CS106B
Winter 07-08

Handout #35S
March 9, 2008

Section Solutions #9

Problem 1: Choosing a Good Hash Function

(a) The table has 2048 buckets and yet this hash function will only place things in 26 of those buckets. And even within those 26 buckets, we will tend to get clustering around the more popular choices for the first letter of peoples' names.

(b) Like above, this hash function will only use part of the available buckets, because the product of three digits only produces values is the range 0 to 729. Further, clustering will occur around some values such as 0 and no primes greater than 7 will get used. Notice our range only has 1000 values and our table has 1000 buckets, why not just map each number to the bucket with that same index. Simpler and very efficient.

(c) One important feature of a hash function is that it must be stable. A given value must hash to the same result each time. Using a random number generator in this hash function means a value can hash to something different each time it is hashed. This is completely unacceptable for a hash function, because we will not be able to reliably find the bucket under which we previously stored an item!

Problem 2: Understanding graph algorithms

a) `Lounge, Conservatory, Ball Room, Billiard Room, Library, Hall, Dining Room, Kitchen, Study`

b) `Kitchen, Ball Room, Dining Room, Study, Billiard Room, Conservatory, Hall, Lounge, Library`

c) Here's what we do:

```
Fix distance to Lounge at 0
Process the arcs out of Lounge (Conservatory, DiningRoom, Hall)
  Enqueue the path: Lounge -> Conservatory (3)
  Enqueue the path: Lounge -> DiningRoom (4)
  Enqueue the path: Lounge -> Hall (8)
Dequeue the shortest path: Lounge -> Conservatory (3)
Fix distance to Conservatory at 3
Process the arcs out of Conservatory (BallRoom, BilliardRoom, Lounge)
  Enqueue the path: Lounge -> Conservatory -> BallRoom (7)
  Enqueue the path: Lounge -> Conservatory -> BilliardRoom (10)
  Ignore Lounge because its distance is fixed
Dequeue the shortest path: Lounge -> DiningRoom (4)
Fix distance to DiningRoom at 4
Process the arcs out of DiningRoom (BallRoom, Hall, Kitchen, Lounge)
  Enqueue the path: Lounge -> DiningRoom -> BallRoom (11)
  Enqueue the path: Lounge -> DiningRoom -> Hall (12)
  Enqueue the path: Lounge -> DiningRoom -> Kitchen (15)
  Ignore Lounge because its distance is fixed
Dequeue the shortest path: Lounge -> Conservatory -> BallRoom (7)
Fix distance to BallRoom at 7
Process the arcs out of BallRoom (BilliardRoom, Conservatory, DiningRoom,
Kitchen)
   Enqueue the path: Lounge -> Conservatory -> BallRoom -> BilliardRoom
(14)
   Ignore Conservatory because its distance is fixed
   Ignore DiningRoom because its distance is fixed
   Enqueue the path: Lounge -> Conservatory -> BallRoom -> Kitchen (14)
 Dequeue the shortest path: Lounge -> Hall (8)
 Fix distance to Hall at 8
 Process the arcs out of Hall (DiningRoom, Library, Lounge, Study)
   Ignore DiningRoom because its distance is fixed
   Enqueue the path: Lounge -> Hall -> Library (15)
   Ignore Lounge because its distance is fixed
   Enqueue the path: Lounge -> Hall -> Study (12)
 Dequeue the shortest path: Lounge -> Conservatory -> BilliardRoom (10)
 Fix distance to BilliardRoom at 10
 Process the arcs out of BilliardRoom (BallRoom, Conservatory, Library)
   Ignore BallRoom and Conservatory because their distances are fixed
   Enqueue the path: Lounge -> Conservatory -> BilliardRoom -> Library (14)
 Dequeue the shortest path: Lounge -> DiningRoom -> BallRoom (11)
 Ignore this path because the distance to the BallRoom is fixed
 Dequeue the shortest path: Lounge -> DiningRoom -> Hall (12)
 Ignore this path because the distance to the Hall is fixed
 Dequeue the shortest path: Lounge -> Hall -> Study (12)
 Fix distance to Study at 12
 Process the arcs out of Study (Hall, Kitchen, Library)
   Ignore Hall because its distance is fixed
   Enqueue the path: Lounge -> Hall -> Study -> Kitchen (15)
   Enqueue the path: Lounge -> Hall -> Study -> Library (19)
 Dequeue the shortest path: Lounge -> Conservatory -> BallRoom ->
BilliardRoom (14)
 Ignore this path because the distance to the BilliardRoom is fixed
 Dequeue the shortest path: Lounge -> Conservatory -> BallRoom -> Kitchen
(14)
 Fix distance to Kitchen at 14
 Process the arcs out of Kitchen (BallRoom, Study, DiningRoom)
   Ignore BallRoom because its distance is fixed
   Ignore DiningRoom because its distance is fixed
   Ignore Study because its distance is fixed
 Dequeue the shortest path: Lounge -> Conservatory -> BilliardRoom ->
Library (14)

 Shortest path: Lounge -> Conservatory -> BilliardRoom -> Library (14)
```

d)

Add arc between Conservatory and Lounge	3
Add arc between Kitchen and Study	3
Add arc between Ball Room and Conservatory	4
Add arc between Billiard Room and Library	4
Add arc between Dining Room and Lounge	4
Add arc between Hall and Study	4
Add arc between Ball Room and Billiard Room	6
Discard arc between Ball Room and Dining Room (forms cycle)	X
Add arc between Ball Room and Kitchen	7

Problem 3: Finding Graph Cycles

```
bool RecCyclicCheck(graphT & graph, nodeT * node, Set<nodeT *> & visited)
{
  if (visited.contains(node))
  {
    return true;
  }

  Set<arcT *>::Iterator arcIter = node->connected.iterator();
  bool result = false;

  visited.add(node);
  while (!result && arcIter.hasNext())
  {
    if (RecCyclicCheck(graph, arcIter.next()->end, visited))
    {
      result = true;
    }
  }
  visited.remove(node);

  return result;
}

bool IsCyclicGraph(graphT & graph)
{
  Set<nodeT *>::Iterator iter = graph.allNodes.iterator();
  Set<nodeT *> visited;
  //We need to check starting from each node, since the graph may not be
  //connected
  while(iter.hasNext())
  {
      Set<nodeT *> currVisited;
      nodeT * currStart = iter.next();
      if(!visited.contains(currStart) &&
        RecCyclicCheck(graph, currStart, currVisited))
            return true;
      visited.unionWith(currVisited);
  }
  return false;
}
```

Problem 4: Graph Construction

a)

```
void createNode(string word, graphT & graph)
{
      nodeT * newNode = new nodeT;
      newNode->name = word;
      graph.allNodes.add(newNode);
}
```

```
void ConstructGraph(graphT & graph, Lexicon & lex)
{
      lex.mapAll<graphT>(createNode, graph);

      Map<nodeT *> wordMap;
      Set<nodeT *>::Iterator iter = graph.allNodes.iterator();

      while(iter.hasNext())
      {
            nodeT* currNode = iter.next();
            wordMap.add(currNode->name, currNode);
      }

      iter = graph.allNodes.iterator();

      while(iter.hasNext())
      {
            nodeT* currNode = iter.next();
            string currWord = currNode->name;
            for(int i = 0; i < currWord.length(); i++)
            {
                  for(int replaceChar = ‘a’; replaceChar <= ‘z’; replaceChar++)
                  {
                        string newWord = currWord.substr(0,i) + (char)replaceChar
                              + currWord.substr(i+1);
                        if(lex.containsWord(newWord) && newWord != currWord)
                        {
                              nodeT* otherNode = wordMap.getValue(newWord);

                              arcT * edge = new arcT;
                              edge->start = currNode;
                              edge->end = otherNode;
                              currNode->connected.add(edge);
                        }
                  }
            }
      }
}
```

b) We want to find the shortest path here (not just any path), so breadth-first search is more appropriate. Depth-first search can be written to find the shortest path, but it will have to examine a much larger number of paths in order to do so.

```
Stack<nodeT *> FindShortestLadder(graphT & graph, nodeT * start, nodeT * end)
{
      Queue<Stack<nodeT *> > ladders;
      // No comparison function; uses pointers, so we want default compare
      Set<nodeT *> visited;

      Stack<nodeT *> startLadder;
      startLadder.push(start);
      visited.add(start);
      ladders.enqueue(startLadder);

      while(!ladders.isEmpty())
      {
            Stack<nodeT *> currLadder = ladders.dequeue();
            nodeT * currNode = currLadder.peek();
            if(currNode == end)
                  return currLadder;
            Set<arcT *>::Iterator iter = currNode->connected.iterator();
            while(iter.hasNext())
            {
                  arcT * currArc = iter.next();
                  if(!visited.contains(currArc->end))
                  {
                        Stack<nodeT *> newLadder = currLadder;
                        newLadder.push(currArc->end);

                        ladders.enqueue(newLadder);
                        visited.add(currArc->end);
                  }
            }
      }
      Stack<nodeT *> emptyStack;
      return emptyStack;
}
```

c) In this case, both breadth-first and depth-first searches will need to visit every single path to find the longest. However, breadth-first search uses much more memory than depth-first since it must keep track of all of the current paths. Therefore, while both search algorithms will solve the problem, depth-first search is a better choice.

Note that this solution passes the visited set and the current path by reference. This is not strictly necessary, but does save a bit of memory. However, because we do this we have to remember to unmodify these data structures when we are finished with a call.

```
void RecFindLongest(nodeT * node, Set<nodeT *> & visited,
            Stack<nodeT *> & current, Stack<nodeT *> & longest)
{
      if(visited.contains(node))
            return;

      current.push(node);
      visited.add(node);

      if(current.size() > longest.size())
            longest = current; //makes a deep copy

      Set<arcT *>::Iterator iter = node->connected.iterator();

      while(iter.hasNext())
      {
            arcT * currArc = iter.next();

            RecFindLongest(currArc->end, visited, current, longest);
      }

      //Have to unmodify current & visited since we are passing them by reference
      current.pop();
      visited.remove(node);
}

Stack<nodeT *> FindLongestLadder(graphT & graph, nodeT * word)
{
      Set<nodeT *> visited;
      Stack<nodeT *> current;
      Stack<nodeT *> longest;

      RecFindLongest(word, visited, current, longest);

      return longest;
}
```