Lecture 17: Weighted Graphs, Shortest Paths: Dijkstra’s Algorithm

Courtesy of Goodrich, Tamassia and Olga Veksler

Instructor: Yuzhen Xie
Weighted Graphs

- In a **weighted graph**, each edge has an associated numerical value, called the **weight** of the edge.
- Edge weights may represent distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.
  - We will use notation $w(u,v)$ to represent the weight of edge $(u,v)$. 

![Weighted Graph Diagram]

- **Vertices**: HNL, LAX, ORD, SFO, PVD, LGA, DFW, MIA
- **Weights** (in miles):
  - HNL to LAX: 2555
  - LAX to SFO: 337
  - SFO to ORD: 1843
  - ORD to DFW: 849
  - DFW to LGA: 142
  - LGA to PVD: 1099
  - PVD to MIA: 1205
  - MIA to DFW: 1120
  - ORD to LAX: 1743
  - SFO to LAX: 1233
  - SFO to DFW: 802
  - ORD to SFO: 1387
Shortest Paths: Problem Statement

- Given a weighted graph and two vertices \( u \) and \( v \), we want to find a path of minimum total weight between \( u \) and \( v \).
  - Length (or distance) of a path is the sum of the weights of its edges.

- Example:
  - Shortest path between PVD and HNL.

- Applications:
  - Internet packet routing
  - Flight reservations
  - Driving directions
  - etc.

\[
\text{HNL} \rightarrow LAX \rightarrow SFO \rightarrow \text{DFW} \rightarrow \text{ORD} \rightarrow \text{PVD} \rightarrow \text{MIA} \rightarrow \text{LGA} \rightarrow \text{DFW} \rightarrow \text{ORD} \rightarrow \text{HNL}
\]

Lengths:
- HNL to SFO: 2555
- SFO to LAX: 337
- LAX to SFO: 1743
- SFO to ORD: 1233
- ORD to LAX: 802
- LAX to DFW: 1387
- DFW to ORD: 1120
- ORD to LGA: 142
- LGA to DFW: 1099
- DFW to MIA: 1205
- MIA to LGA: 1205
- LGA to ORD: 142
- ORD to HNL: 849
- PVD to ORD: 1205
- ORD to PVD: 1205
Shortest Paths: Assumptions

- **Graph is simple**
  - No parallel edges and no self-loops

- **Graph is connected**
  - If not, run the algorithm for each connected component

- **Graph is undirected**
  - It is simple to extend to directed case

- **No negative weight edges**
  - There is an algorithm to compute shortest paths in a graph with negative edges
  - It has higher time complexity
  - Does not work if there is a negative cost cycle
  - Makes no sense to compute shortest paths in the presence of negative cycles
    - in a graph with a negative cycle, shortest path has cost negative infinity
Suppose we need the shortest path between vertices $u$ and $v$.

The worst case complexity of computing shortest path between $u$ and $v$ is the same as for computing the shortest path between $u$ and all other vertices in $G$.

Therefore we will consider an algorithm which computes shortest distance between vertex $s$ (the “source” vertex) and all other vertices in the graph $G$. This results in a tree of shortest paths.

Instead of computing shortest distance between PVD and SFO, compute shortest distance between PVD and all other cities.
Shortest Paths Algorithm: Basic Idea

- The algorithm works by making **incremental** progress.
- We will have a set of special vertices, let’s call them vertices in a “**blue cloud**”.
- We will maintain the property that for any vertex \( v \) in the blue cloud, the shortest distance from \( s \) to \( v \) has been computed correctly.
- Blue cloud starts empty, and at each iteration grows by 1 vertex.
- After \( n \) iterations, blue cloud will consist of all vertices, and thus we will have computed correct shortest paths distances for all the vertices.
Shortest Paths: Distance Function $d[\nu]$

- For each vertex $\nu$, we maintain distance function $d[\nu]$ with the following properties:
  - If vertex $\nu$ is in the blue cloud, $d[\nu]$ is the shortest distance from $s$ to $\nu$.
  - If vertex $\nu$ is not in the blue cloud, $d[\nu]$ is the distance of the best blue path from $s$ to $\nu$.
    - A path is blue if it uses only the blue cloud vertices for all its vertices, except the last vertex $\nu$. 

![Graph diagram with vertices A, B, C, D, E, F and edges connecting them with weights. The vertex E is highlighted in the blue cloud.](image-url)
Shortest Paths: Initialization

- We start with
  1. empty blue cloud
  2. $d[s] = 0$
  3. $d[v] = \infty$ for all $v$ not equal to $s$

- This start guarantees that for all vertices $v$ in the blue cloud, $d[v]$ has the correct shortest distance from $s$ to $v$
  - since blue cloud is empty 😊

- At each iteration, we need to figure out:
  - Which vertex should be inserted next into the “blue cloud”
  - After inserting a vertex $v$ in the blue cloud, since the blue cloud changed, how do we update distances $d[]$?
    - only need to update distances for vertices adjacent to $v$
Shortest Paths Algorithm: Main Part

- The answer to previous questions is:
  - Insert in the blue cloud vertex $u$ which has the **smallest** $d[u]$ and which is **not in** the blue cloud yet
  - For any vertex $z$ which is not in the blue cloud yet, and is adjacent to $u$, update it’s distance $d[z]$ using:
    
    $$d[z] \leftarrow \min\{d[z], d[u] + w(u,z)\}$$

- $w(u,z)$ is the weight of edge $(u,z)$
Shortest Paths: Edge Relaxation

1. Into the blue cloud, add the vertex $u$ which is not in the blue cloud yet and which has the smallest $d[u]$
2. For any vertex $z$ which is not in the blue cloud yet, and is adjacent to $u$, update it’s distance $d[z]$ using:
   \[ d[z] \leftarrow \min\{d[z], d[u] + w(u,z)\}\]

- The second step sometimes is called **edge relaxation**
- After edge relaxation, we may have discovered a shorter path from $s$ to $z$ than previously known. New path goes through $u$

old best path from $s$ to $z$ is in thick green and its length is 75
new best path from $s$ to $z$ is in green and its length is 60
Lemma 1: any sub-path of a shortest path is a shortest path itself.

Proof: Let $P$ be the shortest path from $s$ to $v$.

- Let $u$ and $t$ be any nodes on this path, s.t. $u$ comes before $t$ and let $P_{ut}$ be part of path $P$ from $u$ to $t$.
- Suppose $P_{ut}$ is not the shortest path from $u$ to $t$.
- Then there is a path $Q$ from $u$ to $t$ which is shorter than $P_{ut}$.
- Let $P_{su}$ be the part of path $P$ from $s$ to $u$, and let $P_{tv}$ be the part of path $P$ from $t$ to $v$.
- The combination of paths $P_{su}$, $Q$, and $P_{tv}$ would be a shorter path from $s$ to $v$ than path $P$, which is a contradiction.
Lemma 2: At each step of the algorithm, for any vertex \( v \),
\( d[v] \) is either infinite or the length of some path from \( s \) to \( v \).

Proof: (by contradiction)

- Lemma 2 is true in the beginning, \( d[s] = 0 \), \( d[v] = \infty \) for \( v \neq s \).
- \( d[z] \) is only changed with “edge relaxation” step
  - \( d[z] \leftarrow \min\{d[z], d[u] + w(u,z)\} \)
- Suppose lemma is false. Let \( z \) be first vertex for which lemma is false, i.e. \( d[z] \) is not equal to length of any path from \( s \) to \( z \). Note \( z \neq s \).
  - \( d[z] \) got updated to \( d[u] + w(u,z) \)
  - Then \( d[u] \neq \infty \) and so \( d[u] \) is the length of some path \( P \) from \( s \) to \( u \), since the statement was true for vertex \( u \).
  - There is a path from \( s \) to \( z \) that goes through \( P \) and then through edge \((u,z)\) and the length of this path is \( d[u] + w(u,z) \).
  - Thus after update, \( d[z] \) will hold the length of some path from \( s \) to \( z \), and we have a contradiction.

Thus \( d[v] \) is larger than or equal to the shortest path length from \( s \) to \( v \). Formally we say that \( d[v] \) is an upper bound on the shortest path length from \( s \) to \( v \).
Shortest Paths: Proof of Correctness

Main Theorem: After each iteration for any vertex $v$ in the blue cloud, $d[v]$ is the shortest distance from $s$ to $v$.

Proof: (by contradiction)

- The theorem statement is true after the first iteration, since after first iteration blue cloud only has vertex $s$ and $d[s] = 0$.

- Suppose the theorem statement is false.
  - Let $k$ be the first iteration after which the theorem becomes false.
  - Let $z$ be the vertex inserted into the blue cloud at iteration $k$.
  - Since theorem fails after $z$ is inserted, $d[z]$ > the shortest distance from $s$ to $z$.
    - $d[z]$ can’t be smaller than the shortest distance from $s$ to $z$ according to Lemma 2.

- Consider the situation just before iteration $k$, that is just before vertex $z$ was inserted into the blue cloud.

- Graph connected $\Rightarrow$ there is shortest path $P$ from $s$ to $z$.

- Let $y$ be the first vertex in $P$ which is not in the blue cloud (notice that $y$ could be $z$, and unlike shown in the picture, path $P$ can reenter the blue cloud).

- Let $u$ be vertex immediately before $y$ in $P$ (note that $u$ has to be in the blue cloud and $u$ could be the same vertex as $s$).
Shortest Paths: Proof of Correctness Continued

Let $P_{su}$ be part of path $P$ from $s$ to $u$, and let $P_{yz}$ be part of path $P$ from $y$ to $z$, that is $P$ consists of $P_{su}$, edge $(u, y)$ and $P_{yz}$.

Intuition for the proof:
- $d[z]$ has to be larger than the length of path $P$.
- however, since $z$ is inserted into the blue cloud next, $d[z] \leq d[y]$.
- we will show that $d[y]$ is smaller than or equal to the length of green and red parts of path $P$.
  - due to edge relaxation.
- therefore $d[y]$ is smaller than or equal to the length of path $P$.
  - $d[z] \leq d[y] \leq \text{length of path } P$.

CONTRADICTION
Shortest Paths: Proof of Correctness Continued

- Let $P_{su}$ be part of path $P$ from $s$ to $u$, and let $P_{yz}$ be part of path $P$ from $y$ to $z$, i.e. $P$ consists of $P_{su}$, edge $(u, y)$, and $P_{yz}$.

- $d[y] \leq d[u] + w(u, y)$ since edge $(u, y)$ was relaxed after $u$ got inserted into blue cloud, recall relaxation is $d[y] \leftarrow \min\{d[y], d[u] + w(u, y)\}$.

- $d[u] = \text{length of } P_{su}$
  - $P_{su}$ is the shortest path from $s$ to $u$ by the sub-path lemma 1.
  - $d[u] = \text{length of shortest path from } s \text{ to } u \text{ since } u \text{ is in the blue cloud, and } z \text{ was the first vertex for which theorem statement failed}$.

- Path $P_{su}$ with edge $(u, y)$ is a shortest path from $s$ to $y$ by lemma 1. Length of this path is $d[u] + w(u, y)$.

- Thus $d[y] \leq d[u] + w(u, y) = \text{shortest path length from } s \text{ to } y \leq \text{length of } P$, where the last inequality holds due to non-negativity of edges.

- $d[z] \leq d[y]$ since $z$ is the next vertex chosen to go into the blue cloud.

- Thus $d[z] \leq d[y] \leq \text{length of } P = \text{length of shortest path from } s \text{ to } z$.

- **Contradiction!**, since $d[z]$ was supposed to be bigger than length of $P$. 
Priority

- Frequently elements that we wish to store in a data structure have “priorities”
- Operations should be done in order of the priority
- Examples:
  - Standby passengers for a full flight may have different priorities assigned to them based on their frequent-flyer status, check-in time, etc.
  - Device controller for a shared printer may assign priorities to documents to be printed based on time submitted, size of the document, seniority of the user, etc.
Priority Queue ADT

A priority queue is an abstract data type for storing a collection of prioritized elements, which has 2 main methods:
- Insertion of arbitrary element
- Removal of element of highest priority

In our context, a priority queue stores a collection of entries.

Like for dictionaries, each entry is a pair (key, value):
- The key is the priority associated with the entry
- In our implementation, the smaller key corresponds to higher priority
Priority Queue ADT

- **Main methods of the Priority Queue ADT:**
  - `insert(k, v)`
    - inserts an entry with key `k` and value `v`
  - `removeMin()`
    - removes and returns the entry with smallest key

- **Additional methods**
  - `min()`
    - return, but don’t remove, an entry with smallest key
  - `size()`
  - `isEmpty()`
Dijkstra’s Algorithm

- Invented in 1959
- A priority queue stores the vertices outside the cloud
  - Key: distance
  - Element: vertex
- Locator-based methods
  - \( \text{insert}(k, e) \) returns a locator
  - \( \text{replaceKey}(l, k) \) changes the key of an item
- We store two labels with each vertex:
  - distance \((d[v])\) label
  - locator in priority queue

Algorithm \( \text{DijkstraDistances}(G, s) \)

\[
\begin{align*}
Q & \leftarrow \text{new heap-based priority queue} \\
\text{for all } v & \in G.\text{vertices()} \\
    & \text{if } v = s \\
    & \quad \text{setDistance}(v, 0) \\
    & \text{else} \\
    & \quad \text{setDistance}(v, \infty) \\
    l & \leftarrow Q.\text{insert}(\text{getDistance}(v), v) \\
    & \quad \text{setLocator}(v, l) \\
\text{while } \neg Q.\text{isEmpty()} \\
    u & \leftarrow Q.\text{removeMin()} \\
    & \text{for all } e \in G.\text{incidentEdges}(u) \\
        & \quad \{ \text{relax edge } e \} \\
        z & \leftarrow G.\text{opposite}(u, e) \\
        r & \leftarrow \text{getDistance}(u) + \text{weight}(e) \\
    & \quad \text{if } r < \text{getDistance}(z) \\
        & \quad \quad \text{setDistance}(z, r) \\
        & \quad \quad Q.\text{replaceKey}(\text{getLocator}(z), r)
\end{align*}
\]
Dijkstra’s Algorithm Analysis

- Assume `setDistance` and `setLocator` take $O(1)$ time
- 1st `for` loop takes $O(n \log n)$ time
  - Go over all vertices once, insertion into priority queue is $O(\log n)$
- while loop is executed exactly $n$ times, once for each vertex
  - For one iteration of while loop, we spend time
    - $O(\log n)$ to remove vertex $u$ from priority queue
    - $O(\deg u)$ to look at all incident edges from $u$
    - $O[\deg u (\log n)]$ for `replaceKey`
  - One iteration of while loop takes $O[\deg u (\log n)]$
- total time for while loop is $O(m \log n)$
  - Recall that $\sum_u \deg(u) = 2m$
- Thus total time is $O((n+m) \log n)$

Algorithm `DijkstraDistances(G, s)`

```
Q ← new heap-based priority queue
for all $v \in G.vertices()$
  if $v = s$
    setDistance($v$, 0)
  else
    setDistance($v$, $\infty$)
Q.insert(getDistance($v$), $v$)
setLocator($v$, $l$)

while ¬Q.isEmpty() 
  $u \leftarrow Q.removeMin()$
  for all $e \in G.incidentEdges(u)$
    { relax edge $e$ }
    $z \leftarrow G.opposite(u, e)$
    $r \leftarrow getDistance(u) + weight(e)$
    if $r < getDistance(z)$
      setDistance($z$, $r$)
      Q.replaceKey(getLocator($z$), $r$)
```
Shortest Paths Tree

- We can extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices.
- We store with each vertex a third label:
  - *parent edge* in the shortest path tree
- In the edge relaxation step, we update the parent label.

```java
Algorithm DijkstraShortestPathsTree(G, s)

... ...

for all v ∈ G.vertices()
  ...
  setParent(v, Ø)
  ...

  for all e ∈ G.incidentEdges(u)
    { relax edge e }
    z ← G.opposite(u,e)
    r ← getDistance(u) + weight(e)
    if r < getDistance(z)
      setDistance(z,r)
      setParent(z,u)
      Q.replaceKey(getLocator(z),r)
```

23
Shortest Path Tree

- There is a tree of shortest paths from a start vertex to all the other vertices

Example:
Tree of shortest paths from Providence
Intuitively, Why Dijkstra’s Algorithm Works

Dijkstra’s algorithm is based on the greedy method. It adds vertices by increasing distance.

- Suppose it didn’t find all shortest distances. Let F be the first wrong vertex the algorithm processed.
- When the previous node, D, on the true shortest path was considered, its distance was correct.
- But the edge (D,F) was **relaxed** at that time!
- Thus, so as long as \( d(F) \geq d(D) \), F’s distance cannot be wrong. That is, there is no wrong vertex.
Why It Doesn’t Work for Negative-Weight Edges

- Dijkstra’s algorithm may not work if the graph has negative edges.

  Our proof was based on the fact that a path has larger length than its subpath.

- If negative edges allowed, it’s no longer the case.

*C’s true shortest distance is 2, but it is already in the cloud with *d[C]=3!*
Dijkstra’s Algorithm Summary

- Dijkstra’s algorithm computes the distances of all the vertices from a given start vertex $s$.

- Assumptions:
  - the graph is connected
  - the edges are undirected
    - Easy to extend to directed graph, replace statement
      
      \[
      \text{for all } e \in G.\text{incidentEdges}(u) \quad \text{with statement} \quad \text{for all } e \in G.\text{outgoingEdges}(u)
      \]
  - the edge weights are nonnegative
