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 $u, v \in V$, a directed path $P$ from $u$ to $v$ such that $w(P) = \sum_{k \in P} W[k, j]$ is minimized.

We allow edges to have negative weights, but we assume there are no negative-weight directed cycles in $G$.

EASY SOLUTION

Run Bellman-Ford $m$ times, once for each possible source.

Output: Matrix $D$ of shortest path lengths

Complexity is $O(n^3)$.

Can we do better?

Algorithm: Fastest all-pairs shortest path($W$)

$D_1 \leftarrow W$ for $m \to n - 1$

for $i = 1$ to $n$

for $k = 1$ to $n$

$D_{i+1}[k, j] \leftarrow \min(D_i[k, j], D_i[k, i] + D_i[i, j])$

return ($D_{m}$)

Time complexity: $O(n^3)$

Space complexity:

A bit subtle...

Home Exercise: For $n \leq 3$, we only need $W$ and $D_{max}$. No need to keep $D_{max+1}$. So space is $O(n^2)$.

Note: this is asymptotically the same as input size for dense graphs where $|E| = O(n^2)$.
**BETTER SOLUTION: SUCCESSIVE DOUBLING**

The idea is to construct $L_1, L_2, L_3, \ldots, L_{2^k}$, where $k$ is the smallest integer such that $2^k \geq n - 1$.

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

**Arguing optimal substructure**

Let $P$ be the minimum weight $(i,j)$-path with at most $2^k$ edges, and $k$ be the midpoint node of $P$.

Then $P = P_1 \cup P_2$ where $P_1$ is the minimum weight $(i,k)$-path with at most $2^k$ edges and $P_2$ is the minimum weight $(k,j)$-path with at most $2^k$ edges.

**Updating**

For $m \geq 1$,

$$L_{2^m}[i,j] = \min(L_{2^m-1}[i,k] + L_{2^m-1}[k,j])$$

**Don't know which node is midpoint of $P$, so try all $k$...**

**First solution: sub-problem is a path to the predecessor node**

- Optimally: try all possible predecessor nodes $k$

**Second solution: sub-problems are paths to/from the midpoint node**

- Optimally: 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

- Optimally: try all ways to use new node $k$ as an interior node

---

**FLOYD-WARSHALL ALGORITHM**

Let $D_{i,j}$ denote the length of the minimum-weight $(i,j)$-path in which all interior nodes are in the set of nodes $\{1..k\}$.

**Base case**: $D_{0,j} = W$

**Recurrence**: $D_{i,j} = \min(D_{i-1}[i,j], D_{i-1}[i,k] + D_{i-1}[k,j])$

**Optimal solution**: interior nodes are in $\{1..k\}$

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

- Interior nodes are all in $(1..k−1)$

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

- Interior nodes are all in $(1..k−1)$

**Time complexity**: $O(n^3)$

**Space complexity**: $O(n^3)$

**EXAMPLE**

$D_0 = \begin{pmatrix} 0 & 3 & \infty \\ \infty & 0 & 12 \\ \infty & 4 & \infty \end{pmatrix}$

$D_1 = \begin{pmatrix} 0 & 3 & \infty \\ \infty & 0 & 12 \\ \infty & 4 & \infty \end{pmatrix}$

$D_2 = \begin{pmatrix} 0 & 3 & 15 \\ 3 & 0 & 12 \\ 5 & 7 & 0 \end{pmatrix}$

$D_3 = \begin{pmatrix} 0 & 3 & 15 \\ 3 & 0 & 12 \\ 5 & 7 & 0 \end{pmatrix}$

$D_4 = \begin{pmatrix} 0 & 3 & 15 \\ 3 & 0 & 12 \\ 5 & 7 & 0 \end{pmatrix}$

**Complexity analysis**

$O(n^3)$ runtime

$O(n^3)$ space
STABLE MATCHING PROBLEM
(SOLVED WITH A GREEDY GRAPH ALGORITHM)

An example of an instability: Suppose $x_1$ is matched with $y_2$, $x_2$ is matched with $y_3$, $x_1$ prefers $y_3$ to $y_2$, and $y_2$ prefers $x_1$ to $x_2$.

\[
\begin{array}{c}
\text{x}_1 \\
\text{x}_2
\end{array}
\begin{array}{c}
\text{y}_3 \\
\text{y}_2
\end{array}
\]

Keeps track of current matches
\[
\text{Match} \leftarrow \emptyset
\]
while there exists an unmatched \(x_i\)
\[
\begin{array}{l}
\text{let } \text{y}_j \text{ be the next element in } x_i\text{'s preference list} \\
\text{if } \text{y}_j \text{ is not matched} \\
\text{then } \text{Match} \leftarrow \text{Match} \cup \{x_i, y_j\} \\
\text{suppose } \{x_k, y_j\} \in \text{Match} \\
\text{if } y_j \text{ prefers } x_k \text{ to } x_i \text{ then } \\
\text{Match} \leftarrow \text{Match} \setminus \{x_k, y_j\} \cup \{x_i, y_j\} \\
\text{comment: } x_k \text{ is now unmatched}
\end{array}
\]
return (Match)

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 \(x_k\) to \(x_i\), 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.

EXAMPLE:
Suppose we have the following preference lists:
\[
\begin{array}{c}
x_1 : y_3 > y_1 > y_2 \\
x_2 : y_1 > y_2 > y_3 \\
x_3 : y_2 > y_1 > y_3 \\
y_1 : x_2 > x_3 > x_1 \\
y_2 : x_3 > x_1 > x_2 \\
y_3 : x_1 > x_2 > x_3
\end{array}
\]
The Gale-Shapley algorithm could be executed as follows:

\[
\begin{array}{c|c|c}
\text{proposition} & \text{result} & \text{Match} \\
\hline
x_1 \text{ proposes to } y_2 & y_2 \text{ accepts } & \{x_1, y_2\} \\
x_2 \text{ proposes to } y_1 & y_1 \text{ accepts } & \{x_2, y_1\}, \{x_1, y_2\} \\
x_3 \text{ proposes to } y_1 & y_1 \text{ rejects } & \\
x_2 \text{ proposes to } y_2 & y_2 \text{ accepts } & \{x_2, y_2\}, \{x_3, y_3\}, \{x_1, y_2\} \\
x_1 \text{ proposes to } y_3 & y_3 \text{ accepts } & \{x_2, y_2\}, \{x_3, y_3\}, \{x_1, y_3\}
\end{array}
\]
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 \( V \) 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.

So the algorithm terminates, and each \( x_i \) is matched with some \( y_j \).

Need to argue the matching is stable (i.e., optimal). That is, no \( x_i \) and \( y_j \) prefer each other more than their current partners.

We need to argue the matching is stable.

To prove that the algorithm terminates with a stable matching: Suppose there is an instability: \( x_i \) is matched with \( y_j \), \( z_k \) is matched with \( y_l \), \( y_j \) prefers \( x_k \) to \( y_l \), and \( y_l \) prefers \( x_i \) to \( y_j \).

Observe: \( x_i \) proposes to \( y_j \) before proposing to \( y_l \).

There are three cases to consider:

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

Contradiction! Observe: \( x_i \) proposes to \( y_j \) before proposing to \( y_l \).

Contradiction! \( y_l \) should end up matched with \( x_i \). Contradiction!

Other proposal must be to someone better. Contradiction!

Contradicts our assumption that the instability exists!

All three cases are impossible, so assumption is wrong. We cannot have 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: 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 \( \text{Match} \leftarrow \text{Match} \cup \{ x_i, y_j \} \)

if \( y_j \) prefers \( x_i \) to \( x_k \) then \( \text{Match} \leftarrow \text{Match} \cup \{ x_k, y_j \} \)

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

return \( \text{Match} \)

Maintain a queue of unmatched \( x \) elements

Maintain arrays of matches. \( M_j(i) \) and \( M_i(j) \) contain 1 if \( x_i \) and \( y_j \) are matched

Construct an array \( R_j \), containing the rank of \( x_i \) in \( y_j \)'s preference list

I.e., \( R_j \) is a simple list of preferences

So, we get \( O(n) \) 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:

- 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&I 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.

A MORE FORMAL OPTIMALITY ARGUMENT

For each input point $(x,y)$
add vertices $(x,0), (x,h), (h,y)$ to $V$
For all pairs of vertices $u,v$ in $V$.
If 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 check if any point on the South wall is visited.
Overuse what if each tree had radius $r$?

BONUS SLIDES

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES

A MORE FORMAL OPTIMALITY ARGUMENT

FOR YOUR NOTES