Lecture 15: graph algorithms VI – all pairs shortest paths

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
ALL PAIRS SHORTEST PATHS (APSP) PROBLEM

**Instance:** A directed graph $G = (V, E)$, and a weight matrix $W$, where $W[i, j]$ denotes the weight of edge $i\,j$, for all $i, j \in V, i \neq j$.

**Find:** For all pairs of vertices $u, v \in V, u \neq v$, a directed path $P$ from $u$ to $v$ such that

$$w(P) = \sum_{ij \in P} W[i, j]$$

is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in $G$. 
We use the following conventions for the weight matrix $W$:

$$W[i, j] = \begin{cases} w_{ij} & \text{if } (i, j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise.} \end{cases}$$

from: 

\[
\begin{array}{ccc}
\text{to: } & a & b & c & d \\
\text{a} & & 0 & 3 & \infty & \infty \\
\text{b} & \infty & 0 & 12 & 5 \\
\text{c} & 4 & \infty & 0 & -1 \\
\text{d} & 2 & -4 & \infty & 0 \\
\end{array}
\]
Run Bellman-Ford \( n \) times, once for each possible source

Complexity \( O(n^2 m) \).
(Could be \( O(n^4) \).)

Can we do better?

Matrix \( D \) of shortest path lengths

\[
D[i, j] = \begin{bmatrix}
0 & 3 & 15 & 8 \\
7 & 0 & 12 & 5 \\
1 & -5 & 0 & -1 \\
2 & -4 & 8 & 0
\end{bmatrix}
\]
A Dynamic Programming Approach

Suppose we successively consider paths of length 1, 2, \ldots, n – 1. Let $L_m[i, j]$ denote the minimum-weight $(i, j)$-path having at most $m$ edges.

We want to compute $L_{n-1}$.

Base case: $L_1 = W$

General case: How to express solution in terms of optimal solutions to subproblems?

Express shortest path with $m$ edges in terms of shortest path(s) with < $m$ edges?

For $m \geq 2$,

$$L_m[i, j] = \min\{L_{m-1}[i, k] + L_1[k, j] : 1 \leq k \leq n\}.$$ 

Problem: we don’t know the predecessor of $j$ on the optimal path $P$

Try all possible predecessors $k$

Let $P = \text{minimum weight (i, j)-path with } \leq m$ edges

Let $k$ be the predecessor of $j$ on path $P$

Then $P' = \text{minimum weight (i, k)-path with } \leq m – 1$ edges

(or could shrink $w(P)$; contra!)
Algorithm: \textit{FairlySlowAllPairsShortestPath}(W)

\[ L_1 \leftarrow W \]
\[ \text{for } m \leftarrow 2 \text{ to } n - 1 \]
\[ \quad \text{for } i \leftarrow 1 \text{ to } n \]
\[ \quad \quad \text{for } j \leftarrow 1 \text{ to } n \]
\[ \quad \quad \quad \text{do } \ell \leftarrow \infty \]
\[ \quad \quad \quad \text{for } k \leftarrow 1 \text{ to } n \]
\[ \quad \quad \quad \quad \text{do } \ell \leftarrow \min \{ \ell, L_{m-1}[i, k] + W[k, j] \} \]
\[ \quad \quad \quad L_m[i, j] \leftarrow \ell \]
\[ \text{return } (L_{n-1}) \]

Time complexity? \( O(n^4) \)

Space complexity is a bit subtle…

To compute \( L_m \), only need \( W \) and \( L_{m-1} \). No need to keep \( L_2, \ldots, L_{m-2} \). So space is \( O(|W| + |L_m| + |L_{m-1}|) = O(|L_m|) = O(n^2) \).

Note: this is asymptotically the same as input size for dense graphs where \(|E| \in \Theta(|V|^2)\).

Home exercise: do we need to keep both \( L_m \) and \( L_{m-1} \)? Or can we reuse \( L_{m-1} \) directly as our \( L_m \) array and modify it in-place?
**BETTER SOLUTION: SUCCESSIVE DOUBLING**

The idea is to construct $L_1, L_2, L_4, \ldots, L_{2^t}$, where $t$ is the smallest integer such that $2^t \geq n - 1$.

Initialization: $L_1 = W$ (as before).

Let $P = \text{minimum weight } (i, j)\text{-path with } \leq 2m \text{ edges}$

and $k = \text{midpoint} \text{ node of } P$

Then $P = P_1 \cup P_2$ where:

- $P_1$ is the minimum weight $(i, k)$-path with $\leq m \text{ edges}$ and $P_2$ is the minimum weight $(k, j)$-path with $\leq m \text{ edges}$

Updating: For $m \geq 1$,

$$L_{2m}[i, j] = \min\{L_m[i, k] + L_m[k, j] : 1 \leq k \leq n\}.$$
Second Solution: Successive Doubling

Algorithm: *FasterAllPairsShortestPath*(W)

\[L_1 \leftarrow W\]
\[m \leftarrow 1\]

**while** \(m < n - 1\)

\[\text{for } i \leftarrow 1 \text{ to } n\]

\[\text{for } j \leftarrow 1 \text{ to } n\]

\[\text{do}\]

\[\text{do}\]

\[\text{do}\]

\[\ell \leftarrow \infty\]

\[\text{for } k \leftarrow 1 \text{ to } n\]

\[\text{do}\]

\[\ell \leftarrow \min\{\ell, L_m[i, k] + L_m[k, j]\}\]

\[L_{2m}[i, j] \leftarrow \ell\]

\[m \leftarrow 2m\]

\[\text{return } (L_m)\]

Complexity analysis

\(O(n^3 \log n)\) runtime

\(O(n^2)\) space
First solution: sub-problem is a path to the **predecessor node**
- Optimality: try all possible predecessor nodes \( k \)

Second solution: sub-problems are paths to/from the **midpoint node**
- Optimality: try all possible midpoint nodes \( k \)

**Third solution:** sub-problems are paths in which all interior nodes are in \( \{1..k-1\} \)
- I.e., we restrict paths to using a prefix of all nodes
- Optimality: try all ways to use **new node** \( k \) as an interior node
Let $D_k[i, j]$ denote the length of the minimum-weight path $i \rightarrow j$ in which all interior nodes are in the set $\{1, \ldots, k\}$.

We want to compute $D_n$.

Let $P$ be a min-weight $(i, j)$-path in which all interior nodes are in $\{1, \ldots, k\}$.

**Case 1:** $k$ is not used in $P$

interior nodes are all in $\{1, \ldots, k - 1\}$

Then $D_k[i, j] = D_{k-1}[i, j]$.

**Case 2:** $k$ is used in $P$

interior nodes are all in $\{1, \ldots, k - 1\}$

Then $D_k[i, j] = D_{k-1}[i, k] + D_{k-1}[k, j]$.

Because $P$ would then contain a cycle, and the cycle cannot make $P$ shorter.

So there must be an equivalent or better $P$ without a cycle.

Optimal solution:
interior nodes are all in $\{1, \ldots, k\}$

How can we argue $k$ is not in either $P_1$ or $P_2$?

Because $P$ would then contain a cycle, and the cycle cannot make $P$ shorter.

So there must be an equivalent or better $P$ without a cycle.

more formal proof in bonus slides.
FLOYD-WARSHALL ALGORITHM

- Let $D_k[i, j]$ denote the length of the minimum-weight $(i, j)$-path in which all interior nodes are in the set of nodes $\{1 \ldots k\}$.
- Base case: $D_0 = W$
- Recurrence: $D_k[i, j] = \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}$

This returns distances. Can reconstruct paths from this.

```
FloydWarshall(W[1..n, 1..n])
    D0 = copy of weight matrix W
    D1 = new n * n matrix
    Dlast = pointer to D0
    Dcurr = pointer to D1
    for k = 1..n
        for i = 1..n
            for j = 1..n
                Dcurr[i, j] = min( Dlast[i, j], Dlast[i, k] + Dlast[k, j] )
        swap pointers Dlast and Dcurr
    return Dlast
```

Time complexity? Space complexity?
**EXAMPLE**

\[
D_0 = \begin{pmatrix}
0 & 3 & \infty & \infty \\
\infty & 0 & 12 & 5 \\
4 & \infty & 0 & -1 \\
2 & -4 & \infty & 0
\end{pmatrix} \quad D_1 = \begin{pmatrix}
0 & 3 & \infty & \infty \\
\infty & 0 & 12 & 5 \\
4 & 7 & 0 & -1 \\
2 & -4 & \infty & 0
\end{pmatrix}
\]

\[
D_2 = \begin{pmatrix}
0 & 3 & 15 & 8 \\
\infty & 0 & 12 & 5 \\
4 & 7 & 0 & -1 \\
2 & -4 & 8 & 0
\end{pmatrix} \quad D_3 = \begin{pmatrix}
0 & 3 & 15 & 8 \\
16 & 0 & 12 & 5 \\
4 & 7 & 0 & -1 \\
2 & -4 & 8 & 0
\end{pmatrix}
\]

\[
D_4 = \begin{pmatrix}
7 & 0 & 12 & 5 \\
1 & -5 & 0 & -1 \\
2 & -4 & 8 & 0
\end{pmatrix}
\]
STABLE MATCHING PROBLEM
(SOLVED WITH A GREEDY GRAPH ALGORITHM)
Problem 4.6

Stable Matching

Instance: Two sets of size $n$ say $X = [x_1, \ldots, x_n]$ and $Y = [y_1, \ldots, y_n]$. Each $x_i$ has a preference ranking of the elements in $Y$, and each $y_i$ has a preference ranking of the elements in $X$. \( \text{pref}(x_i, j) = y_k \) if $y_k$ is the $j$-th favourite element of $Y$ of $x_i$; and \( \text{pref}(y_i, j) = x_k \) if $x_k$ is the $j$-th favourite element of $X$ of $y_i$.

Find: A matching of the sets $X$ and $Y$ such that there does not exist a pair \((x_i, y_j)\) which is not in the matching, but where $x_i$ and $y_j$ prefer each other to their existing matches. A matching with this this property is called a stable matching.

Real-world examples (1950s):

- Matching medical interns to hospitals.
- Matching organs to patients requiring transplants

The 2012 Nobel Prize in economics was awarded to Roth and Shapley for their work in the "theory of stable allocation and the practice of market design".
An example of an instability: Suppose $x_i$ is matched with $y_j$, $x_k$ is matched with $y_\ell$, $x_i$ prefers $y_\ell$ to $y_j$, and $y_\ell$ prefers $x_i$ to $x_k$. 

![Diagram]

$x_i \quad x_k \quad y_j \quad y_\ell$
Overview of the Gale-Shapley Algorithm

Elements of $X$ propose to elements of $Y$.

If $y_j$ accepts a proposal from $x_i$, then the pair $\{x_i, y_j\}$ is matched.

An unmatched $y_j$ must accept a proposal from any $x_i$.

If $\{x_i, y_j\}$ is a matched pair, and $y_j$ subsequently receives a proposal from $x_k$, where $y_j$ prefers $x_k$ to $x_i$, then $y_j$ accepts and the pair $\{x_i, y_j\}$ is replaced by $\{x_k, y_j\}$.

If $\{x_i, y_j\}$ is a matched pair, and $y_j$ subsequently receives a proposal from $x_k$, where $y_j$ prefers $x_i$ to $x_k$, then $y_j$ rejects and nothing changes.

A matched $y_j$ never becomes unmatched.

An $x_i$ might make a number of proposals (up to $n$); the order of the proposals is determined by $x_i$’s preference list.
Algorithm: Gale-Shapley \((X, Y, \text{pref})\)

\[ \text{Match} \leftarrow \emptyset \]

while there exists an unmatched \(x_i\)

let \(y_j\) be the next element in \(x_i\)'s preference list

if \(y_j\) is not matched

then \(\text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\}\)

else if \(y_j\) prefers \(x_i\) to \(x_k\)

then \(\text{Match} \leftarrow \text{Match} \setminus \{x_k, y_j\} \cup \{x_i, y_j\}\)

comment: \(x_k\) is now unmatched

return \((\text{Match})\)
EXAMPLE:

Suppose we have the following preference lists:

\[
\begin{align*}
    x_1 : & \; y_2 > y_3 > y_1 \\
    x_2 : & \; y_1 > y_3 > y_2 \\
    x_3 : & \; y_1 > y_2 > y_3 \\
    y_1 : & \; x_1 > x_2 > x_3 \\
    y_2 : & \; x_2 > x_3 > x_1 \\
    y_3 : & \; x_3 > x_2 > x_1
\end{align*}
\]

The *Gale-Shapley algorithm* could be executed as follows:

<table>
<thead>
<tr>
<th>proposal</th>
<th>result</th>
<th>Match</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_1, y_2}$</td>
</tr>
<tr>
<td>$x_2$ proposes to $y_1$</td>
<td>$y_1$ accepts</td>
<td>${x_1, y_2}, {x_2, y_1}$</td>
</tr>
<tr>
<td>$x_3$ proposes to $y_1$</td>
<td>$y_1$ rejects</td>
<td></td>
</tr>
<tr>
<td>$x_3$ proposes to $y_2$</td>
<td>$y_2$ accepts</td>
<td>${x_3, y_2}, {x_2, y_1}$</td>
</tr>
<tr>
<td>$x_1$ proposes to $y_3$</td>
<td>$y_3$ accepts</td>
<td>${x_3, y_2}, {x_2, y_1}, {x_1, y_3}$</td>
</tr>
</tbody>
</table>
Proof of Correctness

First we need to show that the algorithm always terminates, i.e., it is impossible that an unmatched $x_i$ has proposed to every $y_j$.

Termination of the algorithm: Once an element of $Y$ is matched, they are never unmatched. If $x_i$ has proposed to every $y_j$, then every $y_j$ is matched. But then every element of $X$ is matched, which is a contradiction.

<table>
<thead>
<tr>
<th>So the algorithm terminates, and each $x_i$ is matched with some $y_j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Need to argue the matching is <strong>stable</strong> (i.e., optimal)</td>
</tr>
<tr>
<td>That is, no $x_i$ and $y_j$ prefer each other more than their current partners</td>
</tr>
</tbody>
</table>
To prove that the algorithm terminates with a stable matching: Suppose there is an instability: \( x_i \) is matched with \( y_j \), \( x_k \) is matched with \( y_\ell \), \( x_i \) prefers \( y_\ell \) to \( y_j \) and \( y_\ell \) prefers \( x_i \) to \( x_k \).

Observe: \( x_i \) proposes to \( y_\ell \) before proposing to \( y_j \)

There three cases to consider:

1. \( y_\ell \) rejected \( x_i \)'s proposal.
2. \( y_\ell \) accepted \( x_i \)'s proposal, but later accepted another proposal.
3. \( y_\ell \) accepted \( x_i \)'s proposal, and did not accept any subsequent proposal.

Then \( y_\ell \) should end up matched with \( x_i \). Contradiction!

Other proposal must be to someone better. Contradiction!

Contradicts our assumption that this instability exists!

Implies \( y_\ell \) already matched with someone better than \( x_i \)

And \( y_\ell \) can only change to even better partners, so \( y_\ell \)'s current partner is better than \( x_i \)

All three cases are impossible, so assumption is wrong. There cannot be an instability!
COMPLEXITY

It is obvious that the number of iterations is at most $n^2$ since every $x_i$ proposes at most once to every $y_j$.

The average number of iterations is $\Theta(n \log n)$ (but we will not prove this).

But how much **time** does it take **per iteration**?
Algorithm: \textit{Gale-Shapley}(X, Y, \text{pref})

\begin{align*}
\text{Match} & \leftarrow \emptyset \\
\text{while} & \text{ there exists an unmatched } x_i \\
& \text{ do } \\
& \quad \text{let } y_j \text{ be the next element in } x_i \text{’s preference list} \\
& \quad \text{if } y_j \text{ is not matched} \\
& \quad \quad \text{then } \text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\} \\
& \quad \quad \text{suppose } \{x_k, y_j\} \in \text{Match} \\
& \quad \quad \text{if } y_j \text{ prefers } x_i \text{ to } x_k \\
& \quad \quad \quad \text{the} \\
& \quad \quad \quad \text{Match} \leftarrow \text{Match} \setminus \{x_k, y_j\} \cup \{x_i, y_j\} \\
& \quad \text{comment: } x_k \text{ is now unmatched} \\
& \text{end} \\
& \text{return } (\text{Match}) \\
\end{align*}

Maintain a \textbf{queue} of unmatched \( x \) elements

Maintain \textbf{arrays} of matches. If \( x_i \) and \( y_j \) are matched then
\( M_{x}[i] = j \) and \( M_{y}[j] = i \)
(Initially \( M_{x}[i], M_{y}[i] = 0 \))

Simple \textbf{list} of preferences

Want to know \textbf{who} \( y_j \) is matched with

Construct an array \( R[j, i] \) containing the \textbf{rank} of \( x_i \) in \( y_j \)’s preference list

I.e., want \( R[j, i] = k \) if \( x_i \) is \( y_j \)’s \( k \)-th favourite partner

Maintain a \textbf{queue} of unmatched \( x \) elements

Simple \textbf{list} of preferences

Want to know \textbf{who} \( y_j \) is matched with

Construct an array \( R[j, i] \) containing the \textbf{rank} of \( x_i \) in \( y_j \)’s preference list

I.e., want \( R[j, i] = k \) if \( x_i \) is \( y_j \)’s \( k \)-th favourite partner

Finally, we get \( O(1) \) time per iteration, and \( O(n^2) \) time in total

Exercise: try writing pseudocode for this implementation
FORMULATING GRAPH PROBLEMS
Graphs are a very important formalism in computer science. Efficient algorithms are available for many important problems:

- exploration,
- shortest paths,
- minimum spanning trees, etc.

If we formulate a problem as a graph problem, chances are that an efficient non-trivial algorithm for solving the problem is known.

Some problems have a natural graph formulation.

- For others we need to choose a less intuitive graph formulation.
- Some problems that do not seem to be graph problems at all can be formulated as such.
The RootBear Problem:
Suppose we have a canyon with perpendicular walls on either side of a forest.
- We assume a north wall and a south wall.

Viewed from above we see the A&W RootBear attempting to get through the canyon.
- We assume trees are represented by points.
- We assume the bear is a circle of given diameter $d$.
- We are given a list of coordinates for the trees.

Find an algorithm that determines whether the bear can get through the forest.
For each input point \((x,y)\): 
**add vertices** \((x,0)\), \((x,h)\), \((x,y)\) to \(V\)

For all pairs of vertices \(u, v\) in \(V\): 
if \(\text{dist}(u,v) < d\), **add edge** \(uv\)

Also add edges between **all** vertices on each canyon wall

---

Bear **cannot** get through the canyon if North and South walls are **connected**

Test connectivity using BFS from any point on the North wall, and checking if any point on the South wall is visited.

Exercise: what if each tree had radius \(r\)?
Reliable network routing:
- Suppose we have a computer network with many links.
- Every link has an assigned reliability.
  - The reliability is a probability between 0 and 1 that the link will operate correctly.
- Given nodes $u$ and $v$, we want to choose a route between nodes $u$ and $v$ with the highest reliability.
  - The reliability of a route is a product of the reliabilities of all its links.

Reliability of path $a$-$b$-$c$-$d$:
$$0.5 \times 0.9 \times 0.75 = 0.3375$$

Higher reliability via path $a$-$b$-$d$:
$$0.5 \times 0.8 = 0.4$$
Can we turn this into a shortest path problem?

**Problem 1:** need **product** of weights not sum

Use **logs** to turn product of weights into a **sum**.

Recall: \( \log xy = \log x + \log y \). So \( \log \prod w = \sum \log w \).

\[
\log \prod \frac{1}{w} = \log \frac{1}{\prod w} = \log 1 - \log \prod w = -\log \prod w
\]

\[
= -\sum \log w = \sum (-\log w).
\]

\( \Leftarrow \) Want to minimize this!

**Problem 2:** want to **maximize** the product

A path \( P \) has maximum \( \prod w \)
IFF it has maximum \( \log \prod w \)
IFF it has minimum \( \log \prod \frac{1}{w} \)

**Solution:** create a new graph where each weight \( w \) is replaced with weight \( -\log w \)

if \( w \leq 1 \) then \( \log w \leq 0 \)
so \( (-\log w) \geq 0 \)

So we can use Dijkstra!
A MORE FORMAL OPTIMALITY ARGUMENT
FOR YOUR NOTES

Case 1: \( m \) is not used in \( P \)
interior nodes are all in \( \{1 \ldots m - 1\} \)

Then \( w(P) = D_{m-1}[i,j] \) by I.H., and \( D_m[i,j] = D_{m-1}[i,j] \)

(details in slide notes)

Case 2: \( m \) is used in \( P \)
interior nodes are all in \( \{1 \ldots m\} \)

Reduce \( P_1, P_2 \) to subproblems but what if \( m \in P_1, P_2 \)?

Claim: \( \exists \) optimal path \( P' = P'_1, m, P'_2 \) such that \( P'_1 \) and \( P'_2 \) have all interior nodes in \( \{1 \ldots m - 1\} \)

Consider \( P' \)

By I.H., \( w(P'_1) = D_{m-1}[i,m] \) and \( w(P'_2) = D_{m-1}[m,j] \)

And \( w(P'_1) + w(P'_2) = D_{m-1}[i,m] + D_{m-1}[m,j] = D_m[i,j] \)

(If \( m \) appears twice in \( P \), it creates a cycle which can be removed to get \( P' \) with same or better weight)

Let \( P \) be a min-weight \((i,j)\)-path in which all interior nodes are in \( \{1 \ldots m\} \)

(Base case \( D_0[i,j] \) is left as an exercise)