Lecture20 & 21_AllPairsSP

Wednesday, October 21, 2020 10:47 AM

Review:

Linear programing: generic way to formulate a large class of optimalization problems.

Fsibility version of the problem: drop the optimalizaiton functuion, look at only the constrains, which are bunch of linear inequalities.

Special version of the problem: system of diffiriences. Matrix formulation, every row have 1 and -1 and all other coefficient is zero. Satisfies some unknow variables with subject to a list of different constrains.*

-> equal to graph theory problem: shortest path problem

From any source to any destination

Dynamic programing approach

Shortest paths

Single-source shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm: O(E + V lg V)
- General
  - Bellman-Ford algorithm: O(VE)
- DAG
  - One pass of Bellman-Ford: O(V + E) (bfs)

All-pairs shortest paths

- Nonnegative edge weights
  - Dijkstra's algorithm |V| times: O(VE + V 2 lg V)
    "Use n times", n = number of vertex
- General
  - Three algorithms today.

Problem:

Input: Digraph G = (V, E), where V = {1, 2, ..., n}, with edge-weight function w : E → **R**.

Output: n × n matrix of shortest-path lengths δ(i, j) for all i, j ∈ V.

IDEA:
- Run Bellman-Ford once from each vertex.
- Time = $O(V^2E)$.
- Dense graph ($\Theta(n^2)$ edges) $\Rightarrow \Theta(n^4)$ time in the worst case.

Good first try!

Dynamic programming

Consider the $n \times n$ weighted adjacency matrix $A = (a_{ij})$, where $a_{ij} = w(i, j)$ or ∞, and define

$d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges.

Claim: We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \end{cases}$$

For the vertex itself: 0

All other things: infinity

i to j that uses at most m edges.

Claim: We have

$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

For the vertex itself: 0
All other things: infinity

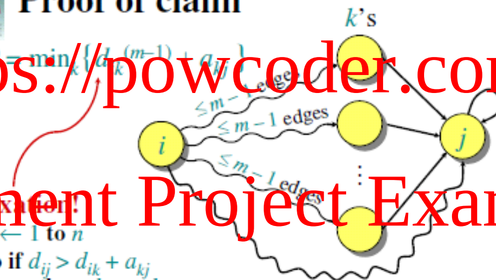
and for $m = 1, 2, \ldots, n-1$,

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$

- Based on how many edges do we use in our solution
- A single path graph could have at most n - 1 edges: limit

Proof of claim

$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$



Relaxation:
for $k \leftarrow 1$ **to** n
  do if $d_{ij} > d_{ik} + a_{kj}$
    then $d_{ij} \leftarrow d_{ik} + a_{kj}$

Note: No negative-weight cycles implies
$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \cdots$

1. To go from i to j, using at most m edges. We must gone somewhere else using at most m - 1 edges. Then use 1 more edge to arrive at j.

Matrix multiplication

Compute $C = A \cdot B$, where C, A, and B are $n \times n$ matrices:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Time $= \Theta(n^3)$ using the standard algorithm.

What if we map "+" → "min" and "·" → "+"?

$$c_{ij} = \min_k \{ a_{ik} + b_{kj} \}.$$

Thus, $D^{(m)} = D^{(m-1)}$ "×" A.

$$\text{Identity matrix} = I = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 0 \end{pmatrix} = D^0 = (d_{ij}^{(0)}).$$

From I to I, the cost is 0
Everybody else we can't arrive

- Instead of making the summation of product using the product of summations
- We are doing n matrix multiplications, Running time: $O(n * n^3)$

The $(\min, +)$ multiplication is ***associative***, and with the real numbers, it forms an algebraic structure called a ***closed semiring***.

Consequently, we can compute

$$\begin{aligned} D^{(1)} &= D^{(0)} \cdot A = A^1 \\ D^{(2)} &= D^{(1)} \cdot A = A^2 \\ &\vdots \\ D^{(n-1)} &= D^{(n-2)} \cdot A = A^{n-1}, \end{aligned}$$

yielding $D^{(n-1)} = (\delta(i, j))$.

Time $= \Theta(n \cdot n^3) = \Theta(n^4)$. No better than $n \times$ B-F.

* not better than repeatedly perform B-F algorithm

Improved matrix multiplication algorithm

Repeated squaring: $A^{2k} = A^k \times A^k$.

Compute $\underbrace{A^2, A^4, \ldots, A^{2^{\lceil \lg(n-1) \rceil}}}_{O(\lg n) \text{ squarings}}$.

Note: $A^{n-1} = A^n = A^{n+1} = \cdots$.

Time $= \Theta(n^3 \lg n)$.

To detect negative-weight cycles, check the diagonal for negative values in $O(n)$ additional time.

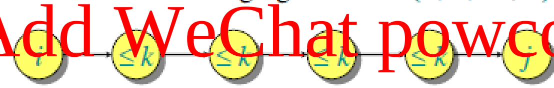
- Negative-weight cycle: starts at I, ends at i.

More improvement!

Floyd-Warshall algorithm

Faster dynamic programming

Define $c_{ij}^{(k)}$ = weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.



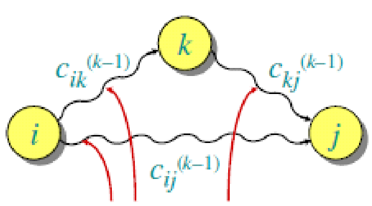
Thus, $\delta(i, j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

- Use a different definition
  Before based on the number of edges.
  Now focus on the vertices: we label the vertices and use the first i vertices (1 to k in i to j)

- Base case: $c_{ij}^{(0)}$: is there any edge that go from i to j?(0 other vertices are allowed to use)
- General case: $c_{ij}^{(n)}$
- Critical vertex: there is an unique vertex that labeled k.

Floyd-Warshall recurrence

$$c_{ij}^{(k)} = \min\{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$



intermediate vertices in $\{1, 2, \ldots, k-1\}$

- To go from I to j, the shortest path either go through vertex k, or it doesn't. (2 options)
  - If it doesn't, $c_{ij}^{(k)} = c_{ij}^{(k-1)}$
  - If it does, $c_{ij}^{(k)} = c_{ik}^{(k-1)} + c_{kj}^{(k-1)}$
  - Checking both of them, and pick minimum

$$
\begin{array}{l}
\textbf{for } k \leftarrow 1 \textbf{ to } n \\
\quad \textbf{do for } i \leftarrow 1 \textbf{ to } n \\
\qquad \textbf{do for } j \leftarrow 1 \textbf{ to } n \\
\qquad\quad \left.\begin{array}{l}\textbf{do if } c_{ij} > c_{ik} + c_{kj} \\ \quad \textbf{then } c_{ij} \leftarrow c_{ik} + c_{kj}\end{array}\right\} \textit{relaxation}
\end{array}
$$

Notes:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in $\Theta(n^3)$ time.
- Simple to code.
- Efficient in practice.

Transitive closure of a directed graph

Number of pair for two vertices -> does these exist a path between two vertices?

Compute $t_{ij} = \begin{cases} 1 & \text{if there exists a path from } i \text{ to } j, \\ 0 & \text{otherwise.} \end{cases}$

IDEA: Use Floyd-Warshall, but with $(\vee, \wedge)$ instead of $(\min, +)$:

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}).$$

Time $= \Theta(n^3)$.

Graph reweighting

- ➢ Is there a way to adjust weight <u>without changing the problem</u> and in the meantime eliminate negative edge weight for the graph

Theorem. Given a function $h : V \to \mathbb{R}$, *reweight* each edge $(u, v) \in E$ by $w_h(u, v) = w(u, v) + h(u) - h(v)$. Then, for any two vertices, all paths between them are reweighted by the same amount.

Proof. Let $p = v_1 \to v_2 \to \cdots \to v_k$ be a path in G. We have

$$
\begin{aligned}
w_h(p) &= \sum_{i=1}^{k-1} w_h(v_i, v_{i+1}) \\
&= \sum_{i=1}^{k-1} \left(w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1})\right) \\
&= \sum_{i=1}^{k-1} w(v_i, v_{i+1}) + h(v_1) - h(v_k) \\
&= w(p) + h(v_1) - h(v_k). \quad \square
\end{aligned}
$$

Same amount!

- Graph reweighting function h
- For every edge u and v, we have an edge weight w(u,v). We have a function h: $w_h(u, v) = w(u, v) + h(u) - h(v)$
  - How can be pick h values so the result is non-negative
  - ➢ Update only the edges between two vertices
  - ➢ h(u) - h(v): the difference between the value

Corollary. $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$. $\square$

IDEA: Find a function $h : V \to \mathbb{R}$ such that $w_h(u, v) \geq 0$ for all $(u, v) \in E$. Then, run Dijkstra's algorithm from each vertex on the reweighted graph.

NOTE: $w_h(u, v) \geq 0$ iff $h(v) - h(u) \leq w(u, v)$.

Johnson's algorithm

Based on graph reweighting

1. Find a function $h : V \to \mathbb{R}$ such that $w_h(u, v) \ge 0$ for all $(u, v) \in E$ by using Bellman-Ford to solve the difference constraints $h(v) - h(u) \le w(u, v)$, or determine that a negative-weight cycle exists.
   - Time $= O(VE)$.
2. Run Dijkstra's algorithm using w_h from each vertex $u \in V$ to compute $\delta_h(u, v)$ for all $v \in V$.
   - Time $= O(VE + V^2 \lg V)$.
3. For each $(u, v) \in V \times V$, compute
   $$\delta(u, v) = \delta_h(u, v) - h(u) + h(v) .$$
   - Time $= O(V^2)$.

Total time $= O(VE + V^2 \lg V)$.

1. Try to find a function h (find the shortest path from a dummy source, Bellman-Ford algorithm)

2. Compute the shortest path for the modified function(same shortest path for the original function) [critical step, bottom neck]
3. Take the weight back

- If E is not n^2 this algorithm is better than the Floyd-Warshall

- Both side of the running time may dominate depends on the circumstance