The Longest Path Problem: an NP-Complete Example

Eginata Kasan / 13517030
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517030@std.stei.itb.ac.id - eginatakasan@gmail.com

***Abstract*—The longest path problem is the problem of finding a path which will result in the maximum length out of all paths possible in a given graph. Its useful applications include giving the critical path of a graph and Static Timing Analysis (STA) in electronical design automation. Unlike that of Shortest Path Problem, the longest path problem is an NP-Complete problem, except for cases where the given graphs are of directed acyclic graphs which has linear time solution.**

***Keywords*— nondeterministic polynomial; longest path; graph; algorithm**

I. Introduction

The longest path problem is the problem of finding the path with maximum length of a given graph. It searches for a vertex to start with, such that when it travels to a certain end node, it has travelled all the vertices in the graph with the sum of the distance travelled being the maximum out of all other paths in other possibilities.

One of the uses of solving the longest path in a graph is to determine the critical path of that graph. A critical path is usually used for Critical Path Analysis (CPA) to help manage scheduling of complex projects. [1] The critical path method has been used in a variety of projects, from construction, aerospace and defense to software and product development, engineering, plant maintenance and more.

The longest path problem is not as popular as its opposite, the shortest path problem. Both problems are very different, in ways that are not just the level of optimization (minimum or maximum length). There are a lot of algorithms to determine the shortest path between two points of location such as: Dijkstra's Algorithm, Breadth-First Search (BFS), the A* algorithm, etc.

The shortest path problems are fairly easy to solve and can be solved in polynomial time, so it is categorized as a P problem. Solving the longest path problem, however, is not as easy as the shortest path.

One answer may state that the answer to the longest path problem should be infinite which is caused by infinite loops/cycle in a graph. In this discussion, the definition of longest path problem is limited to finding a simple path within a graph, with simple meaning that all the vertices in a graph is visited only once, so that there are no infinite cycles to answer the longest path problem.

II. P, NP, and NP-Complete Problems

In the world of Computer Science, problems can be classified into 2 types based on the time needed to solve the problem: Polynomial-time problems (P problem) and the nondeterministic polynomial-time (NP problem). What distinguishes between P problem and NP problem depends on the time an algorithm takes to solve them. If the problem can be solved in polynomial time, it is categorized as a P problem, and if there are no known algorithm that can solve the problem in polynomial time, then it is considered an NP problem. The definition of polynomial time to classify as a P problem is [2] "if there exists a polynomial function ***p(n)*** such that the algorithm can solve any instance of size **n** in a time ***O(p(n))****"*. Generally, problems that are easy to solve and easy to verify are considered as P problems, whereas the ones that are hard to solve but easy to check are considered NP problems.

The most difficult of NP problems can also be categorized as a group called the NP-Complete problems or NPC.

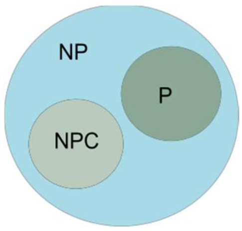


Figure 1.1 the P, NP, and NPC illustrated as Venn diagram

A few popular NP-Complete problems in Computer Science are: Travelling Salesperson Problem, Knapsack Problem, Hamiltonian Path Problem and Graph Coloring Problem. Games such as Sudoku and Minesweeper are also considered NP-Complete problems.

NP-Complete problems can represent every NP problem well. This is most interesting to computer scientists because if it can be proven that an NPC problem can be reduced to being P then that would lead to conclusion that every NP problem are also P. Even though the question "Does P equals NP?" has not been given a definite answer, the majority of computer scientists believe that P does not equal NP.

A few other sources might say that the longest distance problem is an NP-hard problem. [3] A problem is NP-hard if an algorithm for solving it can be translated into one for solving any

NP-problem (nondeterministic polynomial time) problem. NP-Complete is a subset of NP-Hard.

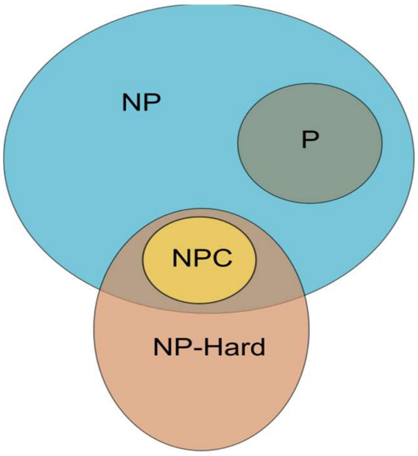


Figure 1.2 N, NP, NPC, and NP-Hard as Venn Diagram

III. Algorithms for The Longest Path Problem

The Longest path problem in general is a NP-Complete Problem because there are no found algorithm to solve the problem in a Hamiltonian graph within polynomial time. However, an exception is made for directed acyclic graph. The longest path in a directed acyclic graph can be solved in linear time.

In below explanation, every algorithm is used only for directed acyclic graph. Topological sort can be used to find the longest path. With a few modifications, Dijkstra's Algorithm can be used to find the longest path in a tree or directed acyclic graphs.

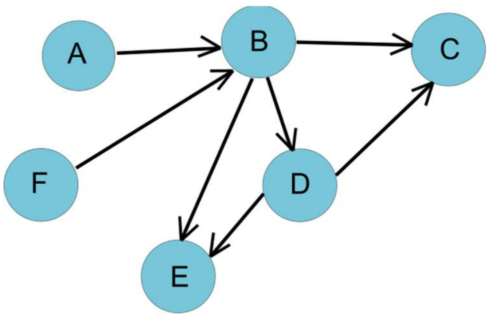


Figure 1.3 Example of a graph

A. Topological Sort

Below are the more detailed steps of finding the solution of a graph with n vertices using topological sort:

1. create an array of distance of size n and initialize the elements with negative infinity, except for the vertex s (source vertex) distance[s] = 0
2. Use topological sort to sort the vertices of the graph
3. For every vertex v in graph, and for every adjacent vertex u of vertex v, check if distance[v] is smaller than distance[u] + weight(v,u), if true, then initialize distance[v] with the value of distance[u] + weight(v,u)

Figure 1.3 can be used as an example. Figure 1.4 shows the topological graph sorted from Figure 1.3

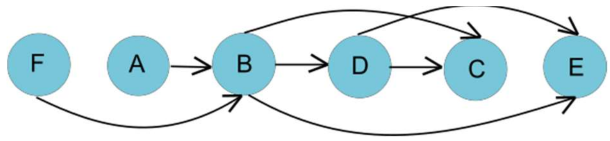


Figure 1.3 Topological Graph of the previous figure

For this example, lets say we store the vertices of the topological graph in a stack. While the stack is not empty, check for every array of distance, if it is not NINF (Negative Infinity), then check every other vertices v adjacent to it to see if distance[v] is smaller than distance[u] + weight(v,u).

At the start, because we initialized all distance except for s as NINF, we will check every adjacent vertices of s, to find that all the distance are smaller than 0. This would mean that the distance[v] will be replaced by the weight of (v,u) + NINF.

The process continues and the result of the longest distance in a graph will be stored in the array of distance.

Below is a C++ source code to calculate the longest distance:

```
   void topologicalSortUtil(int v, bool
visited[],

stack<int>& Stack)
   {
       // Mark the current node as visited
       visited[v] = true;

       // Recur for all the vertices
adjacent to this vertex
       list<AdjListNode>::iterator i;
       for (i = adj[v].begin(); i !=
adj[v].end(); ++i) {
           AdjListNode node = *i;
           if (!visited[node.getV()])

topologicalSortUtil(node.getV(), visited,
Stack);
       }

       // Push current vertex to stack
which stores topological
       // sort
       Stack.push(v);
```

```
}
   void Graph::longestPath(int s)
   {
        stack<int> Stack;
        int dist[V];

        // Mark all the vertices as not visited
        bool* visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = false;

        // Call the recursive helper function to store Topological
        // Sort starting from all vertices one by one
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, Stack);

        // Initialize distances to all vertices as infinite and
        // distance to source as 0
        for (int i = 0; i < V; i++)
            dist[i] = NINF;
        dist[s] = 0;

        // Process vertices in topological order
        while (Stack.empty() == false) {
            // Get the next vertex from topological order
            int u = Stack.top();
            Stack.pop();

            // Update distances of all adjacent vertices
            list<AdjListNode>::iterator i;
            if (dist[u] != NINF) {
                for (i = adj[u].begin(); i != adj[u].end(); ++i)
                    if (dist[i->getV()] < dist[u] + i->getWeight())
                        dist[i->getV()] = dist[u] + i->getWeight();
            }
        }

        // Print the calculated longest distances
        for (int i = 0; i < V; i++)
            (dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";
}
```

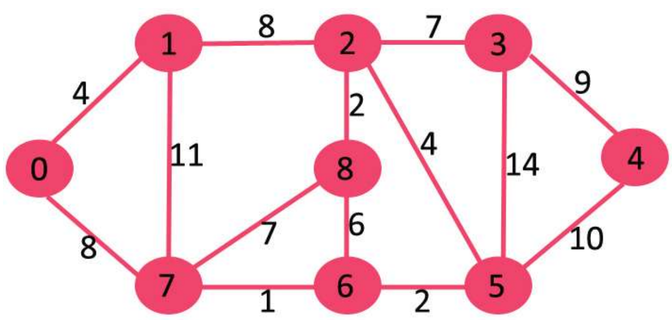
B. *Dijkstra's Algorithm*

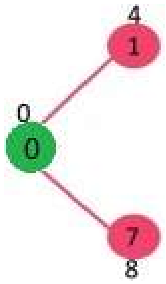
The Dijkstra's Algorithm is made by Edsger Dijkstra, who published a highly detailed description of the development of a depth-first backtracking algorithm. In order to use Dijkstra's algorithm to find the longest path of a directed acyclic graph, it is needed to negate the length c_i into $-c_i$. The rest of the algorithm remains the same, with finding the shortest path of the modified graph as the main goal in the Dijkstra's algorithm.

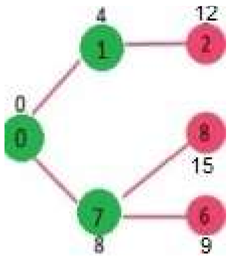
The steps for a Dijkstra Algorithm are as follows:

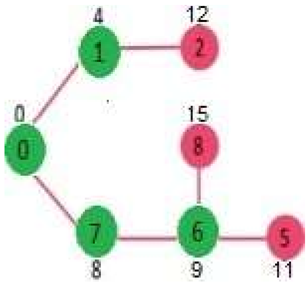
1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node C.
3. For each neighbour N of your current node C: add the current distance of C with the weight of the edge connecting C-N. If it's smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. If there are non-visited nodes, go to step 2.

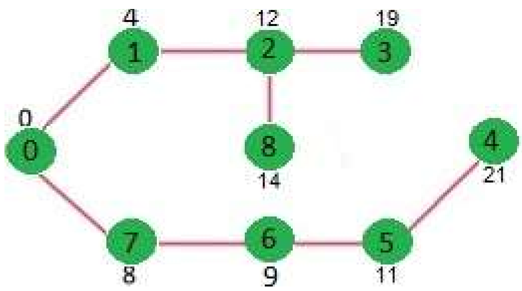
Here are step by step illustration for the Djikstra Algorithm for the shortest route:











Below is a source code for Djikstra Algorithm using Python programming language:

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph

# Library for INT_MAX
import sys

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printSolution(self, dist):
        print "Vertex tDistance from Source"
        for node in range(self.V):
            print node,"t",dist[node]

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initilaize minimum distance for next node
        min = sys.maxint

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    # Funtion that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
```

```
    def dijkstra(self, src):

        dist = [sys.maxint] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shotest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shotest path tree
            for v in range(self.V):
                if self.graph[u][v] > 0 and sptSet[v] == False and
                   dist[v] > dist[u] + self.graph[u][v]:
                        dist[v] = dist[u] + self.graph[u][v]

        self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]
          ];

g.dijkstra(0);

# This code is contributed by Divyanshu Mehta
```

IV. Conclusion

The longest path problem is the problem of finding a path which will result in the maximum length out of all paths possible in a given graph.

Its useful applications include giving the critical path of a graph which can be used for planning of complex projects and Static Timing Analysis (STA) in electronical design automation.

Unlike that of Shortest Path Problem, the longest path problem is an NP-Complete problem, except for cases where the given graphs are of directed acyclic graphs which has linear time solution.

There are algorithms to solve directed acyclic graphs. Using Topological sorting or Modified Djikstra's Algorithm, the longest distance of a graph can be found. Both algorithm can be used to find the shortest route for graph, but also can be used to found the longest route.

Acknowledgment

Author would like to express her deepest appreciation to all those who provided me the possibility to complete this report. Thank God, for His blessings, for it is His grace that made this paper can be finished in time. A thanks to Dr. Ir. Rinaldi Munir, MT. for his teachings, his loving support for all the students, his informative website, and for his ever-glowing spirit he shows everyone in his works.

References

[1] https://www.projectmanager.com/blog/understanding-critical-path-project-management

[2] https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_p_np_class.htm

[3] http://mathworld.wolfram.com/NP-HardProblem.html

[4] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2017

13517030
Eginata Kasan