A Heuristic Algorithm for Longest Simple Cycle Problem

Parveen Kumar
National Institute of Technology Hamirpur
Himachal Pradesh, India
parveen.csed@gmail.com

Nitin Gupta
National Institute of Technology Hamirpur
Himachal Pradesh, India
nitin3041@gmail.com

Abstract: **The longest simple cycle problem is the problem of finding a cycle of maximum length in a graph. As a generalization of the Hamiltonian cycle problem, it is NP-complete on general graphs and, in fact, on every class of graphs that the Hamiltonian cycle problem is NP-complete. The longest simple cycle problem may be solved in polynomial time on the complements of comparability graphs. It may also be solved in polynomial time on any class of graphs with bounded tree width or bounded clique-width, such as the distance-hereditary graphs. However, it is NP-hard even when restricted to split graphs, circle graphs, or planar graphs. In this paper a heuristic algorithm is proposed which can solve the problem in polynomial time. To solve the longest simple cycle problem using adjacency matrix and adjacency list by making a tree of given problem to find the longest simple cycle as the deepest path in tree following reconnect the leaf node of deepest path with root node. The result resolves the open question for the complexity of the problem on simple unweighted graphs. The algorithm is implemented on a simple labeled graph without parallel edges and without self-loop. The worst case time complexity for the proposed algorithm is O(V+E).**

Index Terms**: Adjacency List, Adjacency Matrix, Graph, Tree, NP-Completeness**

I. INTRODUCTION

The area of approximation algorithms for NP-hard optimization problems has received much attention over the past two decades [1], [2]. Although some notable positive results have been obtained, such as the fully polynomial approximation scheme for bin packing [3][4], it has now become apparent that even the approximate solution of a large class of NP-hard optimization problems remains outside the bounds of feasibility. For example, a sequence of results [5][6][7][8][9] established the intractability of approximating the largest clique in a graph. The optimization version of this problem is NP-hard since it includes the Hamiltonian path problem as a special case. Therefore, it is natural to look for polynomial-time algorithms with a small performance ratio, where the performance ratio is defined as the ratio of the longest path in the input graph to the length of the path produced by the algorithm. Our results attempt to pin down the best possible performance ratio achievable by polynomial-time approximation algorithms for longest paths (with same start and end vertices). In this paper, we address the problem of finding the longest Simple Cycle in an undirected graph G=(V, E). V is the set of n vertices and E is the set of m edges. The goal is to find for all u, v ϵ V, the longest path from u to u, using maximum number of edges. We consider simple paths, which do not have any repeated edges or vertices. This problem belongs to the NP-Complete class of problems, as it is a generalization of the Hamiltonian path problem and cannot be solved in polynomial time unless P=NP. For this reason, the algorithm proposed is approximation algorithm. The main objective of this work is to apply various approaches combined to solve this problem (i.e. adjacency list, adjacency Matrix, Tree). In Section 2, various proposed algorithms are discussed and work which is already done on this topic is also discussed. In Section 3, describe the proposed algorithm and its implementation steps. In Section 4, experimental results, and finally the conclusion.

II. PRELIMINARY AND BACKGROUND WORK:

The longest Simple Cycle problem, i.e. the problem of finding a simple path (with same start and end vertices) with the maximum number of vertices, is one of the most important problems in graph theory. The well-known NP-complete Hamiltonian path problem [10,11], i.e. deciding whether there is a simple path that visits each vertex of the graph exactly once, is a special case of the longest path problem. Only a few polynomial-time algorithms are known for the longest path problem for special classes of graphs. Trees are the first class of graphs that a polynomial-time algorithm for finding the diameter of an unweighted tree. Originally, this algorithm was proposed by Dijkstra around 1960 but Bulterman et al. [12] provided a proof for it, and later it was improved by Uehara and Uno [13] for the case of weighted trees. They also solved the problem for block graphs in linear time and for cacti in quadratic time. Recently, Ioannidou et al. [14] showed that the problem is polynomial for general interval graphs. Their algorithm is based on dynamic programming and runs in O (n4) time. More recently, Mertzios and Corneil [15] solved the problem in polynomial time for the larger class of graphs i.e. cocomparability graphs. Also, there is an O (n3)-time algorithm for the problem for complete m-partite digraphs proposed by Gutin [16]. Also, it has been shown that finding a path of length $n - n\epsilon$ is not

possible in polynomial time unless P = NP [17]. To our knowledge, the best-known approximation algorithm for the problem has the ratio of O (n (log log n/ log n)2) [17]. For more related results on approximation algorithms on general graphs see [18][19][20][21]. Monien [7] presented an 0 (k! n m)-time algorithm that finds paths of length k in a Hamiltonian graph with n vertices and m edges. since in polynomial time Monien's algorithm can only find paths of length O(log n/log log n). Furer and Raghavachari [22] present approximation algorithms for the minimum-degree spanning tree problem that delivered absolute performance guarantees (within an additive factor of 1). No hardness results for longest paths were known earlier, although a seemingly related problem has been studied by Berman and Schnitger [23]. They show that the hardness conjecture is true for the problem of approximating the longest induced path in an undirected graph. Note that the induced-path problem is strictly harder and their hardness result does not carry over to the problem under consideration here. Bellare [24] considers a generalization of the longest-paths problem called the longest color-respecting path problem. This involves graphs with 2-colored edges and labeled vertices, and a feasible path must have the property that at each vertex its label specifies whether the incident edges of the path are of the same color or not. He obtains essentially the same hardness results through different techniques.

III. REPRESENTATIONS OF GRAPHS

There are two standard ways to represent a graph G = (V, E): as a collection of adjacency lists or as an adjacency matrix. Either way applies to both directed and undirected graphs. Because the adjacency-list representation provides a compact way to represent sparse graphs—those for which |E| is much less than |V |2 — it is usually the method of choice. Most of the graph algorithms assume that an input graph is represented in adjacency list form. We may prefer an adjacency-matrix representation, however, when the graph is dense—|E| is close to |V|2—or when we need to be able to tell quickly if there is an edge connecting two given vertices.

The adjacency-list representation of a graph G = (V, E): consists of an array Adj of |V| lists, one for each vertex in V. For each u ∈ V, the adjacency list Adj[u] contains all the vertices v such that there is an edge (u,v) ϵ E. That is, Adj[u] consists of all the vertices adjacent to u in G. (Alternatively, it may contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph,. If G is an undirected graph, the sum of the lengths of all the adjacency lists is 2|E|, since if (u,v) is an undirected edge, then u appears in v's adjacency list and vice versa. For undirected graphs, the adjacency-list representation has the desirable property that the amount of memory it requires is ɵ(V+E). It can readily adapt adjacency lists to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function w: E→R. For example, let G=(V,E) be a weighted graph with weight function w. The weight w(u,v) of the edge (u,v) ϵ E simply store with vertex v in u's adjacency list. The adjacency-list representation is quite robust in a way that it can be modified to support many other graph variants. A potential disadvantage of the adjacency-list representation is that it provides no quicker way to determine whether a given edge (u,v) is present in the graph than to search for v in the adjacency list Adj[u]. An adjacency-matrix representation of the graph remedies this disadvantage, but at the cost of using asymptotically more memory.

For the adjacency-matrix representation of a graph G =(V,E), it is assumed that the vertices are numbered 1,2, ……,|V| in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a |V|*|V| matrix A=(aij) such that

$$a_{ij} \begin{cases} 1 & \text{if}(i,j) \in E, \\ 0 & \text{otherwise} \end{cases}$$

Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if G = (V, E) is a weighted graph with edge weight function w, it can simply store the weight w(u,v) of the edge (u,v) ∈ E as the entry in row u and column v of the adjacency matrix. If an edge does not exist, it can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞. Although the adjacency-list representation is asymptotically at least as space efficient as the adjacency-matrix representation, adjacency matrices are simpler, and so they may preferred when graphs are reasonably small. Moreover, adjacency matrices carry a further advantage for unweighted graphs: they require only one bit per entry.

A. Complexity Classes

Mainly three classes of problems are referred: P, NP, and NPC, the latter class being the NP-complete problems. Which are described formally as:

Class P

The class P consists of those problems that are solvable in polynomial time. More specifically, they are problems that can be solved in time O(nk) for some constant k, where n is the size of the input to the problem. A lot of the problems considered in P. Any problem in P is also in NP, since if a problem is in P then it can be solved in polynomial time without even being supplied a certificate. For now it is considered P ⊂ NP. The open question is whether or not P is a proper subset of NP.

The Complexity Class NP

The complexity class NP is the class of languages that can be verified by a polynomial- time algorithm. More precisely, a language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and a constant c such that L = {

x ∈ {0,1} there exists a certificate y with |y| = O(|x|) such that A(x, y) = 1}.
It can be said that algorithm A verifies language L in polynomial time. It is unknown whether P = NP, but most researchers believe that P and NP are not the same class. Intuitively, the class P consists of problems that can be solved quickly. The class NP consists of problems for which a solution can be verified quickly. it is often more difficult to solve a problem from scratch than to verify a clearly presented solution.

Complexity Class NP complete

NP-complete problems arise in diverse domains: boolean logic, graphs, arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, biology, chemistry, physics, and more. In this section, the reduction methodology used to provide NP completeness proofs for a variety of problems drawn from graph theory and set partitioning.

B. The Longest Simple Cycle Algorithm

In the Proposed Algorithm, the input graph considered to be a simple graph (i.e. without self-loop and without parallel edges), the algorithm for Longest Simple Cycle in simple graph is summarized below:

1. Enumerate all the nodes to calculate degree of each node to find the node with highest degree.
2. Assign the node with highest degree as the root for tree.
3. Construct a tree T of the given graph G considering the adjacent nodes as successor and predecessors accordingly for each vertex using adjacency matrix.
4. Do apply the proposed LSC algorithm to find the longest path.
5. Join the leaf node of the longest path with root and retrieve the path considering it as the longest cycle in graph.

From now on, we describe each step of the algorithm in more detail.

1. *Enumerate all the nodes to calculate degree of each node to find the node with highest degree.*

For a given Graph G=(V,E) where V is the set vertices n and E is the set of edges e, first to make an adjacency matrix for all vertices to find the maximum degree vertex, which would become the root of tree.

2. *Assign the node with highest degree as the root for tree.*

Now assign the MAX (which was returned by pseudo code) as root.
ROOT ← MAX

3. *Construct a tree of the given graph considering the adjacent nodes as successor and predecessors accordingly for each vertex.*

The vertex with maximum degree is taken as root and the adjacent vertices are considered as predecessor of root and taken as child of root vertex, now for child vertices their adjacent vertices are taken as predecessor of child vertex and so on. This process of converting graph into tree will go on till all vertices and their adjacent vertices are expended.
Figure 1 shown below the tree constructed from graph G.

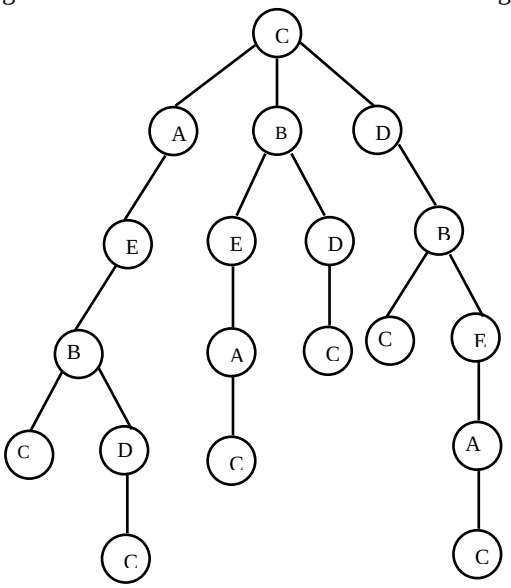

Figure 1. tree constructed by using adjacency matrix to represent and expand adjacent vertex for all vertices

4. *Do apply the proposed LSC algorithm to find the longest path.*

The Proposed algorithm for Longest Simple Cycle Problem is proposed as:

LSC(G)

```
1.  for each vertex u ∈ G.[V- ROOT]
2.     {
3.      Color[u] ← white
4.      Pred[u] ← NIL
5.     }
6.  count = 0
7.  for each vertex u ∈ G.adj[ROOT]
8.     {
9.      if color[u] = white
10.       then LSC_TRAV (u)
11.     endif
12.    }
```

LSC_TRAV (u)

```
1.  Color[u] ← pink   // vertex u has just discovered
2.  Count ← count + 1
3.  Discover[u] ← count
4.  for each v ∈ G.adj(u)            //Explore edge (u,v)
5.     {
6.      if (v = ROOT) then
7.         Finish[u] ← count + 1
8.      else
9.       if (adj(u) = NIL) then
```

```
10.          Finish[u] ← count
11.        else
12.         if color[v] = white then
13.            Pred[v] ← u
14.            LSC_TRAV (G.v)
15.        }
16.  Color[u] ← RED          //make u RED ; it is finished
17.  count ← count + 1
18.  finish[u] ← count
```

Procedure LSC works as follows. Lines 1-4 paint all vertices white and initialize their PRES field to NIL. Line 5 resets the global counter. Line 7-12 check each vertex in V in turn and, when a white vertex is found, visit it using LSC_TRAV. Every Time LSC_TRAV (u) is called in line 10, vertex u becomes the root of a new tree T'. When LSC returns, every vertex u has been assigned a discovery time **Discover[u]** and a finish time **finish[u].**

In each call LSC_TRAV (u), vertex u is initially white. Line 1 paints u pink, line 2 increments the global variable count and line 3 records the new value of time as the discovery time discover[u]. Lines 4-15 examine each vertex v if it is white. As each vertex $v \in adj[u]$ is considered in line 4 we say that each is explored by this step. Finally after each edge leaving u has been explored, line 15 paint u pink and record the finishing time finish[u]. Note that result of Longest simple Cycle Algorithm depend upon the order in which the vertices are examined in line 7 of LSC, and upon the order in which the neighbors of a vertex are visited in line 6,9 and 12 of LSC_TRAV. These different visitation orders tend not to cause problem in execution as any order of exploring result can usually be used effectively, with essentially equivalent results.

The running time of LSC is computed as follows:

The loop on lines 1-5 and lines 7-12 of LSC takes time O(V), exclusive of the time to execute the call LSC_TRAV. As the procedure LSC_TRAV is called exactly once for each vertex $v \in V$, since LSC_TRAV invoked only white vertices and the first thing it does it paint vertices pink. During an execution of LSC_TRAV (v), the loop on line 4-15 is executed |Adj[v]| times. Since

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

The total cost of executing lines 4-15 of LSC_TRAV is $\Theta(E)$. The running time of LSC is therefore $\Theta(V + E)$

5. *Join the leaf node of the longest path with root and retrieve the path considering it as the longest cycle in graph.*

The pseudo code to retrieve longest path from the above tree is:

```
    PRINT_CYCLE (G, ROOT, V)
1.  if (v == ROOT)
2.      then print "ROOT"
3.      else
4.       if (V.PRED = NIL)
5.          then print "no cycle exist"
6.          else
7.         PRINT_CYCLE(G, ROOT, V.PRED)
8.         print V
```

Procedure PRINT_CYCLE runs in time Linear in the number of vertices in the Cycle printed. Since each recursive call is for a path one vertex shorter.

The given graph after connecting deepest leaf with root is shown in figure 2:

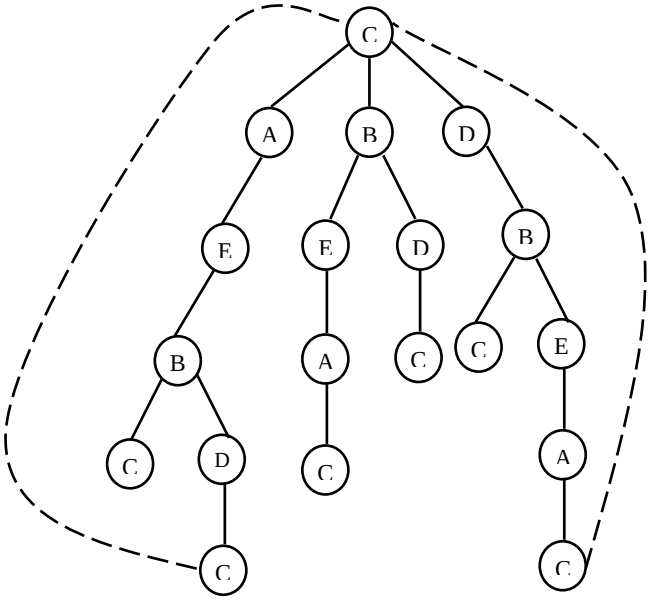

Figure 2. Connecting deepest vertex with root vertex to make a cycle

IV EXPERIMENTAL RESULTS:

For experimental research and proof of algorithm the problem graph is considered is shown in figure 3:

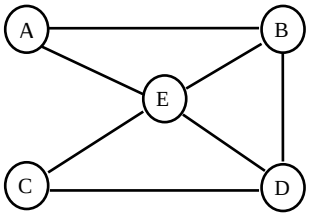

Figure 3. graph G = (V, E) considered to find Longest simple cycle

Now, after applying the proposed Longest Simple Cycle Algorithm the diagrammatic procedure is shown below in figures:

1. *The adjacency matrix for the given problem graph is shown in figure 4:*

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	0	1
C	1	1	0	1	1
D	0	0	1	0	1
E	0	1	1	1	0

Figure 4. Adjacency matrix for Graph G

2. Assianina the node with hiahest dearee as the root for tree.

$D_1 = e_{11} + e_{12} + e_{13} + e_{14} + e_{15}$ 0 + 1 + 0 + 0 + 1 = 2
$D_2 = e_{21} + e_{22} + e_{23} + e_{24} + e_{25}$ 1 + 0 + 1 + 0 + 1 = 3
$D_3 = e_{31} + e_{32} + e_{33} + e_{34} + e_{35}$ 1 + 1 + 0 + 1 + 1 = 4
$D_4 = e_{41} + e_{42} + e_{43} + e_{44} + e_{45}$ 0 + 0 + 1 + 0 + 1 = 2
$D_5 = e_{51} + e_{52} + e_{53} + e_{54} + e_{55}$ 0 + 1 + 1 + 1 + 0 = 3

From the above adjacency matrix it is calculated that D3 has highest degree i.e. Vertex C. Now assign vertex C as Root Node for the tree.

ROOT ← C

3. Construct a tree of the given graph considering the adjacent nodes as successor and predecessors accordingly for each vertex.

After assigning vertex C as ROOT node tree of adjacent vertices is shown in figure 5.

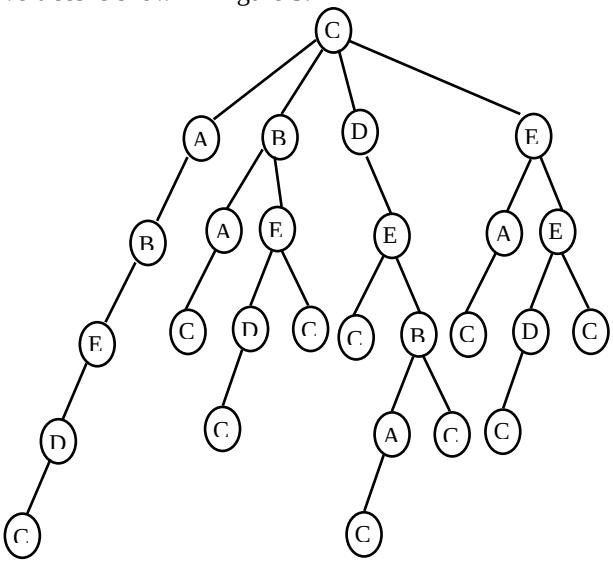

Figure 5. Tree constructed from adjacency matrix for Graph G

4. Do apply the proposed LSC algorithm to find the longest cycle.

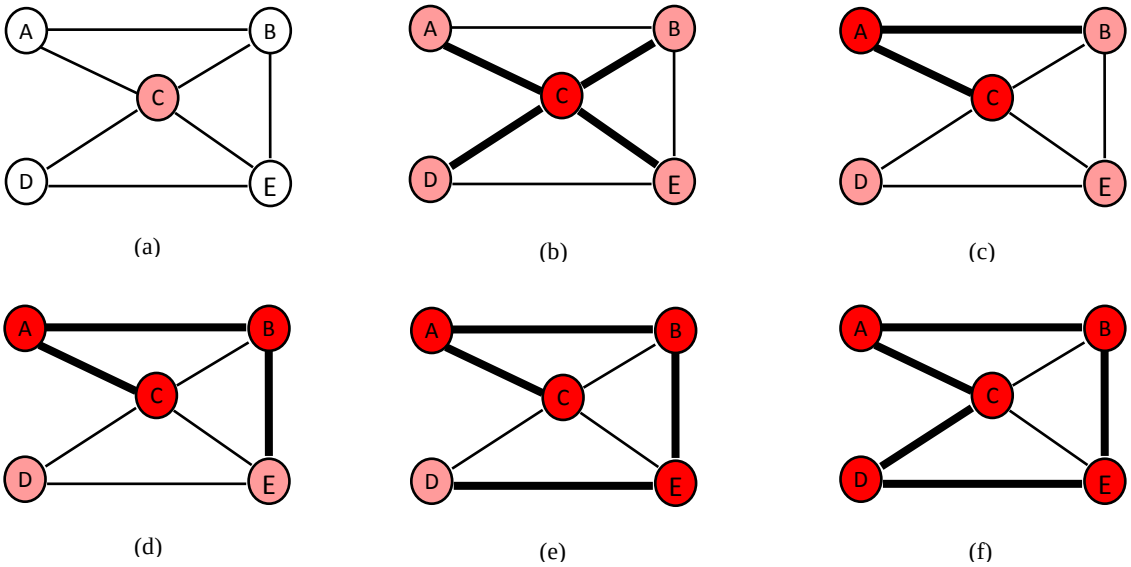

Figure 6. Implementing Steps of Longest Simple Cycle Problem

5. Join the leaf node of the longest path with root and retrieve the path considering it as the longest cycle in graph.

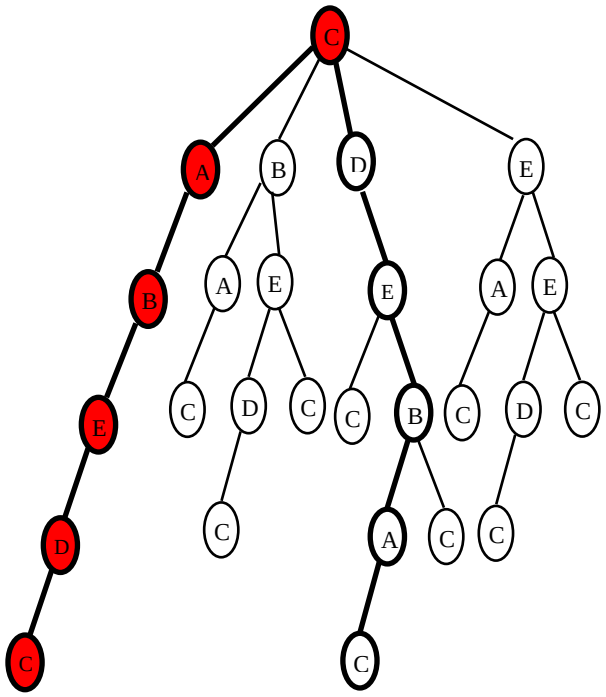

Figure 7. Tree showing the deepest node as the farthest vertex

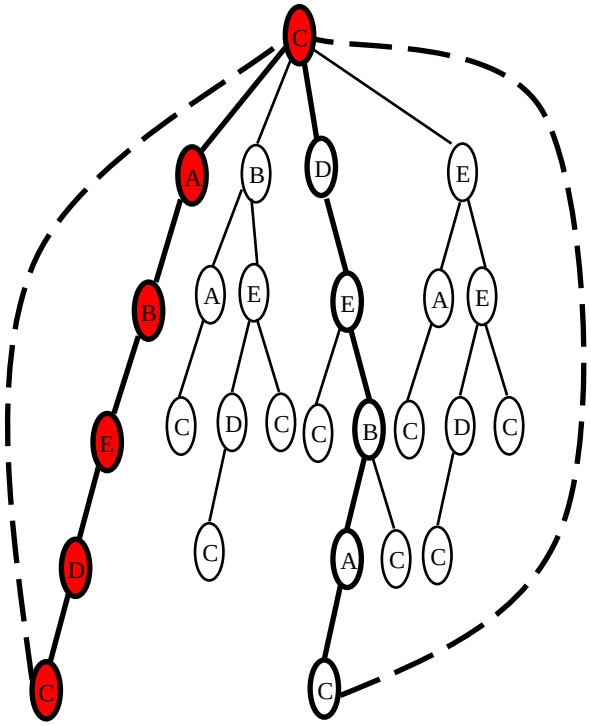

Figure 8. Deepest vertex connected with ROOT vertex to make a cycle

After joining the deepest node with the root the retrieved longest paths are C-A-B-E-D-C and C-D-E-B-A-C.
Now technically both cycles are same but retrieved in reverse directions, So only one cycle is considered to be as longest.

Complexity Computation:
Step 1: compute adjacency Matrix = $|V|^2$
Step 2: compute degree of each vertax = $|V|^2$
Step 3: Retrieve Node with maximum Degree = V
Step 4: Complexity of LSC Algorithm = O(V+E)

Total complexity for the experiment is:
$|V|^2 + |V|^2 + V + (V+E) \rightarrow V^2$

The above computed complexity is for the total result from computing adjacency matrix to finally retrieval of longest path. But the actual complexity of proposed algorithm i.e. Longest Simple Cycle Algorithm is O(V+E).which is quite less than V2.

CONCLUSION:

In this work we help to shed some light on the borderline between P and NP, since the longest simple cycle problem is known to be NP-complete on graphs. It would be interesting to study the complexity of the longest simple cycle problem on distance hereditary and bipartite distance-hereditary graphs, since they admit polynomial solutions for the Hamiltonian path problem, and also since the longest simple cycle problem has been proved to be NP-complete on chordal bipartite graphs, and parity graphs, while it is polynomial on ptolemaic graphs and trees. In this Paper we presented an approximation algorithm for solving the longest simple cycle problem on simple graphs, which find the maximum length cycle in a connected graph (if exist) with average case complexity O(V+E). Various techniques are used to implement the strategy such as adjacency matrix, adjacency List and simple rooted tree. Experimental results are shown in section 4.The work constitute a significant achievement on NP-complete problems to solve it in approximate time.

ACKNOWLEDGMENTS:

The authors would like to thank the referees and reviewers for a very careful reading and many helpful comments and suggestions, which have led to great improvement of the paper.

REFERENCE:

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, CA, 1979.
[2] R. Motwani, Lecture Notes on Approximation Algorithms, Technical Report No. STAN-CS-92-1435, Department of Computer Science, Stanford University, 1992.
[3] W. E de la Vega and G. S. Lueker, Bin packing can be solved within 1 + e in linear time, *Combinatorica*, 1 (1981), 349-355.
[4] N. Karmakar and R. M. Karp, An efficient approximation scheme for the one-dimensional bin packing problem, *Proceedings of the 23rd Annual lEEE Symposium on Foundations of Computer Science*, 1982, pp_ 312-320.
[5] U. Feige, S. Goldwasser, L. Lovfisz, S. Safra, and M. Szegedy, Approximating clique is almost NPcomplete, *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, 1991, pp. 2-12.
[6] A. Blum, Some tools for approximate 3-coloring, *Proceedings of the 31 st Annual IEEE Symposium on Foundations of Computer Science*, 1990, pp. 554-562.161
[7] B. Monien, How to find long paths efficiently, *Annals of Discrete Mathematics*, 25 (1984), 239-254.
[8] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy, Proof verification and hardness of approximation problems, *Proceedings of the 33rd Annual llz'ls Symposium on Foundations of Computer Science*, 1992, pp. 14-23.
[9] V. Chvatal, Tough graphs and Hamiltonian circuits, *Discrete Mathematics*, 5 (1973). 215-228.
[10] R. Diestel, Graph Theory, Springer, New York, 2000.
[11] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, San Francisco, 1979.
[12] R.W. Bulterman, F.W. van der Sommen, G. Zwaan, T. Verhoeff, A.J.M. van Gasteren, W.H.J. Feijen, On computing a longest path in a tree, Inform. Process. Lett. 81 (2) (2002) 93–96.
[13] R. Uehara, Y. Uno, On computing longest paths in small graph classes, Internat. J. Found. Comput. Sci. 18 (5) (2007) 911–930.
[14] K. Ioannidou, G.B. Mertzios, S. Nikolopoulos, The longest path problem is polynomial on interval graphs, in: Proc. Of 34th Int. Symp. on Mathematical Foundations of Computer Science, vol. 5734, Springer-Verlag, Novy Smokovec, High Tatras, Slovakia, 2009, pp. 403–414.

[15] G.B. Mertzios, D.G. Corneil, A simple polynomial algorithm for the longest path problem on cocomparability graphs, in: Proc. of CoRR, 2010.
[16] G. Gutin, Finding a longest path in a complete multipartite digraph, SIAM J. Discrete Math. 6 (2) (1993) 270–273.
[17] D. Karger, R. Montwani, G.D.S. Ramkumar, On approximating the longest path in a graph, Algorithmica 18 (1) (1997) 82–98.
[18] A. Björklund, T. Husfeldt, Finding a path of superlogarithmic length, SIAM J. Comput. 32 (6) (2003) 1395–1402.
[19] T. Feder, R. Motwani, Finding large cycles in Hamiltonian graphs, in: Proc. 16th Annual ACM–SIAM Symp. on Discrete Algorithms, SODA, ACM, 2005, pp. 166–175.
[20] H.N. Gabow, Finding paths and cycles of superpolylogarithmic length, in: Proc. 36th Annual ACM Symp. on Theory of Computing, STOC, ACM, 2004, pp. 407–416.
[21] Z. Zhang, H. Li, Algorithms for long paths in graphs, Theoretical Computer Science 377 (1–3) (2007) 25–34
[22] F. Luccio, C. Mugnia, Hamiltonian paths on a rectangular chessboard, in: Proc. 16th Annual Allerton Conference, 1978, pp. 161–173.
[23] A. Itai, C. Papadimitriou, J. Szwarcfiter, Hamiltonian paths in grid graphs, SIAM J. Comput. 11 (4) (1982) 676–686.
[24] S.D. Chen, H. Shen, R. Topor, An efficient algorithm for constructing Hamiltonian paths in meshes, J. Parallel Comput. 28 (9) (2002) 1293–1305.
[25] Yancai Zhaoa, Liying Kang , Moo Young Sohn " The algorithmic complexity of mixed domination in graphs" *Theoretical Computer Science 412 (2011) 2387–2392.*
[26] Christian Glaßer , Christian Reitwießnera , Victor Selivanov "The shrinking property for NP and coNP" *Theoretical Computer Science 412 (2011) 853–864.*
[27] Ferdinando Cicalese , Tobias Jacobs , Eduardo Laber , Marco Molinaro "On the complexity of searching in trees and partially ordered structures" *Theoretical Computer Science 412 (2011) 6879–6896.*
[28] James K. Lana, Gerard Jennhwa Chang "On the mixed domination problem in graphs" *Theoretical Computer Science 476 (2013) 84–93.*
[29] Van Bang Le, Ragnar Nevries "Complexity and algorithms for recognizing polar and monopolar graphs"*Theoretical Computer Science 528 (2014) 1–11.*
[30] Lei Chena , Weiming Zeng , Changhong Lub "NP-completeness and APX-completeness of restrained domination in graphs" *Theoretical Computer Science 448 (2012) 1–8.*
[31] Kirishima T, Shiono Y, Sugimoto F, Kato C, Yaku T Optimality and Complexity for Drawing Problems of Tree-Structured Diagrams *14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2013, pp. 484 – 489, Honolulu, HI*