Algorithms and Data Structures (CSci 115)

California State University Fresno
College of Science and Mathematics
Department of Computer Science
H. Cecotti

Learning outcomes

- Shortest path algorithms
  - Using graph data structures

Introduction

- **Problem**
  - Find the sequence of vertices and edges in a graph G=(V,E) from
    - Vertex “start” to vertex “end”: single-source shortest-paths problem
- **Rationale**
  - To **not** test all the possible paths, many paths are pointless, large detours
- Remark
  - Breadth First Search (BFS):
    - It is a shortest path algorithm for unweighted graphs (weight=1 for all edges)
    - It can be used for the project IF the terrain is simple ☺
      - Wall + No-Wall : you pass or you don’t

Definitions

- Weighted Directed Graph
  - G=(V,E)
  - Weight function w : E→ R (set of real numbers)
    - Mapping of edges to real valued weights
    - w(p) of path p=<v_0,v_1,…v_k> = sum of all the weights
    - Definition:

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

  - We want to find a shortest path from a given ***source*** vertex
  - Shortest-path weight δ(u,v) from u to v

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

  - A shortest path from u to v is any path p with weight w(p)= δ(u,v)

Variants

- Single-**destination** shortest-paths problem:
  - Find a shortest path **to a given destination** vertex t from **each** vertex v.
  - By reversing the direction of each edge in the graph, we can reduce this problem to a **single-source** problem.
- Single-**pair** shortest-path problem:
  - Find a shortest path from u to v **for given vertices u and v**.
  - If we solve the single-source problem with source vertex u, this problem is also solved.
  - All known algorithms for this problem have the same worst-case asymptotic running time as the best single-source algo
- **All-pairs** shortest-paths problem:
  - Find a shortest path from u to v **for every pair of vertices u and v.**
  - Although we can solve this problem by running a single source algorithm once from each vertex, we usually can solve it faster.

Property

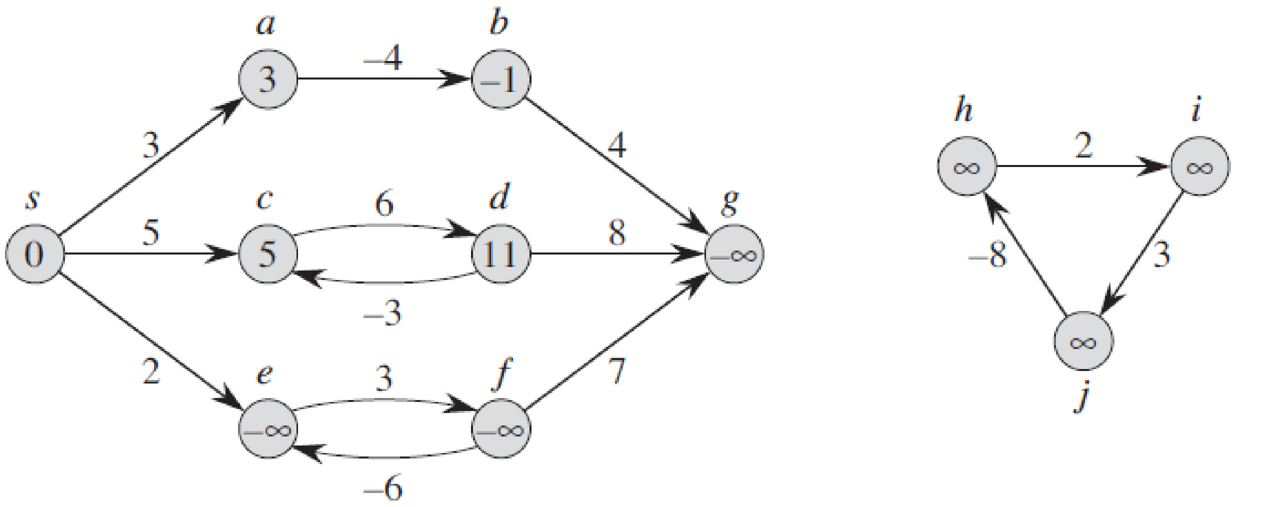
- A shortest path between 2 vertices contains other shortest paths **within** it
  - Notion of subproblen → Dynamic programming or greedy algo could work ☺
- Lemma: Subpaths of shortest paths **are** shortest paths
  - Directed Weighted graph $G=(V,E)$ with weight function $w: E \rightarrow R$
  - Let $p=\langle v_0,v_1,\ldots,v_k\rangle$ be a shortest path from v_0 to v_k,
  - For any $(i,j) \mid 0 \le i \le j \le k$, let $p_{ij}=\langle v_i,v_{i+1},\ldots v_j\rangle$ be the subpath of p from v_i to v_j
  - Then, p_{ij} is a shortest path from v_i to v_j

Property

- **Proof**
  - ➢Decompose p into p_{0i}, p_{ij}, p_{jk}
  - ➢Then $w(p)=w(p_{0i})+w(p_{ij})+w(p_{jk})$
  - ➢We assume ∃ a path p'_{ij} from v_i to v_j with weight $w(p'_{ij})<w(p_{ij})$
  - ➢Then p_{0i}, p'_{ij}, p_{jk} is a path from v_0 to v_k and $w(p_{0i})+w(p'_{ij})+w(p_{jk})<w(p)$
  - ➢→ **Contradiction** ! We assumed p is a shortest path from v_0 to v_k

Special cases

- Negative weight edges



- Negative edge weights in a directed graph.
- The shortest-path weight from source s is within each vertex.
- As vertices e and f form a **negative-weight cycle** reachable from s, they have shortest-path weights of -∞.
- As vertex g is reachable from a vertex whose shortest-path weight is - ∞, it, too, has a shortest-path weight of - ∞.
- Vertices like h, i, and j are not reachable from s, and so their shortest-path weights are - ∞
  - even though they lie on a negative-weight cycle.

Special cases

- It **cannot** contain a negative-weight cycle.
- It **cannot** contain a positive-weight cycle
  - Because removing the cycle from the path produces a path with the same source and destination vertices and a lower path weight.
- →
  - **If** there is a shortest path
    - from a source vertex s
    - to a destination vertex that contains a 0-weight cycle
  - **Then** there is another shortest path from s to without this cycle
- → the shortest path will not have a cycle (otherwise it s not the shortest path)
- Remark
  - As long as a shortest path has 0-weight cycles, we can repeatedly remove these cycles from the path **until** we have a shortest path that is cycle-free.

Representation of shortest paths

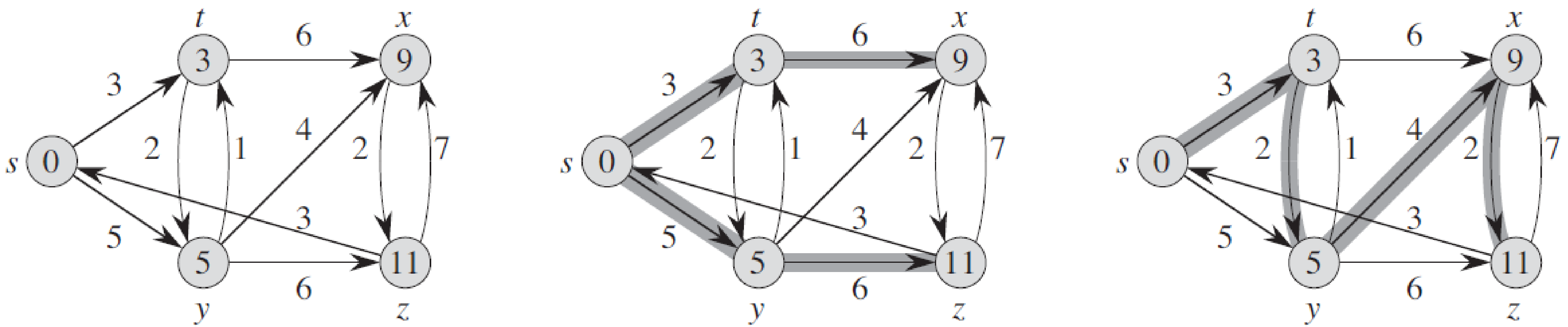
- Let G=(V,E)
  - For each vertex v∈V a ***predecessor*** v.π (another vertex or NIL)
- The shortest-paths algorithms set the π attributes
  - so that the chain of **predecessors**
    - originating at a vertex v runs **backwards** along a shortest path from s to v.
- Same as Breadth-Fist Search (BFS)
  - **predecessor subgraph** $G_\pi=(V_\pi,E_\pi)$induced by the values.
    - V_π : the set of vertices of Gπ with non-NIL predecessors + the source s
    - $V_\pi=\{ v \in V : v.\pi != NIL\} \cup \{s\}$
    - E_π : the set of edges induced by the values for vertices
    - $E_\pi=\{(v.\pi,v) \in E : v \in V_\pi - \{s\}\}$
- A shortest-paths **tree**
  - Same as the breadth-first tree **but** it contains shortest paths
    - from the source defined in terms of **edge weights** instead of **numbers of edges**.

Representation of shortest paths

- Let G=(V,E)
  - ➢a weighted, directed graph with weight function w: E → R
  - ➢Assumption: G contains no negative-weight cycles reachable from the source vertex s∈V (for well defined shortest paths)
- → A **shortest-paths tree** rooted at s is:
  - ➢A directed subgraph G=(V',E') with V' ⊆ V and E' ⊆ E |
    - o V' is the set of vertices reachable from S in G
    - o G' forms a rooted tree with root s
    - o For all v ∈V' the unique simple path from s to v in G' is a shortest path from s to v in G

Example

- **Example**
  - A weighted, directed graph with shortest-path weights from source s.
  - The shaded edges form a shortest-paths tree rooted at the source s.
  - Another shortest-paths tree with the **same** root.



Relaxation

- For each vertex $v \epsilon V$
  - We maintain an attribute v.*d :* an upper bound on the weight of a shortest path from source s to v.
  - **v.d = shortest-path estimate**
- Principle
  - The process of **relaxing** an edge (u,v) consists of testing whether
    - we can improve the shortest path to v found so far by going through u and, if so, updating v.d and v.π

Relaxation

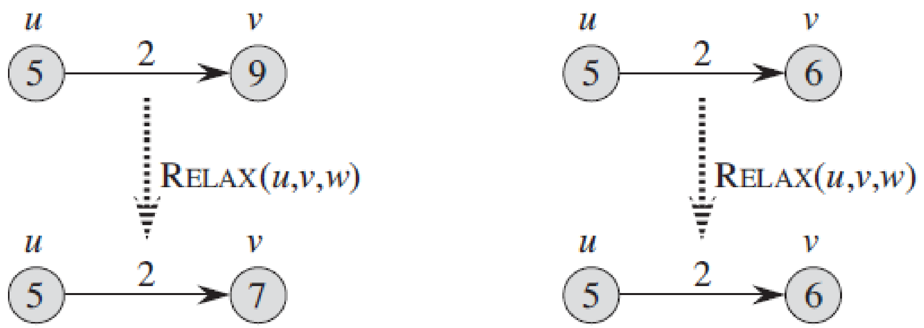
- Algorithms

```
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
```

Just a few lines, but very important functions to be used in the algorithms

Example



Properties

- **Triangle inequality**
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- **Upper-bound property**
  - We always have $v.d \geq \delta(s,v)$ for all vertices $v \in V$ and
  - once $v.d$ achieves the value $\delta(s,v)$ it never changes.
- **No-path property**
  - If there is no path from s to v, then we always have $v.d = \delta(s,v) = \infty$
- **Convergence property**
  - If s ->u →v is a shortest path in G for some $u,v \in V$, and if $u.d=\delta(s,u)$ at any time prior to relaxing edge (u,v)
  - then $v.d= \delta(s,u)$ at all times afterward.
- **Path-relaxation property**
  - If $p=\langle v_0,v_1,\ldots,v_k\rangle$ is a shortest path from $s=v_0$ to v_k, and we relax the edges of p in the order (v_0,v_1) (v_1,v_2) … (v_{k-1},v_k)
  - then $v_k.d=\delta(s,v_k)$
  - This property holds **regardless** of any other relaxation steps that occur, even if they are intermixed with relaxations of the edges of p.
- **Predecessor-subgraph property**
  - Once $v.d=\delta(s,v)$ for all $v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Algorithms

- Shortest path methods
  - Bellman Ford algorithm
  - Dijsktra algorithm (better running time than Bellman Ford algorithm)

Bellman Ford algorithm

- Features
  - Detect whether a negative-weight cycle is reachable from the source
  - Returns a Boolean value (true/false):
    - whether or not there is a negative-weight cycle that is reachable from the source
- Principle
  - It relaxes edges, *progressively decreasing* an estimate v.d on the weight of a shortest path from the source s to each vertex v until it achieves the shortest path weight from s
- Complexity
  - Run time O(VE)

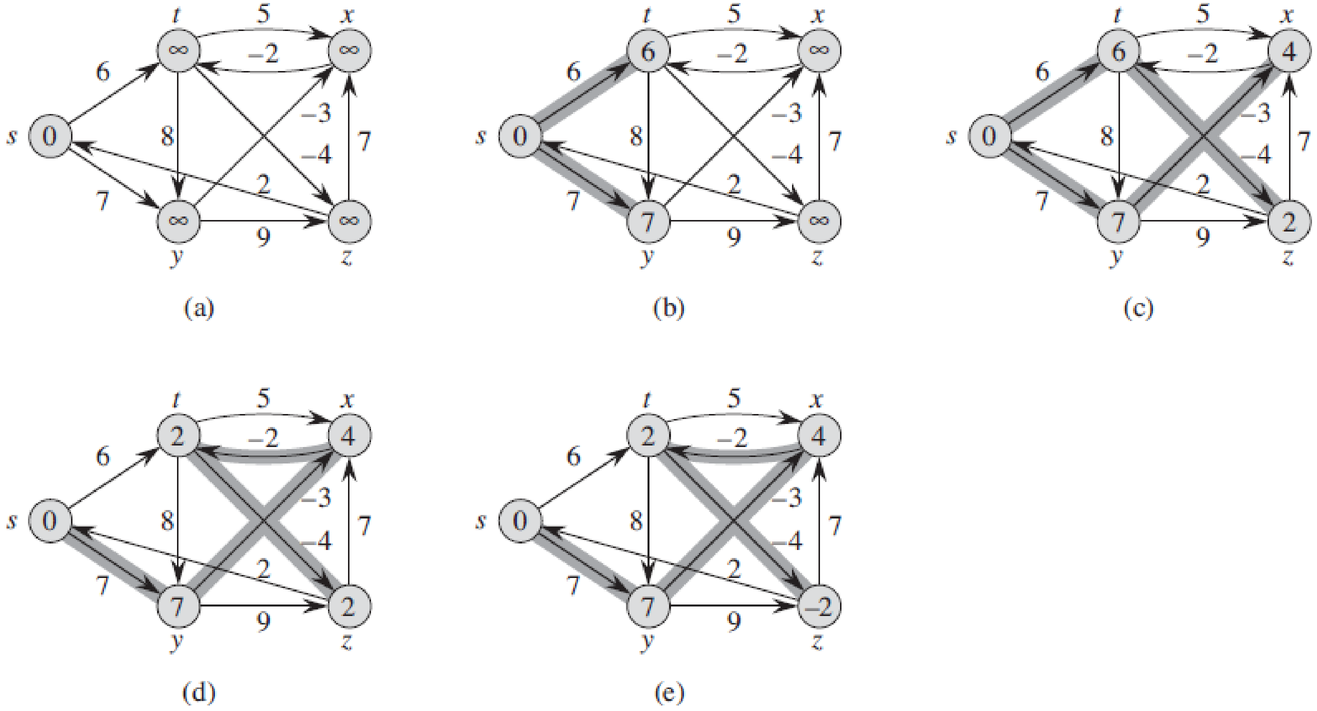
Bellman Ford algorithm

- Pseudo-code

```
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
```

Example



Order of the relaxation:
(t,x) (t,y) (t,z) (x,t) (y,x) (y,z) (z,x) (z,s) (s,t) (s,y)

Dijkstra's algorithm

- Greedy algorithm
- Edge weights in the input graph are non-negative,
  - As in the road-map example
  - $w(u,v) \geq 0$ for each edge $(u,v) \in E$
- **Principle**
  - Repeatedly select the vertex $u \in V-S$ with the minimum shortest-path estimate
  - Adds u to S
  - Relaxes all edges leaving u.
- Data structure needed:
  - min-priority queue of vertices (Q)
    - Vertex added/removed from Q: only 1 time

Dijkstra's algorithm

- Pseudo code

```
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)
```

(invariant: Q=V-S)

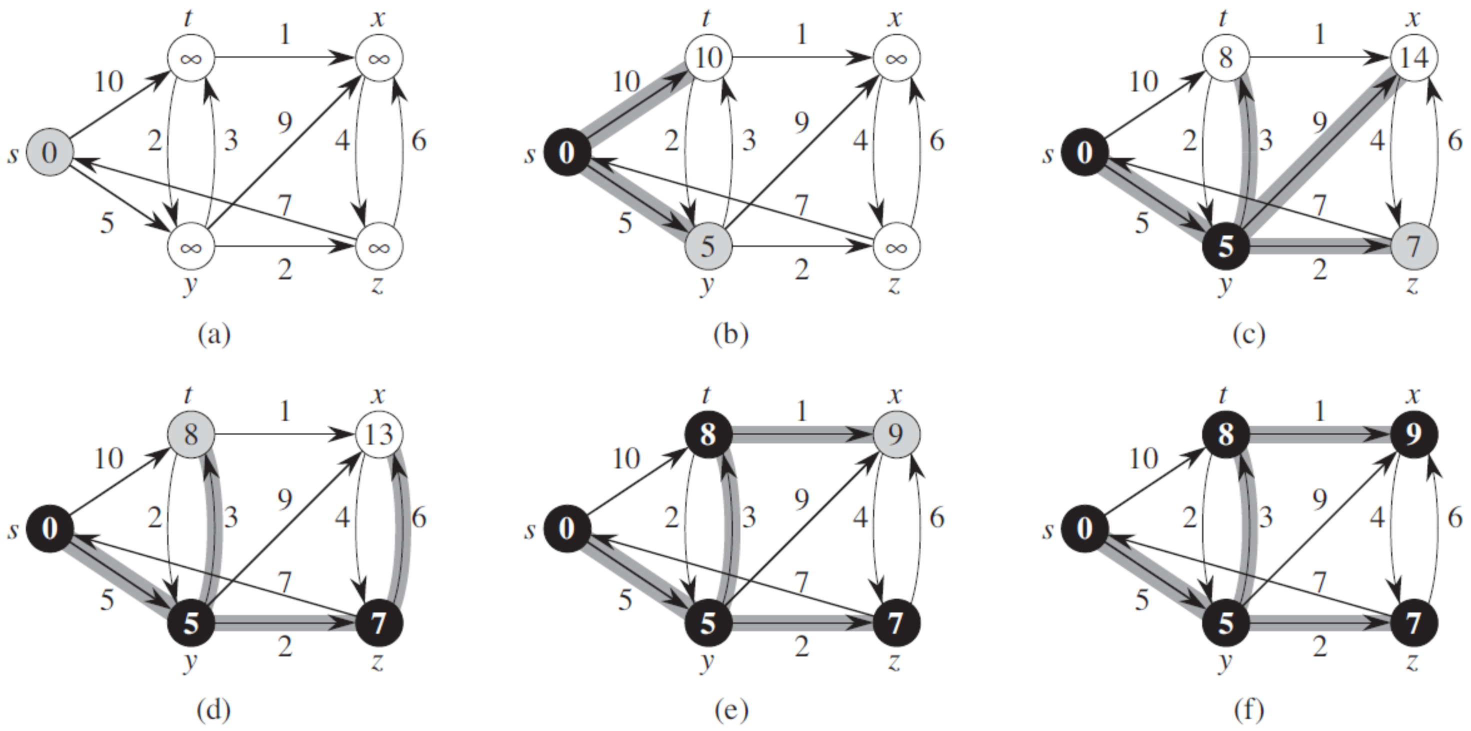
u has the smallest shortest-path estimate of any vertex in V-S.

```
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

- Priority queue
  - ➢Need of Insert, Extract-Min, Decrease-Key

Dijkstra's algorithm

- Example
  - Shortest-path estimates appear within the vertices
  - Shaded edges indicate predecessor values



Dijkstra's algorithm

- **Why** is it a greedy algorithm?
  - **Because** it always chooses the "lightest" or "closest" vertex in V-S to add to set S → greedy strategy
  - Dijkstra's algorithm **does** compute shortest paths.
- To show that it is correct
  - $u.d = \delta(s,u)$ for all vertices $u \in V$

Dijkstra's algorithm

- **Proof**
  - At the start of each iteration of the **while** loop of lines 4–8, $v.d = \delta(s,v)$ for each vertex $v \in S$.
  - It is enough to show for each vertex $u \in V$, we have $u.d = \delta(s,u)$ at the time when u is added to set S.
    - we rely on the upper-bound property to show that the equality holds at all times thereafter !
  - Initialization: $S = \emptyset$ (empty set) (invariant is true)

Dijkstra’s algorithm

- **Proof** (Maintenance part – part 1)
  - Show that in each iteration, u.d= δ(s,u) for the vertex added to set S.
    - For the purpose of **contradiction**,
      - Let u be the first vertex for which **u.d!=δ(s,u) when it is added to set S**.
    - Focus on the situation at the beginning of the iteration of the **while** loop in which u is added to S and derive the contradiction that u.d= δ(s,u) at that time by examining a shortest path from s to u.
    - We must have (u!=s) because s is the 1st vertex added to set S and s.d= δ(s,s)=0 at that time.
    - As (u!=s), we have also that (S!=ø) ; just before u is added to S. ☺

Dijkstra's algorithm

- **Proof** (Maintenance part – part 2)
  - ➢ There must be some path from s to u, for otherwise u.d=δ(s,u)=inf
    - o by the no-path property that would violate our assumption that (u.d!=δ(s,u)).
    - o Because there is **at least** one path, there is a shortest path p from s to u.
  - ➢ Before adding u to S, path p connects
    - o a vertex s ϵ S → a vertex u ϵ V-S
  - ➢ Let us consider the first vertex y along p such that yϵV-S and let xϵS be y's predecessor along p.
  - ➢ → we can decompose path p into p1 and p2 connected by the edge (x,y)
    - o p1: path **from** s **to** x
    - o p2: path **from** y **to** u

Dijkstra's algorithm

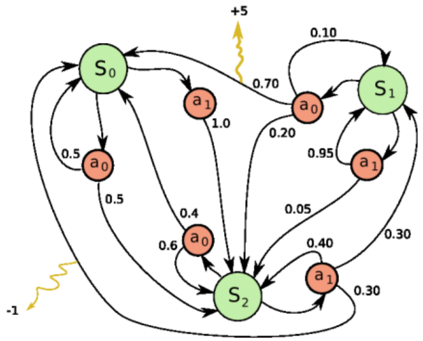
- **Proof** (Maintenance part – part 3)
  - We claim that (y.d=δ(s,y)) when u is added to S.
  - To prove this claim
    - Observe that x∈S.
    - Then, because we pick u as the 1st vertex for which (u.d!=δ(s,u)) when it is added to S,
    - we had (x.d!=δ(s,x)) when x was added to S
  - Edge (x,y) was relaxed at that time, and the claim follows from the convergence property.
    - We can now obtain a **contradiction** to prove that u.d= δ(s,u).
    - Because y comes **before** u on a shortest path from s to u and all edge weights are non-negative, we have
      - δ(s,y)≤ δ(s,u) → y.d≤u.d (through upper bound property)
  - **But** because u and y are in V-S, we have u.d≤y.d !
  - So we have δ(s,y)=δ(s,u)=y.d=u.d → δ(s,u)=u.d → **contradicts the choice of u**
  - → δ(s,u)=u.d when u is added to S.

Dijkstra's algorithm

- **Proof**
  - Termination
    - Q=ø along with our earlier invariant that Q=V-S
    - → it implies that S=V
    - → δ(s,u)=u.d for all vertices u∈V .

About the weights

- Applications
  - Vertices = states
  - Edges = probability to go from one state to another after a particular action
    - From a set of actions
  - The weights in a graph can be used to model probabilities
    - Vertex v with a set of input n vertices
    - → sum of all the vertices = 1
  - Probability after a particular action to go to different vertex
  - To explore further:
    - Markovian Decisional Processes (MDP)



Conclusion

- Shortest path algorithms
  - For robotics, navigation
    - Path planning for cars, robots, …
  - For video games
    - Example: Project #2
- You should know how to
  - Define and describe formally a graph
    - and its properties
  - Implement Bellman-Ford
  - Implement Dijkstra algorithm with appropriate data structures
    - (data structures from previous labs)

Questions ?

- Reading
  - ➢Csci 115 book – Section 9.3
  - ➢Introduction to Algorithms, Chapter 24, 25.