Chinese Postman Problem: Implementation of an Optimization Algorithm based on the Operational Research techniques for Complexity Analysis

Bruno Ferreira, Gonçalo Oliveira, Jorge Correia, Nuno Castro

[a] Instituto Politécnico do Porto, Escola Superior de Tecnologia e Gestão, Rua do Curral, Casa do Curral–Margaride, 4610-156 Felgueiras, Portugal.
8200586@estg.ipp.pt, 8200595@estg.ipp.pt, 8200592@estg.ipp.pt, 8200591@estg.ipp.pt

Abstract. The abstract aims to briefly describe the most relevant work, the best results achieved and the main conclusions.

Keywords: CPP - Chinese Postman Problem

1 Introdução

Este trabalho foi realizado no âmbito da disciplina de Análise Algorítmica e Otimização, onde foi proposto o estudo e resolução do problema do carteiro chinês (PCC), circuito do carteiro ou problema da inspeção de rotas. Este problema, em teoria de grafos, consiste em encontrar um caminho mais curto ou um circuito fechado que visite cada aresta de um grafo (conectado) não-direcionado.

Este problema foi investigado pela primeira vez pelo matemático chinês Kwan Mei-Ko no início da década de 60. Ele colocou a questão, sobre qual seria a menor distância possível de um carteiro que queria percorrer todas as ruas de uma cidade, entregar cartas e voltar ao seu correio.

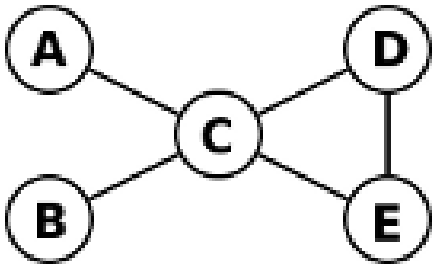
No relatório apresentado serão explicados alguns conceitos básicos de grafos, materiais e métodos utilizados e resultados experimentais com três instâncias do CPP.

Conceitos Básicos

Depois de percebido do que se trata o Problema do Carteiro Chinês, é necessário entender primeiro alguns conceitos que vão ser bastante mencionados e utilizados ao longo do desenvolvimento deste Trabalho Prático.

O que é um grafo?

Um grafo é uma representação abstrata de um conjunto de objetos e das relações existentes entre eles. É definido por um conjunto de nós ou vértices, e pelas ligações ou arestas, que ligam pares de nós. Uma grande variedade de estruturas do mundo real podem ser representadas abstratamente através de grafos.



Caminhos eulerianos e semi-eulerianos

Um caminho euleriano é um caminho num grafo que visita todas as arestas exatamente uma vez. Um Circuito Euleriano é um caminho Euleriano que começa e termina no mesmo vértice. Os grafos que possuem um circuito Euleriano são chamados Grafos Eulerianos.

Se um grafo tem dois vértices ímpares, então o grafo é semi-euleriano. Um caminho pode ser traçado começando num dos vértices ímpares e terminando noutro vértice ímpar.

Caminho euleriano no contexto do CPP

Considerando que temos um grafo que representa uma rede rodoviária de uma cidade, fazendo alusão aos caminhos que o carteiro tem que percorrer. É necessário determinar o custo do menor caminho partindo de um vértice inicial (ponto de partida do carteiro), e passar por todas as arestas (ruas) uma única vez, retornando ao ponto de origem.

Podemos concluir que se o grafo for Euleriano, qualquer circuito de Euler é um percurso ótimo do carteiro chinês, então o caminho pode ser encontrado, e consequentemente o seu respetivo custo, através do algoritmo de Fleury. O custo é dado pela soma dos custos de todas as arestas do grafo.

No entanto, se o grafo não for Euleriano, então algumas arestas terão de ser repetidas. No entanto, não deverão ser repetidas arestas, por isso, é necessário realizar alguns passos para tornar o cálculo do caminho ótimo possível.

1. Determinar os vértices de grau ímpar
2. Listar todos os pares possíveis de vértices de grau ímpar
3. Para cada par, encontrar as arestas que conectam os vértices com o peso mínimo.
4. Encontrar os pares de modo que a soma dos pesos seja minimizada.
5. No gráfico original adicionar as arestas que foram encontradas no passo 4.
6. O comprimento do caminho ótimo do carteiro chinês é a soma de todas as arestas adicionadas no total encontrado no passo 4

2 Materiais e métodos

Materiais utilizados

Python

Para o desenvolvimento do algoritmo capaz de resolver o problema do Carteiro Chinês, foi utilizada a linguagem **Python 3**.

Python é uma linguagem de programação de alto nível, interpretada por script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte.

Visual Studio Code

Como editor de código fonte utilizamos o Visual Studio Code.
Este editor foi desenvolvido pela Microsoft para Windows, Linux e macOS. Ele inclui suporte para depuração, controlo de versões Git incorporado, realce de sintaxe, complementação inteligente de código, snippets e refatoração de código.

Métodos utilizados

Modelação

Metodologia geral

Metodologia geral em Investigação Operacional passa por estudar o problema, construir um modelo, obter uma solução e validar o modelo e solução encontrados. Após a validação, inicia-se a fase de implementação.

Tendo isto em conta, começamos por analisar e estudar o problema do CPP com o objetivo de compreender a finalidade do problema. Após esse estudo detalhado do objetivo do CPP, partimos para a análise de modelos construídos onde tivemos a oportunidade de chegar a uma solução.
Tal como já referido anteriormente, após este processo de investigação e construção e validação de modelos, passamos para a fase de implementação.

Construção de um modelo

Definição dos objetivos e tipos de restrições e exploração de um software capaz de responder ao requisitado que, como foi identificado na secção de materiais utilizados.

Otimização em Redes

Otimização de redes diz respeito ao estudo e resolução de problemas que possam ser representados através de uma rede, como é o caso do CPP que representa o mapa a ser percorrido pelo carteiro.

Caminho ótimo

O objetivo do caminho ótimo é apresentar o melhor caminho a ser realizado pelo carteiro de maneira a que o custo do caminho seja o mínimo possível.

Algoritmo fleury

O algoritmo de Fleury é utilizado para a construção ou identificação de um ciclo euleriano num grafo euleriano.

Algoritmo de Dijkstra

O algoritmo de Dijkstra foi utilizado para encontrar o caminho mais curto num grafo, neste caso não direcionado, com pesos não negativos. Segundo algumas fontes, o tempo computacional deste algoritmo é de ***O(E+Vlog(V))*** ***O(E+Vlog(V))*** sendo V o número de vértices e E o número de arestas.

3 Resultados Experimentais

A complexidade do tempo é a complexidade computacional que descreve a quantidade de tempo que o computador demora para executar um algoritmo.
Então, para calcularmos a complexidade do tempo, decidimos calcular para o pior caso, que seria quando o grafo não fosse euleriano e tivesse o máximo de ligações possíveis. Para isso consideramos um grafo com número par de vértices onde existiria uma aresta entre cada dois vértices.

Para escolher o número de vértices por instância de teste seguimos a sequência seguinte, onde *n* é 2:

$$n \xrightarrow{\times 2} 2n \xrightarrow{\times 2} 4n \xrightarrow{\times 2} 8n \xrightarrow{\times 2} 16n \xrightarrow{\times 2} \dots$$

Nota: Para os seguintes exemplos de grafos utilizados supomos que o peso de todas as arestas é de 1.

Grafo 1

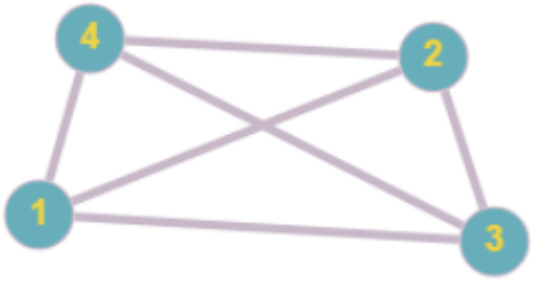
Representação:

Resultado:

```
~/chinese-postman-problem-main  time pypy3 app.py

Problema do Carteiro Chinês | Problema da Inspeção de Rotas:

Grafo 1:

   Caminho Partindo de 1: [1, 0, 1]

   Distância: 2

   Complexidade: 103

------------------------------

pypy3 app.py  0,04s user 0,03s system 82% cpu 0,093 total
```

Grafo 2

Representação:

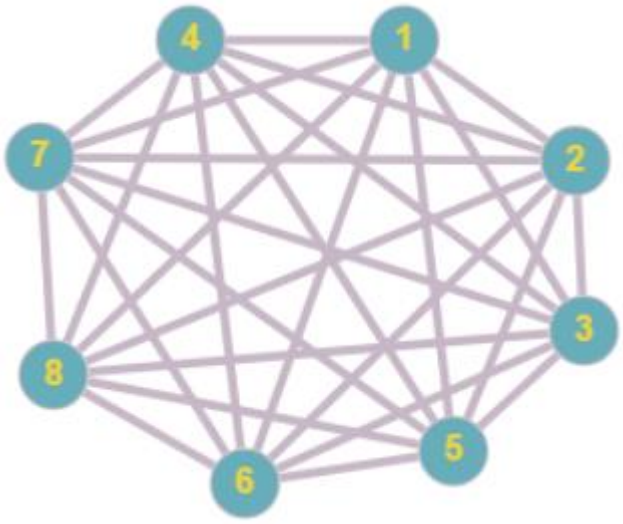


Resultado:

```
~/chinese-postman-problem-main  time pypy3 app.py

Problema do Carteiro Chinês | Problema da Inspeção de Rotas:

Grafo 1:

   Caminho Partindo de 1: [1, 0, 3, 2, 3, 1, 2, 0, 1]

   Distância: 8

   Complexidade: 1771

------------------------------

pypy3 app.py  0,04s user 0,03s system 85% cpu 0,087 total
```

Grafo 3

Representação:



Resultado:

```
~/chinese-postman-problem-main  time pypy3 app.py

Problema do Carteiro Chinês | Problema da Inspeção de Rotas:

Grafo 1:

   Caminho Partindo de 1: [1, 0, 7, 6, 7, 5, 4, 6, 5, 4, 7, 3, 2, 6, 3, 5, 2, 4,
3, 2, 7, 1, 6, 0, 5, 1, 4, 0, 3, 1, 2, 0, 1]

   Distância: 32

   Complexidade: 1244742

------------------------------

pypy3 app.py  0,23s user 0,03s system 93% cpu 0,285 total
```

4 Discussão

Estudando as complexidades obtidas para as instâncias de teste utilizadas nos resultados experimentais, é possível observar que a ordem de complexidade mais próxima é $O(n^3)$

Cubic order: $O(n^3)$

Practical behaviour: If n duplicate, the number of comparisons $comp$ is 8 times more.

Proof: $O(n^3) \rightarrow (2n)^3 = 2^3 n^3 = 8n^3$

$$comp \xrightarrow{\times 8} 8comp \xrightarrow{\times 8} 64comp \xrightarrow{\times 8} 512comp \xrightarrow{\times 8} 4096comp \xrightarrow{\times 8} \dots$$

Pontos fortes:

- Calcular o custo do caminho mais curto;
- Apresentar os caminhos

Pontos fracos:

- Otimização do código

5 Conclusão e trabalho futuro

A realização deste trabalho além de permitir o conhecimento sobre novos métodos, permitiu ainda aplicar vários conceitos lecionados nas aulas da Unidade Curricular e aperfeiçoar o conhecimento em teoria de grafos bem como em algoritmos de otimização para os mesmos, nomeadamente para o problema do carteiro chinês.

Este problema apresenta uma complexidade de tempo bastante elevada à medida que se vai aumentando o número de arestas/vértices, como foi possível observar nos resultados experimentais obtidos o que, se revela um ponto um pouco negativo, tal como referido anteriormente.

Tendo isto em conta, para trabalho futuro seria ideal obter uma melhor otimização de código para melhorar esta vertente da complexidade.

6 Agradecimentos

Aproveitamos para agradecer ao professor Carlos Pereira, docente da Unidade Curricular de Análise Algorítmica e Otimização por toda a disponibilidade e apoio fornecidos ao longo do desenvolvimento do trabalho prático.

7 Referências

1. Suffolk Maths “Chinese Postman Problem” suffolkmaths.co.uk Chinese Postman Problem.pdf (suffolkmaths.co.uk)
2. J.Pires “chinese-postman-problem” github.com. GitHub - jgspires/chinese-postman-problem: Simple Chinese Postman/Route Inspection Problem algorithm written in Python 3.x
3. FEUP “Algoritmos em Grafos: Circuitos de Euler e Problema do Carteiro Chinês” paginas.fe.up.pt - https://paginas.fe.up.pt/~rossetti/rrwiki/lib/exe/fetch.php?media=teaching:1011:cal:08_2.09_1.grafos6.pdf

8 Anexos

8.1 Anexo I - Código com o algoritmo utilizado

Para a estruturação do código foi criada uma pasta “Entities” que contém os ficheiros necessários para manipulação de grafos (*Graph.py* e *Edge.py*), bem como um ficheiro *app.py* onde devem ser criados grafos e o programa possa ser executado.

```
from entities.Graph import Graph
def createGraphCompleted(size=1):
""" Gera um grafo completo com um determinado tamanho """
    g = Graph(size)
    for i in range(size):
        for j in range(i+1, size):
            g.add_edge(i, j, 1)
    return g
def generateGraphs():
    """ Gera 3 grafos completos """
    graphs = []
    # Grafo 1:
    graphs.append(createGraphCompleted(2))

    # Grafo 2:
    graphs.append(createGraphCompleted(4))

    # Grafo 3:
    graphs.append(createGraphCompleted(8))
    return graphs
```

App.py

```python
def main():
    graphs = generateGraphs()
    initialVertex = 1

    print("\nProblema do Carteiro Chinês | Problema da
Inspeção de Rotas:\n")
    for i, graph in enumerate(graphs):
        print(f"Grafo {i+1}:\n")
        postman = graph.chinese_postman(initialVertex)
        print(f"   Caminho Partindo de {initialVertex}:
{postman['path']}")
        print(f"\n   Distância: {postman['distance']}")
        print(f"\n   Complexidade:
{postman['complexity']}")
        print("\n------------------------------\n")


if __name__ == "__main__":
    main()
```

app.py

```python
class Edge:
    def __init__(self, x, y, weight):
        """Instancia uma nova aresta, onde 'x' e 'y' são
os índices dos vértices aos quais a aresta se conecta e
'weight' é o peso."""
        self.vertices = [x, y]
        self.weight = weight
```

Edge,py

```python
from functools import total_ordering
from queue import PriorityQueue
from entities.Edge import Edge
import copy


class Graph:
    def __init__(self, v_num):
        """Instancia um novo grafo, onde 'v_num' é o
número de vértices."""
        self.counter = 0
        self.v_num = v_num
        self.edges = []
        self.possible_combinations = []

    def _increaseComplexity(self, c=1):
        """Incrementar a variavel c ao counter da
complexidade"""
        self.counter += c

    def _getComplexity(self):
        """Retorna a variável counter que contem a
complexidade"""
        return self.counter

    def add_edge(self, x, y, weight):
        """Adiciona uma aresta ao grafo, onde 'x' e 'y'
são os vértices aos quais ela é ligada e 'weight' é o
peso."""
        if x < self.v_num and y < self.v_num:
            self._increaseComplexity(2)
            self.edges.append(Edge(x, y, weight))
        else:
            self._increaseComplexity(3)
            raise IndexError(self.edges)
```

Graph.py

```
def dijkstra(self, v):
        """Executa o algoritmo de dijkstra a partir do vértice
'v'. Utiliza uma Priority Queue para facilitar processamento.
        Retorna uma lista com as menores distâncias entre 'v' e os
outros vértices do grafo."""
        dijkstra_dist = {vertex: float("inf") for vertex in
range(self.v_num)}
        self._increaseComplexity(self.v_num)
        dijkstra_dist[v] = 0
        visited = []
        pq = PriorityQueue()
        pq.put((0, v))
        self._increaseComplexity()
        while not pq.empty():
            self._increaseComplexity()
            (dist, current_vertex) = pq.get()
            visited.append(current_vertex)
            for neighbor in range(self.v_num):
                self._increaseComplexity()
                distance = self.get_edge_weight(current_vertex,
neighbor)
                if distance != 0:
                    if neighbor not in visited:
                        old_cost = dijkstra_dist[neighbor]
                        new_cost = dijkstra_dist[current_vertex] +
distance
                        if new_cost < old_cost:
                            pq.put((new_cost, neighbor))
                            self._increaseComplexity()
                            dijkstra_dist[neighbor] = new_cost
                        self._increaseComplexity()
                    self._increaseComplexity()
                self._increaseComplexity()
            self._increaseComplexity()
        self._increaseComplexity()
        return dijkstra_dist
```

Graph.py

```
def get_edge_weight(self, x, y):
        """Retorna o peso da aresta entre os vértices 'x'
e 'y'. Ou 0 se não existirem arestas entre eles."""
        for i in range(len(self.edges)):
            self._increaseComplexity()
            if x in self.edges[i].vertices and y in
self.edges[i].vertices:
                return self.edges[i].weight
            self._increaseComplexity(2)
        self._increaseComplexity()
        return 0

    def get_odd_vertices(self):
        """Retorna uma lista com todos os índices dos
vértices de grau ímpar presentes no grafo."""
        odd_vertices = []
        for i in range(self.v_num):
            self._increaseComplexity()
            degree = self.get_vertex_degree(i)
            if degree % 2 != 0:
                odd_vertices.append(i)
            self._increaseComplexity()
        self._increaseComplexity()
        return odd_vertices

    def get_vertex_degree(self, v_index):
        """Retorna o grau do vértice de índice
'v_index'."""
        degree = 0
        for i in range(len(self.edges)):
            self._increaseComplexity()
            if v_index in self.edges[i].vertices:
                degree += 1
            self._increaseComplexity()
        self._increaseComplexity()
        return degree
```

Graph.py

```
    def get_possible_combinations(self, pairs, done, final):
        """Retorna uma lista com todas as combinações de pares
possíveis."""
        if pairs == self.get_odd_pairs():
            self.l = (len(pairs) + 1) // 2
        self._increaseComplexity()

        if pairs[0][0][0] not in done:
            done.append(pairs[0][0][0])

            for i in pairs[0]:
                self._increaseComplexity()
                f = final[:]
                val = done[:]
                if i[1] not in val:
                    f.append(i)
                else:
                    self._increaseComplexity(2)
                    continue
                self._increaseComplexity()

                if len(f) == self.l:
                    self.possible_combinations.append(f)
                    return
                else:
                    self._increaseComplexity()
                    val.append(i[1])
                    self.get_possible_combinations(pairs[1:], val, f)
                self._increaseComplexity()
            self._increaseComplexity()
        else:
            self._increaseComplexity()
            self.get_possible_combinations(pairs[1:], done, final)
        self._increaseComplexity()
```

Graph.py

```
def get_odd_pairs(self):
        """Retorna uma lista com todos os pares de
vértices de grau ímpar presentes no grafo."""
        pairs = []
        odd_vertices = self.get_odd_vertices()
        for v_index in range(len(odd_vertices) - 1):
            self._increaseComplexity()
            pairs.append([])
            for i in range(v_index + 1,
len(odd_vertices)):
                self._increaseComplexity()

pairs[v_index].append([odd_vertices[v_index],
odd_vertices[i]])
            self._increaseComplexity()
        self._increaseComplexity()
        return pairs

def chinese_postman(self, start_index):
        """Retorna a menor distância a ser percorrida
para atravessar todas as arestas considerando um
circuito."""

        odd_vertices = self.get_odd_vertices()

        if len(odd_vertices) != 0:

self.eulerify(self.get_shortest_path_distance()["combinat
ion"])
        self._increaseComplexity()
        path = self.get_eulerian_path(start_index)
        distance = self.get_sum_of_edge_weights()
        return {"path": path, "distance": distance,
"complexity": self._getComplexity()}
```

Graph.py

```
def get_eulerian_path(self, start_index):
        """Retorna os vértices de um circuito percorrido
no grafo (que precisa estar eulerizado), partindo do
vértice de índice "start_index"."""
        graph = copy.deepcopy(self)
        length = len(self.edges) +
len(self.possible_combinations)
        self._increaseComplexity(length)
        stack = []
        eulerian_path = []
        current_vertex = start_index
        while len(graph.get_neighbours(current_vertex)) >
0 or len(stack) > 0:
            self._increaseComplexity()
            current_neighbours =
graph.get_neighbours(current_vertex)
            if len(current_neighbours) > 0:
                stack.append(current_vertex)
                neighbour = current_neighbours[0]
                graph.remove_edge(current_vertex,
neighbour)
                current_vertex = neighbour
            else:
                self._increaseComplexity()
                eulerian_path.append(current_vertex)
                current_vertex = stack.pop()
            self._increaseComplexity()
        self._increaseComplexity()
        eulerian_path.append(start_index)
        return eulerian_path
```

Graph.py

```
      def get_neighbours(self, v):
        """Retorna o índice dos vizinhos do vértice "v"."""
        neighbours = []
        for edge in self.edges:
            self._increaseComplexity()
            if edge.vertices[0] == v or edge.vertices[1] == v:
                if edge.vertices[0] != v and edge.vertices[0] not in neighbours:
                    neighbours.append(edge.vertices[0])
                elif edge.vertices[1] not in neighbours:
                    self._increaseComplexity()
                    neighbours.append(edge.vertices[1])
                self._increaseComplexity(2)
            self._increaseComplexity()
        self._increaseComplexity()
        return neighbours
    def get_shortest_path_distance(self):
        """Retorna um dictionary contendo a menor distância e a combinação de
vértices ímpares que nela resultou."""
        self.possible_combinations = []
        self.get_possible_combinations(self.get_odd_pairs(), [], [])
        combinations = self.possible_combinations
        chosen_combo = None
        shortest_distance = None
        for combo in combinations:
            self._increaseComplexity()
            total_distance = 0
            for pair in combo:
                self._increaseComplexity()
                dijkstra_distances = self.dijkstra(pair[0])
                total_distance += dijkstra_distances[pair[1]]
            self._increaseComplexity()
            if shortest_distance is None or shortest_distance > total_distance:
                shortest_distance = total_distance
                chosen_combo = combo
            self._increaseComplexity()
        self._increaseComplexity()
        return {"combination": chosen_combo, "distance": shortest_distance}
```

Graph.py

```
def eulerify(self, combination):
        """Euleriza o grafo, duplicando as arestas entre os vértices
especificados na lista de pares de vértices ímpares 'combination'."""
        for pair in combination:
            self._increaseComplexity()
            for edge in self.edges:
                self._increaseComplexity()
                if edge.vertices[0] == pair[0] and edge.vertices[1]
== pair[1]:
                    self._increaseComplexity(2)
                    weight = edge.weight
                    self.add_edge(pair[0], pair[1], weight)
                    break
                self._increaseComplexity(2)
            self._increaseComplexity()
        self._increaseComplexity()

    def get_sum_of_edge_weights(self):
        """Retorna a soma de todos os pesos das arestas presentes no
grafo."""
        total_weight = 0
        for edge in self.edges:
            self._increaseComplexity()
            total_weight += edge.weight
        self._increaseComplexity()
        return total_weight
```

Graph.py