

Dijkstra's Algorithm

Edsger Dijkstra — 1956

Defn A weighted (directed or undirected) graph is a graph where each edge is assigned a weight, denoted $w(u, v)$ or w_{uv} . The weight of a path is the sum of the weights on edges in the path.

For now, we will consider weights in $\mathbb{R}_{\geq 0}$ (nonnegative real #s).

Dijkstra's algorithm solves the single-source shortest paths (SSSP) problem — find shortest paths from single vertex s to every other vertex.

BFS

layer/level map

visited vertices

queue of pending vertices

parent map

Dijkstra

distance map

Same

Priority.

Same.

BASIC DIJKSTRA(G, s):

$d \leftarrow$ empty dictionary (distance map)
 $parent \leftarrow$ empty dict (parent map)

Mark s VISITED

$d[s] \leftarrow 0$

While not all vertices are VISITED:

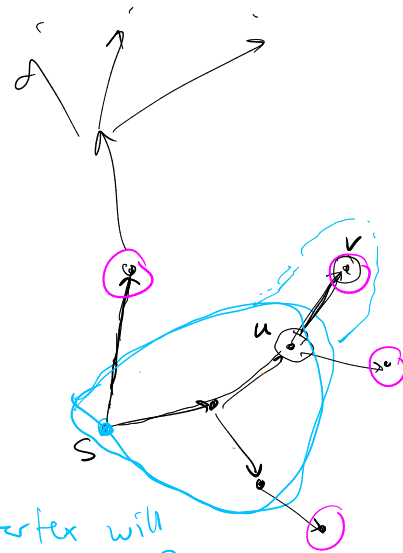
Pick VISITED u & UNVISITED v
 such that $d[u] + w_{uv}$ is as small
 as possible. ie which vertex will
 water reach next?

$d[v] \leftarrow d[u] + w_{uv}$

$parent[v] \leftarrow u$

Mark v VISITED.

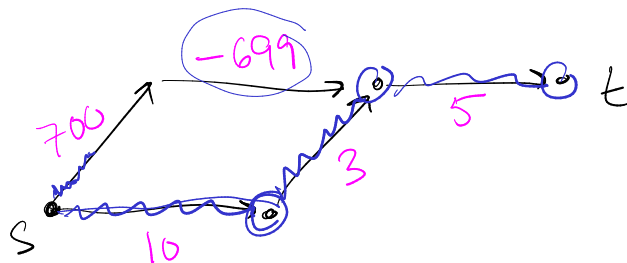
return $d, parent$



Total: $\Theta(1) + O(VE)$
 $= O(VE)$.

Note, E is $O(V^2)$. So this
 is also $O(V^3)$.

Thm. Dijkstra's algorithm correctly solves the SSSP
 problem for any graph with nonnegative edge weights.

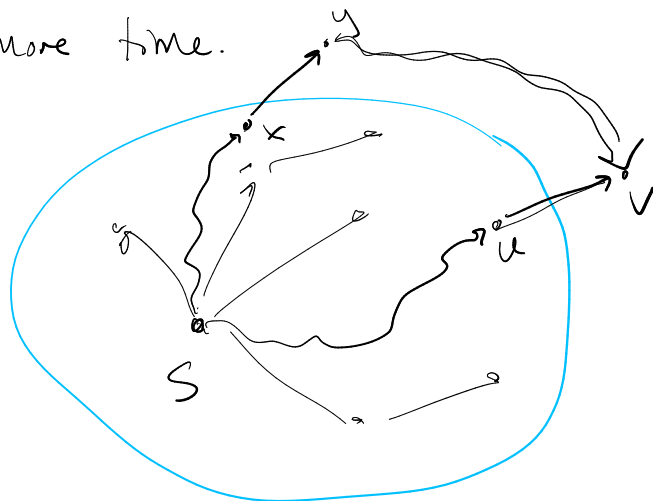


Dijkstra
 does not work
 when there are
 negative weight
 edges (because it is
 greedy).

Proof. We use as a loop invariant the
 proposition that for every visited vertex v ,
 $d[v]$ is the correct shortest distance $s \rightarrow v$.

- Before the loop starts, s is the only visited
 vertex, and $d[s] = 0$ is correct because
 there are no negative weight edges.

- Now suppose the invariant holds and we run the loop one more time.



Suppose the code in the loop picks u, v .

Suppose there is some other path $s \rightarrow v$, which crosses from an unvisited vertex x to visited vertex y somewhere along the path. By assumption, $d[x]$ and $d[u]$ are correct shortest distances to x and u .

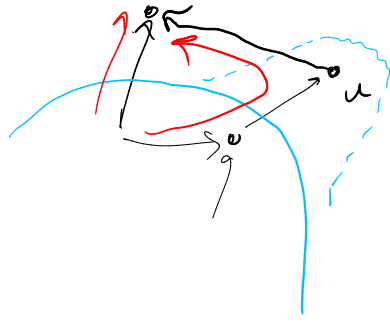
We chose u, v so that $d[u] + w_{uv} \leq d[x] + w_{xy}$

Because all edges are ≥ 0 , the extra part of the path $y \rightarrow v$ can only make it longer. Hence $d[u] + w_{uv}$ is indeed the shortest distance $s \rightarrow v$.

Idea: use a priority queue to keep track of "boundary" or "fringe" vertices + pick the next one w/ smallest distance.

- Generalize d to store shortest known distance so far for boundary vertices
- Similarly parent will store best known parent.

Dijkstra(G, s):



$\Theta(1)$

$d \leftarrow$ empty dict
parent $\leftarrow$ empty dict.

$d[s] \leftarrow 0$

$d[v] \leftarrow \infty$ for all other v

$\Theta(V)$

$Q \leftarrow$ priority queue w/ all vertices, using $d[v]$ as key.

while Q not empty:

$u \leftarrow Q.\text{remove Min}$

$T_{rem}(V)$

for each neighbor v of u :

if $d[u] + w_{uv} < d[v]$:

$\Theta(1)$

$\underline{d[v]} \leftarrow \underline{d[u]} + \underline{w_{uv}}$

parent[v] $\leftarrow u$.

$T_{upd}(V)$

$Q.\text{update Key}(v, d[v])$

$O(E)$
iterations
total

return d , parent.

Overall, $\Theta(1) + \Theta(V) + O(V \cdot T_{rem}(V))$

$+ O(E \cdot T_{upd}(V))$

$= O(V \cdot T_{rem}(V) + E \cdot T_{upd}(V))$

$$O(V \cdot T_{\text{rem}}(V) + E \cdot T_{\text{upd}}(V))$$

Priority queue implementations

<u>Impl</u>	<u>T_{rem}</u>	<u>T_{upd}</u>	<u>Dijkstra</u>
Array of pairs	$O(n)$	$O(n)$	$O(V^2 + EV) = O(EV)$
Sorted array	$O(1)$	$O(n)$	$O(V + EV) = O(EV)$
<u>Binary heap</u>	$O(\lg n)$	$O(\lg n)$	$O(V \cdot \lg V + E \cdot \lg V)$ $= O(E \lg V)$
DST vertices → dst	$O(n)$	$O(1)$	$O(V^2 + E)$ $= O(V^2)$
Fibonacci heap	$O(\lg n)$	$O(1)$	$O(V \lg V + E)$