Weighted graphs

A weighted random number generator just produced a new batch of numbers.

Let's use them to build narratives!

All sports commentary
Weighted graphs

So far we have only considered weighted graphs with “weights $\geq 0$” (Dijkstra is a super-star here)

Now we will consider graphs with any integer edge weight (i.e. negative too)
Cycles

Does a shortest path need to contain a cycle?
Cycles

Does a shortest path need to contain a cycle?

No, case by cycle weight:
positive: why take the cycle?!
zero: can delete cycle and find same length path
negative: cannot ever leave cycle
Bellman-Ford

One of the few “brute force” algorithms that got a name

Idea:
1. Relax every edge (yes, all)
2. Repeat step 1 \(|V|\) times (or \(|V| - 1\))
Bellman-Ford

\[ BF(G, w, s) \]
initialize graph
for i=1 to \(|V| - 1\)
\hspace{1cm} for each edge \((u,v)\) in \(G.E\)
\hspace{1cm} relax\((u,v,w)\)
for each edge \((u,v)\) in \(G.E\)
\hspace{1cm} if \(v.d > u.d+w(u,v)\): return false
return true
Bellman-Ford
Bellman-Ford

Correctness: (you prove)

After BF finishes: if $\delta(s,u)$ exists, then $\delta(s,u) = u.d$
Bellman-Ford

Correctness: (you prove)

After BF finishes: if $\delta(s,u)$ exists, then $\delta(s,u) = u.d$

Relxation property 5, as every edge is relaxed $|V|-1$ times and there are no loops
Bellman-Ford

Correctness: returns false if neg cycle
Suppose neg cycle: \( c = <v_0, v_1, \ldots, v_k> \)
then \( w(c) < 0 \), suppose BF return true
Then \( v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \)
sum around cycle \( c \):
\[
\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i))
\]
\[
\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d \text{ as loop}
\]
Bellman-Ford

Correctness: returns false if neg cycle

\[
\sum_{i=1}^{k} v_i \cdot d \leq \sum_{i=1}^{k} (v_{i-1} \cdot d + w(v_{i-1}, v_i))
\]

\[
\sum_{i=1}^{k} v_i \cdot d = \sum_{i=1}^{k} v_{i-1} \cdot d \text{ as loop}
\]

so \(0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)\)

but \(\sum_{i=1}^{k} w(v_{i-1}, v_i) = w(c) < 0\)

Contradiction!
All-pairs shortest path

So far we have looked at:
Shortest path from a specific start to any other vertex

Next we will look at:
Shortest path from any starting vertex to any other vertex (called “All-pairs shortest path”)

Johnson's algorithm

We will start by doing something a little funny

(This will be the most efficient for graphs without too many edges)

To compute all-pairs shortest path on G, we will modify G to make G'
Johnson's algorithm

To make $G'$, we simply add one “super vertex” that connects to all the original nodes with weight 0 edge.
Next, we use Bellman-Ford (last alg.) to find the shortest path from the “super vertex” in G' to all others (shortest path distance, i.e. d-value)
Johnson's algorithm

Then we will “reweight” the graph:

\[
\hat{w}(u, v) = w(u, v) + h(u) - h(v)
\]

(old weight) \quad \text{d-value in vertex} \quad \text{new weight}

(u,v) is a vertex pair (an edge from u to v)
Johnson's algorithm

Next, we just run Dijkstra's starting at each vertex in G (starting at A, at B, and at C for this graph)

Call these $\hat{\delta}(u, v)$

start A

start B

start C
Finally, we “un-weight” the edges:

$$\delta(u, v) = \hat{\delta}(u, v) - h(u) + h(v)$$

start A
start B
start C
Johnson's algorithm

Johnson(G)
Make G'
Use Bellman-Ford on G' to get $h$
(and ensure no negative cycle)
Reweight all edges (using $h$)
for each vertex $v$ in G
Run Dijkstra's starting at $v$
Un-weight all Dijkstra paths
return all un-weighted Dijkstra paths(matrix)
Johnson's algorithm

Runtime?
Johnson's algorithm

Runtime:
Bellman-Ford = $O(|V| \cdot |E|)$
Dijkstra = $O(|V| \log |V| + E)$

Making G' takes $O(|V|)$ to add edges
Bellman-Ford run once
weight/un-weighting edges = $O(|E|)$
Dijkstra run $|V|$ times $\text{most costly}$
Johnson's algorithm

Runtime:
Bellman-Ford = $O(|V| |E|)$
Dijkstra = $O(|V| \lg |V| + E)$

$O(|V|) + O(|V| |E|) + 2 O(|E|) + |V| O(|V| \lg |V| + E)$

$= O( |V|^2 \lg |V| + |V| |E| )$
Correctness

The proof is easy, as we can rely on Dijkstra's correctness.

We need to simply show:
(1) Re-weighting in this fashion does not change shortest path
(2) Re-weighting makes only positive edges (for Dijkstra to work)
Correctness

(1) Re-weighting keeps shortest paths
Here we can use the optimal sub-structure of paths:

If \( \delta(u, x) = < v_0, v_1, ... v_k > \) with \( v_0 = u \) and \( v_k = x \)
then \( \delta(u, x) = \delta(v_0, v_1) + \delta(v_1, v_2) + ... + \delta(v_{k-1}, v_k) \)

But as \( (v_i, v_{i+1}) \) is the edge taken:
\( \delta(v_i, v_{i+1}) = w(v_i, v_{i+1}) \)
Correctness

(1) Re-weighting keeps shortest paths

Then by definition of $\hat{\delta}(u, x)$

$$
\hat{\delta}(u, x) = \hat{w}(\text{path})
$$

$$
= \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i)
$$

$$
= \sum_{i=1}^{k} w(v_{i-1}, v_i) + h(v_{i-1} - h(v_i))
$$

$$
= h(v_0) - h(v_k) + \sum_{i=1}^{k} w(v_{i-1}, v_i)
$$

$$
= h(v_0) - h(v_k) + \hat{\delta}(u, x)
$$
Correctness

(1) Re-weighting keeps shortest paths
Thus, the shortest path is just offset by “$h(v_0) - h(v_k)$” (also any path)

As $v_0$ is the start vertex and $v_k$ is the end, so vertices along the path have no influence on $\hat{\delta}(u, x)$ (same path)
Correctness

(2) Re-weighting makes edges $\geq 0$

One of our “relaxation properties” is the “triangle inequality”

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$
$$= 0 \leq \delta(s, u) + w(u, v) - \delta(s, v)$$

how $h$ defined

$$0 \leq w(u, v) + h(u) - h(v) = \hat{w}(u, v)$$
What are two ways you can compute the Fibonacci numbers?

\[ F_n = F_{n-1} + F_{n-2} \]

with \( F_0 = 0 \), \( F_1 = 1 \)

Which way is better?
One way, simply use the definition

Recursive:

\[ F(n): \]
\[
\begin{align*}
& \text{if}(n==1 \text{ or } n==0) \\
& \quad \text{return } n \\
& \text{else return } F(n-1)+F(n-2)
\end{align*}
\]
TL;DR dynamic programming

Another way, compute $F(2)$, then $F(3)$ ... until you get to $F(n)$

Bottom up:

- $A[0] = 0$
- $A[1] = 1$
- for $i = 2$ to $n$
TL;DR dynamic programming

This second way is much faster

It turns out you can take pretty much any recursion and solve it this way (called “dynamic programming”)

It can use a bit more memory, but much faster
How many multiplication operations does it take to compute:

\( x^4 \) ?

\( x^{10} \) ?
How many multiplication operations does it take to compute:

\( x^4 \) \? Answer: 2

\( x^{10} \) \? Answer: 4
Can compute \( x^4 \) with 2 operations:

\[
x^2 = x \times x \quad \text{(store this value)}
\]

\[
x^4 = x^2 \times x^2
\]

Save CPU by using more memory!

Can compute \( x^n \) using \( O(\log n) \) ops

Also true if \( x \) is a matrix
Shortest paths using matrices

Any sub-path \( (p_{x,y}) \) of a shortest path \( (p_{u,v}) \) is also a shortest path.

Thus we can recursively define a shortest path \( p_{0,k} = <v_0, ..., v_k> \), as:

\[
w(p_{0,k}) = \min_{k-1} (w(p_{0,\text{"k-1"}}) + w(\text{"k-1"},k))
\]
Shortest paths using matrices

Thus a shortest path (using less than \( m \) edges) can be defined as:

\[
L^m_{i,j} = \min_k (L^{m-1}_{i,k} + L^1_{k,j}),
\]

where \( L^1 \) is the edge weights matrix.

Can use dynamic programming to find an efficient solution.
Shortest paths using matrices

$L^m$ is not the $m^{th}$ power of $L$, but the operations are very similar:

$$L^m = l^m_{i,j} = \min_k (l^{m-1}_{i,k} + l^1_{k,j}) \quad // \text{ours}$$

$$L^m = l^m_{i,j} = \sum_k ( l^{m-1}_{i,k} * l^1_{k,j} ) \quad // \text{real times}$$

Thus we can use our multiplication saving technique here too!

(see: MatrixAPSPmult.java)
Shortest paths using matrices

All-pairs-shortest-paths(W)
L(1) = W, n = W.rows, m = 1
while m < n
    L(2m) = ESP(L(m), L(m))
    m = 2m
return L(m)

(ESP is L min op on previous slide)
Shortest paths using matrices

Runtime:
\[ |V|^3 \lg |V| \]

Correctness:
By definition (brute force with some computation savers)
Floyd-Warshall

The Floyd-Warshall is similar but uses another shortest path property.

Suppose we have a graph $G$, if we add a single vertex $k$ to get $G'$.

We now need to recompute all shortest paths.
Floyd-Warshall

Either the path goes through $k$, or remains unchanged

\[
d^k_{i,j} = \min (d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j})
\]
Floyd-Warshall

Floyd-Warshall(W) // dynamic prog

d^0_{i,j} = W_{i,j}, n = W.rows

for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            d^k_{i,j} = min (d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j})
Floyd-Warshall

Runtime: \(O(|V|^3)\)

Correctness: Again, by definition of shortest path