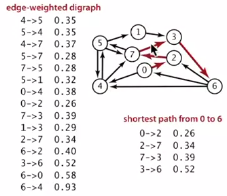
Shortest Paths APIs

Given an edge-weighted digraph, find the shortest path from s to t.



Many applications

* Map routing
* Seam carving
* Robot navigation
* Traffic planning

Shortest path variants

* Source –sink: from one vertex to another
* Single source: from one vertex to every other
* All pairs: between all pairs of vertices

Restrictions on edge weights:

* Nonnegative weights (usually positive edge weights because maps are geometric and the length of an edge is proportional to its distance in the plane)
* Arbitrary weights
* Euclidean weights

Cycles?

* No directed cycles
* No “negative cycles”

Simplifying assumption: shortest paths from s to each vertex v exist

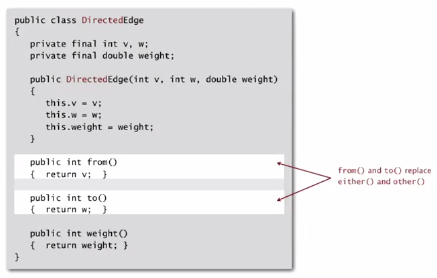
Weighted directed edge API

Public class DirectedEdge  
DirectedEdge(int v, int w, double weight) : weighted edge v -> w  
int from() : vertex v  
int to() : vertex w  
double weight() : weight of this edge  
String toString() : string representation

weight  
(v) ---------🡪 (w)

Idiom for processing an edge e: int v = e.from(), w = e.to();

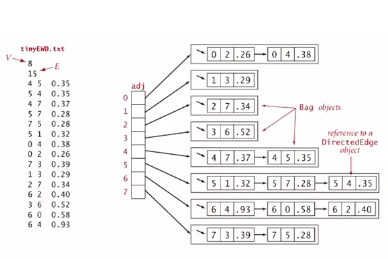
Weighted directed edge implementation



Edge-weighted digraph API

Public class EdgeWeightedDigraph  
**EdgeWeightedDigraph(int V) : edge-weighted digraph with V vertices**  
EdgeWeightedDigraph(In in) : edge-weighted digraph from input stream  
**void addEdge(DirectedEdge e) : add weighted directed edge e  
Iterable<DirectedEdge> adj(int v) : edges pointing from v  
int V() : number of vertices**int E() : number of edges  
Iterable<DirectedEdge> edges() : all edges  
String toString() : string representation

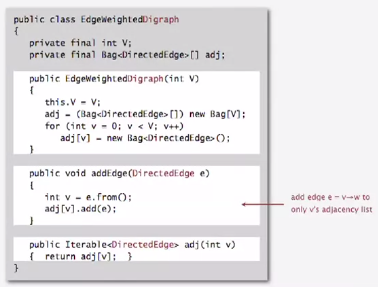
Adjacency-lists representation



Representation is simpler, yet easier, since only one representation of each edge\

Conventions: Allow self-loops and parallel edges

Edge-weighted digraph adjacency lists implementation



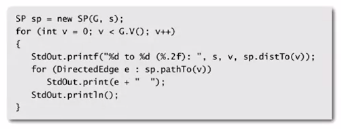
Only two changes:

1. Name is changed
2. Only add edge to one (v’s) adjacency list (the from vertex)

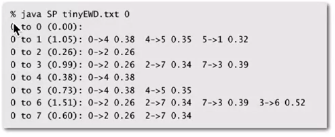
Single-source shortest paths API

(Goal: find the shortest path from s to every other vertex)

Public class SP  
**SP(EdgeWeightedDigraph G, int s) : shortest path from s in graph G  
double distTo(int v) : length of shortest path from s to v  
Iterable<DirectedEdge> pathTo(int v) : shortest path from s to v**  
boolean hasPathTo(int v) : is there a path from v to s?



Result of implementation:



Shortest path properties

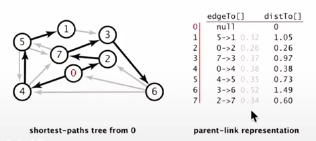
Observation: a shortest-paths-tree (SPT) solution exists. Why?

If no two paths have the same lengths, then of course this is true…:

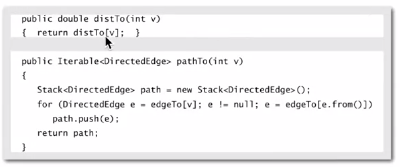
* E.g.: If 2 paths to same vertex, delete last edge on one and keep going until all that’s left is a tree

Consequence: Can represent the SPT with two vertex-indexed arrays:

* distTo[v] is length of shortest path from s to v
* edgeTo[v] is last edge on shortest path from s to v



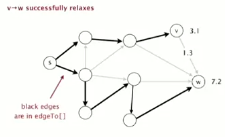
Implementation for the above:

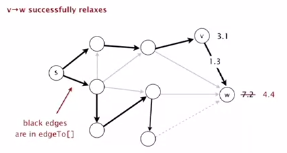


Edge relaxation

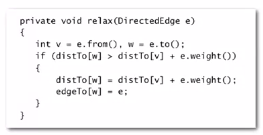
Relax edge e = v -> w

* distTo[v] is length of shortest known path from s to v
* distTo[w] is length of shortest known path from s to w
* edgeTo[w] is last edge on shortest known path from s to w
* If e = v -> w gives shorter path to w through v, update both distTo[w] and edgeTo[w]





Implementation

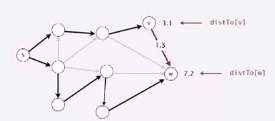


Shortest Path optimality conditions

Proposition: let G be an edge-weighted digraph

Then distTo[] are the shortest path distances from s iff:

* distTo[s] = 0
* For each vertex v, distTo[v] is the length of some path from s to v
* For each edge e = v -> w, distTo[w] <= ditto[v] + e.weight()



Proof:

* Suppose that distTo[w] > distTo[v] + e.weight() for some edge e = v -> w
* Then, e gives a path from s to w (through v) of length less than distTo[w]

Proof:

* Suppose that s = v0 -> v1 -> v2 -> …-> vk = w is the shortest path from s to w  
  distTo[v1] <= distTo[v0] + e1.weight()  
  distTo[v2] <= distTo[v1] + e2.weight()  
  …  
  distTo[vk] <= distTo[vk-1] + ek.weight()  
    
   (*e = ith edge on shortest path from s to w)*
* Add inequalities; simplify; and substitute distTo[v0] = distTo[s] = 0:

distTo[w] = distTo[vk] <= e1.weight() + e2.weight() + … + ek.weight()   
 (weight of shortest path from s to w)

* Thus, distTo[w] (weight of some path from s to w) is the weight of shortest path to w

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize distTo[s] = 0 and distTo[v] = infinity for all other vertices  
Repeat until optimality conditions are satisfied:

* Relax any edge

Proposition: generic algorithm computes SPT (if it exists) from s

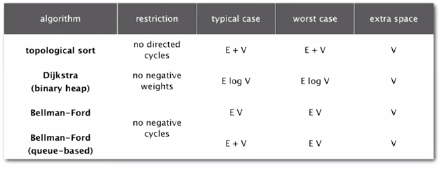
Proof:

* Throughout algorithm, distTo[v] is the length of a simple path from s to v (and edgeTo[v] is last edge on path)
* Each successful relaxation decreases distTo[v] for some v
* The entry distTo[v] can decrease at most a finite number of times

Efficient implementations:

1. Djikstra’s algorithm (nonnegative cycles)
2. Topological sort algorithm (no directed cycles)
3. Bellman-Ford algorithm (no negative cycles)

Single source shortest-paths implementation cost summary



Notes:

* Directed cycles make the problem harder (Djikstra needed)
* Negative weights make the problem harder (Bellman-Ford needed)
* Negative cycles make the problem intractable

Shortest paths summary

Djikstra’s algorithm:

* Nearly linear time when weights are nonnegative
* Generalization encompasses DFS, BFS and Prim

Acyclic edge-weighted digraphs:

* Arise in applications
* Faster than Djikstra’s algorithm
* Negative weights are no problems

Negative weights and negative cycles:

* Arise in applications
* If no negative cycles, can find shortest paths via Bellman-Ford
* If negative cycles, can find one via Bellman-Ford

Shortest paths is a broadly useful problem solving model.