Homework 7 Solutions

Problem R-6.9

See slide 22 of the Directed Graphs lecture.

Problem C-6.13

The two algorithms shown below work together to compute, for each switching station, the switching stations within a 4 hop radius.

Algorithm VideoNetwork-Design(G)
Input: An undirected graph G of n vertices and m edges. The vertices represent switching stations and the edges high speed links.
Output: Graph G with each vertex storing the vertices reachable in 4 hops.

for each vertex v in G
    VideoNetwork-BFS(G,v)

Algorithm VideoNetwork-BFS(G,s):
Input: A graph G and a vertex s of G
Output: The graph G with vertex s storing the vertices reachable in 4 hops.

create an empty container L_0
i <- 0
for each vertex v in G
    mark v as unexplored
mark s as explored
add s to the end of L_0
while L_i is not empty and i <= 4 do
    create an empty container L_{i+1}
    for each vertex v in L_i do
        add v to the s’s set of reachable vertices
        for all edges e in G.incidentEdges(v) do
            let w be the other endpoint of e
            if vertex w is unexplored then
                mark w as explored
                add w to the end of L_{i+1}
    i <- i + 1
The VideoNetwork-BFS algorithm is a slightly modified version of the BFS algorithm presented in Section 6.3.3 of Goodrich-Tamassia. The major difference is that the algorithm explicitly resets the state of each vertex. Since this process can be performed in $O(n)$ time, the entire algorithm will still run in $O(n + m)$ time. Since the VideoNetwork-BFS function is called once for each vertex, the running time of the VideoNetwork-Design algorithm is $O(n^2 + nm)$.

**Problem C-6.19**

Algorithm FindEulerTour(G):
Input: A directed graph G which is connected and for each vertex the in-degree equals the out-degree
Output: A Euler tour of G

```
T <- empty cycle
V <- G.vertices()
while V is not empty
  v <- V.aVertex()
  C <- FindCycle(G,v)
  T <- MergeCycle(T,C,v)
  V <- T.vertices()
  remove all edges in C from G
  remove all unconnected vertices in G from V
return T
```

Algorithm FindCycle(G,v):
Input: A directed graph G and a vertex v
Output: A cycle in G

```
C <- an empty cycle
w <- v
e <- v.anEdge()
do
  C.add(w)
  C.add(e)
  w <- G.opposite(w,e)
  e <- w.anEdge()
while v != w
return C
```

Algorithm MergeCycle(X,Y,v):
Input: Cycles X and Y, vertex v
Output: A single cycle containing all of the vertices and edges in X and Y

```
T <- empty cycle
W <- X.vertices() U Y.vertices()
while W is not empty
  w <- W.aVertex()
  C <- FindCycle(G,w)
  T <- MergeCycle(T,C,v)
  W <- T.vertices()
  remove all edges in C from G
  remove all unconnected vertices in G from W
return T
```
if X is empty
    return Y
else if Y is empty
    return X
else
    return merge X and Y at v

In the FindEulerTour algorithm above, each edge in the graph $G$ is processed exactly once. For each edge, a constant number of operations are performed. Therefore the algorithm runs in $O(m)$ time.

**Problem C-7.8**

The algorithm below is a slightly modified version of Barůvka's algorithm. The maximum weight edge is chosen instead of the minimum weight edge. The running time of the algorithm is unaffected by this change and is therefore $O(m \log n)$.

**Algorithm MaximumBandwidthSpanningTree(G):**

Input: A simple, undirected graph $G$ with $n$ vertices and $m$ edges. Each vertex represents a station and each edge represents a channel. The weight of an edge $e$, $w(e)$, is the bandwidth of the corresponding channel.

Output: A spanning tree $T$ with the maximum bandwidth

1. $T \leftarrow$ vertices of $G$
2. while $T$ has fewer than $n-1$ edges do
   a. for each connected component $C$ in $T$ do
      i. $e \leftarrow$ maximum-weight edge from $C$ to another component in $T$.
      ii. if $e$ is not already in $T$ then
          Add edge $e$ to $T$
3. return $T$
Program

/*
 * File: CtpConstants.java
 *
 * A set of constants
 *
 * @author James Lentini
 */

public interface CtpConstants
{
    public static final byte LOCATION_A = 0;
    public static final byte LOCATION_B = 1;
    public static final byte LOCATION_C = 2;

    public static final byte NUM_LOCATIONS = 3;

    public static final byte MEN_MASK = (byte) 0x55;
    public static final byte WOMEN_MASK = (byte) 0xAA;
}

/*
 * File: CtpAlgorithm.java
 *
 * @author James Lentini
 */
import java.util.Hashtable;
import java.util.Vector;

public class CtpAlgorithm implements CtpConstants
{
    public static void main(String[] args)
    {
        findSolution();
    }

    public static void findSolution()
    {
        Hashtable ht = new Hashtable();
        Vector vec = new Vector();
        byte[] locations = {(byte)~0, 0, 0};
        CtpState start = new CtpState(LOCATION_A, locations, null);
ht.put(start.toString(), start);
vec.add(start);

while ( 0 != vec.size() )
{
    CtpState s = (CtpState) vec.remove(0);
    if ( s.isEnd() )
    {
        printSolution(reconstructPath(s));
        System.exit(0);
    }

    Vector next_vec = s.getNextStates();
    for (int i = 0; i < next_vec.size(); i++)
    {
        CtpState t = (CtpState) next_vec.get(i);
        // if the state has not been found
        if ( null == ht.get(t.toString()) )
        {
            ht.put(t.toString(), t);
            vec.add(t);
        }
    }
}

System.out.println("No Solution Found");
}

public static Vector reconstructPath(CtpState end)
{
    Vector path = new Vector();

    for (CtpState s = end; null != s; s = s.getPrevState() )
    {
        path.add(0, s);
    }

    return path;
}

public static void printSolution(Vector path)
for ( int i = 0; i < path.size(); i++)
{
    System.out.println(path.get(i));
}

import java.util.Vector;

public class CtpState implements CtpConstants
{
    private byte canoe_location_;
    private byte[] locations_;
    private CtpState prev_state_;

    public CtpState(byte canoe_location,
                     byte[] locations,
                     CtpState prev_state)
    {
        canoe_location_ = canoe_location;
        locations_ = new byte[NUM_LOCATIONS];
        System.arraycopy(locations, 0, locations_, 0, NUM_LOCATIONS);
        prev_state_ = prev_state;
    }

    public CtpState getPrevState()
    {
        return prev_state_;
    }

    public boolean isEnd()
    {
        return ((LOCATION_C == canoe_location_ ) &&
            ( ((byte)~0) == locations_[LOCATION_C] ) );
    }
public boolean areMenPresent(byte location) {
    return (0 != (MEN_MASK & location));
}

generated the code for areWomenPresent method

generated the code for isValidLocation method

generated the code for idToCoupleNum method

generated the code for isPresent method
    return (0 != ((0x1 << id) & location));
}

public byte addIdToLocation(int id, byte location)
{
    return (byte)((0x1 << id) | location);
}

public byte removeIdFromLocation(int id, byte location)
{
    return (byte)((0x1 << id) ^ location);
}

public String toString()
{
    String val = "["
    for (int i = 0; i < NUM_LOCATIONS; i++)
    {
        if (i == canoe_location_) { val += "x"; }
        val += locationToString(locations_[i]);
        if (i != NUM_LOCATIONS - 1) { val += ","; }
    }
    val += "]";
    return val;
}

public String locationToString(byte location)
{
    String val = "";

    boolean is_man = true;
    for (int i = 0; i < 8; i++)
    {
        if (isPresent(i,location))
        {
            if (is_man) { val += "H" + idToCoupleNum(i); }
            else { val += "W" + idToCoupleNum(i); }
        }
        is_man = !is_man;
    }
public Vector getNextStates()
{
    if ( LOCATION_A == canoe_location_ )
    {
        Vector x = getNextStates(LOCATION_A, LOCATION_B);
        x.addAll(getNextStates(LOCATION_A, LOCATION_C));
        return x;
    }
    else if ( LOCATION_B == canoe_location_ )
    {
        Vector x = getNextStates(LOCATION_B, LOCATION_A);
        x.addAll(getNextStates(LOCATION_B, LOCATION_C));
        return x;
    }
    else // LOCATION_C == canoe_location_
    {
        Vector x = getNextStates(LOCATION_C, LOCATION_B);
        x.addAll(getNextStates(LOCATION_C, LOCATION_A));
        return x;
    }
}

public Vector getNextStates(byte from_index, byte to_index)
{
    Vector v = new Vector();

    // for each person at the from location
    for ( int i = 0; i < 8; i++ )
    {
        byte from_location = locations_[from_index];
        byte to_location = locations_[to_index];

        if ( isPresent(i,from_location) )
        {
            from_location = removeIdFromLocation(i,from_location);
            to_location = addIdToLocation(i,to_location);

            if ( isValidLocation(from_location) && isValidLocation(to_location) )
            {
            
            }
        }
    }
}
addState(v, from_index, to_index, from_location, to_location);
}

// for every other person at the from location
for ( int j = 0; j < 8; j++ )
{
    if ( i != j )
    {
        byte from_location2 = from_location;
        byte to_location2 = to_location;

        if ( isPresent(j, from_location2) )
        {
            from_location2 = removeIdFromLocation(j, from_location2);
            to_location2 = addIdToLocation(j, to_location2);

            if ( isValidLocation(from_location2) && isValidLocation(to_location2) )
            {
                addState(v, from_index, to_index, from_location2, to_location2);
            }
        }
    }
}

return v;

public void addState(Vector v,
                        byte from_index, byte to_index,
                        byte from_location, byte to_location)
{
    byte[] next_locations = new byte[NUM_LOCATIONS];
    System.arraycopy(locations_, 0, next_locations, 0, NUM_LOCATIONS);
    next_locations[from_index] = from_location;
    next_locations[to_index] = to_location;
    v.add(new CtpState(to_index,
                        next_locations,
                        this));
}