Graph-Shorest Path

December 5, 2017

0.1 Dijsktra algorithm

- Dijkstra’s algorithm time complexity: ***O(VlogV + E)***
- Dijkstra’ algorithm is a greedy one: it tries to find the global optimum with the help of local minimum -> it turns out to be good
- It is greedy -> on every iteration we want to find the minimum distance to the next vertex possible -> appropriate data structures heaps (binary or Fibonacci) or in general a priority queue

```
class Node:
    name
    min_distance
    Node predecesoor

def DijkstraAlgorithm(Graph, source):
    distance[source] = 0
    create vertex queue Q

    # Initialization phase: distance from source is 0,
    # because that is the starting point.
    # All the other nodes distances are infinity
    # because we do not know the distances in advance
    for v in Graph:
        distance[v] = inf
        predecesoor[v] = undefined # previous node in the shortest path
        add v to Q

    while Q not empty:
        u = vertex in Q with min distance # this is why to use heaps!!!

        for each neighbor v of u:
            tempDist = distance[u] + distBetween(u,v)
            if tempDist < distance[v]:
                distance[v] = tempDist
                predecessor[v] = u

    return distance[]
```

```
In [1]: import sys;
        import heapq;

        class Edge(object):

          def __init__(self, weight, startVertex, targetVertex):
            self.weight = weight;
            self.startVertex = startVertex;
            self.targetVertex = targetVertex;

        class Node(object):
          def __init__(self, name):
            self.name = name;
            self.visited = False;
            self.predecessor = None;
            self.adjacenciesList = [];
            self.minDistance = sys.maxsize;

          def __cmp__(self, otherVertex):
            return self.cmp(self.minDistance, otherVertex.minDistance);

          # less than method since we are using the min heap
          def __lt__(self, other):
            selfPriority = self.minDistance;
            otherPriority = other.minDistance;
            return selfPriority < otherPriority;
```

```
In [2]: class Algorithm(object):

          def calculateShortestPath(self, vertexList, startVertex):
            q = [];
            startVertex.minDistance = 0;
            heapq.heappush(q, startVertex); # turn q into the prority queue (min heap)

            while len(q) > 0:
              actualVertex = heapq.heappop(q);

              for edge in actualVertex.adjacenciesList:
                u = edge.startVertex;
                v = edge.targetVertex;
                newDistance = u.minDistance + edge.weight;

                if newDistance < v.minDistance:
                  v.predecessor = u;
                  v.minDistance = newDistance;
                  heapq.heappush(q, v);

          def getShortestPathTo(self, targetVertex):
```

```
    print("Shortest path to vertex is: ", targetVertex.minDistance);

    node = targetVertex;

    while node is not None:
      print("%s " % node.name);
      node = node.predecessor;
```

```
In [6]: node1 = Node("A")
        node2 = Node("B")
        node3 = Node("C")
        node4 = Node("D")
        node5 = Node("E")
        node6 = Node("F")
        node7 = Node("G")
        node8 = Node("H")

        edge1 = Edge(5, node1, node2)
        edge2 = Edge(8, node1, node8)
        edge3 = Edge(9, node1, node5)
        edge4 = Edge(15, node2, node4)
        edge5 = Edge(12, node2, node3)
        edge6 = Edge(4, node2, node8)
        edge7 = Edge(7, node8, node3)
        edge8 = Edge(6, node8, node6)
        edge9 = Edge(5, node5, node8)
        edge10 = Edge(4, node5, node6)
        edge11 = Edge(20, node5, node7)
        edge12 = Edge(1, node6, node3)
        edge13 = Edge(13, node6, node7)
        edge14 = Edge(3, node3, node4)
        edge15 = Edge(11, node3, node7)
        edge16 = Edge(9, node4, node7)

        node1.adjacenciesList.append(edge1);
        node1.adjacenciesList.append(edge2);
        node1.adjacenciesList.append(edge3);
        node2.adjacenciesList.append(edge4);
        node2.adjacenciesList.append(edge5);
        node2.adjacenciesList.append(edge6);
        node8.adjacenciesList.append(edge7);
        node8.adjacenciesList.append(edge8);
        node5.adjacenciesList.append(edge9);
        node5.adjacenciesList.append(edge10);
        node5.adjacenciesList.append(edge11);
        node6.adjacenciesList.append(edge12);
        node6.adjacenciesList.append(edge13);
        node3.adjacenciesList.append(edge14);
        node3.adjacenciesList.append(edge15);
        node4.adjacenciesList.append(edge16);

        vertexList = (node1, node2, node3, node4, node5 , node6, node7, node8);

        algorithm = Algorithm();
        algorithm.calculateShortestPath(vertexList, node1);
        algorithm.getShortestPathTo(node7)
```

```
Shortest path to vertex is:  25
G
C
F
E
A
```

In [ ]:

In [ ]:

0.2 Bellman Ford

- Time complexity: O(V * E)

```
In [18]: import sys;

         class Node(object):
           def __init__(self, name):
             self.name = name;
             self.visited = False;
             self.predecessor = None;
             self.adjacenciesList = [];
             self.minDistance = sys.maxsize;

         class Edge(object):

           def __init__(self, weight, startVertex, targetVertex):
             self.weight = weight;
             self.startVertex = startVertex;
             self.targetVertex = targetVertex;


         class BellmanFord(object):
           HAS_CYCLE = False;

           def calculateShortestPath(self, vertexList, edgeList, startVertex):
             startVertex.minDistance = 0;
```

```
    for i in range(0, len(vertexList)-1):
      for edge in edgeList:

        u = edge.startVertex;
        v = edge.targetVertex;

        newDistance = u.minDistance + edge.weight;

        if newDistance < v.minDistance:
          v.minDistance = newDistance;
          v.predecessor = u;

    for edge in edgeList:
      if self.hasCycle(edge):
        print("Negative cycle detected...");
        BellmanFord.Has_CYCLE = True;
        return;

  def hasCycle(self, edge):
    if (edge.startVertex.minDistance + edge.weight) < edge.targetVertex.minDistance:
      return True;
    else:
      return False;

  def getShortestPathTo(self, targetVertex):

    if not BellmanFord.HAS_CYCLE:
      print("Shortest path exists with value: ", targetVertex.minDistance);

      node = targetVertex;

      while node is not None:
        print("%s " % node.name)
        node = node.predecessor;
    else:
      print("Negative cycle detected...")
```

```
In [19]: node1 = Node("A")
         node2 = Node("B")
         node3 = Node("C")
         node4 = Node("D")
         node5 = Node("E")
         node6 = Node("F")
         node7 = Node("G")
         node8 = Node("H")
```

```
        edge1 = Edge(5, node1, node2)
        edge2 = Edge(8, node1, node8)
        edge3 = Edge(9, node1, node5)
        edge4 = Edge(15, node2, node4)
        edge5 = Edge(12, node2, node3)
        edge6 = Edge(4, node2, node8)
        edge7 = Edge(7, node8, node3)
        edge8 = Edge(6, node8, node6)
        edge9 = Edge(5, node5, node8)
        edge10 = Edge(4, node5, node6)
        edge11 = Edge(20, node5, node7)
        edge12 = Edge(1, node6, node3)
        edge13 = Edge(13, node6, node7)
        edge14 = Edge(3, node3, node4)
        edge15 = Edge(11, node3, node7)
        edge16 = Edge(9, node4, node7)

        node1.adjacenciesList.append(edge1);
        node1.adjacenciesList.append(edge2);
        node1.adjacenciesList.append(edge3);
        node2.adjacenciesList.append(edge4);
        node2.adjacenciesList.append(edge5);
        node2.adjacenciesList.append(edge6);
        node8.adjacenciesList.append(edge7);
        node8.adjacenciesList.append(edge8);
        node5.adjacenciesList.append(edge9);
        node5.adjacenciesList.append(edge10);
        node5.adjacenciesList.append(edge11);
        node6.adjacenciesList.append(edge12);
        node6.adjacenciesList.append(edge13);
        node3.adjacenciesList.append(edge14);
        node3.adjacenciesList.append(edge15);
        node4.adjacenciesList.append(edge16);

        vertexList = (node1, node2, node3, node4, node5 , node6, node7, node8);
        edgeList = (edge1, edge2, edge3, edge4, edge5, edge6, edge7, edge8, edge9, edge10, edge

        algorithm = BellmanFord();
        algorithm.calculateShortestPath(vertexList, edgeList, node1);
        algorithm.getShortestPathTo(node7)
```

```
Shortest path exists with value:  25
G
C
F
E
A
```

```
In [ ]:
```