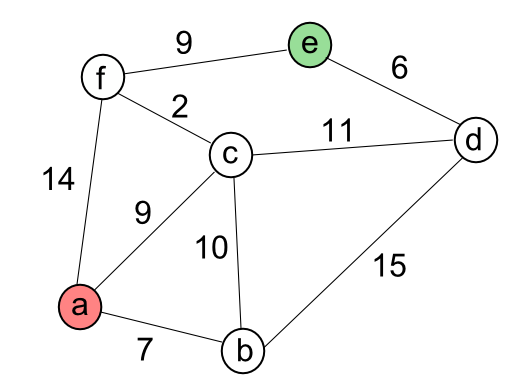
**Dijkstra Shortest Path Algorithm**



Dijkstra is an iterative algorithm that provides us with the shortest path from one particular starting node to all other nodes in the graph.

The algorithm iterates once for every vertex in the graph; however, the order that we iterate over the vertices is controlled by **a priority queue** - a First in First Out (FIFO) queue were its elements must be in order starting form the highest priority to the lowest priority. The priority in Dijkstra is for the least accumulative cost path. The value that is used to determine the order of the objects in the **priority queue** is **distance**.

* For the previous Network of nodes, we want to find the path with the smallest total weight among the possible paths we can take.

A -> Start Node

E -> End Node

* In the code, we create two classes: **Graph**, which holds the master list of vertices, and **Vertex**, which represents each vertex in the graph. The function **dijkstra()** calculates the shortest path. The **shortest()** function constructs the shortest path starting from the **target ('e')** using **predecessors**.
* To keep track of the total cost from the start node to each destination we will make use of the **distance** instance variable in the **Vertex** class. The **distance** instance variable will contain the current ***total weight of the smallest weight path from the start to the vertex***.

When a vertex is first created distance is set to Infinity **float("inf").**

* **The steps to calculates the path are:**

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes. Actually, initialization is done in the Vertex constructor: 

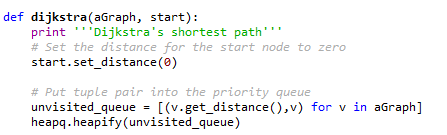
For the starting node, initialization is done in **dijkstra()**



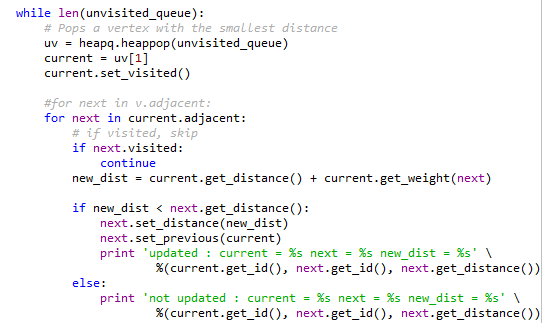
1. Mark all nodes unvisited. This is also done in the **Vertex** constructor:



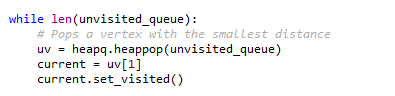
1. Set the initial node as current. Create a list of the unvisited nodes called the **unvisited queue** consisting of all the nodes. We do it using tuple pair**, (distance, v)**

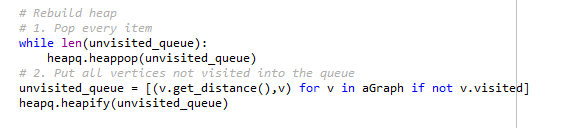


1. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. *Compare the newly calculated tentative distance to the current assigned value and assign the smaller one*. **For example**, if the current **node A** is marked with a distance of **6**, and the edge connecting it with a neighbor **B** has **length 2**, then the **distance** to B (through A) will be 6 + 2 = 8. ***If*** *B was previously marked with a distance greater than 8 then change it to 8*. ***Otherwise****, keep the current value.*



1. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set.

…



For each new node visit, we rebuild the heap: pop all items, refill the **unvisited\_queue**, and then **heapify** it.

1. A visited node will never be checked again. #for next **in v.adjacent**: for next in **current.adjacent**: # if visited, skip **if next.visited**: continue.
2. If there is no **unvisited** node, ***the algorithm has finished***. **Otherwise**, we go back to step 4.
3. Gather predecessors starting from the target **node ('e').** In the code, it's done in **shortest()** function.

