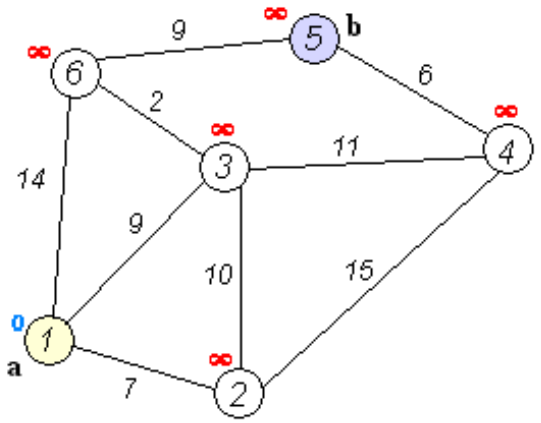
Djikstra's Algorithm

(as in 'dike-stra')

What's the shortest path between nodes in a graph?

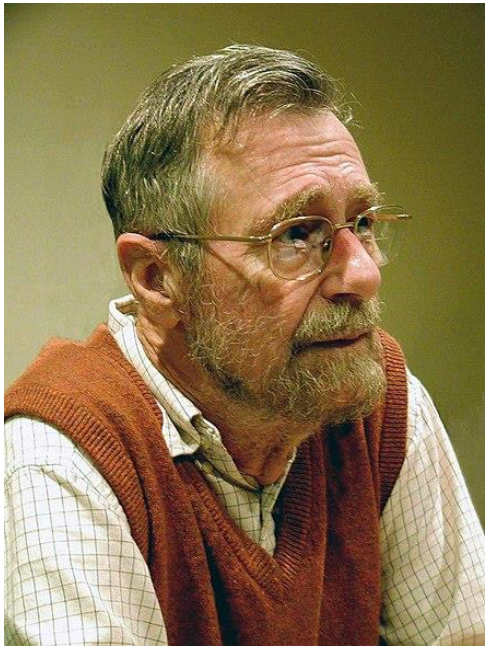


Kiel Williams, University of Illinois Dept. of Physics

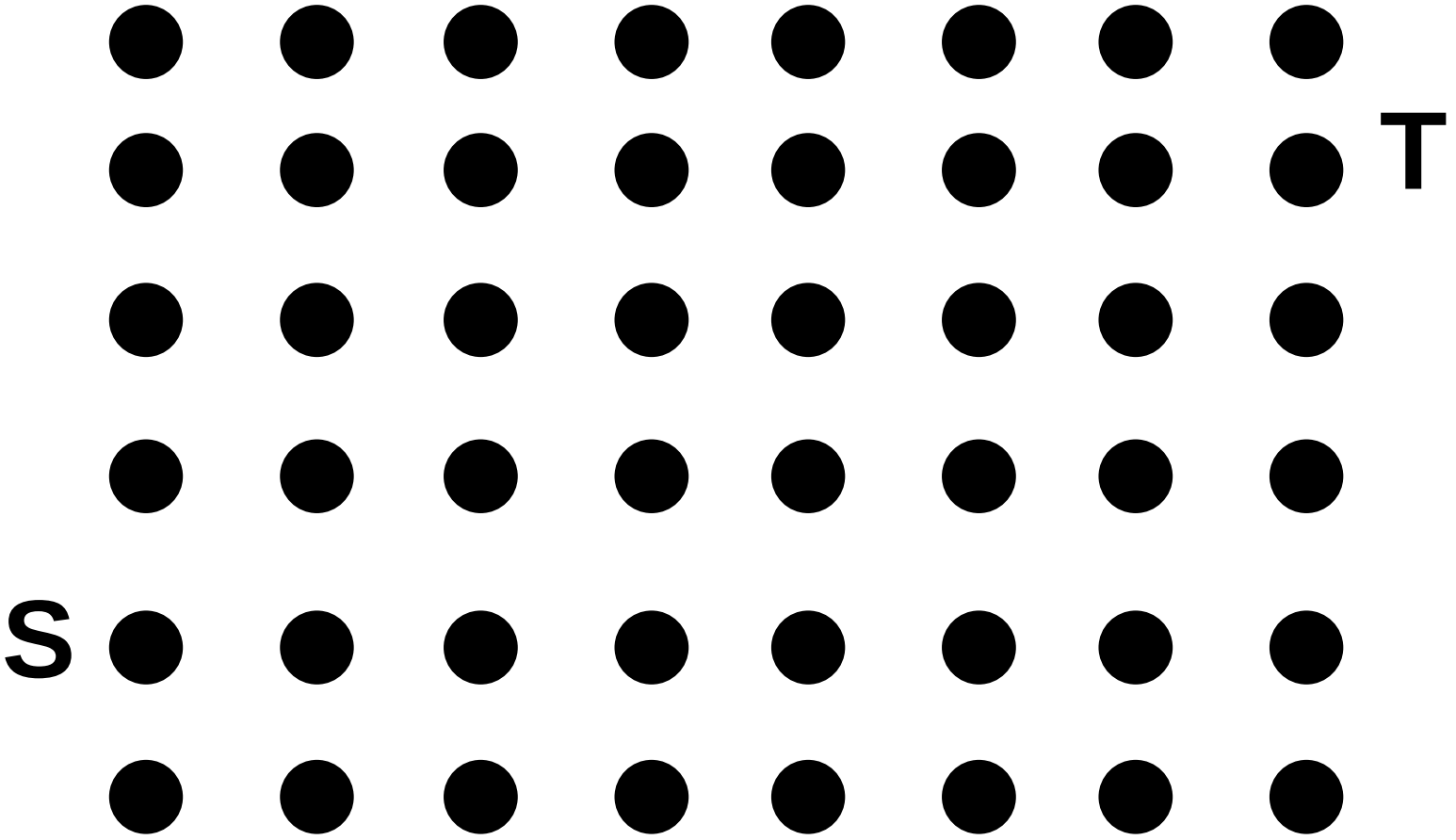
02/08/2018 Algorithm Interest Group Meeting

“*I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to a formal completion only, with a minimum of effort, and to become....., yes what? A programmer? But was that a respectable profession? For after all, what was programming?*”

- Edsger Djikstra (reminiscing about ~1951)

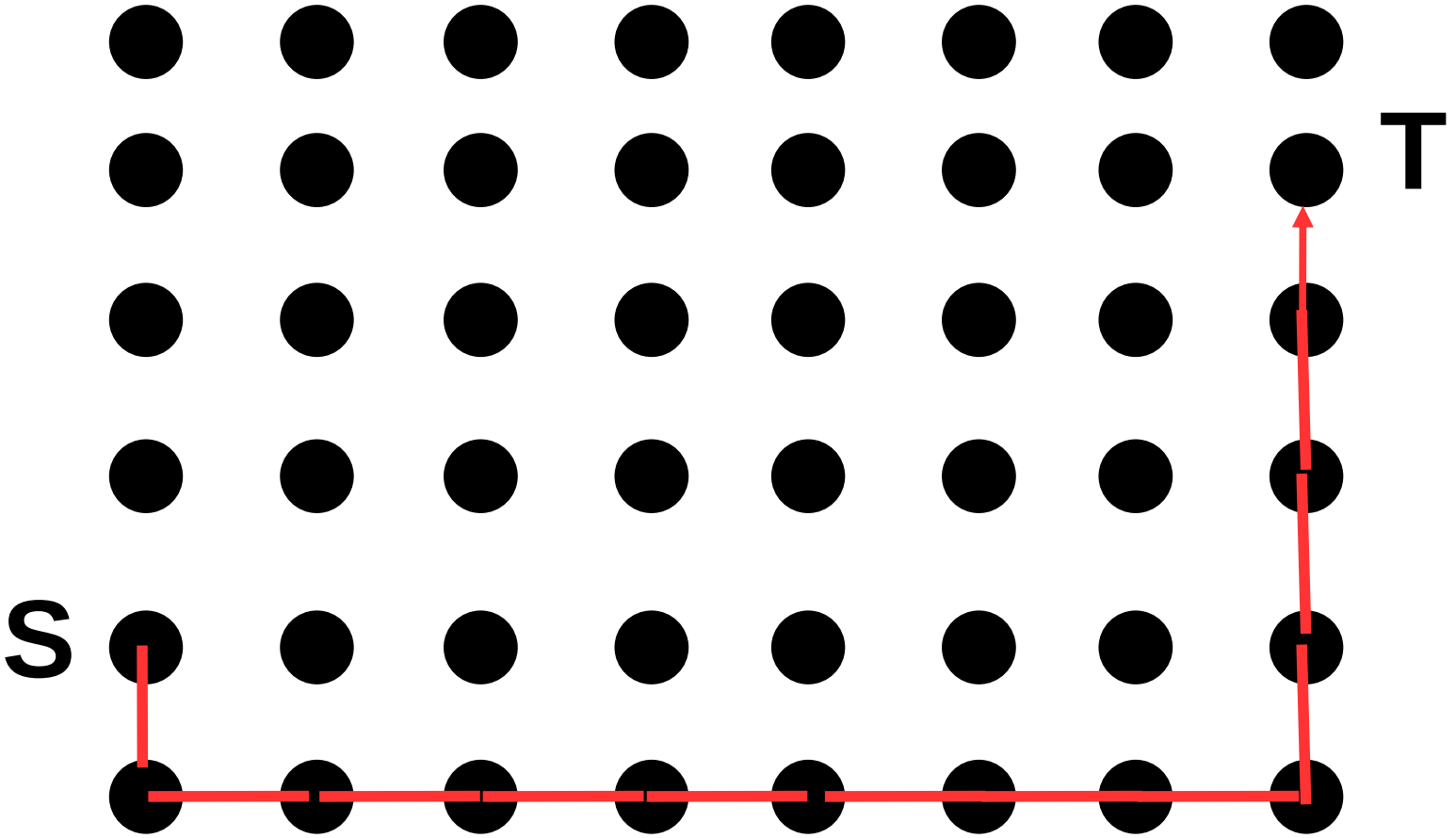


Given a “source node” (**S**) and a “target node” (**T**) in a graph of nodes, how long is the shortest path connecting them?



Given a “source node” (**S**) and a “target node” (**T**) in a graph of nodes, how long is the shortest path connecting them?

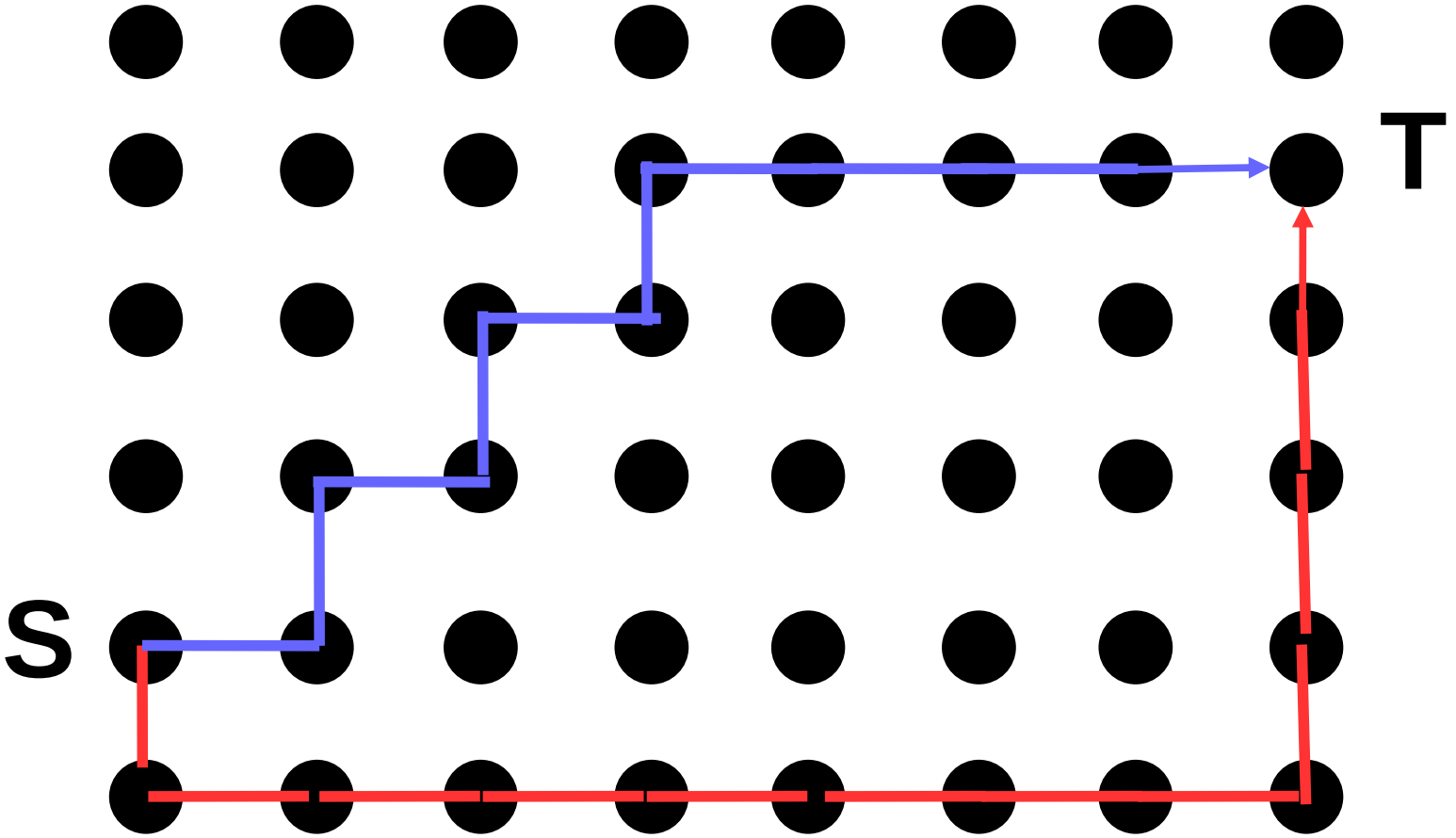
Is this the optimal path?



Given a “source node” (**S**) and a “target node” (**T**) in a graph of nodes, how long is the shortest path connecting them?

Is this the optimal path?

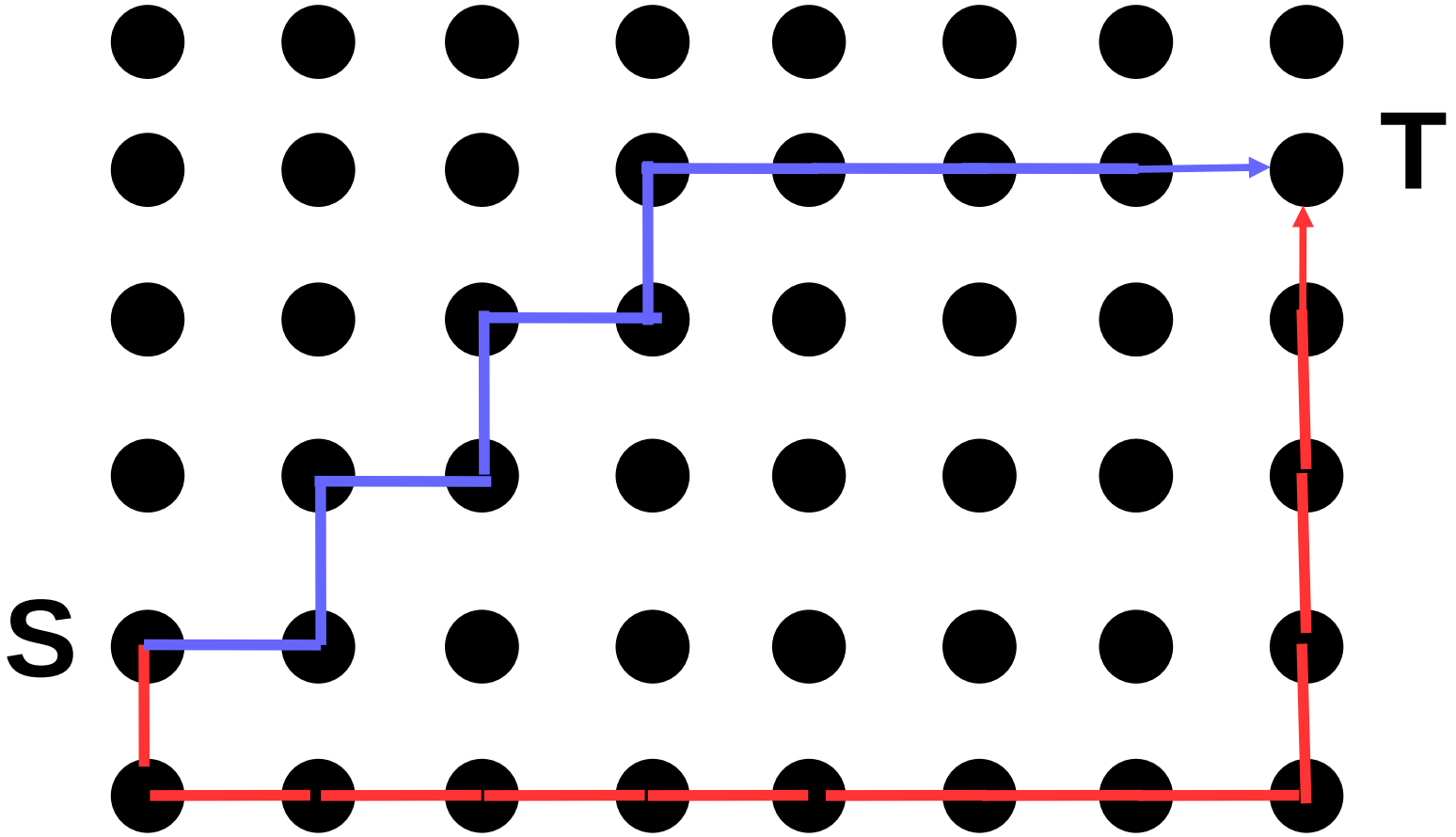
How about this?



Given a “source node” (**S**) and a “target node” (**T**) in a graph of nodes, how long is the shortest path connecting them?

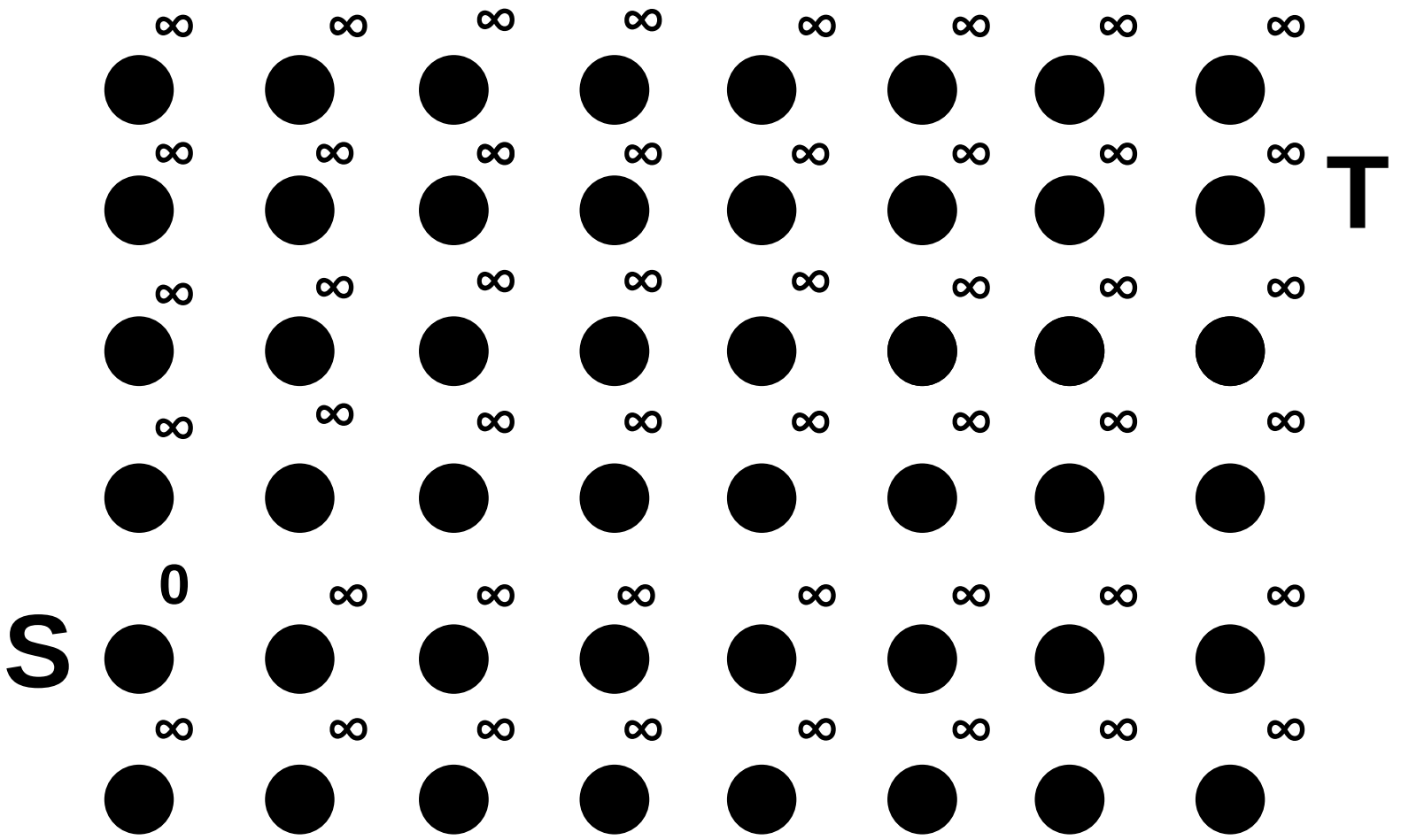
Is this the optimal path?

How about this?

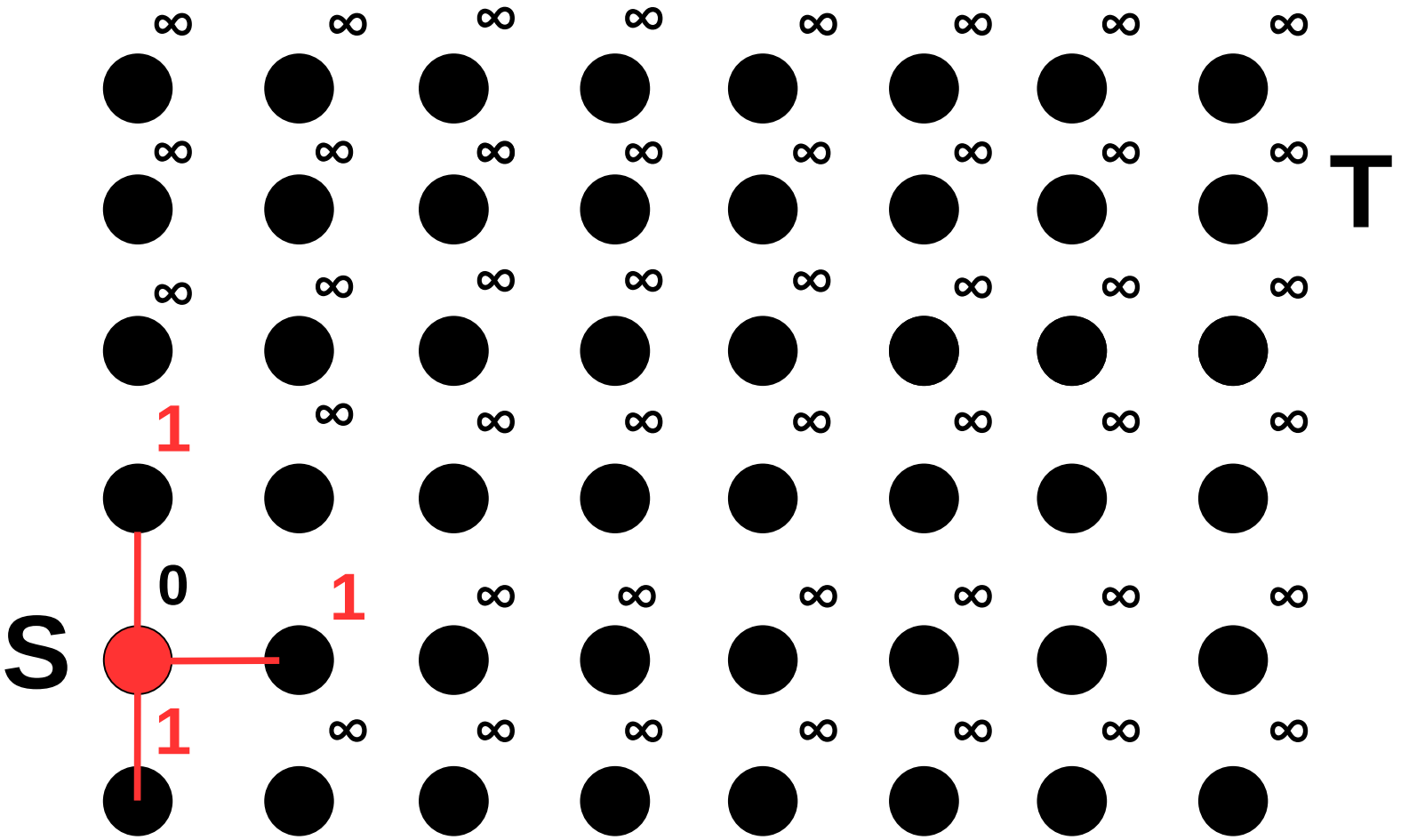


The solution: Dijkstra's algorithm

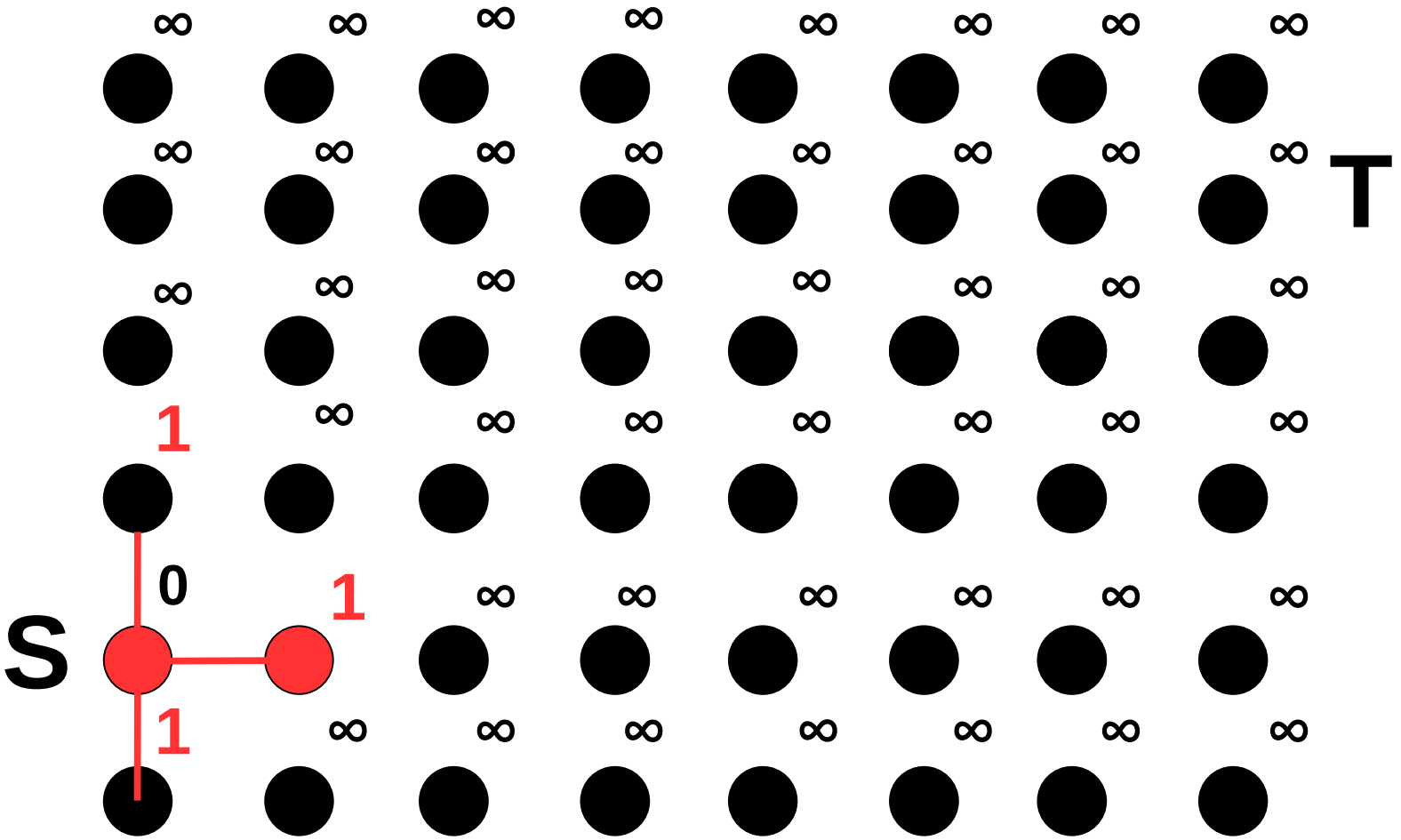
Step 1: Assume each node is "infinitely far" from the source node (source node is distance-0 from itself)



Step 2: Starting on the source node, check the distance from the neighbors of the current node to the source node

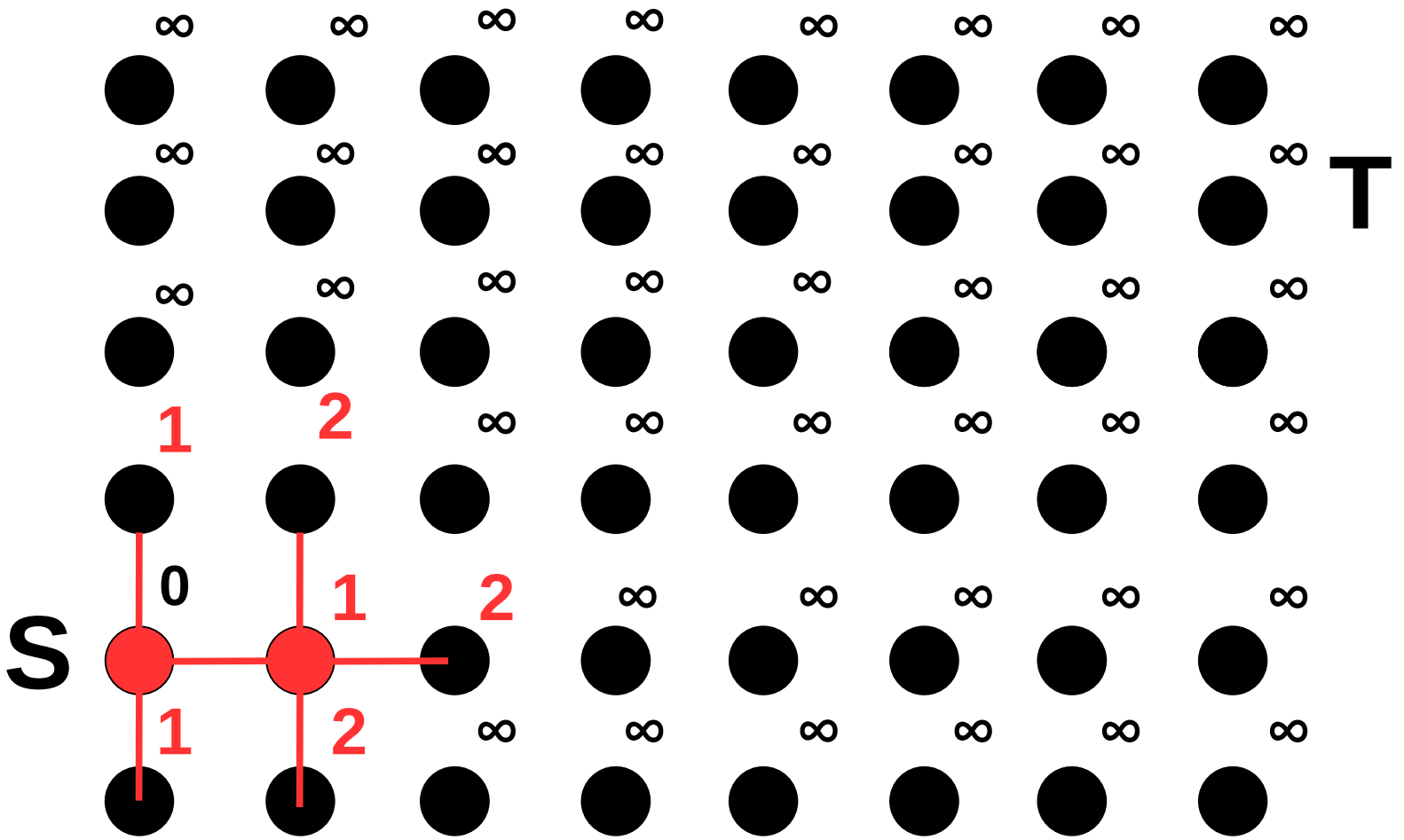


Step 3: Among unvisited (black) nodes, pick one with the lowest “tentative distance” as the next node

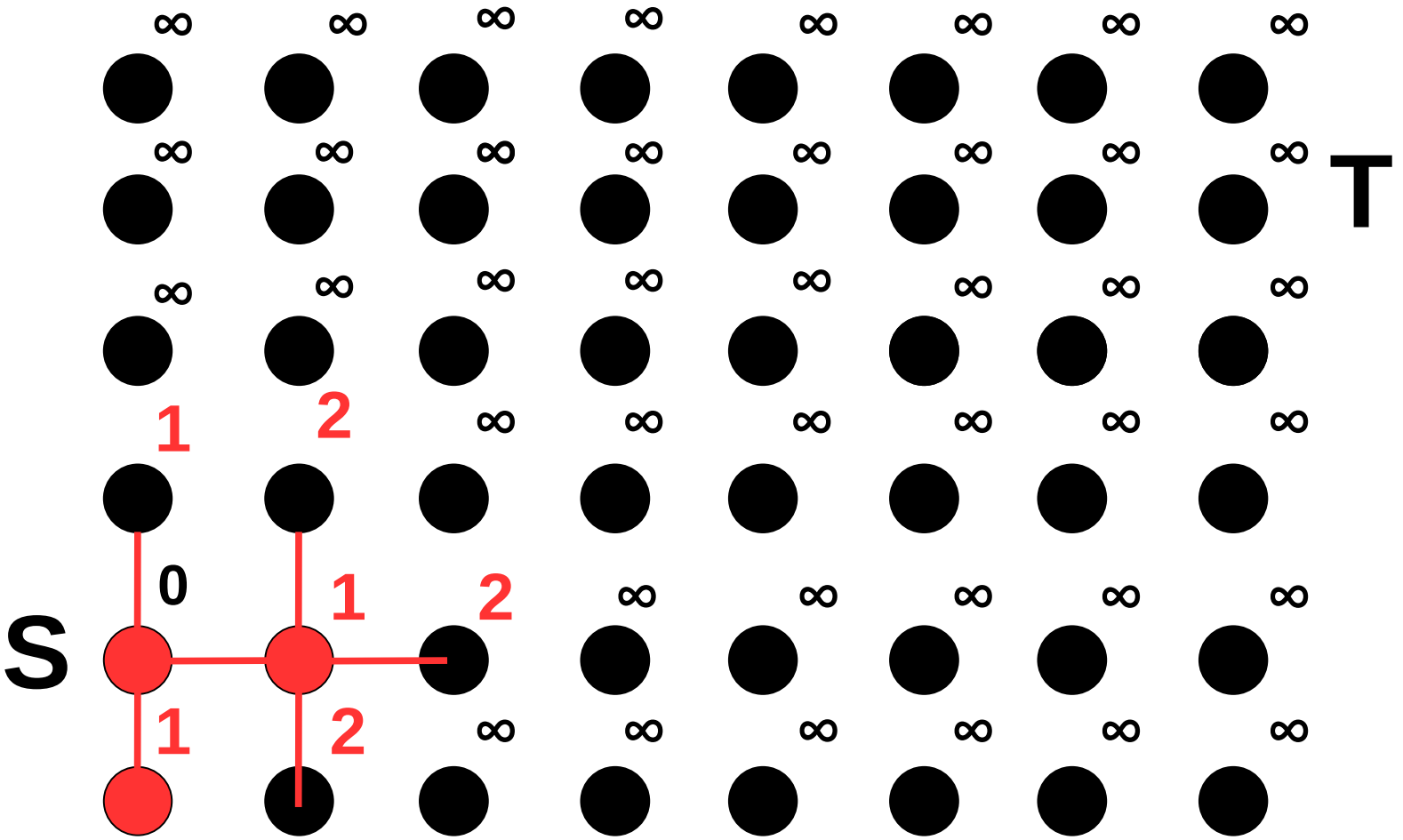


Step 4: Among unvisited (black) neighbors of current node, assign more "tentative distances"

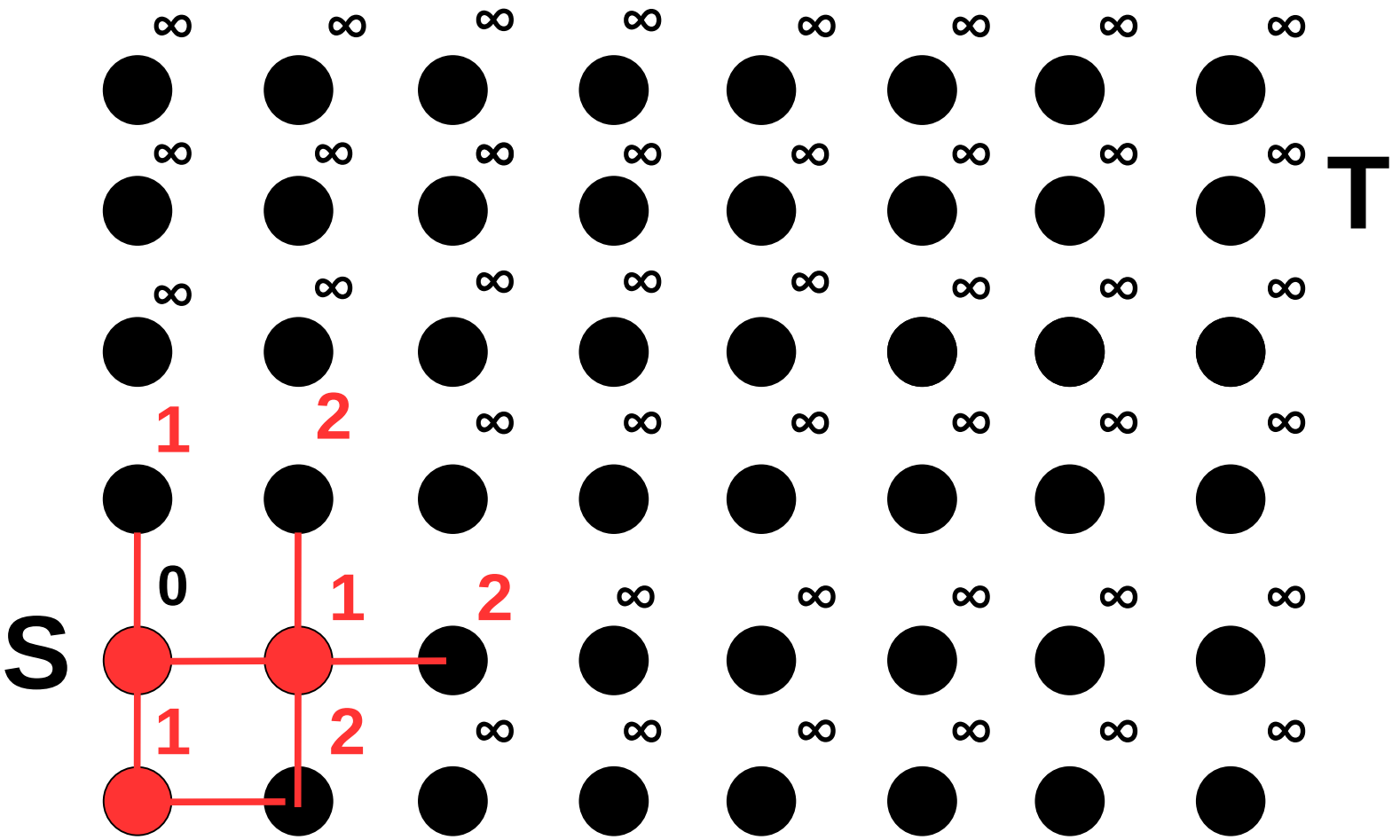
Tentative Distance = (Tentative Distance of Current Node) + (Distance from Current Node to Neighbor Node)



Step 5: Among unvisited (black) nodes, pick another one with minimum “tentative distance” as new current node...

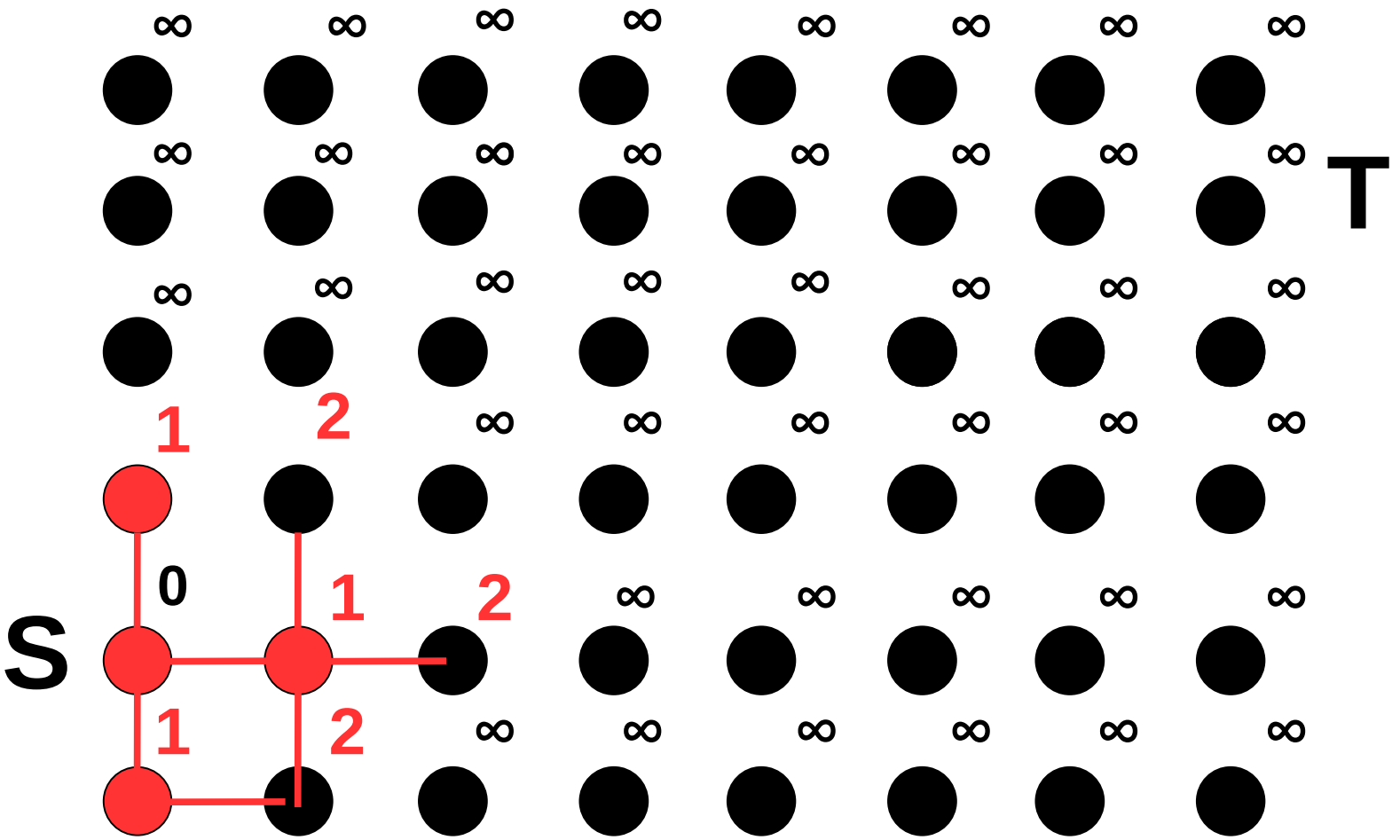


Step 6: Check the unvisited (black) neighbor nodes, and assign tentative distances...



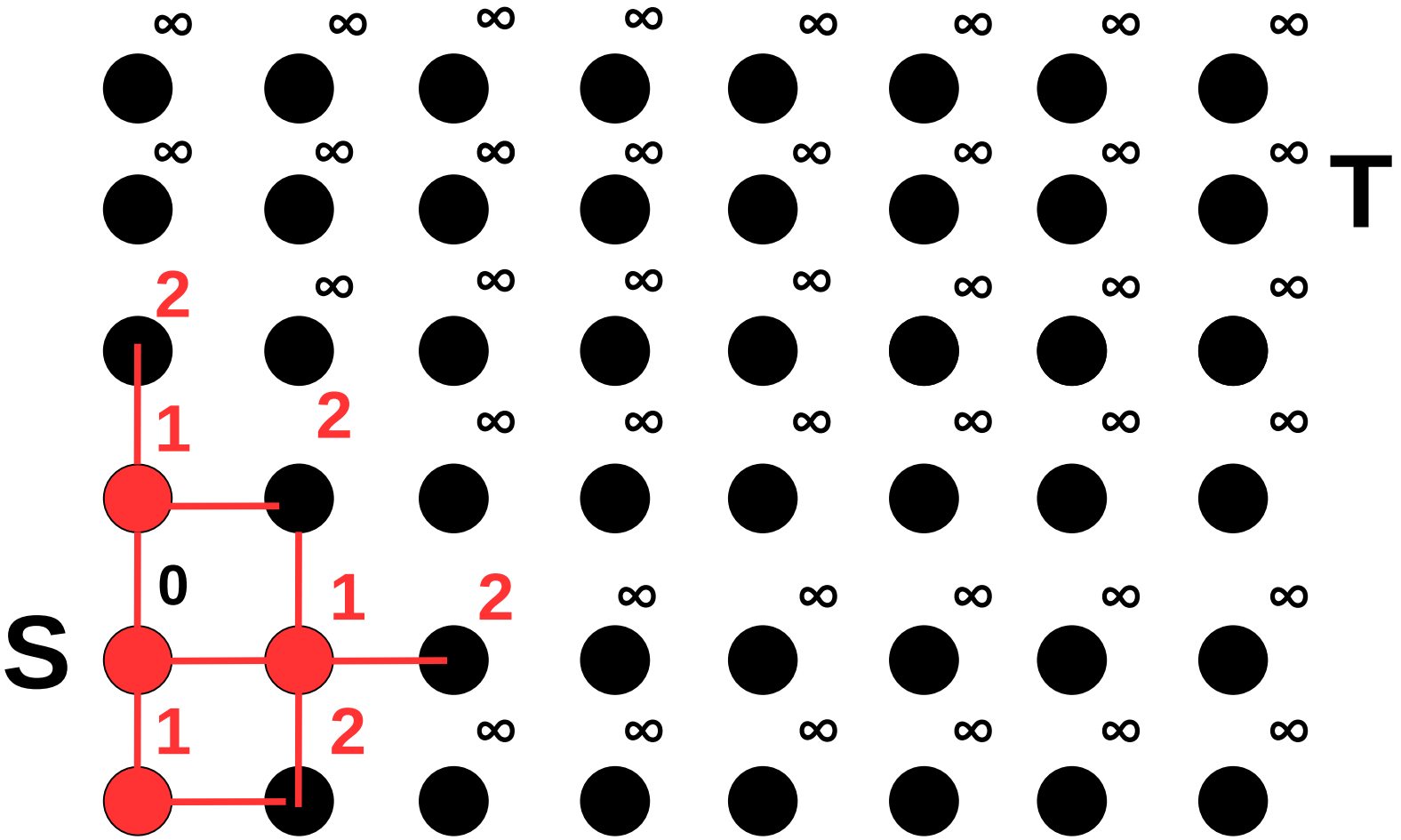
Very Important: if there is a conflict between a new tentative distance, and an old tentative distance, **the lower one wins! This is key!**

Step 7: Among unvisited (black) nodes, pick another one with minimum “tentative distance” as new current node...



Very Important: if there is a conflict between a new tentative distance, and an old tentative distance, **the lower one wins! This is key!**

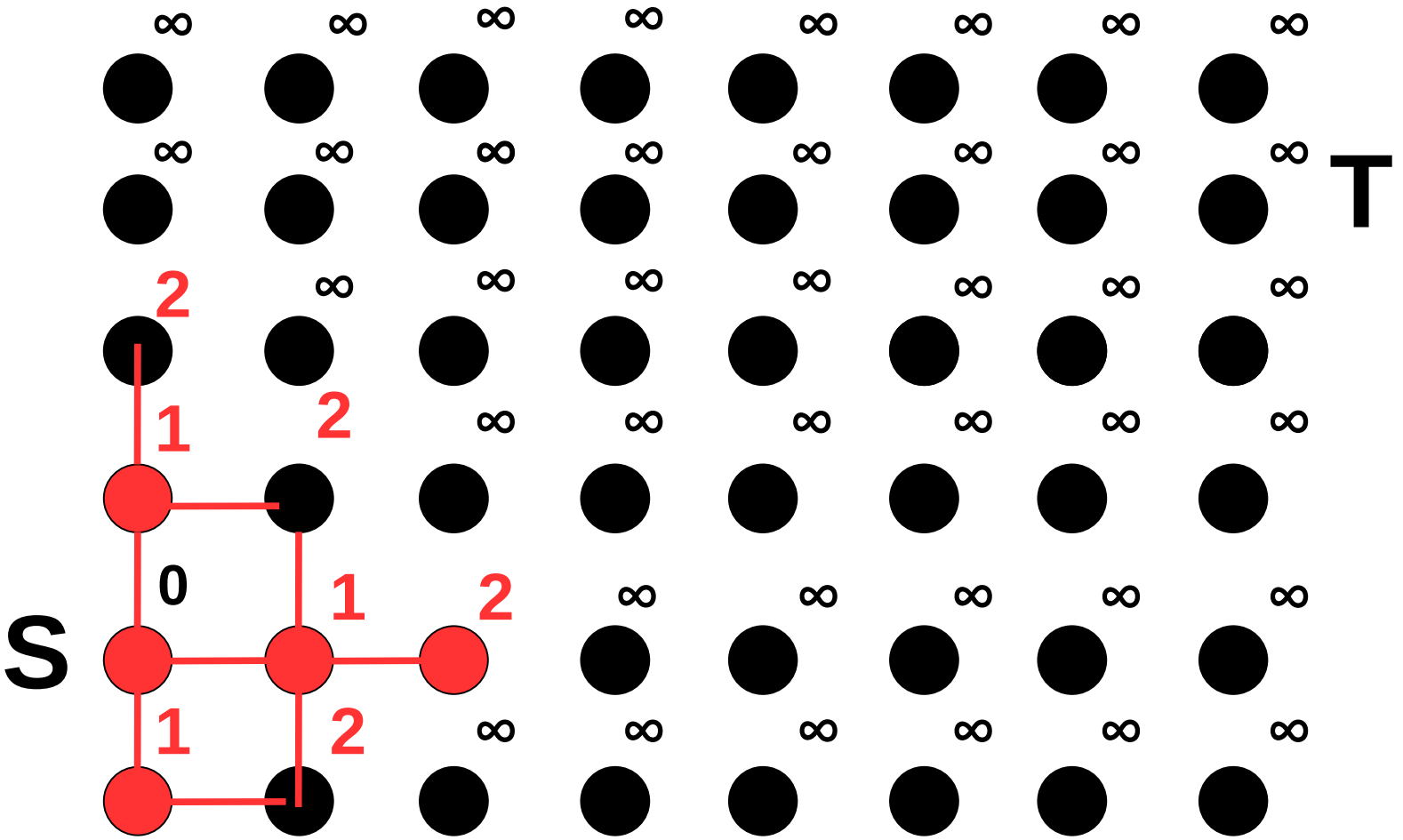
Step 8: Assign new tentative distances to unvisited (black) neighbors, resolving any conflicts...



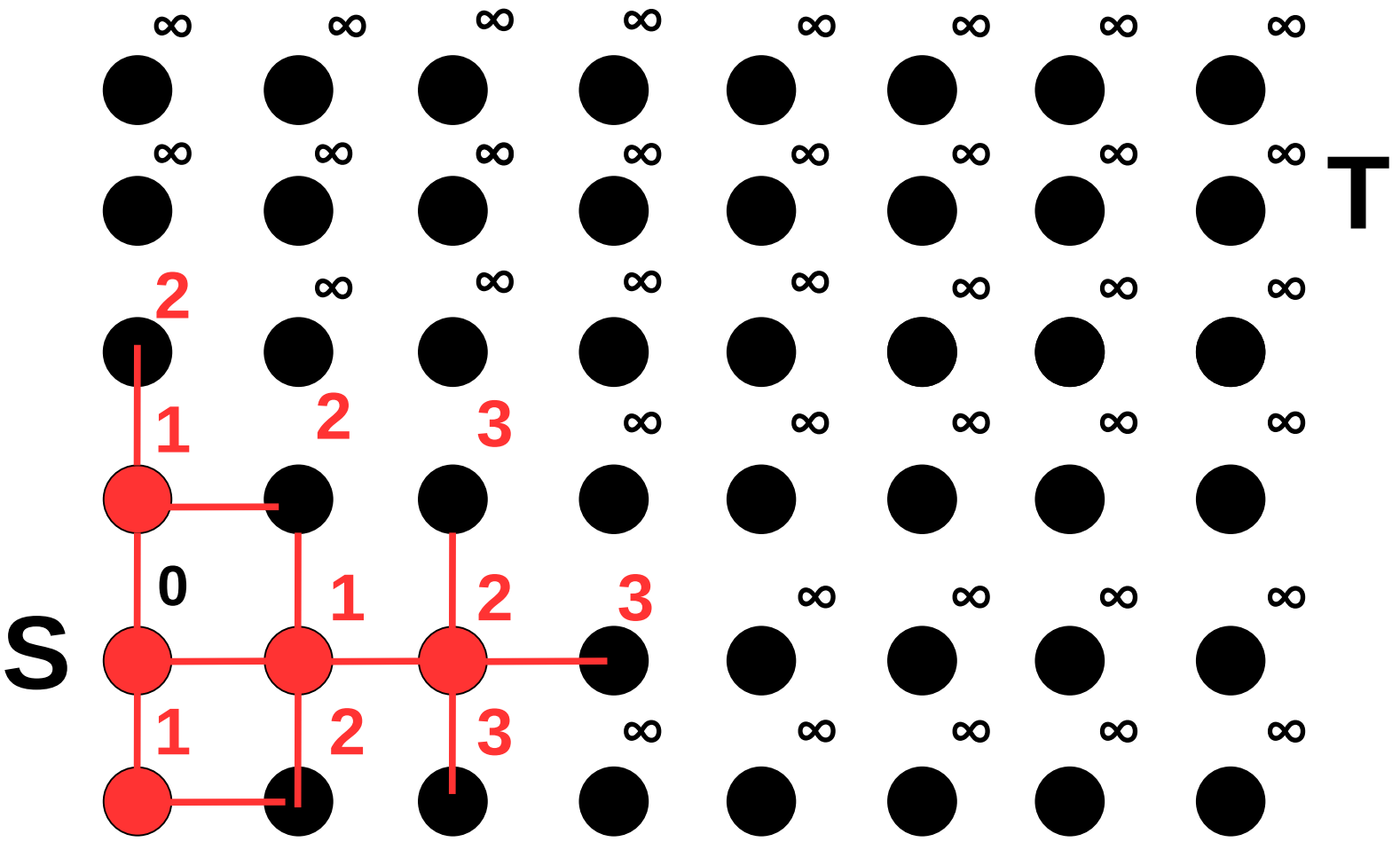
We technically have a conflict here! But anything is less than infinity, so it's obvious what to do.

This is why we initialize everything to infinity at the beginning.

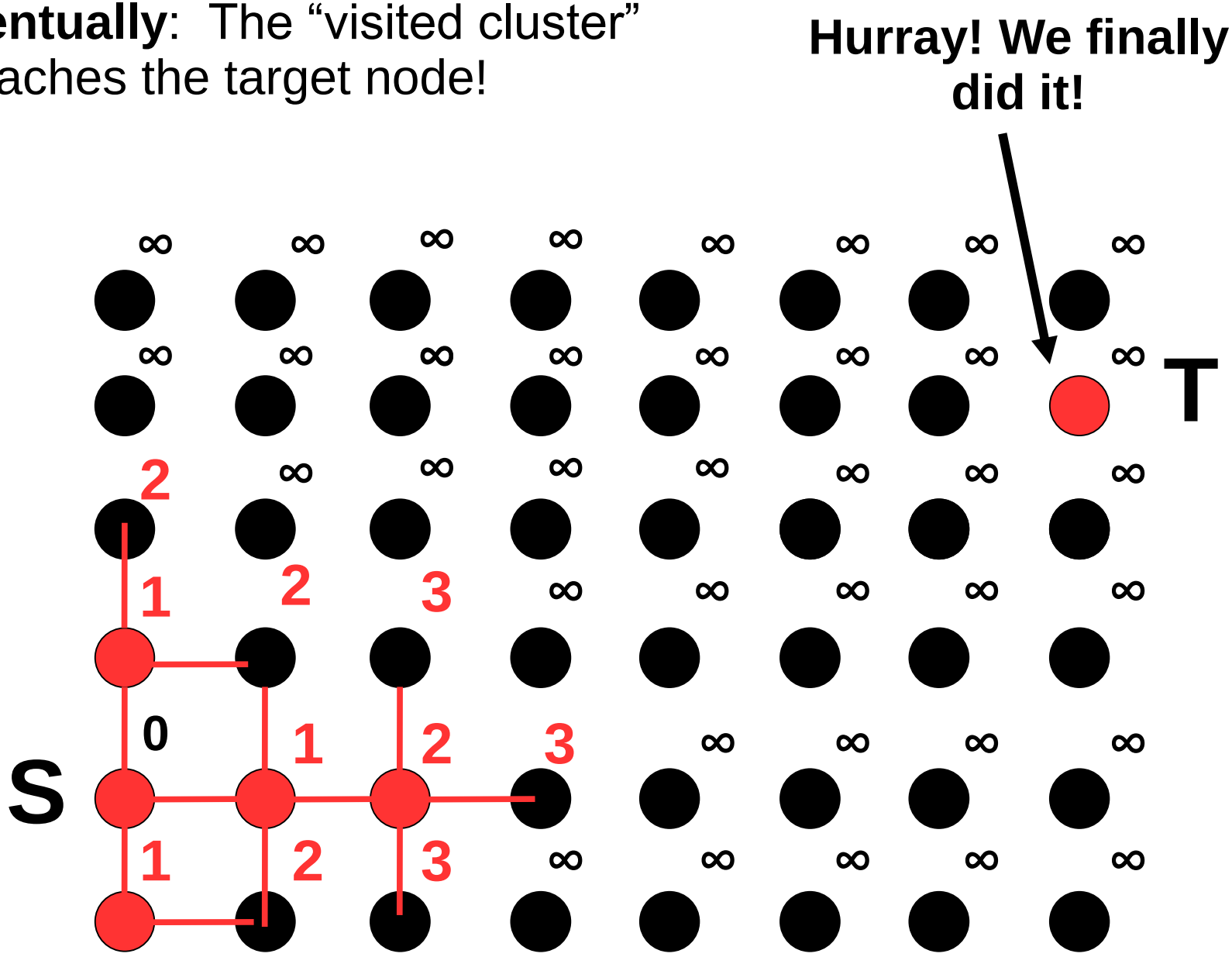
Step 9: Among unvisited (black) nodes, pick another one with minimum “tentative distance” as new current node...



Step 10: Assign new tentative distances to unvisited neighbors, and resolve conflicts...



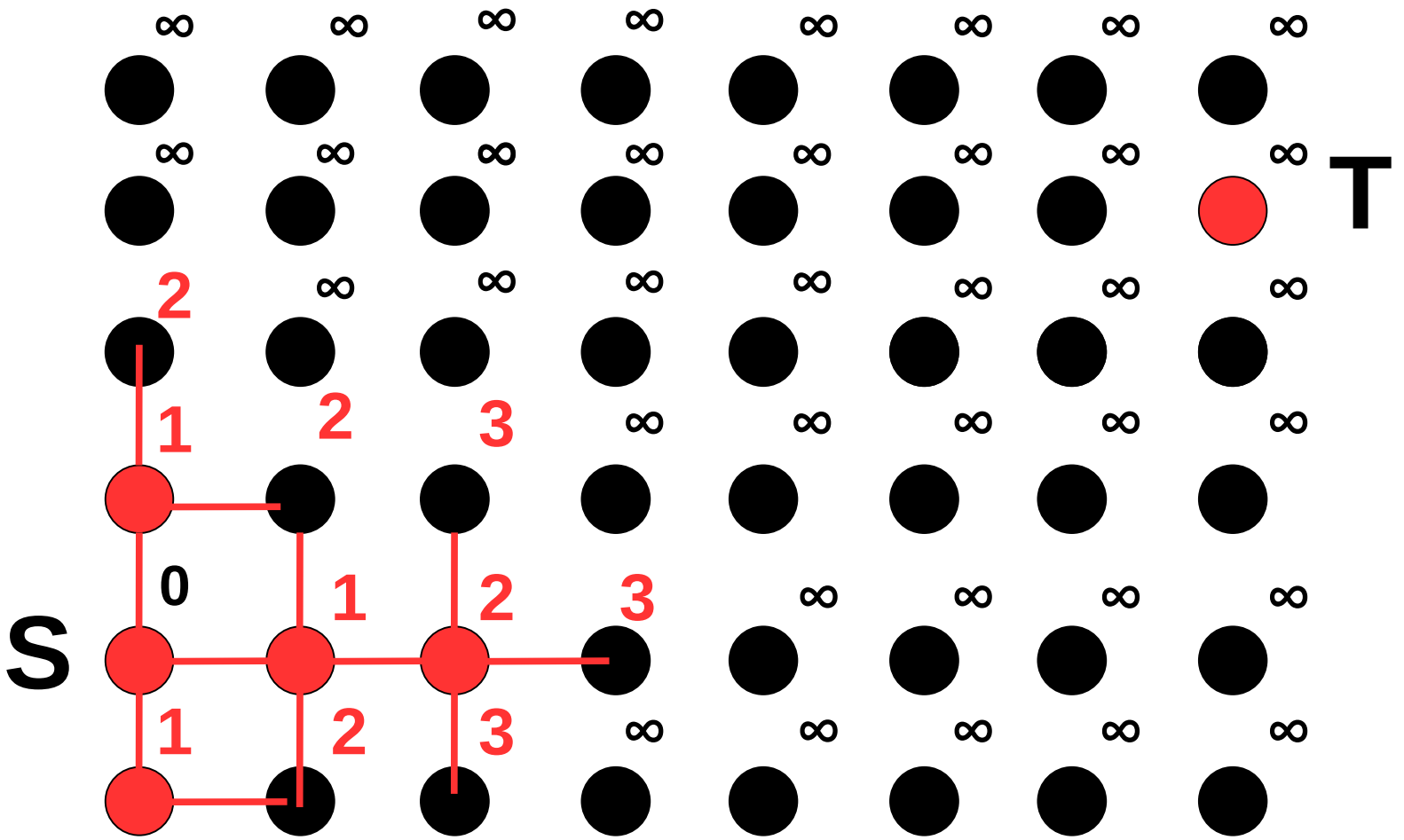
Step Eventually: The "visited cluster" reaches the target node!



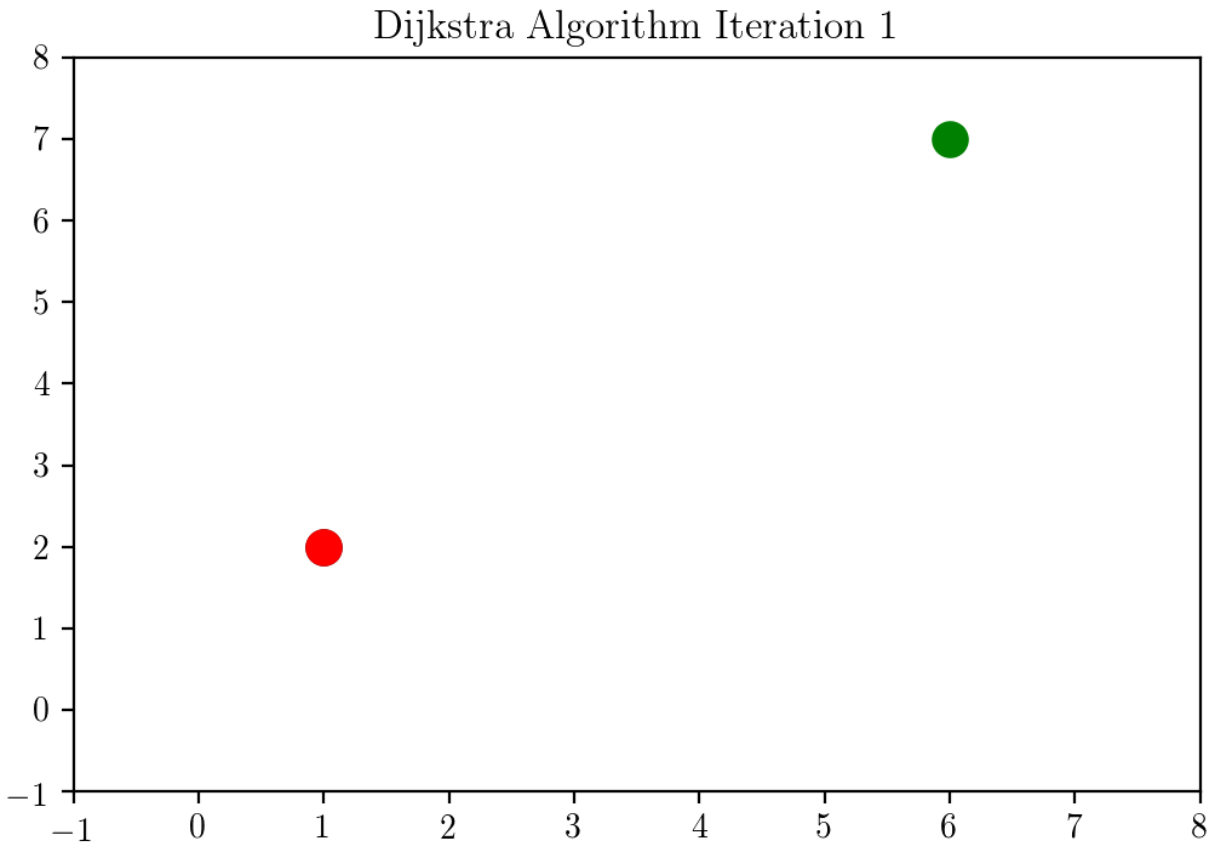
Because the "cluster" expands from around the source node, **the distance assigned to the target node by the time we visit is guaranteed to be minimal!**

Step Eventually: The “visited cluster” reaches the target node!

This all seems a bit trivial on a 2D evenly spaced grid, but **it generalizes easily to other situations**



For Example: what if nodes were unevenly spaced? Each step would work the same! **Keeping assigning tentative distances, and keep moving to the target.**



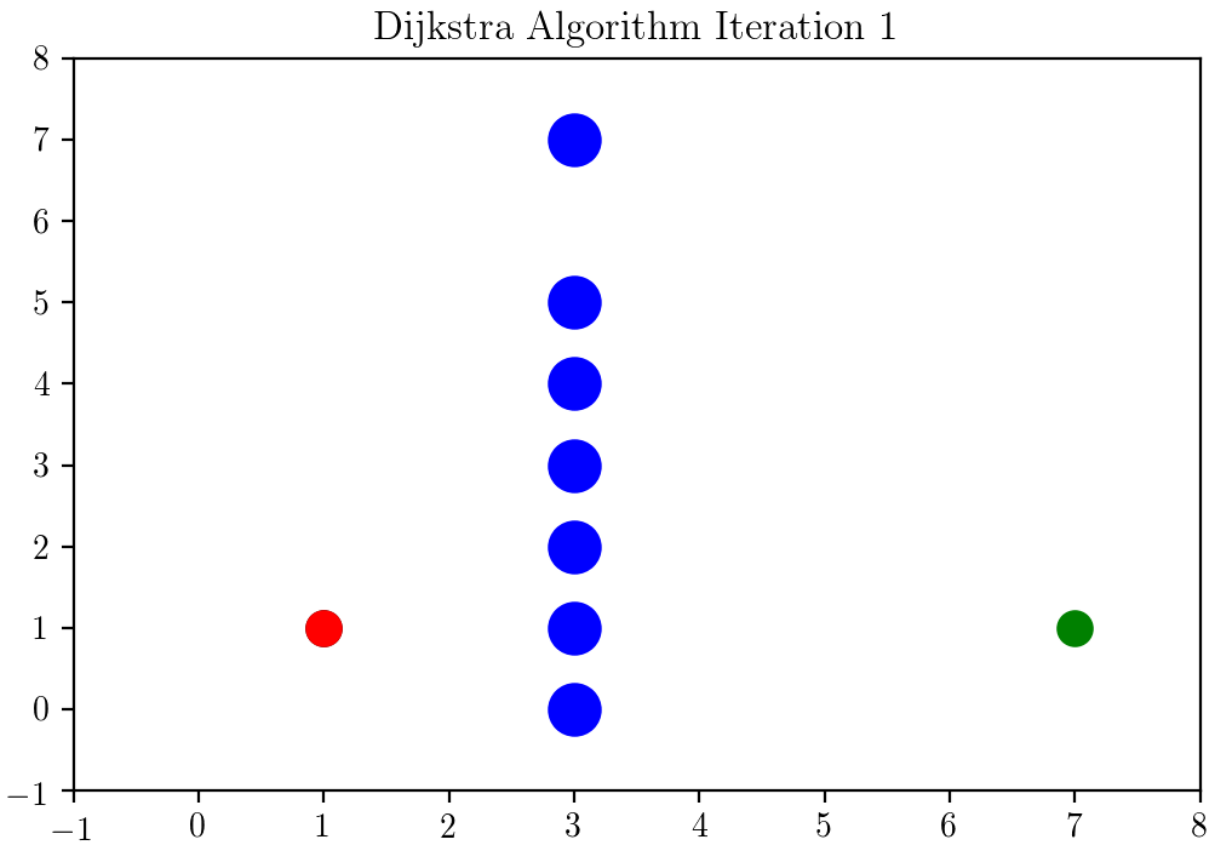
Red – Source Node

Green – Target Node

Black – Visited Node

Here's my code running on a 8**-by-8 grid**. See how the visited points spread-out from the source node!

For a graph with V nodes, runs as $\sim O(|V|^2)$



Red – Source Node

Green – Target Node

Blue – Inaccessible Node

Black – Visited Node

But what if there is a “wall” of inaccessible nodes? The algorithm works the same, and still gives the right answer!

For a graph with V nodes, runs as $\sim O(|V|^2)$

Perhaps I can show you all my hideously written, and unoptimized python code?....

Thank you!