Trabalho Prático II
Algoritmos II

Igor Lacerda Faria da Silva

1 Introdução

O Trabalho Prático 2 da disciplina de Algoritmos II possui como proposta a análise de diferentes algoritmos para resolver o Problema do Caixeiro Viajante (PCV), com algumas restrições. Em suma, foi implementado um gerador de instâncias do problema, que são submetidas aos três algoritmos desenvolvidos (*Twice Around The Tree*, Christofides e *Branch And Bound*) e suas métricas de execução são coletadas e analisadas.

As instâncias do PCV seguem a restrição de possuir como função de custo uma métrica. Ou seja, uma função que atende às seguintes propriedades:

- $c(u, v) \geq 0$
- $c(u, v) = 0 \Leftrightarrow u = v$
- $c(u, v) = c(v, u)$ (Simetria)
- $c(u, v) \leq c(u, w) + c(w, v)$ (Desigualdade Triangular)

As métricas usadas são a distância euclidiana, definida para $P = (p_x, p_y) \wedge Q = (q_x, q_y)$ como:

$$d_{\text{euclidiana}} = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

E a distância de Manhattan, definida como:

$$d_{\text{Manhattan}} = |p_x - q_x| + |p_y - q_y|$$

2 Implementação

O trabalho foi implementado na linguagem Python, versão 3.10.8, no sistema operacional Linux. O programa segue o paradigma de programação procedural, visto que não há uma distinção muito clara de quais seriam as classes em uma abordagem de programação orientada a objetos. Dito isso, foi implementada uma única classe (`Node`). O programa foi testado usando a biblioteca `pytest`.

2.1 Arquivos

O programa está dividido em 4 módulos: `calculate`, `generators`, `measure` e `algorithms`, sendo os 3 primeiros de auxílio ao quarto, que implementa de fato os algoritmos. Além disso, existe um módulo de testes que engloba os testes dos outros módulos.

A divisão dos 3 módulos auxiliares é orientada pelo que cada função faz. Métodos que fazem algum cálculo ficam em `calculate`, e assim por diante. Mais especificamente, o primeiro verbo no nome de um método determina seu módulo.

2.1.1 Calculate

O módulo `calculate` é o mais simples, tendo como propósito apenas o cálculo de algumas medições úteis. Possui duas funções: uma para calcular a distância de um conjunto de pontos usando uma determinada métrica. A outra função recebe um ciclo e um grafo e retorna o custo de se percorrer esse caminho.

A função implementada pela biblioteca `networkx` para o algoritmo de Christofides retorna uma lista de vértices, que inclui o vértice de partida duas vezes (uma vez no final para fechar o ciclo). Para manter a compatibilidade com essa decisão da biblioteca, a função `calculate_cost()` assume que o caminho possui o vértice inicial igual ao vértice final.

2.1.2 Generators

O módulo `generators` possui duas funções. Seu propósito é, naturalmente, gerar instâncias do PCV. A função `generate_points()` gera um conjunto de n pontos no plano cartesiano, entre um piso e teto passados como parâmetros. As coordenadas dos pontos são números inteiros, pela especificação. A função `generate_instances()` cria um grafo do PCV usando as funções `generate_points()` e `calculate_distance()`, do módulo anterior.

2.1.3 Measure

O módulo `measure` possui uma única função: `measure_algorithm()`, que é chamada pelo programa principal na hora de medir o tempo de execução de um algoritmo, usando a biblioteca `time`.

2.1.4 Algorithms

O módulo principal do programa é baseado nos três algoritmos para solução do PCV: algoritmo de Christofides, *Branch And Bound*, *Twice Around The Tree*. Também existe uma camada de abstração que facilita a execução em escala.

- *Twice Around The Tree*

O algoritmo de implementação mais fácil foi o *Twice Around The Tree* (TATT), pois foi permitido o uso da biblioteca `networkx` para fazer o cálculo da árvore geradora e o caminhamento pré-ordem dos vértices. Esse algoritmo é (2) aproximativo e explora as árvores geradoras mínimas (AGM) como ideia principal: é computacionalmente simples calcular a AGM e o ciclo encontrado pelo algoritmo, é, no máximo, duas vezes pior que o ciclo ótimo. Com a AGM em mãos, o grafo é percorrido usando uma DFS para construir o caminho, excluindo repetições.

- Christofides

  O algoritmo de Christofides também não possuiu grandes dificuldades de implementação, porque também foi permitido o uso funções da biblioteca `networkx`. O algoritmo também é aproximativo e começa com o cálculo da AGM. A ideia principal é, como no TATT, caminhar o circuito euleriano, transformando-o em hamiltoniano. Para isso, é feito um *matching* perfeito de peso mínimo no subgrafo induzido pelos vértices de grau ímpar.

  É então construído um multigrafo, com as arestas do *matching* e da AGM. Neste novo grafo é calculado o circuito euleriano, e seus vértices repetidos são excluídos, chegando-se em uma lista de vértices que é o circuito hamiltoniano encontrado pelo algoritmo. Apesar de mais complicado, o fator de aproximação desse algoritmo é 1.5, o que é um ganho significativo em relação ao TATT.

  Como esse é um algoritmo famoso, ele está implementado na `networkx`, e foi criado um teste que compara os custos dos circuitos de ambas as implementações.

- *Branch And Bound*

  Sem dúvida, o algoritmo de implementação mais difícil foi o *branch and bound* (BNB). Seu código foi inspirado no pseudocódigo usado nas aulas, embora a função de *bound* exigiu certa criatividade. Diferentemente dos outros algoritmos, o BNB é exato, e possui complexidade exponencial.

  A ideia principal de algoritmos de ramificar e limitar é explorar toda a árvore de possibilidades, usando como critério para o próximo item da busca uma estimativa "realista" do quanto seria o custo daquele ramo. Uma vez que um caminho foi finalizado, ele passa a ser o melhor atual, e caminhos com estimativas piores do que ele não são explorados. Assim, alguns caminhos não são explorados, o que pode reduzir (na melhor das hipóteses) drasticamente o espaço de busca.

  Para o usar o BNB no PCV, o principal desafio é fazer uma função de estimativa que seja rápida. Nesta implementação, no inicio do algoritmo é feita uma estimativa inicial, em tempo quadrático, que seria o custo "ideal" de um circuito hamiltoniano. Essa estimativa toma como base os dois menores pesos das arestas que saem de cada vértice.

  Para cada nó da árvore de busca, o custo é atualizado da seguinte maneira: como é inserido um único vértice por nó, é adicionada apenas uma aresta, então é necessário fazer apenas uma modificação, a depender se a aresta inserida é uma das menores para aquele vértice do grafo ou outro. Se a aresta inserida é a primeira para dado nó e não é menor para este, então o custo deve ser atualizado pela diferença de peso entre a maior das menores e a aresta inserida[1]. É usado um marcador booleano para indicar se já foi inserida uma aresta naquele nodo.

Essa é explicação de apenas um dos casos de atualização. As 7 combinações possíveis, que consideram se a primeira aresta inserida é a menor, a segunda menor ou qualquer outra (e similarmente para a segunda aresta), estão agrupados em um teste, baseado em um exemplo das aulas. A lógica dos outros casos é similar (e provavelmente também desnecessariamente complicada).

Com a função de *bound* funcionando, a implementação é bastante direta. Na primeira vez que um caminho é completamente percorrido, seu nó passa a ser a solução candidata. As estimativas são comparadas com o custo da candidata e são percorridas mais profundamente se viáveis, atualizando a solução candidata caso necessário. Apesar disso, o algoritmo não deixa de ser exponencial.

Justamente por isso, não foi registrada nenhuma execução para esse algoritmo na seção de Análise Experimental, mesmo para $n = 2^4$. Nos testes realizados, o tamanho máximo que foi possível executar em menos de meia hora foi $n = 12$, com tempos variando de 6 a 28 minutos, tanto para a distância euclidiana como para a de Manhattan. Tendo em vista o custo exponencial, acredita-se que a implementação deixa a desejar em performance, por algumas ordens de magnitude.

Esta classe conta com dois submódulos: a classe `Node`, mencionada anteriormente, cujos únicos propósitos são agrupar cada nó na exploração do BNB e prover um comparador entre eles. Adicionalmente, os métodos para calcular o *bound* são implementados separadamente: `initial_bound()` e `update_bound()` (e seu método auxiliar). A própria função do *branch and bound* também possui um método auxiliar que condensa a inserção de novos nós na busca.

2.2 Programa Principal

O programa principal gera instâncias do PCV, de tamanho variando exponencialmente de $n = 2^4$ a $n = 2^{10}$, usando ambas as métricas apresentadas, as roda sobre os algoritmos *Twice Around The Tree* e Christofides[2] e salva os seguintes resultados em um *DataFrame* da biblioteca `pandas`: tamanho, algoritmo, distância, tempo e custo. O *DataFrame* é convertido em `csv` e salvo como "tsp.csv", na raiz do projeto.

[1] É um pouco mais complicado que isso, pois é necessário pegar o teto da divisão por 2 do incremento, mas para evitar confusões adicionais em um parágrafo que já não é simples de entender, essa informação foi omitida.

[2] O BNB é excluído devido à sua performance inadequada.

3 Análise Experimental

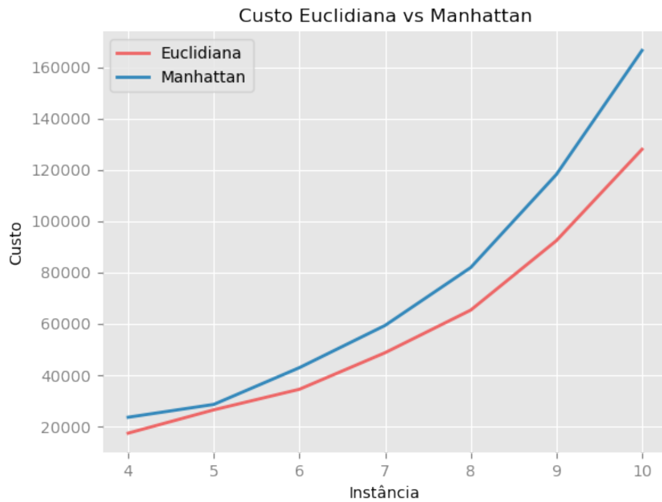
Após o processamento dos dados do *DataFrame*, obteve-se as seguintes tabelas:

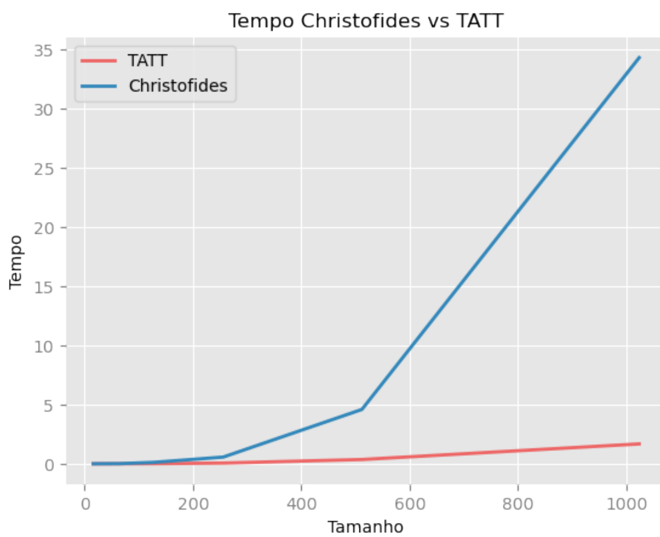
Twice Around The Tree

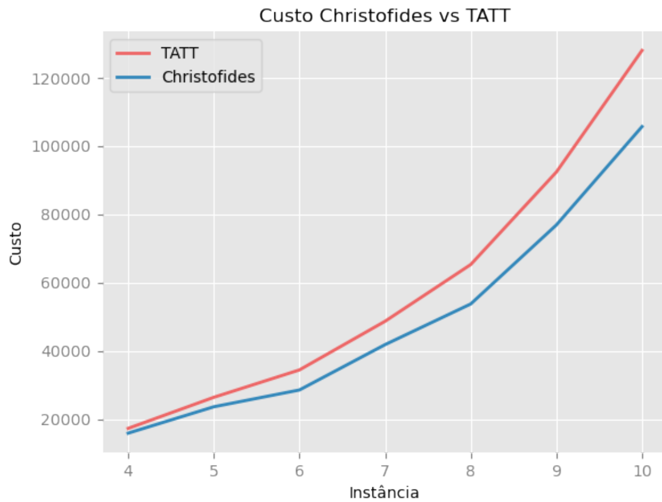
Tamanho	Tipo	Tempo (s)	Custo
4	Euclidiana	0.000587	17298.593
4	Manhattan	0.000451	23502
5	Euclidiana	0.000882	26395.748
5	Manhattan	0.000880	28508
6	Euclidiana	0.003095	34413.646
6	Manhattan	0.003430	42842
7	Euclidiana	0.013013	48664.624
7	Manhattan	0.038881	59302
8	Euclidiana	0.062127	65274.484
8	Manhattan	0.068079	81876
9	Euclidiana	0.363078	92354.840
9	Manhattan	0.367995	118148
10	Euclidiana	1.679754	127960.470
10	Manhattan	1.707548	166504

Christofides

Tamanho	Tipo	Tempo (s)	Custo
4	Euclidiana	0.001500	15898.835
4	Manhattan	0.001099	18564
5	Euclidiana	0.007076	23624.548
5	Manhattan	0.003580	25500
6	Euclidiana	0.012609	28545.995
6	Manhattan	0.013538	33378
7	Euclidiana	0.124593	41825.469
7	Manhattan	0.091646	49168
8	Euclidiana	0.569033	53730.344
8	Manhattan	0.750834	69402
9	Euclidiana	4.593676	76858.895
9	Manhattan	5.491562	96106
10	Euclidiana	34.316589	105658.274
10	Manhattan	33.201210	135150


Custo Euclidiana vs Manhattan
Euclidiana
Manhattan
160000
140000
120000
100000
80000
60000
40000
20000
Custo
4
5
6
7
8
9
10
Instância


Tempo Christofides vs TATT
TATT
Christofides
35
30
25
20
15
10
5
0
Tempo
0
200
400
600
800
1000
Tamanho


Custo Christofides vs TATT
TATT
Christofides
120000
100000
80000
60000
40000
20000
Custo
4
5
6
7
8
9
10
Instância

4 Conclusão