Dijkstra’s Algorithm is one of the most well-known graph algorithms. It is also one of the hardest to spell and pronounce. It’s pronounced “dike-struss” algorithm. Dijkstra’s algorithm is a shortest path algorithm with many variations. In this post we’ll be going over two Python implementations of Dijkstra’s algorithm.

The first is the naive implementation, the second is the “lazy” implementation with a priority queue. We’ll cover both implementations with an [adjacency list](https://en.wikipedia.org/wiki/Adjacency_list) representation. Either way, Dijkstra’s algorithm follows the same pseudocode. Fun fact, Dijsktra came up with this algorithm on a coffee date with his fiancee!

Here’s the pseudocode for Dijsktra’s Algorithm:

1. Create a list of “distances” equal to the number of nodes and initialize each value to infinity
2. Set the “distance” to the starting node equal to 0
3. Create a list of “visited” nodes set to false for each node (since we haven’t visited any yet)
4. Loop through all the nodes
   1. Loop through all the nodes again, and pick the one that is the shortest distance away and not yet visited
   2. Set that node to visited
   3. Set the distance in the distance list to the distance to that node
5. The original “distance” list should now contain the shortest distance to each node or infinity if a node is unreachable from the desired starting node

## Example Adjacency List Graph

To do this example, we’ll have to install the numpy library. You can install the numpy library with pip using the command below in the terminal.

pip install numpy

For this example, we’ll be using the Inf object from numpy, this is a representation of infinity. We use this because if you try to use a really large integer, it could overflow into a negative number. We’ll create an adjacency list representation with 5 connected nodes.

from numpy import Inf

# create our graph using an adjacency list representation

# each "node" in our list should be a node name and a distance

graph = {

0: [(1, 1)],

1: [(0, 1), (2, 2), (3, 3)],

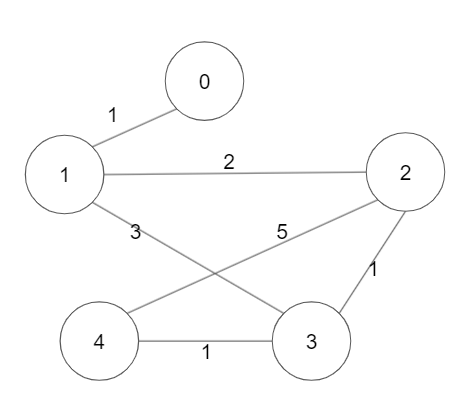
2: [(1, 2), (3, 1), (4, 5)],

3: [(1, 3), (2, 1), (4, 1)],

4: [(2, 5), (3, 1)]

}

A visual representation of our adjacency list should look like this



# Implement Naive Dijkstra’s in Python

Let’s go through each of these steps with a Naive implementation of Dijkstra’s. This implementation of Dijkstra’s has a runtime of O(N^2). We’ll create a function that takes two arguments, a graph argument, and a root argument.

First, we create a list of distances initialized to Infinity.

def naive\_dijkstras(graph, root):

n = len(graph)

# initialize distance list as all infinities

dist = [Inf for \_ in range(n)]

Step 2 is to initialize the distance of the root node to 0.

# set the distance for the root to be 0

dist[root] = 0

Next, we create a list of visited nodes, all initialized to False.

# initialize list of visited nodes

visited = [False for \_ in range(n)]

There are three parts to step 4. First let’s loop through the nodes and pick the next node to visit based on distance. If we’ve looped through all the available nodes and haven’t found a valid one, we break out of the inner loop.

# loop through all the nodes

for \_ in range(n):

# "start" our node as -1 (so we don't have a start/next node yet)

u = -1

# loop through all the nodes to check for visitation status

for i in range(n):

# if the node 'i' hasn't been visited and

# we haven't processed it or the distance we have for it is less

# than the distance we have to the "start" node

if not visited[i] and (u == -1 or dist[i] < dist[u]):

u = i

# all the nodes have been visited or we can't reach this node

if dist[u] == Inf:

break

The second part of step 4 is to add the closest node (which we should now have selected) to the list of visited nodes.

# set the node as visited

visited[u] = True

The last part of step 4 is to set the distance of the visited node to the shortest distance available.

# compare the distance to each node from the "start" node

# to the distance we currently have on file for it

for v, l in graph[u]:

if dist[u] + l < dist[v]:

dist[v] = dist[u] + l

Step 5 is to return the list of distances.

return dist

## Finalized code for Python Implementation of Naive Dijsktra’s

# takes the graph and the starting node

# returns a list of distances from the starting node to every other node

def naive\_dijkstras(graph, root):

n = len(graph)

# initialize distance list as all infinities

dist = [Inf for \_ in range(n)]

# set the distance for the root to be 0

dist[root] = 0

# initialize list of visited nodes

visited = [False for \_ in range(n)]

# loop through all the nodes

for \_ in range(n):

# "start" our node as -1 (so we don't have a start node yet)

u = -1

# loop through all the nodes to check for visitation status

for i in range(n):

# if the node 'i' hasn't been visited and

# we haven't processed it or the distance we have for it is less

# than the distance we have to the "start" node

if not visited[i] and (u == -1 or dist[i] < dist[u]):

u = i

# all the nodes have been visited or we can't reach this node

if dist[u] == Inf:

break

# set the node as visited

visited[u] = True

# compare the distance to each node from the "start" node

# to the distance we currently have on file for it

for v, l in graph[u]:

if dist[u] + l < dist[v]:

dist[v] = dist[u] + l

return dist

This will return the shortest path to each node as a list. Each index in the list corresponds to the node.

print(naive\_dijkstras(graph,1))



Python Simple Dijkstra’s Output

# Implement Lazy Dijkstra’s in Python

Now that we’ve covered the naive implementation of Dijkstra’s in Python, let’s cover the lazy implementation. Why is this called the lazy implementation? We don’t explicitly visit each node in our for loop on step 4. This is a more efficient implementation for sparse graphs (these are graphs in which each point is not connected to every other point). The runtime complexity for this implementation is O(n\*log(n)). This implementation will require us to import the heapq Python module to create a priority queue.

import heapq

Steps through three are pretty much the exact same as for the naive implementation of Dijkstra’s.

def lazy\_dijkstras(graph, root):

n = len(graph)

# set up "inf" distances

dist = [Inf for \_ in range(n)]

# set up root distance

dist[root] = 0

# set up visited node list

visited = [False for \_ in range(n)]

We’ll add a twist here before step 4, we’ll use a priority queue. Since this is Python, we can simply implement our priority queue as a list of tuples. We’ll start by inserting the root node with a distance of 0.

# set up priority queue

pq = [(0, root)]

We’ll do the first and second part of step 4 together. While the priority queue that we initialized to the root node with a distance of 0 has a length of greater than 0, we will process the priority queue. Note that we’ll use the \_ variable here when popping the first entry in our priority queue because we don’t need the distance, we just need the node. If the node has already been visited, we move on, otherwise we add the node to the list of visited nodes.

# while there are nodes to process

while len(pq) > 0:

# get the root, discard current distance

\_, u = heapq.heappop(pq)

# if the node is visited, skip

if visited[u]:

continue

# set the node to visited

visited[u] = True

The third part of step 4 is basically the same as we did in the naive Dijkstra’s implementation. We loop through the nodes adjacent to the node we’re processing and replace the value of that node in the distance list with the shortest distance we find.

# check the distance and node and distance

for v, l in graph[u]:

# if the current node's distance + distance to the node we're visiting

# is less than the distance of the node we're visiting on file

# replace that distance and push the node we're visiting into the priority queue

if dist[u] + l < dist[v]:

dist[v] = dist[u] + l

heapq.heappush(pq, (dist[v], v))

Step 5 is the same as well, we just return the list of distances.

## Finalized code for Dijkstra’s Algorithm with Priority Queue in Python

def lazy\_dijkstras(graph, root):

n = len(graph)

# set up "inf" distances

dist = [Inf for \_ in range(n)]

# set up root distance

dist[root] = 0

# set up visited node list

visited = [False for \_ in range(n)]

# set up priority queue

pq = [(0, root)]

# while there are nodes to process

while len(pq) > 0:

# get the root, discard current distance

\_, u = heapq.heappop(pq)

# if the node is visited, skip

if visited[u]:

continue

# set the node to visited

visited[u] = True

# check the distance and node and distance

for v, l in graph[u]:

# if the current node's distance + distance to the node we're visiting

# is less than the distance of the node we're visiting on file

# replace that distance and push the node we're visiting into the priority queue

if dist[u] + l < dist[v]:

dist[v] = dist[u] + l

heapq.heappush(pq, (dist[v], v))

return dist

When we run our function on node 1 we should see an output like below.

run lazy dijkstra's

lazy\_dijkstras(graph, 1)