TRABALHO FINAL: CARTEIRO CHINÊS

Sèïvè Romaric Mariano Monnou
Francisco Otávio Fleury Teixeira Pinto

Junho 2019

1 Problema

1.1 Descrição

Em várias circunstâncias do mundo real, deseja-se achar um trajeto que passa por todas as vias de uma localidade com o menor custo. Além disso, em situações onde não é possível encontrar tal trajeto, deseja-se saber como transformar a localidade de modo a permitir que tal trajeto exista. Esta questão pode ser modelada como um problema de Teoria dos Grafos utilizando o conceito de ciclo Euleriano.

1.2 Modelagem

Em Teoria dos Grafos, o problema do Carteiro Chinês (PCC) consiste em achar o caminho de menor custo que visita cada aresta de um grafo G = (V, E). No nosso contexto, as arestas ($e \in E$) constituem as vias cujos pontos ligados são vértices ($v \in V$).

1.3 Casos especiais

Caso exista um ciclo euleriano no grafo, então este é a solução ótima para o PCC, já que o circuito euleriano visita cada aresta exatamente uma vez.

1.4 Propriedades

- **Grafo não orientado:** Existe uma aresta (u, v) se e somente existe a aresta (v, u).
- **Grafo Conexo:** Existe caminho entre qualquer par de vértices.

- **Número par de vértices de grau ímpar:** Todo grafo conexo admite uma quantidade par de vértices de grau ímpar.

Caso o grafo não seja euleriano, então existe um número par de vértices de graú impar. A estrategia neste caso é achar o emparelhamento perfeito

1.5 Relação com outros problemas

- **Problema do varredor de New York Street:** Achar o transversal mínimo de um digrafo.
- **Problema do carteiro k-Chinês:** Minimizar o custo do ciclo mais caro de todos os ciclos de k elementos a partir de um local designado de tal forma que cada aresta seja atravessada por pelo menos um ciclo.
- **Problema do carteiro rural:** Dado um subconjunto de arestas, encontrar o mais barato ciclo hamiltoniano contendo cada uma dessas arestas. A única diferênça do PCC é que estamos especifícando quais vértices o ciclo deve conter.

2 Algoritmo

2.1 Descrição

Para simplificar a implementação, focamos somente em grafos não orientados. O algoritmo proposto pode ser resumido nos seguintes passos.

- Se o grafo for euleriano, retorna um circuito euleriano como solução ótima.

- Senão:
  - Procura os vértices de grau um e duplica suas arestas.
  - Procura os vértices de grau ímpar do novo grafo e monta as combinações possíveis desses vértices dois a dois.
  - Encontra usando o algoritmo de Dijkstra todos os caminhos e pesos das combinações montadas no passo anterior.

– Constrói o subgrafo dos vértices ímpares e arestas entre para todas as combinações.
– Encontra o emparelhamento perfeito de menor custo neste subgrafo.
– Duplica as arestas, no grafo original, dos caminhos do emparelhamento escolhido, gerando assim um grafo euleriano.
– Por fim, encontra o caminho euleriano deste novo grafo.

2.2 Corretude

Seja G um grafo no qual todas as arestas possuem um peso positivo. Então o algoritmo descrito encontra a solução de custo mínimo para o PCC.

Demonstração: Como sabemos que se G for euleriano o caminho euleriano é a solução ótima, basta mostrarmos que a extensão G^* que será construída é euleriana de peso mínimo de G.
Seja $S = \{x_1, x_2, ..., x_{2m}\}$ o conjunto de vértices de grau ímpar em G. Seja G^*_{min} uma extensão euleriana de peso mínimo de G e seja $N = G^* - E(G)$. Para cada vértice v de G, temos que $d_N(v) = d_{G^*_{min}}(v) - d_N(v)$. Como G^*_{min} é euleriano, todos os vértices de G^*_{min} tem grau par. Então $d_N(v) \equiv d_G(v) \mod(2)$ para todo $v \in V(G)$. Então o conjunto de vértices de grau ímpar de N é novamente S.
N tem um conjunto P de m caminhos de arestas disjuntas tal que cada vértice em S é o final de exatamente um caminho de P. Seja G' obtido de G ao dobrar as arestas ao longo de cada caminho de P. Então G' não possui vértices de grau ímpar. Dessa forma, G' é uma extensão de G que está contida em G^*_{min}. Pela minimalidade de G^*_{min}, temos que $G^*_{min} = G'$.
O processo que acabamos de descrever corresponde à escolha de um emparelhamento perfeito de custo mínimo no subgrafo construído a partir do algoritmo descrito na subseção acima.

2.3 Estruturas

Os grafos em formato de matriz de adjacência armazenados nos arquivos de entradas, são convertidos no seu torno em lista de adjacência pela função **listAdjacencia** que recebe como parâmentro a matriz no formato mencionado. Em seguida, uma verificação de

se o grafo é euleriano é feita pela função **verificaGrafoEuleriano**. Caso haja, achamos o circuito euleriano. Para isso, optamos usar o algoritmo de **Hierholzer**.

2.4 Algoritmo de Hierholzer

O método basicamente funciona achando vários ciclos. Começamos por um vértice qualquer e percoremos os grafo até voltar nele. Logo após, escolhemos um vertice pertencendo ao circuito achado e buscamos um novo circuito mas desta vez no grafos restante(**grafoRestante**). As arestas a serem dos novos circutos incluidos são apagados do grafo Restante pela rotina **apagarAresta**. Os novos circuitos são incluidos no antigo pela função **incluirNovociclo** Repetimos esse processo até o grafo Restante não tenha mais arrestas. Finalmente é obtido um circuito euleriano retornado pelo método principal **acharEuler-Ciclo**

2.5 Não existência de solução trivial

Caso o grafo não seja euleriano, será tornado euleriano. Para isso, utilizamos as seguintes sub-rotinas:

- **acharFinsDeLinha**: Recebe como parâmetro a lista de adjacência e encontra os vértices que só possuem uma aresta e retorna estas arestas.
- **adicionaAresta**: Adiciona uma aresta a um par de vértices.
- **retornaImpares**: Retorna uma lista dos vértices de grau ímpar do grafo.
- **constroiDuplas**: Constrói todas as duplas possíveis de vértices de grau ímpar.
- **achaCustoTotal**: Acha o custo total do grafo.
- **opcoesDeAresta**: Retorna um dicionário com as arestas disponíveis para o vértice.
- **resumeCaminho**: Resume uma cadeia de vértices anteriores e retorna um caminho.
- **achaCusto**: Retorna o custo mínimo e uma rota do vértice inicial para o final. Aqui, usamos o algoritmo de Dijkstra.

- **achaSolucoesParaDuplas**: Retorna caminho e custo para todas as duplas.
- **duplasUnicas**: Gera conjuntos de pares unicos de vertices de grau ímpar.
- **achaMinimo**: Retorna o menor custo e caminho para todos os conjuntos de duplas.
- **acharArestaCaminho**: Retorna as arestas que serão duplicadas para fazer o grafo virar euleriano.

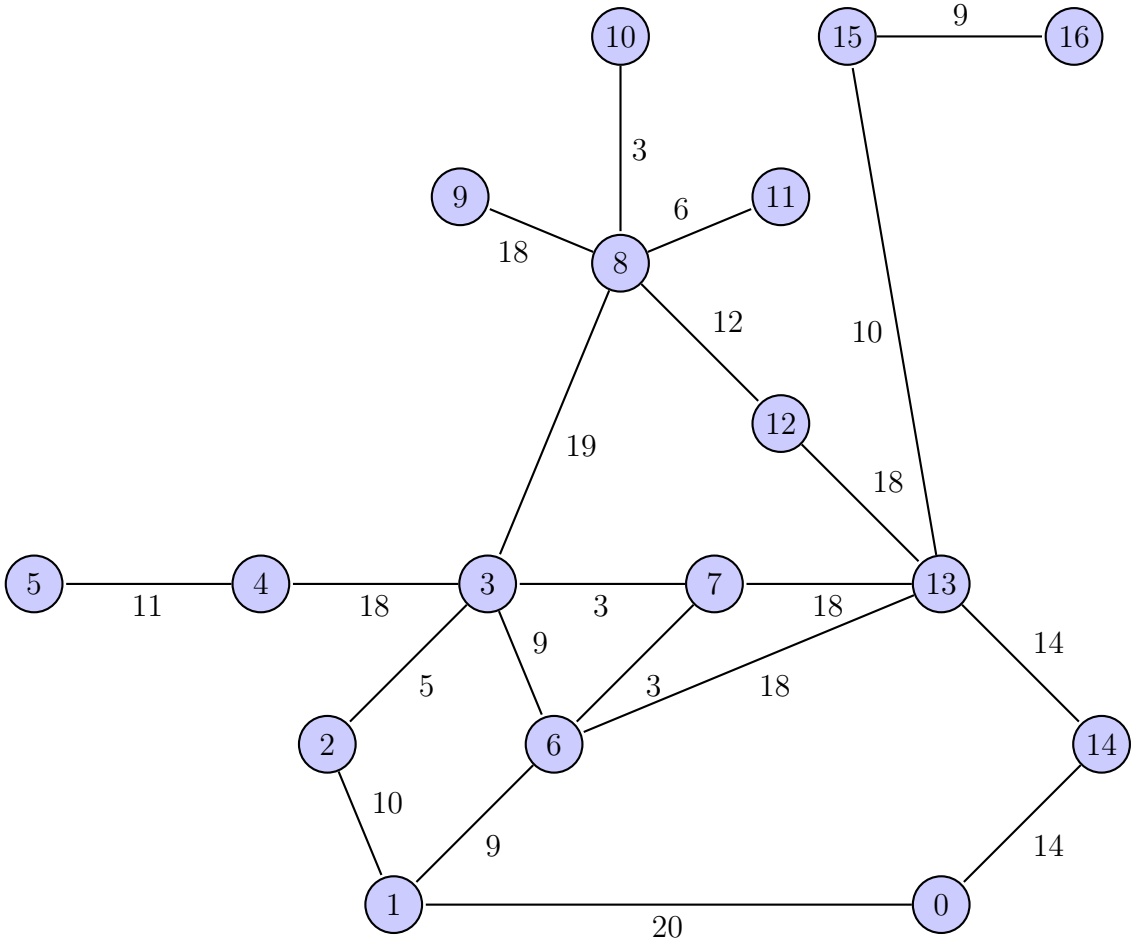
2.6 Complexidade dos principais algoritmos utilizados

- **achaCusto:** Algoritmo de Dijkstra, complexidade $O(|E| + |V| \log |V|)$
- **acharEulerCiclo:** Algoritmo de Hierholzer, complexidade $O(|E|)$
- **duplasUnicas:** Algoritmo para encontrar as $\binom{n}{2}$ combinações de vértices de grau ímpar com n sendo o número de vértices de grau ímpar do grafo, complexidade $O(n!)$.
- **achaMinimo:** Algoritmo que encontra o emparelhamento perfeito de custo mínimo do subgrafo de vértices de grau ímpar. A complexidade desta função é $O(n^2)$, iterando sobre n emparelhamentos possíveis e somando em cada um o custo dos caminhos, para encontrar o de menor custo.

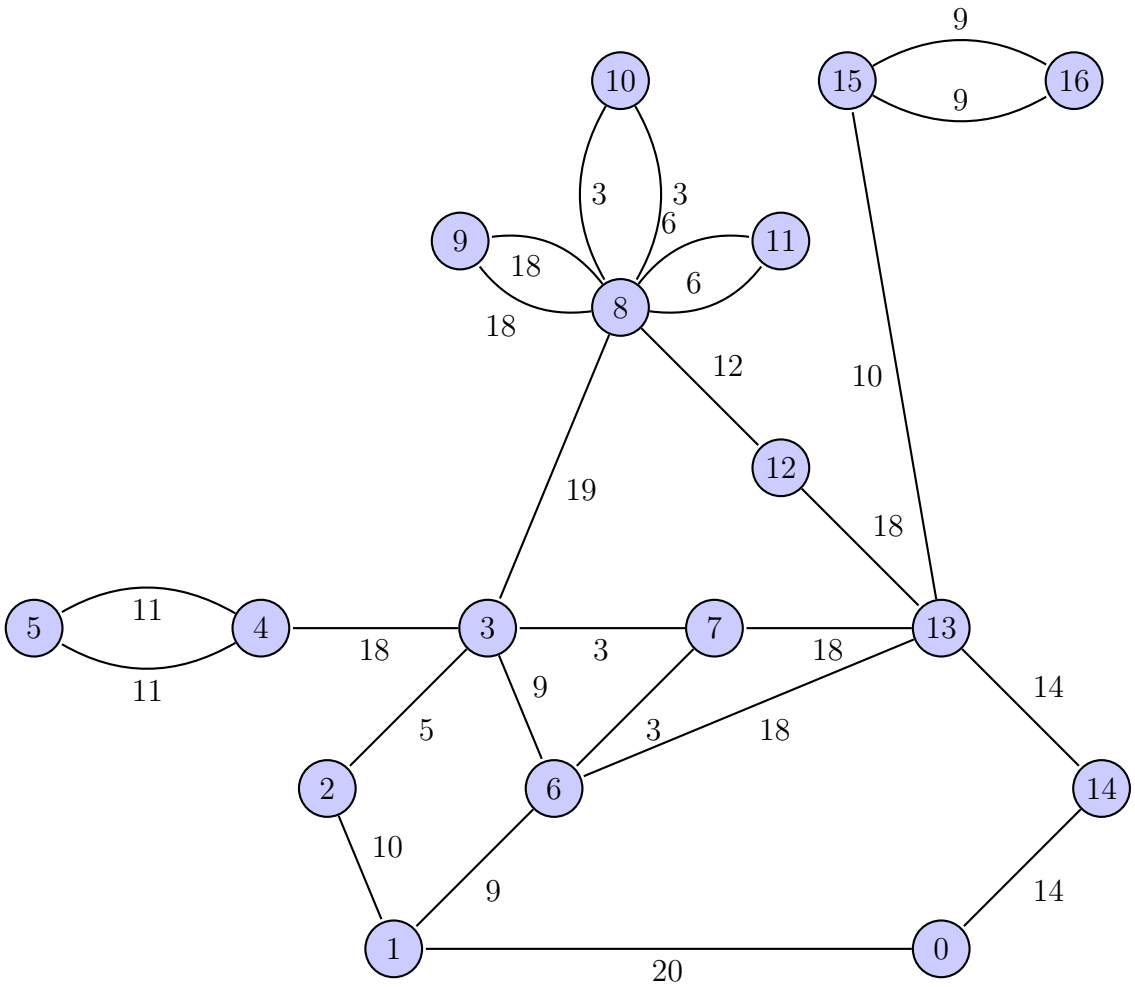
Como **duplasUnicas** domina todas as outras complexidades do algoritmo, a complexidade da solução será $O(n!)$. Ou seja, caso haja um número muito grande de vértices de grau ímpar, o tempo de execução será fatorialmente proporcional, levando um tempo muito grande para processar as combinações. Isso é ilustrado na seção de gráficos de tempo de execução mais abaixo, onde testamos o tempo de execução do algoritmo para grafos completos com número de vértices par.

2.7 Exemplo ilustrado passo a passo do algoritmo

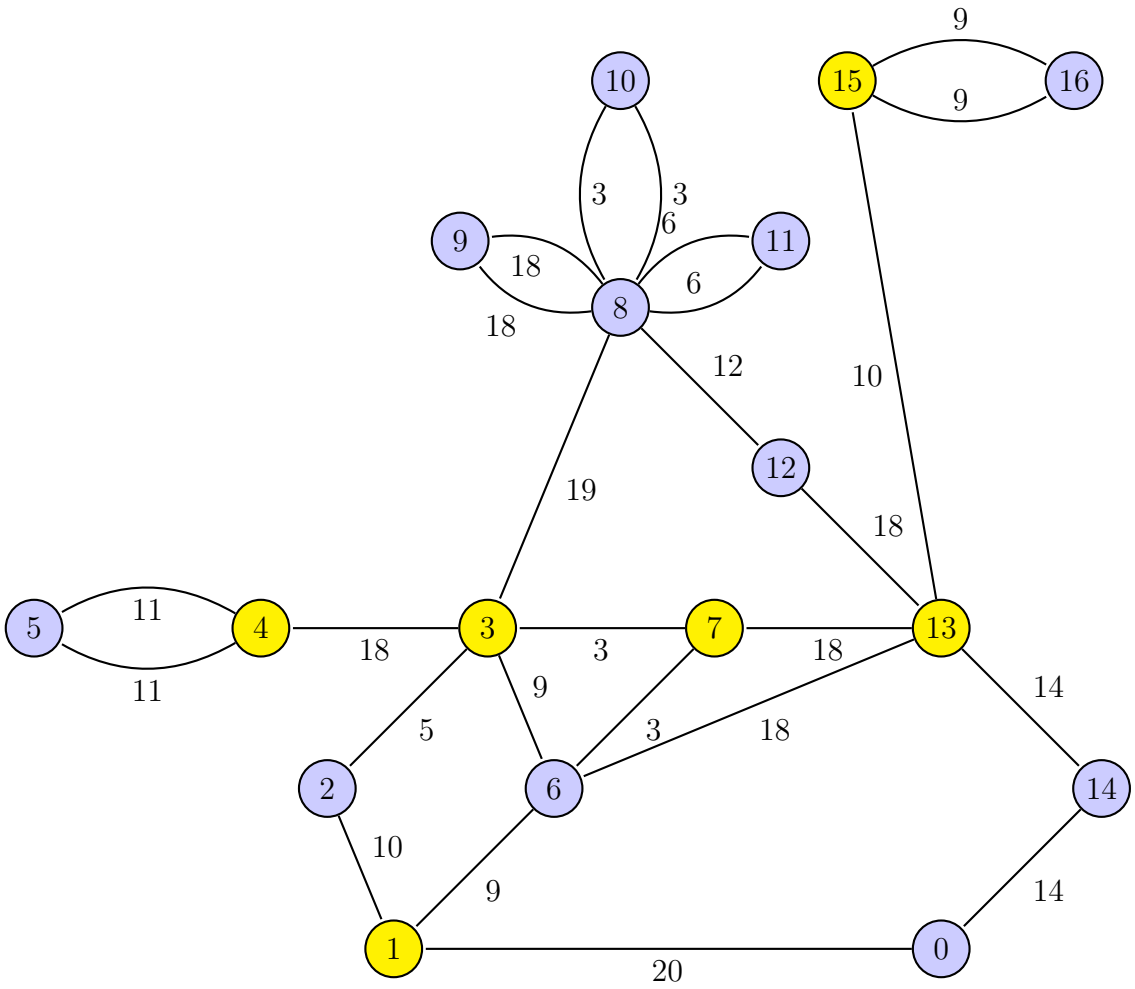
Considere o seguinte grafo:



Passo 1: Primeiramente procuramos por todos os vértices de grau 1 e dobramos as suas respectivas arestas.



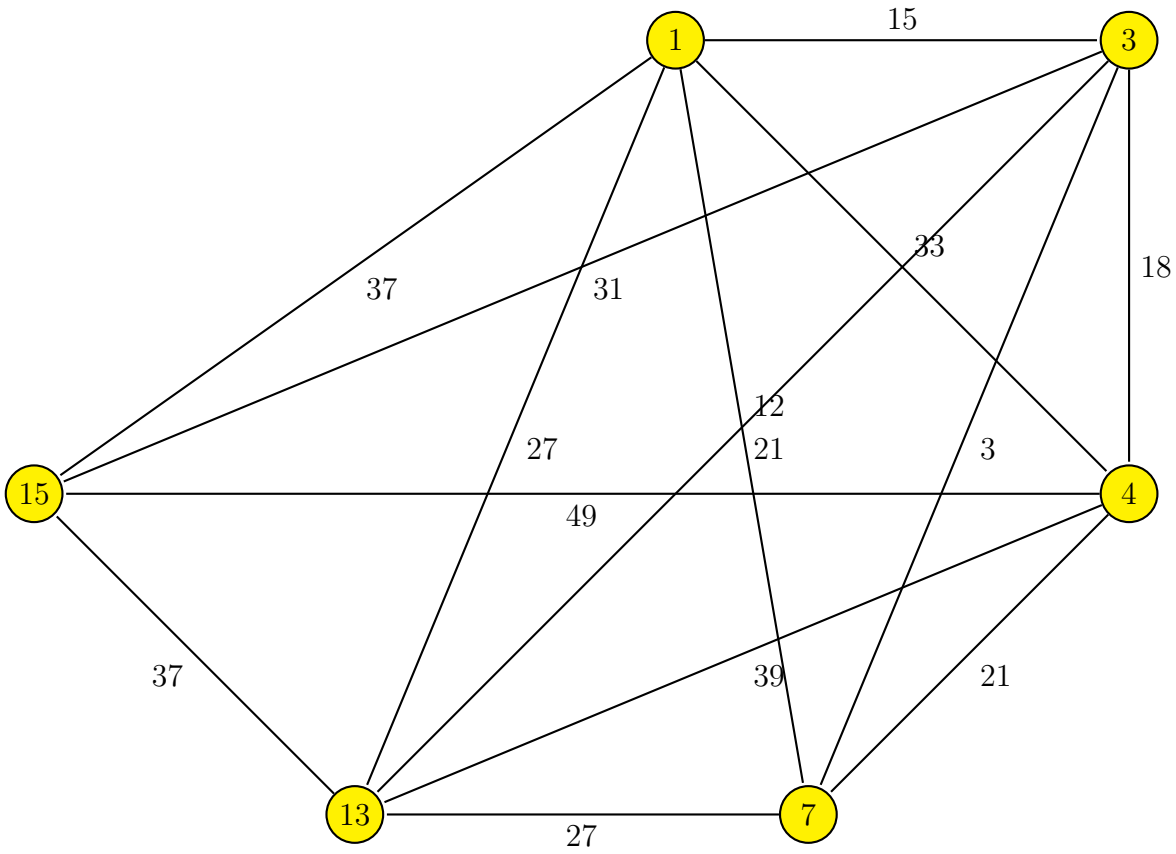
Passo 2: Com a duplicação das arestas dos vértices de grau 1, os adjacentes a estes agora tem grau ímpar. Agora buscamos por todos os vértices de grau ímpar.



Passo 3: Computamos o caminho de menor custos para os vértices de grau ímpar 1, 3, 4, 7, 13, 15, fazendo entre todos os vértices com exatemenente 6^2 iterações do algoritmo de Dikjstra.

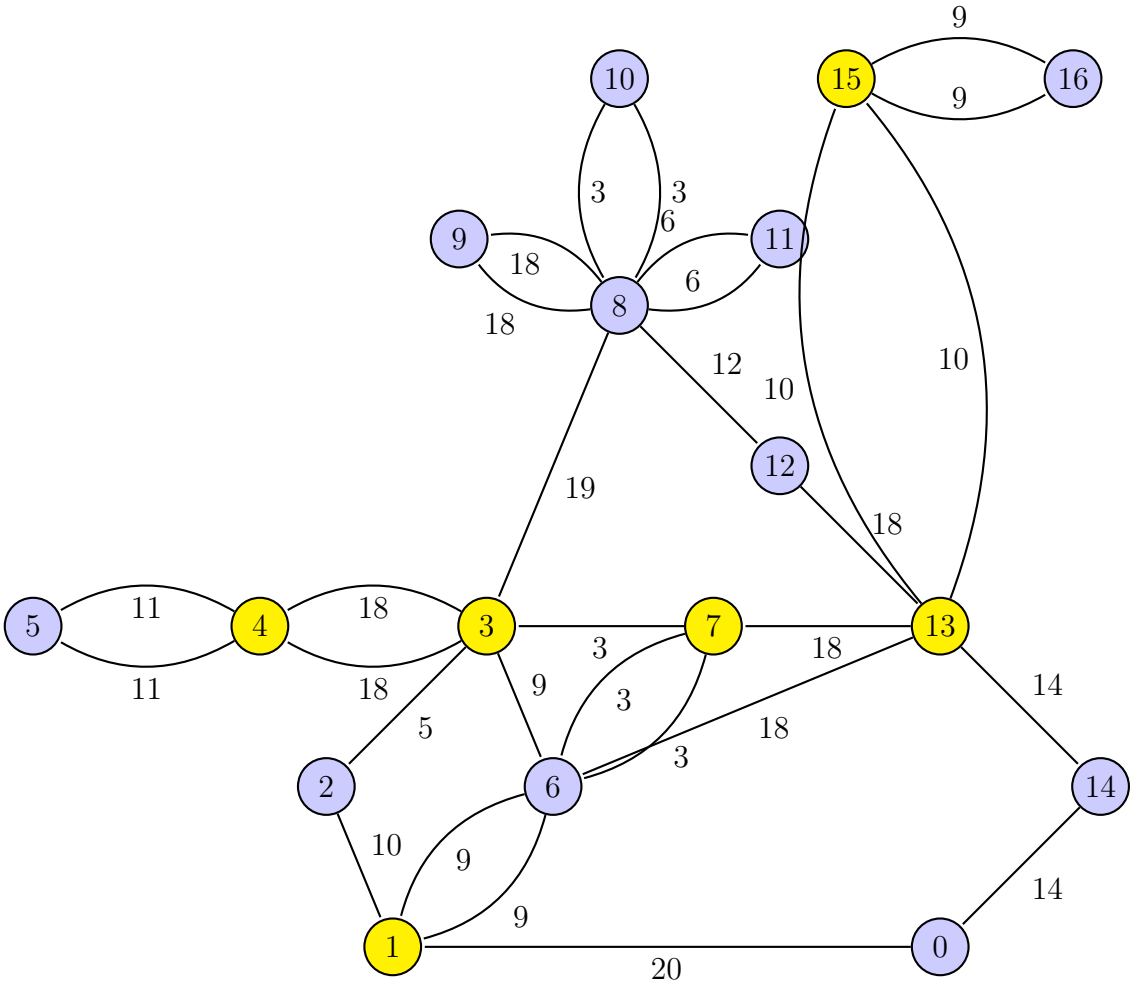
Pare Possíveis	Caminho de custo mínimo	Custo
1 <—>3	1, 2, 3	15
1 <—>4	1, 2, 3, 4	33
1 <—>7	1, 6, 7,	12
1 <—>13	1, 6, 13	27
1 <—>15	1, 6, 13, 15	37
3 <—>4	3, 4	18
3 <—>7	3, 7	3
3 <—>13	3, 7, 13	21
3 <—>15	3, 7, 13, 15	31
4 <—>7	4, 3, 7	21
4 <—>13	4, 3, 7, 13	39
4 <—>15	4, 3, 7, 13, 15	49
7 <—>13	7, 13	18
7 <—>15	7, 13, 15	28
13 <—>15	13, 15	10

Passo 4: Uma vez que sabemos o caminho mínimo entre todos os vértices de grau ímpar, montamos o grafo entre todos os pares desses vértices. O peso de uma aresta entre dois destes vértices é o custo do caminho mínimo de um para outro.



Nesse nível, para achar o emparelhamento perfeito de custo mínimo o algoritmo computa todas as partições possíveis em duplas do conjunto de vértices de grau ímpar. Sendo assim, o emparelhamento perfeito de custo mínimo corresponde à partição de menor custo que cobre todo o conjunto. Por exemplo, a partição de menor custo encontrada foi: (1, 7), (3, 4), (13, 15) de custo 12 + 18 + 37 = 57.

Passo 5: Feito isso dobramos as arestas dos caminhos de menor custo: $\{1, 6, 7\}, \{3, 4\}, \{13, 15\}$ de 1 a 7, 3 a 4, e 13 a 15 respectivamente. Assim grafo se torna euleriano e podemos então aplicar o algoritmo de Hierhozer para achar o circuito euleriano.



Ao rodar o algoritmo de Hierholzer no grafo final, obtemos o circuito euleriano: {0, 1, 2, 3, 4, 5, 4, 3, 6, 1, 6, 7, 3, 8, 9, 8, 10, 8, 11, 8, 12, 13, 15, 16, 15, 13, 6, 7, 13, 14, 0}

2.8 Decisões de implementação

Foi necessário definir o grafo como lista de adjacencia, além disso, em cada item da lista guardamos também o vértice origem para facilitar na identificação.
Optamos por utilizar o algoritmo de Hierholzer em favor do de Fleury, pelo fato do anterior ser mais eficiente.

3 Resultados

Testes gerados aleatoriamente para vértice de tamanho entre 1 e 50

Número de vértices	Número de arestas	Tempo de execução
6	15	0.00232
11	55	0.00155
15	103	0.00582
17	136	0.00389
19	71	0.00504
21	209	0.00926
23	251	0.01434
29	405	0.02501
31	462	0.03305
33	522	0.23292
37	661	0.24259
43	896	0.07766
45	985	0.26248

3.1 Exemplos de grafos

Figure 1: Grafo de Halin, 21 vértices ímpares, não foi possível achar a solução

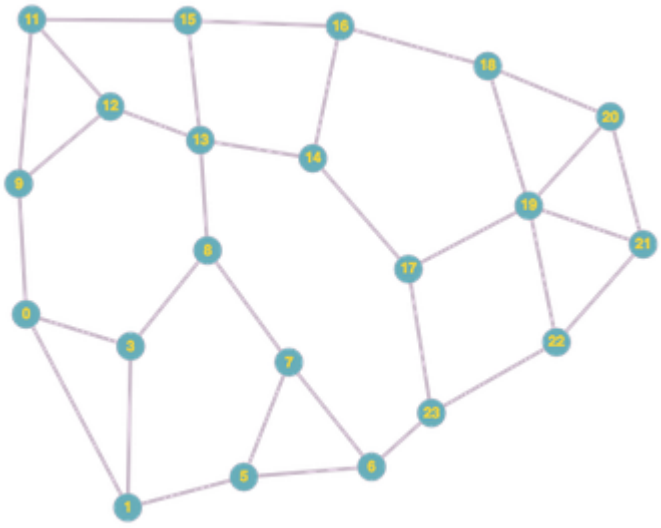


Figure 2: Grafo bipartido, dois vértices ímpares, tempo de execução 0.00399 seg

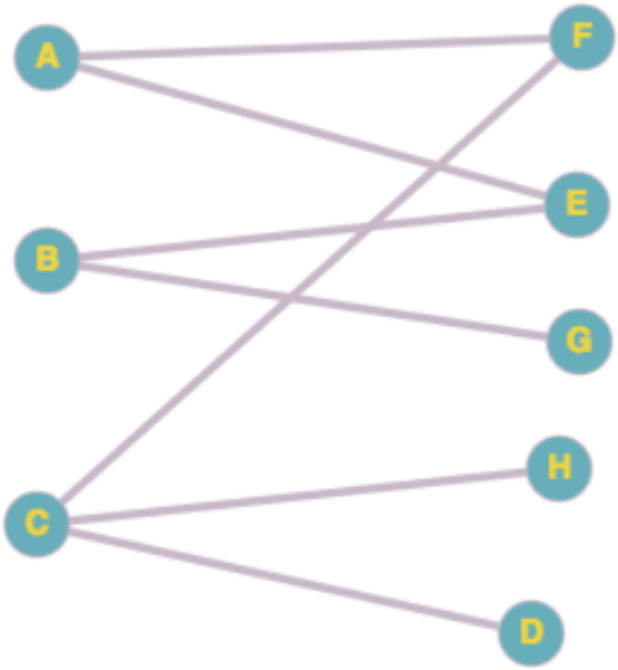


Figure 3: Grafo com pesos, quatro vértices ímpares, tempo de execução 0.00601 seg

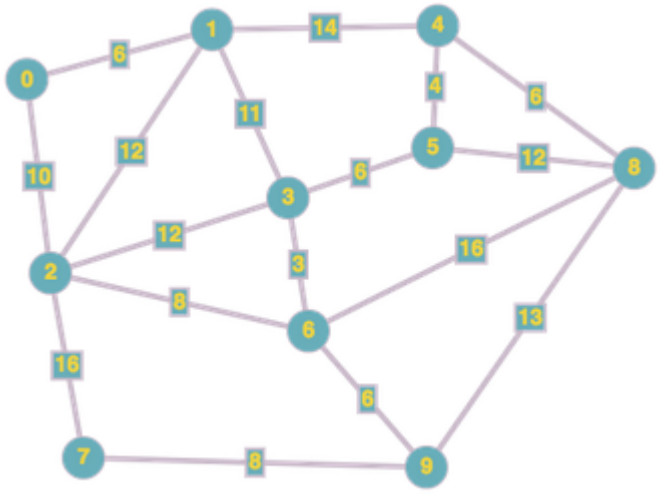
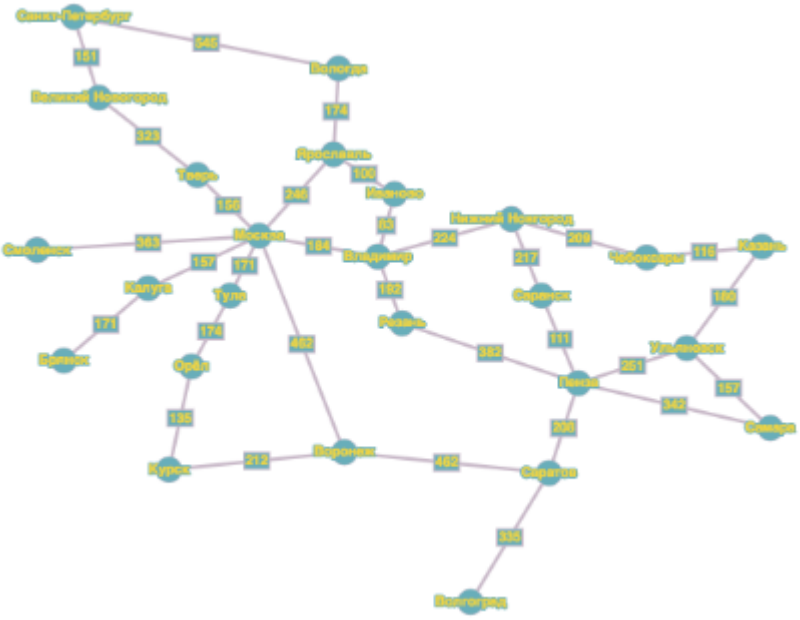


Figure 4: Grafo das cidades européias centrais, seis vértices ímpares, tempo de execução 0.02613 seg



3.2 Gráficos

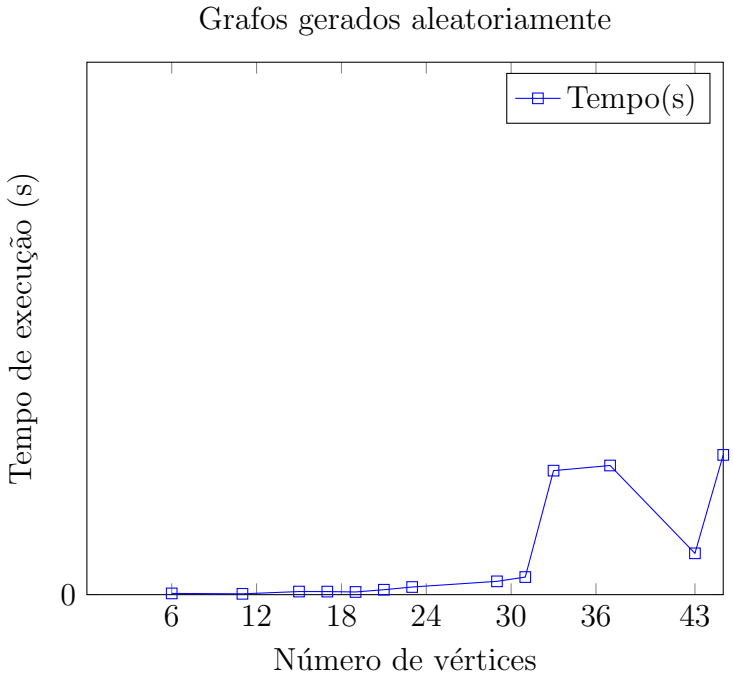


Figura 1: Aqui temos o gráfico da tabela de dados da seção anterior. Há uma variação negativa de tempo entre o teste com 37 vértices e o de 43 vértices. Isso se deve ao fato do grafo de 43 vértices ser quase Euleriano, e o de 37 ter muitos vértices de grau ímpar.

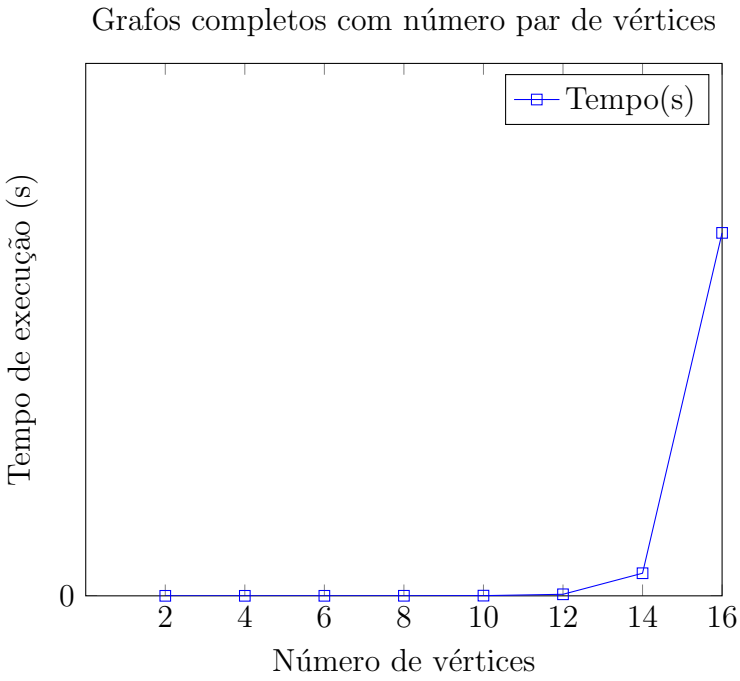


Figura 2: Temos aqui o gráfico de tempo de execução do algoritmo para grafos completos com número par de vértices, ou seja, cada vértice do grafo tem grau ímpar. Percebe-se que a partir de 16 vértices o tempo de execução sobe muito, isso se deve ao fato da função que gera as duplas únicas de vértices ímpares ter complexidade exponencial.

4 Conclusões

Conseguimos resolver o problema do Carteiro Chinês, e vimos que dependendo do número de vértices ímpares no grafo, a solução pode ser encontrada com relativa rapidez, já que os algoritmos de Hierholzer e Dijkstra são bem eficientes. Ao realizar os testes com o algoritmo que desenvolvemos, encontramos um limite de vértices de grau ímpar com os quais conseguimos lidar. Este limite foi de 18 vértices. Ao chegar neste número, o computador não conseguia calcular as duplas únicas de vértices ímpares e por ser uma implementação recursiva, travava a máquina. Uma possível melhoria seria a implementação desta função de forma não recursiva.

5 Referências

As referências foram utilizadas da seguinte forma:

- **[1]:** Entendimento do problema, do algoritmo, e corretude do mesmo.
- **[2]:** Exemplos de grafos utilizados e testes para o algoritmo.

References

[1] Bill Jackson: The chinese postman problem.
`http://www.maths.qmul.ac.uk/ bill/MAS210/ch8.pdf`

[2] Graphonline: Create graphs and find the shortest path.
`https://graphonline.ru/en/`