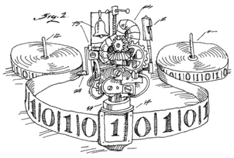
EECS 376: Foundations of Computer Science

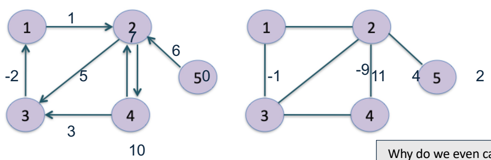
Lecture 06 - Dynamic Programming 3



Agenda

- Shortest paths: Dynamic Programming on Graphs
  - Single-source Shortest Paths (SSSP)
    - The Bellman-Ford Algorithm
    - The Path-Doubling Algorithm
  - All-Pairs Shortest Paths (APSP)
    - The Floyd-Warshall algorithm

Directed and undirected graphs



Distance from s to t, denoted **dist**(s,t): *minimum*, over all *paths* P from s to t, of the *sum of edge weights* in P.

Why do we even care about negative weights?

Notation: V = vertex set, E = edge set, n = |V|, m = |E|.

The shortest-path problems we'll consider

Input: Weighted directed graph. Weights can be negative, but assume no negative-weight cycles (why?).

(1) 0 1 2 0 ∞

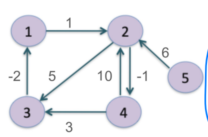
某点到其他所有点

Single-Source Shortest Paths (SSSP): Given a "source" vertex **s**, find a shortest path from **s** to every vertex **t**.

所有点之间

All-Pairs Shortest Paths (APSP): For every pair **s,t** of vertices, find a shortest path from **s** to **t**.

∞表示无法到达



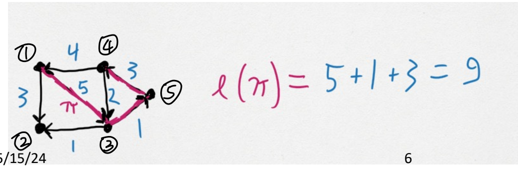
	1	2	3	4	5
1	0	1	2	0	∞
2	0	0	2	-1	∞
3	-2	-1	0	-2	∞
4	1	2	3	0	∞
5	6	6	8	5	0

到自身：总是0.

What about single-pair shortest path?

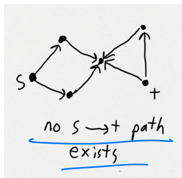
Shortest Paths

- Input:
  - a directed graph $G = (V, E)$
  - length function $\ell : E \to \mathbb{R}$
- Notations:
  - For a path π, **its length** $\ell(\pi)$ is the sum of edge lengths along the path.
  - **Distance from s to t, $dist_G(s,t)$,** is the shortest length of any path from s to t

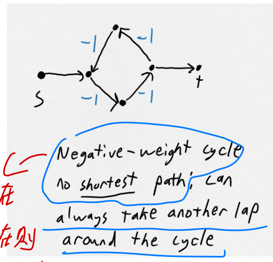


Is dist(s,t) well-defined?

- ***Two reasons*** there could be ***no shortest path***…



$dist(s,t) = \infty$



不能存在
如果存在则
可以一直cycle
到−∞

Usually just assume this doesn't happen

①

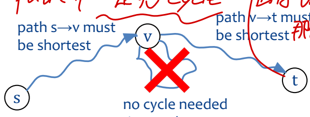
Two Key Observations

如果sh(s,t)经过v，那么sh(s,t)=sh(s,v)+sh(v,t)

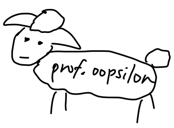
Principle of Optimality

1. If a shortest path from s to t goes through vertex v, then it must be a **shortest path from s to v,** then a **shortest path from v to t.**
2. Since there is no negative-weight cycle in the graph, there is a shortest path from s to t with **no cycle** in it.

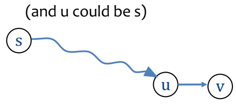
② shortest path 中一定无 cycle （因为既然assume无neg cycle 那么cycle只会加dist.）



Consider the following proposed as a recurrence for SSSP



In the shortest s⟶v path, u is the last vertex before v (and u could be s)



- Recurrence: **dist(s,v)** = **min**$_{(u,v)\in E}${**dist(s,u)** + **ℓ(u,v)**}
- Base case: **dist(s,s)** = 0

✗ not a recurrence

因为dist(s,u) ∀(u,v)∈E 并不是dist(s,v)的subproblem.

Where:

- **ℓ(y,z)** is the weight (or "length") of the edge y⟶z
- **dist(y,z)** is the distance from y to z

This equation is technically correct, but it's not really a recurrence and it doesn't work for DP.

(先算哪个?)

The DP Recipe

you are here

1. Derive a recurrence for the 'value version' of the problem
2. Size of table: How many dimensions? Range of each dimension?
3. What are the base case(s)?
4. To fill in a cell, which other cells need to be filled already? In which order do I fill the table?
5. Which cell(s) contain the final answer?
6. Running time = (size of table) · (time to fill each entry)
7. To reconstruct a solution (instead of just its value) follow "breadcrumbs" from final answer to base case

1.

Bellman-Ford
for single source shortest paths

Bellman-Ford algorithm

- The **Bellman-Ford algorithm** is an algorithm that computes the shortest paths from a single source vertex to each of the other vertices in a weighted digraph.
- It is slower than Dijkstra's algorithm for the same problem, but more versatile, as it can handle graphs in which some of the edge weights are negative numbers.

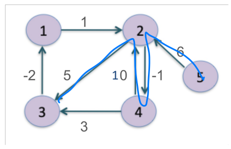
Bellman-Ford algorithm

- Input graph $G = (V, E)$ and source node s
  - n nodes, m edges
  - Assume: no negative-weight cycles (will remove this soon),
  - Algorithm will have $O(mn)$ runtime
- Key Idea: **Dynamic Programming**

Definition

- $dist^{(i)}(s,t)$ = "i-hop distance from s to t" shortest length of an $s \to t$ path using **exactly i edges**, or ∞ if there's no such path
- $dist^{(\le i)}(s,t)$ = "at-most-i-hop distance from s to t" shortest length of an $s \to t$ path using **at most i edges**

Examples



What is…

$dist^{(0)}(5,3)$? ∞ (no)

$dist^{(1)}(5,3)$? ∞ (no)

$dist^{(2)}(5,3)$? 11 (⑤②③)

$dist^{(3)}(5,3)$? 8 (⑤②④③)

$dist^{(4)}(5,3)$? 20 (⑤②④②③)

(显然. 因为无 neg-length ⇒ 不可能往回 ⇒ 最多 traverse every node.

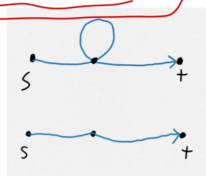
Lemma:
In n-node graph without neg-length cycles,
$dist^{(\le n-1)}(s,t) = dist(s,t)$

Proof Sketch:

A path with n hops hits $n+1$ nodes, so it repeats a node, so it contains a cycle.

This cycle has nonnegative length.

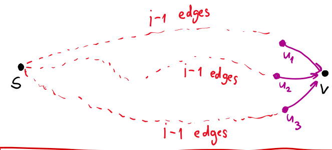
This cycle can be removed from the path without increasing its length.



So… we only need to compute $dist^{(\le n-1)}(s,t)$.
Can we do this recursively?

Recursive Formulation

- **Pause and think**:
- How do you compute $dist^{(i)}(s,v)$ from $dist^{(i-1)}(s,\cdot)$?



recursion: (显然

$$dist^{(i)}(s,v) = \min_{(u,v)\in E} dist^{(i-1)}(s,u) + \ell(u,v)$$

- Why ?
  - "i-hop shortest path = $(i-1)$-hop shortest path + the last edge"
  - Take the best one among all in-coming neighbors to v

base case: $dist^{(0)}(s,v) = \begin{cases} 0, & s=v \\ \infty, & \text{otherwise.} \end{cases}$

- Bellman-Ford(G, s) assume no neg-weight cycles in G
  - Initialize array $dist$, indexed by i,t index entries by $dist^{(i)}(s,t)$ $dist_s[n][m]$
  - All entries initially ∞ $dist$ is like table in previous lectures
  - $dist^{(0)}(s,s) \leftarrow 0$ base case ($dist^{(0)}(s,\neg s) = \infty$)
  - For $i = 1, \dots, n-1$: O(n) loops
    - For each vertex v, $\sum_v \deg(v) = O(m)$ time/loop
      - $dist^{(i)}(s,v) \leftarrow \min_{(u,v)\in E} dist^{(i-1)}(s,u) + \ell(u,v)$
  - Return $dist^{(\le n-1)}(s,\cdot) = \min_{i \le n-1} dist^{(i)}(s,\cdot)$ return subarray

$d^{(i)}(s,v)$

$l(u,v)=\infty$ 因为 no such edge

	$i=0$	$i=1$	$i=2$	$i=3$	$\cdots$ ($i=n-1$)
$v=s$	0	$0+\infty=\infty$	∞	∞	$\cdots$
$v=v_1$	∞	$0+l(s,v_1)$	$\min+l(s,v_1)$	$\min+l(s,v_1)$	$\cdots$
$v=v_2$	∞	$0+l(s,v_2)$	$\min+l(s,v_2)$	$\min+l(s,v_2)$	$\cdots$
$v=v_3$	∞	$0+l(s,v_3)$	$\min+l(s,v_3)$	$\min+l(s,v_3)$	$\cdots$
$v=v_4$	∞	$0+l(s,v_4)$	$\min+l(s,v_4)$	$\min+l(s,v_4)$	$\cdots$
$\vdots$	$\vdots$	(if exists (s,v), otherwise ∞)	$\vdots$	$\vdots$	

$d(s,v_k) = \min(\text{row}[v=v_k])$ 即 $dist^{\le n-1}(s,v_k)$

2. Detecting Neg-Length Cycles

- Slightly harder problem:
  - Input graph **G**, source node **s**
  - If **G** has no negative-length cycles, output all distances $dist(s,t)$
  - If **G** has a negative-length cycle, output "oh no a negative length cycle"
- **Observe:**
  - If v is in a negative-length cycle, then $dist^{(\le n)}(s,v) < dist^{(\le n-1)}(s,v)$
  - Bellman-ford correctly computes $dist^{(i)}(s,v)$ for any i

Challenge: If neg cycle exists, then for some v, $dist^{(\le n)}(s,v) < dist^{(\le n-1)}(s,v)$

- Bellman-Ford(G,s)
  - Initialize array $dist$, indexed by i,t — index entries by $dist^{(i)}(s,t)$
  - All entries initially ∞
  - $dist^{(0)}(s,s) \leftarrow 0$ — base case
  - For $i=1,\dots,n$: O(n) loops
    - For each vertex v, O(m) time/loop
      - $dist^{(i)}(s,v) \leftarrow \min_{(u,v)\in E} dist^{(i-1)}(s,u) + \ell(u,v)$
  - If $dist^{(\le 2n-1)}(s,v) < dist^{(\le n-1)}(s,v)$ for any v
    - Output "oh no a negative length cycle"

    Easy fix!
  - Else return $dist^{(\le n-1)}(s,\cdot)$

我们已证明了:

① shortest path 一定存在

② 无 negative cycles ⇒ the shortest can be be found within $d^{\le n-1}(s,v)$ cycles

Now we claim:

∃ negative cycle iff

$$\min_{i \le 2n-1}\left(dist^{(i)}(s,v)\right) < \min_{i \le n-1}\left(dist^{(i)}(s,v)\right)$$

(再循环一轮, 一定能找出 negative cycle)

3.

Path-Doubling:

Bellman-Ford for all-pairs shortest paths

All-pairs shortest paths

- New game: compute **all pairs** distances.
- One option: run Bellman-Ford from every source node.
  - $O(mn) \times n = O(mn^2)$
- Can we do better?

path-doubling

$O(n^3 \log n)$

看似 $n^3\log n$ 更大

实则 n^2m 通常 $> n^3\log n$

(即 $n\log n < m$)

因为 $|E|$ 一般 $>> |V|$

(edges) (nodes)

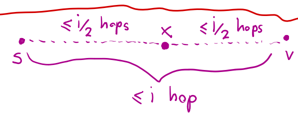
Better Idea?

- **Bellman-Ford's recursive strategy:** compute $dist^{(i)}(s,v)$ using $dist^{(i-1)}(s,\cdot)$
- **New idea**: Can you compute $dist^{(\le i)}(s,v)$ using array $dist^{(\le i/2)}(\cdot,\cdot)$?

Path-doubling for APSP

Key idea:

"each path must have a middle node"



Think: write a recurrence for $dist^{(\le i)}(s,v)$ in term of $dist^{(\le i/2)}(\cdot,\cdot)$

$$dist^{(\le i)}(s,t) = \min_{x} dist^{(\le i/2)}(s,x) + dist^{(\le i/2)}(x,t)$$

Question: why couldn't we use this idea for single-source shortest path?

因为只有 APSP 才会自然要算这个

否则是一个累赘计算(不减 time 反而加 time)

- All-Pairs Bellman-Ford(G) assume no neg-length cycs
  - Initialize array $dist$ indexed by i, s, t index entries by $dist^{(\le 2^i)}(s,t)$
  - $dist^{(\le 1)}(s,t) \leftarrow \begin{cases} 0 & \text{if } s = t \\ \ell(s,t) & \text{if } (s,t) \in E \\ \infty & \text{if } (s,t) \notin E \end{cases}$ for all s,t

    New base case! Have to start doubling from 1
  - For $i = 1, \dots, \lceil \log n \rceil$:

    Total time: $O(n^3 \log n)$ operations
    - For all nodes s, t:

      New part
      - $dist^{(\le 2^i)}(s,t) = \min_x \left(dist^{(\le 2^{i-1})}(s,x) + dist^{(\le 2^{i-1})}(x,t) \right)$
  - Return $dist^{(\le n)}$

即：尝试把每个点都作为中点，看看哪个最短

Faster Algorithms for SSSP

- Bernstein, Nanongkai, Wulff-Nilsen, 2022: **O(m ⋅ log^{8}n)** ← integer weights

  Wein's postdoc advisor Saranurak's PhD advisor
- Fineman, 2023: **O(mn$^{7/8}$)** ← any weights
- If no negative weights and Dijkstra's algorithm: **O((m + n) log n)** using binary heap and **O(m + n log n)** using Fibonacci heap

Initial idea for solving APSP: Run SSSP from every vertex!

That works, but the algorithm you're about to see is faster for *dense* graphs: **O(n³)** instead of **Θ(mn²)** (better when m >> n).

Floyd-Warshall

for all-pairs shortest paths

APSP options

- Bellman-Ford (naïve method):
  - $O(mn^2)$ time $|E||V|^2$
- Bellman-Ford (with path-doubling):
  - $O(n^3 \log n)$ time $|V|^3 \log |V|$
- Floyd-Warshall (next):
  - $O(n^3)$ time $|V|^3$

Floyd–Warshall algorithm

- The Floyd–Warshall algorithm, using dynamic programming, is an algorithm for finding all-pairs shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles).

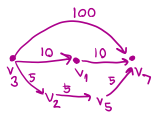
Floyd-Warshall APSP

- **Ordered** vertex set $V = \{v_1, v_2, \dots, v_n\}$.
- For a path $\pi = (s, u_1, u_2, \dots, u_{k-1}, t)$ from s to t, say that $\{u_1, u_2, \dots, u_{k-1}\}$ are its ***intermediate vertices***.

Definition

$dist^{[i]}(s,t)$ is the "middle-restricted distance:"
Shortest length of an $s \to t$ path that
only uses $\{v_1, \dots, v_i\}$ as intermediate vertices
(but s, t can be anything)

- **Example**:
  - $dist^{[0]}(s,t) = 100$
  - $dist^{[1]}(s,t) = 20$
  - $dist^{[5]}(s,t) = 15$



Floyd-Warshall APSP

- **Ordered** vertex set $V = \{v_1, v_2, \dots, v_n\}$.
- For a path $\pi = (s, u_1, u_2, \dots, u_{k-1}, t)$ from s to t, say that $\{u_1, u_2, \dots, u_{k-1}\}$ are its ***intermediate vertices***.

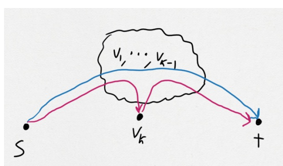
Definition

$dist^{[i]}(s,t)$ is the "middle-restricted distance:"
Shortest length of an $s \to t$ path that
only uses $\{v_1, \dots, v_i\}$ as intermediate vertices
(but s, t can be anything)

- **Final Goal:** for all s, t, $dist^{[n]}(s,t)$ (same as $dist(s,t)$, why?)
- **Strategy:** compute $dist^{[k]}(s,t)$ from $dist^{[k-1]}(\cdot,\cdot)$.

Recursive Strategy

Key idea:
"Shortest k-middle-restricted path either go through v_k or not"



write a recurrence for $dist^{[k]}(s,t)$ in term of $dist^{[k-1]}(\cdot,\cdot)$

$$dist^{[k]}(s,t) = \min \begin{cases} dist^{[k-1]}(s,t) \\ dist^{[k-1]}(s,v_k) + dist^{[k-1]}(v_k,t) \end{cases}$$

Floyd-Warshall APSP

- (Base Case) $\text{dist}^{[0]}(s,t) := \begin{cases} 0 & \text{if } s = t \\ \ell(s,t) & \text{if } (s,t) \in E \\ \infty & \text{otherwise} \end{cases}$

No midpoints allowed
Only direct s-t path allowed
(if it exists)

- For all $k = 1, \ldots, n$:
  - For all vertices s, t:
    - $dist^{[k]}(s,t) = \min \begin{cases} dist^{[k-1]}(s,t) \\ dist^{[k-1]}(s, v_k) + dist^{[k-1]}(v_k, t) \end{cases}$
- Return $dist^{[n]}$

Total time: $O(n^3)$ operations

Pseudocode for Floyd–Warshall

```
Algorithm APSP(G)
    table := 3D-array (1..n, 1..n, 0..n)
    // first two dimensions represent vertices v1,...,vn,
      third dimension represents restricting to the first i internal vertices
    for s = 1 to n:
        for t = 1 to n:
            table(s, t, 0) = w(s,t)   // base case
    for s = 1 to n:
        for t = 1 to n:
            for i = 1 to n:
                table(s,t,i) = min{table(s,t,i-1), table(s,i,i-1) + table(i,t,i-1)}
    Return table(s, t, n) for all s,t
```


Progress on APSP since Floyd-Warshall

Author	Runtime	Year
Fredman	$n^3 \log\log^{1/3} n / \log^{1/3} n$	1976
Takaoka	$n^3 \log\log^{1/2} n / \log^{1/2} n$	1992
Dobosiewicz	$n^3 / \log^{1/2} n$	1992
Han	$n^3 \log\log^{5/7} n / \log^{5/7} n$	2004
Takaoka	$n^3 \log\log^2 n / \log n$	2004
Zwick	$n^3 \log\log^{1/2} n / \log n$	2004
Chan	$n^3 / \log n$	2005
Han	$n^3 \log\log^{5/4} n / \log^{5/4} n$	2006
Chan	$n^3 \log\log^3 n / \log^2 n$	2007
Han, Takaoka	$n^3 \log\log n / \log^2 n$	2012
Williams	$n^3 / \exp(\sqrt{\log n})$	2014

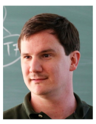
Conclusion: Maybe $O(n^{2.999})$ is impossible?

Maybe $O(n^{2.999})$ is impossible?

Either ALL of the following have $O(n^{<3})$ time algorithms or NONE of them do: (Virginia Vassilevska Williams, Ryan Williams, 2010)

1. APSP
2. Minimum Weight Triangle
3. Metricity
4. Minimum Cycle
5. Distance Product
6. Second Shortest Path
7. Replacement Paths
8. Negative Triangle Listing

…



State of the art

- No $O(n^{2.99})$ algorithm for APSP is known.
  - One of the three biggest open problems in algorithms!
  - Plays a role like SAT/NP-Hardness: lots of problems are "APSP-Hard" under the conjecture that no $O(n^{2.99})$ algorithm exists.

Quick reflection

All shortest paths algorithms so far are just **dynamic programming on graphs**.