22 Dijkstra's Shortest Path Algorithm

Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/19/

Bachelor of Science in Computing, School of Public Administration, Macao Polytechnic Institute

April 24, 2019

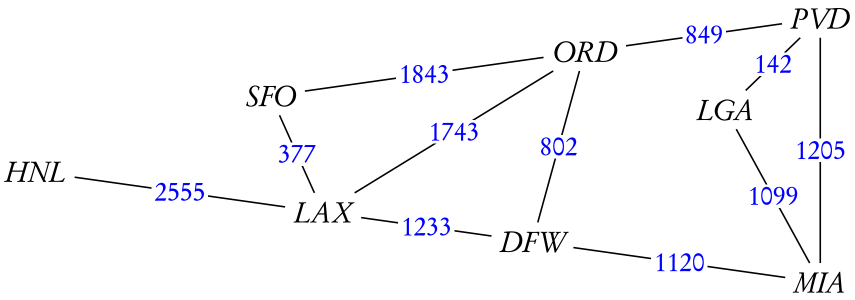
Outline

1. Edge-Weighted Graphs
2. Shortest Paths
3. Dijkstra's Algorithm
4. Analysis

Edge-Weighted Graphs

- An *edge-weighted* graph is a graph having a weight, or number, associated with each edge.
- Some algorithms require all weights to be non-negative.
- Edge weights may represent distances, costs, delays, etc.

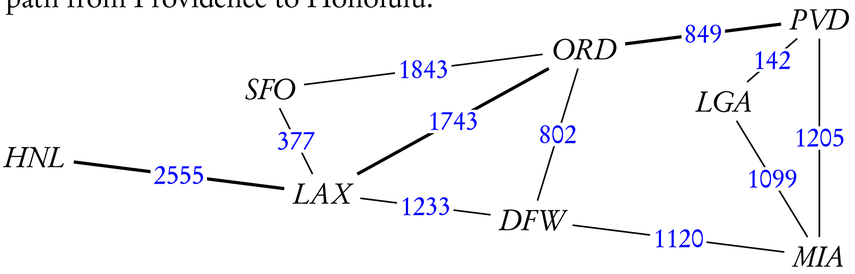
A flight route graph:



Shortest Paths

- Given an edge-weighted graph and two vertices u and v, we want to find a path with the minimum total weight between u and v. This is the *shortest path problem*.
- Some applications:
  - Flight reservations
  - Driving directions
  - Network packet routing

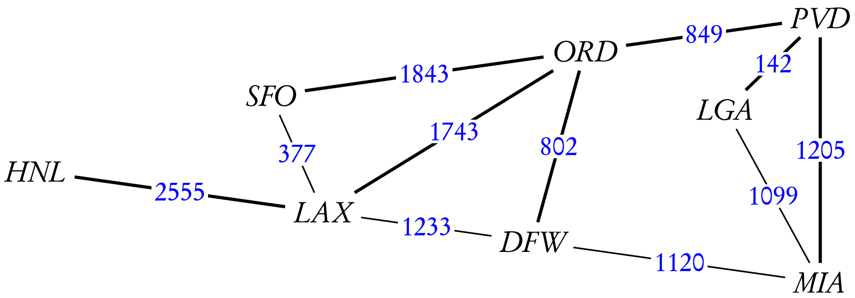
The shortest path from Providence to Honolulu:



Properties of Shortest Paths

- A sub-path of a shortest path is itself a shortest path.
- The shortest paths from a starting vertex *s* to all the other vertices form a tree rooted at *s*.

The tree of shortest paths from Providence:

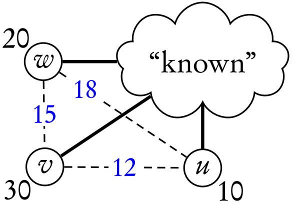


Dijkstra (1930–2002)'s Algorithm

- The distance of a vertex v from a vertex s is the total weight of a path between s and v.
- Dijkstra's algorithm computes the shortest distances of all the vertices from a given starting vertex s.
- Assumptions:
    - The graph is connected. Edges of infinite weight can be introduced to apply the algorithm to a general graph.
    - The edge weights are *non-negative*. A path can not be shortened by appending more edges.

Dijkstra's Algorithm (2)

- We grow a set of "known" vertices,
  - whose shortest distances are already known and can not be shortened further,
  - beginning with s and eventually containing all the vertices.
- We store with each vertex v a field $dist(v)$, called the *distance* of v, representing the shortest distance of v from s in the *subgraph* consisting of
  - the set of "known" vertices, often called the "cloud", and
  - their adjacent vertices, with only the edges from the "known" vertices (the cloud).
- At each step:
  - we add to the set the vertex u outside the set with the shortest distance field, then
  - we update the distance fields of the vertices adjacent to u, if the fields can be shortened.

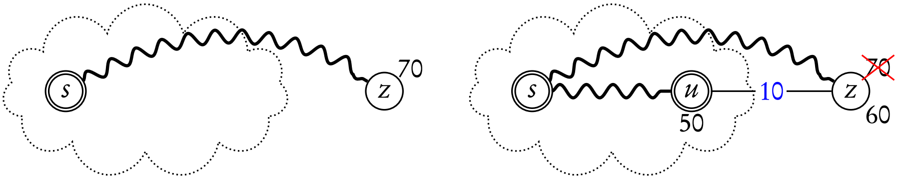


Edge Relaxation

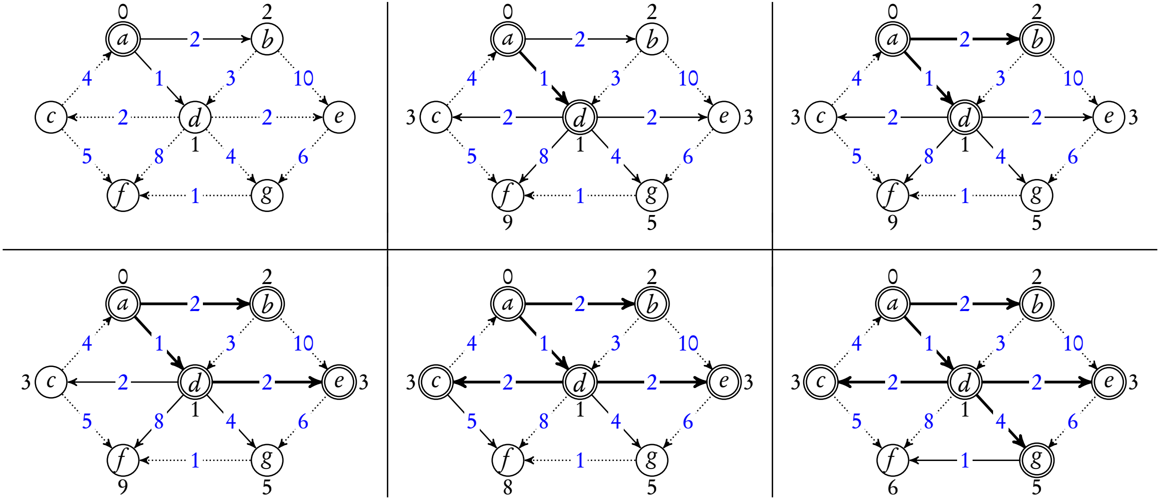
- Consider an edge $e = (u, z)$ such that
  - u is the vertex most recently added to the "known" set.
  - z is not in the "known" set.
- The relaxation of the edge e updates $dist(z)$, the distance of z, as follows:

$$dist(z) \leftarrow \min(dist(z), dist(u) + weight(e)).$$

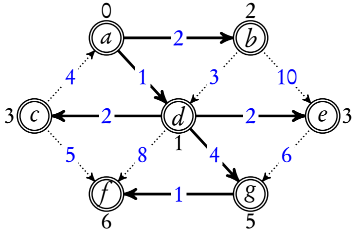
- We also record u as the parent of z in a field $parent(z)$. We can then use the field to track back the path from s to z.



Dijkstra's Algorithm — Illustrated



Dijkstra's Algorithm — Illustrated (2)



Step	Next Vertex	Current Distance (Parent)						
		a	b	c	d	e	f	g
—	—	$0_{(-)}$	$\infty_{(-)}$	$\infty_{(-)}$	$\infty_{(-)}$	$\infty_{(-)}$	$\infty_{(-)}$	$\infty_{(-)}$
1.	a	✔	$2_{(a)}$		$1_{(a)}$			
2.	d			$3_{(d)}$	✔	$3_{(d)}$	$9_{(d)}$	$5_{(d)}$
3.	b		✔					
4.	e					✔		
5.	c			✔			$8_{(c)}$	
6.	g						$6_{(g)}$	✔
7.	f						✔	

Revisiting the Array-based Heap

- We store the vertices in a heap, and compare vertices based on their distances.
- There's a need to decrease a specified vertex in the heap during a distance update.
- In an array-based heap, an element can be sifted up and down. Therefore, modifying an element is possible.
- We need a vertex to have a place to store its position in the heap.
- When a vertex has been modified, we get its position and notify the heap to sift the vertex at that position up or down.
- When we move a vertex in the heap, we also update its position.

Analysis

- Each vertex is enqueued and dequeued once: $\mathcal{O}(|V|\log|V|)$.
- Each edge is visited once and each visit may call *priorityDecreased* on the destination vertex:

$$\mathcal{O}(|E|\log|V|).$$

- The total running time is $\mathcal{O}((|V|+|E|)\log|V|)$.
- or, since the graph is (weakly) connected ($|E| \geq |V|-1$), the overall running time is

$$\mathcal{O}(|E|\log|V|).$$

- If the graph is a DAG, we can select the vertices according to their topological order.
- This selection rule works because when a vertex v is selected, its distance, $dist(v)$, can no longer be shortened, since by the topological order it has no incoming edges from unknown vertices.
- Since the selection takes constant time, the running time is $\mathcal{O}(|V|+|E|)$.