A Dynamic Programming Approach

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

We want to compute \( L_m(i, j) \).

Base case: \( L_0(i, j) = W \).

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 \( \leq m \) edges.

For \( m \geq 2 \),

\( L_m(i, j) = \min \{ L_{m-1}(i, k) + L_1(k, j) \mid 1 \leq k \leq n \} \).

A problem we don’t know the predecessor of \( j \) on the optimal path \( P \).

By any possible predecessor?

Arguing optimal substructure

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 \( P’ \)’s control)

The \( \{P’\} \) is a subset...

Algorithm: Fahey-SlowAllPairsShortestPath(W)

\( L_m(i, j) = W \)

for \( m = 2 \) to \( n - 1 \)

for \( i = 1 \) to \( n \)

for \( j = 1 \) to \( n \)

\( L_m(i, j) = \min \{ L_{m-1}(i, k) + W[k, j] \mid 1 \leq k \leq n \} \)

return \( L_{n-1} \).

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

Space complexity: \( O(n^3) \)

Home exercise: do we need to keep \( L_n \) and \( L_{n-1} \)?

If we can reuse \( L_n \) directly as \( L_{n-1} \), we only need \( L_{n-1} \).

So space is \( O(n^2) \) if \( n \) is small.

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

The idea is to construct $L_1, L_2, L_3, \ldots, L_n$, where $n$ is the smallest integer such that $2^n \geq 1 + n$.

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

Arguing optimal substructure,

Let $P = \text{minimum weight}$ $i,j$-path with $\leq m$ edges and $k = \text{midpoint node of } P$.

Then $P = P_1 \cup P_2$ where:

1. $P_1$ is the minimum weight $i,k$-path with $\leq m$ edges.
2. $P_2$ is the minimum weight $k,j$-path with $\leq m$ edges.

**Arguing optimal substructure** (or else we could improve $P$ by improving $P_1$ or $P_2$).

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

**SUMMARY & WHAT'S NEXT**

1. **First solution:** subproblems are paths to the predecessor node.
   - Optimality: try all possible predecessor nodes $k$.

2. **Second solution:** subproblems are paths to/from the midpoint node.
   - Optimality: try all possible midpoint nodes $k$.

3. **Third solution:** subproblems are paths in which all interior nodes are in $\{1, \ldots, k\}$.
   - 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.

**THIRD SOLUTION: FLOYD-WARSHALL**

Let $D_k[i,j]$ denote the length of the minimum-weight $i,j$-path in which all interior nodes are in $(1, k)$.

- **Base case:** $D_0[i,j] = W$.
- **Recurrence:** $D_k[i,j] = \min(D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j])$.

```
FLOYD-WARSHALL [O(n^3) runtime, O(n^2) space]
```

**Example**

```
D_0 = \begin{pmatrix}
0 & 3 & \infty & \infty \\
3 & 0 & 12 & 5 \\
\infty & 12 & \infty & \infty \\
\infty & \infty & \infty & \infty
\end{pmatrix}
```

```
D_1 = \begin{pmatrix}
0 & 3 & \infty & \infty \\
3 & 0 & 12 & 5 \\
15 & 12 & \infty & \infty \\
16 & 12 & \infty & \infty
\end{pmatrix}
```

```
D_2 = \begin{pmatrix}
0 & 3 & 15 & 8 \\
3 & 0 & 12 & 5 \\
15 & 12 & \infty & \infty \\
16 & 12 & \infty & \infty
\end{pmatrix}
```

```
D_3 = \begin{pmatrix}
0 & 3 & 15 & 8 \\
3 & 0 & 12 & 5 \\
15 & 12 & \infty & \infty \\
16 & 12 & \infty & \infty
\end{pmatrix}
```

```
D_n = \begin{pmatrix}
0 & 3 & 15 & 8 \\
3 & 0 & 12 & 5 \\
15 & 12 & \infty & \infty \\
16 & 12 & \infty & \infty
\end{pmatrix}
```

**More formal proof in bonus slides**
STABLE MATCHING PROBLEM
(SOLVED WITH A GREEDY GRAPH ALGORITHM)

Overview of the Gale-Shapley Algorithm

Algorithm: Gale-Shapley (X, Y, pref)

\[ \text{Match} \leftarrow \emptyset \]
while there exists an unmatched \( x_i \),
begin
let \( y_j \) be the next element in \( x_i \)'s preference list,
if \( y_j \) is not matched,
begin
\[ \text{Match} \leftarrow \text{Match} \cup \{ x_i, y_j \} \]
end else begin
if \( y_j \) prefers \( x_i \) to \( x_k \),
begin
\[ \text{Match} \leftarrow \text{Match} \setminus \{ x_k, y_j \} \cup \{ x_i, y_j \} \]
end end
end
end
Keep track of current matches
Termination is not so obvious...
Unmatched \( y_j \) accepts any proposal
Propose to most desired \( y_j \)
Match \( y_j \) considers upgrading

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 \( y_j \) has a preference ranking of the elements in \( X \).
\( \text{pref}(x_i, j) = y_j \) if \( y_j \) is the \( j \)-th favourite element of \( x_i \) in \( X \), and \( \text{pref}(y_j, i) = x_i \) if \( x_i \) is the \( j \)-th favourite element of \( y_j \) in \( Y \).

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

Real-world examples (1960s):
- 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.”

EXAMPLE:
Suppose we have the following preference lists:

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

The Gale-Shapley algorithm could be executed as follows:

\[ \begin{array}{c|c|c}
\text{proposal} & \text{result} & \text{Match} \\
\hline
x_1 & y_3 & (x_1, y_3) \\
x_2 & y_1 & (x_2, y_1) \\
x_3 & y_2 & (x_3, y_2) \\
y_1 & x_3 & (x_3, y_1) \\
y_2 & x_2 & (x_2, y_2) \\
y_3 & x_1 & (x_1, y_3) \\
\end{array} \]
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.

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 \( y_l \), \( x_i \) prefers \( y_j \) to \( y_k \) and \( y_l \) does not prefer \( x_i \) to \( x_k \).

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

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

Other proposal must be to someone better.

Contradicts our assumption that this instability exists!

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: Gale-Shapley\((X, Y, \text{pref})\)

\[ \text{Match} \rightarrow \text{ill} \]

while there exists an unmatched \( x_i \) and \( y_j \) do

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

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

end

end

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.

**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.

**Problem 1:** need reliability via path \( a \). 

**Problem 2:** want to turn product of weights into a sum of weights \( \sigma \).

**Reliability of path \( a \):**

\[
\text{Reliability of path } a = 0.5 \times 0.9 \times 0.75 = 0.3375
\]

Higher reliability between:

- \( a \rightarrow b \rightarrow c \rightarrow d \): 0.5 - 0.8 = 0.4