Washington University in St. Louis

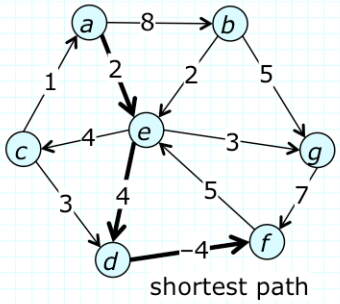
Engineering

Shortest Paths in Directed Graphs

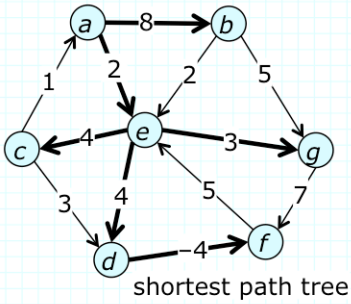
Jon Turner
Computer Science & Engineering
Washington University

www.arl.wustl.edu/~jst

Shortest Paths in Graphs



shortest path



shortest path tree

- Let $G = (V,E)$ be a directed graphs and let *length* be a real-valued function on E
  - »the *length of a path* is the sum of the lengths of its edges
  - »a *shortest path* is a path of minimum length
  - »edge lengths can be negative, so path lengths can be also
    - but no shortest path in presence of negative length cycles

Shortest Path Problems

- The *shortest path problem* has several versions
  - »*single pair problem*: given a *source vertex s* and a *sink vertex t*, find a shortest path from *s* to *t*
  - »*single source problem*: given a source vertex *s*, find a shortest path from *s* to every other vertex in *G*
  - »*single sink problem*: given a sink vertex *t*, find a shortest path to *t* from every other vertex in *G*
  - »*all pairs problem*: find a shortest path between every pair of vertices
- The single source and single sink problems are essentially the same
- Most solutions to the single pair problem effectively solve a single source problem

Shortest Path Trees

- *Theorem 7.1*. Let G be digraph with path from s to t. There is a shortest path from s to t if and only if no path from s to t contains a negative cycle. If there is a shortest path from s to t, there is one that is simple

  Proof. If some s-t path includes a negative cycle, we can produce an arbitrarily short path by traversing the cycle enough times.

  If there is no negative cycle then we can make any s-t path simple without increasing its length by deleting cycles. ■
- A *shortest path tree* for vertex s is a directed spanning tree with root s in which all paths are shortest paths.
- *Theorem 7.2*. G contains shortest paths from s to every other vertex if and only if it contains a shortest path tree with root s. *Proof deferred*

- Let T be any spanning tree of G with root s
  - » let *distance*(v) to be length of the path from s to v in T
- *Theorem 7.3*. T is a shortest path tree if and only if, *distance*(w)≤*distance*(v)+*length*(v,w), for every edge [v,w] in G

  Proof. If T is shortest path tree in which *distance*(w)>*distance*(v) +*length*(v,w) for some edge [v,w], then there is a shorter path than the one in T

  Conversely, suppose *distance*(w)≤*distance*(v)+*length*(v,w) for every edge [v,w] and let p be any path from s to any vertex x
  - show by induction on number of edges in p that *distance*(x)≤*length*(p); the basis, $|p|=1$ is immediate
  - for inductive step, assume *distance*(x')≤*length*(p') for all vertices x' and paths p' with $|p'|=k$; let p be a path from s to some vertex x of length $k+1$, let $q=p-[x]$ and let y be last vertex on q;
  - by induction, *distance*(y)≤*length*(q), so *distance*(x)≤ *distance*(y)+*length*(y,x)≤*length*(q)+*length*(y,x)=*length*(p) ■

The Labeling Method

- Labeling method finds an SPT by repeatedly applying Theorem 7.3
  - » uses two mappings: $dist(v)$ is length of shortest path so far from s to v, $p(v)$ is the parent of v in the current partial SPT
  - » initially, $p(s)$=**null**, $dist(s)$=0 and $p(v)$=**null**, $dist(v)=\infty$ for $v \neq s$
  - » *Labeling Step*: select edge $[v,w]$ for which $dist(w) > dist(v) + length(v,w)$; let $p(w)=v$, $dist(w)=dist(v)+length(v,w)$
- *Lemma 7.1*. The labeling method maintains invariant that if $dist(v)$ is finite, there is a path from s to v of length $dist(v)$

  Proof. By induction on the number of labeling steps ■
- *Lemma 7.2*. If p is a path from s to any vertex v, then $dist(v) \leq length(p)$ when the labeling method halts

  Proof. By induction on number of edges in p ■

- *Theorem 7.4.* When labeling method halts, $dist(v)$ is length of a shortest path from s to v if v is reachable from s; $dist(v)=\infty$ otherwise. If there is a negative cycle reachable from s, the method never halts.

  Proof. Follows from Theorem 7.1 and Lemmas 7.1, 7.2 ■
- *Lemma 7.3.* Labeling method maintains the invariant that if $p(v)\neq$**null**, $dist(v)\geq dist(p(v))+length(p(v),v)$ with equality when the method halts.

  Proof. By induction on number of labeling steps ■
- *Lemma 7.4.* At each step of labeling method, either the edges $[p(v),v]$ with $p(v)\neq$**null** form a tree rooted at s spanning all vertices v with $dist(v)<\infty$, or there is a cycle of parent pointers.

  Proof. A vertex $v\neq s$ has $p(v)\neq$**null** iff $dist(v)<\infty$. If $dist(v)<\infty$ and $p(v)\neq$**null** then $dist(p(v))<\infty$. Thus, if we follow parent pointers from any vertex v, we either encounter s or repeat a vertex. ■

- *Lemma 7.5*. If labeling method creates a cycle of parent pointers, the corresponding cycle of G is negative

  Proof. Suppose that a labeling step on edge $[x,y]$ creates a cycle of parent pointers; just before the step, $p^k(x)=y$ for some k; by Lemma 7.3, $dist(p^i(x)) \geq dist(p^{i+1}(x))+length(p^{i+1}(x),p^i(x))$ for $0 \leq i < k$; summing these inequalities gives

$$\sum_{i=0}^{k-1} dist(p^i(x)) \geq \sum_{i=1}^{k} dist(p^i(x)) + \sum_{i=0}^{k-1} length(p^{i+1}(x), p^i(x))$$

  adding the inequality $dist(y)>dist(x)+length(x,y)$ and noting that $y=p^k(x)$ and $x=p^0(x)$ gives

$$\sum_{i=0}^{k} dist(p^i(x)) > \sum_{i=0}^{k} dist(p^i(x)) + \sum_{i=0}^{k-1} length(p^{i+1}(x), p^i(x)) + length(x,y)$$

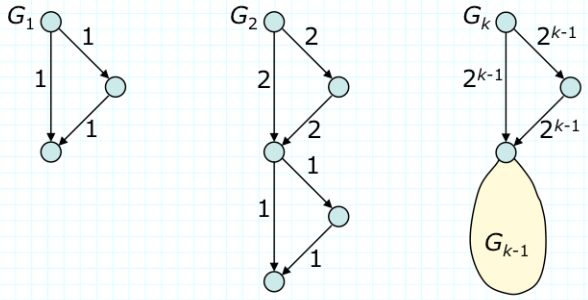
$$0 > \sum_{i=0}^{k-1} length(p^{i+1}(x), p^i(x)) + length(x,y)$$

  hence the cycle of parent pointers created by the labeling step corresponds to a negative cycle of G ■

- *Theorem 7.5*. When the labeling method halts, the parent pointers define a shortest-path tree for the subgraph of G reachable from s

  Proof. Immediate from Theorem 7.4 and Lemmas 7.3, 7.4, 7.5 ■

- Labeling method can take $\Omega(2^n)$ time if order in which edges are chosen is poor
  - » for example, labeling method can take $\Omega(2^k)$ steps on the graph G_k defined below



Scanning and Labeling

- Scanning and labeling method is a special case of the labeling method, where we select all "eligible" edges leaving a particular vertex as a group
  - » each vertex can be in one of three states: *unlabeled*, *labeled*, or *scanned*; initially *s* is labeled and all others are unlabeled
  - » *Scanning Step*. Select a *labeled* vertex *v* and convert *v* to the scanned state; for all [*v*,*w*] with *dist*(*w*)>*dist*(*v*)+ *length*(*v*,*w*), replace *p*(*w*) by *v* and *dist*(*w*) by *dist*(*v*)+ *length*(*v*,*w*) convert *w* to the labeled state
  - » correctness follows directly from correctness of labeling method
- Different vertex selection rules yield different variants
- For acyclic graphs select vertices to be scanned in topological order
  - » no vertex is scanned more than once giving *O*(*m*) running time

Shortest First Scanning

- In shortest first scanning, select vertex *v* for which *dist*(*v*) is minimum
  - » if every edge has non-negative length, each vertex will be scanned at most once.
- *Theorem 7.6*. If every edge has non-negative length and scanning is shortest first, then after vertex *v* is scanned, *dist*(*v*) is length of shortest path from *s* to *v*.

  Proof. Just before a vertex *v* is scanned, $dist(v) \le dist(w)$ for any labeled vertex *w*

  In addition, $dist(v) \le dist(w)$ for any vertex *w* that becomes labeled during or after the scanning of *v*

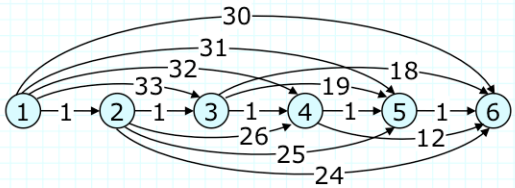
  Consequently, the vertices are scanned in non-decreasing order of distance from *s* and once a vertex is scanned it cannot become labeled ■

Dijkstra's Algorithm

- Similar to Prim's MST algorithm; same running time

```
procedure dijkstra(digraph G=(V,E), vertex s,
                   mapping p: vertex→vertex);
    vertex u,v; heap S;
    for u∈V ⇒ dist(v):=∞; p(u):=null; rof;
    dist(s):=0;  insert(s,0,S);
    do S≠{} ⇒
       u:=deletemin(S);
       for {u,v}∈out(u) ⇒
          if dist(v)>dist(u)+length(u,v) ⇒
             p(v):=u; dist(v):=dist(u)+length(u,v);
             if  v∉S  ⇒ insert(v,dist(v),S);
              | v∈S  ⇒ changekey(v,dist(v),S);
          fi; fi;
rof; od; end;
```

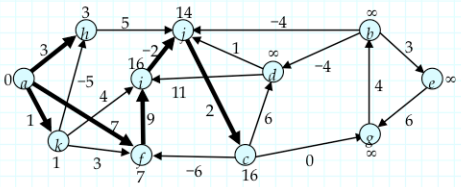
Worst Case for Dijkstra



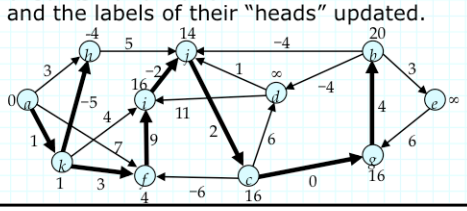
- $cost(v_i,v_j)=n(n-i)+(n-j)$ for $i<j-1$, $cost(v_i,v_{i+1})=1$
- After step k the "current tree" includes path $v_1,v_2,\ldots,v_k$ and direct edge from v_k to v_j for $j>k$
- Number of *changekeys* is $1+2+\ldots+(n-2)=\Omega(n^2)$
- If edges processed in "increasing-head order", each item moves from "bottom of heap" to "near top"
- Can remove edges $[v_i,v_j]$ with $1<i<j-1$ to get sparse graph with $\Omega(m)$ *changekey* ops

Exercises

1. The figure below shows an intermediate state in the execution of the labeling algorithm for shortest paths starting at vertex *a*. The heavy edges are the tree edges defined by parent pointers in the implementation. The distance values are shown next to the nodes in the diagram.



Show the state of the algorithm after edges (*c*,*g*), (*g*,*b*), (*k*,*h*), (*k*,*f*) have been selected and the labels of their "heads" updated.



2. Let *G* be a digraph in which edges have "capacities". The *bottleneck capacity* of a path in *G* is the minimum edge capacity on the path. A *best bottleneck path tree* is a spanning tree of *G* in which each path has the largest possible bottleneck capacity.

   Let *T* be a spanning tree of *G* with root *s* and let *bcap*(*u*) be the bottleneck capacity of the path from *s* to *u* in *T*. Show that if *T* is a best bottleneck path tree, then *bcap*(*v*)≥min{*bcap*(*u*), *cap*(*u*,*v*)}, for all edges (*u*,*v*) in *G*.

   If bcap(*v*)<min{*bcap*(*u*),*cap*(*u*,*v*)} *for some edge* (*u*,*v*) *then there is a path to v with a larger bottleneck capacity than the tree path, implying that T is not a best bottleneck path tree. Therefore, if T is a best bottleneck path tree then bcap*(*v*)≥ min{*bcap*(*u*),*cap*(*u*,*v*)}, *for all edges* (*u*,*v*) *in G*.

   Show that if there is a non-tree path *p* from *s* to some vertex *x*, that has bottleneck capacity larger than *bcap*(*x*), then there must be some edge (*u*,*v*) for which *bcap*(*v*)<min{*bcap*(*u*),*cap*(*u*,*v*)}.

Assume that p is a shortest path (by edge count) satisfying the condition and that path p has k edges. If k=1, p consists of the single edge (s,x) and has bottleneck capacity equal to cap(s,x)=min{bcap(s),cap(s,x)}, and since p is assumed to have larger bottleneck capacity than the tree path to x, it follows that bcap(x)<min{bcap(s),cap(s,x)}.

If k>1, let (w,x) be the last edge on p and note that since p is the shortest non-tree path with larger bottleneck capacity than the corresponding tree path, the tree path from s to w is a best bottleneck path. Consequently, the bottleneck capacity of p is at most min{bcap(w),cap(w,x)}. Since p has larger bottleneck capacity than the tree path to x, this means that bcap(x)<min{bcap(w),cap(w,x)}.

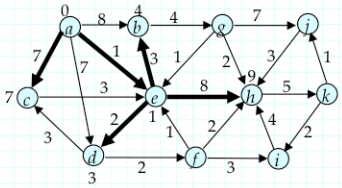
3. Consider a version of the shortest path problem with positive edge lengths and *multiplicative costs*. In this version, the length of a path is the *product* of the edge lengths, rather than the sum. Explain how shortest path algorithms can be modified to handle multiplicative costs. What is the implication of having positive edge costs that are less than 1?

   One approach is to replace the original costs by their logarithms and then use a standard shortest path algorithm on the graph with the transformed costs. The resulting distances must then be transformed back to give the multiplicative distances. Since the log of a product equals the sum of the logs, this yields the shortest multiplicative paths.

   *Alternatively, one can modify the original algorithms to replace expressions of the form dist(u) + length(u,v) with dist(u) * length(u,v) in the shortest path algorithm. The correctness of this follows from the logarithm transformation.*

   Positive edge costs less than 1 reduce path lengths and have the same effect as negative edge lengths in the standard problem. If such edges are present, Dijkstra's algorithm cannot be used.

4. The figure below shows an intermediate state in the execution of Dijkstra's algorithm. The bold edges in the graph are the edges defined by the parent pointers, and the numbers next to the vertices are the current distance values. Fill in the blanks (as appropriate) in the arrays that implement the *d*-heap (assume *d*=2 and that vertices *b*, *c*, *d* and *h* have not yet been "scanned").



	1	2	3	4	5	6	7	8	9	10	11	12
h	*d*	*b*	*c*	*h*								

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*
key		4	7	3				9				
h^{-1}		2	3	1				4				

Show how the heap contents changes after the next iteration.

	1	2	3	4	5	6	7	8	9	10	11	12
h	*b*	*f*	*c*	*h*								

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*
key		4	6			5		9				
h^{-1}		1	3			2		4				