TWMS J. App. and Eng. Math. V.15, N.9, 2025, pp. 2398-2405

A COMPARATIVE STUDY ON PRIME AND VERTEX k-PRIME LABELING OF ONE POINT UNION OF PATH GRAPHS

A. TERESA AROCKIAMARY S[1], B. VIJAYALAKSHMI P[2*], §

Abstract. In our study, we investigate the structure of one point union of path graphs and prove that they admit a vertex k-prime labeling. Further a study on comparison between prime and vertex k-prime labeling for one point union of path graph has been analysed and applied in the field of computer science with the help of C^{++} programming language.

Keywords: labeling, prime labeling, k-prime labeling, vertex k-prime labeling, one point union of path graphs.

AMS Subject Classification: 05C78, 05C90, 68R10.

1. Introduction

One of the rapidly growing fields of research in mathematics is graph theory and it is widely recognised as a crucial dynamic tool in a wide range of disciplines. A graph G consists of three components: a nonempty set of vertices $V(G)$ or points, a set of edges or lines $E(G)$, and a function that connects each edge to two vertices, referred to as its endpoints [2]. Graph labeling is one of the many fields of study within graph theory, having originated in the 1960s. Rosa [4] first introduced graceful labeling in 1967, and since then, many different graph labeling techniques have been developed. These techniques are growing into a more often used approach to mathematical modeling for a range of applications which includes communication networks, hierarchical structure, software testing, timetable scheduling, encrypting and decrypting numbers, missile guidance, cloud computing, signal processing, network analysis, image processing, computer vision and much more. An extensive analysis of a number of graph labeling applications is given by Bloom and Golomb [1].

Entringer introduced the idea of prime labeling, which Tout, Dabboucy and Howalla [8] originally proposed in a study. A prime labeling of a graph G is a one-one function $g : V(G) \to \{1, 2, 3, ..., |V(G)|\}$ such that for every pair of adjacent vertices u and v,

[1] University of Madras, Department of Mathematics, Stella Maris College, Chennai - 600086, India.
e-mail: teresa_aroc@yahoo.co.in; ORCID: https://orcid.org/0000-0001-5887-0875.
[2] Anna University, Department of Mathematics, CEG Campus, Chennai - 600025, India.
e-mail: viji.gsekar@gmail.com; ORCID: https://orcid.org/0000-0002-0730-0925.
* Corresponding author.
§ Manuscript received: June 12, 2024; accepted: October 26, 2024.
TWMS Journal of Applied and Engineering Mathematics, Vol.15, No.9

$\gcd(g(u), g(v)) = 1$. A graph G that admits prime labeling is called a prime graph [3]. The concept of vertex k-prime labeling was introduced by Teresa et al. [6]. The results on cyclic snake graphs and corona graphs of the form $mC_n \bigodot K_1$ [5], theta-related graphs and centralised generalised theta graph [7] were proved to be vertex k-prime.

Vertex k-prime labeling is defined as follows: A vertex k-prime labeling of a graph G is a one-one and onto function $g : V(G) \to \{k, k+1, k+2, ..., k+|V|-1\}$ for some positive integer k such that $\gcd(g(u), g(v)) = 1 \ \forall e = uv \in E(G)$. A graph G that admits vertex k-prime labeling is called a vertex k-prime graph [6].

This study has attempted to investigate the existence of vertex k-prime labeling of one point union of path graphs. In addition, the application of one point union of path graphs has been examined where prime labeling and vertex k-prime labeling are compared for time complexity which is applied in the field of computer science with the help of C^{++} programming language.

Definition 1.1. *One point union of path graph P_n^r is a tree of r paths and n vertices with exactly r vertices of degree one, one vertex of degree r and $r(n-1)$ vertices of degree two.*

2. Main Result

In our study, we assume that there is at least one prime number in $\{k, k+1, ..., k+|V|-1\}$.

Notation: The one point union of the path graph is represented by P_n^r, where n is the number of vertices in each path and r is the number of copies of P_n. The graph is constructed by considering v_0 as the central vertex and r copies of P_n with n vertices and $n-1$ edges. We indicate the vertices of y^{th} copy of P_n^r as v_x^y $(1 \le x \le n, 1 \le y \le r)$.

Theorem 2.1. *The one point union of path graph P_n^r is vertex k-prime for $k \ge 1, n, r \ge 2$.*

Proof. Let $G = P_n^r$ be the one point union of path graph of order $nr+1$ and size nr. Let
$V(P_n^r) = \{v_0\} \cup \{v_x^y : 1 \le x \le n, 1 \le y \le r\}$
$E(P_n^r) = \begin{cases} v_x^y v_{x+1}^y : & 1 \le x \le n-1, \quad 1 \le y \le r \\ v_0 v_1^y : & 1 \le y \le r \end{cases}$
See Figure 1(a). Let l be the largest prime number from k to $k+nr$. Define a one-one and onto function $g : V(P_n^r) \to \{k, k+1, ..., k+nr\}$ as follows.

Case 1. $n = 2^m, m \ge 1$ and k odd

Subcase (i). k is l

$g(v_0) = l$

$g(v_x^y) = l + (y-1)n + x, \quad 1 \le x \le n, 1 \le y \le r$

For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(l+(y-1)n+x, l+(y-1)n+x+1) = 1$ since $l+(y-1)n+x$ and $l+(y-1)n+x+1$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, l+(y-1)n+1) = 1$ since l is prime.

Subcase (ii). $k+nr$ is l

$g(v_0) = l$

$g(v_x^y) = k + (y-1)n + x - 1, \quad 1 \le x \le n, 1 \le y \le r$

For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.

Subcase (iii). $k+(r-1)n$ is l

$g(v_0) = l$

$g(v_x^y) = \begin{cases} k+(y-1)n+x-1 & : \quad 1 \le x \le n, 1 \le y \le r-1 \\ k+(y-1)n+x & : \quad 1 \le x \le n, y = r \end{cases}$

For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.
Subcase (iv). $k+nr$ is not prime and $k+(r-1)n \neq l$
$g(v_0) = l$
$g(v_x^y) = k+(y-1)n+x-1, \quad 1 \leq x \leq n, 1 \leq y \leq r-1$
Label the remaining vertices $v_1^r, v_2^r, ..., v_n^r$ with the integers $k+(r-1)n$ to $k+nr$ other than l from $\{l+1, l+2, ..., k+nr-1, k+nr, k+(r-1)n, k+(r-1)n+1..., l-2, l-1\}$ satisfying the condition that $gcd(g(v_x^r), g(v_{x+1}^r)) = 1$.
Case 2. n is an odd prime number and $k \not\equiv 0(mod\ n)$
Subcase (i). k is l
$g(v_0) = l$
$g(v_x^y) = l+(y-1)n+x, \quad 1 \leq x \leq n, 1 \leq y \leq r$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(l+(y-1)n+x, l+(y-1)n+x+1) = 1$ since $l+(y-1)n+x$ and $l+(y-1)n+x+1$ are successive positive integers. For each $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, l+(y-1)n+1) = 1$ since l is prime.
Subcase (ii). $k+nr$ is the largest prime l
$g(v_0) = k+nr$
$g(v_x^y) = k+(y-1)n+x-1, \quad 1 \leq x \leq n, 1 \leq y \leq r$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.
Subcase (iii). $k+(r-1)n$ is l
$g(v_0) = l$
$$g(v_x^y) = \begin{cases} k+(y-1)n+x-1 & : \quad 1 \leq x \leq n, 1 \leq y \leq r-1 \\ k+(y-1)n+x & : \quad 1 \leq x \leq n, y = r \end{cases}$$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), f(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.
Subcase (iv). $k+nr$ is not prime and $k+(r-1)n \neq l$
$g(v_0) = l$
$g(v_x^y) = k+(y-1)n+x-1, \quad 1 \leq x \leq n, 1 \leq y \leq r-1$
Label the remaining vertices $v_1^r, v_2^r, ..., v_n^r$ with the integers $k+(r-1)n$ to $k+nr$ other than l from $\{l+1, l+2, ..., k+nr-1, k+nr, k+(r-1)n, k+(r-1)n+1..., l-2, l-1\}$ satisfying the condition that $gcd(g(v_x^r), g(v_{x+1}^r)) = 1$.
Case 3. n is a composite number other than 2^n for $n \geq 1$ and k not a multiple of any factor of n
Subcase (i). k is l
$g(v_0) = l$
$g(v_x^y) = l+(y-1)n+x, \quad 1 \leq x \leq n, 1 \leq y \leq r$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(l+(y-1)n+x, l+(y-1)n+x+1) = 1$ since $l+(y-1)n+x$ and $l+(y-1)n+x+1$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, l+(y-1)n+1) = 1$ since l is prime.
Subcase (ii). $k+nr$ is l
$g(v_0) = l$
$g(v_x^y) = k+(y-1)n+x-1, \quad 1 \leq x \leq n, 1 \leq y \leq r$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), f(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.

As it will be observed from the Figure 1(b).
Subcase (iii). $k+(r-1)n = l$
$g(v_0) = l$
$g(v_x^y) = \begin{cases} k+(y-1)n+x-1 & : \quad 1 \le x \le n, 1 \le y \le r-1 \\ k+(y-1)n+x & : \quad 1 \le x \le n, y = r \end{cases}$
For each $v_x^y v_{x+1}^y \in E(P_n^r), gcd(g(v_x^y), g(v_{x+1}^y)) = gcd(k+(y-1)n+x-1, k+(y-1)n+x) = 1$ since $k+(y-1)n+x-1$ and $k+(y-1)n+x$ are successive positive integers. For the edge $v_0 v_1^y \in E(P_n^r), gcd(g(v_0), g(v_1^y)) = gcd(l, k+(y-1)n) = 1$ since l is prime.
Subcase (iv). $k+nr$ is not prime and $k+(r-1)n \neq l$
$g(v_0) = l$
$g(v_x^y) = k+(y-1)n+x-1, \quad 1 \le x \le n, 1 \le y \le r-1$
Label the remaining vertices $v_1^r, v_2^r, ..., v_n^r$ with the integers from $k+(r-1)n$ to $k+nr$ other than l from $\{l+1, l+2, ..., k+nr-1, k+nr, k+(r-1)n, k+(r-1)n+1..., l-2, l-1\}$ satisfying the condition that $gcd((g(v_x^r), g(v_{x+1}^r)) = 1$.
Thus P_n^r admits a vertex k-prime labeling. □

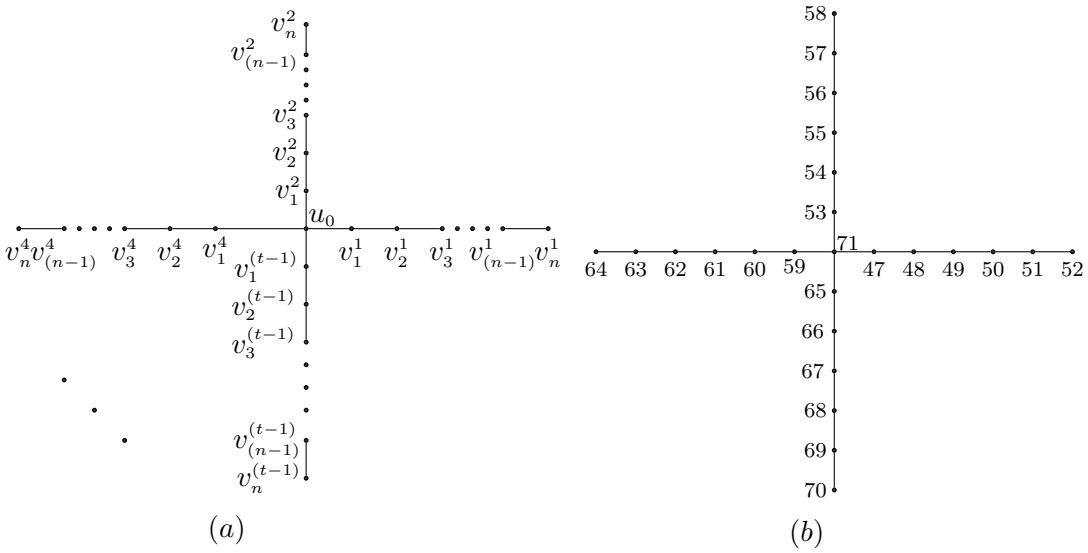

FIGURE 1. (a) P_n^r (b) Vertex k-prime labeling of P_6^4 for $k = 47$

Remark 2.1. *In case* 1 *of Theorem* 2.1*, we observe that vertex k-prime labeling does not admit for even k as k and $k+|V|-1$ are even.*

3. Application of one point union of path graph

The prime labeling and vertex k-prime labeling for one point union of the path graph has been constructed and the results are analysed. The application is carried out with the help of descendant search. A node that can be reached by repeatedly traversing from parent to child is called a descendant. For instance, the original mother and father are the ancestors of children, grand children, great-grandchildren and great-great-grandchildren. They are in the direct line of descent. The descendant in graph theory refers to any vertex that is either the child of a vertex v or the descendant of v's children is considered a descendant of v. The most used method for applying labeling techniques is the descendant search. The search is commonly used to locate a certain branch in the path with varying depths. We

examined the search using varying source and destination nodes. Using inputs at different depths, the research provided us with the time required for prime and vertex k-prime labeling to locate the target node.

This is an application generally carried out to find a specific branch in hierarchy. The tree starts from root and traverses to the leaf node that has no child nodes. This application helps us find the distance between two nodes connected at different levels. To understand this application, let us consider a social network such as Facebook or Instagram where we interact with our friends by sharing stories, photographs or videos. In such a network, when we receive a suggestion for a friend as proposed by the social network, it will revolve around our close common friends. This application uses the descendant search algorithm to find a person close to our interest.

Descendant search is used in this study because it is the primary experiment to analyse how the labeling scheme under discussion can change the efficiency of traversing a tree or in finding a specific branch in a tree. This labeling scheme determines the relationship between two nodes by comparing their labels. In the graph, the adjacent labeled vertices are considered and all nodes whose greatest common divisor is 1 are descendants of the node.

Source and destination nodes are specific and modified

When we first designed this application, we thought it would be useful to understand the extent of impact of vertex k-prime and prime labeling with specific source and destination will bring, to descendant search. We conducted experiments by using different sources and destinations based on how the tree had branched out. An illustration is given in Figure 2.

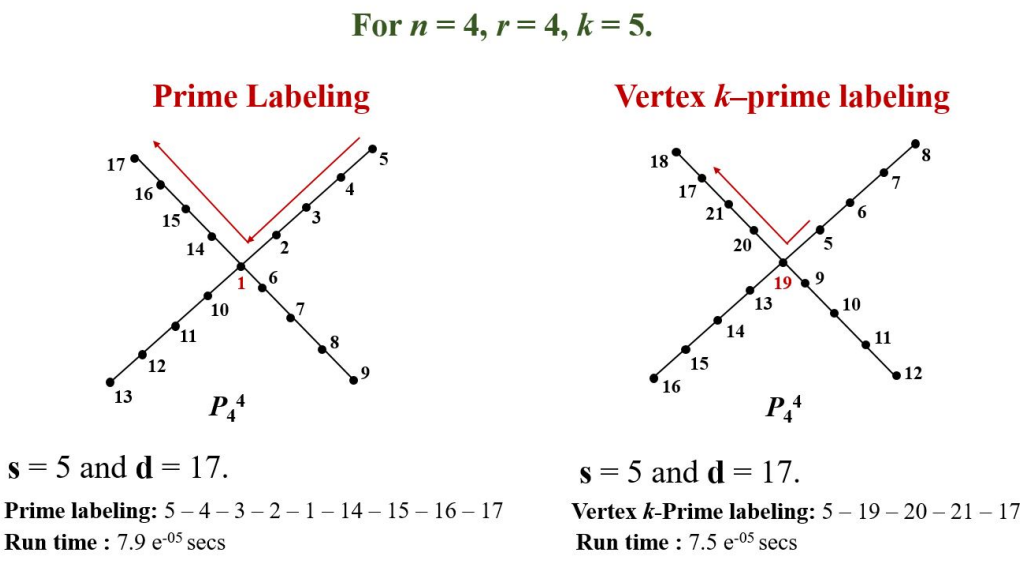

FIGURE 2. A comparison of prime and vertex k-prime labeling using descendant search technique for one point union of path graph

The experiment provides us with valid results; however, the source and the destination nodes can depend on how the tree has branched out. It will not be valid if the source and destination are closer in the case of vertex k-prime labeling but not in prime labeling. The results are not conclusive since the labeling might modify the nodes altogether. To remove the discrepancy and validate the results based on different labeling schemes, we

modified the above-mentioned source and destination algorithm to run the descendant search from the root to all other nodes present in the tree. We calculated the time taken for the descendant search to complete the processing of all the nodes from the root and completed our findings.

Conclusion

We ran different scenarios using the above-discussed traversal mechanisms to figure out how the applications scale-out for each scenario. The table of various nodes, number of tree paths, vertex k-prime value and total nodes for each scenario to understand different labeling techniques, are elaborated in the thesis. Vertex k-prime labeling requires an additional k value to start the processing as root and the nodes are placed around the largest prime number within the total nodes. This processing might have increased the time for vertex k-prime to traverse each level.

The results are interesting for descendant search since, there are two different scenarios tested to validate our findings. In the first scenario with different sources and destinations, vertex k-prime is faster than prime labeling based on the source and destination present in the tree. If, the nodes are far apart then, vertex k-prime labeling might have taken more time to compute the distance between them. This scenario did not provide us with valid results since, it is dependent on tree arrangement and how source and destination nodes are placed in the tree. To find the real results associated with the descendant search, we came up with the second scenario.

In the second scenario, the run time taken from the source to all the nodes is calculated. This provides us with more genuine results when compared to the first scenario. The time complexity of both prime and vertex k-prime labeling using the second scenario is calculated and depicted in Table 1. We observe that the time taken by prime labeling and vertex k-prime labeling varies which depends upon the distribution of the labeling on the tree.

The coding Language C^{++} of one point union of path graph for both prime labeling and vertex k-prime labeling is given in the Appendices **A** and **B**. The run time complexity for each descendant search with various number of vertices and edges is Big O (Number of vertices $\times$ Number of edges), that is, Big O$((nr+1)\times(nr))$ where when the value of n is 40 and r is 25, descendant search of prime labeling is 1.279 secs and descendant search of vertex k-prime labeling is 1.017 secs, respectively. The time taken for each descendant search with different parameters are provided in the table using Processor Intel (R) Core (TM) i5$-$8250U CPU@1.60 GHZ 1.80 GHZ with installed RAM 8.00 GB and C^{++} code is given in Appendix **C**.

TABLE 1. Run time of prime labeling and vertex k-prime labeling for $k = 400$ using second scenerio

No of nodes per t	**No of paths**	**Total nodes**	**Largest prime number**	**Prime labeling**	**Vertex k-Prime ($k = 400$)**
n	r	$n * r$	l	**Descendent search (in secs)**	**Descendent search (in secs)**
10	10	100	499	0.014	0.016
40	10	400	797	0.198	0.211
40	25	1000	1399	1.279	1.017
50	40	2000	2399	4.235	3.656
50	50	2500	2897	6.372	6.465
75	40	3000	3391	9.645	9.81
70	50	3500	3889	10.462	16.726
100	40	4000	4397	22.087	21.782
200	40	8000	8389	99.477	87.596
250	35	8750	9137	109.195	107.589
300	30	9000	9397	74.296	75.956
250	40	10000	10399	131.749	123.297
300	40	12000	12391	226.448	197.246
10	40	400	797	0.245	0.288
40	100	4000	4397	21.654	23.392
40	200	8000	8389	88.335	90.818
9	6	54	449	0.007	0.005
5	5	25	421	0.002	0.001
8	12	96	491	0.012	0.015
59	5	295	691	0.135	0.139

4. Conclusions

This paper attempts to prove one point union of path graph admits vertex k-prime labeling and have provided a comparison of the time complexity of prime labeling and vertex k-prime labeling which has been obtained by the application of one point union of path graphs.

Acknowledgement. The authors would like to extend their gratitude to the reviewers for their valuable comments and suggestions.

References

[1] Bloom, G. S. and Golomb, S. W., (1977), Applications of numbered undirected graphs, Proc. IEEE, 65, pp. 562-570.
[2] Bondy, J. A. and Murty, U. S. R., (1982), Graph Theory with Applications, Macmillon, London.
[3] Gallian, J. A., (2023), A dynamic survey of graph labeling, The Electronic Journal of Combinatorics, 18, DS6.
[4] Rosa, A., (1967), On certain valuations of the vertices of a graph, Theory of Graphs (Internat. Symposium, Rome, July 1966), Gordon and Breach, N. Y. and Dunod Paris, pp. 349-355.

[5] Teresa Arockiamary, S. and Vijayalakshmi, G., (2023), Vertex k-Prime Labeling of Cyclic Snakes, Communications in Mathematics and Applications, 14 (1), pp. 9-20.
[6] Teresa Arockiamary, S. and Vijayalakshmi, G., (2023), Vertex k-prime labeling on graphs, European Chemical Bulletin, 12 (Special Issue 4), pp. 9627-9633.
[7] Teresa Arockiamary, S. and Vijayalakshmi, G., (2023), Vertex k-Prime Labeling of Theta Graphs, Indian Journal of Science and Technology, 16 (26), pp. 2008-2015.
[8] Tout, A., Dabboucy, A. N. and Howalla, K., (1982), Prime labeling of graphs, Nat. Acad. Sci. Letters, 11, pp. 365-368.

Dr. TERESA AROCKIAMARY S. Associate Professor, Department of Mathematics, Stella Maris College has 22 years of experience in teaching at Stella Maris College. Areas of interest include Graph Theory, Algebra and Analysis. Title of the thesis is "Total Edge Irregularity Strength of Certain Interconnection Networks". Guiding PhD students since 2018. Currently, four students are pursuing under my guidance. One student have submitted her thesis and three students completed their viva. Published 39 research papers in reputed journals. Also was a reviewer of research papers, chairperson and scientific committee member. Was external examiner, resource person, subject expert to colleges and organizing member for conferences.

Dr. VIJAYALAKSHMI P. is working as a Guest Faculty in the Department of Mathematics at Anna University, Chennai, Tamilnadu. She received her Doctor of Philosophy and Master of Philosophy degrees from the Stella Maris College, Affiliated to the University of Madras, Chennai. She has completed her Master's in Mathematics at The Ramanujan Institute for Advanced Study in Mathematics, University of Madras, Chennai. Titile of the thesis is "k-prime total labeling of graphs". Published 10 research papers in reputed journals. Her area of interest includes Graph Theory and its applications.

APPENDIX A

```
#define PRIME_GRAPH
#define PRIME_GRAPH
#include <iostream>
#include <unordered_map>
#include <vector>
#include <queue>
#include <list>
#define INT_MAX 2147483647
using namespace std;
class Prime_modeling {
private:
	int n; // nodes per strand
	int t; // no of strands
	unordered_map<int, vector<int>> adj_list;
	int total_nodes;
	int source = 1;
public:
	//constructor
	Prime_modeling(int no_of_strands, int nodes_per_strand, int nodes) :t(no_of_strands), n(nodes_per_strand), total_nodes(nodes) {}
	void create_adj_list();
	void display();
	bool predCheck(int start,int search_node,vector<int>& pred,vector<int>& distance);
	void find_descendent_search(int start,int search_node);
};

/* Adjcency list creation for undirected graph */
void Prime_modeling::create_adj_list() {
	if (n == 0 || t == 0 || (n * t < total_nodes)) {
		// exception handling
	}
```

```
	int count = 2;
	for (int i = 1; i <= t; i++) {
		adj_list[1].push_back(count);
		adj_list[count].push_back(1);
		for (int j = 1; j < n; j++) {
			int temp = count;
			int temp1 = ++count;
			adj_list[temp].push_back(temp1);
			adj_list[temp1].push_back(temp);
		}
		count++;
	}
}

/* Display the source node and destination nodes*/
void Prime_modeling::display() {
	cout << "source" << '\t' << "destination" << endl;
	for (auto i : adj_list) {
		for (int j : i.second) {
			cout << i.first << '\t' << j << endl;
		}
	}
}

/*stores the distance and pred of a node*/
bool Prime_modeling::predCheck(int start,int search_node,vector<int>& pred,vector<int>&
distance){
	list<int> queue;
	vector<bool> visited(total_nodes);
	visited[start] = true;
	distance[start] = 0;
	queue.push_back(start);
```

```
        // standard BFS algorithm
        while (!queue.empty()) {
        int u = queue.front();
        queue.pop_front();
        for (int i = 0; i < adj_list[u].size(); i++) {
                if (visited[adj_list[u][i]] == false) {
                        visited[adj_list[u][i]] = true;
                        distance[adj_list[u][i]] = distance[u] + 1;
                        pred[adj_list[u][i]] = u;
                        queue.push_back(adj_list[u][i]);
                          // We stop BFS when we find
                    // destination.
                        if (adj_list[u][i] == search_node)
                                return true;
                }
        }
        }
        return false;
}

/* works only when both the search nodes are there within total nodes */
void Prime_modeling::find_descendent_search(int start,int search_node) {
        vector<int> pred(total_nodes+1,-1);
        vector<int> distance(total_nodes+1,INT_MAX);

        if(predCheck(start,search_node,pred,distance) == false){
                cout<<"The graph is not connected"<<endl;
                return;
        }
        vector<int> path;
        int endNode = search_node;
        path.push_back(endNode);
```

```
        while(pred[endNode] != -1){
                path.push_back(pred[endNode]);
                endNode = pred[endNode];
        }
}
#endif
```

APPENDIX B

```
#define VERTEX K_PRIME
#define VERTEX K_PRIME
#include <iostream>
#include <unordered_map>
#include <vector>
#define INT_MAX 2147483647
using namespace std;
class Vertex_K_Prime_modeling {
private:
	int n; // nodes per strand
	int t; // no of strands
	int k; // k - prime value
	int lar_p; //lar_p - largest prime number within K + total_nodes
	std::unordered_map<int, vector<int>> adj_list;
	int total_nodes;
	int source = 0;
public:
	// constructor
	Vertex_K_Prime_modeling(int no_of_strands, int largest_prime, int
nodes_per_strand, int k_prime, int nodes) :t(no_of_strands), lar_p (largest_prime),
n(nodes_per_strand), k(vertex_k_prime), total_nodes(nodes + vertex_k_prime) {}
	void create_adj_list();
	void display();
	void find_descendent_search(int start,int search_node);
	bool predCheck(int start,int search_node,vector<int>& pred,vector<int>& distance);
};
// Finding a prime number for the total_nodes
int find_prime(int total_nodes) {
	vector<bool> prime_no(total_nodes + 1,false);
	for (int i = 2; i <= total_nodes; i++) {
```

```
            if (prime_no[i] == false) {
                for (int j = 2; i * j <= total_nodes; j++) {
                    prime_no[i * j] = true;
                }
            }
        }
        for (int i = prime_no.size() - 1; i >= 0; i--) {
            if (!prime_no[i]) {
                return i;
            }
        }
}

/* Adjcency list creation */
void Vertex_K_Prime_modeling::create_adj_list() {
        if (n == 0 || t == 0 || (n * t < total_nodes)) {
            // exception handling
        }
        int prime = lar_p;
        source = prime;
        int count = k;
        int temp_count = 0;
        for (int i = 1; i <= t; i++) {
            if (i < t) {
                adj_list[prime].push_back(count);
                adj_list[count].push_back(prime);
                for (int j = 1; j < n; j++) {
                    int temp = count;
                    int temp1 = ++count;
                    adj_list[temp].push_back(temp1);
```

```
                adj_list[temp1].push_back(temp);
            }
            count++;
        }
        if (i == t) {
            // last path ...
    /* Path creation when the number of nodes are greater than the greater prime */
            if (prime < total_nodes) {
                temp_count = count;
                int start = prime + 1;
                adj_list[prime].push_back(start);
                adj_list[start].push_back(prime);
                while (start < total_nodes) {
                    int temp = start;
                    int temp1 = ++start;
                    adj_list[temp].push_back(temp1);
                    adj_list[temp1].push_back(temp);
                }
                adj_list[start].push_back(temp_count);
                adj_list[temp_count].push_back(start);
                while (temp_count < prime - 1) {
                    int temp = temp_count;
                    int temp1 = ++temp_count;
                    adj_list[temp].push_back(temp1);
                    adj_list[temp1].push_back(temp);
                }
            }
            else {
                /* Path creation when the number of nodes are equal to the
greater prime */
                adj_list[prime].push_back(count);
```

```
                                adj_list[count].push_back(prime);
                                for (int j = 1; j < n; j++) {
                                        int temp = count;
                                        int temp1 = ++count;
                                        adj_list[temp].push_back(temp1);
                                        adj_list[temp1].push_back(temp);
                                }
                        }
                }
        }
}

/* Display the source node and destination nodes*/
void K_Prime_modeling::display() {
        cout << "source" << '\t' << "destination"<<endl;
        for (auto i : adj_list) {
                for (int j : i.second) {
                        cout << i.first << '\t' << j << endl;
                }
        }
}
/*stores the distance and pred of a node*/
bool Vertex_K_Prime_modeling::predCheck(int start,int search_node,vector<int>&
pred,vector<int>& distance){
      list<int> queue;
      vector<bool> visited(total_nodes);
      visited[start] = true;
      distance[start] = 0;
      queue.push_back(start);
      // standard BFS algorithm
      while (!queue.empty()) {
```

```
            int u = queue.front();
            queue.pop_front();
            for (int i = 0; i < adj_list[u].size(); i++) {
                if (visited[adj_list[u][i]] == false) {
                    visited[adj_list[u][i]] = true;
                    distance[adj_list[u][i]] = distance[u] + 1;
                    pred[adj_list[u][i]] = u;
                    queue.push_back(adj_list[u][i]);
                   // We stop BFS when we find
                    // destination.
                    if (adj_list[u][i] == search_node)
                        return true;
                }
            }
        }
        return false;
}
/* works only when both the search nodes are there within total nodes */
void Vertex_K_Prime_modeling::find_descendent_search(int start,int search_node) {
        vector<int> pred(total_nodes+1,-1);
        vector<int> distance(total_nodes+1,INT_MAX);
        if(predCheck(start,search_node,pred,distance) == false){
            //cout<<"the nodes are not in the graph"<<endl;
            return;
        }
        vector<int> path;
        int endNode = search_node;
           path.push_back(endNode);
           while(pred[endNode] != -1){
               path.push_back(pred[endNode]);
```

APPENDIX C

```
#include "Prime_graph.h"
#include "Vertex_K_Prime.h"
#include <time.h>
#include <iostream>
int main()
{
   /* UI for finding the prime and vertex_k_prime modeling*/
   cout << "1. Prime Modeling" << endl;
   cout << "2. Vertex K_Prime Modeling" << endl;
   int in;
   cin >> in;

   //prime modeling
   if (in == 1) {
      int n, t, type_DS, total;
      cout << "n = ";
      cin >> n;
      cout << "t = ";
      cin >> t;
      cout << "total nodes = ";
      cin >> total;
      total = total + 1;
      Prime_modeling  p_m(t, n, total); //inputs for prime modeling
      p_m.create_adj_list(); // prime modeling adj_list creation
      p_m.display(); // prime modeling list display
      int source, target;
      cout << "DESCENDENT SEARCH" << endl; // Descendent search
            cout << "1. Individual search check with different source and target" << endl;
            cout << "2. Prime number 1 as source and all other nodes as target" << endl;
             cin >> type_DS;

      if (type_DS == 1)
      {
         cout << "Source = ";
         cin >> source;
         cout << "Target = ";
         cin >> target;

         if(target > total+1){
            cout<<"check the inputs"<<endl;
         }
         vector<int> pathNodes;
         clock_t start1 = clock();
         p_m.find_descendent_search(source, target);
         cout<<endl;
         //time of execution for descendent search
```

```
        cout << "time taken : "<<double(clock() - start1) / double(CLOCKS_PER_SEC) <<
"sec" << endl;
      }
      if (type_DS == 2)
      {
         clock_t start2 = clock();
         source = 1;
         for (int i = 2; i <= total; i++) {
            p_m.find_descendent_search(source, i);
         }
         cout<<endl;
         cout << double(clock() - start2) / double(CLOCKS_PER_SEC) << "sec" << endl;
      }

   }

   //Vertex K-prime modeling
   if (in == 2) {
      int n, t, k, i,lar_p,type_DS, total;
      clock_t start2;
      cout << "n = ";
      cin >> n;
      cout << "t = ";
      cin >> t;
      cout << "k = ";
      cin >> k;
      cout << "Largest prime within K + total nodes:";
      cin >> lar_p;
      cout << "total nodes = ";
      cin >> total;

      if (lar_p > (k + total)) {
         cout << "largest prime is greater than the total nodes" <<endl;
      }

      Vertex_K_Prime_modeling k_m(t,lar_p,n,k,total); //inputs for Vertex_K_prime
      k_m.create_adj_list(); // adj_list creation
      k_m.display(); // display the list of connected nodes
      int source, target;
      cout << "DESCENDENT SEARCH" << endl;
      cout << "1. Individual search check with different source and target :" << endl;
      cout << "2. Vertex K Prime with largest prime number as "
         "source and all other nodes as target:" << endl;
      cin >> type_DS;

      if (type_DS == 1)
      {
         cout << "Source = ";
```

```
            cin >> source;
            cout << "Target = ";
            cin >> target;

            start2 = clock();
            k_m.find_descendent_search(source, target);
            cout<<endl;
            //time of execution for descendent search
            cout << double(clock() - start2) / double(CLOCKS_PER_SEC) << "sec" << endl;
        }
        if (type_DS == 2)
        {
            start2 = clock();
            source = lar_p;
            for (i = 1; i <= total; i++) {
                k_m.find_descendent_search(source, (k + i));
            }
            cout << double(clock() - start2) / double(CLOCKS_PER_SEC) << "sec" << endl;
        }

    }
    return 0;
}
```