CS2210
Data Structures and Algorithms

Lecture 17: Shortest Paths
Outline

- Weighted Graphs
  - Shortest Paths Algorithm (Dijkstra’s)
Weighted Graphs

- Each edge has an associated numerical value, called the weight.
- Edge weights may represent: distances, costs, etc.
- Example: in a flight route graph, edge weight may represent the distance in miles between the endpoint airports.
- Notation $w(u,v)$ denotes the weight of edge $(u,v)$.
Shortest Paths: Problem Statement

- Given a weighted graph and two vertices $u$ and $v$, find a path of minimum total weight between $u$ and $v$
  - length of a path is the sum of its edge weights
- Example
  - shortest path between PVD and HNL
- Applications
  - internet packet routing
  - driving directions
  - etc.
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**
  - 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
Shortest Paths Tree

- Suppose need the shortest path between vertices $u$ and $v$
- Worst case complexity of computing shortest path between $u$ and $v$ is the same as for the shortest path between $u$ and all other vertices in $G$
- Algorithm computes shortest distance between source vertex $s$ and all other vertices
  - tree of shortest paths
Shortest Paths Algorithm: Overview

- Algorithm makes incremental progress
- Marked vertices are called a “blue cloud”
- Maintain property
  - 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 has all vertices
  - shortest paths distances for all the vertices computed
Shortest Paths: Distance Function $d[v]$

- For each vertex $v$, maintain distance function $d[v]$ s.t.
  - If $v$ is in the blue cloud, $d[v]$ is the shortest distance from $s$ to $v$
  - If $v$ is in **not** the blue cloud, $d[v]$ is the distance of the best **blue** path from $s$ to $v$
    - a path is **blue** if it uses only the blue cloud vertices before reaching $v$
Shortest Paths: Initialization

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

- Initially, for all $v$ in the blue cloud, $d[v]$ has the correct shortest distance from $s$ to $v$

- At each iteration, need to figure out
  - which vertex $v$ to inserted next into the blue cloud
  - how do we update distances $d[]$
Shortest Paths Algorithm: Main Part

- Insert into blue cloud vertex $u$ which is outside the blue cloud and has the smallest $d[u]$
- For any vertex $z$ which is not in blue cloud, and is adjacent to $u$, update its distance $d[z]$
  \[
d[z] \leftarrow \min\{d[z], d[u] + w(u,z)\}
\]
- $w(u,z)$ is the weight of edge $(u,z)$

First iteration
Shortest Paths: Edge Relaxation

- The second step is called **edge relaxation**
- After edge relaxation, may have discovered a shorter path from $s$ to $z$
  - the new path goes through $u$

1. Insert into blue cloud vertex $u$ which is outside the blue cloud and has the smallest $d[u]$
2. For any vertex $z$ which is not in blue cloud, and is adjacent to $u$
   $$ d[z] \leftarrow \min\{d[z], d[u] + w(u,z)\} $$

- The new path from $s$ to $z$ has length 60
Example
Example (cont.)
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 $P$, and let $P_{ut}$ be part of $P$ 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 be $P_{su}$ part of $P$ from $s$ to $u$, and $P_{tv}$ part of $P$ from $t$ to $v$
- Combination of $P_{su}$, $Q$ and $P_{tv}$ is a shorter path from $s$ to $v$ than $P$
- Contradiction!
Lemma 2: Upper Bound

Lemma 2: \( d[v] \) is either infinite or length of some path from \( s \) to \( v \)

Proof:
- True after first iteration since
  - \( d[s] = 0 \) and
  - \( d[v] = \infty \) for all other vertices
- \( d[v] \) changes only due to update
  - \( d[v] = d[u] + w(u,v) \)
  - after update, \( d[v] \) is the length of some path that goes from \( s \) to \( u \) and then from \( u \) to \( v \)
- Thus \( d[v] \) larger than or equal to the length of shortest path from \( s \) to \( v \)
Shortest Paths: Proof of Correctness

**Main Theorem**: When vertex \( v \) is placed into the blue cloud, \( d[v] \) is equal to the shortest path length from \( s \) to \( v \)

**Proof**: (by contradiction)

- True after first iteration, since blue cloud has only \( s \) and \( d[s] = 0 \)
- Let \( k \) be the first iteration after which the theorem is false
- Let \( z \) be the vertex inserted into the blue cloud at iteration \( k \)
- Since the theorem fails after \( z \) is inserted, \( d[z] > \text{shortest distance from } s \text{ to } v \)
  - \( d[z] \) cannot be smaller according to lemma 2
- Consider situation just before \( z \) 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
  - \( y \) could be \( z \), and \( P \) could reenter the blue cloud
- Let \( u \) be vertex immediately before \( y \) in \( P \)
  - \( u \) has to be in the blue cloud and \( u \) could be \( s \)
Proof of Correctness Continued

- Let $P_{su}$ be part of $P$ from $s$ to $u$, and $P_{yz}$ part of $P$ from $y$ to $z$
  - $P$ 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
  - relaxation: $d[y] \leftarrow \min\{d[y], d[u] + w(u, y)\}$
- $d[u] = \text{length of } P_{su}$
  - $P_{su}$ is the shortest path from $s$ to $u$ by lemma 1
  - $d[u] = \text{length of shortest path from } s \text{ to } u \text{ since } u \text{ is in the blue cloud}$
  - and $z$ is the first vertex for which theorem fails
- $P_{su}$ 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 $d[y] \leq d[u] + w(u, y) = \text{shortest path length from } s \text{ to } y \leq \text{length of } P$
  - 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 $d[z] \leq d[y] \leq \text{length of } P = \text{length of shortest path from } s \text{ to } z$
- but since $d[z]$ was supposed to be bigger than length of $P$
- Contradiction!
Shortest Paths: Dijkstra’s Algorithm

- Invented in 1959
- **Adaptable** priority queue $Q$ stores vertices outside blue cloud
  - entries are aware of their location in $Q$
  - each entry has 3 fields
    - Key: distance
    - Value: vertex
    - Location:
      - Position of the entry in the heap
      - Needs to be updated when performing upheap/downheap
- Locator-based methods
  - $\text{insert}(k,v)$ returns new entry $e$
    - $e.l$ is the location of the new entry in the priority queue
  - $\text{replaceKey}(e,k)$ changes key of entry $e$
- Store with each vertex $v$
  - distance $d[v]$
  - locator in priority queue

---

**Algorithm DijkstraDistances($G,s$)**

1. $Q \leftarrow$ new heap-based priority queue
2. for all $v \in G.\text{vertices}()$
   - if $v = s$
     - $v.\text{setDistance}(0)$
   - else
     - $v.\text{setDistance}(\infty)$
3. $l \leftarrow Q.\text{insert}(v.\text{getDistance}(v),v)$
4. $v.\text{setLocator}(l)$
5. while $\neg Q.\text{isEmpty}()$
6.   - $u \leftarrow Q.\text{removeMin}()$
7.   - for all $e \in G.\text{incidentEdges}(u)$
      - //relax edge $e$
      - $z \leftarrow G.\text{opposite}(u,e)$
      - $r \leftarrow u.\text{getDistance}()+e.\text{weight}()$
      - if $r < z.\text{getDistance}()$
        - $z.\text{setDistance}(r)$
        - $Q.\text{replaceKey}(\text{getLocator}(z),r)$
Dijkstra’s Algorithm Analysis

- Assume `setDistance` and `setLocator` is $O(1)$.
- First `for` loop takes $O(n \log n)$ time:
  - Insert each vertex into priority queue, one insertion is $O(\log n)$.
- `while` loop is executed exactly $n$ times, once for each vertex:
  - For one iteration of `while` loop:
    - $O(\log n)$ 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 $O[(\deg u)(\log n)]$.
- Time for `while` loop is $O(m \log n)$:
  - Recall $\sum u \deg(u) = 2m$.
- Total time is $O((n+m) \log n)$.

```python
Q ← new heap-based priority queue
for all v ∈ G.vertices():
    if v = s
        v.setDistance(0)
    else
        v.setDistance(∞)
l ← Q.insert(v.getDistance(v), v)
v.setLocator(l)
while ¬Q.isEmpty():
    u ← Q.removeMin()
    for all e ∈ G.incidentEdges(u):
        //relax edge e
        z ← G.opposite(u, e)
        r ← u.getDistance() + e.weight()
        if r < z.getDistance():
            z.setDistance(r)
            Q.replaceKey(getLocator(z), r)
```
Shortest Paths Tree

- Extend Dijkstra’s algorithm to return a tree of shortest paths from the start vertex to all other vertices
- Store with each vertex the parent
- during edge relaxation step, update the parent

Algorithm DijkstraShortestPathsTree(G,s)

```
Q ← new heap-based priority queue
for all v∈G.vertices()
    if v = s
        v.setDistance(0)
    else
        v.setDistance(∞)

l ← Q.insert(v.getDistance(v),v)
v.setLocator(l)

while ¬Q.isEmpty()
    u ← Q.removeMin()
    for all e∈G.incidentEdges(u)
        //relax edge e
        z ← G.opposite(u,e)
        r ← u.getDistance()+e.weight()
        if r < z.getDistance()
            z.setDistance(r)
            z.setParent(u)
    Q.replaceKey(getLocator(z),r)
```
Shortest Path Tree

- Tree of shortest paths from a start vertex to all other vertices
- Example: Tree of shortest paths from Providence
Dijkstra’s algorithm may fail if the graph has negative edges

- correctness proof was based on the fact that length of a path is larger than or equal than length of any of its subpath
- If negative edges allowed, it is not longer the case

shortest distance from s to C 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 and undirected
  - for directed graph, replace
    
    $$\text{for all } e \in G.\text{incidentEdges}(u)$$
    
    with
    
    $$\text{for all } e \in G.\text{outgoingEdges}(u)$$
  
  - edge weights are nonnegative