15-210 Assignment ThesaurusLab

Roy Sung
roysung@andrew.cmu.edu
Section E
3/3/2014

6: Fire! Fire! Fire!

Task 6.1
Our implementation of the graph will be a table where the key is a vertex and it gets mapped to a sequence of vertices that is connect to vertex key by an edge. So we first state that we begin this problem by stating that when a road becomes blocked with traffic this means that we cannot not take this road(or we would get stuck in traffic). So that means that we have to find some other way around the road block. This means in our graph representation that one of the edges gets taken out of the graph. So our algorithm will be doing a breadth first search and find the shortest path to the single source v to all other vertices in our graph representation of Pittsburgh. So our code will look like this.

```
fun PittsburghRobust G alledges = (*enters in G a graph and
                                    alledges: all edges in G*)
  let
  fun doEachEdge G singleEdge = (* Graph G and a singleEdge is a single
                                   edge from G *)
    let
      val newG = erase(G,x) (*removes the edge from the graph
                             note this erase is not Table.erase*)
      val (spG,spnewG) = (SP(G),SP(newG))
      val result = checker(spG,spnewG)
               (*function checker checks if all paths are within one
                 note that this does not cost as much as SP*)
    in
    result
    end
  in
  reduce (fn (x,y) => x andalso y)
         (true)
         (map (fn i => doEachEdge G i)(alledges))
  end
```

Task 6.2
We can note that in the algorithm for above, we test for all edges. This may not be necessary as the shortest paths to each node for the single source may only use a certain number of edges and not all the edges. So so we just need to test for the edges that makes a path to a certain vertex, the shortest. We also argue that if we do this for all vertices n then we can say that Pittsburgh is robust or not. The edge that we will take out will be the last edge that connects the second to last vertex to the last vertex in a shortest path from the given source. This will make the shortest path

to the last vertex invalid as the edge does not exist and it will also not affect the shortest paths that appear before the last vertex. So our algorithm will look like this

```
fun pittsburghRobust G listVertices =
let
val originalSP = SP(G)(*In this case SP will return the sequence of
                         vertices of shortest path*)
fun doOnEachVert G v =
 let
 val shortestpathtoV = find(originalSP,v)
                                            (*retrieve the sequence of vertices*)
 val index = length(shortestpathtoV)-1
 val secondtoLastNode = (nth shortestpathV index,v)
 val newG = erase G secondtoLastNode
                                                     (*erases the edge from G
                                                     note this eras is not
                                                      Table.erase*)
 val newSP = SP(newG)                              (*creates new shortest path*)
 val shortestpathL = length(find(newSP,v))
                                          (*gets the length of new shortest path*)
 in
 if(shortestpathL - 1 <= length(shortestpathtoV)
                                                       (*compares the two lengths*)
 then true
 else false
 end
in
reduce (fn (x,y) = x andalso y)
       (true)
       (map (fn i => doOnEachVert G i)(listVertices))
end
```

Task 6.3
So in our new algorithm we would be calling $SP(G)$ only once. Essentially our algorithm would be checking for alternate paths and our a path that is of the original length plus one. In order to do this we would have to iterate over all nodes, but the trick would be not to be calling $SP(G)$ each time. We can do this by verifying if there is an edge or not which we can do from our original G. Our SP will return a table of sequence of vertices. So our code would look like this

```
fun fasterRobustVerf G allVerts =
let
val spG = SP(G) (*Get all shortest path from node*)
fun doOnEachVert v =
 let
 val shortpath = find(spG)(v) (*retrieve the shortest path to v*)
 val d = length(shortpath)
 val secondToLast = nth shortpath d-1 (*Get second to last vertex*)
 val rightlength = (fn x => (length(x) = d || length(x) = (d-1)))
```

```
                              (*check lengths*)
       val withoutSTL = (fn x => if(x = secondToLast)
                                 then false
                                 else true) (*take out secondToLast*)
       val filterlength = filter rightlength spG
       val finfilter = filterk withoutSTL spG
       val check = (fn i => findEdge(G)(i,v) of
                              (*findEdge returns true if there is an edge
                                present else it returns false*)
       val leftoverkeys = toSeq(domain finfilter)
       val result = filter check leftoverkeys
       in
       if(length(result) > 0)
       then true
       else false
       end
      in
      reduce (fn (x,y) => x andalso y)
             (true)
             (map doOnEachVert allVerts)
      end
```

Task 6.4
This algorithm would be the same as the previous one in that we are first verifying if there is such a path to a vertex that if one of the edges gets blocked you could still reach that same vertex in just a minute more than usual. So we would go down through each vertex and make sure that there is an edge if there isn't an edge then we create an edge and that would be one of the roads that we created. There would always be an a free vertex that we could create an edge for. We know this because first we recognize that the worst case of a unrobust graph is a graph with the minimal amount of edges possible in a simple connected graph. The smallest amount of edges possible in a simple connected graph would be $n-1$ if there are n vertices. The only two configurations of this is a straight line and single vertex that has an edge to all other vertices. These two configuration can also be turned into a robust graph.