1 Weighted Version of Shortest Paths

- BFS solves unweighted shortest path problem
- Every edge traversed adds one to path length
- What if edges have nonuniform weights? Let \( w(u, v) \) be weight of edge \((u, v)\)

Some intuition...

- BFS finds closest set of vertices \((d = 1)\) to source, then next closest set \((d = 2)\), and so on
- IOW, repeatedly process vertices closest to source.
- Tricky part was proving that every vertex is reached via a shortest path.
- Can we use the same idea in weighted case?

2 Dijkstra’s Algorithm

Here is an algorithm that works when \( w(u, v) \geq 0 \) for all edges \((u, v)\).

- Uses min-first priority queue \( Q \) of vertices
- key is estimated distance from \( s \) to each vertex
- Initially, \( Q \) contains all vertices in \( G \), but distances are unknown \((\infty)\)
- Repeatedly extract vertex \( x \) that is closest to \( s \)
- As in BFS, knowing distance from \( s \) to \( x \) tells us something about distance to \( x \)’s neighbors.
- Decrease the key of every vertex \( y \) in Adj[\( x \)] to at most \( d(x) + w(x, y) \).

Example
3 Pseudocode

Given graph $G = (V, E)$, starting vertex $s$. (Note: does not show handle manipulation)

\begin{verbatim}
Dijkstra(G, s)
    for u ∈ V do ▷ initialize
        u.distance ← ∞
        u.parent ← null
        Q.insert(u, ∞)

    s.distance ← 0
    Q.decreaseKey(s, 0)

    while Q is not empty do
        u ← Q.extractMin()
        if u.distance = ∞ ▷ cannot reach any more vertices from s
            stop

        for v ∈ Adj[u] do
            if Q.decreaseKey(v, u.distance + w(u, v))
                v.distance ← u.distance + w(u, v)
                v.parent ← u
\end{verbatim}

4 Running Time

- cost dominated by priority queue ops (queue size $n$)
- initialization: one insert per vertex ($n$)
- outer loop: one extractMin per vertex ($n$)
- inner loop: one decreaseKey per edge out of each $u$ ($m$)

Hence, can write

$$T(m, n) = nT_{\text{insert}}(n) + nT_{\text{extractMin}}(n) + mT_{\text{decreaseKey}}(n)$$

- Cost depends on priority queue implementation!
- For binary heaps, all queue ops are $O(\log n)$, so
  $$T(m, n) = (2n + m)O(\log n) = O(m \log n)$$
- For a Fibonacci heap, insert and decreaseKey are amortized $O(1)$
- Hence, revised run time would be
  $$T(m, n) = nO(1) + nO(\log n) + mO(1) = O(n \log n + m)$$
- Is this an improvement? Yes, if graph is dense.
5 Correctness

As before, we need to show that every vertex receives its correct shortest-path distance from \( s \). Note that \( u \).distance never changes after \( u \) is removed from the priority queue.

**Theorem**: when vertex \( u \) is removed from the queue, \( u \).distance is length of a shortest path from \( s \) to \( u \).

- Proceed by induction on order of removal from queue.
- **Bas**: \( s \) is removed first from queue, and it has correct distance 0.
- **Ind**: Assume that vertex \( u \) is next to be dequeued, but it does not have its shortest-path distance.
- Consider a shortest path \( p \) connecting \( s \) to \( u \).

- \( s \) has been dequeued and \( u \) has not, so there is some last vertex \( x \) on this path that *has* already been dequeued.
- By IH, \( x \) has its correct shortest-path distance.
- Let \( y \) be \( x \)'s successor on path \( p \) (which has not been dequeued yet), and let \( p' \) be the prefix of \( p \) connecting \( s \) to \( y \).
- **Prefix \( p' \) is shortest path from \( s \) to \( y \)**. Otherwise, could replace it with a shorter path \( p'' \), which would give a shorter path than \( p \) from \( s \) to \( u \).
- Hence, \( y \) received its correct shortest-path distance when \( x \) was processed, since edge \( x \rightarrow y \) was explored.
- To finish up, two possibilities:
  1. If \( y = u \), then \( u \) has its correct shortest-path distance, which contradicts our assumption that this distance is wrong.
  2. If \( y \) precedes \( u \), then \( y \)'s shortest-path distance is \( \leq u \)'s shortest-path distance. Hence, \( y \)'s s-p distance is strictly less than \( u \)'s current (non-s-p) distance. Conclude that \( y \) will be dequeued before \( u \), which contradicts our assumption that \( u \) is next vertex to be dequeued.
- Conclude that \( u \) must have its correct shortest-path distance. QED
6 Other Ways to Get Shortest Paths

Remember, Dijkstra’s algorithm has an important limitation!

- Requires that \( w(u, v) \geq 0 \) for all edges \((u, v)\)
- **Problem:** assumes that no prefix of a path \( p \) can have length > \( p \).
- If edge weights can be negative, this assumption is violated.
- Hence, can end up dequeueing a vertex before path of least total weight is found.

- How could this happen? “Shortest” path could be measured in terms other than distance.
- For example, suppose that on each edge \((u, v)\) you may be charged a fee \( (w(u, v) > 0) \) or paid a bonus \( (w(u, v) < 0) \). Goal is to find path with smallest total cost!
- In this case, you want an algorithm that deals with negative-weight edges.
- **Bellman-Ford** algorithm can do it.
- Also can detect cycles of negative weight (causes paths with arbitrarily low weight, so no “shortest”).
- Cost is \( O(nm) \), which is worse than Dijkstra in general.
- **Special case:** if graph is a DAG, can reduce cost to \( \Theta(m + n) \).
- Finally, suppose you need to know the shortest paths from ALL vertices to ALL vertices in \( G \).
- If
  - you can negative-weight edges;
  - you cannot have negative-weight cycles (use Bellman-Ford on an augmented version of \( G \) to check!)

  there is a \( \Theta(n^3) \) algorithm for this problem due to Floyd and Warshall. Unless your graph is sparse, this is asymptotically faster than running Bellman-Ford once per starting vertex.
- Another algorithm for the same problem, due to Johnson, takes time \( O(n^2 \log n + nm) \) when implemented with a Fibonacci heap – same as Floyd-Warshall for dense graphs, but faster for non-dense graphs.