T.C.

GEBZE TECHNICAL UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

MINIMUM K-CHINESE POSTMAN PROBLEM

AKİF KARTAL

SUPERVISOR
PROF. DR. DİDEM GÖZÜPEK

GEBZE
2022

T.C.
GEBZE TECHNICAL UNIVERSITY
FACULTY OF ENGINEERING
COMPUTER ENGINEERING DEPARTMENT

MINIMUM K-CHINESE POSTMAN PROBLEM

AKİF KARTAL

SUPERVISOR
PROF. DR. DİDEM GÖZÜPEK

2022
GEBZE

GEBZE TECHNICAL UNIVERSITY	GRADUATION PROJECT JURY APPROVAL FORM

This study has been accepted as an Undergraduate Graduation Project in the Department of Computer Engineering on 16/06/2022 by the following jury.

JURY

Member
(Supervisor) : PROF. DR. DİDEM GÖZÜPEK

Member : YRD.DOÇ.DR. ZAFEİRAKİS ZAFEİRAKOPOULOS

ABSTRACT

In this project, we present a heuristic and exhaustive search algorithm for the minimum k-Chinese postman problem. We considered the minimum k-Chinese postman problem is, given a multigraph $G = (V, E)$ initial vertex $s \in V$ length $l(e) \in N$ for each $e \in E$ the *minimum k-Chinese postman problem* is to find *k* tours(cycles) such that each containing the initial vertex *s* and each edge of the graph has been traversed at least once and the most expensive tour is minimized. This problem is NP-hard and we tried to solve it with a polynomial-time algorithm. For this purpose, we created one polynomial-time algorithm and one exponential-time algorithm and we made a complexity analysis for these algorithms. After creating algorithms we compare them with different parameters by looking at the results and running time. We saw that when the k value is increasing, the polynomial-time heuristic algorithm produces better results in a very short time.

Keywords: heuristic, exhaustive search, NP-hard, polynomial-time, exponential-time, parameters, running time, complexity analysis

ÖZET

Bu projede minimum k-Chinese postman problemi için sezgisel ve kapsamlı arama algoritması sunuyoruz. Minimum k-Chinese postman probleminin tanımı şu şekildedir. Verilen bir multigrafda s adında bir başlangıç noktası vardır ayrıca her 2 nokta arasındaki yol için bir yol uzunluğu vardır. Bu probleme göre bizim amacımız, bu graf üzerinde öyle bir k tane tur ya da dolaşım bulacağız ki her bir turda başlangıç noktası olacak ve graf'taki her bir yoldan en az bir kere geçilmiş olacaktır. Bizim amacımız bu k tane turdaki en büyük uzunluğu sahip turun uzunluğunu en aza indirmektir. Bu problem bir NP-hard problemdir ve biz bu problemi bir polinom zamanlı algoritma ile çözmeye çalıştık. Bu amaç için bir tane polinom zamanlı bir tanede üstel zamanlı olmak üzere iki tane algoritma geliştirdik ve bu algoritmaların karmaşıklık analizini yaptık. Bu algoritmaları oluşturduktan sonra farklı parametrelerle test edip birbirleri ile sonuç ve çalışma süresi bakımından karşılardık ve sonuç olarak gördük ki k değerini artırdıkça çok kısa bir sürede polinom zamanlı sezgisel algoritma daha iyi sonuçlar buluyor.

Anahtar Kelimeler: sezgisel, kapsamlı arama, NP-hard, polinom zamanlı, üstel zamanlı, çalışma süresi, karmaşıklık analizi

ACKNOWLEDGEMENT

My dear supervisor, who does not spare his interest and support in the planning, research, execution, and formation of this project, whose vast knowledge and experience I have benefited from, I would like to express my eternal and sincere thanks to Didem Gözüpek.

Also, I would like to express my endless and sincere thanks to my teacher, Zafeirakis Zafeirakopoulos who is leading the way for the improvement of this project.

Lastly, I would like to express my respect and love to my family, who supported me in every way during my education, and to all my teachers who set an example for me with their lives.

Akif Kartal

LIST OF SYMBOLS AND ABBREVIATIONS

Abbreviation	:	**Explanation**
GUI	:	Graphical User Interface
CPP	:	Chinese Postman Problem
MIN	:	Minimum
MAX	:	Maximum

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

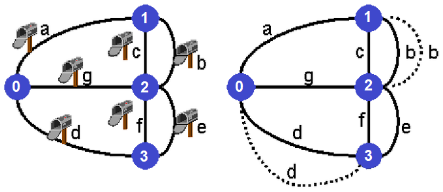
A graph is a data structure composed of a set of objects (nodes) equipped with connections (edges) among them. Graphs can be directed if the connections are oriented from one node to another (e.g. Alice owes money to Bob), or undirected if the orientation is irrelevant and the connections just represent relationships (e.g. Alice and Bob are friends). A graph is said to be complete if all nodes are connected to each other. A directed graph with no loops is said to be acyclic. A few practical examples of graphs are friendship networks (e.g. on social media), genealogical (family) trees, molecules, particles produced at the Large Hadron Collider, and a company's organizational chart.[1]
To find the most efficient way of traversing an entire graph is a widely spread problem in today's society. For a snow truck to plow all snow on every street in a town in the minimal consumed time is only one of a vast amount of applications of this problem. Finding solutions for these problems would lead to both a financial and an environmental improvement for companies and cities all over the world. Therefore in this project we have tried to solve such a problem called the minimum k-Chinese postman problem.[2]

1.1. Project Definition

In this project, we tried to solve Minimum k-Chinese Postman Problem which is given a multigraph $G = (V, E)$ initial vertex $s \in V$ length $l(e) \in N$ for each $e \in E$ the *minimum k-Chinese postman problem* is to find *k* tours(cycles) such that each containing the initial vertex *s* and each edge of the graph has been traversed at least once and the most expensive tour is minimized.
To solve this problem, we have implemented 2 different algorithms and evaluated them in different perspectives.

This problem has following inputs.

- *s*, initial vertex
- *k*, given positive number
- *l(e)*, length for each edge
- *n*, number of vertices(nodes)
- *e*, number of edges



(a) Simple Multigraph Example

(b) Postmans

1.2. The Goal of the Project

Making reason for this project is to implement a heuristic algorithm for the Minimum k-Chinese Postman Problem from literature.

In literature, there are a reasonable number of algorithms for this problem but finding implemented one is nearly impossible because these types of algorithms are NP-hard. Therefore implementing such algorithms is very important.

Also, by using this study, people can see how to solve these types of problems, how to make a complexity analysis, and how to make a comparison between different algorithms.

Lastly, after publishing this project people can use these solutions and improve them.

2. Project Design and Details

In order to make this project, we need to consider and determine details about this project.

2.1. Project Design Plan

In the following image you can see the project design plan in a good manner.

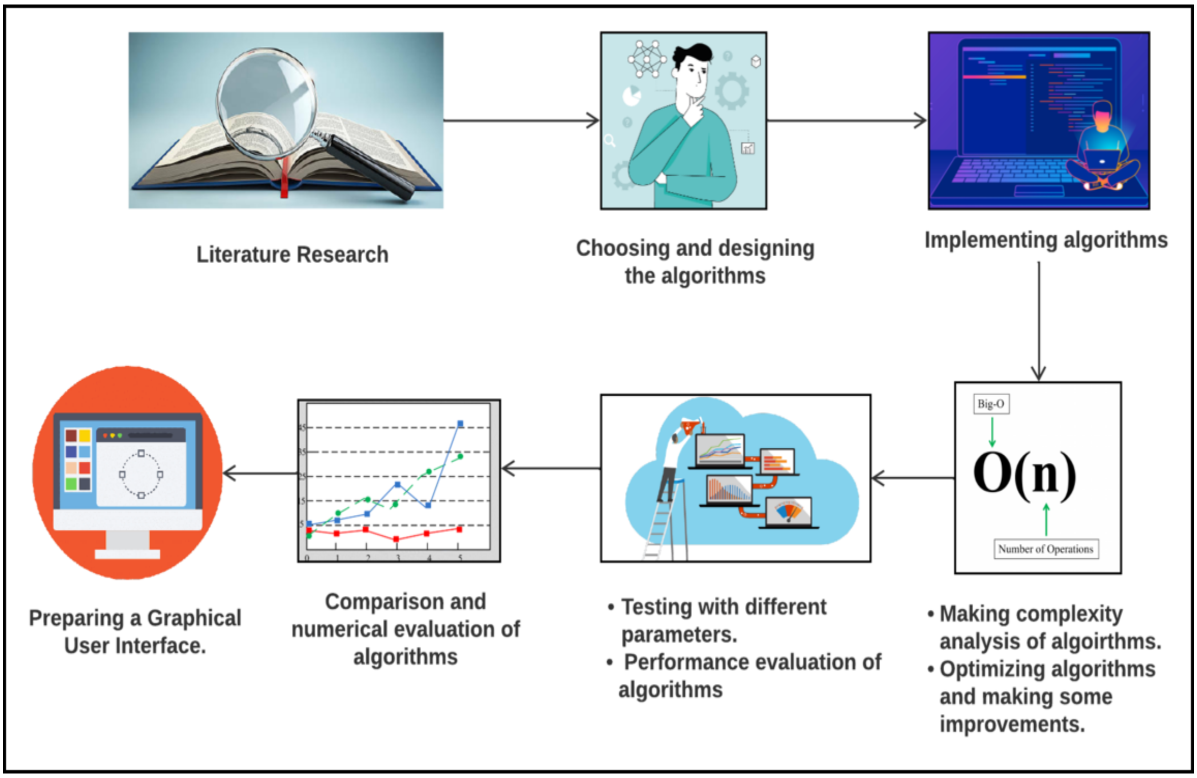


Figure 2.1: Project Design Plan

2.2. Project Requirements

- Making literature research and understanding the problem.
- Choosing and designing algorithms.
- Implementing both heuristic and exhausted search algorithms.
- Making complexity analysis of the algorithms.

- Testing with different parameters and performance evaluation of algorithms.
- Making comparison and numerical evaluation of algorithms.
- Showing the comparison average results on the charts.
- Creating a GUI and running algorithms on that GUI.

2.2.1. Literature Research

In order to solve this problem we have to make a deep research and reading in literature. For Minimum k-CPP, I have read following articles.

Author	Article
Dino Ahr, Gerhard Reinelt	New heuristics and lower bounds for the min-max k-chinese postman problem[4]
Kaj Holmberg	Heuristics for the weighted k-Chinese/rural postman problem with a hint of fixed costs with applications to urban snow removal[5]
G. Gutin, G. Muciaccia	Parameterized Complexity of k-Chinese Postman Problem[6]
Anton Hölscher	A Cycle-Trade Heuristic for the Weighted k-Chinese Postman Problem [2]
Dino Ahr, Gerhard Reinelt	A tabu search algorithm for the min–max k-Chinese postman problem[7]

Table 2.1: Found and Read Articles while Researching

2.2.2. Tools and Technologies

In order to make this project following tools and technologies have been used.

1. **Python 3.9:** This is used as as programming language to implement algorithms and gui.
2. **Windows 10:** This is used as operating system.
3. **PyCharm IDE:** This is used to as development environment.
4. **igraph:** This python library is used to generate and draw graphs.
5. **PyQt5, Qt Designer:** These are used to make graphical user interface.
6. **Git and Github:** These are used to keep source code.

Figure 2.2: Tools and Technologies

3. Heuristic Algorithm

3.1. Augment-Merge Algorithm

In order to solve the Minimum k-Chinese Postman Problem, we have implemented a heuristic augment-merge algorithm.[4]

The idea of the algorithm is roughly as follows. We start with a closed walk C_e for each edge $e = v_i, v_j \in E$, which consists of the edges on the shortest path between the depot node v_1 and v_i, the edge e itself, and the edges on the shortest path between v_j and v_1, i.e. $C_e = (SP(v_1, v_i), e, SP(v_j, v_1))$. Then we successively merge two closed walks trying to keep the tour weights low and balanced until we arrive at *k* tours.[4]

Steps of this algorithm are as follows.

1. Sort the edges e in decreasing order according to their weight.
2. In decreasing order according to $w(C_e)$, for each $e = v_i, v_j \in E$, create the closed walk $C_e = (SP(v_1, v_i)$, $e, SP(v_j, v_1))$, if *e* is not already covered by an existing tour.
3. Let $C = (C_1$, ... ,$C_m)$ be the resulting set of tours. If $m = k$ we are done and have computed an optimal *k* -postman tour.
4. If $m < k$ we add *k* − *m* "dummy" tours to *C*, each consisting of twice the cheapest edge incident to the depot node.
5. While $|C| > k$ we merge tour C_{k+1} with a tour from $C_1, ..., C_k$ such that the weight of the merged tour is minimized.

3.2. Implementation of Algorithm Steps

In this part, we will see the implementation of each step. Note that **tour and cycle are the same things.** Sometimes we will use tour sometimes we'll use the cycle keyword.

3.2.1. Graph Generating

In order to generate graph and print it, I have used the python **igraph** library.[8] I have created random graphs by generating random edges between vertices.

```
from igraph import *
def generate_random_graph(self, number_of_vertex, number_of_edges,
   initial_vertex):
    self.__initial_vertex = initial_vertex
    self.__g = Graph()
    self.__g.add_vertices(number_of_vertex)

    for i in range(len(self.__g.vs)):
        self.__g.vs[i]["id"] = i
        self.__g.vs[i]["label"] = i

    rand_edges = []
    parallel_edges = []
    isOkey = True
    degrees = [0] * number_of_vertex
    while isOkey:
        rand_edges = []
        parallel_edges = []
        for x in range(0, number_of_edges):
            value = random.sample(range(0, self.__g.vcount()), 2)
            while value in rand_edges:
                value=random.sample(range(0, self.__g.vcount()), 2)
            temp_val = [value[1], value[0]]
            if temp_val in rand_edges:
                parallel_edges.append(temp_val)
            else:
                for node in value:
                    degrees[node] = degrees[node] + 1
            rand_edges.append(value)
        if degrees[initial_vertex] == 0:
            isOkey = True
        else:
            isOkey = False

    self.__g.add_edges(rand_edges)
    rand_weights = []
    for x in range(0, len(self.__g.get_edgelist())):
        rand_weights.append(random.randint(5, 40))
    self.__g.simplify(combine_edges=None)
    self.__g.es['weight'] = rand_weights
```

Listing 3.1: Random Graph Generating

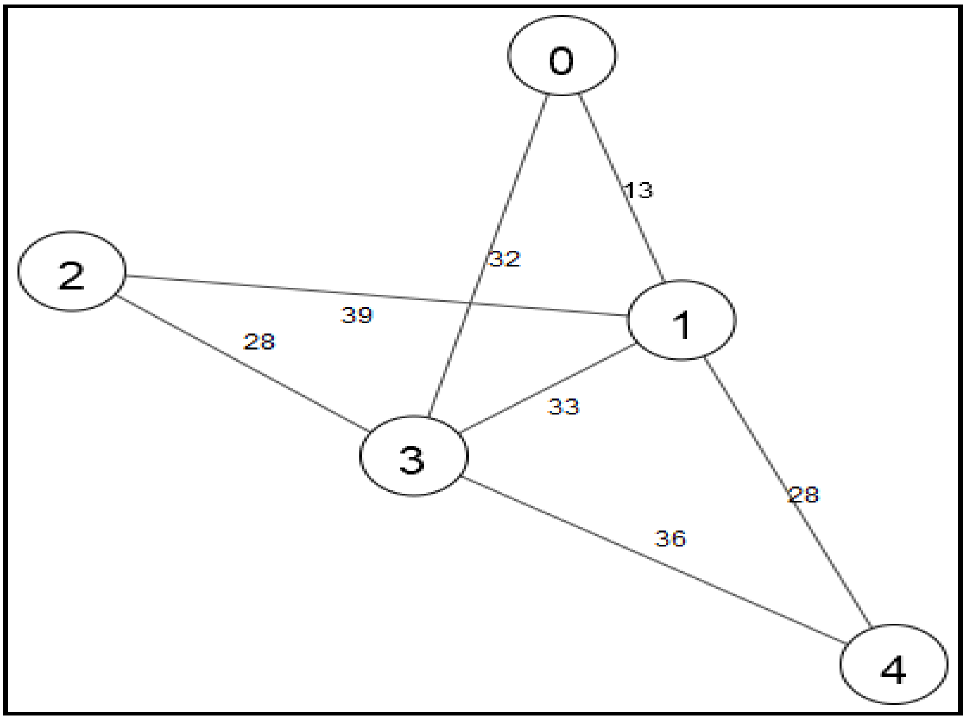


Figure 3.1: Generated Random Graph

3.2.2. Sorting the edges

- Sort the edges e in decreasing order according to their weight.

In order to sort edges, I have used the **bubble sort** algorithm because it is a polynomial-time algorithm and easy to implement.

```
def sort_edges_descending(self):
    weights = self.__my_graph.get_weights()
    edges = self.__my_graph.get_edges()
    self.__edges = edges
    n = len(weights)
    for i in range(n):
        for j in range(0, n - i - 1):
            if weights[j] < weights[j + 1]:
                edges[j], edges[j + 1] = edges[j + 1], edges[j]
                weights[j], weights[j+1] = weights[j+1], weights[j]
```

Listing 3.2: Sorting Edges with Bubble Sort

3.2.3. Creating the Closed Walks

- In decreasing order according to $w(C_e)$, for each $e = v_i, v_j \in E$, create the closed walk $C_e = (SP(v_1, v_i), e, SP(v_j, v_1))$, if e is not already covered by an existing tour.

In the following code, in order to create cycles we are using the igraph library shortest path algorithm. Igraph is using Dijkstra's algorithm to get the shortest path.

```python
def create_closed_walk(self, k):
    # for each edge
    for e in self.__sorted_edges:

        # e = {vi, vj}
        path3 = [e['start_node'], e['end_node']]
        walk = []
        # if e is not already covered by an existing tour.
        if not self.check_added(path3):
            # SP(v1, vi)
            path1 = self.__my_graph.get_shortest_path(self.__my_graph.get_initial_vertex(), path3[0])[0]
            # SP(vj, v1)
            path2 = self.__my_graph.get_shortest_path(path3[1], self.__my_graph.get_initial_vertex())[0]

            # try to create closed walk
            if self.try_to_merge(path1, path2, walk):
                self.add_edge_to_walk(walk, path3)
            # try to create closed walk
            elif self.try_to_merge(path1, path3, walk):
                self.add_edge_to_walk(walk, path2)
            # try to create closed walk
            elif self.try_to_merge(path2, path3, walk):
                self.add_edge_to_walk(walk, path1)
            else:
                walk.extend(self.get_maximum(path1, path2, path3))
            if len(walk) > 1:
                if walk[0] != walk[-1] and self.is_in_edge_list([walk[0], walk[-1]]):
                    walk.append(walk[0])
                self.__closed_walks.append(
                    {'cycle': walk, 'length': self.get_walk_length(walk), 'count': len(walk)})
```

Listing 3.3: Creating the Set of Cycles

3.2.4. Adding Dummy Tours

- If number of cycle(m) $< k$ add $k - m$ "dummy" tours to C, each consisting of twice the cheapest edge incident to the depot node.

In the following code, If number of cycle(m) $< k$ we will add $k - m$ new cycle. These cycles will be a simple tour on the edge that has minimum length.

```
if len(self.__closed_walks) < k:
    self.add_dummy_tours(k - len(self.__closed_walks))

def add_dummy_tours(self, missing_number):
    listLen = len(self.__sorted_edges)
    for i in range(listLen):
        e = self.__sorted_edges[(listLen - i) - 1]
        walk = [e['end_node'], e['start_node'], e['end_node']]
        if self.__initial_vertex in walk:
            for j in range(missing_number):
                self.__closed_walks.append(
                    {'cycle': walk, 'length': self.get_walk_length(
    walk), 'count': len(walk)})
            break
```

Listing 3.4: Adding Dummy Tours to the Cycle List

3.2.5. Merging Tours

- While number of cycle(m) $> k$ merge tour C_{k+1} with a tour from $C_1, ..., C_k$ such that the weight of the merged tour is minimized.

In this algorithm, if the number of cycles(m) $> k$, we have to merge these cycles with other cycles until we get exactly k cycle. In the following code pieces, you can see this operation.

```
elif len(self.__closed_walks) > k:
    self.merge_tours(k)

def merge_tours(self, k):
    listLen = len(self.__closed_walks)
    n = listLen - k
    is_ok = False
    for i in range(n):
        if i == 0:
            if self.merge_round2():
                print("round2")
                is_ok = True
```

```python
                listLen = len(self.__closed_walks)
                if listLen == k:
                    return True
            if not is_ok and self.merge_round1():
                print("round1")
                is_ok = True
                listLen = len(self.__closed_walks)
                if listLen == k:
                    return True
        if i > 0 and is_ok:
            is_ok = False
            if self.merge_round2():
                print("round2")
                is_ok = True
                listLen = len(self.__closed_walks)
                if listLen == k:
                    return True
            if not is_ok and self.merge_round1():
                print("round1")
                is_ok = True
                listLen = len(self.__closed_walks)
                if listLen == k:
                    return True
```

Listing 3.5: While number of cycle(m) > k merge tours

In the following merge round 1 code, we are trying to merge the cycle that has a minimum length with other cycles and if we merge, we remove it.

```python
def merge_round1(self):
    listLen = len(self.__closed_walks)
    for i in range(listLen):
        sm_el = self.__closed_walks[listLen - i - 1]
        walk_path = sm_el['cycle']
        for j in range(i, listLen - 1):
            next1 = self.__closed_walks[(listLen - j - 1) - 1]
            next_path = next1['cycle']
            if walk_path[-1] == next_path[0]:
                next_path.pop()
                next_path.extend(walk_path)
                self.__closed_walks[(listLen - j - 1) - 1] = {'cycle
    ': next_path, 'length': self.get_walk_length(next_path),'count':
    len(next_path)}
                del self.__closed_walks[listLen - i - 1]
                return True
    return False
```

Listing 3.6: Merge Round 1

In the following merge round 2 code, we try to merge the cycle that has a minimum number of nodes such that all edges in that cycle have already been visited. If we found such a cycle we remove it directly without merging.

```
def merge_round2(self):
    listLen = len(self.__closed_walks)
    self.__closed_walks = sorted(self.__closed_walks, key=itemgetter
    ('count'))
    for i in range(listLen):
        sm_el = self.__closed_walks[i]
        sm_walk = sm_el['cycle']
        for j in range(i + 1, listLen):
            big_el = self.__closed_walks[j]
            big_walk = big_el['cycle']
            n = len(sm_walk)
            big_ok = True
            for k in range(0, n):
                if k + 1 != n:
                    edge = [sm_walk[k], sm_walk[k + 1]]
                    is_ok = False
                    if self.sub_list_exists(big_walk, edge):
                        is_ok = True
                    edge.reverse()
                    if self.sub_list_exists(big_walk, edge):
                        is_ok = True
                    if not is_ok:
                        big_ok = False
            if big_ok:
                del self.__closed_walks[i]
                self.__closed_walks = sorted(self.__closed_walks,
    key=itemgetter('length'), reverse=True)
                return True
    self.__closed_walks = sorted(self.__closed_walks, key=itemgetter
    ('length'), reverse=True)
    return False
```

Listing 3.7: Merge Round 2

3.3. Source Code

You can see my heuristic algorithm all source code by using following links.

My Graph Generation Source Code

My Heuristic Algorithm Source Code

4. Exhaustive Search Algorithm

In this project, **to make a comparison**, we need to implement another algorithm. For this purpose, I have implemented a simple exhaustive search algorithm.

4.1. Algorithm Steps

Steps of this algorithm are as follows.

1. Firstly, it finds all possible cycles in a graph by using a simple recursive algorithm like Depth-first search.
2. Then, for the cycles found in the previous step, it finds all distinct combinations of a given length k.

   $$C(\text{all possible cycles}, k)$$

3. Then, it finds all set of cycles that satisfy and holds the problem conditions in found combinations.
4. Lastly, we choose the k cycle that has a minimum length in set of cycles that have been found in the previous step.

4.2. Implementation of Algorithm Steps

In this part, we will see the implementation of each step.

4.2.1. Finding All Cycles

In the following code, in order to find all possible cycles in a graph we are using a simple recursive algorithm like Depth-first search.[9]

```
for edge in self.graph:
    for node in edge:
        self.findNewCycles([node])

def findNewCycles(self, path):
    start_node = path[0]
    next_node = None
    sub = []

```

```
    # visit each edge and each node of each edge
    for edge in self.graph:
        node1, node2 = edge
        if start_node in edge:
            if node1 == start_node:
                next_node = node2
            else:
                next_node = node1
            if not self.visited(next_node, path):
                # neighbor node not on path yet
                sub = [next_node]
                sub.extend(path)
                # explore extended path
                self.findNewCycles(sub)
            elif len(path) > 2 and next_node == path[-1]:
                # cycle found
                p = self.rotate_to_smallest(path)
                inv = self.invert(p)
                if self.isNew(p) and self.isNew(inv):
                    self.cycles.append(p)
```

Listing 4.1: Finding all Possible Cycles in a Graph

4.2.2. Finding All k Combinations

In the following code, we will find all k combinations of found cycles in the previous step.[10]

```
self.findCombinations(self.cycles, self.k)

def findCombinations(self, A, k, out=(), i=0):

    # invalid input
    if len(A) == 0 or k > len(A):
        return

    # base case: combination size is ‘k‘
    if k == 0:
        # check problem conditions
        self.findMatch(out)
        return

    # start from the next index till the last index
    for j in range(i, len(A)):
        self.findCombinations(A, k - 1, out + (A[j],), j + 1)
```

Listing 4.2: Finding All k Combinations of Found Cycles

4.2.3. Finding All Proper Set of Cycles

In the previous step, while finding combinations, we have to choose the set of cycles that satisfy and holds the problem conditions. In the following code, you can see this operation.

```
def findMatch(self, cycle):
    if self.checkConditions(cycle):
        self.found.append(cycle)
        return True
    else:
        return False

def checkConditions(self, cycle):
    lst = [0] * len(self.graph)
    for e in cycle:
        i = 0
        for edg in self.graph:
            if self.check_added2(e, edg):
                lst[i] = 1
            i = i + 1
    if 0 in lst:
        return False
    return True

def check_added2(self, cycle, edge):
    if self.sub_list_exists(cycle, edge):
        return True
    temp_edge = [edge[1], edge[0]]
    if self.sub_list_exists(cycle, temp_edge):
        return True
    return False

def sub_list_exists(self, list1, list2):
    if len(list2) < 2:
        return False
    return ''.join(map(str, list2)) in ''.join(map(str, list1))
```

Listing 4.3: Finding All Set of Cycles that Satisfy the Problem Conditions

4.2.4. Choosing the Optimal k Cycle

After finding all set of cycles that satisfy and hold the problem conditions we have to choose k cycles among them such that the maximum length of these cycles is minimum. In the following code piece, you can see this operation.

```python
smp = MySimpleAlgorithm(self.__my_graph.get_edges(), self.__my_graph
    .get_initial_vertex(), self.__k, self.__n, cycles)

found = smp.main()

minLen = sys.maxsize
for e in found:
    lenList = []
    simple_closed_walk = []
    for walk in e:
        len1 = self.get_walk_length(walk)
        lenList.append(len1)
        simple_closed_walk.append({'cycle': walk, 'length': len1, '
    count': len(walk)})
    tempmax = max(lenList)
    if tempmax < minLen:
        self.__second_closed_walks = simple_closed_walk
        minLen = tempmax
```

Listing 4.4: Choosing the k Cycle that has a Minimum Length

4.3. Source Code

You can see my exhaustive search algorithm all source code by using following links.

My Exhaustive Search Algorithm Source Code

My Choosing the k Cycle that has a Minimum Length Source Code

5. Complexity Analysis of Algorithms

In this part, we will make complexity analysis of algorithms step by step.

5.1. Complexity Analysis of Heuristic Algorithm

In the following table, you can see the complexity analysis for each step of the heuristic algorithm. As you can see the merging tours take more time, therefore, it is worst-case of the heuristic algorithm.

* n = |E|

Complexity	Algorithm Step
$\mathcal{O}(n^2)$	Sort the edges e in decreasing order according to their weight.
$\mathcal{O}(n^3)$	For each $e = v_i, v_j \in E$, create the closed walk.
$\mathcal{O}(n^2)$	If number of cycle(m) $< k$ add $k - m$ "dummy" tours.
$\mathcal{O}(n^4)$	While number of cycle(m) $> k$ merge tour C_{k+1} with a tour from $C_1, ..., C_k$.

Table 5.1: Complexity Analysis of Heuristic Algorithm Steps

Overall Complexity of Heuristic Algorithm

Best Case: $\mathcal{O}(n^3)$
Average Case: $\mathcal{O}(n^3)$
Worst Case: $\mathcal{O}(n^4)$

5.2. Complexity Analysis of Exhaustive Search Alg.

In the following table, you can see the complexity analysis for each step of the exhaustive search algorithm. Since, finding all k combinations of all possible cycles takes the most time, it is the best, average and worst-case of the exhaustive search algorithm.

* m = |V|, n = |E|

Complexity	Algorithm Step
$\mathcal{O}(m+n)$	Finding all cycles in undirected graphs.[3]
$\mathcal{O}(n^n)$	Finding all k combinations of found cycles in the previous step.[10]
$\mathcal{O}(n^3)$	Finding all set of cycles that satisfy the problem conditions in found combinations.
$\mathcal{O}(n^4)$	Choosing the k cycle that has a minimum length in set of cycles that have been found.

Table 5.2: Complexity Analysis of Exhaustive Search Algorithm Steps

Overall Complexity of Exhaustive Search Algorithm

Best Case: $\mathcal{O}(n^n)$
Average Case: $\mathcal{O}(n^n)$
Worst Case: $\mathcal{O}(n^n)$

6. Performance Evaluation of Algorithms

In this part, since exhaustive search algorithm can only run with limited data sizes, we will evaluate the performance of the two algorithms separately.
To evaluate the performance of the algorithms, we will create **50 different graphs** and we evaluate the average of the results for each test case.

6.1. Performance Evaluation of Heuristic Algorithm

6.1.1. Changing Graph Size

In this test, we will increase the node count and edge count and we will see the running time of the algorithm. The graph density will be determined by the node count. Parameters will be as follows.

- **initial vertex** = 0
- **k** = 20
- **number of nodes** = 8, 10, 12, 14, 16
- **number of edges** = $((n * (n - 1))/2) - 5$

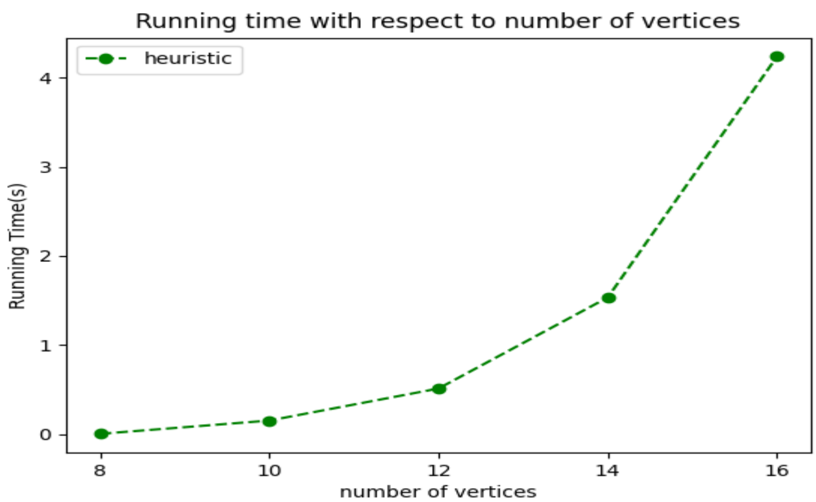


Figure 6.1: Running time with respect to number of vertices

In the above chart, you can see that while node count is increasing, the running time increases in the heuristic algorithm. This is happening because the **k value is constant** which means when the graph size is increasing we have more cycles than 20 such as 35, therefore, **we need to merge some cycles** and this operation takes time because it is the worst case for our heuristic algorithm.

6.1.2. Changing Graph Size and k Value

In this test, we will increase the node count, edge count and the k value according to node count and we will see the running time of the algorithm. The graph density will be determined by the node count. Parameters will be as follows.

- **initial vertex** = 0
- **number of nodes** = 8, 10, 12, 14, 16
- **number of edges** = $((n * (n - 1))/2) - 5$
- **k** = $n * 5$

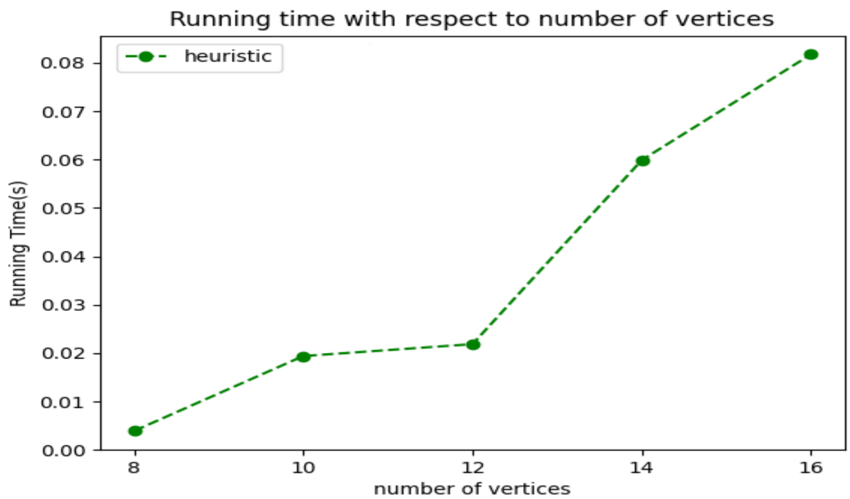


Figure 6.2: Running time with respect to number of vertices and k value

As you can see in the chart, still running time is increasing but in this case, times are very small. In the previous test, we have more than 4 seconds but now we have 0.081 seconds. This happens because the **k value is proportional with graph size** therefore, we don't have to merge tours and we gain time. **This means on the same graph if the k value is increased, taken time will decrease in the heuristic algorithm.**

6.2. Performance Evaluation of Exhaustive Search Alg.

6.2.1. Changing Graph Size

In this test, we will increase the node count and edge count and we will see the running time of the algorithm. The graph density will be determined by the node count. Parameters will be as follows.

- **initial vertex** = 0
- **k** = 4
- **number of nodes** = 4, 5, 6
- **number of edges** = 5, 8, 10

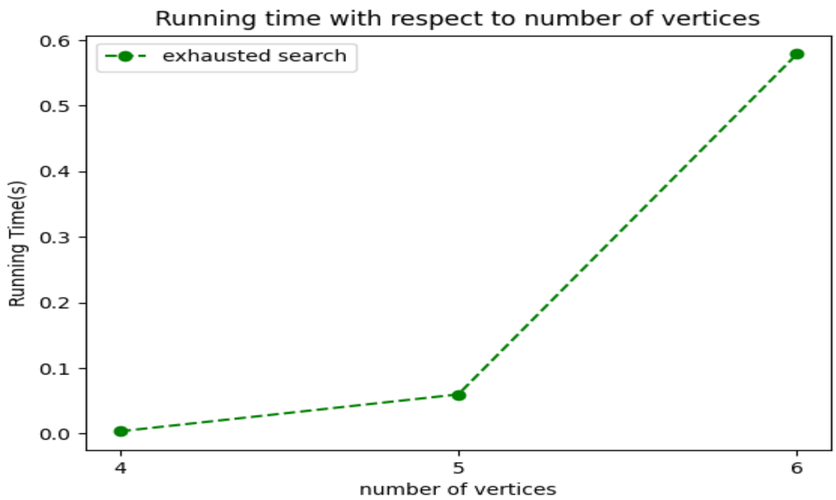


Figure 6.3: Running time with respect to number of vertices

In the above chart, you can see that while node count is increasing, the running time increases in the exhausted search algorithm. As you can see in the chart running time is small according to the exhausted search. This happens because the **k value is constant and small** also the graph size is small.

6.2.2. Changing Graph Size and k Value

In this test, we will increase the node count and edge count and the k value according to node count and we will see the running time of the algorithm. The graph density will be determined by the node count. Parameters will be as follows.

- **initial vertex** = 0
- **number of nodes** = 4, 5, 6
- **number of edges** = 5, 8, 10
- **k** = 3, 5, 7

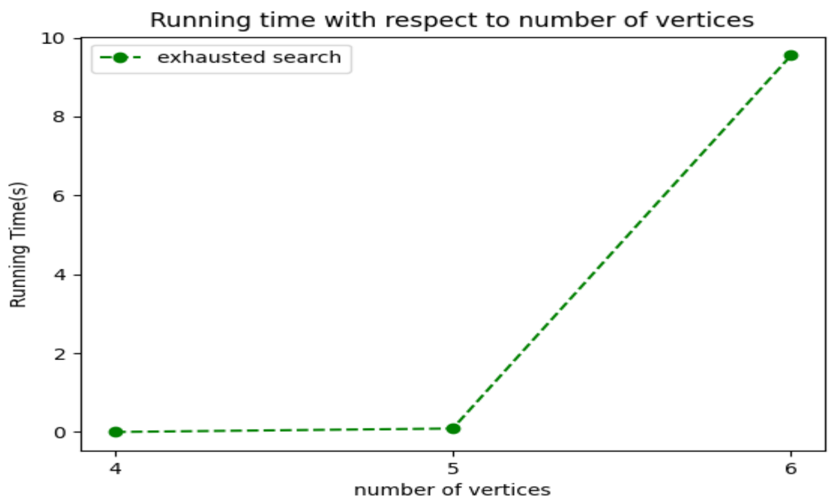


Figure 6.4: Running time with respect to number of vertices and k value

As you can see in the chart, still running time is increasing but in this case, times are very big. In the previous test, we have less than 1 second but now we have almost 10 seconds. This happens because the **k value is proportional with graph size and it gets a bigger value** this means on the same graph **if the k value is increased, taken time will increase in the exhausted search algorithm.**

7. Comparison and Numerical Evaluation

In this part, we will compare the two algorithms for running time and maximum length of cycles. Also, we will make a numerical evaluation of the results.

In order to make numerical evaluation we have following test cases.

- We will change the both number of nodes and the number of edges which means we will have a bigger graph. Also, the k value will be constant.
- We will change only the number of edges which means the density of the graph will change. Also, the k value and number of nodes will be constant.
- Lastly, we will change only the k value. Also, number of node and number of edge will be constant.

To compare the algorithms, we will create **20 different graphs** and we compare the average of the results for each test case.

7.1. Test Case 1

In this test, we will change the both number of nodes and the number of edges which means we will have a bigger graph. The k value will be constant.
Parameters will be as follows.

- **initial vertex** = 0
- **k** = 3
- **number of nodes** = 4, 5, 6, 7, 8
- **number of edges** = 6,9,10,12,12

7.1.1. Running Time Comparison

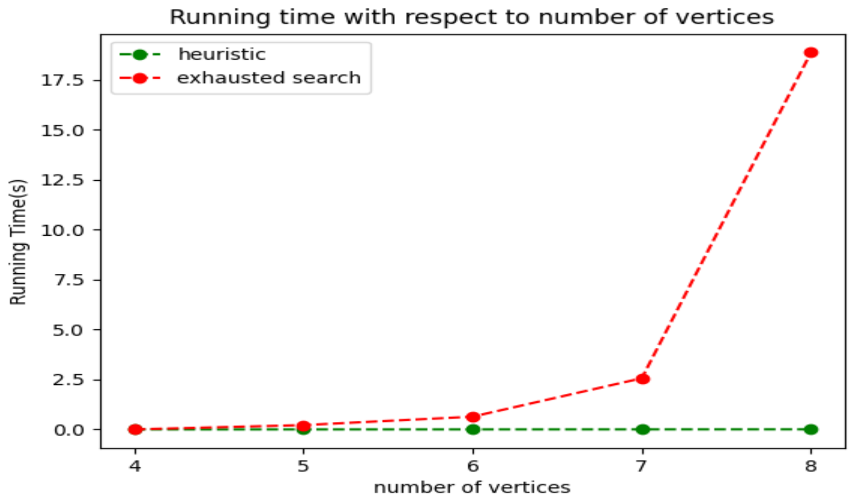


Figure 7.1: Running time with respect to number of vertices

In the above chart when the graph is growing running time of the exhausted search algorithm is increasing exponentially. This is an expected result because in bigger graphs we have more cycles and to get k combination of that cycles we need more time.

7.1.2. Maximum Length Comparison

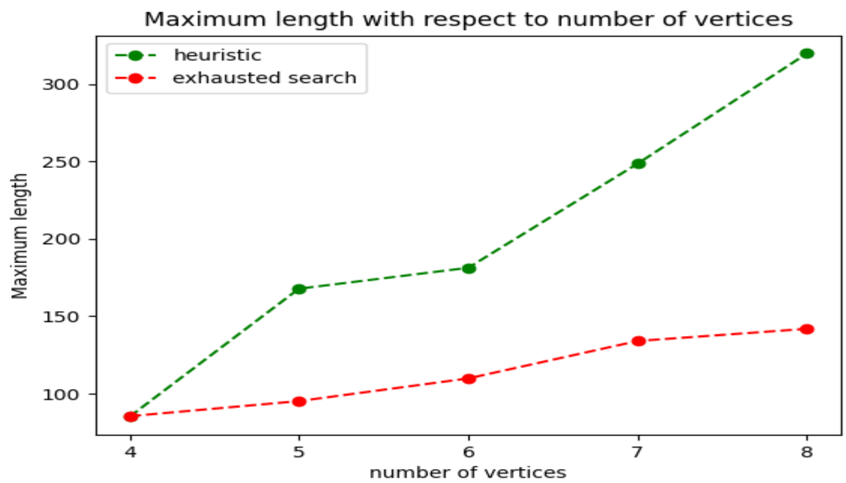


Figure 7.2: Maximum length with respect to number of vertices

In the above chart when the graph is growing and the k value is small and constant heuristic algorithm gets worse results. This is an expected result because when the **k value is small** we have to **merge** found tours after merging maximum length is increasing. For example, if k value 3 then let's say we get 10 cycles in heuristic algorithm in order to reduce 10 to 3 we have to merge them. After merging maximum length is increasing therefore exhausted search gets better results.

7.2. Test Case 2

In this test, we will change only the number of edges which means the density of the graph will change. The k value and number of nodes will be constant.
Parameters will be as follows.

- **initial vertex** = 0
- **k** = 4
- **number of nodes** = 6
- **number of edges** = 7, 8, 9, 10, 11

7.2.1. Running Time Comparison

Figure 7.3: Running time with respect to number of edges

In the above chart when the graph density is growing running time of the exhausted search algorithm is increasing exponentially. This is again an expected result because in the dense graphs **we have more cycles** and to get the k combination of that cycles we need more time.

7.2.2. Maximum Length Comparison

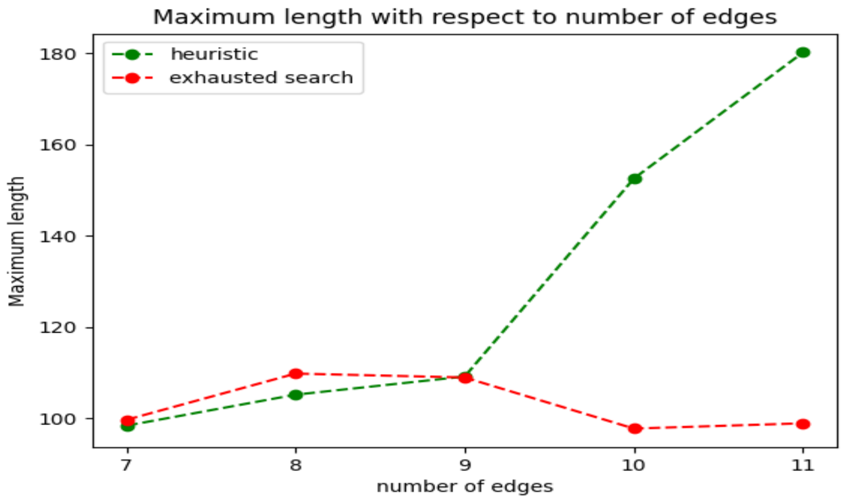


Figure 7.4: Maximum length with respect to number of edges

In the above chart when the graph density is growing for the points in which the graph is not dense, the heuristic algorithm gets a better result. But in dense graphs, the heuristic algorithm gets a worse result. This is again an expected result because in the dense graphs heuristic algorithm produce more cycle and since the **k value is constant** it has to merge them after merging operation result is increasing.

7.3. Test Case 3

In this test, we will change only the k value. Number of node and number of edge will be constant.
Parameters will be as follows.

- **initial vertex** = 0
- **k** = 5, 6, 7, 8, 9
- **number of nodes** = 6
- **number of edges** = 10

7.3.1. Running Time Comparison

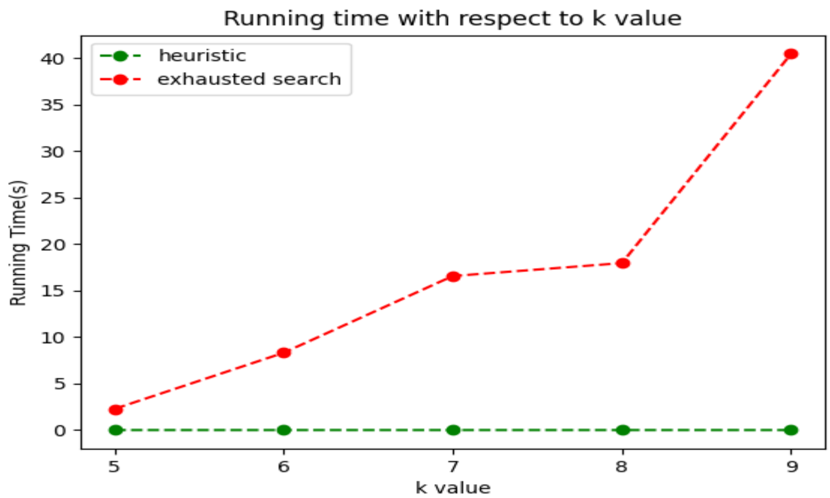


Figure 7.5: Running time with respect to k value

In the above chart when the k value is increasing running time of the exhausted search algorithm is increasing. This is again an expected result because the exhausted search algorithm has to get the k combination in any case and this operation takes time.

7.3.2. Maximum Length Comparison

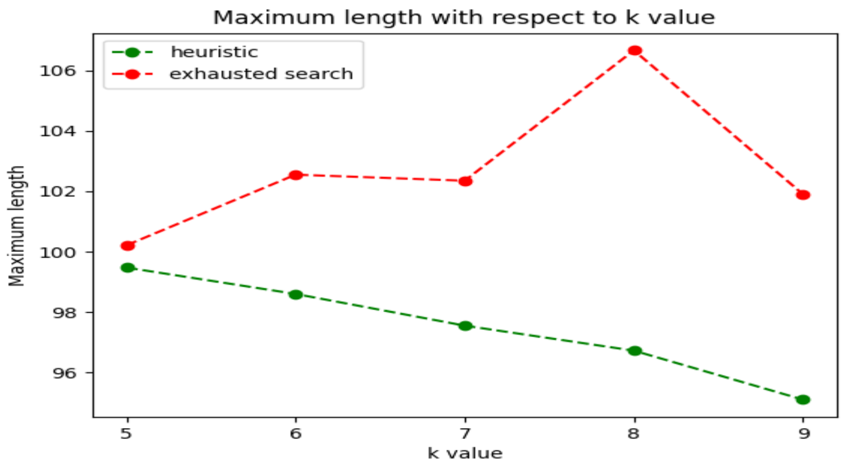


Figure 7.6: Maximum length with respect to k value

In the above chart when the **k value is increasing** heuristic algorithm gets **better results.** This is again an expected result because when the k value is big heuristic algorithm doesn't need to make a merging operation therefore it produces better results as we expect in this project.

7.4. Summary of Comparisons

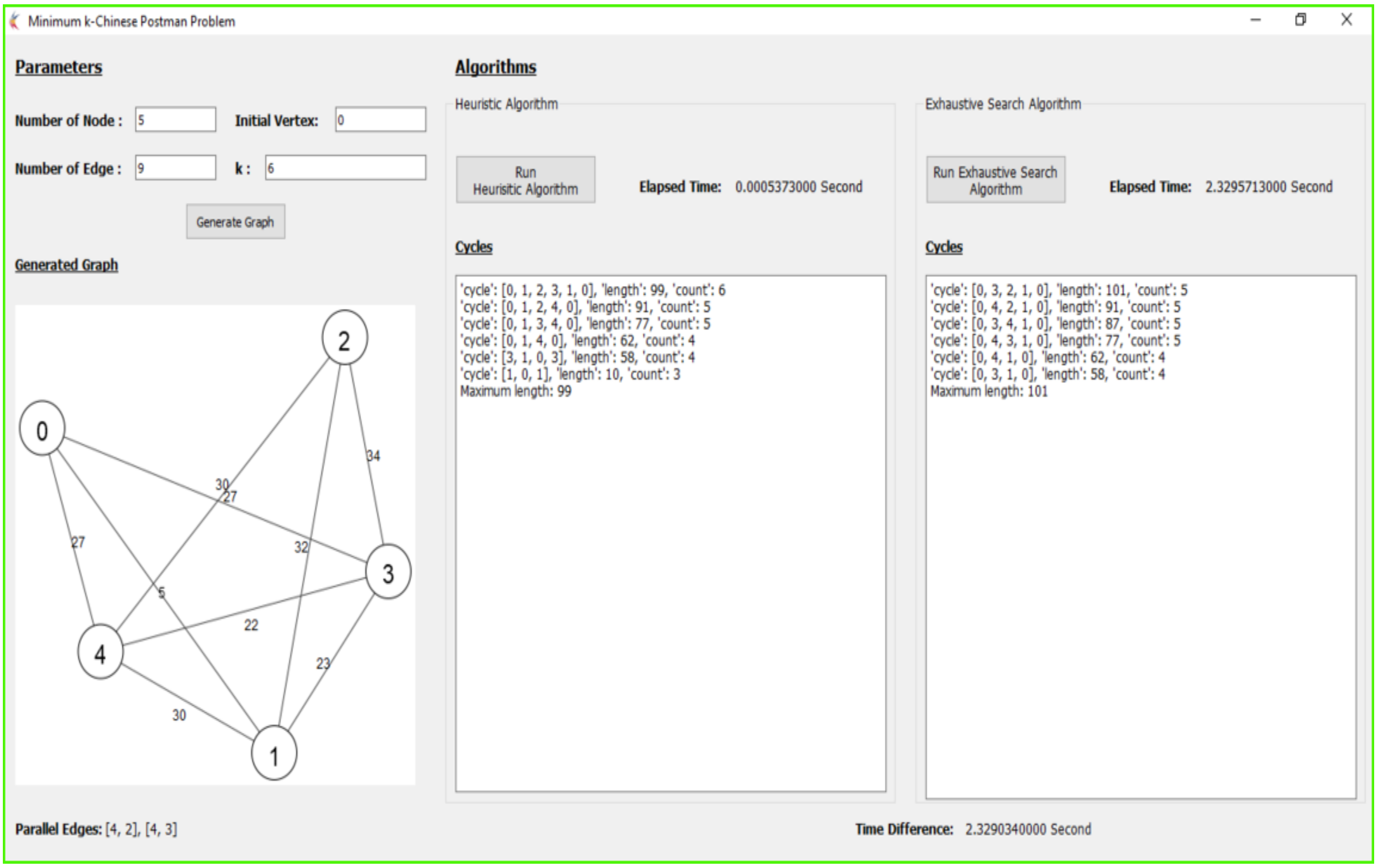
As we have seen in the above comparisons unfortunately in any case **exhausted search takes a long time.** But still, it can produce very good results. On the other hand heuristic algorithm is good for running time. But it produces bad results when the k value is small and not proportional to graph size. Because in that case, it has to make merge operation after merging the result is not good according to the exhausted search algorithm that means we can optimize the merging operating. But **when the k value is big and proportional to graph size heuristic algorithm produces good results** than the exhausted search algorithm in a very short time.

8. Graphical User Interface

For this project in order to show the results and how algorithms work, we need to create a Graphical User Interface. In this part, we will present our graphical user interface and its features. Features of my graphical user interface are as follows.

- User can enter the parameter values.
- User can generate and see graph.
- User can see parallel edges as text.
- User can run the algorithms.
- User can see the algorithm results.
- User can see the elapsed time for each algorithm.
- User can see the elapsed time difference between algorithms.

In the following image, you can see my graphical user interface.



* Click to see my GUI source code

9. Success Criteria

For this project, we have determined 4 success criteria. These are;

1. Heuristic algorithm complexity will be better than $\mathcal{O}(|E|^4)$
2. Creating 50 different random graphs for each test case in performance testing and creating 20 different random graphs for each comparison case and taking the average of them.
3. Getting results with the heuristic algorithm in less than 1 second when number of nodes < 25 and *k* is not constant.
4. When *k* is big and proportional to the number of edges, the results of the heuristic algorithm is at least %4 better than the exhaustive search algorithm.

Next, we will see that how I have accomplished these criteria one by one.

9.1. Criterion 1

- Heuristic algorithm complexity will be better than $\mathcal{O}(|E|^4)$

I have accomplished this criterion successfully. In heuristic algorithm I have $\mathcal{O}(|E|^3)$ complexity. For more detailed information check table 5.1.

9.2. Criterion 2

- Creating 50 different random graphs for each test case in performance testing and creating 20 different random graphs for each comparison case and taking the average of them.

I have accomplished this criterion successfully. In following images you can see my graph generation codes.

```python
self.init_values1()
for i in range(50):
    self.algo.generate_graph(s, n, e, k, i)
    res = self.algo.my_algorithm(k)
    self.time1_sum = self.time1_sum + res[1]

self.time1_avg = self.time1_sum / 50.0
self.time1_x.append(n)
self.time1_y.append(self.time1_avg)
```

(a) Graph Generation for Heuristic Algorithm

```python
self.init_values1()
for i in range(50):
    self.algo.generate_graph(s, n, e, k, i)
    res = self.algo.simple_algo(k)
    self.time1_sum = self.time1_sum + res[1]

self.time1_avg = self.time1_sum / 50.0
self.time1_x.append(n)
self.time1_y.append(self.time1_avg)
```

(b) Graph Generation for Exhausted Search Algorithm

Figure 9.1: 50 Random Graph Generation for Algorithms

```python
self.init_values1()
missing = 0
for i in range(20):
    self.algo.generate_graph(s, n, e, k, i)
    res = self.algo.my_algorithm(k)
    res2 = self.algo.simple_algo(k)
    cycles = res[0]
    cycles2 = res2[0]
    self.time1_sum = self.time1_sum + res[1]
    self.time2_sum = self.time2_sum + res2[1]
    if len(cycles) > 0 and len(cycles2) > 0:
        maxx = cycles[0]
        lenth = maxx['length']
        self.max1_sum = self.max1_sum + lenth
        missing = missing + 1
        maxx2 = cycles2[0]
        lenth2 = maxx2['length']
        self.max2_sum = self.max2_sum + lenth2

self.time1_avg = self.time1_sum / 20.0
self.max1_avg = self.max1_sum / float(missing)
self.time2_avg = self.time2_sum / 20.0
self.max2_avg = self.max2_sum / float(missing)
```

Figure 9.2: 20 Random Graph Generation for Comparison

9.3. Criterion 3

- Getting results with the heuristic algorithm in less than 1 second when number of nodes < 25 and k is not constant.

I have accomplished this criterion successfully.

- **initial vertex** = 0
- **number of nodes** = 8, 10, 12, 14, 16
- **number of edges** = $((n * (n - 1))/2) - 5$
- **k** = $n * 5$

In following images you can see the results.

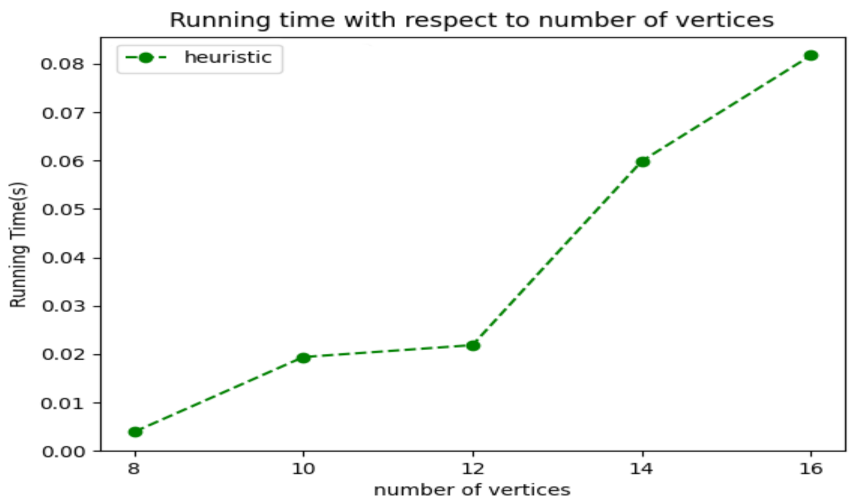


Figure 9.3: Running time with respect to number of vertices and k value

```
n = [8, 10, 12, 14, 16]
time(s) = [0.0038, 0.0193, 0.0218, 0.0599, 0.0817]
```

Figure 9.4: Exact running time results with respect to number of vertices

As you can see in results, we have less than 1 second as running time.

9.4. Criterion 4

- When k is big and proportional to the number of edges, the results of the heuristic algorithm is at least %4 better than the exhaustive search algorithm.

I have accomplished this criterion successfully. In the following images, I will show the chart, maximum length results, and difference calculation results.

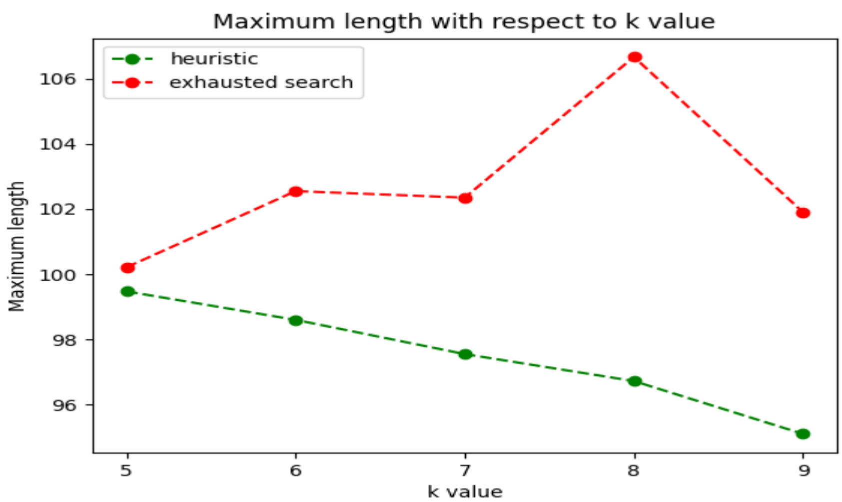


Figure 9.5: Maximum length with respect to k value

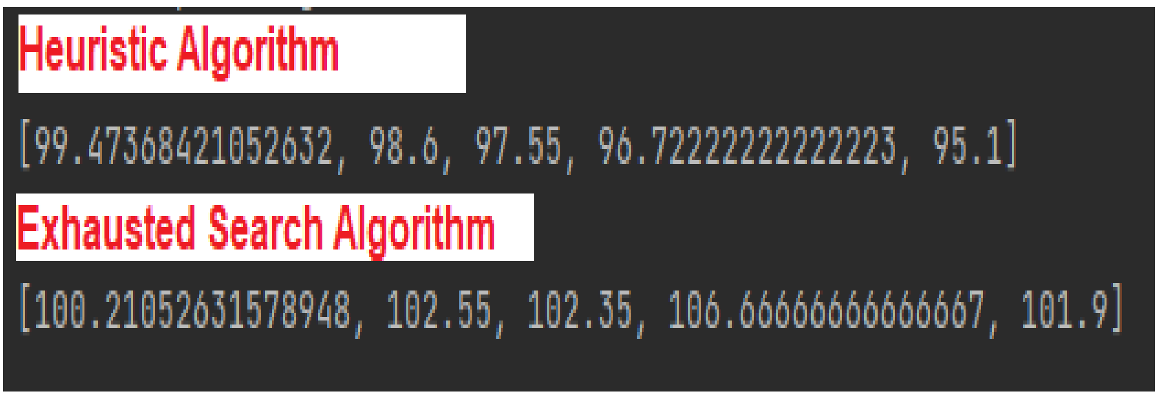


Figure 9.6: Exact Results of Maximum Lengths

$$
\begin{aligned}
&= \frac{V_{observed} - V_{true}}{V_{true}} \\
&= \frac{102.35 - 97.55}{97.55} \\
&= \frac{4.8}{97.55} \\
&= 4.9205535622758\%
\end{aligned}
$$

(a) Difference Calculation as Percentage k is 7

$$
\begin{aligned}
&= \frac{V_{observed} - V_{true}}{V_{true}} \\
&= \frac{106.66 - 96.72}{96.72} \\
&= \frac{9.94}{96.72} \\
&= 10.277088502895\%
\end{aligned}
$$

(b) Difference Calculation as Percentage k is 8

$$
\begin{aligned}
&= \frac{V_{observed} - V_{true}}{V_{true}} \\
&= \frac{101.9 - 95.1}{95.1} \\
&= \frac{6.8}{95.1} \\
&= 7.1503680336488\%
\end{aligned}
$$

(c) Difference Calculation as Percentage k is 9

Figure 9.7: Difference Calculation of Exact Results

As you can see in above images we have at least %4.90 better results in heuristic algorithm when k value is big and proportional to the number of edges.

9.5. Summary of Success Criteria Results

In the following table, we can see the all success criteria's expected and actual results. Note that for each success criterion we got the expected results and we are successful but in the heuristic algorithm **merging tours steps is open to improvement.**

Success Criterion	**Expected**	**Actual**	**Result**
Heuristic algorithm complexity will be better than $\mathcal{O}(\|E\|^4)$	$\mathcal{O}(\|E\|^4)$	$\mathcal{O}(\|E\|^3)$	Successful
Creating 50 different random graphs for each test case in performance testing and creating 20 different random graphs for each comparison case and taking the average of them.	50 and 20 different random graphs for each test case	50 and 20 different random graphs for each test case	Successful
Getting results with the heuristic algorithm in less than 1 second when number of nodes < 25 and k is not constant.	in less than 1 second	in 0.0817 second	Successful
When k is big and proportional to the number of edges, the results of the heuristic algorithm is at least %4 better than the exhaustive search algorithm.	at least %4 better	at least %4.90 better	Successful

Table 9.1: Success Summary of Success Criteria Results

10. Conclusions

In this project, we have tried to solve the Minimum k-Chinese Postman Problem and we have created 2 different algorithms. As we have seen this problem is an NP-Hard problem. Therefore, we had to take action according to this and we did mostly.

In order to solve this problem, I have applied the following steps;

1. Making literature research and understanding the problem.
2. Choosing and designing algorithms.
3. Implementing both heuristic and exhausted search algorithms.
4. Making complexity analysis of the algorithms.
5. Testing with different parameters and performance evaluation of algorithms.
6. Making comparison and numerical evaluation of algorithms.
7. Showing the comparison average results on the charts.
8. Creating a GUI and running algorithms on that GUI.

By dividing the big project into these steps I conquered each step so that steps were small to solve and test. After finishing each step I have created this project as expected.

Implementing this project is very important because finding implemented version of these projects is not easy. By making this project, we can help the people who are interested in these types of problems. After publishing this project people can see how to solve these types of problems, how to make a complexity analysis, and how to make a comparison between different algorithms. Also, they can use these solutions and improve them.

BIBLIOGRAPHY

[1] https://towardsdatascience.com/quick-guide-to-graph-traversal-analysis-1d510a5d05b5.

[2] A. Hölscher, *A cycle-trade heuristic for the weighted k-chinese postman problem*, 2018.

[3] D. B. Johnson, *Finding all the elementary circuits of a directed graph*, 1975.

[4] D. Ahr and G. Reinelt, *New heuristics and lower bounds for the min-max k-chinese postman problem*, 2002.

[5] K. Holmberg, *Heuristics for the weighted k-rural postman problem with applications to urban snow removal*, 2015.

[6] A. Y. Gregory Gutin Gabriele Muciaccia, *Parameterized complexity of k-chinese postman problem*, 2014.

[7] D. Ahr, *A tabu search algorithm for the min–max k-chinese postman problem*, 2005.

[8] https://igraph.org/.

[9] https://stackoverflow.com/questions/12367801/finding-all-cycles-in-undirected-graphs.

[10] https://www.techiedelight.com/find-distinct-combinations-of-given-length/.