CMSC 351: Dijkstra's Algorithm

Justin Wyss-Gallifent

November 8, 2023

1 Introduction

Dijkstra's Algorithm is essentially an extension of the shortest path algorithm in which the graph is weighted. In this case instead of looking for a shortest path we are looking for a path of minimal weight.

What we'll actually do is better, we'll find a shortest weight tree which is a tree that is a subgraph of the graph such that, treating the starting vertex as the root, explicitly shows us how to get to every other vertex with minimal weight.

2 Algorithm

Starting with a weighted, undirected simple graph and a starting vertex s, Dijkstra's Algorithm proceeds as follows. Here the graph has V vertices and $w(x, y)$ is the weight of the edge from x to y.

(a) Create a set $S = \{\}$.

(b) Create a distance array d of length V consisting of all ∞ except set $d[s] = 0$.

(c) Create a predecessor array p of length V consisting of all $NULL$.

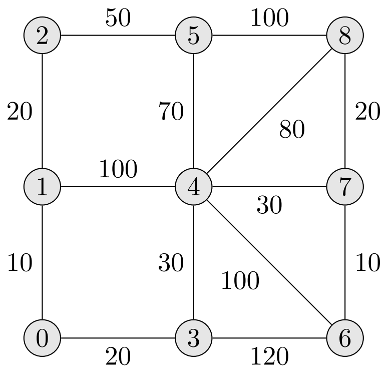
(d) Pick a vertex x with minimal distance which is not already in S. Add it to S. For all vertices y adjacent to x, if $d[x] + w(x, y) < d[y]$ then assign $d[y] = d[x] + w(x, y)$ and set $p[y] = x$.

(e) Repeat step (d) until S contains all vertices, meaning we can go no further.

Note 2.0.1. In step (d) there may be more than one option for x. In such a case we may pick any of them.

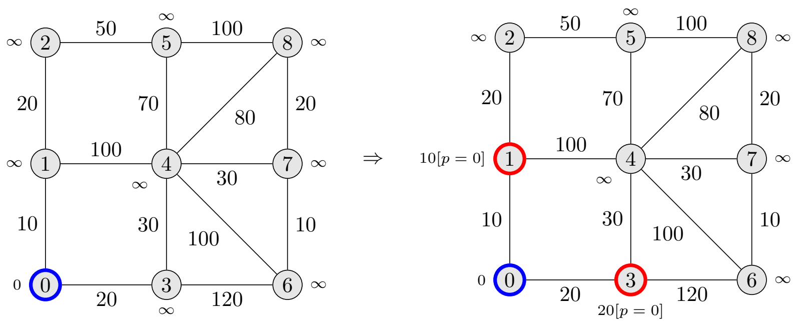
3 Working Through an Example

Example 3.1. Consider the following graph:



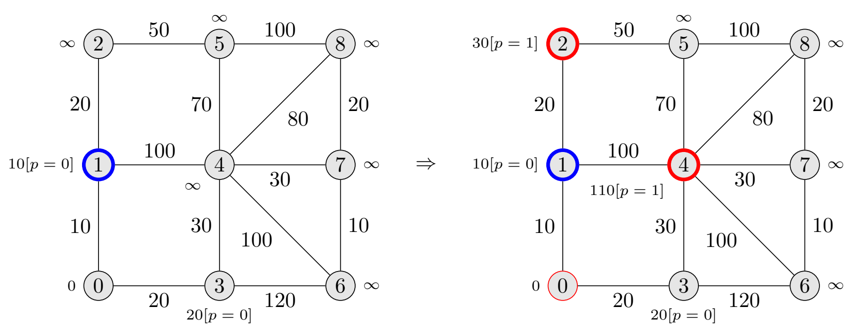
Suppose we choose 0 to be our starting vertex. We set S and P as instructed.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{\}$. This is of course currently vertex 0 with distance 0. so we look at all vertices connected to vertex 0 and assign their distances as vertex 0's distance plus the edge weight. We only do this if this value is smaller than their current weight but since their current weight is ∞ then of course we replace it. We also set those vertices' predecessor to be 0.



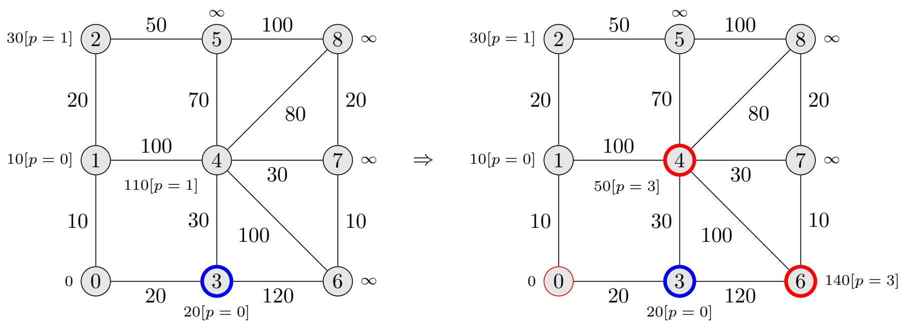
We put this vertex in S, so $S = \{0\}$

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0\}$. This is of course vertex 1 with distance 10. Then we look at all vertices connected to vertex 1 and assign their distances as vertex 1's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 1.



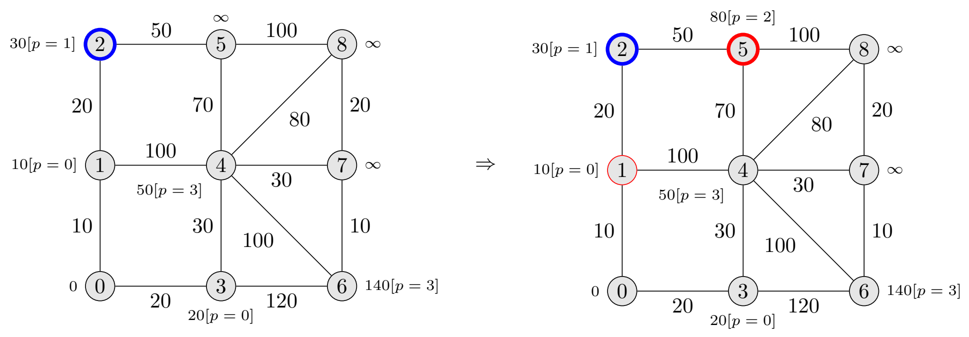
We put this vertex in S, so $S = \{0, 1\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1\}$. This is of course vertex 3 with distance 20. Then we look at all vertices connected to vertex 3 and assign their distances as vertex 3's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 3. Note that vertex 4 gets its weight reassigned because it was 110 but from vertex 3 it's $20 + 30 = 50 < 110$.



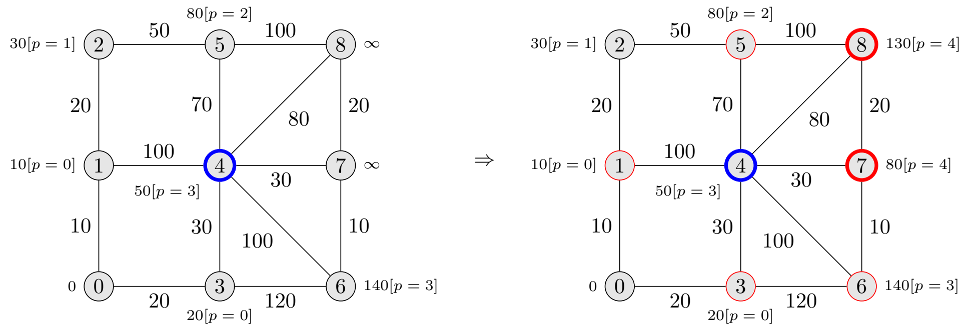
We put this vertex in S, so $S = \{0, 1, 3\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3\}$. This is of course vertex 2 with distance 30. Then we look at all vertices connected to vertex 2 and assign their distances as vertex 2's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 2.



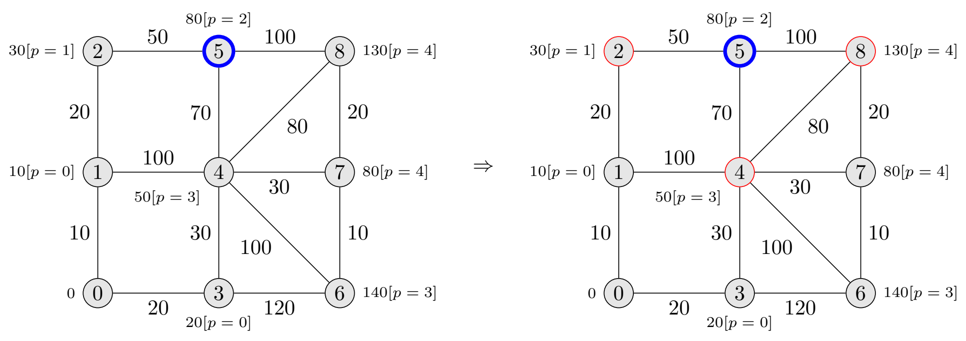
We put this vertex in S, so $S = \{0, 1, 3, 2\}$.

Iterate! we look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2\}$. This is of course vertex 4 with distance 50 Then we look at all vertices connected to vertex 4 and assign their distances as vertex 4's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 4.



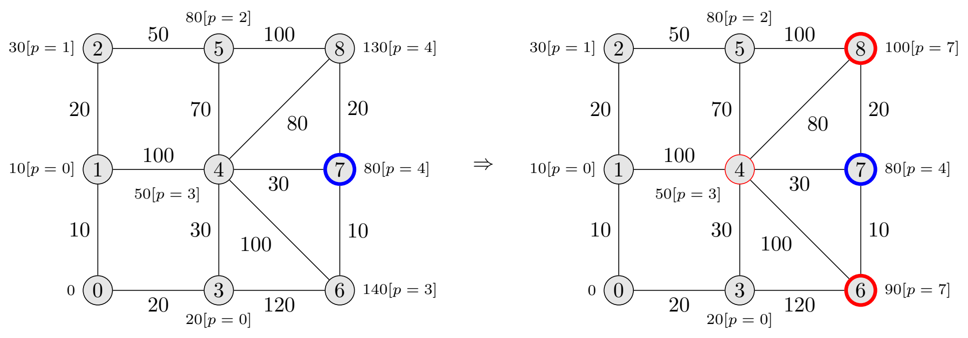
We put this vertex in S, so $S = \{0, 1, 3, 2, 4\}$

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4\}$. Both vertices 5 and 7 work so we can pick either. Let's choose vertex 5 with distance 80. Then we look at all vertices connected to vertex 5 and assign their distances as vertex 5's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 5. Here there are no changes.



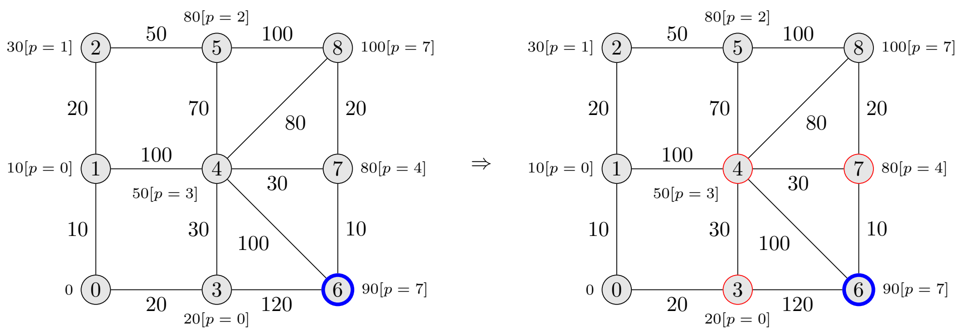
We put this vertex in S, so $S = \{0, 1, 3, 2, 4, 5\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5\}$. This is vertex 7 with distance 80. Then we look at all vertices connected to vertex 7 and assign their distances as vertex 7's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 7. Here there are no changes.



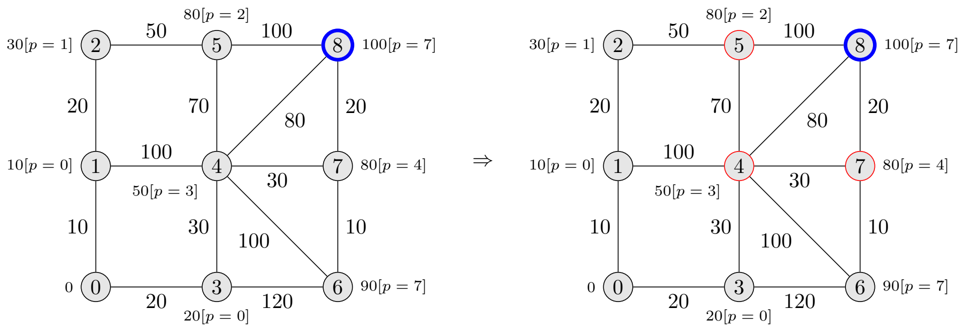
We put this vertex in S, so $S = \{0, 1, 3, 2, 4, 5, 7\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5, 7\}$. This is vertex 6 with distance 90. Then we look at all vertices connected to vertex 6 and assign their distances as vertex 6's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 6. Here there are no changes.



We put this vertex in S, so $S = \{0, 1, 3, 2, 4, 5, 7, 6\}$.

Iterate! We look at the graph and choose the vertex with minimum assigned distance which is not already in $S = \{0, 1, 3, 2, 4, 5, 7, 6\}$. This is vertex 8 with distance 100. Then we look at all vertices connected to vertex 8 and assign their distances as vertex 8's distance plus the edge weight. We only do this if this value is smaller than their current weight and if so we also set those vertices' predecessor to be 8. Here there are no changes.



We put this vertex in S, so $S = \{0, 1, 3, 2, 4, 5, 7, 6, 8\}$

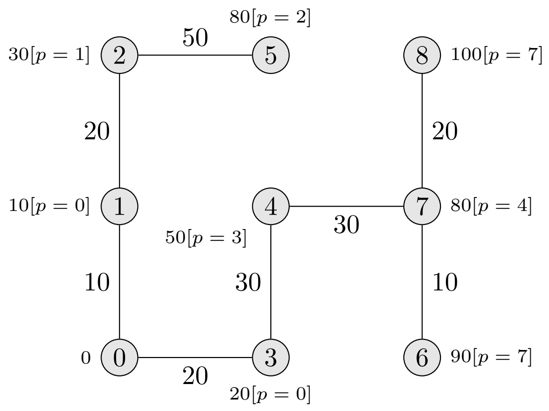
Now S contains every vertex and we are done.

Our array of predecessors is $P = [NULL, 0, 1, 0, 3, 2, 7, 4, 7]$ and this tells us how to construct our tree in the sense that the predecessor of each vertex is its parent and this tells us the edges. For example $P[0] = NULL$ because there is no predecessor of 0 as it is the root vertex, $P[1] = 0$ and so we need to have the edge $(1, 0)$, and so on.

In other words this array gives us the set of edges:

$$(1, 0), (2, 1), (3, 0), (4, 3), (5, 2), (6, 7), (7, 4), (8, 7)$$

In the graph we keep only those edges we get the following. The labels show the minimum weight path back to vertex 0 and the brackets show the predecessors still.



4 Rudimentary Pseudocode

Here is the pseudocode for a very rudimentary implementation. This code returns an array `pred` with the property that `pred[v]` gives the predecessor of the vertex `v` in the minimal weight tree.

```
\\ PRE: G is a graph with V vertices.
\\ PRE: s is the starting vertex.
def dijkstra(G,start):
    dist = [inf,...,inf] of length V.
    pred = [NULL,...,NULL] of length V.
    S = []
    dist[start] = 0
    while length(S) != V
        x = vertex in G-S with smallest distance
        for each vertex y connected to x
            if dist[x] + (Weight of Edge x,y) < dist[y]:
                dist[y] = dist[x] + (Weight of Edge x,y)
                pred[y] = x
            end
        end
       append x to S
    end
    return(pred)
end
```

5 Rudimentary Pseudocode Time Complexity

The argument for time complexity is similar to but not exactly the same as that for the Shortest Path Algorithm. Assuming we have stored the graph as an adjacency list:

- There is $\Theta(V)$ time required inside the function but excluding the while loop.
- The while loop iterates V times and the body of the while loop, excluding the for loop, takes $\Theta(V)$ time. This is due to the process of finding the vertex in $G - S$ with smallest distance. You should consider how this might be done. This then a total of $\Theta(V^2)$.
- Over the course of the entire algorithm the body of the for loop iterates twice for each edge, taking constant time for each, so that's $\Theta(2E) = \Theta(E)$.

Thus we can say for certain that in the worst-case:

$$T(V, E) = \Theta(V^2 + V + E) = \Theta(V^2 + E)$$

Since for a connected graph we have $V < 2E$ and $E \leq V^2 - V$ this is also $\mathcal{O}(E^2)$ and $\mathcal{O}(V^2)$.

6 Elegant Pseudocode

Here is the pseudocode for a more standard implementation. Instead of using simple lists we will manage the vertices using two structures kept in alignment:

- A min-heap `MH` such that each node contains a vertex number and the corresponding distance from `s`. The distance is the value used for min-heapedness. This is initialized with root node `MH[1]` storing vertex `s` with distance `0` and all other nodes the other vertices all with distance `INF`.
- A management structure `MS` indexed by vertex which contains the corresponding distance from `s`, the vertex's predecessor, the heap location of the vertex, and a flag indicating whether the vertex is (still) in the heap. This is initialized with `MS[s]` storing distance `0` with heap location `0` and predecessor `NULL` and all other `MS[i]` storing distance `INF` with corresponding heap location and predecessor `NULL`.

```
\\ PRE: G is a graph with n vertices.
\\ PRE: s is the starting vertex.
def dijkstra(G,s):
    initialize MH                                          }
    initialize MS                                          }(A)
    while MH is not empty
        Extract root node (u,udist) from MH                }
        Update MS to indicate u no longer in heap          }(B)
        for every edge attached to u
            v = vertex at the other end                    }
            (vdist,vinheap,vpred) = MS[v]                  }
            if vinheap and udist + weight(u,v) < vdist     }(C)
                update MH,MS with new distance, pred       }
            endif                                          }
        endfor
    endwhile
    return(MS)
end
```

7 Elegant Pseudocode Time Complexity

It's tempting to say that since the `while` loop iterates V times and the `for` loop iterates at most E times that there are V iterations of (B) and EV iterations of (C).

However we can be a bit more careful here. All together the `for` loop will follow each edge exactly twice. This is because an edge from u_i to u_j is followed once for u_i, directly after u_i is removed, and once for u_j, directly after u_j is removed. It is then never visited again. Thus in total (C) will iterate $2E$ times. This update of `MH` and `MS` is $\mathcal{O}(\lg V)$ for a total of $\mathcal{O}(2E \lg V)$.

In addition (B) will iterate V times. Extraction and update of `MH` and `MS` is $\mathcal{O}(\lg V)$ for a total of $\mathcal{O}(V \lg V)$.

Along with (A), which we presume is $\mathcal{O}(1)$, we have a total time complexity of:

$$\mathcal{O}(2E \lg V + V \lg V + 1) = \mathcal{O}((2E + V) \lg V) = \mathcal{O}(E \lg V)$$

Note 7.0.1. Arguably since we're allocating structures in (A) one might suggest that they're both $\mathcal{O}(V)$ since each has V items. This does not change the final time complexity calculation.

8 Mathematical Proof that it Works

It may (or may not) make intutive sense that Dijkstra's Algorithm does what it is claimed to do, but a proof is fairly straightforward.

The key point to understand is that while, in general, assigning a distance to a vertex is not final since that distance may be updated later, when a vertex is added to S the distance assigned to that vertex is in fact final and will not be updated later. It's this latter point we need to prove.

By $d(x)$ we denote the distance as assigned to vertex x by the algorithm. By $c(x_1, ..., x_j)$ we mean the total cost along the path $\langle x_1, ..., x_j \rangle$.

Note 8.0.1. This can also be rephrased in terms of a loop invariant and proven using the Loop Invariant Theorem. In that case the loop invariant is the statement about S at each iteration and the proof of the maintenance step is essentially what follows.

Theorem 8.0.1. We claim that whenever we put some x in S that the minimal cost path from s to x is the assigned $d(x)$.

Proof. Suppose not and that at some point we have an S and we are about to add (but have not yet added) some very first $x \notin S$ for which $d(x)$ is not equal to the minimal cost path from s to x.

We know that there must be some some minimal cost path from s to x and since it starts at $s \in S$ and ends at $x \notin S$ we write:

$$\langle s = v_1, v_2, ..., v_i, v_{i+1}, ..., v_k = x \rangle$$

Where v_i, v_{i+1} is the first pair along the path with $v_i \in S$ and $v_{i+1} \notin S$. Note that possibly $v_{i+1} = x$ here.

That path must be cheaper than $d(x)$ because $d(x)$ is not minimal:

$$c(s = v_1, ...v_i, v_{i+1}, ..., v_k = x) < d(x) \tag{I}$$

Since $v_i \in S$ we know two things about it. First, since v_i was added earlier, so $d(v_i)$ is minimal:

$$d(v_i) \leq c(v_1, ..., v_i) \tag{II}$$

Second, that when the algorithm processed v_i all adjacent nodes had their d-values set or updated, so:

$$d(v_{i+1}) \leq d(v_i) + c(v_i, v_{i+1}) \tag{III}$$

From here putting (III) and (II) together:

$$d(v_{i+1}) \leq d(v_i) + c(v_i, v_{i+1}) \leq c(v_1, ..., v_i) + c(v_i, v_{i+1}) = c(v_1, ..., v_{i+1}) \tag{IV}$$

In addition if $x = v_{i+1}$ then $d(x) = d(v_{i+1})$ and if $x \neq v_{i+1}$ then the algorithm chose x over v_{i+1} (when selecting what to put in S next) and so $d(x) \leq d(v_{i+1})$. Either way we have:

$$d(x) \leq d(v_{i+1}) \tag{V}$$

Now putting (V) and (IV) and (I) together:

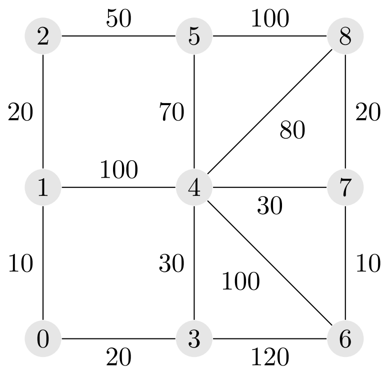
$$d(x) \leq d(v_{i+1}) \leq c(v_1, ..., v_{i+1}) \leq c(s = v_1, ...v_i, v_{i+1}, ..., v_k = x) < d(x)$$

This is a contradiction and we are done.

$\mathcal{QED}$

9 Thoughts, Problems, Ideas

1. For the graph given in the notes and replicated here, derive the minimal weight tree starting at vertex 7:



2. If the algorithm simply needs to find the minimum weight path from s to a specific vertex t at which point can it stop and why? Modify the pseudocode accordingly.

3. Dijkstra's Algorithm produces a mimimal weight tree rooted at a specific vertex s. Even though this tree is still a tree when we consider some other vertex $s' \neq s$ as the root this tree need not be a minimal weight tree rooted at s'. Show by a four-vertex example that this is the case. Justify your example.

4. A *spanning tree* for a graph G is a subgraph of G which contains all vertices of G and is also a tree. A *minimal spanning tree* is a spanning tree that has smallest possible weight. Show by a 3-vertex example that Dijkstra's Algorithm does not necessarily produce a minimal spanning tree. Justify your example.

10 Python Test and Output

The following code is applied to the graph above. This follows the model of the pseudocode and in addition creates and returns a list of the vertices in the order in which they were visited.

Code:

```
# Return the minimum value and index in dist but not in S.
def min(G,dist,S):
    n = len(G)
    mdist = float('inf')
    # Find a minimum value.
    for v in range(n):
        if v not in S:
            if dist[v] < mdist:
                mdist = dist[v]
    # Find the first index corresponding to that value.
    mi = None
    for v in range(n):
        if (v not in S) and (dist[v] == mdist):
            mvertex = v
            break

    return(mvertex)

def dijkstra(G,u):
    n = len(G)
    dist = [float('inf')] * n
    pred = [None] * n
    S = []
    dist[u] = 0
    print('S: ' + str(S))
    print('dist: ' + str(dist))
    while len(S) != n:
        print('')
        x = min(G,dist,S)
        print('Vertex not in S with minimum distance: ' + str(x))
        S.append(x)
        print('S = ' + str(S))
        for y in range(n):
            if G[x][y] != 0:
                print('Vertex: ' + str(y) + ': ',end='')
                if dist[x] + G[x][y] < dist[y]:
                    print('Update from '+str(dist[y]),end='')
                    print(' to ' + str(dist[x]+G[x][y]))
                    dist[y] = dist[x] + G[x][y]
                    pred[y] = x
                else:
                    print('Do not update from '+str(dist[y]),end='')
                    print(' to ' + str(dist[x]+G[x][y]))
        print('dist = ' + str(dist))
    return(pred)

G = [[  0, 10,  0, 20,  0,  0,  0,  0,  0],
     [ 10,  0, 20,  0,100,  0,  0,  0,  0],
     [  0, 20,  0,  0,  0, 60,  0,  0,  0],
     [ 20,  0,  0,  0, 30,  0,120,  0,  0],
     [  0,100,  0, 30,  0, 70,100, 30, 80],
     [  0,  0, 60,  0, 70,  0,  0,  0,100],
     [  0,  0,  0,120,100,  0,  0, 10,  0],
     [  0,  0,  0,  0, 30,  0, 10,  0, 20],
     [  0,  0,  0,  0, 80,100,  0, 20,  0]]
u = 0
pred = dijkstra(G,u)
print(pred)
```

Output:

```
S: []
dist: [0, inf, inf, inf, inf, inf, inf, inf, inf]

Vertex not in S with minimum distance: 0
S = [0]
Vertex: 1: Update from inf to 10
Vertex: 3: Update from inf to 20
dist = [0, 10, inf, 20, inf, inf, inf, inf, inf]

Vertex not in S with minimum distance: 1
S = [0, 1]
Vertex: 0: Do not update from 0 to 20
Vertex: 2: Update from inf to 30
Vertex: 4: Update from inf to 110
dist = [0, 10, 30, 20, 110, inf, inf, inf, inf]

Vertex not in S with minimum distance: 3
S = [0, 1, 3]
Vertex: 0: Do not update from 0 to 40
Vertex: 4: Update from 110 to 50
Vertex: 6: Update from inf to 140
dist = [0, 10, 30, 20, 50, inf, 140, inf, inf]

Vertex not in S with minimum distance: 2
S = [0, 1, 3, 2]
Vertex: 1: Do not update from 10 to 50
Vertex: 5: Update from inf to 90
dist = [0, 10, 30, 20, 50, 90, 140, inf, inf]

Vertex not in S with minimum distance: 4
S = [0, 1, 3, 2, 4]
Vertex: 1: Do not update from 10 to 150
Vertex: 3: Do not update from 20 to 80
Vertex: 5: Do not update from 90 to 120
Vertex: 6: Do not update from 140 to 150
Vertex: 7: Update from inf to 80
Vertex: 8: Update from inf to 130
dist = [0, 10, 30, 20, 50, 90, 140, 80, 130]

Vertex not in S with minimum distance: 7
S = [0, 1, 3, 2, 4, 7]
Vertex: 4: Do not update from 50 to 110
Vertex: 6: Update from 140 to 90
Vertex: 8: Update from 130 to 100
```

```
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 5
S = [0, 1, 3, 2, 4, 7, 5]
Vertex: 2: Do not update from 30 to 150
Vertex: 4: Do not update from 50 to 160
Vertex: 8: Do not update from 100 to 190
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 6
S = [0, 1, 3, 2, 4, 7, 5, 6]
Vertex: 3: Do not update from 20 to 210
Vertex: 4: Do not update from 50 to 190
Vertex: 7: Do not update from 80 to 100
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]

Vertex not in S with minimum distance: 8
S = [0, 1, 3, 2, 4, 7, 5, 6, 8]
Vertex: 4: Do not update from 50 to 180
Vertex: 5: Do not update from 90 to 200
Vertex: 7: Do not update from 80 to 120
dist = [0, 10, 30, 20, 50, 90, 90, 80, 100]
[None, 0, 1, 0, 3, 2, 7, 4, 7]
```