Trabalho Prático 2 - Algoritmos 2
Caixeiro Viajante

Filipe A. Mendes[1]

[1]Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

filipemendes@dcc.ufmg.br

Abstract. *This project delves into the Traveling Salesman Problem, recognized as NP-Complete. We explore three distinct algorithms for its resolution: Branch-and-Bound, seeking an optimal solution, Twice-Around-the-Tree, and Christofides, for approximate solutions. Through comprehensive metric analysis, the Christofides algorithm emerged as the superior choice across general scenarios.*

Resumo. *O trabalho aborda o problema do Caixeiro Viajante, reconhecido como NP-Completo. Exploramos três distintos algoritmos para sua resolução: o Branch-and-Bound, buscando uma solução ótima, o Twice-Around-the-Tree e o Christofides, para soluções aproximadas. Ao analisar diversas métricas, o algoritmo de Christofides destacou-se como a escolha superior em cenários gerais.*

1. Introdução

O problema do caixeiro viajante consiste em encontrar a rota de menor peso que passe por todas as cidades uma única vez em dado conjunto e então retornando à cidade de origem. O problema é NP-Completo e é combinatório, com a solução força bruta tendo complexidade de tempo O(n!). Entretanto, para utilizar algoritmos aproximativos, precisamos restringir o problema, tratando do caixeiro viajante euleriano.

O objetivo do trabalho é comparar soluções exatas e aproximadas para problemas difíceis, assim como comparar soluções aproximadas entre si. Para isso, serão utilizados os algoritmos de Branch-and-Bound, Twice-Around-the-Tree e, por fim, o algoritmo de Christofides, esclarecendo, assim, qual a melhor solução para cada caso, mas com um tempo de execução total de 30 minutos.

O branch-and-bound se mostrou inviável em tempos pequenos, não conseguindo resultado nem no dataset de menor quantidade de vértices, e dentre os aproximativos, o de christofides é o melhor a se usar, mas em grandes quantidades de vértices ele fica com um tempo consideravelmente maior que o twice-around-the-tree, que ainda gera boas soluções, então se o tempo for uma prioridade, o twice-around-the-tree é uma escolha melhor.

Na seção 2 será discutido em mais detalhes sobre os algoritmos em questão e as decisões que foram tomadas na implementação de cada um deles, depois, na seção 3, serão apresentados experimentos realizados com 78 datasets e feitas algumas comparações sobre as métricas coletadas para cada algoritmo e cada dataset. Na seção 4, apresentamos

as conclusões, discutindo qual a melhor solução para diferentes casos. Por fim, na seção 5 será explicado brevemente como executar os algoritmos.

2. Implementação

2.1. Branch-and-Bound

A técnica de branch-and-bound consiste na exploração exaustiva das soluções, mas cortando, ou podando, galhos da árvore de busca que são certamente ou inviáveis ou piores que uma solução já obtida. Tomando como exemplo o próprio problema do caixeiro viajante, se uma solução é inviável ela repete uma cidade no meio do trajeto, ou então, se temos uma solução com peso x, e chegamos em um galho que só pode ter peso total maior ou igual a x, não precisamos verificar a solução. Dessa maneira, apesar do problema continuar com complexidade fatorial, reduzimos consideravelmente a árvore de busca. Foi escolhido c++ para a implementação já que o código escrito em c++ seria mais rápido que o escrito em python sem bibliotecas, reduzindo o tempo de execução, que seria essencial em um algoritmo exponencial.

Uma parte essencial é decidir a função de estimativa do limite inferior de uma solução. É possível ter uma função complexa para ter uma estimativa melhor, mas como essa função é executada o tempo todo no algoritmo, ter uma função simples pode resultar em um tempo igual ou até melhor que uma função muito sofisticada. Por isso, a função de estimativa foi a mais simples possível: as arestas escolhidas representam o custo atual e a estimativa = (custo atual) + (custo da aresta de melhor valor do último e primeiro vértices escolhidos + custo das 2 arestas de melhor valor para os vértices restantes) / 2.

Para o algoritmo em si, existem algumas possibilidades na implementação, como fazer a busca ou em profundidade (DFS), visando sempre descer para uma folha na árvore, ou ir procurando a melhor estimativa de nó e expandi-lo (BFS). Para isso, foram feitas algumas considerações, como memória utilizada e tempo de execução.

Em termos de memória, a DFS vence completamente, já que é possível fazer com que seja gasto espaço O(n), por exemplo utilizando um vector para acesso aleatório e utilizando o princípio de last-in first-out, de uma pilha, enquanto a BFS necessita guardar os vértices, ou cidades, escolhidos em cada nó da árvore, utilizando assim, memória proporcional ao tamanho da árvore, ou seja, O(n!), entretanto, ao ir visitando os nós, esses vetores de vértices escolhidos vão sendo retirados da fila de prioridade, então não existe em momento algum, n! vetores no programa, sendo difícil quantificar, de fato, a quantidade total de memória gasta.

Já, em termos de tempo, a BFS tem uma vantagem pelo fato de priorizar soluções melhores, melhorando as podas pelo limite inferior ser pior que a melhor solução encontrada, enquanto isso, a DFS encontra soluções piores em geral inicialmente, o que faz com que as podas na DFS não sejam tão boas, então o algoritmo tem que passar por um espaço de busca maior, afetando negativamente o tempo de execução.

Inicialmente foi implementada uma DFS, já que a melhora em memória da BFS para a DFS é bem melhor que a melhora em tempo da DFS para a BFS, porém após testar o menor dataset, com 51 vértices, durante os 30 minutos máximos, a implementação em DFS não foi capaz de produzir a solução ótima, então foi tentado novamente com a BFS. Para encontrar uma solução inicial para começar a podar a árvore mais cedo, foi

inicializada a BFS com um algoritmo guloso, descendo para as folhas pegando sempre o caminho com melhor estimativa, e assim que uma solução inicial fosse encontrada, começar, de fato, a BFS, mas assim como a DFS, não foi capaz de conseguir a solução ótima dentro dos 30 minutos.

2.2. Twice-Around-the-Tree

O algoritmo twice-around-the-tree pega o grafo G, cria uma árvore geradora mínima e com essa AGM, caminha em pré-ordem para encontrar o ciclo hamiltoniano que dá uma solução do caixeiro viajante euleriano no máximo 2 vezes pior que a solução ótima. A implementação foi feita em Python já que a biblioteca networkx possui algoritmos muito bem otimizados para geração da AGM e para o caminhamento na árvore. Com a biblioteca networkx, a implementação foi trivial, sem ter decisões que fossem alterar significativamente o algoritmo final.

2.3. Christofides

De forma similar ao twice-around-the-tree, o algoritmo de christofides cria uma AGM com o grafo G, mas computa um subgrafo G' a partir de G com os vértices da AGM de grau ímpar, para depois achar um matching M de peso mínimo em G', adicionar as arestas de M na AGM e então achar um caminho hamiltoniano, que resulta em uma solução no máximo 1,5 vezes pior que a solução ótima. Novamente foi feito em Python com a biblioteca networkx, que tornou a implementação trivial pelos mesmos motivos que o twice-around-the-tree.

3. Experimentos

Para os experimentos, foram utilizados 78 datasets da biblioteca TSPLIB [1] testando cada um para os tres algoritmos diferentes, coletando as métricas: tempo de execução, memória utilizada e qualidade da solução, com o tempo de execução limitado a 30 minutos.

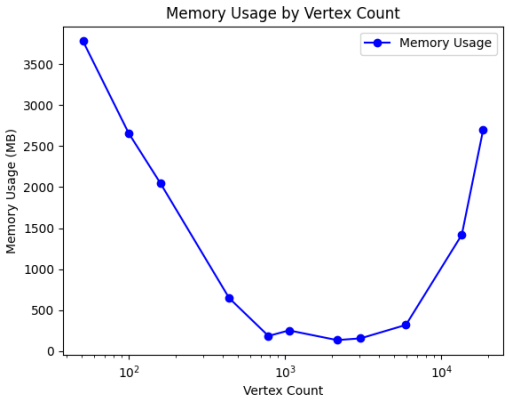
As métricas foram coletadas da seguinte maneira: o tempo e a quantidade de memória gasta foram coletados utilizando o comando /usr/bin/time -v no bash e a qualidade da solução é o custo dado pelo algoritmo / custo da solução ótima, que já foi calculada para cada um dos datasets.

3.1. Branch-and-Bound

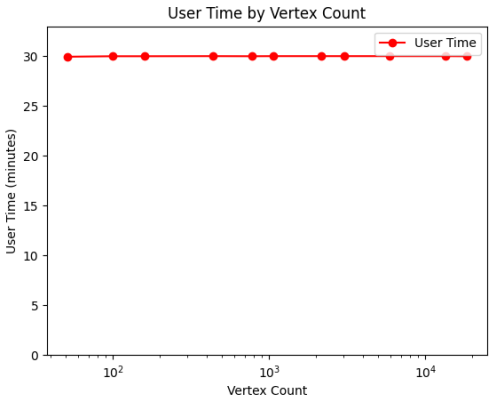
O algoritmo branch-and-bound não foi capaz de produzir a solução ótima nem no dataset de menor quantidade de vértices, que foi o eil51, com 51 vértices, então serão mostrados os dados de somente alguns datasets, que serão suficientes para entender o desempenho do algoritmo:

Em teoria, quanto maior a quantidade de vértices no branch-and-bound usando a BFS, maior o uso de memória, já que são guardados cada vez mais vetores de vértices escolhidos e eles tem tamanho máximo maior, mas como mostrado no gráfico acima, isto não ocorre aqui. O uso de memória chega em 3GB em quantidades pequenas de vértices, diminui para menos de 500MB e depois aumenta novamente.

O uso de memória nos dois extremos se dá por causas diferentes. No início, a memória é utilizada pelos vetores de vértices escolhidos, mas no final, essa memória é utilizada quase toda somente pela matriz de distâncias, que possui complexidade de



espaço O(n^2). No meio, ocorre esse uso baixo de memória por que a matriz de distâncias ainda não tem tamanho tão significativo, mas os vetores de vértices escolhidos, apesar de existirem mais deles, tem tamanho pequeno. Para uma comparação, se existem 10 vetores de vértices escolhidos em um dataset com 10 vértices, cada um dos vetores terá 1 vértice cada, mas se o dataset tiver somente 5 vértices, 5 vetores terão 1 vértice e outros 5 terão 2 vértices cada, ou seja, gasta 1,5 vezes a memória. Então, por causa disso, quanto maior a quantidade de vértices, menor a média do tamanho dos vetores que se consegue nos 30 minutos máximos de teste, então o espaço ocupado pelos vetores vai diminuindo, mas se não houvesse o limite de 30 minutos, rapidamente a solução utilizando uma BFS se tornaria inviável pelo uso de memória.



O tempo, por sua vez, é constante em 30 minutos, já que nenhum deles conseguiu terminar a execução e foram limitados a 30 minutos.

Sobre a qualidade da solução, como o branch-and-bound é um algoritmo para soluções exatas, seria a solução ótima, mas devido ao tempo limite de 30 minutos, nenhum dos datasets teve a solução dada.

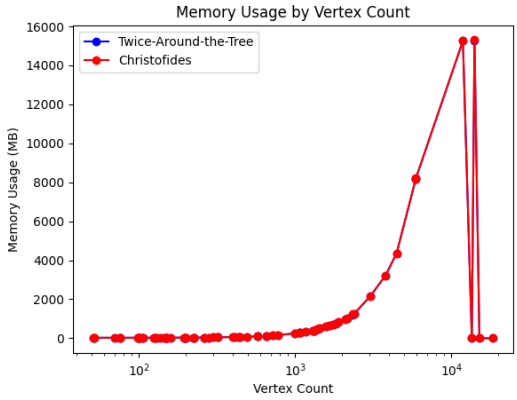
Abaixo temos uma tabela com as métricas analisadas, em qualidade da solução, como não foi possível obter nenhuma solução dentro dos 30 minutos, ficou como ”NA”:

Dataset	Vertex Count	Memory Usage (MB)	User Time (Min)	Quality
eil51	51	3778.45703125	29.95	NA
kroA100	100	2656.58984375	30.0	NA
u159	159	2051.6640625	30.0	NA
pr439	439	647.7734375	30.01	NA
rat783	783	186.2734375	30.0	NA
u1060	1060	253.73828125	30.01	NA
u2152	2152	135.98828125	30.01	NA
pcb3038	3038	156.8671875	30.01	NA
rl5934	5934	321.1015625	30.01	NA
usa13509	13509	1419.296875	30.01	NA
d18512	18512	2701.9140625	30.0	NA

Ambos os gráficos foram feitos com menos datasets que os próximos, mas os casos com os 78 datasets seguiram esse mesmo padrão.

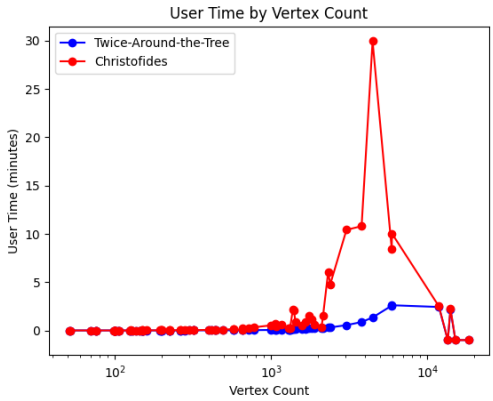
3.2. Aproximativos

Os aproximativos tiveram muito mais sucesso que o branch-and-bound, somente não conseguindo gerar soluções para os últimos datasets, necessitando de mais memória do que estava disponível no sistema.



É possível verificar que quantidades pequenas de vértices utilizam pouca memória, diferente do branch-and-bound com uma BFS, mas com o aumento da quantidade de vértices, o uso de memória aumenta de forma polinomial, ambos tendo o mesmo custo de espaço, então espaço não entra em consideração na hora de se escolher o algoritmo.

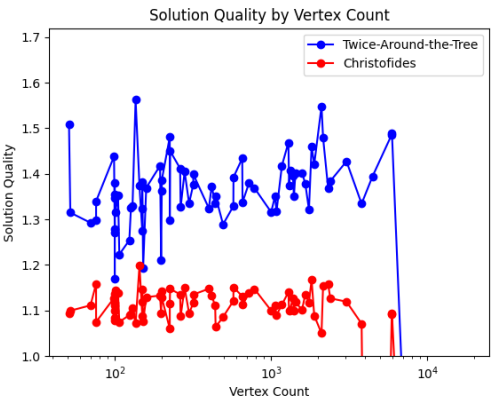
Também é possível verificar que no final, o uso de memória para de crescer com o aumento do número de vértices, isso se dá pois a memória utilizada alcançou o limite da memória disponível no sistema, não tendo como mensurar valores maiores. Além disso, alguns datasets tiveram valores em -1, isso foi por que nem todos os datasets com mais de 11 mil vértices conseguiram rodar e medir a memória utilizada, os que deram -1 não obtiveram resposta.



Para uma quantidade de vértices pequena, ambos algoritmos são bem eficientes, dando uma solução em no máximo 5 minutos, mas com uma quantidade de vértices chegando nos 3000, o algoritmo de christofides bem mais tempo, mas o tempo de execução oscila, enquanto o twice-around-the-tree não tem oscilações. só cresce aos poucos.

As oscilações no christofides são decorrentes da parte de encontrar um matching de peso mínimo, que para alguns datasets é facil encontrar, levando pouco tempo, com alguns executando o algoritmo de christofides inteiro no mesmo tempo que o twice-around-the-tree, mas para outros é difícil encontrar o matching, como o dataset ”fnl4461”, com 4461 vértices, não foi executado por completo no limite de 30 minutos.

No final do grafo, 2 pontos estão em menos de 5 minutos, já que levou poucotempo para chegar no limite de memória do sistema, e outros estão em -1, já que não conseguiu obter a resposta da memória ou do tempo utilizados, mas se tivesse a quantidade necessária de memória, com essas quantidades de vértices, possivelmente demoraria mais que 30 minutos para gerar a solução, dependendo do matching.



Em termos de qualidade de solução, o algoritmo de christofides é bem melhor, ficando com uma média por volta de 1,1 vezes pior que a solução ótima, enquanto o twice-around -the-tree ficou por volta de 1,4 vezes pior. Em geral, ambos dão soluções satisfatórias para um problema difícil.

No final do gráfico, é possível ver as linhas descendo abaixo de 1,0, isto ocorre pois para esses datasets os algoritmos não produziram solução. O de christofides desce

e sobe, antes de descer novamente, essa primeira descida é o dataset ”fnl4461”, que não terminou no limite de 30 minutos por causa da dificuldade de encontrar o matching.

Abaixo temos a tabela para o twice-around-the-tree:

Dataset	Vertex Count	Memory Usage (MB)	User Time (Min)	Quality
eil51	51	23.0546875	0.0	1.51
berlin52	52	23.2890625	0.0	1.31
st70	70	23.5546875	0.0	1.29
eil76	76	23.91796875	0.0	1.3
pr76	76	23.96484375	0.0	1.34
rat99	99	24.890625	0.0	1.44
kroA100	100	24.95703125	0.0	1.28
kroB100	100	24.94921875	0.0	1.17
kroC100	100	24.8828125	0.0	1.35
kroD100	100	25.01953125	0.0	1.27
kroE100	100	25.0234375	0.0	1.38
rd100	100	25.08984375	0.0	1.35
eil101	101	25.03515625	0.0	1.32
lin105	105	25.2578125	0.0	1.35
pr107	107	25.25	0.0	1.22
pr124	124	26.15625	0.0	1.25
bier127	127	26.39453125	0.0	1.33
ch130	130	26.4765625	0.0	1.33
pr136	136	26.71875	0.0	1.56
pr144	144	27.21875	0.0	1.37
ch150	150	27.46875	0.0	1.28
kroA150	150	27.6171875	0.0	1.32
kroB150	150	27.5078125	0.0	1.38
pr152	152	27.484375	0.0	1.19
u159	159	28.15625	0.0	1.37
rat195	195	31.3671875	0.0	1.42
d198	198	31.64453125	0.0	1.21
kroA200	200	31.94140625	0.0	1.36
kroB200	200	32.02734375	0.0	1.39
ts225	225	34.046875	0.0	1.48
tsp225	225	34.19921875	0.0	1.3
pr226	226	34.09375	0.0	1.45
gil262	262	38.05078125	0.0	1.41
pr264	264	38.18359375	0.0	1.33
a280	280	39.9375	0.0	1.4

Dataset	Vertex Count	Memory Usage (MB)	User Time (Min)	Quality
pr299	299	42.265625	0.01	1.34
lin318	318	44.6875	0.01	1.38
linhp318	318	44.76953125	0.01	1.4
rd400	400	59.953125	0.01	1.32
fl417	417	62.921875	0.01	1.37
pr439	439	67.0	0.01	1.34
pcb442	442	67.18359375	0.01	1.35
d493	493	77.0	0.02	1.29
u574	574	94.6484375	0.02	1.33
rat575	575	95.01953125	0.02	1.39
p654	654	114.328125	0.02	1.43
d657	657	114.97265625	0.02	1.34
u724	724	146.30078125	0.03	1.38
rat783	783	164.87109375	0.03	1.37
pr1002	1002	246.00390625	0.05	1.32
u1060	1060	269.91015625	0.07	1.35
vm1084	1084	280.5703125	0.07	1.32
pcb1173	1173	320.8828125	0.07	1.42
d1291	1291	379.375	0.08	1.47
rl1304	1304	385.90625	0.09	1.37
rl1323	1323	395.83203125	0.09	1.41
nrw1379	1379	475.078125	0.1	1.4
fl1400	1400	487.51171875	0.1	1.35
u1432	1432	506.4140625	0.1	1.4
fl1577	1577	595.76171875	0.12	1.4
d1655	1655	650.59375	0.15	1.38
vm1748	1748	717.05078125	0.19	1.32
u1817	1817	767.34765625	0.2	1.46
rl1889	1889	821.1171875	0.2	1.42
d2103	2103	993.8203125	0.25	1.55
u2152	2152	1037.02734375	0.28	1.48
u2319	2319	1188.21875	0.32	1.37
pr2392	2392	1257.7734375	0.31	1.38
pcb3038	3038	2171.6328125	0.55	1.43
fl3795	3795	3225.41796875	0.87	1.33
fnl4461	4461	4356.2265625	1.34	1.39
rl5915	5915	8162.4296875	2.57	1.49
rl5934	5934	8213.77734375	2.61	1.49
rl11849	11849	15253.8359375	2.43	NA
usa13509	13509	NA	NA	NA
brd14051	14051	15268.1953125	2.15	NA
d15112	15112	NA	NA	NA
d18512	18512	NA	NA	NA

Abaixo temos a tabela para o christofides:

Dataset	Vertex Count	Memory Usage (MB)	User Time (Min)	Quality
eil51	51	23.484375	0.0	1.09
berlin52	52	23.51171875	0.0	1.1
st70	70	24.12890625	0.0	1.11
eil76	76	24.265625	0.0	1.16
pr76	76	24.0390625	0.0	1.08
rat99	99	25.55078125	0.0	1.13
kroA100	100	25.640625	0.0	1.1
kroB100	100	25.41796875	0.0	1.08
kroC100	100	25.6015625	0.0	1.14
kroD100	100	25.2578125	0.0	1.11
kroE100	100	25.64453125	0.0	1.08
rd100	100	25.65625	0.0	1.12
eil101	101	25.83203125	0.0	1.14
lin105	105	25.80859375	0.0	1.14
pr107	107	25.9296875	0.0	1.08
pr124	124	26.37890625	0.0	1.09
bier127	127	27.3046875	0.01	1.09
ch130	130	27.09765625	0.0	1.1
pr136	136	27.2578125	0.0	1.07
pr144	144	27.30078125	0.0	1.2
ch150	150	28.34375	0.0	1.09
kroA150	150	28.5546875	0.01	1.12
kroB150	150	28.50390625	0.01	1.15
pr152	152	27.6796875	0.0	1.08
u159	159	28.82421875	0.01	1.13
rat195	195	31.63671875	0.01	1.13
d198	198	32.08203125	0.01	1.09
kroA200	200	33.20703125	0.01	1.14
kroB200	200	33.015625	0.01	1.13
ts225	225	34.2421875	0.0	1.06
tsp225	225	34.15625	0.01	1.11
pr226	226	34.328125	0.01	1.15
gil262	262	39.19921875	0.02	1.14
pr264	264	38.23046875	0.01	1.09
a280	280	39.86328125	0.01	1.15

Dataset	Vertex Count	Memory Usage (MB)	User Time (Min)	Quality
pr299	299	42.1328125	0.02	1.09
lin318	318	44.6484375	0.02	1.12
linhp318	318	44.73046875	0.02	1.14
rd400	400	60.9140625	0.06	1.15
fl417	417	62.7421875	0.03	1.13
pr439	439	66.765625	0.03	1.11
pcb442	442	67.1875	0.04	1.06
d493	493	77.08984375	0.08	1.09
u574	574	94.37109375	0.14	1.12
rat575	575	94.81640625	0.13	1.15
p654	654	114.1328125	0.03	1.13
d657	657	114.9375	0.19	1.11
u724	724	146.44921875	0.25	1.14
rat783	783	164.6796875	0.33	1.15
pr1002	1002	245.69140625	0.53	1.1
u1060	1060	269.9765625	0.73	1.11
vm1084	1084	280.375	0.4	1.09
pcb1173	1173	320.5859375	0.64	1.11
d1291	1291	379.6171875	0.22	1.14
rl1304	1304	386.62109375	0.21	1.1
rl1323	1323	396.0	0.21	1.1
nrw1379	1379	474.921875	2.2	1.13
fl1400	1400	487.5859375	2.11	1.1
u1432	1432	506.3984375	0.85	1.12
fl1577	1577	594.9296875	0.48	1.1
d1655	1655	651.76953125	0.91	1.14
vm1748	1748	716.74609375	1.5	1.12
u1817	1817	766.33203125	1.12	1.17
rl1889	1889	821.25	0.61	1.09
d2103	2103	993.51953125	0.32	1.05
u2152	2152	1036.921875	1.55	1.15
u2319	2319	1224.70703125	6.05	1.16
pr2392	2392	1258.87109375	4.79	1.13
pcb3038	3038	2171.1484375	10.42	1.12
fl3795	3795	3225.95703125	10.8	1.07
fnl4461	4461	4357.7421875	29.96	NA
rl5915	5915	8161.84375	8.42	1.09
rl5934	5934	8213.87890625	10.04	1.09
rl11849	11849	15259.48046875	2.5	NA
usa13509	13509	NA	NA	NA
brd14051	14051	15299.9296875	2.24	NA
d15112	15112	NA	NA	NA
d18512	18512	NA	NA	NA

4. Conclusões

O método branch-and-bound para o caixeiro viajante não é eficiente o bastante para ser utilizado para conseguir a solução ótima em grafos com dezenas de vértices em pouco tempo. O código deste trabalho não foi o mais eficiente possível, mas mesmo com um código muito eficiente, utilizando multi-threading, é improvável que o menor dataset utilizado, que contém apenas 51 vértices, retorne a solução ótima em menos de uma hora, se não um dia. Por isso, ele só é adequado em casos com bem poucos vértices, também sendo recomendado um alto grau de otimização.

Para o caso geral, utilizamos os algoritmos aproximativos, mais especificamente o de christofides. Ele pode ser utilizado sem nenhum problema até uns 2000 vértices, mas acima disso, ocorrem oscilações mais significativas no tempo de execução por causa do matching de custo mínimo, então se estabilidade no tempo de execução for uma prioridade, o twice-around-the-tree é recomendado.

Por fim, em casos com uma quantidade bem grande de vértices, o twice-around-the-tree leva significativamente menos tempo que o de christofides, então se quiser achar uma solução relativamente rápido, mas possivelmente pior, o twice-around-the-tree é mais recomendado. A estratégia de utilizar o twice-around-the-tree para quantidades muito grandes de vértices para obter uma solução temporária e então utilizar essa solução enquanto espera por uma solução no de christofides também é uma opção, já que se uma solução pelo algoritmo twice-around-the-tree demora um dia para ser encontrada, o de christofides pode demorar vários dias.

5. Execução

Cada algoritmo se encontra em um arquivo diferente. O branch-and-bound possui um diretório próprio, já que ele foi feito em c++.

Para executar o branch-and-bound, é necessario entrar no diretório pelo terminal e dar o comando make, que então compilará o programa e criará o arquivo ”run.out” no diretório ”bin”. Então, para executar o programa do diretório ”bin”, basta digitar o comando ./run.out ¡dataset¿ no terminal, onde ¡dataset¿ é o nome do arquivo do dataset, que deve estar no diretório ”bin” também. Assim, o programa será executado e imprimirá a solução ótima do problema no terminal.

Para executar os outros dois algoritmos, twice-around-the-tree e christofides, basta entrar no diretório onde eles se encontram pelo terminal e digitar python3 ¡algoritmo¿ ¡dataset¿, onde ¡algoritmo¿ é ”tatt.py” ou ”christofides.py” e ¡dataset¿ é o nome do arquivo do dataset, que também deve se encontrar no mesmo diretório. Assim, o programa será executado e imprimirá a solução aproximada no terminal.

6. References

References

[1] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/