Análise de desempenho de algoritmos aproximativos e exatos para o Problema do Caixeiro Viajante (TSP)

Gabriel Pains de Oliveira Cardoso[1]

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

gabrielpains@ufmg.br

Abstract. *This paper aims to discuss solutions for euclidean instances of the Traveling Salesman Problem (TSP). The following approximation algorithms were evaluated: Twice Around the Tree and Christofides. The Branch and Bound optimal algorithm for optimization problems was also evaluated. Moreover, analysis data such as elapsed time, memory usage and approximation ratio were collected for each test instance and the results were analyzed for each algorithm.*

Resumo. *Este trabalho explora diferentes soluções para instâncias euclideanas do Problema do Caixeiro Viajante (TSP). São testados os algoritmos aproximativos Twice Around the Tree e Christofides e o algoritmo exato de Branch and Bound. São medidas métrica de tempo de execução, memória e qualidade de solução para cada instância de teste, por fim os resultados obtidos são comparados entre si para cada algoritmo.*

1. Introdução

O problema do caixeiro viajante é um dos mais estudados no campo de teoria da complexidade e vários algoritmos aproximativos e exatos surgiram ao longo do tempo. O presente artigo tem por objetivo comparar três algoritmos para resolver instâncias euclidianas do problema, sendo dois aproximativos e um exato, são eles: *Twice Around the Tree*, *Christofides* e *Branch and Bound*, respectivamente.

São abordados os detalhes de implementação, as escolhas realizadas e os testes feitos para comparar os resultados obtidos, bem como as análises dos resultados.

2. Implementação

Cada algoritmo foi implementado de maneira a facilitar os testes e integração em um ambiente Python. Para os algoritmos aproximativos foi utilizado a biblioteca *networkx* [4] para manipulação de grafos, para o algoritmo exato nenhuma biblioteca especial de grafos foi utilizada.

Todos os grafos manipulados pelos algoritmos são grafos completos devido ao uso de instâncias euclidianas. Os algoritmos foram implementados com base nos pseudocódigos disponíveis em [2] e [3].

2.1. Twice Around The Tree

Neste algoritmo, o cálculo da árvore geradora mínima foi realizado pelo networkx. A extração do circuito inicial por DFS e a conversão para circuito hamiltoniano foram feitos por funções separadas customizadas. Isto porque os métodos fornecidos pelo networkx para tais tarefas não forneciam os resultados adequadas para este algoritmo em específico.

Para este algoritmo, o fator de aproximação no pior caso é 2. Sua complexidade é dominada pelo algoritmo de Kruskal usada no cálculo da árvore geradora mínima, sendo O($m * log(m)$), onde m é o número de arestas do grafo. Ainda, por ser um grafo completo a complexidade por ser reescrita como O($n^2 * log(n)$), onde n é o número de vértices do grafo, ou seja, o número de cidades.

Por fim, para sua execução, o código em Python foi convertido para C utilizando Cython e compilado a fim de criar um módulo que pode ser importado no ambiente Python onde os testes foram realizados. Este procedimento fornece um maior desempenho e garante que a execução do algoritmo é feita sem interferência do interpretador Python.

2.2. Christofides

Assim como no algoritmo anterior, o cálculo da árvore geradora mínima foi realizado pelo networkx. O cálculo do subgrafo induzido, do matching perfeito de peso mínimo e do circuito euleriano também foi realizada pelo networkx. Somente a conversão para circuito hamiltoniano foi feita por uma função separada customizada.

Para este algoritmo, o fator de aproximação no pior caso é 1.5. Sua complexidade é dominada pelo algoritmo de Blossom usada no cálculo do matching perfeito de peso mínimo, sendo O(n^3)), onde n é o número de vértices do grafo, ou seja, o número de cidades.

Por fim, para sua execução, o código em Python também foi convertido para C utilizando Cython [1] e compilado no mesmo módulo do algoritmo *Twice Around the Tree*.

2.3. Branch and Bound

Neste algoritmo foi utilizado uma estruturas de dados para cada nó que é visitado na busca. Esta estrutura contém os seguintes campos: estimativa de custo, o nível atual do nó na árvore, a última cidade visitada e as cidades na solução até o momento. A fim de economizar memória e tempo de execução, as cidades na solução são representadas por uma máscara de bits, o que garante interações O(1).

Ainda, para a estimativa de custo, são pré-computadas as duas menores arestas de cada nó e a estimativa de custo inicial para o nó raiz. Assim, atualizar a estimativa para qualquer nó tem custo O(1). A busca da árvore foi implementada como uma pilha, o que efetivamente implementa uma DFS iterativa. Isso foi feito para o algoritmo convergir a uma solução em um nó folha mais rapidamente e evitar um maior consumo de memória, o que seria o caso da implementação por fila de prioridade.

A execução do algoritmo começa com a melhor distância encontrada pelo algoritmo de *Christofides*, o que leva a vários cortes de ramos da árvore de busca já no início da execução.

Por fim, o código deste algoritmo foi feito em C++ com o uso da biblioteca *pybind11* [5] para criação de um módulo para Python a partir do código C++ compilado. Apesar de todas estas modificações e otimizações, a complexidade de tempo da busca ainda é O($n!$) no pior caso, já que dependendo do grafo o algoritmo pode ou não cortar muitos ramos do espaço de busca.

3. Metodologia de Testes

Como detalhado na seção anterior, todos os códigos dos algoritmos foram compilados para módulos importáveis em Python. A biblioteca *memory_profiler* foi utilizada para

medir o consumo de memória máximo dos algoritmos e a biblioteca *time* foi utilizada para medir o tempo. Para cada teste o algoritmo testado possui um limite de 30 minutos para finalizar sua execução, no caso do *Branch and Bound*, um *timeout* que aborta a execução da busca após os 30 minutos foi implementado diretamente no código C++.

As métricas coletadas foram: tempo de execução, memória máxima e qualidade da solução. Os algoritmos foram executados em todas as 78 instâncias euclidianas do TSPLIB 95 [6]. A partir da 73º instância (mais de 6000 cidades) todos os algoritmos ficaram sem memória e não finalizaram sua execução. Em específico para o *Branch and Bound*, nenhuma instância finalizou sua execução em 30 minutos e algumas em específico apresentaram erro de memória máxima excedida.

A seção seguinte apresenta e discute os resultados obtidos, as tabelas de métricas para cada algoritmo aproximativo entre as 78 instâncias estão presentes nos **Apêndices A e B**. Não há tabela para o *Branch and Bound* devido a ausência de dados, pois o algoritmo não executou em nenhuma instância conforme supracitado.

4. Experimentos e Resultados

Foram elaborados gráficos demonstrando o comportamento dos algoritmos aproximativos para cada métrica em relação ao número de nós ou instância de testes. Os algoritmos analisados são os aproximativos, já que conforme citado na seção anterior, o *Branch and Bound* não finalizou sua execução para nenhuma instância de teste.

4.1. Tempo de Execução

O seguintes gráficos mostram em azul o tempo gasto pelo *Twice Around the Tree* e em verde o tempo gasto pelo *Christofides*.

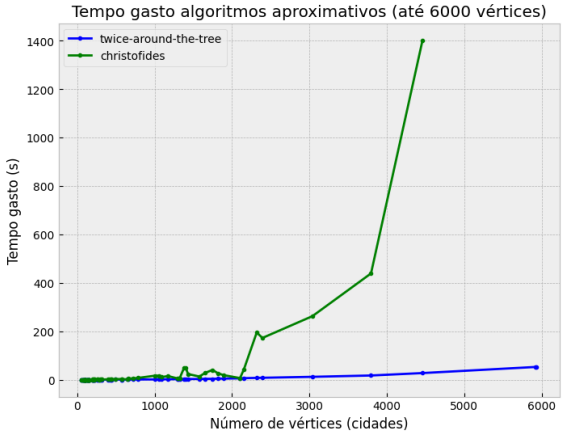

Figura 1. Tempo gasto em todas as instâncias testadas

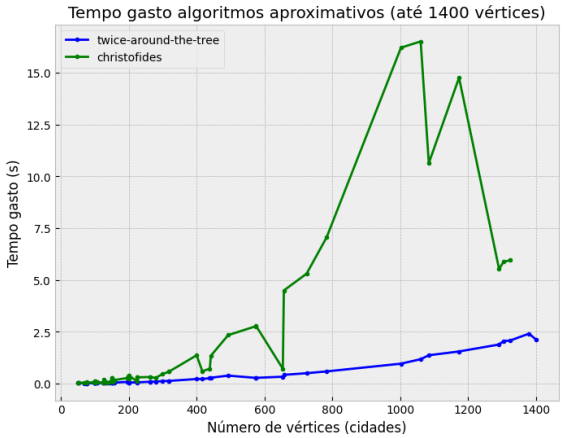

Figura 2. Tempo gasto nas instâncias de até 1400 vértices

É possível ver na **Figura 2** que até 200 vértices o tempo gasto pelos dois algoritmos é similar. A partir de 200 vértices o algoritmo de *Christofides* começa a exibir um tempo maior de execução. Ainda, na **Figura 1** que mostra o panorama do tempo de execução para todas as instâncias é possível ver a disparidade de tempo do algoritmo de Christofides se torna muito maior, em especial para a instância **fnl4461** de 4461 vértices o *Twice Around the Tree* gastou 27 segundos ao passo que o *Christofides* gastou 1400 segundos.

Isto é, o algoritmo começa a mostrar a sua complexidade O(n^3) decorrente do algoritmo de Blossom nas instâncias maiores levando a um tempo de execução muito maior do que o algoritmo *Twice Around the Tree* que possui complexidade O($n^2 * log(n)$) como mencionado na seção 2.

4.2. Memória Máxima Utilizada

O seguinte gráfico mostra em azul a memória máxima utilizada pelo *Twice Around the Tree* e em verde a memória máxima utilizada pelo *Christofides*.

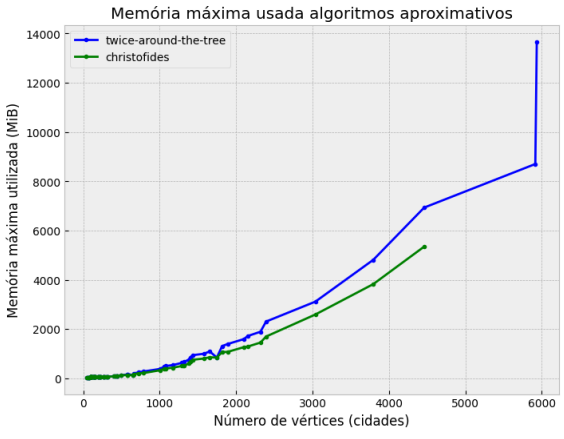

Figura 3. Evolução da memória utilizada pelos algoritmos aproximativos

É possível ver na **Figura 3** que a memória utilizada pelos algoritmos é bem similar até 1500 vértices, a partir desse ponto o *Twice Around the Tree* começa a utilizar um pouco mais de memória e para a instância de quase 6000 vértices, este utiliza quase 14000 MiB de memória.

Os dois algoritmos realizam passos similares e/ou que não gastam muita memória auxiliar, porém o *Twice Around the Tree* anda na árvore salvando os nós do caminho na ida e na volta da DFS, o que implica na quantidade de nós salvos durante todo o trajeto ser maior do que as arestas salvas durante o percurso do circuito euleriano no *Christofides*. Essa variação leva a diferença de memória observada mesmo que os passos tomadas pelos algoritmos sejam similares.

4.3. Qualidade da Solução

O seguinte gráfico mostra em azul a qualidade da solução do *Twice Around the Tree* e em verde a qualidade da solução do *Christofides*. Além das respectivas médias em cores mais claras.

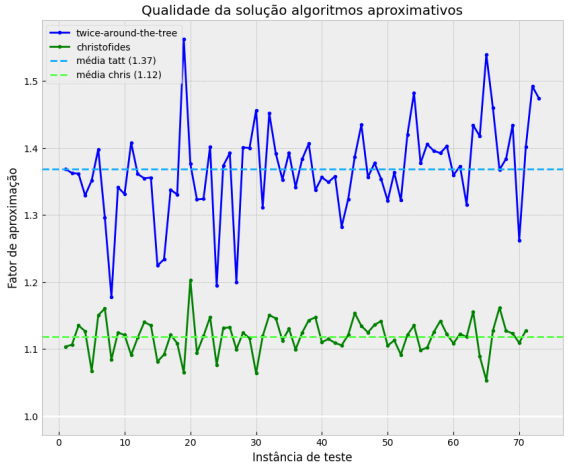

Figura 4. Razão da solução encontrada para com o ótimo

É possível ver pela **Figura 4**, que independente da instância utilizada o algoritmo de *Christofides* possui fator de aproximação melhor do que o *Twice Around the Tree*. Ainda, a variância do fator de aproximação do *Christofides* foi menor já que a melhor aproximação teve fator 1.05 e a pior aproximação teve fator 1.2. Para o *Twice Around the Tree*, a melhor aproximação teve fator 1.18 e a pior teve fator 1.56 (ver Apêndices A e B).

5. Conclusões

Após os experimentos, é possível ver que a execução do algoritmo de *Branch and Bound* pode se tornar inviável dependendo do tempo disponível para execução do problema, mesmo que esse forneça a solução ótima. Não somente, mesmo se tempo não for um problema, pode ser que a instância seja tão grande que a memória demandada pela árvore de busca também torna seu uso inviável.

Os algoritmos aproximativos por outro lado fornecem soluções próximas mas executam em tempos mais razoáveis e que podem ser mais adequados para a tarefa em questão. Quanto a qualidade da solução o algoritmo de *Christofides* é impressionante, apesar de seu pior caso ser 1.5 aproximado, a média das aproximações foi 1.12, ou seja, em média o algoritmo foi 12% pior que ótimo sendo executado em tempo polinomial. O *Twice Around the Tree* apresentou uma média maior de 1.37, também executa em tempo polinomial mas é mais rápido que o *Christofides*. Dessa forma, se o limite de tempo for ainda mais restrito é possível escolher esse algoritmo em detrimento da qualidade da solução.

Referências

[1] CYTHON DEVELOPMENT TEAM (Estados Unidos). **Source Files and Compilation**. 2023. Disponível em: http://docs.cython.org/en/latest/src/userguide/source_files_and_compilation.html. Acesso em: 07 dez. 2023.

[2] KLEINBERG, Jon; TARDOS, Eva. **Algorithm Design**. 2. ed. Ithaca: Pearson, 2022. 984 p.

[3] LEVITIN, Anany. **Introduction to the Design and Analysis of Algorithms**. 3. ed. Villanova: Pearson, 2011. 600 p.

[4] NETWORKX DEVELOPMENT TEAM. **Software for complex networks**. 2023. Disponível em: https://networkx.org/. Acesso em: 07 dez. 2023.

[5] PYBIND11 DEVELOPMENT TEAM (Estados Unidos). **Building with setuptools**. 2023. Disponível em: https://pybind11.readthedocs.io/en/stable/compiling.html#building-with-setuptools. Acesso em: 08 dez. 2023.

[6] GERHARD REINELT (Alemanha). Universität Heidelberg. **TSPLIB**. 1995. Disponível em: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Acesso em: 06 dez. 2023.

Apêndice A

Tabela 1. Resultados para o algoritmo *Twice Around the Tree*

Instância	Tempo (s)	Memória máx. (MiB)	Razão	Resultado	Ótimo
eil51	0.02	37.21	1.37	583	426
berlin52	0.02	37.79	1.36	10276	7542
st70	0.01	38.46	1.36	919	675
eil76	0.02	39.28	1.33	715	538
pr76	0.02	39.28	1.35	146155	108159
rat99	0.02	40.66	1.40	1693	1211
kroA100	0.02	42.71	1.30	27584	21282
kroB100	0.02	43.34	1.18	26073	22141
kroC100	0.02	43.35	1.34	27827	20749
kroD100	0.02	43.41	1.33	28346	21294
kroE100	0.02	43.59	1.41	31067	22068
rd100	0.02	43.59	1.36	10765	7910
eil101	0.02	43.63	1.35	852	629
lin105	0.02	43.73	1.36	19495	14379
pr107	0.02	43.77	1.22	54237	44303
pr124	0.03	44.80	1.23	72829	59030
bier127	0.03	44.80	1.34	158205	118282
ch130	0.03	44.81	1.33	8129	6110
pr136	0.04	46.64	1.56	151238	96772
pr144	0.05	46.81	1.38	80552	58537
ch150	0.06	46.81	1.32	8636	6528
kroA150	0.04	46.69	1.32	35119	26524
kroB150	0.04	46.69	1.40	36625	26130
pr152	0.04	47.87	1.19	87995	73682
u159	0.05	47.90	1.37	57791	42080
rat195	0.08	51.02	1.39	3234	2323
d198	0.08	50.29	1.20	18936	15780
kroA200	0.05	50.41	1.40	41133	29368
kroB200	0.05	50.44	1.40	41209	29437
ts225	0.06	51.96	1.46	184426	126643
tsp225	0.06	52.95	1.31	5140	3919
pr226	0.06	54.03	1.45	116650	80369
gil262	0.09	54.39	1.39	3308	2378
pr264	0.09	54.62	1.35	66470	49135
a280	0.09	55.91	1.39	3592	2579
pr299	0.12	68.69	1.34	64645	48191
lin318	0.13	61.65	1.38	58138	42029
linhp318	0.13	60.67	1.41	58138	41345
rd400	0.22	76.12	1.34	20431	15281
fl417	0.22	79.00	1.36	16082	11861
pr439	0.25	83.59	1.35	144623	107217

Continua na próxima página

Instância	Tempo (s)	Memória máx. (MiB)	Razão	Resultado	Ótimo
pcb442	0.28	98.87	1.36	68940	50778
d493	0.38	114.98	1.28	44882	35002
u574	0.27	153.23	1.32	48849	36905
rat575	0.27	165.57	1.39	9393	6773
p654	0.33	133.52	1.43	49699	34643
d657	0.42	195.98	1.36	66339	48912
u724	0.50	237.85	1.38	57721	41910
rat783	0.59	272.59	1.35	11921	8806
pr1002	0.96	376.16	1.32	342216	259045
u1060	1.17	469.55	1.36	305496	224094
vm1084	1.36	496.38	1.32	316286	239297
pcb1173	1.55	535.43	1.42	80761	56892
d1291	1.88	624.54	1.48	75282	50801
rl1304	2.05	652.85	1.38	348373	252948
rl1323	2.07	669.34	1.41	379775	270199
nrw1379	2.41	753.00	1.40	79047	56638
fl1400	2.12	851.09	1.39	28022	20127
u1432	2.53	937.08	1.40	214569	152970
fl1577	2.78	993.05	1.36	30182	22204
d1655	3.43	1089.13	1.37	85283	62128
vm1748	3.48	839.12	1.32	442755	336556
u1817	4.19	1304.88	1.43	82013	57201
rl1889	4.52	1389.31	1.42	448663	316536
d2103	5.26	1590.73	1.54	123085	79952
u2152	6.16	1714.15	1.46	93790	64253
u2319	7.31	1888.43	1.37	320310	234256
pr2392	7.74	2301.61	1.38	523107	378032
pcb3038	11.71	3112.16	1.43	197477	137694
fl3795	17.20	4808.78	1.26	36256	28723
fnl4461	27.24	6931.55	1.40	255829	182566
rl5915	52.32	8700.80	1.49	842903	565040
rl5934	50.56	13671.25	1.47	816642	554070
rl11849	NA	NA	NA	NA	920847
usa13509	NA	NA	NA	NA	19947008
brd14051	NA	NA	NA	NA	468942
d15112	NA	NA	NA	NA	1564590
d18512	NA	NA	NA	NA	644650

Apêndice B

Tabela 2. Resultados para o algoritmo *Christofides*

Instância	Tempo (s)	Memória máx. (MiB)	Razão	Resultado	Ótimo
eil51	0.03	37.63	1.10	470	426
berlin52	0.03	38.00	1.11	8343	7542
st70	0.05	39.19	1.13	766	675
eil76	0.06	39.28	1.13	606	538
pr76	0.04	39.28	1.07	115425	108159
rat99	0.06	41.09	1.15	1393	1211
kroA100	0.07	43.09	1.16	24690	21282
kroB100	0.06	43.34	1.08	24001	22141
kroC100	0.07	43.35	1.12	23328	20749
kroD100	0.07	43.45	1.12	23864	21294
kroE100	0.09	43.59	1.09	24073	22068
rd100	0.11	43.59	1.12	8834	7910
eil101	0.10	43.73	1.14	717	629
lin105	0.07	43.73	1.13	16320	14379
pr107	0.06	43.88	1.08	47891	44303
pr124	0.05	44.80	1.09	64438	59030
bier127	0.19	44.80	1.12	132584	118282
ch130	0.10	44.81	1.11	6773	6110
pr136	0.05	46.64	1.06	103027	96772
pr144	0.07	46.81	1.20	70404	58537
ch150	0.12	46.64	1.09	7143	6528
kroA150	0.25	46.69	1.12	29688	26524
kroB150	0.26	46.69	1.15	29981	26130
pr152	0.06	47.90	1.08	79311	73682
u159	0.17	47.64	1.13	47586	42080
rat195	0.26	50.91	1.13	2630	2323
d198	0.33	50.20	1.10	17347	15780
kroA200	0.23	50.14	1.12	33004	29368
kroB200	0.40	50.59	1.12	32851	29437
ts225	0.10	51.85	1.06	134702	126643
tsp225	0.32	52.96	1.12	4386	3919
pr226	0.30	54.22	1.15	92446	80369
gil262	0.32	53.92	1.15	2724	2378
pr264	0.30	54.55	1.11	54662	49135
a280	0.28	54.90	1.13	2914	2579
pr299	0.45	68.69	1.10	52969	48191
lin318	0.57	61.65	1.12	47246	42029
linhp318	0.57	60.67	1.14	47246	41345
rd400	1.36	76.12	1.15	17527	15281
fl417	0.59	79.00	1.11	13166	11861
pr439	0.72	83.54	1.11	119525	107217

Continua na próxima página

Instância	Tempo (s)	Memória máx. (MiB)	Razão	Resultado	Ótimo
pcb442	1.34	98.87	1.11	56303	50778
d493	2.34	114.98	1.11	38692	35002
u574	2.76	133.00	1.12	41360	36905
rat575	2.79	119.77	1.15	7811	6773
p654	0.71	133.40	1.13	39292	34643
d657	4.49	161.07	1.12	55005	48912
u724	5.30	200.96	1.14	47601	41910
rat783	7.05	208.04	1.14	10052	8806
pr1002	16.21	322.04	1.10	286203	259045
u1060	16.51	378.15	1.11	249341	224094
vm1084	10.64	394.64	1.09	261199	239297
pcb1173	14.76	429.11	1.12	63767	56892
d1291	5.52	497.10	1.13	57655	50801
rl1304	5.87	499.29	1.10	277772	252948
rl1323	5.95	514.80	1.10	297715	270199
nrw1379	48.13	594.51	1.12	63710	56638
fl1400	49.50	673.24	1.14	22980	20127
u1432	23.18	749.36	1.12	171615	152970
fl1577	12.90	800.18	1.11	24606	22204
d1655	28.38	851.18	1.12	69719	62128
vm1748	40.09	839.64	1.12	376203	336556
u1817	26.54	1062.68	1.16	66103	57201
rl1889	18.74	1070.01	1.09	344568	316536
d2103	6.50	1263.86	1.05	84205	79952
u2152	41.69	1285.95	1.13	72422	64253
u2319	195.02	1448.38	1.16	272116	234256
pr2392	172.29	1693.12	1.13	425940	378032
pcb3038	262.29	2589.79	1.12	154651	137694
fl3795	438.85	3823.00	1.11	31850	28723
fnl4461	1402.74	5346.15	1.13	205811	182566
rl5915	NA	NA	NA	NA	565040
rl5934	NA	NA	NA	NA	554070
rl11849	NA	NA	NA	NA	920847
usa13509	NA	NA	NA	NA	19947008
brd14051	NA	NA	NA	NA	468942
d15112	NA	NA	NA	NA	1564590
d18512	NA	NA	NA	NA	644650