Project 3: Dynamic Programing Report

Due: Thursday 12/3/2015

Chris Frazier U58831412

Connor McGaughey U54395119

John Gangemi U68714612

---

1. *Describe in English how you can break down the larger problem into one or more smaller problem(s). This description should include how the solution to the larger problem is constructed from the subproblems.*

A naive approach to finding the longest path in a graph would be to try all combinations of paths for each vertex and store the intermediate results in a data structure. While correct, the naive algorithm is inefficient and slow compared to other, more viable methods.

For any path in the graph, there must exist at least a single edge. The longest path will be the accumulation of the greatest number of edges. Therefore, assumptions can be made as to what constitutes a valid path...

A. For any path from **i** to **j** there must exist edges from vertex **i** to vertex **j**.
B. Ignore any path from **i** to **j** that encounters a nonexistent edge.

Therefore, to find the longest path in a graph one must find all valid paths to correctly identify the longest path. Identifying the longest path can occur after all valid paths have been cataloged or during the accumulation of valid paths.

---

2. *What recurrence can you use to model the problem using dynamic programming?*

Given all edges for a directed graph where every path moves from a lower vertex to a higher vertex, the longest path can be found by recursively calling an algorithm for every possible set of starting and ending vertices. This algorithm, called recursively, will find the longest path from the given starting vertex to the given ending vertex. The number of edges in each path can be stored as the algorithm recursively executes and then analyzed afterward to find the largest path.

The base case in this instance would be the last possible path for the given set of vertices. Once the size of the last path is found, the base case is met, and the next set of vertices is analyzed.

---

3. *Prove that your recurrence is correct.*

Assume the recursive algorithm I just described will **store each path size** from the first given starting vertex to the first given ending vertex. The algorithm will keep calling itself with a different path to check until **every possible path is checked** (the base case is reached). Assume the algorithm will stop recursing at this point. The loop that first called the algorithm will then move onto the next set of vertices and call the algorithm again until **every possible set of vertices is recursively checked for path sizes**. The stored list of path sizes will then be iterated over to **find the largest total path**.

Since every possible set of vertices is being checked, and every possible path size between those vertices is being stored, then **it is impossible that we won't find the largest path** when we iterate over this list.

---

4. *Describe a pseudocode algorithm that uses memoization to solve the problem.*

All edges in a graph must be represented in a data structure, in particular a two-dimensional array. An edge's adjacent vertices are used as indices to appropriately set a cell in the two dimensional array to one.

```
Input: numberOfEdges
Input: edges[]
Input: memo[][]
Algorithm: BuildMemo

for i = 0 to numberOfEdges - 1
{
    source = source vertex of edges[i] - 1;
    target = target vertex of edges[i] - 1;

    memo[source][target] = 1;
}
return;
```

Memoization involves using recursion to traverse a data structure while storing calculation results into a global solution array. Our solution below checks to see if the

current two vertices have a path connecting them. If not, our base case will cause the algorithm to stop recursing calls itself with the next cell in the graph. This ensures every possible path length is checked. At the end of the function, if the longest path is found, it is appended to the solution string.

```
int Graph::memoPath(int i, int j){
    int vertex1LongestPath;
    int vertex2LongestPath;
    if (i == j) return 0; //Base case
    for (int k = 1; k <= j - (i + 1); k++)
    {
        if (left cell path length == 1) vertex1LongestPath = left cell path length
        else vertex1LongestPath = memoPath(i, j - k);

        if(below cell path length == 1) vertex2LongestPath = below cell path length;
        else vertex2LongestPath = memoPath(j - k, j);

        if (vertex1LongestPath != 0 && vertex2LongestPath != 0)
        {
            int temporaryPath = vertex1LongestPath + vertex2LongestPath;
            if ( temporaryPath > current cell path length)
            {
                current cell path length = temporaryPath length
                if (current cell path lentgh > longestPath)
                    longestPath = current cell path length

                append the path to solution string
            }
        }
    }
    set current path as visited
    return current path length
}
```

---

5. *Analyze the complexity of your memoized algorithm.*

Traversal of the two-dimensional array begins in the top-most right cell and recursively moves to the left in the current row until the base case is satisfied. Next, traversal is performed down recursively in the current column until the base case is satisfied. Every element in the upper diagonal of the two-dimensional array will be visited once, this takes O(n) time worst-case. Recursion will continuously store the output using memoization, this technique reduces the number of computations drastically. However, once visiting a cell, there are k + n traversals that need to be computed additionally. Therefore, overall worst-case complexity is O(n) * (k +n), or $O(n^2 + k)$.

---

6. *Describe a pseudocode algorithm that solves the problem iteratively (using dynamic programming). Your algorithm should be optimal in terms of its time and space complexity.*

To solve the problem iteratively using dynamic programming, all edges in a graph must be represented in a data structure, in particular a two-dimensional array. An edge's adjacent vertices are used as indices to appropriately set a cell in the two dimensional array to one.

```
Input: numberOfEdges
Input: edges[]
Input: memo[][]
Algorithm: BuildMemo

for i = 0 to numberOfEdges - 1
{
    source = source vertex of edges[i] - 1;
    target = target vertex of edges[i] - 1;

    memo[source][target] = 1;
}
return;
```

After initializing the two-dimensional array with edges from a graph, traversal of the upper-right diagonal elements will find the longest path that exist in a graph. If possible, the path from **i** to **j** is a set vertices indicated by the *source* variable and a set of edges indicated by the *count* variable. Once a valid path has been found, its size is checked against the previous path's size to determine the maximum path, if the path is a maximum then the longest path becomes the current path. Next, the cell with indices **i** and **j** is changed to the value of count, representing the number of edges in a path from vertex **i** to vertex **j**. Thus, the cell with the greatest value is longest path from vertex **i** to vertex **j**.

```
Input: numberOfVertices
Input: memo[][]
Output: longestPath
Algorithm: FindLongestPath

longestPathSize = 0;

for i = 0 to numberOfVertices - 1
{
    for j = i + 2 to numberOfVertices - 1
    {
        path = empty array of integers;
        source = i;
        target = i + 1;
        count = 0;

        while target <= j
        {
            if memo[source][target] is 1
            {
                add source + 1 to path;
                source = target;
                target = source + 1;
                increment count by 1;

                if source == j
                {
                    add source + 1 to path;

                    if size of path > longestPathSize
                    {
                        longestPathSize = size of path;
                        longestPath = path;
                    }

                    memo[i][j] = count;
                    break;
                }
            }
        }
    }
}
return longestPath;
```

Printing the number of edges and edge ids is a trivial task. The number of edges in the longest path is the number of vertices (size of longest path) minus one. Iterating through all vertices in the longest path will find appropriate edges, and consequently their ids. The results are stored and returned once all edges have been identified.

```
Input: edges[]
Output: result
Algorithm: PrintLongestPath

BuildMemo();
longestPath = FindLongestPath();

result = size of longestPath - 1;

for i = 0 to size of longestPath - 1
{
    source = longestPath[i];
    target = longestPath[i + 1];

    foreach edge in edges
    {
        if source == source vertex of edge and
           target == target vertex of edge
        {
            result += edge id;
        }
    }
}
return result;
```