Shortest Paths

-Introduction

i) Definition

ii) Special property

iii) Edge Relaxation

-Dijkstra’s algorithm(aka lowest cost first)

i) Sample Applet

ii) Applications

Problem statement:

given a graph G with n vertices and m edges the minimal path with respect to the number of vertices can be found by doing a BFS on the graph. However, given the same graph with a weighting function f(e) such that e is in the set of edges, f(e) is in the set of positive real numbers, we want to find the minimal path from vertex v0 to all vi in the graph.

The Greedy condition:

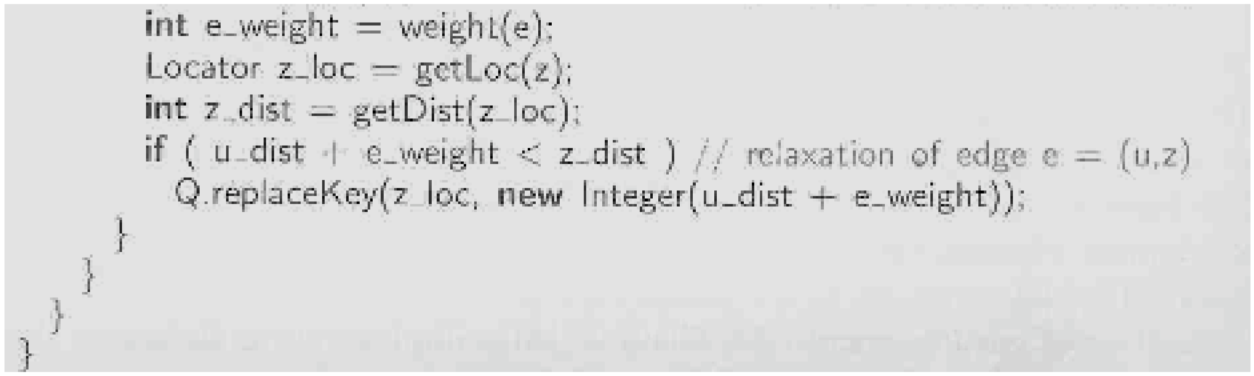
given an optimaization problem if we can find the optimal global solution based on optimal local solutions to subsections of this problem then we say that the given problem satisfies the greedy condition. It is Because the shortest paths problem satisfies this condition that Dijkstra's algorithm works.

Edge relaxation:

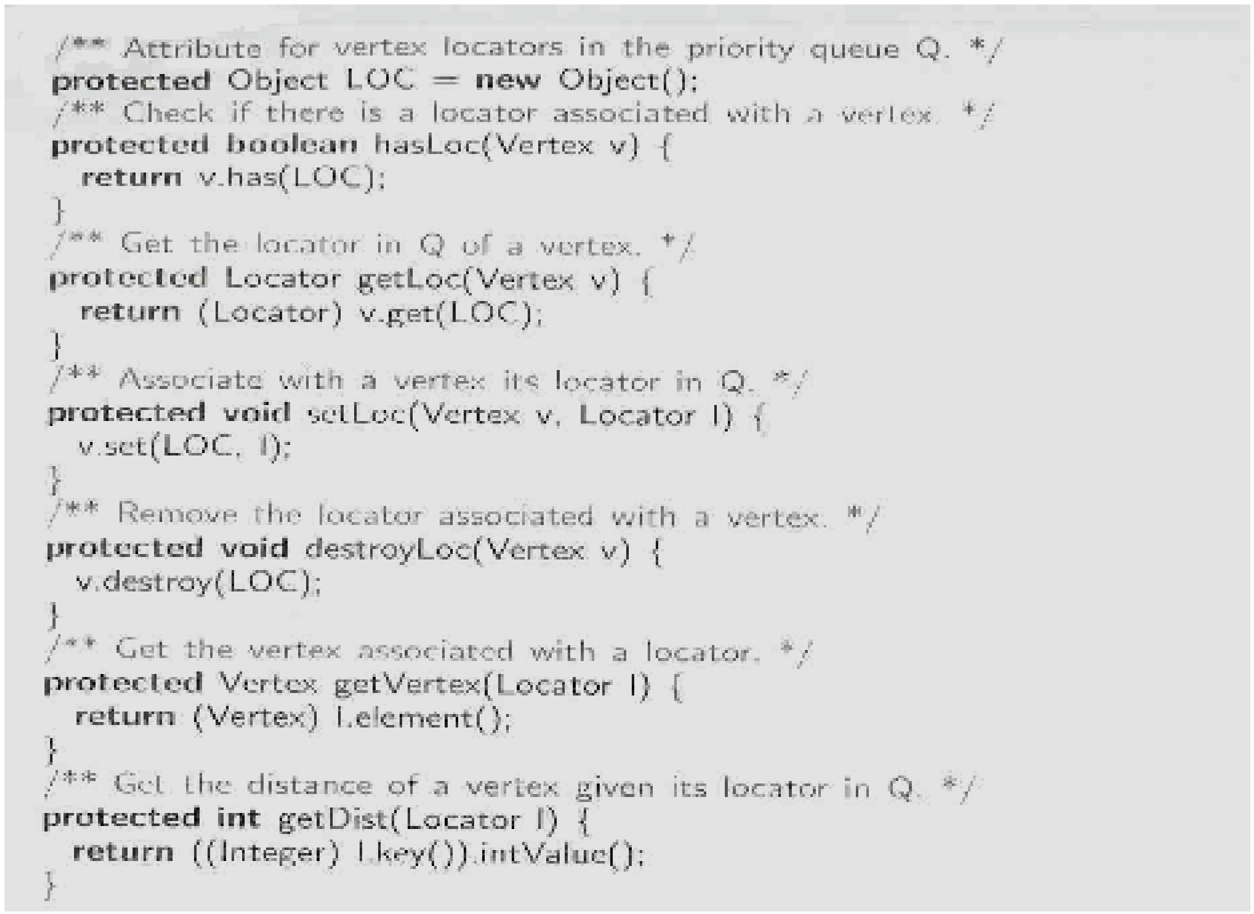
Edge relaxation is a technique used to incrementally assign values to the paths from v0 to all vi in the given graph G. all shortest path algorithms use some form of edge relaxation to make correct optimal local choises. Dijkstra's algorithm looks like :

```
for all z adjacent to vi
    if(D[vi]+f(e(vi,z)) < D[z])
        D[z] = D[vi]+f(e(vi,z));
```

```
public abstract class Dijkstra {
  /** Execute Dijkstra's algorithm. */
  public void execute(InspectableGraph g, Vertex source) {
    graph = g;
    dijkstraVisit(source);
  }
  /** Attribute for vertex distances. */
  protected Object DIST = new Object();
  /** Set the distance of a vertex. */
  protected void setDist(Vertex v, int d) {
    v.set(DIST, new Integer(d));
  }
  /** Get the distance of a vertex from the source vertex. This method
   * returns the length of a shortest path from the source to u after
   * method execute has been called. */
  public int getDist(Vertex u) {
    return ((Integer) u.get(DIST)).intValue();
  }
  /** This abstract method must be defined by subclasses.
   * @return weight of edge e. */
  protected abstract int weight(Edge e);
  /** Infinity value. */
  public static final int INFINITE = Integer.MAX_VALUE;
  /** Input graph. */
  protected InspectableGraph graph;
  /** Auxiliary priority queue. */
  protected PriorityQueue Q;
protected void dijkstraVisit (Vertex v) {
  // initialize the priority queue Q and store all the vertices in it
  Q = new ArrayHeap(new IntegerComparator());
  for (VertexIterator vertices = graph.vertices(); vertices.hasNext();) {
    Vertex u = vertices.nextVertex();
    int u_dist;
    if (u==v)
      u_dist = 0;
    else
      u_dist = INFINITE;
    // setDist(u, u_dist);
    Locator u_loc = Q.insert(new Integer(u_dist), u);
    setLoc(u, u_loc);
  }
  // grow the cloud, one vertex at a time
  while (!Q.isEmpty()) {
    // remove from Q and insert into cloud a vertex with minimum distance
    Locator u_loc = Q.min();
    Vertex u = getVertex(u_loc);
    int u_dist = getDist(u_loc);
    Q.remove(u_loc); // remove u from the priority queue
    setDist(u, u_dist); // the distance of u is final
    destroyLoc(u); // remove the locator associated with u
    if (u_dist == INFINITE)
      continue; // unreachable vertices are not processed
    // examine all the neighbors of u and update their distances
    for (EdgeIterator edges = graph.incidentEdges(u); edges.hasNext();) {
      Edge e = edges.nextEdge();
      Vertex z = graph.opposite(u,e);
      if (hasLoc(z)) { // check that z is in Q, i.e., it is not in the cloud
```



```
        int e_weight = weight(e);
        Locator z_loc = getLoc(z);
        int z_dist = getDist(z_loc);
        if ( u_dist + e_weight < z_dist ) // relaxation of edge e = (u,z)
          Q.replaceKey(z_loc, new Integer(u_dist + e_weight));
      }
    }
  }
}
```



```
/** Attribute for vertex locators in the priority queue Q. */
protected Object LOC = new Object();
/** Check if there is a locator associated with a vertex. */
protected boolean hasLoc(Vertex v) {
  return v.has(LOC);
}
/** Get the locator in Q of a vertex. */
protected Locator getLoc(Vertex v) {
  return (Locator) v.get(LOC);
}
/** Associate with a vertex its locator in Q. */
protected void setLoc(Vertex v, Locator l) {
  v.set(LOC, l);
}
/** Remove the locator associated with a vertex. */
protected void destroyLoc(Vertex v) {
  v.destroy(LOC);
}
/** Get the vertex associated with a locator. */
protected Vertex getVertex(Locator l) {
  return (Vertex) l.element();
}
/** Get the distance of a vertex given its locator in Q. */
protected int getDist(Locator l) {
  return ((Integer) l.key()).intValue();
}
```

```
/** A specialization of class Dijkstra that extracts edge weights from
 * decorations. */
public class MyDijkstra extends Dijkstra {
  /** Attribute for edge weights. */
  protected Object WEIGHT;
  /** Constructor that sets the weight attribute. */
  public MyDijkstra(Object weight_attribute) {
    WEIGHT = weight_attribute;
  }
  /** The edge weight is stored in attribute WEIGHT of the edge. */
  public int weight(Edge e) {
    return ((Integer) e.get(WEIGHT)).intValue();
  }
}
```

The Algorithm:

```
Dijkstra( graph G , vertex v0)
{
    // initialization

    Priority Queue q  = new PriorityQueue();
    D[v] = 0;
    q.push(v0,D[v0]);
    for all vi != v0
    {
        D[vi] = infinity;
        q.push(vi,D[v0])
    }

    //start of algorithm

    vertex temp;
    while(!q.isEmpty())
    {
        temp =  q.getMin();
        for all vertices z adjacent to temp
        {
            if(D[temp]+weight(temp,z) <D[z])
            D[z] =  D[temp]+weight(temp,z);
            q.changeKey(z,D[z]);
        }
    }
    return D[ ];
}
```

Sample Applet:

http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/DijkstraApp.shtml?demo1

Proof of correctness for Dijkstra's algorithum:

Intuitively we can see why Dikstra's algorithm works which is because it is vey similer to BFS but it uses a priority queue instead of a queue with the weights as the keys. At every iteration the vertex with the minimal weight is chosen and edge relation is preformed on its neighbores.

for a formal proof see goodrich book page 346.
http://ww3.algorithmdesign.net/sample/ch07-weights.pdf

Time analysis for the algorithm:

We are looking at simple (no loops or parallel edges) non-negative graphs so we can't have comple graphs as input . In other words the degree of every vertex is n-1. Recall that the sum of the degrees of all the verticies < 2 * number of edges at worst case we will have that the degree of every vertex is n-1 which means that in this case Sum(deg(all vi)) = 2 * # of edges. So, for a graph with n vertices we have that the sum of the degrees of the vertices is (n^2)-n. This implies that we have ((n^2)-n)/2 edges to consider. so for a graph with n edges we have running time is O(n^2 log(n)).
A best case senarion would be if we had a Spanning Tree. In this case we have n-1 edges to consider so the running time is simply O(n log(n)).

Applications of dijkstra’s algorithm:

- The most obvious applications arise in transportation or communications, such as finding the best route to drive between Chicago and Phoenix or figuring how to direct packets to a destination across a network.

- Consider the problem of image segmentation, that is, separating two characters in a scanned, bit-mapped image of printed text. We need to find the separating line between two points that cuts through the fewest number of black pixels. This grid of pixels can be modeled as a graph, with any edge across a black pixel given a high cost. The shortest path from top to bottom defines the best separation between left and right.

- A major problem in speech recognition is distinguishing between words that sound alike (homophones), such as *to*, *two*, and *too*. We can construct a graph whose vertices correspond to possible words, with an edge between possible neighboring words. If the weight of each edge measures the likelihood of transition, the shortest path across the graph defines the best interpretation of a sentence.

- Suppose we want to draw an informative picture of a graph. The center of the page should cooorespond to the ``center'' of the graph, whatever that means. A good definition of the center is the vertex that minimizes the maximum distance to any other vertex in the graph. Finding this center point requires knowing the distance (i.e. shortest path) between all pairs of vertices.

References:

-Algorithm Design Foundation, Analysis, and Internet Examples. Michle T. Goodrich, Roberto tamassia. chapter 7.
-Mathimatical Programming. http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/main/index.shtml
-shortest path. http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE162.HTM
-Notes from last years shortest path seminar...