Announcements

- HW1 grading is out
  - Regrading due by next Monday
  - Please submit requests to **Gradescope**

- Likely, Midterm 1 will be postponed to Oct 15'th
  - Time will be finalized during next lecture, but likely it's the 15'th

CS6161: Design and Analysis of Algorithms (Fall 2020)

Shortest Path Problems

Instructor: Haifeng Xu

Outline

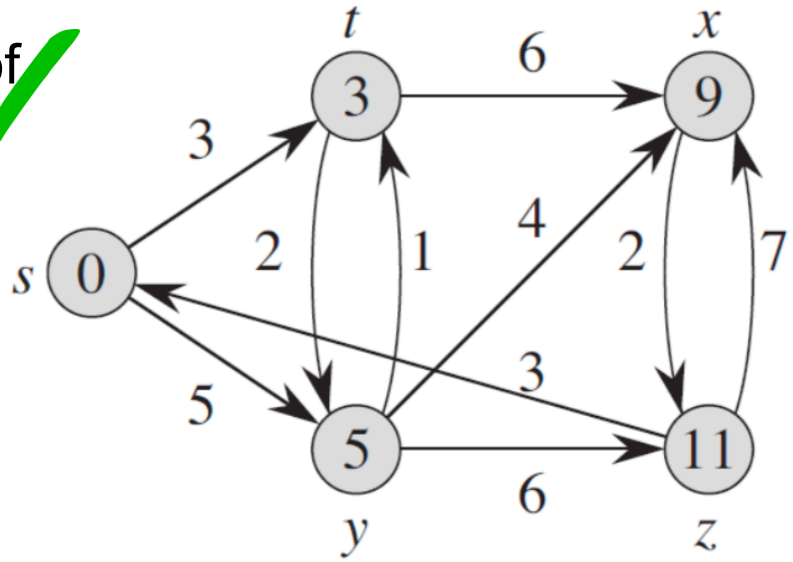
- Shortest Path Problems, and Properties
- Dijkstra's Algorithm

Shortest Paths Problems

- Typically for weighted graph, directed or undirected
  - Weights capture traversal cost or distance
- Try to find shortest way to go from one node to another

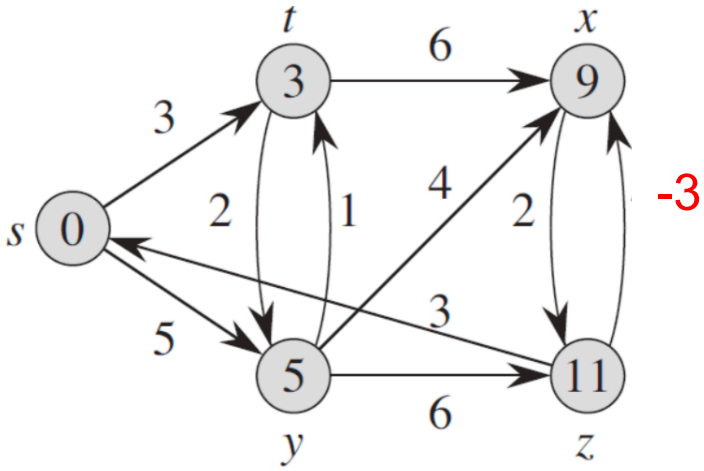
Q: How to represent a path?

- A sequence of edges; Or
- A sequence of nodes, and each pair of consecutive nodes have an edge ✓



Many Variants

- Single pair
- Single source, all destinations
- All pairs



Important Remarks

- One important condition is whether edges have negative costs
- Will focus on directed graphs from now on
  - Undirected graphs reduce to directed ones, by splitting each edge into two directed edges

Applications

Really a lot

- Traffic routing
- Network analysis
- Financial trading (e.g., can you find an arbitrage opportunity through a cycle trading?)
- Robotics, circuit design, …
- Metrics embedding
- A building block for many other algorithms (e.g., the DP for TSP)

Triangle Inequality of Shortest Distance

- Let $dist(s, v) =$ length of the shortest path from s to v

Lemma: for any s, v, t, we have $dist(s, v) + dist(v, t) \geq dist(s, t)$.

- Proof: the shortest path from s to v and the path from v to t form a path from s to t, and thus has cost at least $dist(s, t)$.

Corollary: for any $e = (v, t)$, we have $dist(s, v) + c(e) \geq dist(s, t)$.

Because $dist(v, t) \leq c(e)$

Triangle Inequality of Shortest Distance

- Let $dist(s, v) =$ length of the shortest path from s to v

Lemma: for any s, v, t, we have $dist(s, v) + dist(v, t) \geq dist(s, t)$.

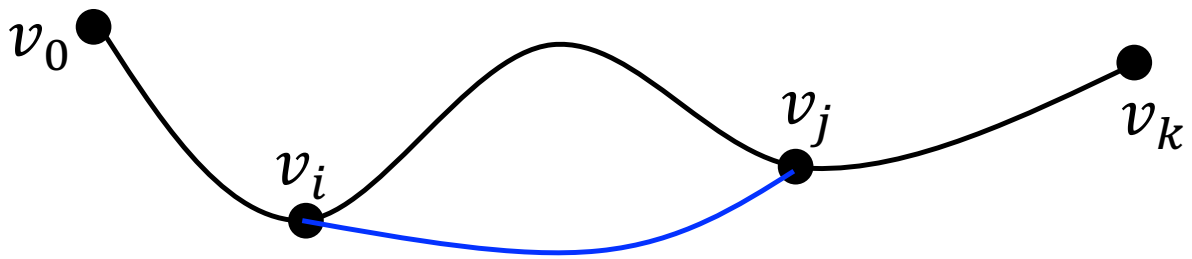
- Remark: triangle inequality is a natural property of distance functions, and is very useful in algorithm design

A Useful Lemma

Lem: If $P = (v_0, v_1, \cdots, v_k)$ is a shortest path from v_0 to v_k, then for any $0 \leq i < j \leq k$, $(v_i, \cdots, v_j)$ is a shortest path from v_i to v_j.

Proof: by contradiction

- If not, let $(v_i, u_1, \cdots, u_l, v_j)$ is a strictly shorter path from v_i to v_j
- Then substituting $(v_i, \cdots, v_j)$ in P by $(v_i, u_1, \cdots, u_l, v_j)$ will lead to a strictly shorter path from v_0 to v_k, contradicting the definition of P



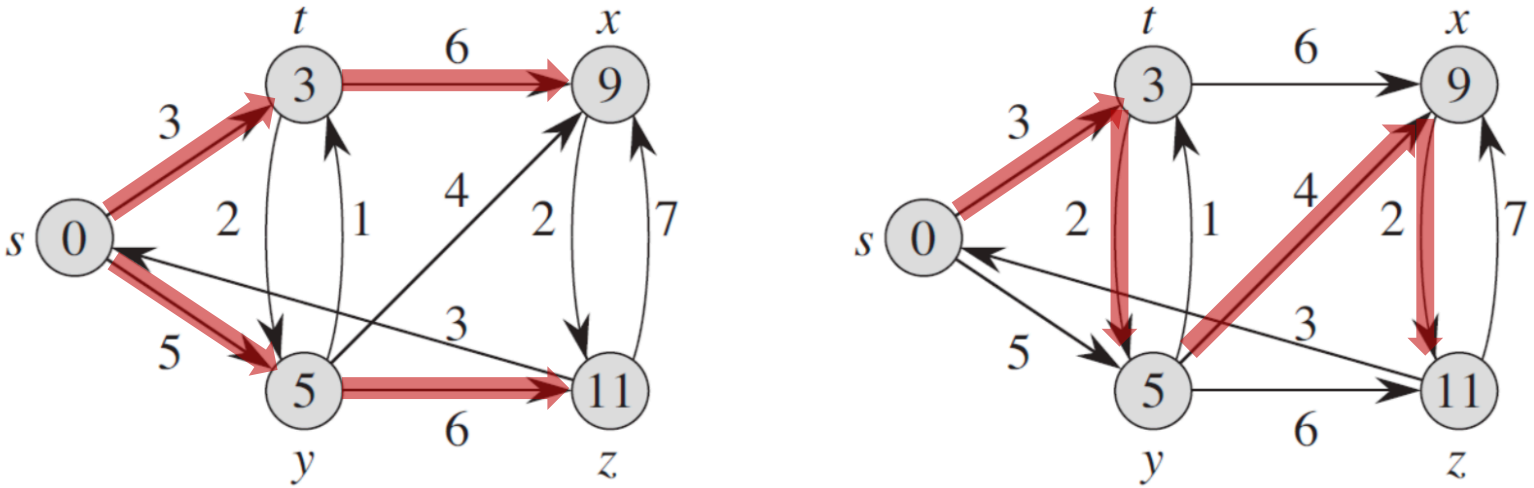
An Efficient Way to Store Shortest Paths

Shortest-Paths Trees

A shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$:

- V' = set of vertices reachable from s in G
- G' forms a rooted tree with root s
- For all $v \in V'$, the (unique) path from s to v in G' is a shortest path from s to v in G.

Generally, not unique



An Efficient Way to Store Shortest Paths

Shortest-Paths Trees

A shortest-paths tree rooted at s is a directed subgraph $G' = (V', E')$:

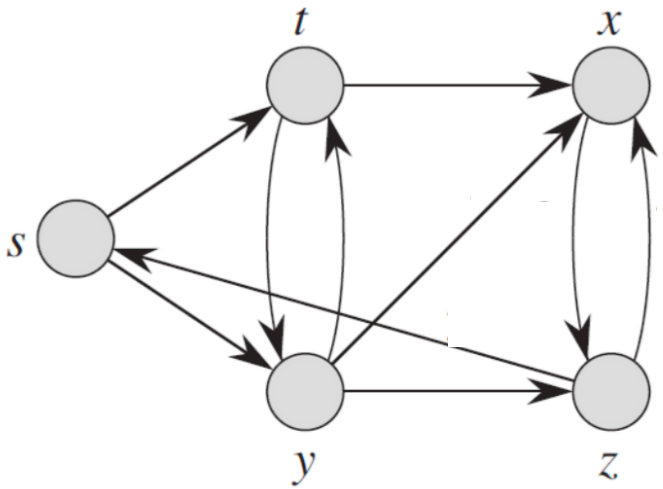
- V' = set of vertices reachable from s in G
- G' forms a rooted tree with root s
- For all $v \in V'$, the (unique) path from s to v in G' is a shortest path from s to v in G.

Remarks:

- This definition relies on previous lemma – in a shortest path P from s to v_k, any point v_i on P is visited through a shortest path as well
- This tree stores a shortest path from s to any reachable vertex

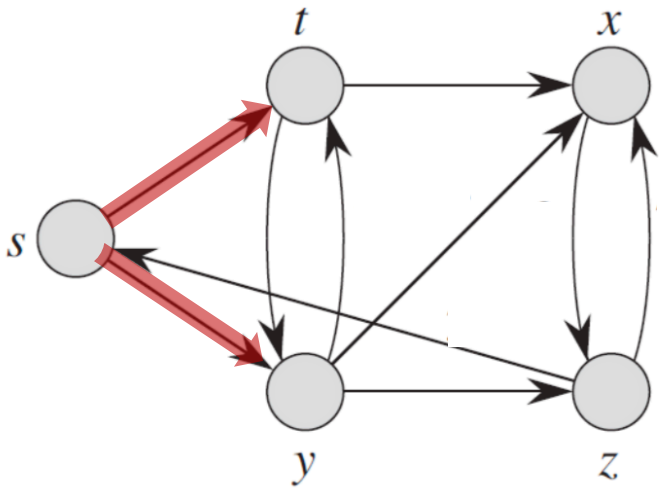
Actually, You Have Seen Shortest Path Algorithms Before!

- Recall BFS



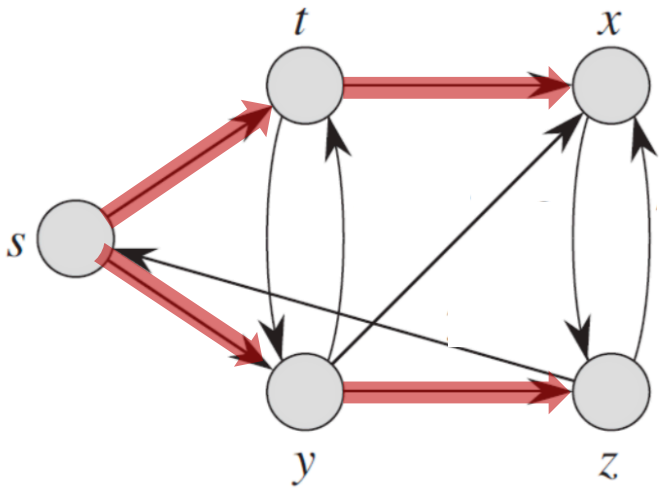
Actually, You Have Seen Shortest Path Algorithms Before!

- Recall BFS



Actually, You Have Seen Shortest Path Algorithms Before!

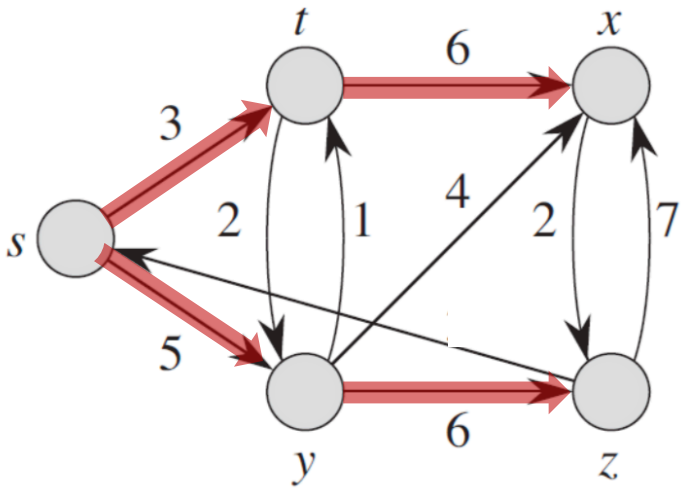
- Recall BFS



- The path reaching any node v is the "shallowest"
- If edge weights are all 1, it is the shortest path and the BFS tree is the shortest-paths tree!

Actually, You Have Seen Shortest Path Algorithms Before!

- Recall BFS



What to do if edges have different weights?

Note: assume positive weights for now (negative weights are trickier)

Outline

- Shortest Path Problems, and Properties
- Dijkstra's Algorithm

Ideas

- Why does BFS work?
- BFS(s) explores nodes in increasing distance from s
- Let $dist(s, v)$ denote the shortest path length from s to v

```
ShortestPathAlgoIdea(s, G)
1    Initialize dist(s, v) = ∞ for each v ∈ V;
2    Initialize S = ∅;
3
4                                                    ;
5
6
```

Ideas

- Why does BFS work?
- BFS(s) explores nodes in increasing distance from s
- Let $dist(s, v)$ denote the shortest path length from s to v

```
ShortestPathAlgoIdea(s, G)
1    Initialize dist(s, v) = ∞ for each v ∈ V;
2    Initialize S = ∅;
3    while S ≠ V do
4        find vertex v ∈ V − S that is the closest to s;
5        update dist(s, v);
6        S = S ∪ {v};
```

Ideas

- Why does BFS work?
- BFS(s) explores nodes in increasing distance from s
- Let $dist(s, v)$ denote the shortest path length from s to v

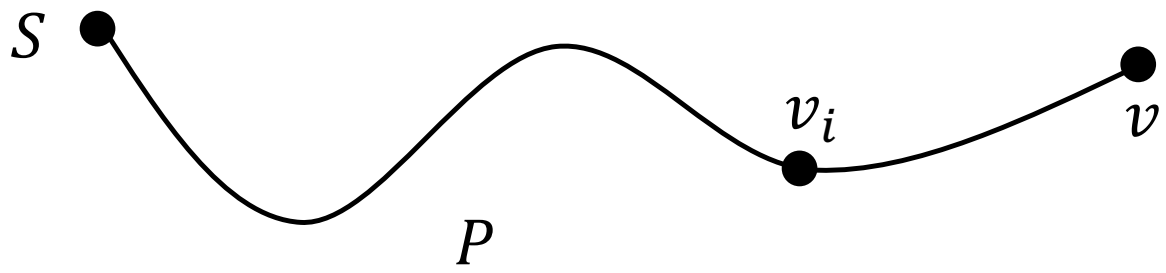
```
ShortestPathAlgoIdea(s, G)
1    Initialize dist(s, v) = ∞ for each v ∈ V;
2    Initialize S = ∅;
3    while S ≠ V do
4        find vertex v ∈ V − S that is the closest to s;
5        update dist(s, v);
6        S = S ∪ {v};
```

How to find the next closest vertex?

Finding the ith Closest Vertex

- At ith iteration, S already stores the $i-1$ closest vertices to s
- What do we know about the ith closest vertex?

Claim: Let P be any shortest path from s to v where v is the ith closest, then all intermediate vertices in P belongs to S.



$dist(s, v_i) < dist(s, v)$ due to positive edge weights

Finding the ith Closest Vertex

- At ith iteration, S already stores the $i-1$ closest vertices to s
- What do we know about the ith closest vertex?

Claim: Let P be any shortest path from s to v where v is the ith closest, then all intermediate vertices in P belongs to S.

Therefore, the next closest vertex must be adjacent to S

Well, but adjacent to which node in S?

Finding the ith Closest Vertex

- Let S contain the $(i-1)$ closest vertices to s
- For each $u \in V - S$, define $\pi(u) = \min_{a \in S}(dist(s,a) + c(a,u))$
  - $c(a,u) = \infty$ if (a,u) is not an edge

Lem: If u is the ith closest vertex to s, then:
(1) $\pi(u) = dist(s,u)$;
(2) $\pi(v) \geq \pi(u)$, for all $v \in V - S$
(3) If $\pi(v) = \pi(u)$ for some $v \in V - S$, then v is also the ith closest

Remark: this lemma tells us which node to be added at ith iteration

- Maintain array $\pi(u)$
- Add the $u \in V - S$ with minimum $\pi(u)$

Proof of the Lemma

- Let S contain the $(i-1)$ closest vertices to s
- For each $u \in V - S$, define $\pi(u) = \min_{a \in S}(dist(s,a) + c(a,u))$

Lem: If u is the ith closest vertex to s, then:
(1) $\pi(u) = dist(s,u)$;
(2) $\pi(v) \geq \pi(u)$, for all $v \in V - S$
(3) If $\pi(v) = \pi(u)$ for some $v \in V - S$, then v is also the ith closest

Proof: for (1)

- By previous lemma, u is adjacent to some $a \in S$
- So $dist(s,u) = dist(s,a) + c(a,u)$ for the a achieving the minimum possible value
- So $dist(s,u) = \pi(u)$

Proof of the Lemma

- Let S contain the $(i-1)$ closest vertices to s
- For each $u \in V - S$, define $\pi(u) = \min_{a \in S}(dist(s,a) + c(a,u))$

Lem: If u is the ith closest vertex to s, then:
(1) $\pi(u) = dist(s,u)$;
(2) $\pi(v) \geq \pi(u)$, for all $v \in V - S$
(3) If $\pi(v) = \pi(u)$ for some $v \in V - S$, then v is also the ith closest

Proof: for (2)

- $\pi(v) \geq dist(s,v)$ as $s \to a \to v$ is a path to v
- So $\pi(v) \geq dist(s,v) \geq dist(s,u)$ for any $v \in V - S$

Proof of the Lemma

- Let S contain the $(i-1)$ closest vertices to s
- For each $u \in V - S$, define $\pi(u) = \min_{a \in S}(dist(s,a) + c(a,u))$

Lem: If u is the ith closest vertex to s, then:
(1) $\pi(u) = dist(s,u)$;
(2) $\pi(v) \geq \pi(u)$, for all $v \in V - S$
(3) If $\pi(v) = \pi(u)$ for some $v \in V - S$, then v is also the ith closest

Proof: for (3)

- If $\pi(v) = \pi(u)$, we must have $\pi(v) = dist(s,v)$ as well
  - Otherwise, $\pi(v) > dist(s,v)$, and u cannot be the ith closest
- Now, by definition, v is also the ith closest

Algorithm

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3

4

5

6

7

8

Algorithm

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3 **while** $S \neq V$ **do**

4 find vertex $u \in V - S$ with minimum $\pi(u)$;

5

6

7

8

Algorithm

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3 **while** $S \neq V$ **do**

4 find vertex $u \in V - S$ with minimum $\pi(u)$;

5 $\text{dist}(\text{s}, \text{u}) = \pi(u)$;

6 $S = S \cup \{u\}$;

7

8

Algorithm

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3 **while** $S \neq V$ **do**

4 find vertex $u \in V - S$ with minimum $\pi(u)$;

5 $\mathrm{dist}(\mathrm{s}, \mathrm{u}) = \pi(u)$;

6 $S = S \cup \{u\}$;

7 foreach $v \in V - S$ do

8 $\pi(v) = \min_{a \in S}(dist(s, a) + c(a, v))$

Algorithm

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3 **while** $S \neq V$ **do**

4 find vertex $u \in V - S$ with minimum $\pi(u)$;

5 $\mathrm{dist}(\mathrm{s}, \mathrm{u}) = \pi(u)$;

6 $S = S \cup \{u\}$;

7 foreach $v \in V - S$ do

8 $\pi(v) = \min_{a \in S}(dist(s, a) + c(a, v))$

- Correctness: by induction using previous lemmas

Algorithm: Running Time

ShortestPathAlgo(s, G)

1 Initialize $\pi(v) = \infty$ for each $v \in V$;

2 Initialize $S = \emptyset, \pi(s) = 0$;

3 while $S \neq V$ **do** → repeat n times

4 find vertex $u \in V - S$ with minimum $\pi(u)$;

5 $\text{dist}(s, u) = \pi(u)$;

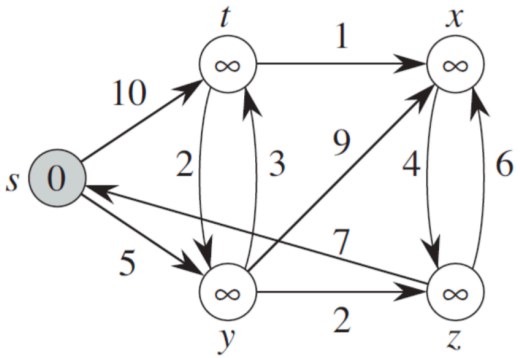
6 $S = S \cup \{u\}$;

7 foreach $v \in V - S$ do → $O(m)$ time

8 $\pi(v) = \min_{a \in S}(dist(s, a) + c(a, v))$

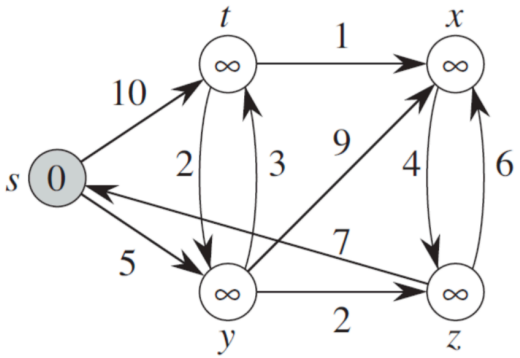
- Total: $O(nm)$ time.

Example Execution

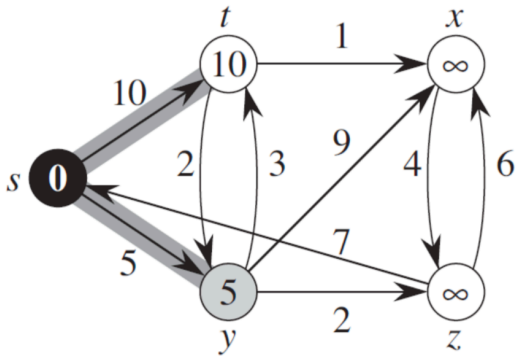


(a)

Example Execution

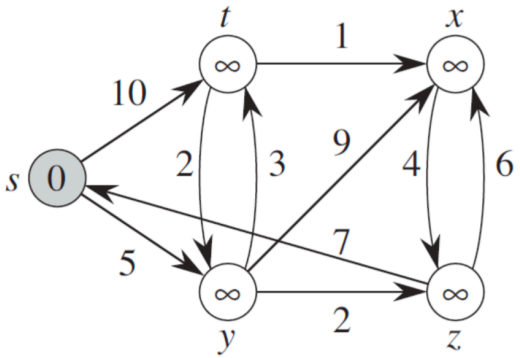


(a)

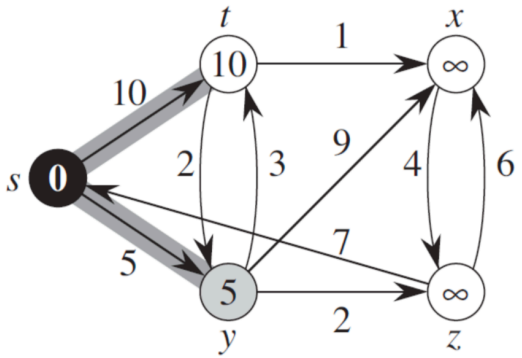


(b)

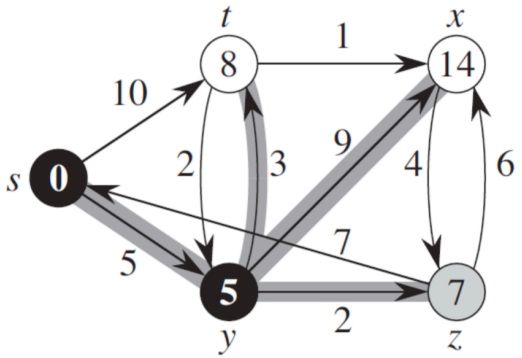
Example Execution



(a)

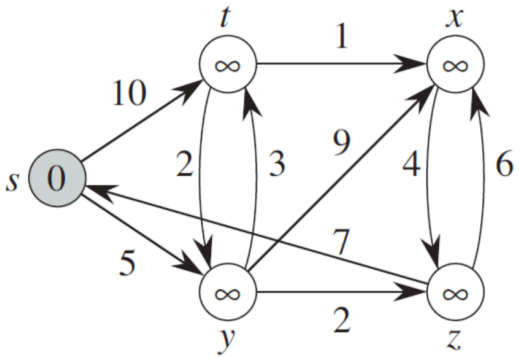


(b)

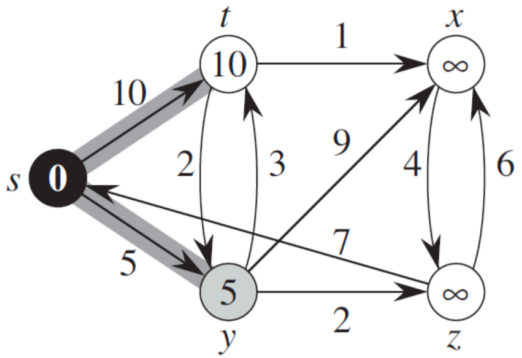


(c)

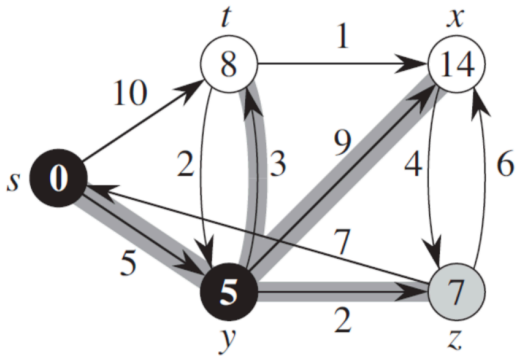
Example Execution



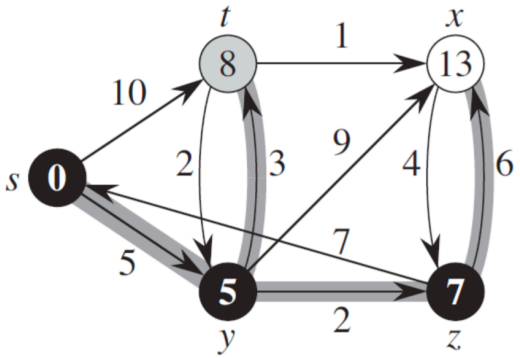
(a)



(b)

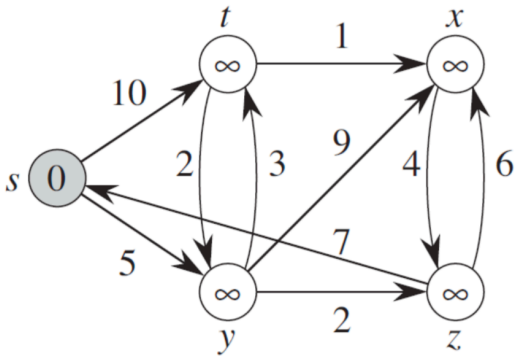


(c)

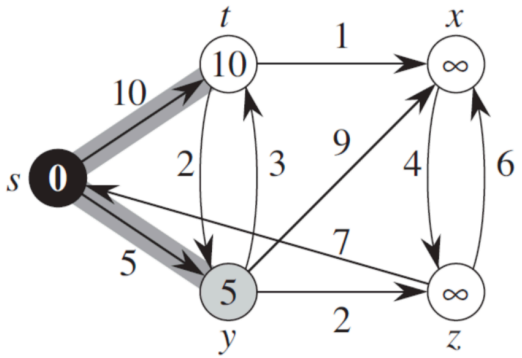


(d)

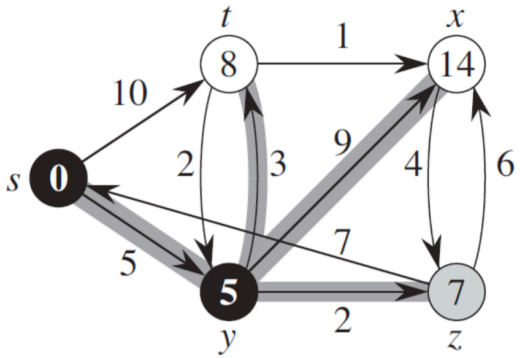
Example Execution



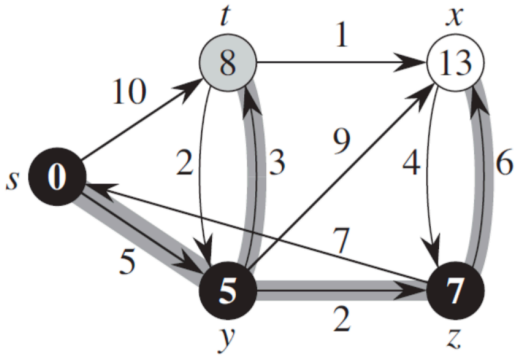
(a)



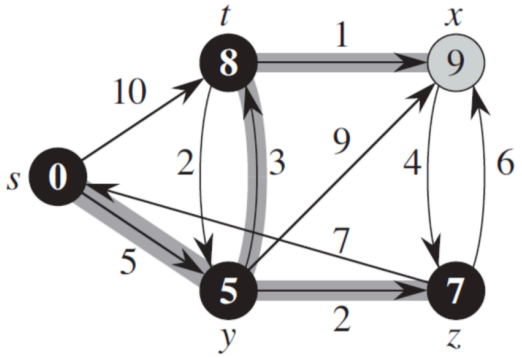
(b)



(c)

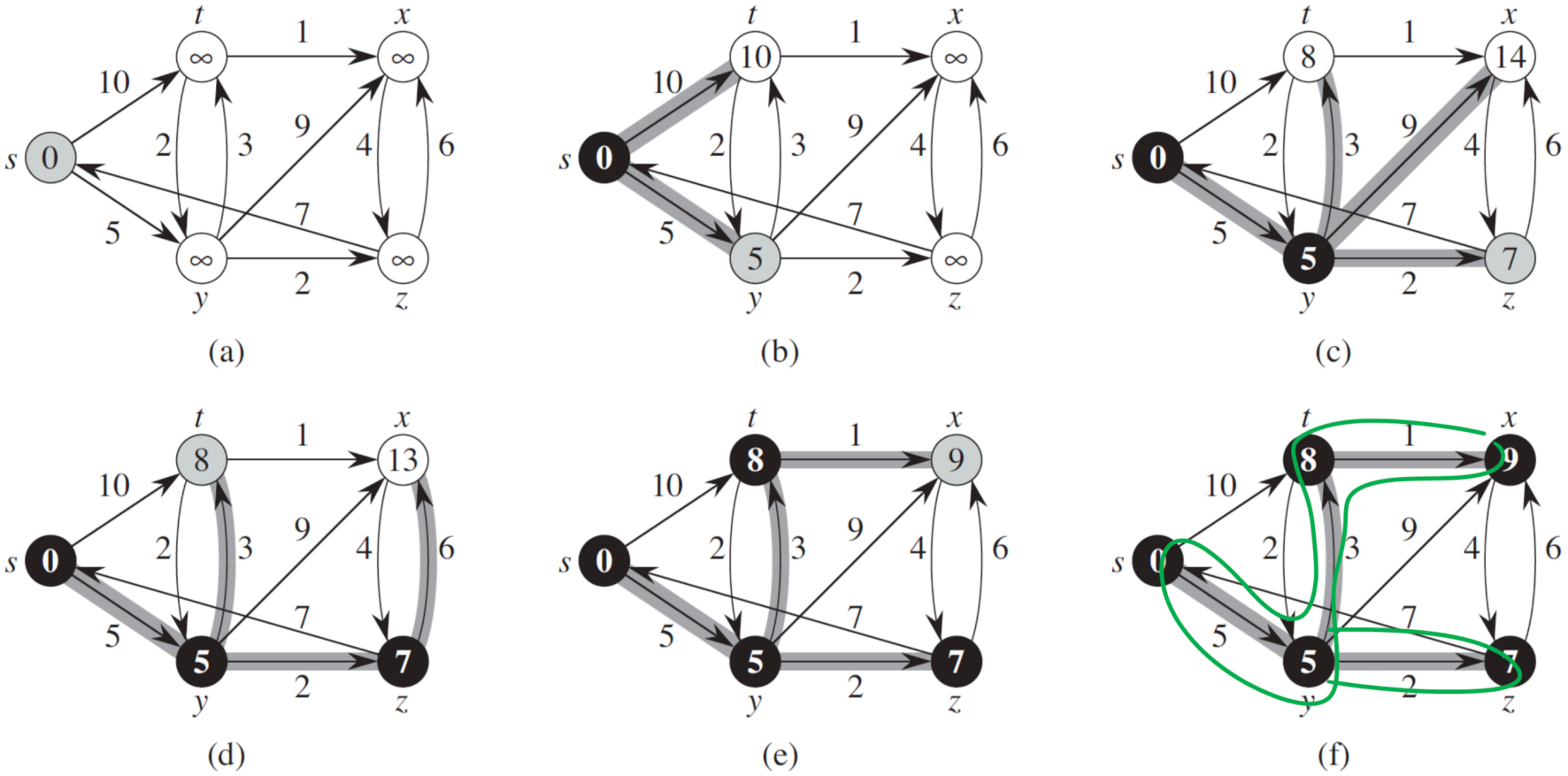


(d)



(e)

Example Execution



More Efficient Implementation

Critical Optimization: For unexplored vertex $v \in V - S$, explicitly maintain $\pi(v)$ instead of computing directly from formula:

$$\pi(v) = \min_{u \in S}(dist(s,u) + c(u,v)).$$

Main Idea:

- For each $v \notin S, \pi(v)$ can only decrease, because S only increase
- Suppose u is added to S and there is an edge (u,v) leaving u. Then, it suffices to update

$$\pi(v) = \min\{\pi(v), dist(s,u) + c(u,v)\}$$

Dijkstra's Algorithm

- Eliminate $\pi(v)$ and let $dist(s,v)$ maintain it
- Update $dist$ values after adding u by scanning only edges leaving u

```
ShortestPathAlgo(s, G)
1    Initialize dist(s, v) = ∞ for each v ∈ V;
2    Initialize S = ∅, dist(s, s) = 0;
3    while S ≠ V do
4        find vertex u ∈ V − S with minimum d(s, u);
5        S = S ∪ {u};
6        foreach v ∈ Adj(u) do
7            dist(s, v) = min{dist(s, v), dist(s, u) + c(u, v)}
```

All nodes adjacent to u

- Step (6) (7) are fast now, but step (4) becomes a bottleneck
- How to update values and retrieve min values efficiently?

Dijkstra's Algorithm

- Eliminate $\pi(v)$ and let $dist(s,v)$ maintain it
- Update $dist$ values after adding u by scanning only edges leaving u

ShortestPathAlgo(s,G)

1 Initialize $dist(s,v) = \infty$ for each $v \in V$;
2 Initialize $S = \emptyset, dist(s,s) = 0$;
3 **while** $S \neq V$ **do**
4 find vertex $u \in V - S$ with minimum $d(s,u)$;
5 $S = S \cup \{u\}$;
6 **foreach** $v \in Adj(u)$ **do**
7 $dist(s,v) = \min\{dist(s,v), dist(s,u) + c(u,v)\}$

All nodes adjacent to u

- Use priority queue!
  - Step(4) in $O(1)$ time
  - Each update of $dist$ in $O(\log n)$ time

Total: $O(m \log n)$ time

More on Dijkstra and Priority Queues

- Dijkstra's algorithm gives shortest paths from s to all vertices in V.
- There is an even faster implementation in $O(n \log n + m)$ time using Fibonacci Heaps.
  - If $m = \Omega(n \log n)$, running time is linear in input size.
- Data structures are complicated to analyze/implement. Recent work has obtained data structures that are easier to analyze and implement, and perform well in practice. Rank-Pairing Heaps (European Symposium on Algorithms, September 2009)

Thank You

Haifeng Xu

University of Virginia

hx4ad@virginia.edu