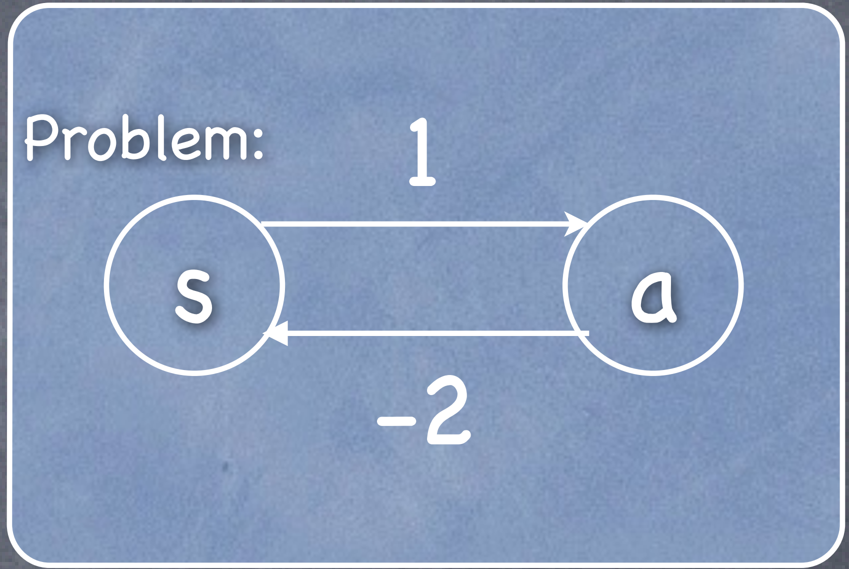
Dijkstra's Single Source Shortest Path Algorithm

Andreas Klappenecker

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 v in V, find a path of minimum total weight from the source node s to v.

Special Case

Suppose that the weights of all edges are the same. Then 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.

Goal: Solve the more general problem of single-source shortest path problems with arbitrary (non-negative) edge weights.

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. It 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 key to the lower value k, where k < key[x].

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 even realize priority queues with worst case times as above]

Dijkstra's Single Source Shortest Path Algorithm

Dijkstra's SSSP Algorithm

We assume all edge weights are nonnegative.

Start with source node s and iteratively construct a tree rooted at s.

Each node keeps track of the tree node that provides cheapest path from s.

At each iteration, we include the node into the tree whose cheapest path from s is the overall cheapest.

Implementation Questions

How can each node keep track of its best path to s?

How can we know which node that is not in the tree yet has the overall cheapest path?

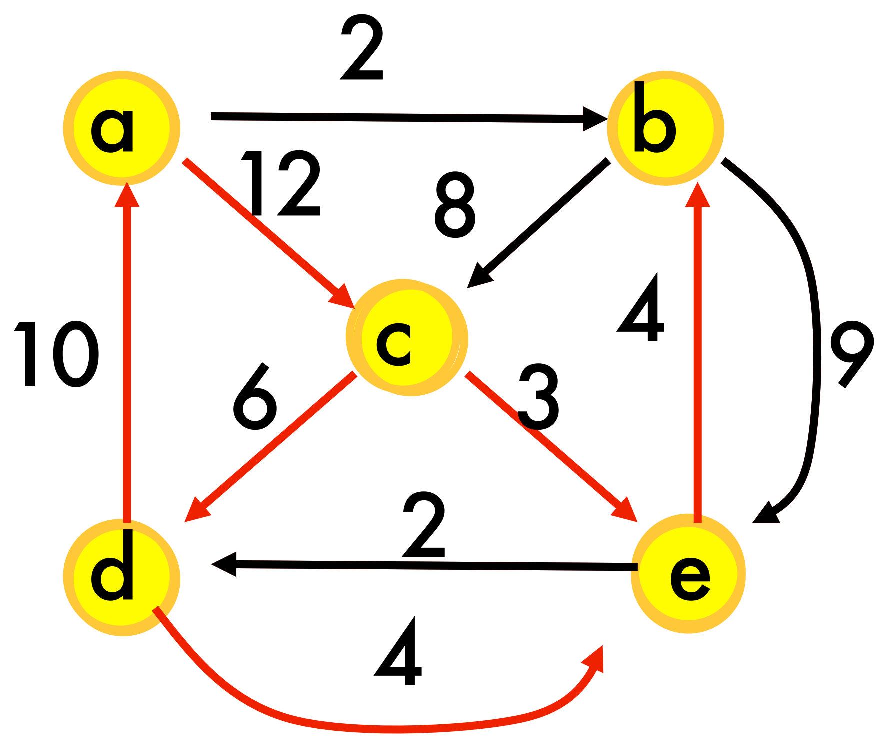
How can we maintain the shortest path information of each node after adding a node to the shortest path tree?

Dijkstra's Algorithm

```
Input:  G = (V,E,w) and source node s

for all nodes v in V {

   d[v] := infinity

}

d[s] := 0

Enqueue all nodes in priority queue
Q
```

```
while (Q is not empty) {

  u := extract-min(Q)

  for each neighbor v of u {

      if (d[u] + w(u,v) < d[v]) { // 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	∅
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

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 $d[s] = 0$.

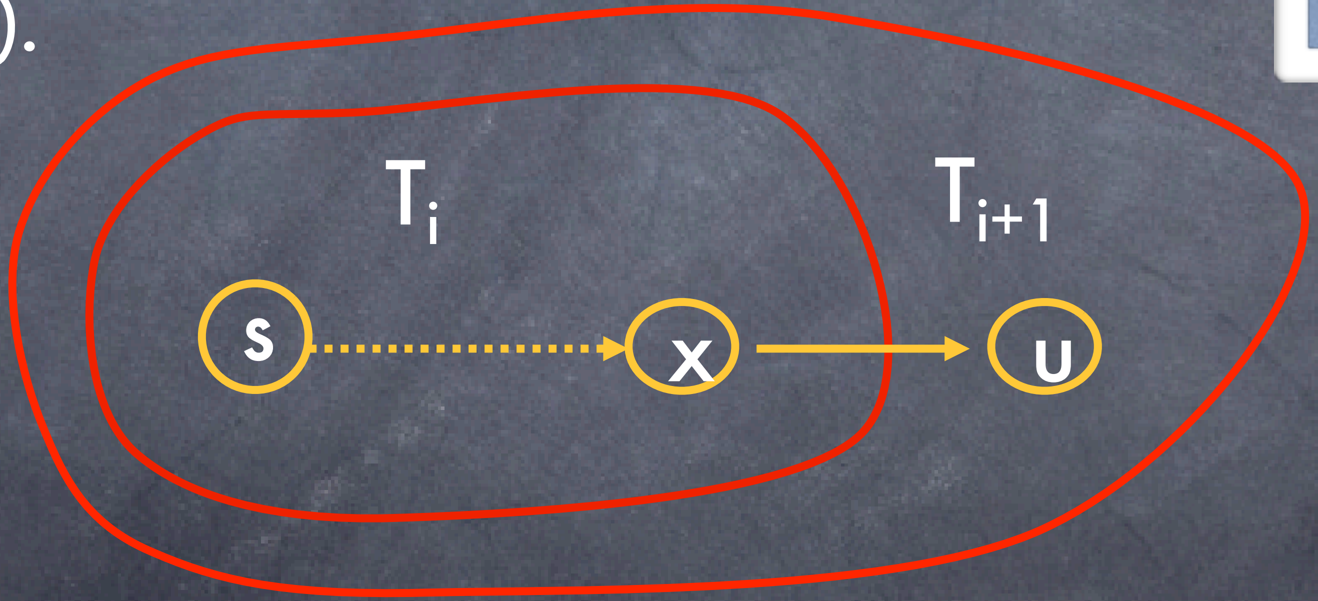
Correctness

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.

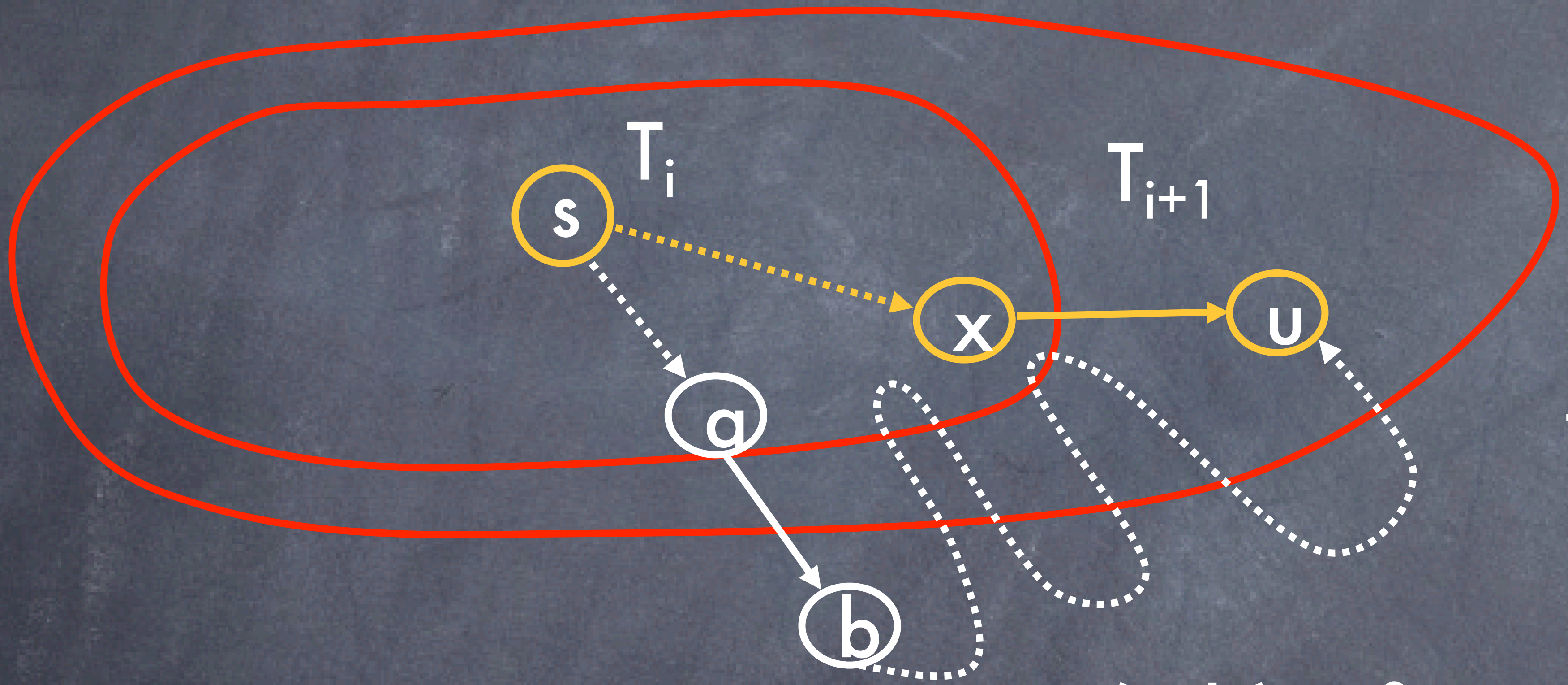
Let u be the node added in iteration i.

Let x = parent(u).

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



Correctness



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

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)

$\geq$ w(P1) + w(a,b) (since weight are nonnegative)

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

$\geq$ w(s->x path in T_i) + w(x,u) (alg chose u in iteration i and d-values are accurate, by I.H.)

= w(P).

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

Running Time

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 => $O(E\ T_{dec})$

Total is $O(V\ (T_{ins} + T_{ex}) + E\ T_{dec})$ // details depend on min-queue implementation

Running Time using Binary Heaps and Fibonacci Heaps

Recall, total running time is $O(V(T_{ins} + T_{ex}) + E \bullet T_{dec})$

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 $O(V \log V + E)$

Running Time using Simpler Heap

In general, running time is $O(V(T_{ins} + T_{ex}) + E \cdot T_{dec})$.

If graph is dense, say $|E| = \Theta(V^2)$, then T_{ins} and T_{ex} can be $O(V)$, but T_{dec} should be $O(1)$.

Implement priority queue with an unsorted array:

- $T_{ins} = O(1)$, $T_{ex} = O(V)$, $T_{dec} = O(1)$
- Total running time is $O(V^2)$

Credits

This set of slides is based on slides prepared by Jennifer Welch.