CS 151
Graph Algorithms
Unweighted Shortest Paths Problem

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.)

Each vertex $v$ will have dist and prev fields.
$v$.dist is length of s.p. from $s$ to $v$, $v$.prev is Vertex preceding $v$ on this path.
Ex: unweightedSP( A ) will determine the s.p. from A to B, C, and D.
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.)

idea: unweightedSP($s$)
  • initialize $s$.dist = 0 and $s$.pred = null
  • for each neighbour $v$ of $s$, set $v$.dist = 1 and $v$.pred = $s$
  • for each as-of-yet-unexplored neighbour $w$ of some neighbour $v$ of $s$
    • set $w$.dist = $v$.dist + 1 = 2 and $w$.pred = $v$
  • repeat this process, exploring the nodes outwards in “concentric circles”

Does this remind you of any algorithm we’ve seen before?
It is just BFS (breadth-first search), but on a graph instead of a tree!

Questions to Address:
  • how will we keep track of explored vertices and who is “next in line”?
  • what happens if $G$ is not connected? (how does it end?)
  • does this algorithm actually find the shortest path? (does it work?)
unweightedSP( Vertex s )

- Let S be the set of “explored” nodes, initially \( S = \emptyset \)
- for each node \( v \) (incl. \( s \)) set \( v\).pred = null, \( v\).dist = \text{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 \( Q \)
      - set \( w\).prev = \( v \)
      - set \( w\).dist = \( v\).dist + 1
      - add \( w \) to \( Q \)

printPath( Vertex w )

- if \( w\).dist == \text{infty}, print “unreachable”
- else if (\( w\).prev != null)
  - printPath( \( w\).prev )
  - print “ \( \rightarrow \) w “

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

RT: (precise) for loop is \( O(\text{deg}(v)) \)
and
\[
\sum_{v \in V} \text{deg}(v) = 2|E|
\]
so \( O(m+n) \)

RT: just \( O(n) \)
Let \( S \) be the set of “explored” nodes, initially \( S = \emptyset \).

- for each node \( v \) (incl. \( s \)) set \( v.\text{pred} = \text{null}, \) \( v.\text{dist} = \text{infty} \)
- initialize queue \( Q \) to contain \( s \), set \( s.\text{dist} = 0 \)
- 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 \( w.\text{prev} = v \)
      - set \( w.\text{dist} = v.\text{dist} + 1 \)
Let $S$ be the set of “explored” nodes, initially $S = \emptyset$.

For each node $v$ (incl. $s$) set $v.\text{pred} = \text{null}$, $v.\text{dist} = \text{infty}$.

Initialize queue $Q$ to contain $s$, set $s.\text{dist} = 0$.

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 $w.\text{prev} = v$
    - Set $w.\text{dist} = v.\text{dist} + 1$.
Let $S$ be the set of “explored” nodes, initially $S = \emptyset$
for each node $v$ (incl. $s$) set $v.pre = \text{null}$, $v.dist = \infty$
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 $w.pre = v$
      • set $w.dist = v.dist + 1$
Proof of Correctness

Turns out this algorithm is a special case of Dijkstra’s algorithm; we’ll prove Dijkstra’s later and that will suffice as a proof for BFS, too.
boolean 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

RT: O(m+n)

int numComponents( )
  • while there is some vertex v that is still unmarked
    • numComponents++
    • run bfs/unweightedSP from v

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

boolean 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 $c_e > 0$ and a start vertex $s$

goal: determine the cost of the least-cost path from $s$ to every vertex in $V$
(and while you’re at it, compute the path itself.)

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( s )
• 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
• once we explore a vertex, we’ve found its shortest path
• as we explore each vertex $v$, we can see new path to its neighbors that
  may or may not be better than the paths we already know about
Let $S$ be the set of “explored” nodes, initially $S = \emptyset$.

For each node $v$ (incl. $s$) set $v.\text{pred} = \text{null}$, $v.\text{dist} = \text{infty}$.

Initialize priority queue $Q$ to contain $s$, set $s.\text{dist} = 0$.

While $Q$ is non-empty,

let $v$ be the min element of $Q$ (v has min dist of all unexplored vertices).

Add $v$ to $S$ (mark it as “explored”).

For each of $v$’s unexplored nbrs $w$,

if $v.\text{dist} + \text{cost}(v,w) < w.\text{dist}$

set $w.\text{prev} = v$

set $w.\text{dist} = v.\text{dist} + \text{cost}(v,w)$

if $w$ not in $Q$ then add $w$ to $Q$

else, $Q.\text{decreaseKey}(w)$ (b/c $w$’s dist has decreased)

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

$$\sum_{v \in V} (\log n + \text{deg}(v) \cdot \log n) \leq (n + 2m) \log n \in O((n + m) \log n)$$