Minimum k-Chinese Postman Problem

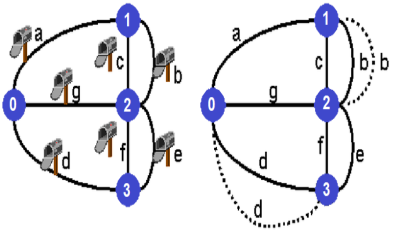
Final Presentation

Akif Kartal

Advisor: Doç. Dr. Didem GÖZÜPEK

15 June 2022

Project Definition



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

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.[1]

What we did?

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

Algorithm Steps

In order to solve this problem, I have implemented a heuristic augment-merge algorithm.[2] Steps of this algorithm are following.

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 $C = (C_1, \ldots, C_m)$ 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 $C_1, \ldots, C_k$ such that the weight of the merged tour is minimized.

Algorithm Implementation

In order to implement this algorithm, I have used python **igraf** library. [3]

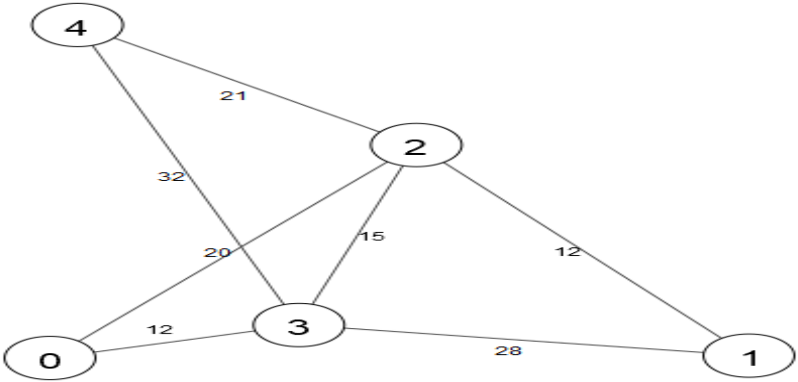
1. Generate random graph.

```
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 = []
    for x in range(0, number_of_edges):
        value = random.sample(range(0, self.__g.vcount()), 2)
        if value not in rand_edges:
            rand_edges.append(value)
```

***** We are generating a random graph by generating random edges between vertices.

Algorithm Implementation

Generated graph with 5 vertex.



* This graph includes parallel edges but it doesn't show on the drawing.

Algorithm Implementation

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

By using bubble sort;

```
def sort_edges_descending(self):
    weights = self.__my_graph.get_weights()
    edges = self.__my_graph.get_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]
    self.create_edge_dict(edges, weights)
```

```
[{'start_node': 2, 'end_node': 4, 'length': 40}, {'start_node': 2, 'end_node': 3, 'length': 32},
```

Algorithm Implementation

3. For each $e = v_i, v_j \in E$, create the closed walk, if e is not already covered by an existing tour.

```
def create_closed_walk(self, k):
    print(self.__sorted_edges)
    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.g
            # SP(vj, v1)
            path2 = self.__my_graph.get_shortest_path(path3[1], self.__

            # 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):
```

Algorithm Implementation

4. 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.

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

```python
def add_dummy_tours(self, missing_number):
    listLen = len(self.__sorted_edges)
    k = 0
    for i in range(listLen - 1):
        e = self.__sorted_edges[(listLen - i) - 1]
        walk = [e['end_node'], e['start_node'], e['end_node']]
        self.__closed_walks.append({'cycle': walk, 'length': self.get_walk_length(walk),
        k = k + 1
        if k == missing_number:
            break
```

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

```
def merge_tours(self, k):
    listLen = len(self.__closed_walks)
    n = listLen - k
    print(n)
    is_ok = False
    for i in range(n):
        listLen = len(self.__closed_walks)
        if i == 0:
            if self.merge_round2():
                print("round2")
                is_ok = True
                if listLen == k:
                    return True
            if not is_ok and self.merge_round1():
                print("round1")
                is_ok = True
                if listLen == k:
                    return True
```

If we run the algorithm with this parameters;

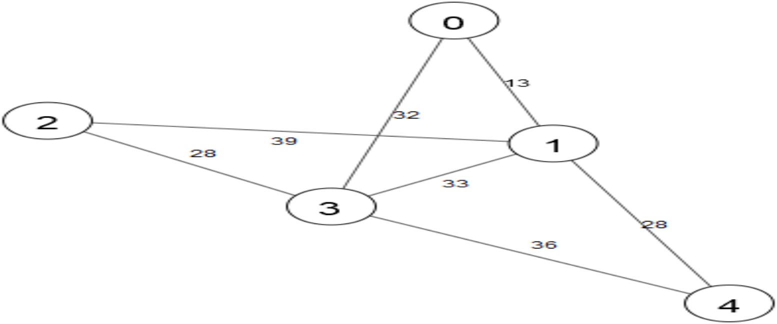
- *initial vertex* $= 0$, $k = 4$
- *number of vertices(n)* $= 5$
- *number of edges* $= n * (n - 1)/2$

```
def my_algorithm(self, s, k, n, i):
    self.generate_graph(s, n, i)
    print("graph generated")
    self.sort_edges_descending()
    print("sorted edges:")
    print(self.__sorted_edges)
    self.create_closed_walk(k)
    print("cycles:")
    print(self.__closed_walks)
```

```
alg = MyAlgorithm()
alg.my_algorithm(0, 4, 5, 5)
```

Algorithm Results

Generated random graph and k cycles.




```
[{'cycle': [0, 3, 2, 1, 0], 'length': 112, 'count': 5}, {'cycle': [0, 1, 4, 3, 0], 'length': 109, 'count': 5},
{'cycle': [0, 1, 2, 1, 0], 'length': 104, 'count': 5}, {'cycle': [0, 1, 3, 0], 'length': 78, 'count': 4}]
```

Algorithm Complexity

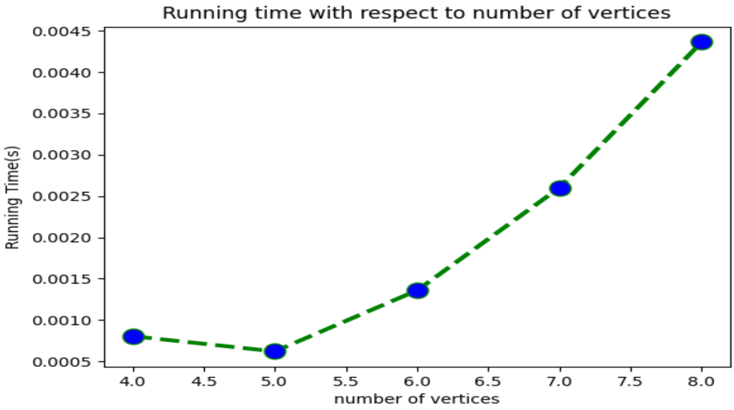
Complexity	Algorithm Step
$\mathcal{O}(n^2)$	Sort the edges e in decreasing order according to their weight.
$\mathcal{O}(n^3)$	For each $e = v_i, v_j \in E$, create the closed walk.
$\mathcal{O}(n^2)$	If number of cycle(m) $< k$ add $k - m$ “dummy” tours.
$\mathcal{O}(n^4)$	If number of cycle(m) $>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

Table: Complexity Analysis of Heuristic Algorithm Steps

Overall Complexity of Heuristic Algorithm

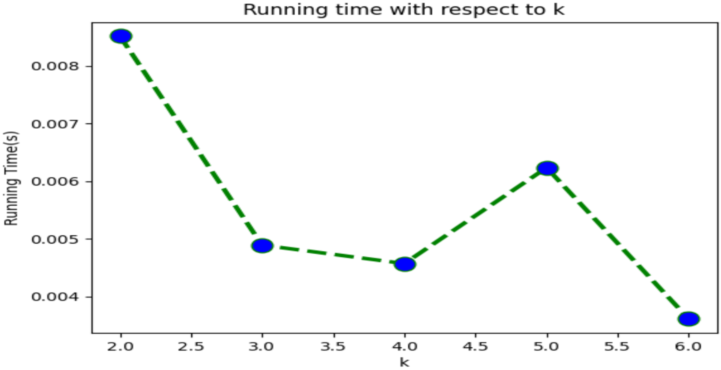
Best Case: $\mathcal{O}(n^3)$
Average Case: $\mathcal{O}(n^3)$
Worst Case: $\mathcal{O}(n^4)$

Running Time of Algorithm



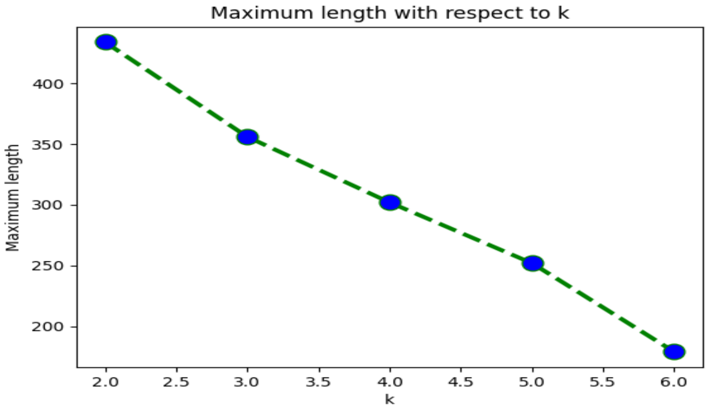
In this test, $k = 3$ and number of node changes 4 to 8. As you can see running time increase while number of vertices is increasing.

Running Time of Algorithm



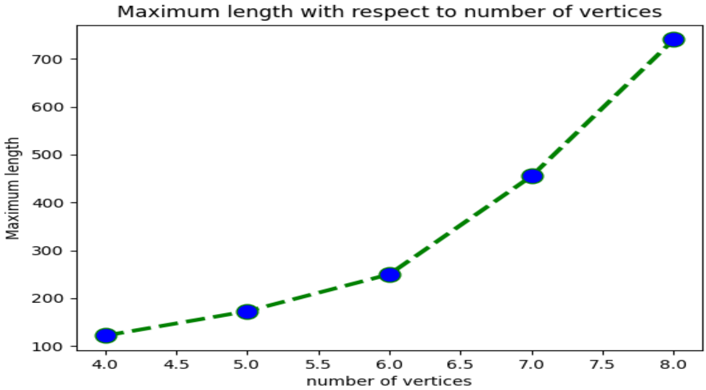
In this test, we have 7 node same graph and k changes 2 to 6. As you can see running time depend on the algorithm worst case which is If number of cycle(m) $> k$, it will take more time.

Maximum Length of The k Cycles



In this test, we have 7 node same graph and k changes 2 to 6. As you see maximum length of k cycles is decreasing while k increase.

Maximum Length of The k Cycles



In this test, $k = 3$ and number of node changes 4 to 8. As you can see maximum length of k cycles increase while number of vertices is increasing.

Exhaustive Search Algorithm

In order to make a comparison, I have implemented another algorithm. This algorithm is a simple exhaustive search algorithm such that;

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

Algorithm Complexity

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

Table: 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: $\mathcal{O}(n^n)$

Summary

As you have seen that;

- We have implemented a heuristic augment-merge algorithm.
- We made algorithm complexity analysis.
- We test the algorithm with different parameters.
- Lastly, we have started to implement simple algorithm.

Project Timeline

1^{st} Meeting	Making literature research. Understanding the problem.
2^{nd} Meeting	Continue literature research. Determine the steps of the heuristic algorithm. Start to implement heuristic algorithm.
3^{rd} Meeting	Finish heuristic algorithm implementation. Make complexity analysis of heuristic algorithm. Test with different parameters. Start to implement simple algorithm.
4^{th} Meeting	Finish simple algorithm implementation. Make complexity analysis of simple algorithm. Test with different parameters and compare. Prepare a graphical user interface.

References

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

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

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

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

[5] https://igraph.org/r/doc/distances.html.

[6] https://www.geeksforgeeks.org/graph-plotting-in-python-set-1/.

Thank You