CS 341: ALGORITHMS

Lecture 16: graph algorithms V – single source shortest path

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca
DIJKSTRA’S ALGORITHM

Single-source shortest path in a graph with non-negative edge weights
**PROBLEM: SINGLE SOURCE SHORTEST PATHS (SSSP)**

- Input: graph $G = (V, E)$ and a non-negative weight function $w(e)$ defined for every edge $e$

- Problem: for every node $v \neq s$, output a path $s \rightarrow v$ with the smallest total weight (among all paths $s \rightarrow v$)

- I.e., each path $P$ should minimize $w(P) = \sum_{e \in P} w(e)$

And so on... one path for each node.
APPLICATION: **DRIVING DISTANCE TO MANY POSSIBLE DESTINATIONS**

- Single source: from where you are
- **Shortest paths**: to **all** destinations
  - Display a subset of destinations
  - Include the optimal distances computed using SSSP algorithm
- **Other heuristics**… traffic? Lights?
- **Weights** can combine many factors
Game AI: path finding with waypoints

Divide game world into **linear paths**, then send game characters in **straight lines** between waypoints.

Otherwise use BFS to find shortest sequence of waypoints (with **fewest** waypoints).

If some linear paths are much faster/slower, use **weighted** SSSP.
**Dijkstra’s Algorithm**

**Illustrative Example**

Start node $s$ is here

Showing $dist$-values

This $dist$ is **optimal**!

Can we use this optimal $dist$ to **improve** the $dist$ of neighbours

We call this **relaxing** the neighbours

After relaxing, the smallest $dist$ (that we didn’t already know was optimal) is now optimal.

This $dist$ is **optimal**!
Dijkstra(adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    pq = new priority queue

    dist[s] = 0
    for u = 1..n
        pq.enqueue(u, dist[u])

    while pq is not empty
        u = pq.dequeueMin()
        for v in adj[u]
            if dist[u] + w(u,v) < dist[v]
                dist[v] = dist[u] + w(u,v)
                pred[v] = u
                pq.changePriority(v, dist[v])

    return pred, dist

Maintain nodes in priority order, ordered by smallest distance
Enqueue all nodes with distance $\infty$ except for $s$ with distance 0
Eventually dequeue all nodes (no more enqueues)
Each dequeued node $u$ has optimal $\text{dist}$
Relax neighbour $v$
CORRECTNESS: INTUITION

- Dijkstra’s algorithm iteratively construct a set $OPT$ of nodes for which we know the shortest path from $s$ (initially $OPT = \{s\}$).

- After each relaxation step, we grow $OPT$ by adding the node in $V\setminus OPT$ with the smallest $dist$.
**PROOF**

- **Theorem:** At the end of the algorithm, for all $u$, $dist[u]$ is exactly the total weight of the shortest $s \rightarrow u$ path.

  - We prove this in two parts:
    - $dist[u] \leq$ the total weight of the shortest $s \rightarrow u$ path
    - $dist[u] \geq$ the total weight of the shortest $s \rightarrow u$ path
\( \text{dist}[u] \leq \text{SHORTEST } s \mapsto u \text{ PATH} \) [ERICKSON THM.8.5]

- Let \( P \) be any arbitrary \( s \mapsto u \text{ path} \) \( v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_{\ell} \) where \( v_0 = s \) and \( v_{\ell} = u \)
- For any index \( j \) let \( L_j \) denote \( w(v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_j) \)
- We prove by induction: \( \text{dist}[v_j] \leq L_j \) for all \( j \)

\( w(\text{orange edges}) = L_j \)
Prove by induction: \( \forall j : \text{dist}[v_j] \leq L_j \)

Base case: \( \text{dist}[v_0] = \text{dist}[s] = 0 = L_0 \)

Ind. step: suppose \( \forall j>0 : \text{dist}[v_{j-1}] \leq L_{j-1} \)

- When dequeueMin() returns \( v_{j-1} \) we do: we check if \( \text{dist}[v_{j-1}] + w(v_{j-1}, v_j) < \text{dist}[v_j] \)
- If so, we set \( \text{dist}[v_j] = \text{dist}[v_{j-1}] + w(v_{j-1}, v_j) \)
- If not, \( \text{dist}[v_j] \leq \text{dist}[v_{j-1}] + w(v_{j-1}, v_j) \)

In both cases, \( \text{dist}[v_j] \leq \text{dist}[v_{j-1}] + w(v_{j-1}, v_j) \)

- By l.H. \( \text{dist}[v_{j-1}] \leq L_{j-1} \) so \( \text{dist}[v_j] \leq L_{j-1} + w(v_{j-1}, v_j) \)
- And \( L_{j-1} + w(v_{j-1}, v_j) = L_j \) by definition

So \( \text{dist}[v_j] \leq L_j \)
\[ \text{dist}[u] \geq \text{SHORTEST } s \rightarrow u \text{ PATH} \]

- Let \( P' \) be the path \( s \rightarrow \cdots \rightarrow \text{pred}[\text{pred}[u]] \rightarrow \text{pred}[u] \rightarrow u \)
  - i.e., the reverse of following \( \text{pred} \) pointers from \( u \) back to \( s \)
- We show \( \text{dist}[u] \) is as long as this path (and hence as long as the shortest path)
- Denote the nodes in \( P' \) by \( v_0, v_1, \ldots, v_\ell \) where \( v_0 = s \) and \( v_\ell = u \)
- Let \( L_j = w(v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_j) \)
- **Prove by induction:** \( \forall j > 0 : \text{dist}[v_j] = L_j \)
- **Base case:** \( \text{dist}[v_0] = \text{dist}[s] = 0 = L_0 \)
\( \text{dist}[u] \geq \text{SHORTEST } s \rightarrow u \text{ PATH} \)

- \( P' = v_0 \rightarrow \cdots \rightarrow v_\ell = s \rightarrow \cdots \rightarrow \text{pred[pred}[u]] \rightarrow \text{pred}[u] \rightarrow u \)
- \( L_j = w(v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_j) \)
- **Inductive step**: suppose \( \forall j > 0 : \text{dist}[v_{j-1}] = L_{j-1} \)
- When we set \( \text{pred}[v_j] = v_{j-1} \), we set \( \text{dist}[v_j] = \text{dist}[v_{j-1}] + w(v_{j-1}, v_j) \)

Recall:

\[
\begin{align*}
\text{if dist}[u] + w(u,v) &< \text{dist}[v] \\
\text{dist}[v] &\leq \text{dist}[u] + w(u,v) \\
\text{pred}[v] &\leq u
\end{align*}
\]

- By I.H., \( \text{dist}[v_j] = L_{j-1} + w(v_{j-1}, v_j) \)
- By definition \( L_j = L_{j-1} + w(v_{j-1}, v_j) \)
- So \( \text{dist}[v_j] = L_j \)

So \( \text{dist}[u] \geq \text{length of shortest } s \rightarrow u \text{ path} \)

That means it’s **equal** to the length of the shortest path!
Dijkstra(adj[1..n], s)
    pred[1..n] = [null, null, ..., null]
    dist[1..n] = [infty, infty, ..., infty]
    pq = new priority queue
    dist[s] = 0
    for u = 1..n
        pq.enqueue(u, dist[u])
    while pq is not empty
        u = pq.dequeueMin()
        for v in adj[u]
            if dist[u] + w(u,v) < dist[v]
                dist[v] = dist[u] + w(u,v)
                pred[v] = u
                pq.changePriority(v, dist[v])
    return pred, dist

Space complexity?

RUNTIME

• Each node enqueued and dequeueMin’d once
  • $O(n \log n)$

• For each dequeueMin, do $O(\log n)$ per neighbour
  • $O(\log n)$ for each edge
  • $O(m \log n)$ w/adjacency lists

• Total time $O((n + m) \log n)$
OUTPUTTING ACTUAL SHORTEST PATH(S)?

• To compute the actual shortest path $s \rightarrow t$
• Inspect $pred[t]$
  • If it is NULL, there is no such path
  • Otherwise, follow $pred$ pointers back to $s$, and return the reverse of that path
AN ALTERNATIVE IMPLEMENTATION

- Instead of using a priority queue
- Find the minimum \( \text{dist}[\cdot] \) node to add to \( \text{OPT} \) via \textbf{linear search}

- Runtime?
  - \( O(n^2) \)
  - Better or worse than \( O((n + m) \log n) \)?
WEBSITE DEMONSTRATING DIJKSTRA’S ALG

• https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html
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.
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).

```
BellmanFord(n, E[1..m], s)

pred[1..n] = new array filled with null
D[1..n] = new array filled with infinity
D[s] = 0
for i = 1..n
    for (u,v,w) in E
        if D[u] + w < D[v]
            D[v] = D[u] + w
            pred[v] = u
return (D, pred)
```

Could be \( O(n^3) \)

Total \( O(nm) \)
Edges happen to be processed left to right by the inner loop.

It technically suffices to do one iteration of the outer loop.

```python
BellmanFord(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
    return (D, pred)
```
Edges happen to be processed right to left by the inner loop.

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

Since the longest possible path without a cycle can be $n - 1$ edges, the edges must be scanned $n - 1$ times to ensure the shortest path has been found for all nodes.

Need $n$ iterations of outer loop.
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 $n - 1$ iterations, if $\exists$ path $P$ with at most $n - 1$ 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 $u$

Of course every simple path has at most $n - 1$ edges

So what if we do **another iteration**, and some $D[u]$ improves?

There is a negative cycle!
```python
BellmanFordCheck(n, E[1..m], s)
    pred[1..n] = new array filled with null
    D[1..n] = new array filled with infinity
    D[s] = 0
    for i = 1..n
        changed = false
        for (u,v,w) in E
            if D[u] + w < D[v]
                D[v] = D[u] + w
                pred[v] = u
                changed = true
        if not changed
            exit loop
    if i == n // assert: changed == true
        return NEGATIVE_CYCLE
    return (D, pred)
```