

Trabalho Prático 2 - Algoritmos Aproximativos

Thiago Henrique Moreira Santos¹

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

thiago.santos@dcc.ufmg.br

1. Introdução

A ideia do trabalho foi implementar alguns algoritmos para serem aplicados no problema do caixeiro viajante métrico. Estes problemas são instâncias especiais do problema do caixeiro viajante clássico e nos permite utilizar algoritmos aproximativos com fator de aproximação constante nos problemas que se encaixam nessa classe. Para uma instância do caixeiro viajante ser considerada métrica, ela precisa satisfazer três propriedades:

- Simetria: $d_{ij} = d_{ji}$
- Não-negatividade: $d_{ij} \geq 0$
- Desigualdade Triangular: $d_{ij} \leq d_{ik} + d_{kj}$

São esses os problemas que serão utilizados nesse trabalho. Os três algoritmos a serem implementados são o **branch and bound**, **twice around the tree** e **christofides**. A seguir será apresentada três sessões, uma sessão onde vou tratar dos detalhes de implementação, como estruturas de dados utilizadas, algoritmos e algumas otimizações que foram realizadas, a segunda será composta por análises da performance dos algoritmos para algumas instâncias para comparar o desempenho entre eles e para finalizar, uma sessão com as conclusões do trabalho tentando identificar os cenários que cada algoritmo pode ser preferido em relação aos outros.

2. Detalhes de Implementação

Nesta seção vou comentar um pouco sobre as implementações que fiz para os algoritmos do trabalho. A linguagem que utilizei para isso foi o **C++** e agora, para cada algoritmo, comentarei sobre algumas decisões e também vou apresentar alguns dos métodos que acho pertinente.

2.1. Branch and Bound

Bom, esse é o único algoritmo exato presente neste artigo e a ideia dele é bastante simples: utilizamos dos conhecimentos específicos que temos do problema para tentar determinar limites que possam os ajudar a **evitar ter que explorar alguns nós**, em última instância é um algoritmo de força bruta "mais inteligente".

Eu tentei implementar esse algoritmo mais ou menos do jeito que vi nas aulas. Antes de falar do algoritmo em si, a representação que adotei para o grafo foi por meio de uma **matriz de adjascência** a vantagem de utilizá-la para esse caso é que a consulta do custo de uma aresta é feita em tempo constante e isso deixa a parte de computar o custo do circuito no final mais rápida. A primeira coisa que eu fiz foi usar uma fila de prioridade para armazenar os nós da árvore, assim quando eu encontrar um vértice promissor vou adicionar lá de modo a ser o primeiro a ser explorado. Então começa um

```

void tspBranchAndBound(std::vector<std::vector<int>>& adjMatrix) {
    std::vector<Node> Q;

    Node v, u;
    u.path.push_back(0);
    u.lower_bound = 0;
    u.cost = 0;

    Q.push_back(u);

    int min_cost = INT_MAX;
    while (!Q.empty()){
        u = Q.front();
        Q.erase(Q.begin());

        if (u.lower_bound < min_cost) {
            int i = u.path[u.path.size() - 1];

            for (int j = 0; j < adjMatrix.size(); j++){
                if (std::count(u.path.begin(), u.path.end(), j)) {
                    v.path = u.path;
                    v.path.push_back(j);

                    if (v.path.size() == adjMatrix.size()) {
                        v.cost = u.cost + adjMatrix[i][j] + adjMatrix[j][0];
                        if (v.cost < min_cost)
                            min_cost = v.cost;
                    }
                    else {
                        v.cost = u.cost + adjMatrix[i][j];
                        v.lower_bound = calcularLowerBound(adjMatrix, v);
                        if (v.lower_bound < min_cost)
                            Q.push_back(v);
                    }
                }
            }
        }
    }
}

```

Figura 1. Método principal do branch and bound (versão do código sem comentários para uma melhor visualização)

loop que permanece até que a fila fique vazia. Dentro do loop, o if mais externo verifica se é possível fazer uma poda na árvore, o "lower_bound" é o menor valor possível que pode ser obtido a partir de um nó, min_cost é a melhor solução viável que tenho até o momento, se o lower_bound for maior do que uma solução que eu já tenho, não preciso continuar explorando aquele ramo. Em relação ao cálculo do lower_bound em si, o que eu fiz foi criar um método chamado "computeLowerBound" que dado um vértice, vai descendo na árvore pegando a média das duas melhores arestas possíveis, ainda que isso gere uma solução inválida, isso já nos auxilia, porque uma válida será no máximo igual aquilo (se o "aquilo" for válido). Continuando, pegamos o primeiro elemento da fila e verificamos se ele já apareceu no caminho usando a função count que retorna a quantidade de vezes que um elemento aparece num dado intervalo, se retornar 0 o elemento não aparece, este é o caso em que vamos aumentar o caminho. Depois há uma verificação se chegamos na folha que é o momento onde verificamos se nosso "min_cost" precisa ser atualizado.

Honestamente, eu acredito que essa não é a implementação mais adequada para as instâncias desse trabalho. Quando fui fazer os testes, percebi que meu algoritmo nunca terminava nem para a menor instância do TSP Lib que tem 51 cidades (cheguei a deixar rodando por mais de 7 horas), achei isso estranho e fui buscar por outras instâncias menores até que achei uma (que estará nas referências) com 5 cidades e ele retornou o resultado ótimo correto. O que eu acho que tá acontecendo é que pelo fato de eu ter feito usando busca em profundidade ele tem que ir até o fundo da árvore para achar uma solução válida que vai permitir o recurso das podas, esse processo já é demorado por si e pode piorar se ele ficar achando soluções ruins por várias vezes. Talvez uma mistura dessas duas coisas (largura e profundidade) seria o ideal, lembro de ter visto algo assim

em Pesquisa Operacional,

2.2. Twice-Around-The-Tree

Esse é o primeiro algoritmo aproximativo que vou escrever sobre aqui nesse documento. A ideia geral dele pode ser resumida nos seguintes passos:

- Encontrar a árvore geradora mínima MST do grafo do problema.
- Duplicar cada aresta da MST como forma de lidar com vértices de grau ímpar.
- Encontrar um circuito euleriano (vista todas as arestas uma vez)
- Remover os vértices repetidos.

Double Tree Algorithm

- **Find a minimum spanning tree T**
- Duplicate every edge in T
- Find an Eulerian tour of resulting multi-graph
- Shortcut Euler tour to avoid repeated vertices

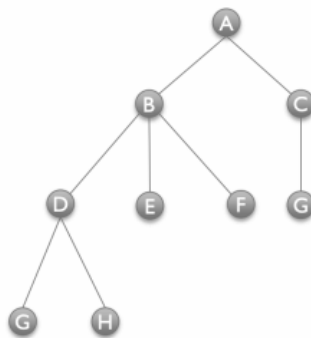


Figura 2. Algoritmo Twice-Around-The-Tree

Em relação a implementação desse algoritmo, acredito que a parte mais interessante de se mencionar é como foi computada a árvore geradora mínima. Eu escolhi o algoritmo Kruskal para computar a árvore geradora mínima usando a estratégia **union-find**, que foi o que eu vi em Algoritmos I.

A primeira coisa a se fazer é ordenar as arestas em ordem crescente, sempre vamos querer tentar adicionar a aresta mais barata para a solução. Para o resto da iteração vamos checar se os endpoints dela estão no mesmo conjunto do union-find ou não, isso é verificado através do método **findRoot** para encontrar qual é a raiz de cada endpoint. Se eles forem iguais, então eles estão no mesmo componente conectado e se adicionarmos essa aresta, os ciclos serão inválidos. Teríamos uma situação parecida com a imagem acima se considerássemos apenas os vértices D, G, H, quando o caixeiro for de D para G ou H, só daria pra voltar pra D e isso é inválido. Para os casos em que eles estão em

componentes diferentes, podemos adicionar essa aresta e continuar o processo e fazer a união dos componentes conectados em um grafo só. O processo continua até a MST estar completa.

Depois disso basta computar o custo do caminho euleriano e multiplicar por 2 para ter o resultado final. Não consegui pensar muitas formas de otimizar o código para esse algoritmo, a única coisa que fiz, melhora um pouco o resultado, mas não a execução em si. Que é a última etapa que esta nessa imagem que coloquei, onde desconsideramos os vértices repetidos.

Em termos de complexidade de tempo e espaço, o algoritmo é bem eficiente. Na complexidade de tempo quem domina o custo é a ordenação, com complexidade $O(E \log E)$ usando um algoritmo de ordenação eficiente. No espaço, temos um custo associado com os vetores que representam a MST, o vetor de "parents" dos vértices e a matriz de adjacência que representa o grafo, logo $O(2 \cdot E + V)$.

2.3. Christofides

O algoritmo de Christofides é o segundo algoritmo aproximativo e o último algoritmo implementado para o trabalho. A ideia dele é parecida com a do algoritmo anterior, mas ele tem um passo diferente na forma de lidar com os vértices de grau ímpar na MST e isso gera uma melhoria para o algoritmo tornando ele **1,5 aproximativo** ao invés do algoritmo anterior que é **2 aproximativo**.

Christofides Algorithm

- Find the minimum spanning tree T
- Compute O : the set of odd degree vertices in T
- Find the min-cost perfect matching M of subgraph induced by O
- Return shortcut of Euler tour of $T \cup M$

$|O|$ must be even

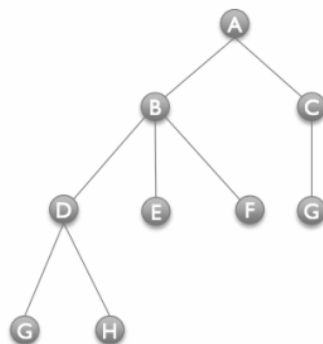


Figura 3. Algoritmo de Christofides

Basicamente, depois de encontrar a MST, ao invés de duplicar as arestas para lidar com os vértices de grau ímpar, ele primeiro constrói um conjunto com todos os vértices de grau ímpar, depois toma o grafo O , que é o grafo induzido por esses vértices de grau

ímpar. Após isso, vamos encontrar o **emparelhamento perfeito de custo mínimo M** nesse grafo e depois vamos adicionar as arestas de M no grafo O para gerar um novo grafo $G = M \cup O$ e é nesse grafo que vamos fazer o circuito euleriano. Esse algoritmo é muito semelhante ao último, então vou comentar apenas os pontos que foram diferentes:

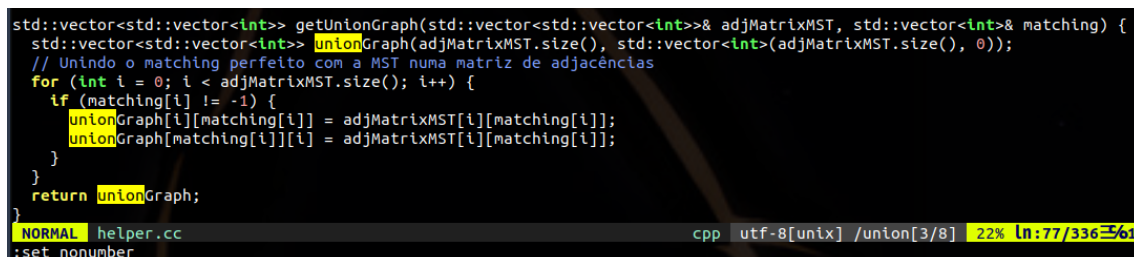
- Implementação do algoritmo de blossom
- Implementação de um método para unir os dois grafos
- Tentativa de implementação de uma otimização conhecida para o algoritmo

2.3.1. Algoritmo de Blossom

Tive de implementar o algoritmo de Blossom para encontrar o emparelhamento perfeito de custo mínimo. A ideia principal do algoritmo é encontrar os "augmenting paths" que são uma sequência de vértices casados e não-casados onde o primeiro e o último vértice não tem casamento. Tem um teorema que diz que se um grafo contém um "augmenting path" o matching dele não é máximo, o que fazemos então é iterativamente aumentar o matching até que não haja um "augmenting path". A complexidade do algoritmo é $O(V^3)$.

2.3.2. Método para fazer a união de dois grafos

Método bem simples, basicamente adicionei as arestas do grafo O na MST:



```
std::vector<std::vector<int>>> getUnionGraph(std::vector<std::vector<int>>& adjMatrixMST, std::vector<int>& matching) {
    std::vector<std::vector<int>>> unionGraph(adjMatrixMST.size(), std::vector<int>(adjMatrixMST.size(), 0));
    // Unindo o matching perfeito com a MST numa matriz de adjacências
    for (int i = 0; i < adjMatrixMST.size(); i++) {
        if (matching[i] != -1) {
            unionGraph[i][matching[i]] = adjMatrixMST[i][matching[i]];
            unionGraph[matching[i]][i] = adjMatrixMST[i][matching[i]];
        }
    }
    return unionGraph;
}
```

Figura 4. Método para unir a MST com o subgrafo induzido nos vértices de grau ímpar

2.4. Tentativa de otimizar o algoritmo

No algoritmo clássico depois de computar o grafo que é união da MST com o subgrafo induzido pelos vértices de grau ímpar, ele faz um circuito euleriano no grafo, depois transforma ele num circuito hamiltoniano removendo as arestas repetidas e adicionando a aresta final que volta para o início do tour. O que fiz foi utilizar o "pre-order dfs" que é quase a mesma coisa do dfs clássico, mas no momento que ele passa pelo vértice já visita ele. Apenas aplicando o dfs pre-order já conseguimos encontrar o circuito hamiltoniano do tour. Ele não precisa desse passo intermediário do circuito euleriano e o mais interessante é que, como ele não visita vértices repetidos (isso vem do dfs clássico mesmo) ele acaba por utilizar da desigualdade triangular para usar aqueles caminhos diretos entre um vértice e outro que é mais barato do que passar por um intermediário, isso melhora um pouco a solução lá no final pelos testes que fiz. Ex:

Na imagem acima, quando o dfs estiver em E (portanto passou por B) ele terá que voltar para B e depois ir para F. Como o dfs não visita nós repetidos, o que ocorre

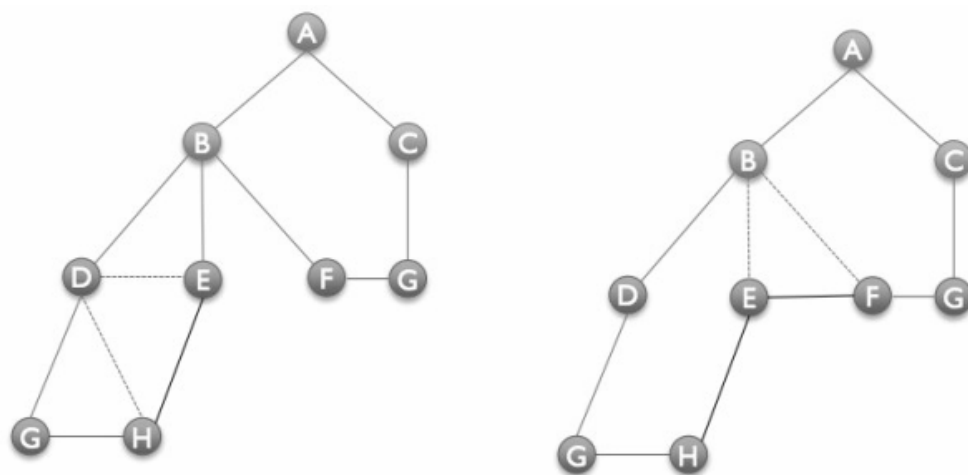


Figura 5. Ilustração do uso do caminho mais curto

na prática que o tour vai de E para F, a propriedade da desigualdade trigular nos permite fazer isso. Então quando estou computando o tour para depois obter o custo dele, eu tenho pequenos ganhos.

3. Experimentos

Nesta seção farei alguns experimentos com os algoritmos, comparando o desempenho dos três entre si utilizando algumas das instâncias da biblioteca TSPLib. Para medir o tempo dos algoritmos vou utilizar a biblioteca **chrono**. Segue abaixo duas tabelas que fiz com o resultado dos experimentos realizados e após isso uma sessão com comentários acerca deles.

Tabela 1. Tempos de Execução para Algoritmos TSP (segundos)

Instância	B & B	Twice Around the Tree	Christofides
bier127	∞	0	0
d657	∞	0	27
d1655	∞	1	429
d2103	∞	1	877
fl3795	∞	2	>1800
fnl4461	∞	4	>1800
rl5915	∞	7	>1800
brd14051	∞	41	>1800
d15112	∞	42	>1800
d18512	∞	98	>1800

3.0.1. Comentários sobre os resultados

A primeira coisa que tenho que mencionar foi que não consegui implementar uma forma automática do programa ser interrompido após ele ficar executando por mais do que 30

Tabela 2. Espaço utilizado pelos algoritmos

Instância	B & B	Twice Around the Tree	Christofides
bier127	∞	3968 KB	4096 KB
d657	∞	8724 KB	12948 KB
d1655	∞	39312 KB	62992 KB
d2103	∞	70512 KB	99184 KB
fl3795	∞	158724 KB	∞
fnl4461	∞	278544 KB	∞
rl5915	∞	534036 KB	∞
brd14051	∞	2348708 KB	∞
d15112	∞	2469764 KB	∞
d18512	∞	4489332 KB	∞

Tabela 3. Valor da solução para Algoritmos TSP

Instância	B & B	Twice Around the Tree	Christofides	Ótimo
bier127	∞	177204	155562	118282
d657	∞	84116	65826	48912
d1655	∞	112342	84524	62128
d2103	∞	150182	136756	80450
fl3795	∞	44074	∞	28772
fnl4461	∞	333686	∞	182566
rl5915	∞	1042340	∞	565530
brd14051	∞	847586	∞	469385
d15112	∞	2846974	∞	1573084
d18512	∞	1172324	∞	645238

minutos, tentei rodar o programa numa thread separada para ver se eu poderia interrompê-la depois do tempo de 30 minutos mas não deu certo. O que eu fiz então foi medir o tempo usando a biblioteca chrono e, para os casos em que eu suspeitei que poderia passar de 30 minutos, eu fiz a verificação manual colocando um cronômetro para iniciar junto com a execução, dessa forma, se passasse dos 30 minutos eu mesmo interrompia o programa. Sobre os resultados propriamente ditos, acho que é válido comentar um pouco sobre cada algoritmo separadamente dado que cada um deles apresentou características que valem a pena ser mencionadas, todos me surpreenderam de alguma forma.

3.0.2. Branch and Bound

Para a minha surpresa, o branch and bound não terminou a execução para nenhuma das instâncias do TSPLib no tempo determinado. Eu já esperava que o algoritmo extrapolaria esse limite rapidamente, mas foi mais rápido do que pensei. A menor instância do TSPLib tem 51 cidades, eu deixei o algoritmo rodando por mais de **7 horas** e ele não terminou. Obviamente eu comecei a suspeitar que tinha alguma coisa de errado na minha implementação, então busquei por instâncias menores para testar o algoritmo, até que

achei duas instâncias menores, uma com 5 cidades e outra com 15. A com 5 cidades termina rapidamente, coisa de 2 segundos e retorna o resultado correto. A instância com 15 cidades não terminou em menos de 30 minutos então abandonei. Os testes com o branch and bound me ajudaram a ter uma noção prática do porque falamos que esses problemas são "intratáveis", é bem difícil conseguir uma solução ótima para esse problema, mesmo que estejamos trabalhando com instâncias muito específicas do problema original. Acho que isso de alguma forma realça a importância dos algoritmos aproximativos, dado que os tempos observados foram bem mais praticáveis. Em termos de espaço não consegui obter muitas informações dado que o algoritmo não terminou para nenhuma das execuções que fiz com as instâncias do TSPLib...

3.0.3. Twice-Around-The-Tree

Ao contrário do último, esse me surpreendeu "na outra direção", foi bem mais rápido do que pensei. É claro que esse ganho em tempo exige um relaxamento na qualidade da solução, mas o fato de ser praticável e ainda de termos uma garantia de qualidade, **fator de aproximação**, torna esse algoritmo bem interessante. Se observamos no gráfico, este foi o único algoritmo que conseguiu terminar, **com folga**, todas as instâncias no tempo pré determinado e podemos verificar nos resultados que a solução sempre respeita o fator de aproximação, nunca sendo pior do que 2 vezes a solução ótima.

3.1. Christofides

Confesso que tinha expectativas altas para os resultados utilizando desse algoritmo, mas que rapidamente eu percebi que esse 0.5 de ganho na qualidade da solução cobra um preço muito caro em termos de tempo. A primeira execução que fiz, com 127 cidades, terminou instantaneamente e apresentando uma solução consideravelmente mais próxima do que consegui com o algoritmos anterior. Mas já no segundo teste, uma instância com aproximadamente 500 cidades a mais do que a anterior, o algoritmo demorou **mais de 7 minutos** para encontrar uma solução, enquanto que o anterior demorou **1 segundo**. E foi piorando cada vez mais, com uma instância de pouco mais de 2000 cidades (apenas 500 a mais do que o teste anterior), o algoritmo demorou quase **15 minutos** para encontrar uma solução... a partir daí ele não terminava mais em menos de 30 minutos. A minha primeira impressão depois disso foi um pouco de decepção, mas olhando com mais praticidade, 2103 cidades é um número considerável e conseguimos um resultado mais próximo do ótimo, em termos práticos isso pode ser bem interessante. Com certeza é mais animador do que esperar o branch and bound encontrar a solução ótima.

4. Conclusão

Em termos de espaço, os algoritmos mostraram um compartimento semelhante, com exceção do branch and bound. No caso deste o último não conseguir obter muitas informações dado que o algoritmos não terminou para nenhuma das instâncias do TSPLib. Enquanto que nos outros dois podemos ver que o crescimento do espaço utilizado acontece de forma "mais comprtada" na medida que aumentamos o tamanho da entrada, no caso do Twice Around The Tree ele começa gastando cerca de **4 MB** para a instância bier127, chegando até quase **5 GB**.

Em relação ao tempo, apesar de certa surpresa com alguns dos resultados é claro que isso fica bem mais claro quando lembramos que, apesar de serem instâncias especiais, o problema ainda continua sendo **exponencial**. É por esse motivo que, quando passamos da instância **d15112** para **d18512** (uma instância 1.22 vezes maior), observamos que o tempo para resolver foi 2.33 vezes maior, e isso usando um algoritmo que tolera uma solução 2 vezes pior do que a ótima... Essa discrepância só vai aumentando com o aumento do tamanho da entrada, que é característico de um problema exponencial.

As minhas considerações finais são de que os algoritmos aproximativos tem grande valor prático. Se tomarmos um exemplo de uma distribuidora que atende um estado que é representada por aquela instância com 1655 cidade por exemplo, podemos observar que o algoritmo de Christofides dá uma solução muito boa e em apenas **7 minutos**, enquanto que o branch and bound poderia demorar um tempo impraticável e o twice around the tree provê uma solução que é consideravelmente pior do que o algoritmo de christofides. Então, variando de caso a caso, consigo enxergar diversas aplicações que poderiam ser feitas utilizando desses algoritmos.

Referências

- [1] Yumin Lee - Union-Find-Algorithm. <https://yuminlee2.medium.com/union-find-algorithm-ffa9cd7d2dba>
- [2] Williams University - Approximating the TSP Problem. <http://cs.williams.edu/~shikha/teaching/fall19/cs256/lectures/Lecture32.pdf>
- [3] Wikipedia - Blossom Algorithm. https://en.wikipedia.org/wiki/Blossom_algorithm
- [4] jkinable - Extra Instances for Tests 1. <https://github.com/coin-or/jorlib/tree/master/jorlib-core/src/test/resources/tspLib/tsp>
- [5] Bratet - Extra Instances for Tests 2. https://github.com/Bratet/VNS_TSP/blob/main/benchmark_dataset/5.txt