Algorithm Idea

- **input:** a graph G=(V,E) and a start vertex s
- **goal:** determine the length of the shortest path from s to every vertex in V
  (and while you're at it, compute the path itself.)

Algorithm Details

```plaintext
unweightedSP (Vertex s)
    • Let S be the set of "explored" nodes, initially S = ∅
    • for each node v (incl. s) set v.prev = null, v.dist = ∞
    • initialize queue Q to contain s, set s.dist = 0
    • while Q is non-empty
        • let v be the front element of Q
        • add v to S (mark it as "explored")
        • for each of v's unexplored_nbrs w
            • if w not already in Q
                • set w.prev = v
                • set w.dist   = v.dist + 1
                • add w to Q
    printPath (Vertex w)
        • if w.dist == ∞, print "unreachable"
        • else if (w.prev != null)
            • printPath (w.prev)
            • print " ---> w"
```

RT: (vague) for loop is O(n)
while loop is O(n)
therefore O(n^2)

RT: (precise) for loop is O(deg(v))
and so O(m+n)

\[ \sum_{v \in V} \text{deg}(v) = 2 |E| \]

RT: just O(n)

S = ∅
Let $S$ be the set of "explored" nodes, initially
for each node $v$ (incl. $s$) set $v$.pred = null, $v$.dist = $\infty$
initialize queue $Q$ to contain $s$, set $s$.dist = 0
while $Q$ is non-empty
  let $v$ be the front element of $Q$
  add $v$ to $S$ (mark it as "explored")
  for each of $v$'s unexplored nbrs $w$
    if $w$ not already in $Q$
      add $w$ to $Q$
      set $w$.prev = $v$
      set $w$.dist = $v$.dist + 1

Example

Proof of Correctness

Turns out this algorithm is a special case of Dijkstra's algorithm. We'll prove

Dijkstra's later and then will suffice as a proof for BFS, too.

Example
**Connectivity**

- **isConnected( )**
  - if `G` is empty, return true
  - otherwise, pick some arbitrary vertex `s` and call `unweightedSP(s)`
  - if any vertex `v` has `v.dist = \infty` then return false; else return true

- **numComponents( )**
  - while there is some vertex `v` that is still unmarked
    - `numComponents++`
    - run bfs/unweightedSP from `v`

  **RT: O(m+n)**

  ... depends on how quickly you can find an unmarked vertex

- **isAcyclic( )**
  - for each connected component
    - run bfs from some unmarked vertex `s` in the current component
    - if, during bfs, one of `v`'s neighbours is already marked, return false
  - return true

**Q:** What if we used dfs (stack) instead of bfs (queue) in all of these methods?

---

**Positive) Weighted Shortest Paths**

**Input:** a graph `G=(V,E)` with **positive** edge costs and a start vertex `s`

**Goal:** determine the cost of the least-cost path from `s` to every vertex in `V`

**Q:** will `unweightedSP` (aka bfs) still work?

**A:** sadly, no

- need to adjust the order in which we remove vertices from queue
- rather than removing first-in, want to remove closest-to-`s`
- we'll need to adjust `w.dist` many times, as we find shorter distances

**idea: weightedSP( )**

- explore vertices out of `s` at a uniform rate
- think of edge costs as lengths of pipe, and follow water flowing out of `s`
- we'll explore vertices in the order the water hits them

**RT:**

- `O(\text{unweightedSP}(\text{data}))` will still work
- `O(\text{unweightedSP}(\text{data}))` will determine the cost of the least-cost path from `s` to every vertex in `V`

**Input:** a graph `G=(V,E)` with **positive** edge costs `\geq 0` and a start vertex `s`

---

**Dijkstra's Algorithm**

**weightedSP( Vertex s ) a.k.a. DIJKSTRA'S ALGORITHM (die-kstra)**

**RT:** the loop is `O(\text{deg}(v) \cdot \text{RT of decreaseKey}) = O(\text{deg}(v) \cdot \ln n)` and `\text{getMin}` is `O(\ln n)`. We do this at most once per vertex. Total RT is thus

**S = \emptyset \sum_{v \in V} (\ln n + \text{deg}(v) \cdot \ln n) \in O(n + 2m \ln n) \in O((n + m) \ln n)$$

---

**Connectivity**