Introduction to Algorithms
Lecture 10 All Pairs Shortest Paths

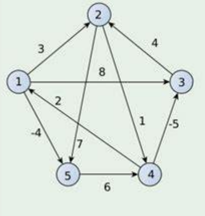
Xue Chen
xuechen1989@ustc.edu.cn
2024 spring in

Outline

All Pairs Shortest Paths (APSP)

Problem Description

Given a weighted direct-graph G, compute the shortest paths between all pairs $s \in V$ and $t \in V$.



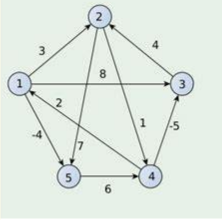
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

1. Fundamental problem in CS, e.g., compute the table of fastest transportation between major cities.

All Pairs Shortest Paths (APSP)

Problem Description

Given a weighted direct-graph G, compute the shortest paths between all pairs $s \in V$ and $t \in V$.



	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

1. Fundamental problem in CS, e.g., compute the table of fastest transportation between major cities.
2. Use Adjacency matrix to store $\binom{n}{2}$ shortest paths
3. Notation: $w(i, j)$ denotes (negative) weight of edge (i, j)
   $\delta(i, j)$ denotes shortest distance between i and j
   $d(i, j)$ denotes ALG's output

Overview

Known: Single Source Shortest Paths

1. Non-negative weights: Dijkstra's algorithm in time $O(m \log n)$
   $\Rightarrow$ APSP for non-negative weights in time $O(nm \log n)$

Overview

Known: Single Source Shortest Paths

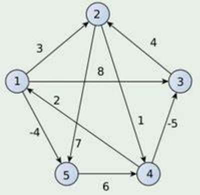
1. Non-negative weights: Dijkstra's algorithm in time $O(m \log n)$
   $\Rightarrow$ APSP for non-negative weights in time $O(nm \log n)$
2. Negative weights: Bellman-Ford algorithm in time $O(nm)$
   $\Rightarrow$ APSP in time $O(n^2 m)$

Overview

Known: Single Source Shortest Paths

1. Non-negative weights: Dijkstra's algorithm in time $O(m \log n)$
   $\Rightarrow$ APSP for non-negative weights in time $O(nm \log n)$
2. Negative weights: Bellman-Ford algorithm in time $O(nm)$
   $\Rightarrow$ APSP in time $O(n^2 m)$

Always allow negative weights in this lecture:



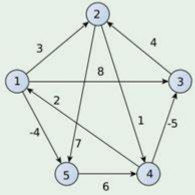
	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

Overview

Known: Single Source Shortest Paths

1. Non-negative weights: Dijkstra's algorithm in time $O(m \log n)$
   $\Rightarrow$ APSP for non-negative weights in time $O(nm \log n)$
2. Negative weights: Bellman-Ford algorithm in time $O(nm)$
   $\Rightarrow$ APSP in time $O(n^2 m)$

Always allow negative weights in this lecture:



	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

1. Dense graph: Floyd-Warshall Algorithm in time $O(n^3)$ — better than running Bellman-Ford$\times n$ times
2. Sparse graph: Johnson's algorithm in time $O(nm \log n)$ — generalizes Dijkstra's ALGO to negative weights

Outline

Introduction

Basic Idea

Consider dynamic programming instead of the greedy method (b.c. of negative weights)

Introduction

Basic Idea

Consider dynamic programming instead of the greedy method (b.c. of negative weights)

1. 1st idea: Define $d^{\ell}(i,j)$ as the shortest distance from i to j over all paths of $\leqslant \ell$ edges

   — leads to involved algorithms with time $o(n^3)$

Introduction

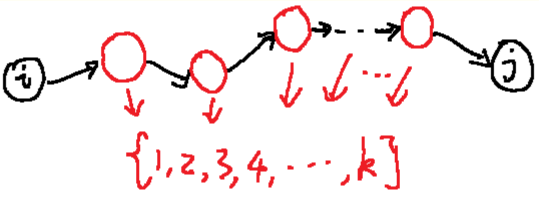
Basic Idea

Consider dynamic programming instead of the greedy method (b.c. of negative weights)

1. 1st idea: Define $d^\ell(i,j)$ as the shortest distance from i to j over all paths of $\leqslant \ell$ edges

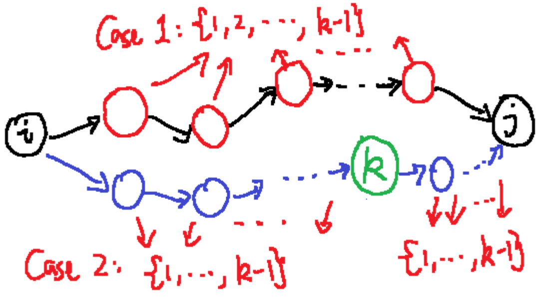
   — leads to involved algorithms with time $o(n^3)$

2. 2nd idea: Define $d^k(i,j)$ as the shortest distance from i to j over all paths where all intermediate vertices are in $\{1,2,\ldots,k\}$



— called Floyd-Warshall Algorithm

Dynamic Programming



Recursive Solution

$$d^k(i,j) = \begin{cases} w(i,j) & \text{if } k = 0; \\ \min\Big\{ \underbrace{d^{k-1}(i,j)}_{Case\ 1}, \underbrace{d^{k-1}(i,k) + d^{k-1}(k,j)}_{Case\ 2} \Big\} & \text{if } k > 0; \end{cases}$$

Algorithm Description

procedure FLOYD-WARSHALL(G)
  Initialize $d(i,j) = w(i,j)$ for all i and j
  for $k = 1, \ldots, n$ **do**
    for $i = 1, \ldots, n$ **do**
      for $j = 1, \ldots, n$ **do**
        $d(i,j) = \min\{d(i,j), d(i,k) + d(k,j)\}$

1. Running Time: $O(n^3)$

Algorithm Description

```
procedure FLOYD-WARSHALL(G)
    Initialize d(i, j) = w(i, j) for all i and j
    for k = 1, ..., n do
        for i = 1, ..., n do
            for j = 1, ..., n do
                d(i, j) = min{d(i, j), d(i, k) + d(k, j)}
```

1. Running Time: $O(n^3)$
2. Record decisions $\Pi(i, j)$ to find the path
3. Elegant and easy to implement

Algorithm Description

---

procedure FLOYD-WARSHALL(G)
    Initialize $d(i,j) = w(i,j)$ for all i and j
    for $k = 1, \ldots, n$ **do**
        for $i = 1, \ldots, n$ **do**
            for $j = 1, \ldots, n$ **do**
                $d(i,j) = \min\{d(i,j), d(i,k) + d(k,j)\}$

---

1. Running Time: $O(n^3)$
2. Record decisions $\Pi(i,j)$ to find the path
3. Elegant and easy to implement
4. Next: While it works well for dense graphs, how about sparse graphs?

Outline

Motivation

Recall Single-Source Shortest Paths

1. Dijkstra's algorithm in time $O(m \log n)$ for non-negative weights
2. Bellman-Ford algorithm in time $O(mn)$ for negative weights

Goal: Design an APSP algorithm with 2 properties
(1) Time $O(n \cdot m \log n)$ (2) Allow negative weights.

Motivation

Recall Single-Source Shortest Paths

1. Dijkstra's algorithm in time $O(m \log n)$ for non-negative weights
2. Bellman-Ford algorithm in time $O(mn)$ for negative weights

Goal: Design an APSP algorithm with 2 properties
(1) Time $O(n \cdot m \log n)$ (2) Allow negative weights.

Overview

1. Faster than Floyd-Warshall algorithm for sparse graphs

Motivation

Recall Single-Source Shortest Paths

1. Dijkstra's algorithm in time $O(m \log n)$ for non-negative weights
2. Bellman-Ford algorithm in time $O(mn)$ for negative weights

Goal: Design an APSP algorithm with 2 properties
(1) Time $O(n \cdot m \log n)$ (2) Allow negative weights.

Overview

1. Faster than Floyd-Warshall algorithm for sparse graphs
2. New idea: Reweight all edges by potential functions s.t.
   (a) All new weights are non-negative
   (b) New weights "preserve" the length of each path

Main Idea of preservation

1. Define potential function $h : V \to \mathbb{R}$ on every vertex
2. Reweight each edge $w(i, j)$ as $\hat{w}(i, j) = w(i, j) + h(i) - h(j)$.

Main Idea of preservation

1. Define potential function $h : V \to \mathbb{R}$ on every vertex
2. Reweight each edge $w(i,j)$ as $\hat{w}(i,j) = w(i,j) + h(i) - h(j)$.
3. Length of path $i_1 \to i_2 \to \ldots \to i_\ell$ becomes

$$\sum_{j=1}^{<\ell} \hat{w}(i_j, i_{j+1}) = \sum_{j=1}^{<\ell} \left[w(i_j, i_{j+1}) + h(i_j) - h(i_{j+1}) \right]$$
$$= h(i_1) - h(i_\ell) + \sum_{j=1}^{<\ell} w(i_j, i_{j+1})$$

— is preserved

4. Next: How to find a potential function h?

Choose h

Goal

$\hat{w}(i,j) = w(i,j) + h(i) - h(j) \geqslant 0$ for all i, j

1. Recall that for single-source shortest paths: $d(s,j) \leqslant d(s,i) + w(i,j) \Rightarrow w(i,j) + d(s,i) - d(s,j) \geqslant 0$ for all (i,j) — how to choose s?

Choose h

Goal

$\hat{w}(i,j) = w(i,j) + h(i) - h(j) \geqslant 0$ for all i, j

1. Recall that for single-source shortest paths: $d(s,j) \leqslant d(s,i) + w(i,j) \Rightarrow w(i,j) + d(s,i) - d(s,j) \geqslant 0$ for all (i,j) — how to choose s?
2. Say $s = 1$ and apply Bellman-Ford — but the graph may be dis-connected

Choose h

Goal

$\hat{w}(i,j) = w(i,j) + h(i) - h(j) \geqslant 0$ for all i, j

1. Recall that for single-source shortest paths: $d(s,j) \leqslant d(s,i) + w(i,j) \Rightarrow w(i,j) + d(s,i) - d(s,j) \geqslant 0$ for all (i,j) — how to choose s?
2. Say $s = 1$ and apply Bellman-Ford — but the graph may be dis-connected
3. A lazy solution: create a new source $s = 0$ and add (s,i) of weight 0 for each i
4. Then apply Bellman-Ford to obtaining $h(i) = d(s,i)$

Choose h

Goal

$\hat{w}(i,j) = w(i,j) + h(i) - h(j) \geqslant 0$ for all i, j

1. Recall that for single-source shortest paths: $d(s,j) \leqslant d(s,i) + w(i,j) \Rightarrow w(i,j) + d(s,i) - d(s,j) \geqslant 0$ for all (i,j) — how to choose s?
2. Say $s = 1$ and apply Bellman-Ford — but the graph may be dis-connected
3. A lazy solution: create a new source $s = 0$ and add (s, i) of weight 0 for each i
4. Then apply Bellman-Ford to obtaining $h(i) = d(s, i)$
5. Finally apply Dijkstra's algorithm n times on the non-negative weighted graph

Description

JOHNSON(G, w)

compute G', where $G'.V = G.V \cup \{s\}$,
  $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
  $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD$(G', w, s) ==$ FALSE
  print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
    set $h(v)$ to the value of $\delta(s, v)$
      computed by the Bellman-Ford algorithm
  for each edge $(u, v) \in G'.E$
    $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$
  let $D = (d_{uv})$ be a new $n \times n$ matrix
  for each vertex $u \in G.V$
    run DIJKSTRA$(G, \widehat{w}, u)$ to compute $\widehat{\delta}(u, v)$ for all $v \in G.V$
    for each vertex $v \in G.V$
      $d_{uv} = \widehat{\delta}(u, v) + h(v) - h(u)$
  return D

Description

JOHNSON(G, w)

compute G', where $G'.V = G.V \cup \{s\}$,
  $G'.E = G.E \cup \{(s, v) : v \in G.V\}$, and
  $w(s, v) = 0$ for all $v \in G.V$
if BELLMAN-FORD(G', w, s) == FALSE
  print "the input graph contains a negative-weight cycle"
else for each vertex $v \in G'.V$
    set $h(v)$ to the value of $\delta(s, v)$
      computed by the Bellman-Ford algorithm
  for each edge $(u, v) \in G'.E$
    $\widehat{w}(u, v) = w(u, v) + h(u) - h(v)$
  let $D = (d_{uv})$ be a new $n \times n$ matrix
  for each vertex $u \in G.V$
    run DIJKSTRA$(G, \widehat{w}, u)$ to compute $\widehat{\delta}(u, v)$ for all $v \in G.V$
    for each vertex $v \in G.V$
      $d_{uv} = \widehat{\delta}(u, v) + h(v) - h(u)$
  return D

Running time: $O(nm \log n)$.

Outline

Matrix Multiplication?

Back to the idea: Define $d^{\ell}(i,j)$ as the shortest distance from i to j over all paths of $\leqslant \ell$ edges

$$\text{Example: } d^2(i,j) = \min_k \left\{ d^1(i,k) + d^1(k,j) \right\}$$

Matrix Multiplication?

Back to the idea: Define $d^{\ell}(i,j)$ as the shortest distance from i to j over all paths of $\leqslant \ell$ edges

$$\text{Example: } d^2(i,j) = \min_k \left\{ d^1(i,k) + d^1(k,j) \right\}$$

Let $A^{(\ell)}(i,j)$ denote the matrix with entries $d^{\ell}(i,j)$. If we can define some algebraic operation $(\odot, \oplus)$:

1. $d^1(i,k) + d^1(k,j) = A^{(1)}(i,k) \odot A^{(1)}(k,j)$
2. $\min_k \left\{ d^1(i,k) + d^1(k,j) \right\} = \oplus_k \left\{ A^{(1)}(i,k) \odot A^{(1)}(k,j) \right\}$

Matrix Multiplication?

Back to the idea: Define $d^{\ell}(i,j)$ as the shortest distance from i to j over all paths of $\leqslant \ell$ edges

$$\text{Example: } d^2(i,j) = \min_k \left\{ d^1(i,k) + d^1(k,j) \right\}$$

Let $A^{(\ell)}(i,j)$ denote the matrix with entries $d^{\ell}(i,j)$. If we can define some algebraic operation $(\odot, \oplus)$:

1. $d^1(i,k) + d^1(k,j) = A^{(1)}(i,k) \odot A^{(1)}(k,j)$
2. $\min_k \left\{ d^1(i,k) + d^1(k,j) \right\} = \oplus_k \left\{ A^{(1)}(i,k) \odot A^{(1)}(k,j) \right\}$
3. Then $A^{(2)} = A^{(1)} \odot A^{(1)}$ and $A^{(n)} = A^{(n-1)} \odot A^{(1)} = (A^{(1)})^{\odot n}$
4. If we have a faster matrix multiplication algorithm for these algebraic operation ... ☺

Intuition

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

Daydream?

Intuition

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

Daydream?

1. Question 1: Consider all known graph algorithms, what is the fastest one to solve Transitive Closure?

Intuition

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

Daydream?

1. Question 1: Consider all known graph algorithms, what is the fastest one to solve Transitive Closure?
2. Question 2: Can we solve it by fast matrix multiplication?

Intuition

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

Daydream?

1. Question 1: Consider all known graph algorithms, what is the fastest one to solve Transitive Closure?
2. Question 2: Can we solve it by fast matrix multiplication?
3. 1st try: Define $\odot$ as $\wedge$ and $\oplus$ as $\vee$ (See Chapter 25.1 and 25.2 in CLRS) — ☹ but they are not standard operation in $\mathbf{F}_2$

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i,j) = T$ only if $A^{n-1}(i,j) > 0$

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i, j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i, j) = T$ only if $A^{n-1}(i, j) > 0$
2. Real Question: Can we extend it to an APSP algorithm for undirected & unweighted graphs?

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i,j) = T$ only if $A^{n-1}(i,j) > 0$
2. Real Question: Can we extend it to an APSP algorithm for undirected & unweighted graphs?
3. $\delta(i,j) = \arg\min_\ell \{A^\ell(i,j) > 0\}$ and apply binary search on ℓ
4. The goal is to determine every bit of $\delta(i,j)$

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T,F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i,j) = T$ only if $A^{n-1}(i,j) > 0$
2. Real Question: Can we extend it to an APSP algorithm for undirected & unweighted graphs?
3. $\delta(i,j) = \arg\min_{\ell}\{A^{\ell}(i,j) > 0\}$ and apply binary search on ℓ
4. The goal is to determine every bit of $\delta(i,j)$
5. Rough idea: Find k such that $A^{2^k-1}(i,j) > 0$ and $A^{2^{k-1}}(i,j) = 0$
   $\Rightarrow$ the highest bit is k

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T, F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i,j) = T$ only if $A^{n-1}(i,j) > 0$
2. Real Question: Can we extend it to an APSP algorithm for undirected & unweighted graphs?
3. $\delta(i,j) = \arg\min_{\ell}\{A^{\ell}(i,j) > 0\}$ and apply binary search on ℓ
4. The goal is to determine every bit of $\delta(i,j)$
5. Rough idea: Find k such that $A^{2^k-1}(i,j) > 0$ and $A^{2^{k-1}}(i,j) = 0$
   $\Rightarrow$ the highest bit is k
6. Determine bit $k-1$: Check $A^{2^{k-1}+2^{k-2}-1}(i,j)$, and so on...

Formal Description

An easier problem: Transitive Closure

The goal is to return $T(i,j) \in \{T,F\}$ that indicates $i \rightsquigarrow j$ or no

1. 2nd try: Let A be the adjacency matrix, $T(i,j) = T$ only if $A^{n-1}(i,j) > 0$
2. Real Question: Can we extend it to an APSP algorithm for undirected & unweighted graphs?
3. $\delta(i,j) = \arg\min_{\ell}\{A^{\ell}(i,j) > 0\}$ and apply binary search on ℓ
4. The goal is to determine every bit of $\delta(i,j)$
5. Rough idea: Find k such that $A^{2^k-1}(i,j) > 0$ and $A^{2^{k-1}}(i,j) = 0$
   $\Rightarrow$ the highest bit is k
6. Determine bit $k-1$: Check $A^{2^{k-1}+2^{k-2}-1}(i,j)$, and so on...
7. Reorganize it s.t. the binary search is applied for all $\delta(i,j)$ — Easier to determine the last bit by consider A^2 reversely
8. Challenge: find those paths and weighted graphs!

On the All-Pairs-Shortest-Path Problem

RAIMUND SEIDEL*

Computer Science Division
University of California, Berkeley
Berkeley CA 94720

Abstract

The following algorithm solves the distance version of the all-pairs-shortest-path problem for undirected, unweighted n-vertex graphs in time $O(M(n) \log n)$, where $M(n)$ denotes the time necessary to multiply two $n \times n$ matrices of small integers (which is currently known to be $o(n^{2.376})$):

Input: $n \times n$ 0-1 matrix A, the adjacency matrix of undirected, connected graph G

Output: $n \times n$ integer matrix D, with d_{ij} the length of a shortest path joining vertices i and j in G

function APD(A : $n \times n$ 0-1 matrix) : $n \times n$ integer matrix
  let $Z = A \cdot A$
  let B be an $n \times n$ 0-1 matrix, where $b_{ij} = 1$ iff $i \neq j$ **and** ($a_{ij} = 1$ **or** $z_{ij} > 0$)
  if $b_{ij} = 1$ for all $i \neq j$ **then return** $n \times n$ matrix $D = 2B - A$
  let T = APD(B)
  let $X = T \cdot A$
  return $n \times n$ matrix D, where $d_{ij} = \begin{cases} 2t_{ij} & \text{if } x_{ij} \geq t_{ij} \cdot degree(j) \\ 2t_{ij} - 1 & \text{if } x_{ij} < t_{ij} \cdot degree(j) \end{cases}$

We also address the problem of actually finding a shortest path between each pair of vertices and present a randomized algorithm that matches APD() in its simplicity and in its expected running time.

Questions?