CS 344 Assignment 3

Craig Perkins, Alex Tang, Steve Grzenda

Due April 4, 2014

Problem 1

Assume we have a function `dist(a,b)` which returns the distance between two pairs of coordinates in constant time and a function `min(a,b)` which returns the minimum of `a` and `b` in constant time.

```
// red[], blue[] are arrays of coordinates
function Takeshi(red[], blue[]):
  redIndex = 0
  blueIndex = 0
  minRope = dist(red[0], blue[0])
  while (redIndex != length(red) and blueIndex != length(blue)):
    bothMove  = dist(red[redIndex + 1], blue[blueIndex + 1])
    redMoves  = dist(red[redIndex + 1], blue[blueIndex])
    blueMoves = dist(red[redIndex], blue[blueIndex + 1])
    if (bothMove <= minRope):
      redIndex++
      blueIndex++
    else if (redMoves <= minRope):
      redIndex++
    else if (blueMoves <= minRope):
      blueIndex++
    else:
      minRope = min(redMoves, blueMoves)
  return minRope
```

This algorithm will return minimum length of rope needed. It will run in worst case $O(n+m)*O(1)$ where n is the length of the red path and m is the length of the blue path and the $O(1)$ is the cost of calling `dist` on two points.

Problem 2

A. A greedy approach to the chopstick matching problem could be to start by sorting the chopsticks from least length to greatest length. The sorting can be achieved using quick sort which runs in $O(n \log n)$ time. After sorted we can make the greedy choice to try and pair the first 2 chopsticks in the sorted set. If they can make a pair then we will continue to the next 2 chopsticks and try to make them a pair, otherwise we will take the second from the chopsticks and compare it to the next chopstick in the list. Let the sequence of sorted chopsticks be $\{l_1, l_2, \ldots, l_i, l_{i+1}, \ldots, l_n\}$, the algorithm described will make the greedy choice and compare l_i and l_{i+1}. If $l_{i+1} - l_i < k$, then the algorithm will mark them as a pair and then continue with comparing l_{i+2} and l_{i+3}, otherwise it will compare l_{i+1} and l_{i+2}. In the worst case, no chopsticks will be able to be matched and there will be at most $n - 1$ comparisons for a total running time of $O(n \log n) + O(n - 1)$.

B. To solve this variant on the problem, imagine constructing a graph where the nodes of the graph are the chopsticks and a weighted edge exists between 2 nodes if the chopsticks can be paired together. The weight of this edge will correspond to the sum of the weights of the 2 chopsticks. Now the problem has been reduced to finding the maximum weighted matching problem. There is a solution to the maximum weighted matching problem known as the Blossom Algorithm which runs in $O(n^2)$ time.

Problem 3

A. In the worse case, if there is an exponential number of paths, then in order to output all of them, you will need an exponential amount of time. If you were to save these paths, it would take exponential memory space. The following algorithm returns a set of all the paths from `s` to `t`:

```
// path is defined as a list from source s to destination t, i.e.:
//   path = (s, ... , t)
// path.last() returns the last vertex in the path list, i.e.:
//   (s, ..., t).last() returns t
// path.extend(v) means add v to the end of the path, i.e.:
//   (s1, s2).extend(v) returns (s1, s2, v)

function find_all_paths(G, s, t):
```

```
all_paths = {}
queue = { (s) }

while (queue is not empty):
  path = queue.remove()
  u = path.last()
  for every edge(u, v) in G:
    if (v == t):
      all_paths.add(path)  // reached destination t
    else:
      queue.add(path.extend(v))
return all_paths
```

B. Topologically sort the graph and check if there is an edge between each consecutive vertex. This algorithm will run in $O(|V| + |E|)$.

```
function path_exists(G):
  H = topological_sort(G)
  // vertexes are now labeled 1, 2, ..., n in topological order
  for i in [1, n - 1]:
    if (edge(i, i+1) does not exist in G):
      return false
  return true
```

Problem 4

Given the list of international conflicts in the last ten years, create a graph where each vertex represents a country. Create an edge between two vertices if there is a conflict between those two countries. Next, check if the graph `G` is bipartite using the following algorithm:

```
// assume every vertex v can be uniquely referenced by v.number
function is_bipartite(G):
  color_array[]  // size of color_array = |V|
  for i in [1, |V|]:
    color_array[i] = none

  color = red
  source = get_any_vertex(G)  // can start check from any vertex in G
  color_array[source.number] = color
```

```
  queue = { source }

  red_set = {}
  blue_set = {}

  while (queue is not empty):
    u = queue.remove()

    if (color is red):
      color = blue
      set = blue_set
    else:
      color = red
      set = red_set

    for every edge(u, v) in G:
      if (color[v.number] is none):
        color_array[v] = color
        set.add(v)
        queue.add(v)
      else if (color[v.number] is not color):
        return false

  return true

// upon success,
// red_set will contain all the vertices colored red
// blue_set will contain all the vertices colored blue
```

If the function returns true, then the graph is bipartite and it is possible to make the assignment. One of the tables will contain all of the vertices from `red_set` and the other from `blue_set`. Creating a representation of the graph will take $O(|V| + |E|)$ and `is_bipartite`, which runs BFS, will run in $O(|V| + |E|)$. Overall, the solution will run in linear time.