CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

• Single source shortest path
  • Dijkstra’s algorithm (non-negative weights)
  • Bellman-Ford (no negative cycles)
• All pairs shortest path
  • Two attempts at simple DP algorithms
  • Floyd-Warshall
DIJKSTRA’S ALGORITHM

**Single-source** shortest path in a graph with **non-negative edge weights**
DIJKSTRA’S ALGORITHM

- Recall:
  - Maintain a set $S$ of nodes for which shortest paths are known
  - Also maintain a distance $D[v]$ to node $v$ and predecessor $\pi[v]
  - Until $S = V$: choose $v \notin S$ with the smallest $D[v]$, add $v$ to $S$, and update $D$ and $\pi$ appropriately

**Lemma 6.12**

Suppose $v$ has the smallest $D$-value of any vertex not in $S$. Then $D[v]$ equals the weight of the shortest path $s \rightarrow v$, which we denote $P$

- Add $v$ to $S$ and try to relax its neighbours
We know the shortest path $s \rightarrow s$

Relax the neighbours of $s$

Repeatedly find the node with the minimum $D$-value and relax its neighbours
DIJKSTRA’S ALGORITHM

• Initially \( S = \{s\} \) and \( D(v) = \begin{cases} 0, & v = s \\ \infty, & v \neq s \end{cases} \)

• Until \( S = V \): choose \( v \notin S \) with the smallest \( D[v] \), then add \( v \) to \( S \), and relax its neighbours

Consider Dijkstra’s from here

Showing \( D \)-values

\( a \notin S \) has smallest \( D \)-value

relax...

\( D \) holds distances to each node.
Reverse edges in \( \pi[v], \pi[\pi[v]], \ldots \) to get shortest path to \( v \)!

a,0

b,5
d,2
c,7
f,24
e,23
h,39
i,62
j,50
k,75

l,69

\( 1,69 \)

\( 6 \)

\( 4 \)

\( 20 \)

\( 12 \)

\( 19 \)

\( 5 \)

\( 18 \)

\( 10 \)

\( 15 \)

\( 2 \)

\( 17 \)

\( 1 \)

\( 16 \)

\( 14 \)

\( 7 \)

\( 8 \)

\( 22 \)

\( 11 \)

\( 2 \)

\( 3 \)

\( 2 \)}
OUTPUTTING ACTUAL SHORTEST PATH(S)

Algorithm: \textit{FindPath}(s, \pi, v)

\begin{align*}
\text{path} & \leftarrow v \\
\text{path} & \leftarrow v \\
\text{while } u \neq s \\
\text{do } \left\{ \begin{array}{l}
  u \leftarrow \pi[u] \\
  \text{path} \leftarrow u \parallel \text{path}
\end{array} \right.
\end{align*}

\text{return } (\text{path})
Algorithm: \( \text{Dijkstra}(G, w, s) \)

\[
\begin{align*}
S & \leftarrow \{s\} \\
D[s] & \leftarrow 0 \\
\text{for all } v \in V \setminus \{s\} & \text{ do } \\
& \quad \{ D[v] \leftarrow w(s, v) \} \\
\pi[v] & \leftarrow s \\
\text{while } |S| < n & \text{ do } \\
& \quad \{ \text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \} \\
& \quad \{ S \leftarrow S \cup \{v\} \} \\
& \quad \{ \text{for all } v' \in V \setminus S \text{ do } \} \\
& \quad \quad \{ \text{if } D[v] + w(v, v') < D[v'] \} \\
& \quad \quad \quad \{ D[v'] \leftarrow D[v] + w(v, v') \} \\
& \quad \quad \quad \{ \pi[v'] \leftarrow v \} \\
\text{return } (D, \pi) \\
\end{align*}
\]

Build heap with all distances set to \( \infty \) except for \( s \)

After \textbf{relaxing} \( v \), we call \( Q.\text{updatePriority}(v, D[v]) \)

How can we more efficiently implement this step?

Keep nodes in a \textbf{priority queue} \( Q \), sorted by \( D \) values!

\( v = Q.\text{removeMin()} \)

Can use array of bool to maintain \( S \)

Adjacency lists & \textbf{priority queue}: \( O((n + m) \log n) \)

[Simple implementation]

Adjacency matrix & linear search: \( O(n^2) \)

After \textbf{relaxing} \( v' \), we call \( Q.\text{updatePriority}(v', D[v']) \)
BELLMAN-FORD

Single-source shortest path in a graph with negative edge weights but no negative cycles
Shortest Paths and Negative Weight Cycles

Subsequent algorithms we will be studying will solve shortest path problems as long as there are no cycles having negative weight.

If there is a negative weight cycle, then there is no shortest path (why?). There is still a shortest simple path, but there are apparently no known efficient algorithms to find the shortest simple paths in graphs containing negative weight cycles.

If there are no negative weight cycles, we can assume WLOG that shortest paths are simple paths (any path can be replaced by a simple path having the same weight).

Negative weight edges in an undirected graph are not allowed, as they would give rise to a negative weight cycle (consisting of two edges) in the associated directed graph.
BELLMAN-FORD

The Bellman-Ford algorithm solves the single source shortest path problem in any directed graph without negative weight cycles.

The algorithm is very simple to describe:

Repeat $n - 1$ times: relax every edge in the graph (where relax is the updating step in Dijkstra’s algorithm).

```
1 BellmanFord(V[1..n], E[1..m], s)
2 pred[1..n] = new array filled with null
3 D[1..n] = new array filled with infinity
4 D[s] = 0
5 for i = 1..n
6     for (u, v, w) in E
7         if D[u] + w < D[v]
8             D[v] = D[u] + w
9             pred[v] = u
10 return (D, pred)
```
WORST CASE EXECUTION

Edges happen to be processed right to left by the inner loop. Best case is left-to-right order.

Dijkstra’s is similar, but consistently achieves good ordering using its priority queue.

```
1   BellmanFord(V[1..n], E[1..m], s)
2       pred[1..n] = new array filled with null
3       D[1..n] = new array filled with infinity
4       D[s] = 0
5       for i = 1..n
6           for (u,v,w) in E
7               if D[u] + w < D[v]
8                   D[v] = D[u] + w
9                   pred[v] = u
10      return (D, pred)
```

Since the longest possible path without a cycle can be \(|V| - 1\) edges, the edges must be scanned \(|V| - 1\) times to ensure the shortest path has been found for all nodes.
Recall every simple path has at most $V - 1$ edges.

So, if $D[u] \neq \infty$, it is equal to the weight of some path $s \rightarrow u$; and

if there is a path $P = (s \rightarrow u)$ with at most $i$ edges, then $D[u] \leq w(P)$

So, after $|V| - 1$ iterations, if $\exists$ path $P$ with at most $|V| - 1$ edges, then $D[u] \leq w(P)$. (Note: any more edges would create a cycle.)

So, if $u$ is reachable from $s$, then $D[u]$ is the length of the shortest simple path from $s$ to $u$
All-Pairs Shortest Paths

Problem 6.13
All-Pairs Shortest Paths
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 $W$:

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

Could we solve this problem with a reduction? Run Bellman-Ford $n$ times, once for each possible source. Complexity $O(n^2 m)$. (Could be $O(n^4)$.) Can we do better?
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 \( L_{n-1} \). We can use a dynamic programming approach to do this.

Initialization: \( L_1 = W \).

Optimal structure: Let \( k \) be the predecessor of \( j \) on the minimum-weight \((i, j)\)-path \( P \) having at most \( m \) edges. Then the portion of \( P \) from \( i \) to \( k \), say \( P' \), is a minimum-weight \((i, k)\)-path having at most \( m - 1 \) edges. This is the optimal structure required in order to find a dynamic programming algorithm.

Updating: For \( m \geq 2 \),

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

(Note that \( k = i, j \) does not cause any problems.)
**First Solution**

**Algorithm:** `FairlySlowAllPairsShortestPath(W)`

\[ L_1 \leftarrow W \]

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

\[ \text{do}\]

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

\[ \text{do}\]

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

\[ \ell \leftarrow \infty \]

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

\[ \ell \leftarrow \min\{\ell, L_{m-1}[i, k] + W[k, j]\} \]

\[ L_m[i, j] \leftarrow \ell \]

\[ \text{return } (L_{n-1}) \]

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?

**Complexity?**

\( O(n^4) \) runtime

Space complexity is more tricky...

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 \( |E| \in \Theta(|V|^2) \)
SECOND 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).

Optimal structure: Let $k$ be the midpoint of $j$ on the minimum-weight $(i, j)$-path $P$ having at most $2m$ edges. Then the portion of $P$ from $i$ to $k$ is a minimum-weight $(i, k)$-path having at most $m$ edges, and the portion of $P$ from $k$ to $j$ is a minimum-weight $(k, j)$-path having at most $m$ edges.

Updating: For $m \geq 1$,

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

Complexity: ?

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

Let $k = \textbf{midpoint} \text{ node of } P$

Then $P = P_1 \cup P_2$ where

$P_1$ is the minimum weight $(i, k)$-path with $\leq m$ edges

$P_2$ is the minimum weight $(k, j)$-path with $\leq m$ edges
Second Solution: Successive Doubling

Algorithm: \textit{FasterAllPairsShortestPath}(W)

\begin{align*}
L_1 & \leftarrow W \\
m & \leftarrow 1 \\
\text{while } m < 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[i, k] + L_m[k, j]\} & \\
\quad \quad \quad \quad L_{2m}[i, j] \leftarrow \ell & \\
\quad \quad m & \leftarrow 2m
\end{align*}

return \((L_m)\)
• First solution: sub-problem is a path to the predecessor node
  • Optimality: try all sub-problems by trying all edges into $j$

• Second solution: sub-problems are paths to/from the midpoint node
  • Optimality: try all sub-problems by trying all possible midpoints

• Third solution: sub-problems are paths in which all interior nodes are in $\{1 \ldots m - 1\}$
  • i.e., we restrict paths to using a prefix of all nodes
  • Optimality: try all ways to use a new node $m$ as an interior node
Let $D_m[i, j]$ denote the minimum-weight $(i, j)$-path in which all interior vertices are in the set $\{1, \ldots, m\}$. We want to compute $D_n$.

Initialization: $D_0 = W$.

Updating: For $m \geq 1$,

$$D_m[i, j] = \min\{D_{m-1}[i, j], D_{m-1}[i, m] + D_{m-1}[m, j]\}.$$ 

**Optimality argument:**

By induction: **suppose $D_{m-1}[i, j]$ is correct** for all $i, j$. We show $D_m[i, j]$ is correct.

**Case 1:** all interior nodes in $P$ are in $\{1 \ldots m-1\}$

Then $w(P) = D_{m-1}[i, j]$ by I.H., and $D_m[i, j] = D_{m-1}[i, j]$ [Proof by contradiction in slide notes]

**Case 2:** $m$ is an interior node in $P$

$m$ prevents us from reducing to a sub-problem

Claim: $\exists$ a minimum-weight $(i, j)$-path with **ONE** interior $m$

By I.H., $w(P_1) = D_{m-1}[i, m]$ and $w(P_2) = D_{m-1}[m, j]$

And $w(P) = w(P_1) + w(P_2) = D_{m-1}[i, m] + D_{m-1}[m, j] = D_m[i, j]$
\[
D_0 = \begin{pmatrix}
0 & 3 & \infty & \infty \\
\infty & 0 & 12 & 5 \\
4 & \infty & 0 & -1 \\
2 & -4 & \infty & 0
\end{pmatrix}
\]

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

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

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

\[
D_4 = \begin{pmatrix}
0 & 3 & 15 & 8 \\
\boxed{7} & 0 & 12 & 5 \\
\boxed{1} & -5 & 0 & -1 \\
2 & -4 & 8 & 0
\end{pmatrix}
\]
NEXT TIME

• Intractability (hardness of problems)
  • Decision problems
  • Complexity classes P and NP
  • Reductions