CS 341: ALGORITHMS

Lecture 18: graphs – single-source shortest paths (SSSP), all-pairs shortest paths (APSP)

Slides by Trevor Brown (some material from Doug Stinson)

trevor.brown@uwaterloo.ca
(http://tbrown.pro)
DIJKSTRA’S ALGORITHM

Single-source shortest path in a graph with non-negative edge weights
Dijkstra’s Algorithm

- Iteratively construct a set $S$ of nodes for which we know the shortest path from $s_0$ (initially $S = \{s_0\}$)
- Maintain a distance $D[v]$ for each node $v$
  - If $v \in S$ then $D[v] =$ weight of the shortest path $s_0 \rightarrow v$
  - Otherwise, $D[v] =$ weight of the shortest path $s_0 \rightarrow v$
    such that all interior nodes on the path are in $S$
  - If there is no such path then $D[v] = \infty$
- Grow $S$ by adding the $v \notin S$ that has the smallest $D[v]$ value

Why can we add this $v$ to $S$? Do we know the shortest path $s_0 \rightarrow v$?

Proof: Suppose $D[v]$ is correct for all $v$ before adding $v$ to $S$. We show this holds after...
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 \( s_0 \rightarrow v \), which we denote \( P \).

Proof.

Suppose there is a path \( s_0 \rightarrow v' \) with weight less than \( D[v] \). Let \( v' \) be the first vertex of \( P' \) not in \( S \). Observe that \( v' \neq v \). Decompose \( P' \) into two paths: a \( s_0 \rightarrow v' \) and a path \( v' \rightarrow v \). We have

\[
\begin{align*}
    w(P') &= w(P_1) + w(P_2) \\
    &\geq D[v'] + w(P_2) \\
    &\geq D[v]
\end{align*}
\]

Since all \( D \)-values are correct before adding \( v \) to \( S \), and by def'n of \( D[v'] \)

This is a contradiction because we assumed \( w(P') < w(P) \).

Hence, we know the weight of the shortest path \( s_0 \rightarrow v \).

Suppose \( v' = v \) for contra. Then \( D[v'] = D[v] \), so \( w(P') \geq D[v] \)

Contradicts \( w(P') < D[v] \)

Because \( D[v'] \geq D[v] \) (by assumption) and \( w(P_2) \geq 0 \) (non-negative weights)
Lemma 6.12 says we can add \( v \) to \( S \).

To update a value \( D[v'] \) for \( v' \notin S \), we consider the new “candidate” path consisting of the shortest \((s_0, v)\)-path together with the edge \( vv' \).

If this is shorter than the current best \((s_0, v')\)-path, then update.

Updating is only required for vertices \( v' \in \text{Adj}[v] \).

Considering a new candidate path to \( v' \), and possibly updating \( D[v'] \), is known as **relaxing** \( v' \).

To reconstruct an actual shortest path from distances, we also maintain a **predecessor** \( \pi[v] \) for each node \( \pi[v] = \text{previous node on a path } s_0 \leftrightarrow v \text{ with weight } D[v] \)
Algorithm: \texttt{Dijkstra}(G, w, s)

1. \(S \leftarrow \{s_0\}\)
2. \(D[s_0] \leftarrow 0\)
3. \textbf{for all} \(v \in V \setminus \{s_0\}\) \textbf{do}
   \begin{align*}
   &D[v] \leftarrow w(s_0, v) \\
   &\pi[v] \leftarrow s_0 \\
   \end{align*}
4. \textbf{while} \(|S| < n\) \textbf{do}
   \begin{align*}
   &\text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \\
   &S \leftarrow S \cup \{v\} \\
   &\textbf{for all} \ v' \in V \setminus S \textbf{ do}
   \begin{align*}
   &\text{if } D[v] + w(v, v') < D[v'] \\
   &\quad \textbf{do}
   \begin{align*}
   &D[v'] \leftarrow D[v] + w(v, v') \\
   &\pi[v'] \leftarrow v
   \end{align*}
   \end{align*}
   \end{align*}
5. \textbf{return} \((D, \pi)\)

- Can use \textbf{array of Booleans} to maintain \(S\)
- We \textbf{know} the shortest path \(s_0 \rightarrow s_0\)
- Relax the neighbours of \(s_0\)
- Repeatedly find the node with the minimum \(D\)-value and relax its neighbours
- Basic algorithm: this is a \textbf{loop} over all \(v\) in \(V \setminus S\)
- Set predecessor when we improve \(D\)

[Complexity of the basic algorithm] \(O(n^2)\) with adjacency matrix & linear search
**DIJKSTRA’S ALGORITHM**

- Initially \( S = \{s_0\} \) and \( D(v) = \begin{cases} 0, & v = s_0 \\ \infty, & v \neq s_0 \end{cases} \)

- 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...

Reverse edges in \( \pi[v], \pi[\pi[v]], \ldots \) to get shortest path to \( v \)!

Done!
OUTPUTTING ACTUAL SHORTEST PATH(S)

Algorithm: \textit{FindPath}(s_0, \pi, v)

\begin{align*}
\text{path} & \leftarrow v \\
u & \leftarrow v \\
\text{while } u \neq s_0 \\
\text{do } & \begin{cases} u & \leftarrow \pi[u] \\
\text{path} & \leftarrow u \ || \ \text{path} \\
\end{cases} \\
\text{return } (\text{path})
\end{align*}
Algorithm: \textit{Dijkstra}(G, w, s_0)

\begin{align*}
S &\leftarrow \{s_0\} \\
D[s_0] &\leftarrow 0 \\
\text{for all } v \in V \setminus \{s_0\} &\text{ do } \\
&\left\{ D[v] \leftarrow w(s_0, v) \\
&\pi[v] \leftarrow s_0
\right\}
\text{while } |S| < n &\text{ do } \\
&\left\{ \begin{array}{l}
\text{choose } v \in V \setminus S \text{ such that } D[v] \text{ is minimized} \\
S \leftarrow S \cup \{v\}
\end{array}
\right\}
\text{for all } v' \in V \setminus S &\text{ do } \\
&\left\{ \begin{array}{l}
\text{if } D[v] + w(v, v') < D[v'] \\
\text{then } \\
D[v'] \leftarrow D[v] + w(v, v') \\
\pi[v'] \leftarrow v
\end{array}
\right\}
\text{return } (D, \pi)
\end{align*}

Build heap with all distances set to $\infty$ except for $D[s_0] = 0$

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

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

How \textbf{can we more efficiently implement this step?}

$v := Q$.\texttt{removeMin}()

Still an array of Booleans

Adjacency lists & priority queue: $O((n + m) \log n)$

Compare to $O(n^2)$ for adj matrix & linear search

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

*Single-source* shortest path in a graph with possibly *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)
```

$O(n)$ outer iterations
$O(n)$ inner iterations per outer iteration
$O(m)$ inner iterations

Could be $O(n^3)$

Total $O(nm)$
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.

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.

---

**WORST CASE EXECUTION**

**Need $|V|$ iterations of outer loop**

**Need one iteration**

---

**Edges happen**

right to left** by the inner loop

**Best case is**

left-to-right order

---

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.

---

Dijkstra’s is similar, but consistently achieves good ordering using its priority queue.
WHY BELLMAN-FORD WORKS

• **Not** going to prove this (by induction), but the crucial **lemma** is:

  • After *i* iterations of the outer *for*-loop,
    • if \( D[u] \neq \infty \), it is equal to the weight of some path \( s \leadsto u \); and
    • if there is a path \( P = (s \leadsto u) \) 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 (no cycles) from \( s \) to \( u \)

Recall every simple path has at most \(|V| - 1\) edges  
So what if we do another iteration, and some \( D[u] \) improves?  
There is a negative cycle!
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 $ij$, 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 $W$:

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

Could we solve this problem with a reduction? Run Bellman-Ford $n$ times, once for each possible source.

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

Can we do better?
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 \(L_{n-1}\). We can use a dynamic programming approach to do this.

**Initialization:** \(L_1 = W\).

**Base case:**

- Let \(P\) be the predecessor of \(j\) on path \(P\) of length \(m\).
- Define subproblems
- Show where the final solution can be found

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

**Updating:** 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 \(P\)

- Express shortest path with \(m\) edges in terms of shortest path(s) with fewer edges?

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

(or could shrink \(w(P)\); contra!)

**Arguing optimal substructure**

Let \(k\) be the predecessor of \(j\) on path \(P\)

- Try all possible predecessors \(k\)

**Define subproblems**

- Let \(P' = \text{minimum weight} \((i, k)\)-path with \(\leq m - 1\) edges
**Algorithm:**  \( \text{FairlySlowAllPairsShortestPath}(W) \)

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

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 \textbf{input size} for dense graphs where \( |E| \in \Theta(|V|^2) \).

**Complexity?**

\( O(n^4) \) runtime

Space complexity is more tricky…

Home exercise: do we need to keep \textbf{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?
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 $m \geq 1$,

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

$\text{Complexity: } ?$

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

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

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

Then $P = P_1 \cup P_2$ where

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

Algorithm: \textit{FasterAllPairsShortestPath}(W)

\begin{align*}
L_1 & \leftarrow W \\
 m & \leftarrow 1 \\
\textbf{while } m < n - 1 \\
& \quad \textbf{for } i \leftarrow 1 \textbf{ to } n \\
& \quad \quad \textbf{for } j \leftarrow 1 \textbf{ to } n \\
& \quad \quad \quad \textbf{do } \ell \leftarrow \infty \\
& \quad \quad \quad \quad \textbf{for } k \leftarrow 1 \textbf{ to } n \\
& \quad \quad \quad \quad \quad \textbf{do } \ell \leftarrow \min\{\ell, L_m[i, k] + L_m[k, j]\} \\
& \quad \quad \quad \quad \text{ } L_{2m}[i, j] \leftarrow \ell \\
& \quad \quad \textbf{do } m \leftarrow 2m \\
\textbf{return } (L_m)
\end{align*}

Complexity?

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

$O(n^2)$ space
• 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 **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:**

**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 \( D_m[i, j] = D_{m-1}[i, j] \)

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

**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 \( m \in P_1, P_2 \)?

Consider \( P' \)

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

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

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

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

**Third Solution:**

**FLOYD-WARSHALL**

(Third solution: Floyd-Warshall)

(Base case \( D_0[i, j] \) is left as an exercise)
Let $D_k[i,j]$ denote 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: $D_k[i,j] = \min\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$

```python
FloydWarshall(V[1..n], W[1..n, 1..n])
D0 = 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 Dcurr
```

This returns distances. Can reconstruct paths from this. Easy to code, hard to teach. Not covering this.
**EXAMPLE**

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

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

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

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

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