All-pairs shortest path

Overview

$G=(V,E)$ is a weighted directed graph with no negative cycles
- Weights may be given in an adjacency matrix
- Output is shortest paths as a matrix, where $D[i, j] = \text{minimum distance from } i \text{ to } j$.

Approaches
- Repeated applications of Single-source shortest-path
  - Bellman-Ford: $\Theta(V^2E)$
  - Dijkstra: $\Theta(V\log V + VE)$—Requires no negative-weight edges
- Dynamic programming
  - Based on adding last edge
  - Similar to matrix multiplication problem $\Theta(V^3\log V)$
  - Based on using/not using a particular vertex
  - Floyd-Warshall $\Theta(V^3)$
  - Johnson: $\Theta(V^3\log V + VE)$—can detect negative-weight cycles

Dynamic Programming

Define $OPT(i, j, m)$ as the minimum path from $i$ to $j$ using at most $m$ edges
- $OPT(i, j, 0) = \infty$ if $i \neq j$, 0 if $i = j$
- $OPT(i, j, m) = \min(OPT(i, j, m-1), \min_{1 \leq k \leq V} OPT(i, k, m-1) + w_{kj})$
  - Minimal distance in $\leq m$ edges is either the same as $m-1$ or involves minimal distance to some vertex $k$ and then one extra edge.
- If there are no negative-weight cycles, $OPT(i, j, n-1)$ is minimum path from $i$ to $j$

Implementation

Augment matrix $L$ with shortest paths so far by using paths in matrix $W$:
- Extend-shortest-paths($L$, $W$)
  - $L'= \text{new } n \times n \text{ matrix initialized to } \infty$
  - for $i = 1 \to n$
    - for $j = 1 \to n$
      - for $k = 1 \to n$
        - $L'[i, j] = \min(L[i, j], L[i, k] + W[k, j])$
  - Total time of this routine is $\Theta(n^3)$
- Run it $n-1$ times to calculate shortest paths:
  - $L^n = \text{Extend-shortest-paths}(L^{n-1}, W)$
- Overall solution: $\Theta(V^4)$
Better implementation

Observation: We don’t need all of $L^1, L^2, \text{etc.}$ All we need is $L^{n-1}$

Observation 2: If there are no negative cycles, $L^{n-1} = L^n = L^{n+1} = \ldots$

Compute $L^1, L^2, L^4, L^8, \ldots$, $L^{\lceil \log(n-1) \rceil}$

* $L^{2m} = \text{Extend-shortest-paths}(L^m, L^n)$
* Execute $\Theta(\log n)$ times

Total running time: $\Theta(V^3 \log V)$

How to detect negative-weight cycles?

Floyd-Warshall

Define $\text{OPT}(i, j, k)$ as the minimum path from $i$ to $j$ using intermediate vertices only from $V_1$ to $V_k$

* $\text{OPT}(i, j, 0) = W[i, j]$

* $\text{OPT}(i, j, k) = \min(\text{OPT}(i, j, k-1), \text{OPT}(i, k, k-1) + \text{OPT}(k, j, k-1))$

- Either don’t use vertex $k$, or use vertex $k$ and find minimal path from $i$ to $k$ and $k$ to $j$ using vertices not including $V_k$.

* Algorithm:
  - Let $D[i, j, 0] = W[i, j]$
    - for $i = 1$ to $n$
    - for $j = 1$ to $n$
    - for $k = 1$ to $n$
      - $D(i, j, k) = \min(D(i, j, k-1), D(i, k, k-1) + D(k, j, k-1))$

* Overall time: $\Theta(V^3)$

* Overall space: $\Theta(V^2)$ (only need $D(*, *, k-1)$ to calculate $D(*, *, k)$)

How to detect negative-weight cycles?

Johnson’s algorithm

Use Bellman-Ford—$O(V^2 E)$

* to detect negative-weight cycles
* to determine distance from some $S$ to each vertex

Create new graph $G’$ (with one extra vertex) and new weights, $W’$,

where all weights are positive, but minimum paths aren’t affected—$O(V+E)$

* Weights may change, but $p$ is a shortest path using $w$ iff $p$ is a shortest path using $W$

Use Dijkstra’s algorithm for each vertex using $W’=O(V^2 \log V + VE)$

* To determine minimum paths for each pair of vertices

Postprocess path lengths to convert from $W’$ to $W=O(V^2)$

Total time: $O(V^2(E+\log V) + VE)$

Create new weights

Let $G’ = (V’, E’)$

* $V’ = V \cup \{s\}$
* $E’ = E \cup \{(s, v): v \in V\}$
* $w(s, v) = 0$ for all $v \in V$

Calculate $d(s, v)$ for all $v \in V$ (Bellman-Ford)

Let $W'[i, j] = W[i, j] + d(s, i) - d(s, j)$

* $d(s, i) = \text{minimal distance from node } s \text{ to } i$

Claim 1: $W’$ has no negative edge weights

* $d(s, j) = d(s, i) + w(i, j)$
* $w(i, j) + d(s, i) - d(s, j) \geq 0$
* $w(i, j) > 0$

Claim 2: paths are unchanged

* For any path from $i$ to $j$, $p$, sum of weighted edges in $P$ using $W’$, $w'(p) = w(p) + d(s, i) - d(s, j)$

Thus, if $w(p)$ is minimal weight to connect $i$ and $j$, $w'(p)$ is minimal
Final Johnson algorithm

Johnson(G)
Compute G’ (as stated previously)
If Bellman-Ford(G’, w, s) = FALSE then
  return nil
else
  foreach vertex v in V
    h(v) = value of d(s, v) calculated by Bellman-Ford
  foreach edge e in E
    w’(u, v) = w(u, v) + h(u) - h(v)
  foreach vertex v in V
    d’(u, v) = Dijkstra(G, w’, u)
  foreach vertex v in V
    d(u, v) = d’(u, v) + h(v) - h(u)
  return d
endif

Network Flow

Definition: Flow network G=(V, E) where
- each edge has capacity c(u, v) >= 0
- two distinguished nodes: source, s, and sink, t
  - No edges into source
  - No edges out of sink
- every vertex is on some path from s to t
Flow:
- Assign flows, f, to each edge such that (slightly different from CLRS):
  - 0 <= f(u, v) <= c(u, v) Capacity constraint
  - for all u in v (except s, t) sum, f(u, v) = sum, f(v, u) Flow conservation
- Total flow: f(v) = sum, f(s,v) Total flow out of s
Models
- Material flowing through edges
- Real world has multiple sources or sinks. Can we model?

Cut

Definition: an s-t cut is partition (A, B) of V where s in A and t in B
Definition: capacity of a cut is sum_u in A, v in B c(u, b)
- Capacity of edges crossing from A to B
- Is not net capacity: doesn’t include edges from B to A
Problems

Min-cut problem: Find an s-t cut of a flow network G with minimum capacity

Max-flow problem: Find a flow, f, of maximum value in a flow network G

Relationship between Flows and Cuts

Given a flow f, the net flow, f(A, B), across a cut (A, B) is

Proof by induction on |A|
- Base case: A = {s}. True by definition of value of f
- Inductive step. Assume true for all cuts (A, B) where |A| < k. Show true for any cut (A', B') where |A'| = k
  - A' = A U {v} for some v <> s or t
  - By inductive hypothesis, f(A, B) = v(f)
  - f(A', B') = v(f) + sum_{u in V} f(v, u) - sum_{u in V} f(u, v) = v(f)

Relationship between flows and cuts

For any flow, f, and cut (A, B) v(f) <= c(A, B)
- The value of a flow is limited by the capacity of any cut

v(f) = f(A, B)
  = sum_{u in A, v in B} f(u, v) - sum_{u in A, v in B} f(v, u)
  <= sum_{u in A, v in B} f(u, v)
  <= sum_{u in A, v in B} c(u, v)
  = c(A, B)

Corollary: The min cut is a bound on the max flow
- A proof of max flow is demonstrating a min cut whose capacity = v(f)

Ford-Fulkerson method

Ford-Fulkerson-method(G, s, t)
Set f(e) = 0 for all e in E
while there exists an augmenting path p
  augment flow f along p
return v(f)

Simple greedy algorithm can fail to achieve maximum flow:
Residual graph

Need to deal with canceling flow

Create a graph $G_f$ where edges represent possible flow in $G$
- For each edge $e=(u,v)$ in $E$ where $f(e) < c(e)$, create $e'=(u,v)$ with capacity $c(e) - f(e)$
- For each edge $e=(u,v)$ in $E$ where $f(e) > 0$, create $e'=(v,u)$ with capacity $f(e)$

Example:

An augmenting path $p$ in $G_f$ corresponds to an augmenting path in $G$
- An augmenting path $p'$ is a path from $s$ to $t$ with $c(e') > 0$ for each edge $e'$ in $p'$, and with a flow $f' = \min$ capacity of the edges in $p'$
- For $e'=(u,v)$ in $p'$, if $c(e') > 0$, one of two possibilities:
  - There is an edge $e=(u,v)$ in $G$, and $f(e) + f' \leq c(e)$. So, $f(e) += f'$
  - There is an edge $e=(v,u)$ in $G$ and $f(e) \geq f$. So, $f(e) -= f'$
- We continue to maintain conservation and capacity constraints
- $f$ is now increased by $f'$

Max-flow Min-cut theorem

For a flow network $G$, with a flow $f$, the following are equivalent:
- There exists a minimal cut $(A, B)$ with $c(A, B) = v(f)$
- $f$ is a maximum flow
- There is no augmenting path in the residual graph, $G_f$

Proof by contrapositive: ~iii->~ii
- If there is an augmenting path in the residual graph of $G$, $f$ can be increased. Thus, $f$ is not a maximum flow.

Integrality

If all capacities are integers, flow calculated is always an integer
- If flows are integral, residual graph contains integral values
- If residual graph contains integral values, augmented flow is integral

If all capacities are integers, there exists a max flow where flow on each edge is an integer
Runtime

Calculating residual graph: \( O(E) \)
Finding augmenting path: \( O(E) \)
Updating flow in \( G \): \( O(E) \)

**How many times do we need to try and augment path?**

- Each augmentation increases flow by at least 1
- Need to augment no more than max flow, \( v(f^*) \leq VC \)
  - \( v(f^*) \) is maximum flow
  - \( C \) is maximum capacity

Total runtime: \( O(EVC) \)  

**Edmonds-Karp**

Find shortest augmenting path!
Use Breadth-first search

Total runtime: \( O(VE^2) \)

Idea of proof:

- With each augmentation (augmenting from \( f \) to \( f' \)), the distance (unit-edges) from \( s \) to any node \( V \) in the residual graph never decreases
  - By contradiction: take shortest path that did decrease. It ends in \((u,v)\), not in \( E_f \). Flow increased from \( v \) to \( u \). \( d(s, v) = d(s, u) - 1 \leq d'(s, u) - 1 \leq d'(s, v) - 2 \)
  - A critical edge \((u, v)\) in augmentation is min edge in path. Critical edge isn’t present in next augmentation. Can’t reappear until flow is decreased ((\( v, u \)) appears on augmenting path). Distance of \( u \) from source must increase by \( \geq 2 \) by that time. \((u, v)\) can become critical at most \((V-2)/2\) times. Total number of augmentations: \( O(VE) \)

Push-relabel algorithms

Preflow temporarily violates conservation

- Store extra flow in vertex reservoir
- Relaxed conservation:
  - \( e(u) = \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) \geq 0 \) \((u \neq s)\)
  - Call this excess flow
- Every vertex and reservoir has a height.
  - We only push flow from a higher vertex to a lower vertex
  - \( s \) is at height \( |V| \), \( t \) is at height 0. Other vertices start at 0 and increase over time.

**Push step** for \( u \): Can be done if: \( u \) is overflowing and there exists a lower neighbor in the residual graph
- Send as much flow as possible to a lower neighbor, \( v \)
- Store excess flow in reservoir at \( v \) (overflowing \( v \))

**Relabel step**: Can be done if: \( u \) is overflowing and all neighbors (must be at least one) in residual graph are not at a lower height
- Set height to \( 1 + \) min height of neighbor in residual graph

Generic Push-relabel Algorithm

Algorithm: While a push or relabel is possible, do it

At end of algorithm:

- There are no overflowing vertices
- Therefore, the preflow is actually a flow
- There is no path from \( s \) to \( t \) in residual graph
  - Need lemma about height functions to show this

Runtime (hand-wave) \( O(V^2E) \)

- Lemma: max height of any vertex: \( 2V-1 \)
- Relabel operations: \( <2V \) per vertex = \( <2V^2 \) overall
- Saturating pushes: \( <2VE \) overall
  - Height of \( u \) or \( v \) must increase by at least 2 between successive saturating pushes between \( u \) and \( v \). \( <2V \) saturating pushes/edge.
- Non-saturating pushes: \( 4V^2(E+V) \)
  - Potential function \( \phi = \sum_{\text{non-saturated nodes}} v \cdot \text{height}(v) \)
  - Non-saturating pushes decrease \( \phi \) by at least 1
  - Saturating pushes increase \( \phi \) by less than \( 2V \)
  - Relabels increase \( \phi \) by less than \( 2V \)
Push-label algorithm

Given a data structure with cost of (Exercise 26.4-1):
- $O(1)$ per Relabel
- $O(1)$ per Push
- $O(1)$ to pick Push/Relabel

Push-label can run in $O(V^2E)$ time

Other implementation exist that apply Relabel/Push in better ways
- Relabel-to-Front
  - Keeps list of vertices to be operated on.
  - Does push/Relabels on vertex at the front of the list.
  - When a vertex is relabeled, it is moved to the front of the list.
  - Operates in $O(V^3)$ time