Shortest Path Algorithms

Andreas Klappenecker

[based on slides by Prof. Welch]

Single Source Shortest Path

Single Source Shortest Path

- Given:
  - a directed or undirected graph $G = (V,E)$
  - a source node s in V
  - a weight function $w: E \to R$.

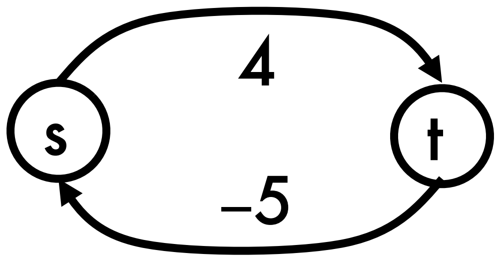
Single Source Shortest Path

- Given:
  - a directed or undirected graph G = (V,E)
  - a source node s in V
  - a weight function w: E -> R.
- Goal: For each vertex t in V, find a path from s to t in G with minimum weight

Single Source Shortest Path

- Given:
  - a directed or undirected graph G = (V,E)
  - a source node s in V
  - a weight function w: E -> R.
- Goal: For each vertex t in V, find a path from s to t in G with minimum weight

Warning! Negative weight cycles are a problem:



Constant Weight Functions

Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?

Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?

Breadth-first search can be used to solve the single-source shortest path problem.

Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?

Breadth-first search can be used to solve the single-source shortest path problem.

Indeed, the tree rooted at s in the BFS forest is the solution.

Intermezzo: Priority Queues

Priority Queues

A min-priority queue is a data structure for maintaining a set S of elements, each with an associated value called key.

This data structure supports the operations:

- insert(S,x) which realizes $S := S \cup \{x\}$
- minimum(S) which returns the element with the smallest key.
- extract-min(S) which removes and returns the element with the smallest key from S.
- decrease-key(S,x,k) which decreases the value of x's

Simple Array Implementation

Suppose that the elements are numbered from 1 to n, and that the keys are stored in an array key[1..n].

- insert and decrease-key take O(1) time.
- extract-min takes O(n) time, as the whole array must be searched for the minimum.

Binary min-heap Implementation

Suppose that we realize the priority queue of a set with n element with a binary min-heap.

- extract-min takes O(log n) time.
- decrease-key takes O(log n) time.
- insert takes O(log n) time.

Building the heap takes O(n) time.

Fibonacci-Heap Implementation

Suppose that we realize the priority queue of a set with n elements with a Fibonacci heap. Then

- extract-min takes O(log n) amortized time.
- decrease-key takes O(1) amortized time.
- insert takes O(1) time.

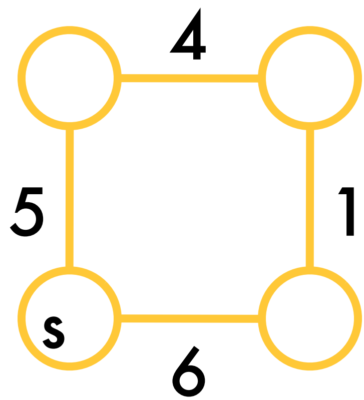
[One can realize priority queues with worst case times as above]

Dijkstra's Single Source Shortest Path Algorithm

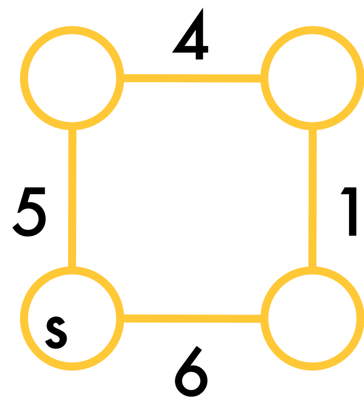
Dijkstra's SSSP Algorithm

- Assumes all edge weights are nonnegative
- Similar to Prim's MST algorithm
- Start with source node s and iteratively construct a tree rooted at s
- Each node keeps track of tree node that provides cheapest path from s (not just cheapest path from any tree node)
- At each iteration, include the node whose cheapest path from s is the overall cheapest

Prim's vs. Dijkstra's

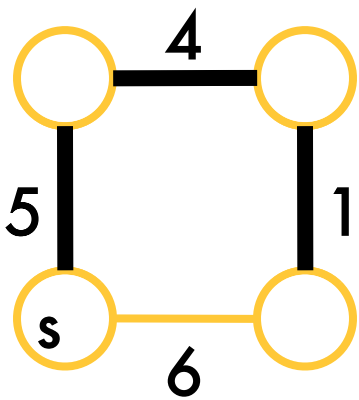


Prim's MST

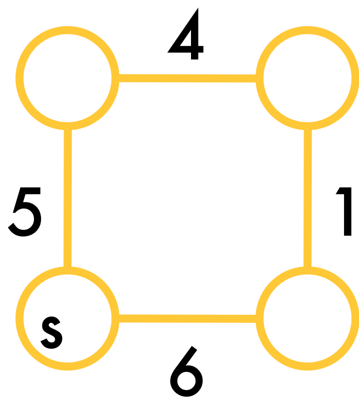


Dijkstra's SSSP

Prim's vs. Dijkstra's

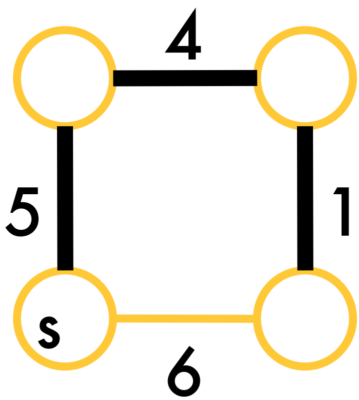


Prim's MST

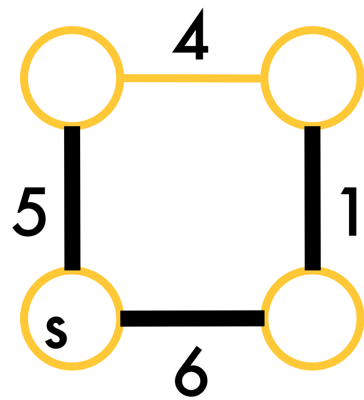


Dijkstra's SSSP

Prim's vs. Dijkstra's



Prim's MST



Dijkstra's SSSP

Implementing Dijkstra's Alg.

- How can each node u keep track of its best path from s?
- Keep an estimate, d[u], of shortest path distance from s to u
- Use d as a key in a priority queue
- When u is added to the tree, check each of u's neighbors v to see if u provides v with a cheaper path from s:
  - compare d[v] to d[u] + w(u,v)

Dijkstra's Algorithm

- input: G = (V,E,w) and source node s

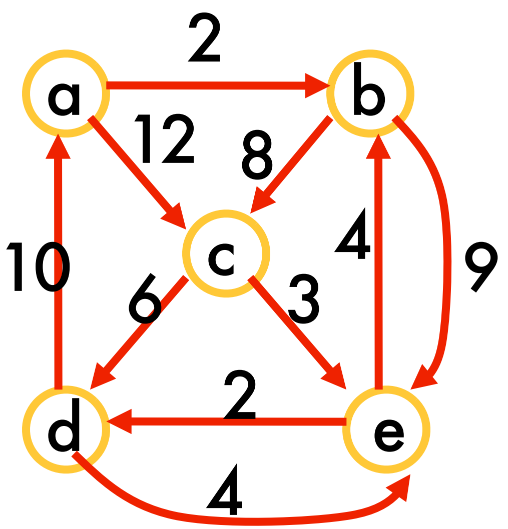
// initialization

- d[s] := 0
- d[v] := infinity for all other nodes v
- initialize priority queue Q to contain all nodes using d values as keys

Dijkstra's Algorithm

- while Q is not empty do
  - u := extract-min(Q)
  - for each neighbor v of u do
    - if d[u] + w(u,v) < d[v] then // relax
      - d[v] := d[u] + w(u,v)
      - decrease-key(Q,v,d[v])
      - parent(v) := u

Dijkstra's Algorithm Example

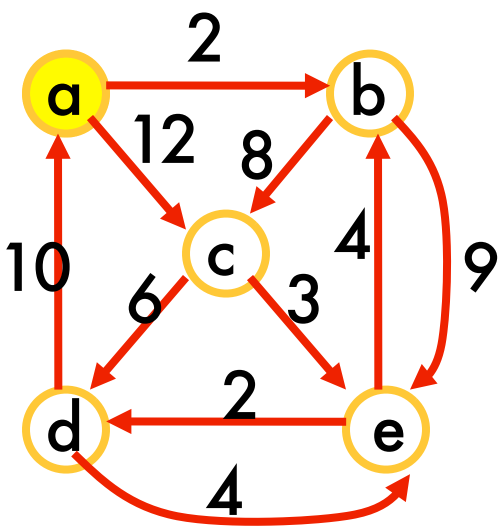


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example

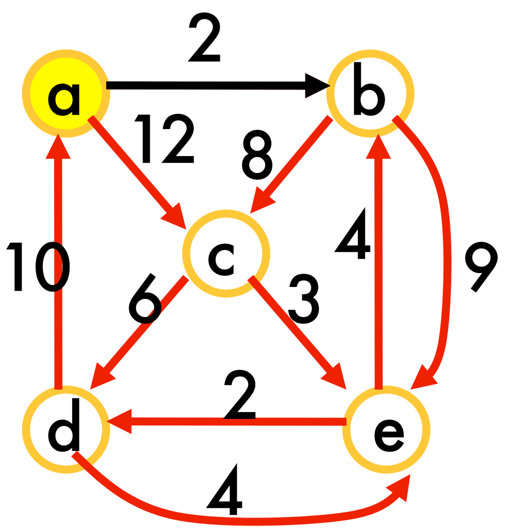


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example

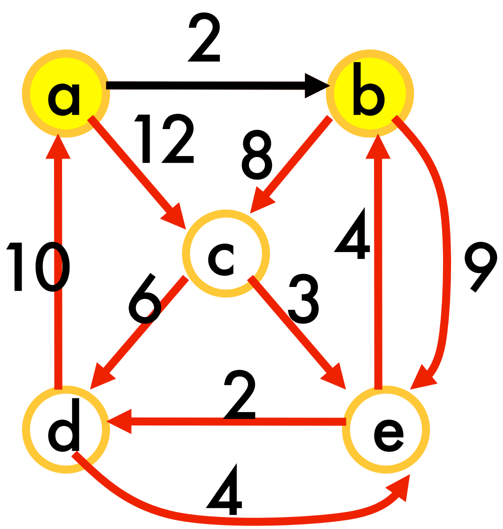


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example

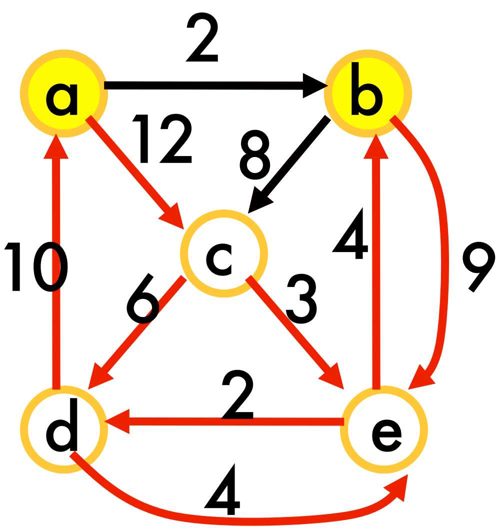


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example

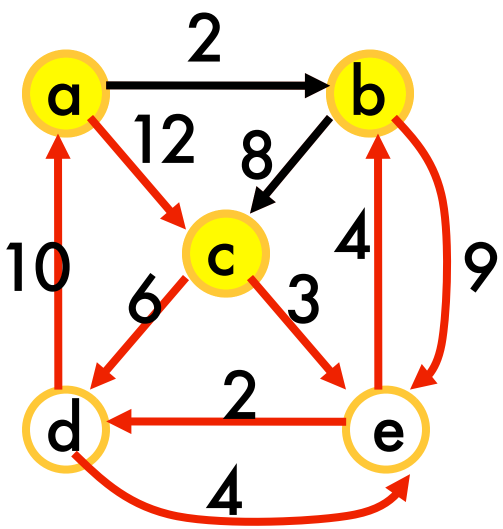


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example



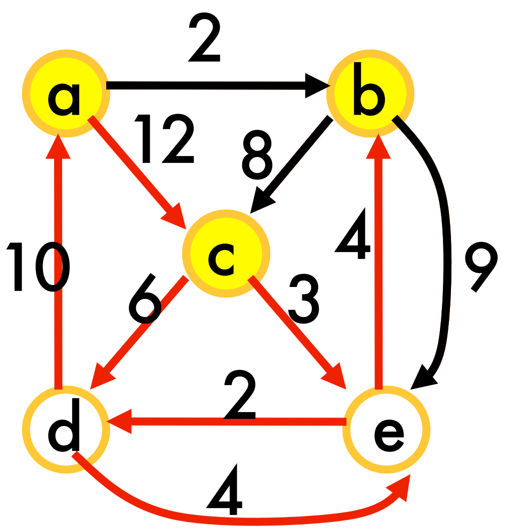
a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Friday, October 12, 2012

Dijkstra's Algorithm Example

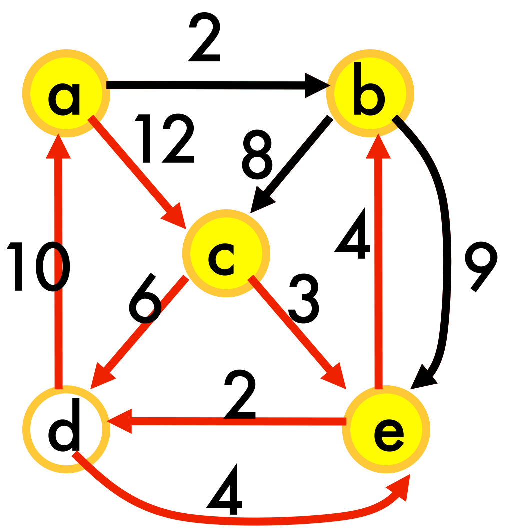


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example

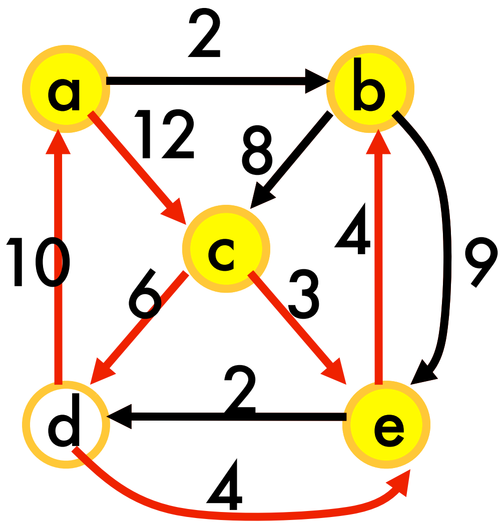


a is source node

iteration

	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

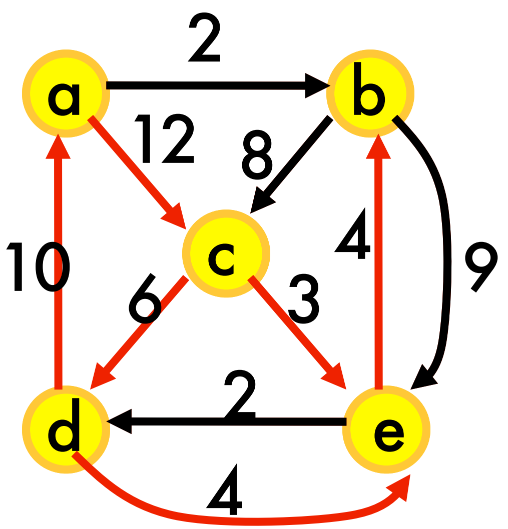
Dijkstra's Algorithm Example



a is source node

	iteration					
	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

Dijkstra's Algorithm Example



a is source node

	iteration					
	0	1	2	3	4	5
Q	abcde	bcde	cde	de	d	Ø
d[a]	0	0	0	0	0	0
d[b]	∞	2	2	2	2	2
d[c]	∞	12	10	10	10	10
d[d]	∞	∞	∞	16	13	13
d[e]	∞	∞	11	11	11	11

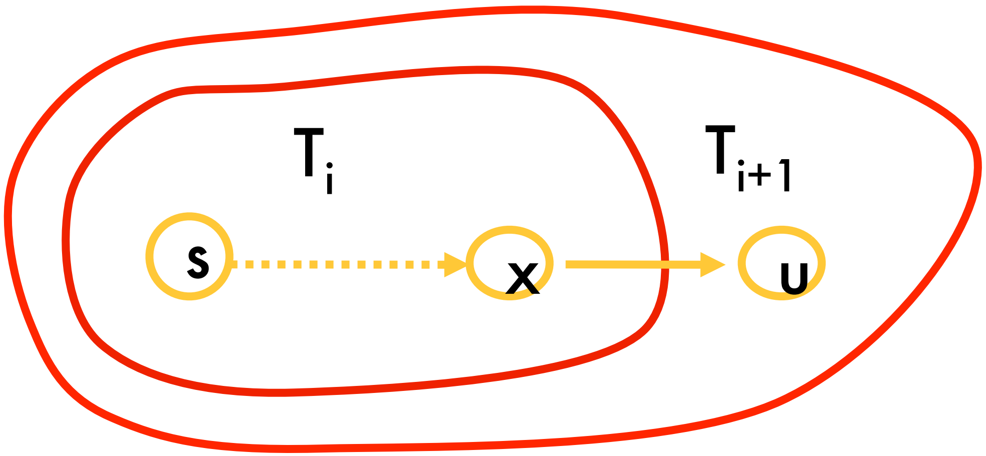
Correctness of Dijkstra's Alg.

- Let T_i be the tree constructed after i-th iteration of the while loop:
  - The nodes in T_i are not in Q
  - The edges in T_i are indicated by parent variables
- Show by induction on i that the path in T_i from s to u is a shortest path and has distance d[u], for all u in T_i.
- Basis: i = 1.

  s is the only node in T_1 and d[s] = 0.

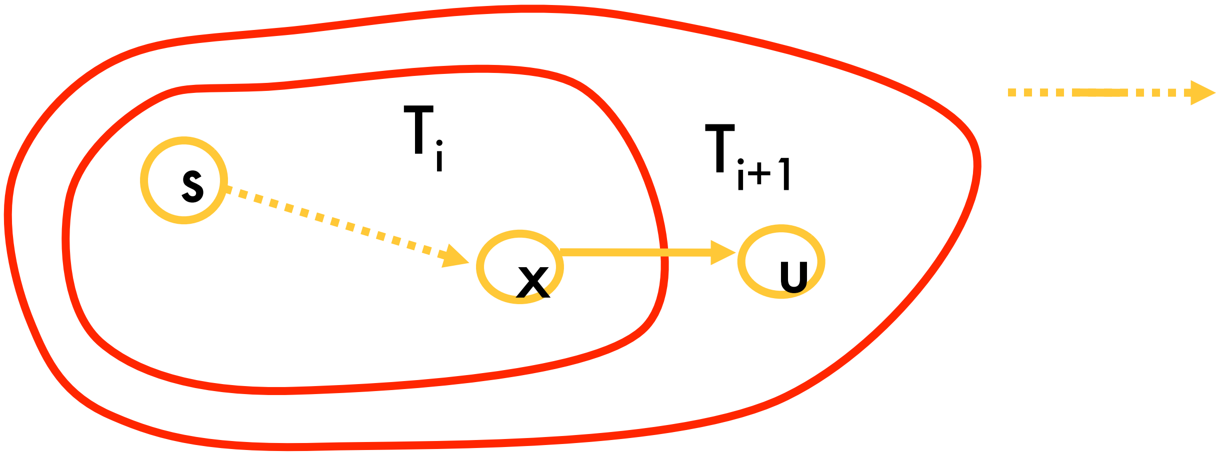
Correctness of Dijkstra's Alg.

- Induction: Assume T_i is a correct shortest path tree. We need to show that T_{i+1} is a correct shortest path tree as well.
- Let u be the node added in iteration i.
- Let x = parent(u).



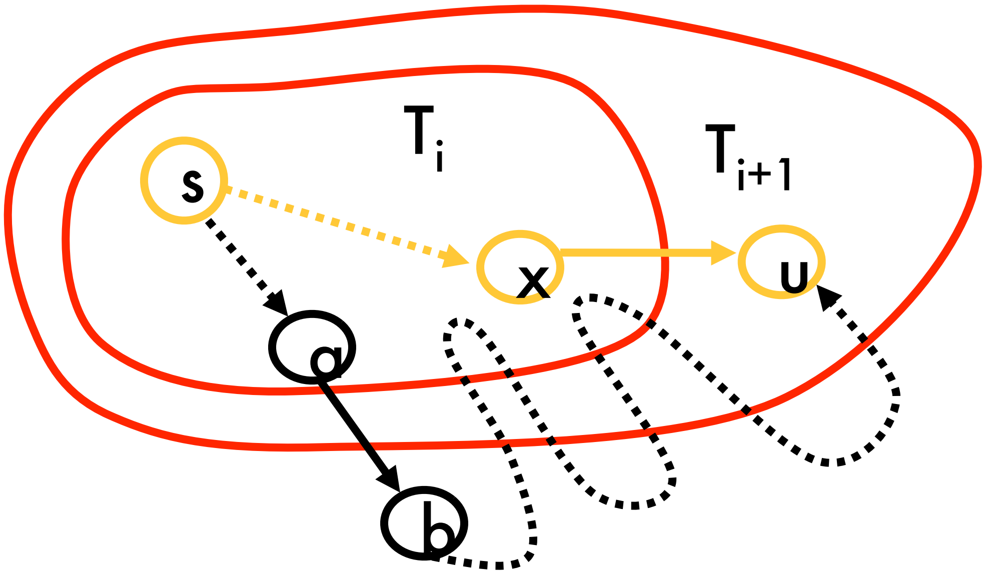
Need to show path in T_{i+1} from s to u is a shortest path, and has distance d[u]

Correctness of Dijkstra's Alg



P, path in T_{i+1} from s to u

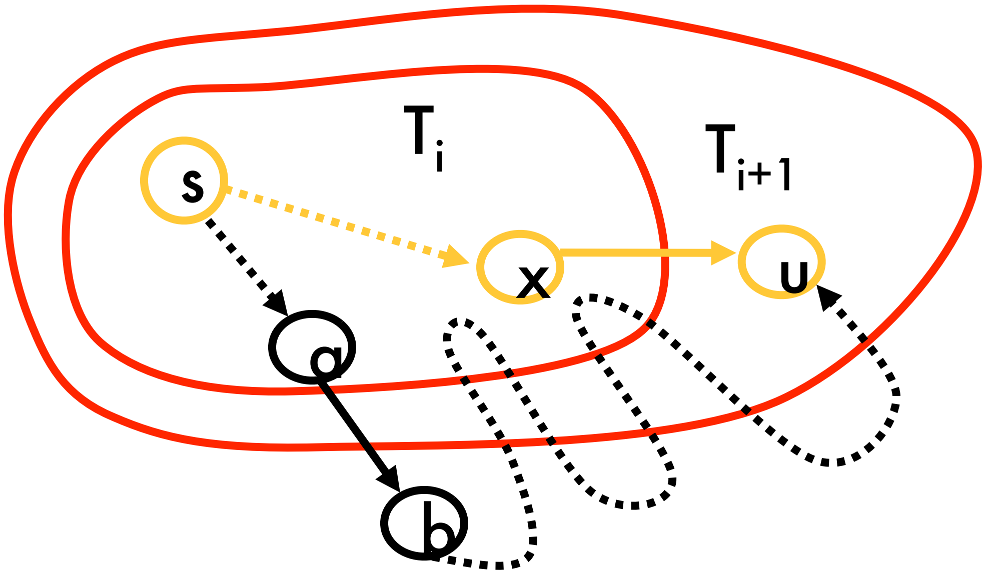
Correctness of Dijkstra's Alg



P, path in T_{i+1} from s to u

P', another path from s to u

Correctness of Dijkstra's Alg

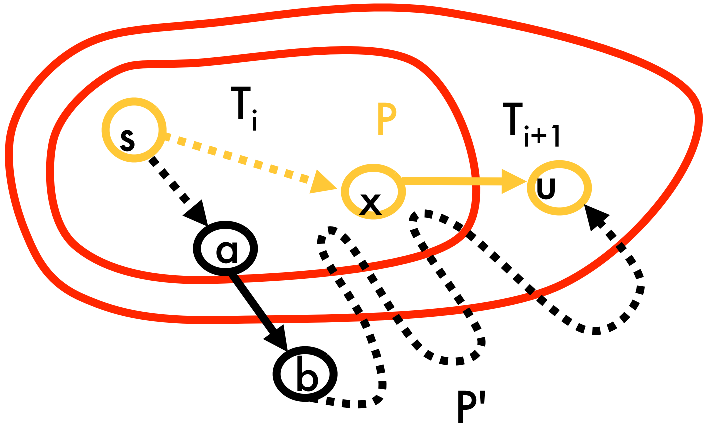


P, path in T_{i+1} from s to u

P', another path from s to u

(a,b) is first edge in P' that leaves T_i

Correctness of Dijkstra's Alg



Let P1 be part of P' before (a,b).

Let P2 be part of P' after (a,b).

w(P') = w(P1) + w(a,b) + w(P2)

≥ w(P1) + w(a,b) (nonneg wts)

≥ wt of path in T_i from s to a + w(a,b) (inductive hypothesis)

≥ w(s->x path in T_i) + w(x,u) (alg chose u in iteration i and

d-values are accurate, by inductive hypothesis

= w(P).

So P is a shortest path, and d[u] is accurate after iteration i+1.

Running Time of Dijstra's Alg.

- initialization: insert each node once
  - $O(V\ T_{ins})$
- O(V) iterations of while loop
  - one extract-min per iteration => $O(V\ T_{ex})$
  - for loop inside while loop has variable number of iterations…
- For loop has O(E) iterations total
  - one decrease-key per iteration => $O(E\ T_{dec})$

Running Time using Binary Heaps and Fibonacci Heaps

- $O(V(T_{ins} + T_{ex}) + E \bullet T_{dec})$
- If priority queue is implemented with a binary heap, then
  - $T_{ins} = T_{ex} = T_{dec} = O(\log V)$
  - total time is $O(E \log V)$
- There are fancier implementations of the priority queue, such as Fibonacci heap:
  - $T_{ins} = O(1)$, $T_{ex} = O(\log V)$, $T_{dec} = O(1)$ (amortized)
  - total time is $O(V \log V + E)$

Using Simpler Heap

- $O(V(T_{ins} + T_{ex}) + E \bullet T_{dec})$
- If graph is dense, so that $|E| = \Theta(V^2)$, then it doesn't help to make T_{ins} and T_{ex} to be at most $O(V)$.
- Instead, focus on making T_{dec} be small, say constant.
- Implement priority queue with an unsorted array:
  - $T_{ins} = O(1)$, $T_{ex} = O(V)$, $T_{dec} = O(1)$

The Bellman-Ford Algorithm

What About Negative Edge

- Dijkstra's SSSP algorithm requires all edge weights to be nonnegative. This is too restrictive, since it suffices to outlaw negative weight cycles.
- Bellman-Ford SSSP algorithm can handle negative edge weights. [It even can detect negative weight cycles if they exist.]

Bellman-Ford: The Basic Idea

- Consider each edge (u,v) and see if u offers v a cheaper path from s
  - compare d[v] to d[u] + w(u,v)
- Repeat this process |V| - 1 times to ensure that accurate information propgates from s, no matter what order the edges are considered in

Bellman-Ford SSSP Algorithm

- input: directed or undirected graph G = (V,E,w)

//initialization

- initialize d[v] to infinity and parent[v] to nil for all v in V other than the source
- initialize d[s] to 0 and parent[s] to s

// main body

- for i := 1 to |V| - 1 do
  - for each (u,v) in E do // consider in arbitrary order
  - if d[u] + w(u,v) < d[v] then
    - d[v] := d[u] + w(u,v)
    - parent[v] := u

Bellman-Ford SSSP Algorithm

// check for negative weight cycles

- for each (u,v) in E do
  - if d[u] + w(u,v) < d[v] then
    - output "negative weight cycle exists"

Running Time of Bellman-Ford

- O(V) iterations of outer for loop
- O(E) iterations of inner for loop
- O(VE) time total

Correctness of Bellman-Ford

Assume no negative-weight cycles.

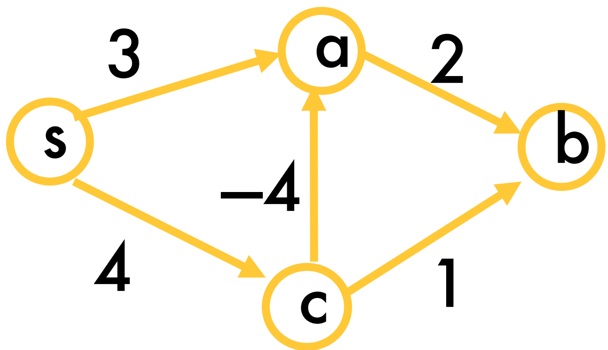
Lemma: d[v] is never an underestimate of the actual shortest path distance from s to v.

Lemma: If there is a shortest s-to-v path containing at most i edges, then after iteration i of the outer for loop, d[v] is at most the actual shortest path distance from s to v.

Theorem: Bellman-Ford is correct.

This follows from the two lemmas and the fact

Bellman-Ford Example



process edges in order
(c,b)
(a,b)
(c,a)
(s,a)
(s,c)

Exercise!

Correctness of Bellman-Ford

- Suppose there is a negative weight cycle.
- Then the distance will decrease even after iteration |V| - 1
  - shortest path distance is negative infinity
- This is what the last part of the code checks for.

The Boost Graph Library

The BGL contains generic implementations of all the graph algorithms that we have discussed:

- Breadth-First-Search
- Depth-First-Search
- Kruskal's MST algorithm
- Prim's MST algorithm
- Strongly Connected Components
- Dijkstra's SSSP algorithm
- Bellman-Ford SSSP algorithm

I recommend that you gain experience with this useful library. Recommended reading: The Boost Graph Library by J.G. Siek, L.-Q. Lee, and A. Lumsdaine, Addison-Wesley, 2002.