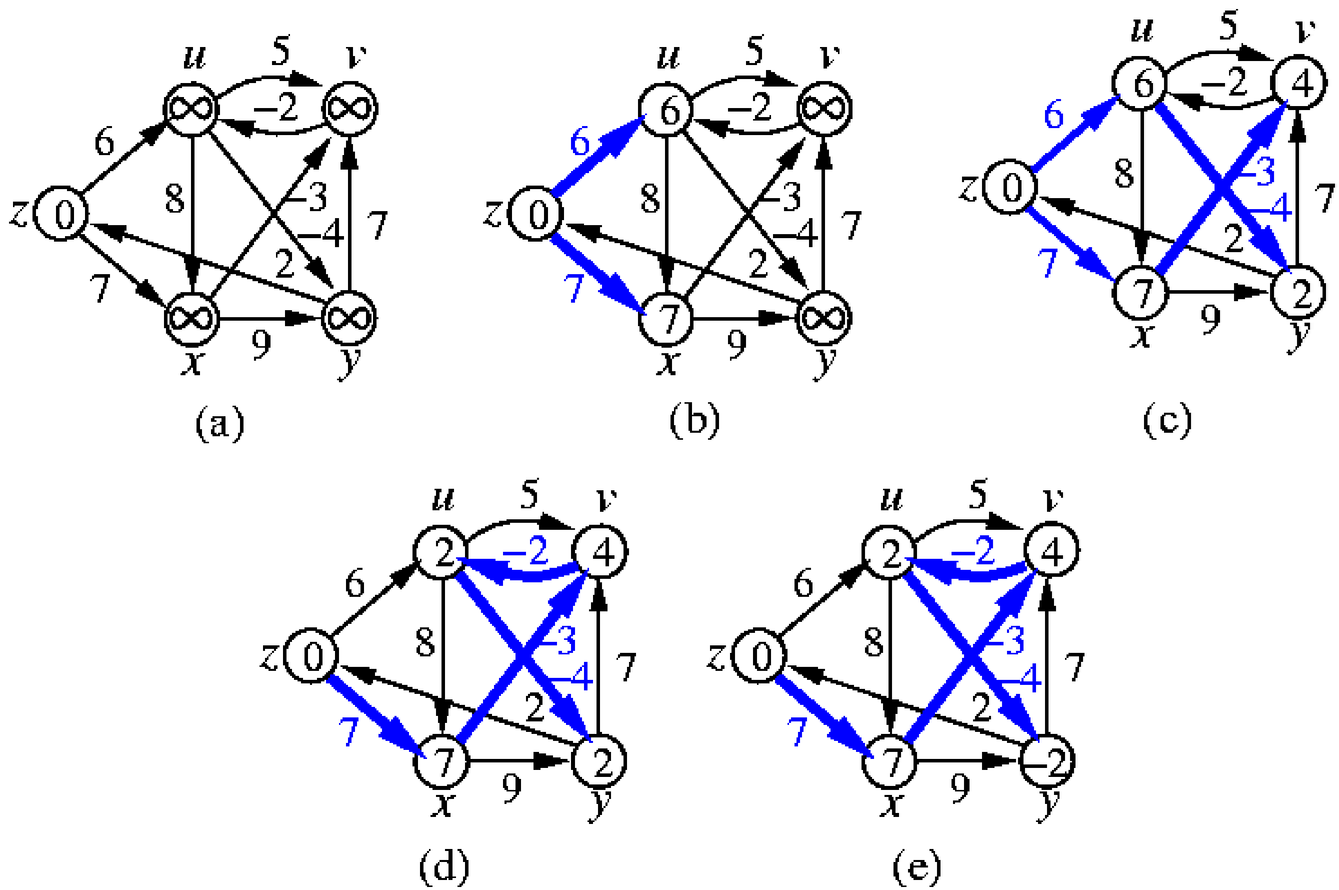
```
Relax(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

- Solve the case where **edge weights can be negative.**
- Return FALSE if there exists a **negative-weight cycle** reachable from the source; TRUE otherwise.
- Time complexity: $O(VE)$.

Induction on the number of edges

Example: The Bellman-Ford Algorithm

relax edges in lexicographic order: (u, v), (u, x), (u, y), ..., (z, u), (z, x)

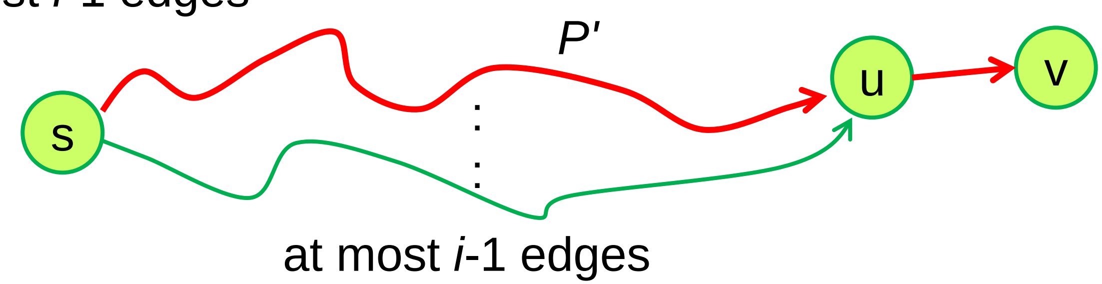


Path-relaxation property:
If $p = \langle s, v_1, \dots, v_k \rangle$ is a shortest path and edges are relaxed in order, $v_k.d = \delta(s, v_k)$ regardless of any other relaxation steps that occur

Bellman-Ford Algorithm: Correctness

- Induction on edges
- $\delta(s, v, i)$ = length of shortest s-v path P with at most i edges
  - $\delta(s, t, |V|-1)$ = length of shortest s-t path
  - Case 1: P uses at most i-1 edges
    - $\delta(s, v, i) = \delta(s, v, i-1)$

  - Case 2: P uses exactly i edges
    - $\delta(s, v, i) = \delta(s, u, i-1) + w(u,v)$
    - If (u, v) is the final edge, then P uses (u, v) and selects the shortest s-u path using at most i-1 edges

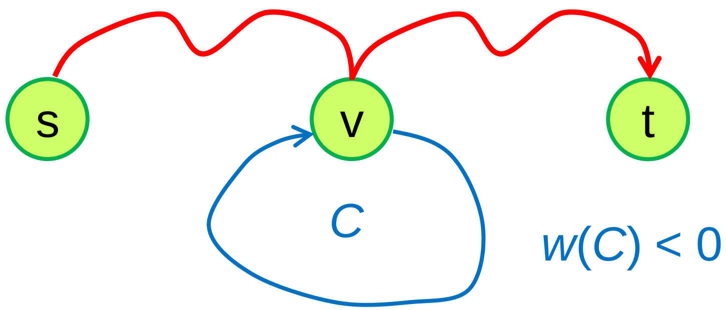


$\delta(s, v) = \delta(s, u) + w(u, v)$

For all edges $(u, v) \in E$, $\delta(s, v) \le \delta(s, u) + w(u, v)$

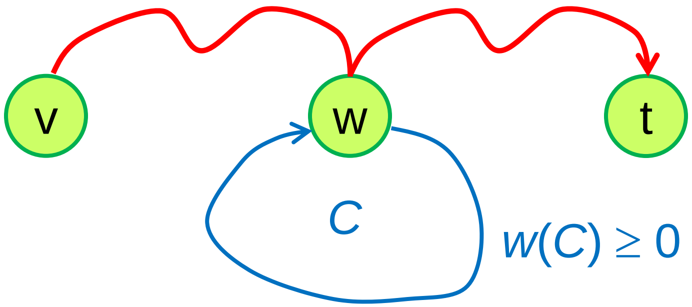
Negative Cycles?

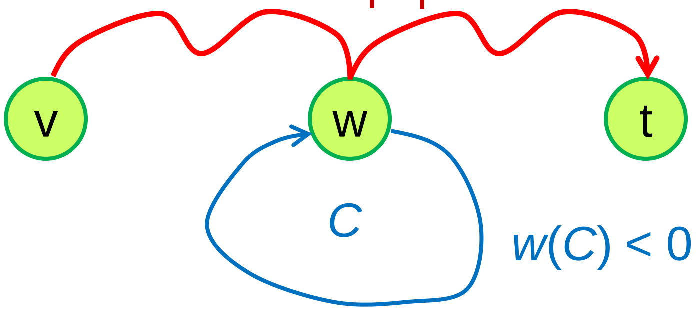
- If a *s*-*t* path in a general graph *G* passes through node *v*, and *v* belongs to a negative cycle *C*, Bellman-Ford algorithm fails to find the shortest *s*-*t* path.
  - Reduce cost over and over again using the negative cycle



Negative Cycle Detection by Bellman-Ford

- If $\delta(s, v, |V|) = \delta(s, v, |V|\text{-}1)$ for all v, then no negative cycles.
  - Bellman-Ford: $\delta(s, v, i) = \delta(s, v, |V|\text{-}1)$ for all v and $i \geq |V|$.

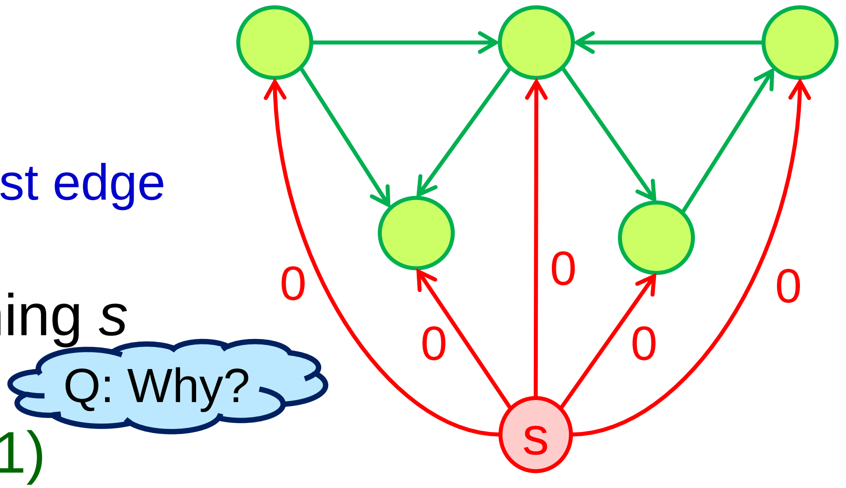




- If $\delta(s, v, |V|) < \delta(s, v, |V|\text{-}1)$ for some v, then shortest path contains a negative cycle.
- Pf: by contradiction
  - Since $\delta(s, v, |V|) < \delta(s, v, |V|\text{-}1)$, P has exactly $|V|$ edges.
  - Every path using at most $|V|$-1 edges costs more than P.
  - (By pigeonhole principle,) P must contain a cycle C.
  - If C were not a negative cycle, deleting C yields a s-v path with $< |V|$ edges and no greater cost. $\rightarrow\leftarrow$

Detecting Negative Cycles by Bellman-Ford

- Augmented graph *G′* of *G*
  1. Add new node *s*
  2. Connect all nodes to *s* with 0-cost edge
- *G* has a negative cycle
  iff *G′* has a negative cycle reaching *s*



- Check if $\delta(s, v, |V|) = \delta(s, v, |V|-1)$
  – If yes, no negative cycles
  – If no, then extract cycle from shortest path from *s* to *v*

- Procedure:
  – Build the augmented graph *G′* for *G*
  – Run Bellman-Ford on *G′* for $|V|$ iterations (instead of $|V|-1$)
  – Upon termination, Bellman-Ford predecessor variables trace a negative cycle if one exists

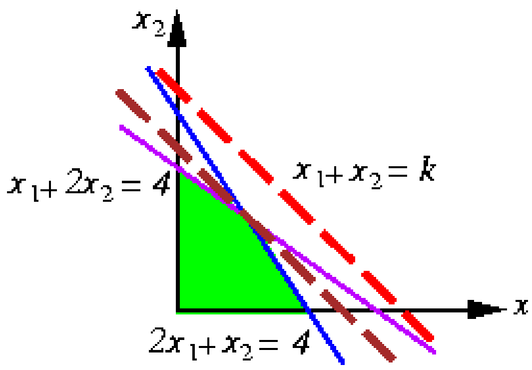
Linear Programming

- **Linear programming (LP):** Mathematical program in which the **objective function** is **linear** in the unknowns and the **constraints** consist of **linear** equalities (inequalities).
- Standard form:

$$\begin{aligned} &\text{min/max} && c_1 x_1 + c_2 x_2 + \dots + c_n x_n \\ &\text{subject to} && a_{11}x_1 + a_{12}x_2 + \dots + a_{1n} x_n \le b_1 \\ & && \dots \\ & && a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \le b_m \\ &\text{and} && x_1 \ge 0, x_2 \ge 0, \dots, x_n \ge 0. \end{aligned}$$

- Compact vector form:

$$\begin{aligned} &\text{min/max} && \mathbf{c}^T\mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} \le \mathbf{b} \text{ and } \mathbf{x} \ge \mathbf{0}, \end{aligned}$$



where $\mathbf{c}^T$ is an n-dimensional row vector, $\mathbf{x}$ is an n-dimensional column vector, $\mathbf{A}$ is an $m \times n$ matrix, and $\mathbf{b}$ is an m-dimensional column vector. $\mathbf{x} \ge \mathbf{0}$ means that each component of $\mathbf{x}$ is nonnegative.

Difference Constraints

- **System of difference constraints:** Each row of the linear-programming matrix **A** contains exactly one 1, one -1, and 0's for all other entries.
  - Form: $x_j - x_i \le b_k$, $1 \le i, j \le n$, $1 \le k \le m$.
- **Example:**

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \le \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}$$

- Equivalent to finding the unknowns x_i, i = 1, 2, …, 5 s.t. the following difference constraints are satisfied:

$$\begin{aligned} x_1 - x_2 &\le 0 \\ x_1 - x_5 &\le -1 \\ x_2 - x_5 &\le 1 \\ x_3 - x_1 &\le 5 \\ x_4 - x_1 &\le 4 \\ x_4 - x_3 &\le -1 \\ x_5 - x_3 &\le -3 \\ x_5 - x_4 &\le -3 \end{aligned}$$

Constraint Graph

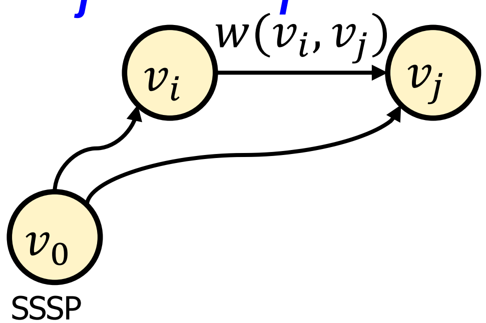
- $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ is a solution to $\mathbf{Ax} \le \mathbf{b}$ of difference constraints $\Rightarrow$ so is $\mathbf{x} + d = (x_1 + d, x_2 + d, \ldots, x_n + d)$.
- **Constraint graph:** Weighted, directed graph $G=(V, E)$.
  - $V = \{ v_0, v_1, \ldots, v_n\}$
  - $E = \{(v_i, v_j): x_j - x_i \le b_k\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$
  - $w(v_i, v_j) = b_k$ if $x_j - x_i \le b_k$ is a difference constraint.
- If G contains no negative-weight cycle, then $\mathbf{x} = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$ is a feasible solution; no feasible solution, otherwise.
- **Example:** $\mathbf{x} = (-5, -3, 0, -1, -4)$ and $\mathbf{x}' = (d-5, d-3, d, d-1, d-4)$ are feasible solutions.

$x_j - x_i \le b_k \leftrightarrow x_j \le x_i + b_k$

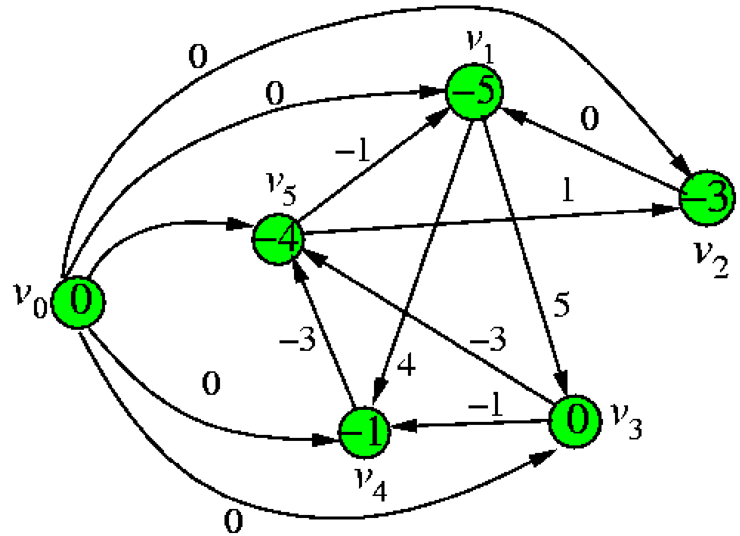
Triangle inequality:

$\boldsymbol{\delta(v_0, v_j) \le \delta(v_0, v_i) + w(v_i, v_j)}$

$\boldsymbol{x_j}$ $\boldsymbol{x_i}$ $\boldsymbol{b_k}$



$$
\begin{aligned}
x_1 - x_2 &\le 0 \\
x_1 - x_5 &\le -1 \\
x_2 - x_5 &\le 1 \\
x_3 - x_1 &\le 5 \\
x_4 - x_1 &\le 4 \\
x_4 - x_3 &\le -1 \\
x_5 - x_3 &\le -3 \\
x_5 - x_4 &\le -3
\end{aligned}
$$



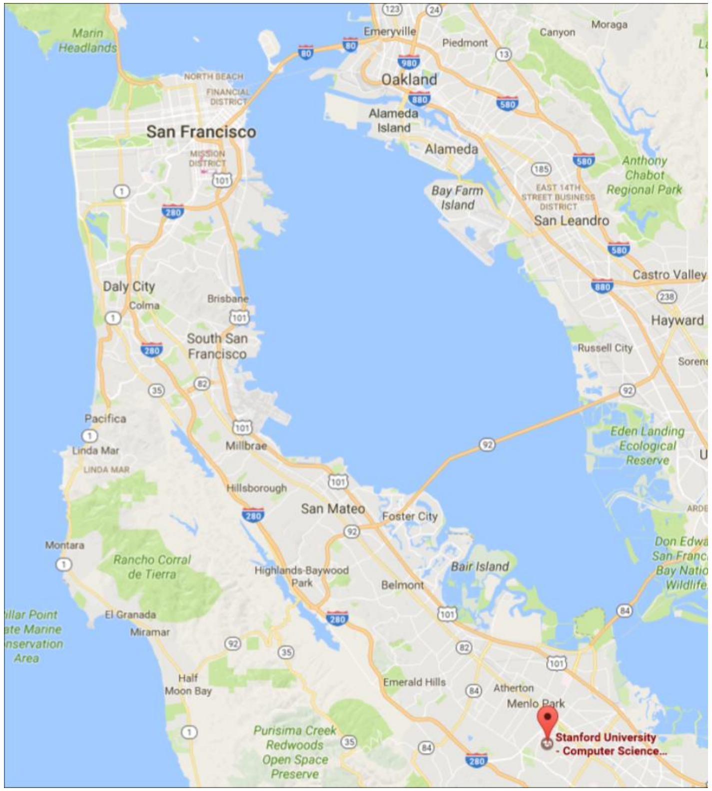
A* Search

Single-source single-destination shortest paths
Redundancy removal
AI applications

Stanford CS 106X, Lecture 23: Dijkstra and A* Search
P. E. Hart, N. J. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100-107, July 1968, doi: 10.1109/TSSC.1968.300136.

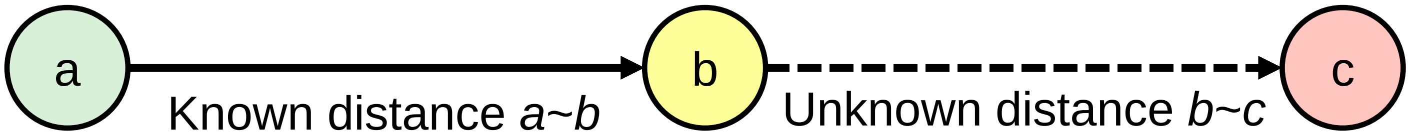
Improving on Dijkstra’s

- If we want to travel from Stanford to San Francisco, Dijkstra's algorithm will look at path distances around Stanford.
- We know that we generally need to go Northwest from Stanford.
- This is more information! Let's not only prioritize by weights, but also give some priority to the direction we want to go.
  - E.g., we will add more information based on a heuristic, which could be direction in the case of a street map



Dijkstra Observations (1/2)

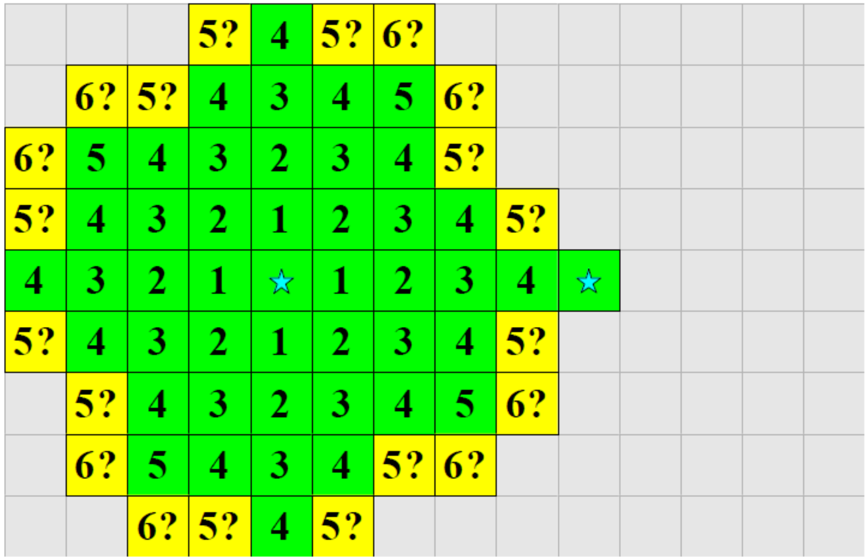
- Dijkstra's algorithm uses a priority queue and examines possible paths in increasing order of their known cost or distance.
  - The idea is that paths with a lower distance-so-far are more likely to lead to paths with a lower total distance at the end.



  - But what about the remaining distance? What if we knew that a path that was promising so far will be unlikely to lead to a good result?
  - Can we modify the algorithm to take advantage of this information?

Dijkstra Observations (2/2)

- Dijkstra's algorithm works by incrementally computing the shortest path to intermediary nodes in the graph in case they prove to be useful.
  - Some of these paths are in the "wrong" direction.

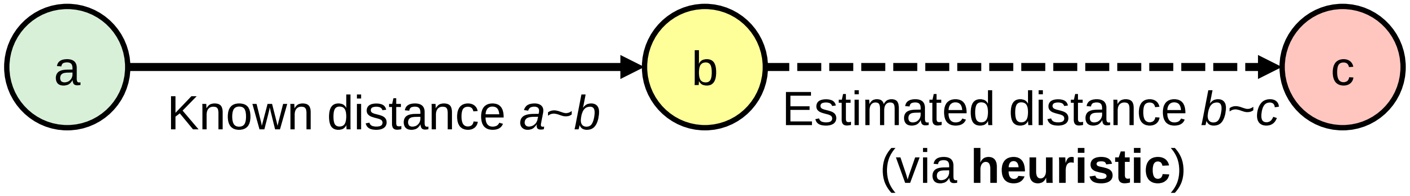


- The algorithm has no "big-picture" conception of how to get to the destination; the algorithm explores outward in all directions.
  - Could we give the algorithm a hint? Explore in a smarter order?
  - What if we knew more about the vertices or graph being searched?

Heuristics

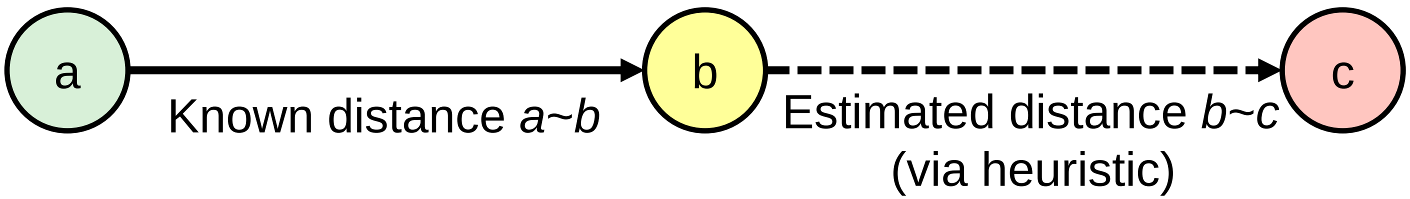
- **Heuristic**: A speculation, estimation, or educated guess that guides the search for a solution to a problem
  - In the context of graph searches: A function that approximates the distance from a known vertex to another destination vertex
  - Example: Estimate the distance between two places on a Google Maps graph to be the direct straight-line distance between them

- **Admissible heuristic**: One that never overestimates the distance (optimal!)
  - Okay if the heuristic underestimates sometimes
  - Only ignore paths that in the *best case* are worse than your current path



The A* Algorithm

- **A*** ("A star"): A modified version of Dijkstra's algorithm that uses a heuristic function to guide its order of path exploration
- Suppose we are looking for paths from start vertex *a* to *c*



  - Any intermediate vertex *b* has two costs:
  - The known (exact) cost from the start vertex *a* to *b*
  - The heuristic (estimated) cost from *b* to the end vertex *c*
- **Idea:** Run Dijkstra's algorithm, but use this priority in the priority queue: (best first search)
  - priority(*b*) = cost(*a*, *b*) **+ heuristic(*b*, *c*)**
  - Choose to explore paths with lower estimated cost

Example: Maze Heuristic

- Idea: Use "Manhattan distance" between the points
  - $H(p_1, p_2) = abs(p_1.x - p_2.x) + abs(p_1.y - p_2.y)$ // $\Delta x + \Delta y$
  - The idea: Dequeue/explore neighbors with lower (cost+Heuristic)

8	7	6	5	4	5
7	6	5	4	3	4
6	5	4	3	2	3
5	4	3	2	1	2
4	a	2	1	c	1
5	4	3	2	1	2
6	5	4	3	2	3

Dijkstra

8	7	6	5	4	5	6	7	8	9?				
7	6	5	4	3	4	5	6	7	8	9?			
6	5	4	3	2	3	4	5	6	7	8	9?		
5	4	3	2	1	2	3		7	8	9?			
4	3	2	1	☆	1	2		8	☆				
5	4	3	2	1	2	3		7	8	9?			
6	5	4	3	2	3	4	5	6	7	8	9?		
7	6	5	4	3	4	5	6	7	8	9?			
8	7	6	5	4	5	6	7	8	9?				

A*

Best First Search and Redundancy Removal