CS 301 - Algorithms

Project Progress Report

Problem Title: Longest Circuit Problem

Group Number: Group #17

Group Members: Ali Arda Girgin (26993)

Elif Aydın (28061)

Kaan Atmaca (28239)

Kamil Atakan Çelikyürek (26896)

Table of Contents:

1. Problem Description

A. Basic Definition

The basic definition of our problem is to find the length of the longest simple cycle in an undirected and unweighted graph with the given vertices and edges. A simple cycle is a chain in which both the initial and end vertices are linked. Each vertex of the simple cycle will be visited precisely once if you go along it. We must obtain a simple path, which means we will not have any repeated edge or a vertex. It will be a cycle where the starting and ending vertex are the same. Therefore, the goal is to find for all u, v ϵ V, the longest path from u to u using maximum number of edges.

B. Real Life Application

The algorithm behind finding the longest circuit problem can be used in Supply Chain Management. Supply Chain Management is the management of the flow of goods and services. It aims to cut excess cost and deliver the products to the consumer in an efficient way. For example, a routing plan is about to be made for delivering a certain item. Vehicle has a starting point and wants to visit as many customers as possible to deliver items and return back to the starting point to recharge. We can take customers' location and starting point as vertices and roads as connected edges. Then we can use the longest circuit problem algorithm to find a route which maximizes the number of customers visited. Also, we can use the algorithm for constructing rail and road networks. Consider we have different train stops in a given city. We want to make a rail network which is cyclic and maximizes the number of trains stops included on the rail network. We can use the algorithm for maximizing the number of trains stops in this cyclic rail network.

2. Algorithm Description & Implementation

There are only a few algorithms which have polynomial time complexity are known for the longest circuit problem for special classes of graphs.[1]

As the algorithm for the longest circuit problem, we are going to use an algorithm that has a $O(V!)$ time complexity.

The steps of our algorithm are as follows:

1. Enumerate all vertices of graph and form an array which has size of all vertices to store visit condition of all nodes.

2. Create an empty stack and push the initial vertex to the stack. Also, create a list of stacks to save possible paths.

3. Apply LSC algorithm to find the longest path.

3.1. First, assign current vertex as the vertex in top of the stack.

3.2. If current vertex is initial vertex and is visited, then return from function call.

3.3. Mark the current vertex as visited.

3.4. For each of vertices that are adjacent to current vertex apply the following procedure.

3.4.1. If adjacent vertex is either not visited or adjacent vertex is the start vertex; and not equal to predecessor of current vertex, then push this adjacent vertex to the top of stack.

3.4.2. Recursively apply the LSC algorithm.

3.4.3. After recursive operations are done, check the top of the stack, whether it is the initial vertex or not; if it is initial vertex, save the current state of stack to the array of stacks.

3.4.4. Pop the top element of stack.

4. Compare the paths and find the path which has the most vertices. (There will be at least two paths that have the most vertices, finding one of them is sufficient.)

[1] (Kumar and Gupta 2014, 202)

The implementation of the algorithm in C++ language as follows:

```
01. void LongestSimpleCircuit(int start, int last, vector<int> isVisited, const vector<vector<int>>
      &adjacencyMatrix, vector<Stack<int>> &paths, Stack<int> currentStack)
02. {
03.   int current = currentStack.getTop();
04.   if (current == start && isVisited[current] == 1)
05.   {
06.     return;
07.   }
08.   isVisited[current] = 1;
09.   for (int index = 0; index < (int)adjacencyMatrix[current].size(); index++)
10.   {
11.     if (adjacencyMatrix[current][index] == 1 && (isVisited[index] == 0 || index == start) &&
          last != index)
12.     {
13.       currentStack.push(index);
14.       LongestSimpleCircuit(start, current, isVisited, adjacencyMatrix, paths, currentStack);
15.       if (currentStack.getTop() == start)
16.       {
17.         paths.push_back(currentStack);
18.       }
19.       currentStack.pop();
20.     }
21.   }
22. }
```

3. Algorithm Analysis

A. Correctness

The algorithm that we discussed is not the optimal algorithm for the longest circuit problem, however, it finds the correct answer. Therefore, in the algorithm, we trade speed for correctness.

In the implementation, we started from the vertex which has the largest degree of the graph. Before each of recursive LSC procedure call, we pushed a new vertex to the stack, and after the procedure call, we pop that element, as there might be other possible vertices from any arbitrary vertex. Moreover, before each of recursive LSC procedure call, we checked for whether it is visited on the current path to prevent any infinite cycles and compared the adjacent vertex with the predecessor of current vertex to prevent iterating to the backwards of the path.

B. Time Complexity Analysis

1. Enumerate all vertices ... $\Rightarrow O(V)$
2. Create an empty stack ... $\Rightarrow O(1)$
3. Apply LSC algorithm ... $\Rightarrow O(V) \times (O(1) + T(V-1) + O(1)) + O(1) + O(1)$

3.1. First, assign current ... $\Rightarrow O(1)$

3.2. If current vertex is not ... $\Rightarrow O(1)$

3.3. Mark the current vertex ... $\Rightarrow O(1)$

3.4. For each of vertices that ... $\Rightarrow O(V) \times (O(1) + T(V-1) + O(1))$

3.4.1. If adjacent vertex is ... $\Rightarrow O(1)$

3.4.2. Recursively apply the LSC $\Rightarrow T(V-1)$

3.4.3. After recursive operations ... $\Rightarrow O(1)$

3.4.4. Pop the top element ... $\Rightarrow O(1)$

4. Compare the paths and $\Rightarrow O\left(\sum_{k=2}^{V}\binom{V}{k}\right) = O(2^V)$

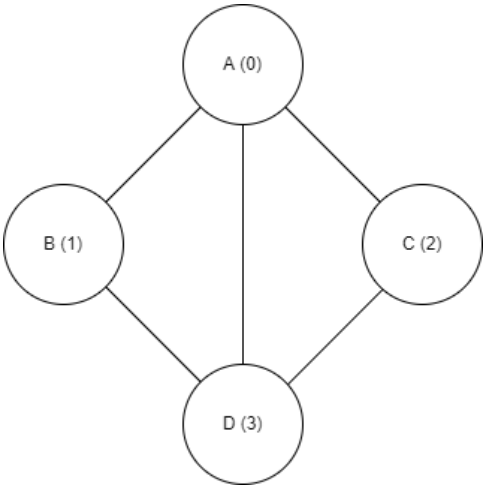
Total: $T(V) = O(1) + O(V) + O(V) \times \left(O(1) + T(V-1)\right) + O(2^V)$

$T(V) = V \times T(V-1) + 2^V = V! + 2^V = O(V!)$

4. An Initial Testing of Implementation of the Algorithm

Firstly, we implemented a random sample generator to test our algorithm. Then, we created some sample cases, and tried these cases on our C++ implementation. We will analyze these sample cases one by one. The sample cases will be analyzed as the trace of the current path, i.e., the current state of the stack.

Sample Case #1



Initial State: A (0)

A (0) → B (1)

A (0) → B (1) → D (3)

A (0) → B (1) → D (3) → A (0) ⇒ Found a path, pop the top element to find new paths

A (0) → B (1) → D (3) → C (2)

A (0) → B (1) → D (3) → C (2) → A (0) ⇒ Found a path, pop the top element to find new paths

A (0) → B (1) → D (3) → C (2) ⇒ No other possible way, pop the top element

A (0) → B (1) → D (3) ⇒ No other possible way, pop the top element

A (0) → B (1) ⇒ No other possible way, pop the top element

A (0) → C (2)

A (0) → C (2) → D (3)

A (0) → C (2) → D (3) → A (0) ⇒ Found a path, pop the top element to find new paths

A (0) → C (2) → D (3) → B (1)

A (0) → C (2) → D (3) → B (1) → A (0) ⇒ Found a path, pop the top element to find new paths

A (0) → C (2) → D (3) → B (1) ⇒ No other possible way, pop the top element

A (0) → C (2) → D (3) ⇒ No other possible way, pop the top element

A (0) → C (2) ⇒ No other possible way, pop the top element

A (0) → D (3)

A (0) → D (3) → B (1)

A (0) → D (3) → B (1) → A (0) ⇒ Found a path, pop the top element to find new paths

A (0) → D (3) → B (1) ⇒ No other possible way, pop the top element

A (0) → D (3)

A (0) → D (3) → C (2)

A (0) → D (3) → C (2) → A (0) ⇒ Found a path, pop the top element to find new paths

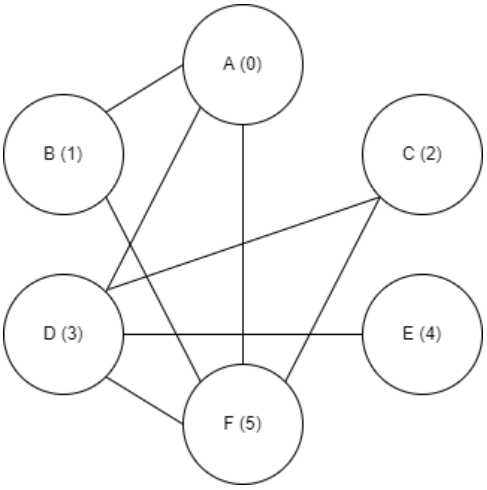
A (0) → D (3) → C (2) ⇒ No other possible way, pop the top element

A (0) → D (3) ⇒ No other possible way, pop the top element

A (0) ⇒ All paths that are started with vertex A are traced

Note that we can also start from the vertex D, as it has same degree with the vertex A.

Sample Case #2



Initial State: D (3)

D (3) → A (0)

D (3) → A (0) → B (1)

D (3) → A (0) → B (1) → F (5)

D (3) → A (0) → B (1) → F (5) → C (2)

D (3) → A (0) → B (1) → F (5) → C (2) → D (3) ⇒ Found a path, pop the top element

D (3) → A (0) → B (1) → F (5) → C (2) ⇒ No other possible way, pop the top element

D (3) → A (0) → B (1) → F (5)

D (3) → A (0) → B (1) → F (5) → D (3) ⇒ Found a path, pop the top element

D (3) → A (0) → B (1) → F (5) ⇒ No other possible way, pop the top element

D (3) → A (0) → B (1) ⇒ No other possible way, pop the top element

D (3) → A (0)

D (3) → A (0) → F (5)

D (3) → A (0) → F (5) → *B* (1) ⇒ Stuck in a vertex, pop the top element

D (3) → A (0) → F (5)

D (3) → A (0) → F (5) → C (2)

D (3) → A (0) → F (5) → C (2) → D (3) ⇒ Found a path, pop the top element

D (3) → A (0) → F (5) → C (2) ⇒ No other possible way, pop the top element

D (3) → A (0) → F (5)

D (3) → A (0) → F (5) → D (3) ⇒ Found a path, pop the top element

D (3) → A (0) → F (5) ⇒ No other possible way, pop the top element

D (3) → A (0)

D (3) → C (2) → … ⇒ Same paths will be visited, just we did not demonstrate here

D (3) → E (4) ⇒ Stuck in a vertex, pop the top element

D (3) → F (5) → … ⇒ Same paths will be visited, just we did not demonstrate here

Note that we can also start from the vertex F, as it has same degree with the vertex D.

5. References

Kumar, P., & Gupta, N. (2014). A Heuristic Algorithm for Longest Simple Cycle. *International Conference on Wireless Networks (ICWN'14)*, (pp. 202-208). Las Vegas.