Análise de Desempenho na Resolução do TSP

Átila A. Carvalho

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
CEP – 31.270-901 – Belo Horizonte – MG – Brazil

Resumo. *Este artigo analisa o desempenho comparado de 3 algoritmos para resolver o problema do caixeiro viajante (TSP), sendo desses algoritmos, 2 aproximativos e 1 exato.*

1. Objetivo

Nesse trabalho, o algoritmo Twice-Around-the-Tree, o algoritmo de Christofides e o algoritmo de Branch-and-Bound foram usados em um conjunto de instâncias TSP. O objetivo desse trabalho é comparar o custo-benefício entre 2 soluções aproximadas polinomiais e 1 solução exata exponencial para resolver esses problemas.

2. Metodologia

Os 3 algoritmos discutidos aqui eram executados sobre um mesmo grafo, criado uma única vez com o auxilio de uma biblioteca chamada networkx.
Para cada instância do TSP avaliada, um único comando era chamado e a partir desse comando o programa criava o grafo completo representando o TSP e então rodava o Twice-Around-The-Three, o algoritmo de Christofides e o Branch-and-Bound em sequência em um período de 30 minutos.
Os gráficos ilustrando os resultados desse trabalho estão no notebook em anexo (não consegui coloca-los no arquivo Latex devido ao tamanho das imagens).

3. Conjunto de teste

O conjunto de teste consistiu em 78 problemas públicos, disponíveis no site da TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/) em que o peso de cada aresta era a distância euclidiana 2D entra as coordenadas dos vértices.

4. Execução

O trabalho vai com um makefile. O comando "make run" executa o código para todas as instâncias de teste na sub-pasta Data.
Após executar, "tp2.py" cria um arquivo de log para cada instância em uma sub-pasta chamada results. As 2 primeiras linhas desse log são respectivamente saídas dos algoritmos Twice-Around-the-Tree e Christofides, e contém um ciclo hamiltoniano, o valor desse ciclo e o gasto em memória do algoritmo que encontrou esse ciclo. A terceira linha é referente ao Branch-and-Bound, e tem o horário em que o algoritmo começou, o horário em que ele achou o melhor caminho hamiltoniano encontrado, o custo desse caminho e o tamanho máximo da pilha. Essa última informação pode estar incorreta, mas isso não atrapalhou o trabalho pois como será argumentado na análise de memória, essa informação pode ser calculada sem executar o programa apenas analisando o código.
2 instâncias são especialmente problemáticas.

4.1. linhp318.tsp

Essa instância não segue o padrão das outras, ela é a única em que os dados começam na linha 10, enquanto em todas as outras instâncias eles começam na linha 7, por isso o código fonte em ”tp2.py” não funciona corretamente nela, uma versão modificada foi preparada apenas para ler essa instância e está em um arquivo chamado ”tp2_linhp318.py”. Para executar, basta usar o comando ”timeout 30m python3 tp2_linhp318.py Data/linhp318.tsp”.

4.2. r1002.tsp

Ao contrário das outras instâncias, essa não tinha o arquivo terminando em ’EOF’ seguida de uma quebra de linha ”.
Eu apenas adicionei essa linha ao final da instância para que ”tp2.py” funcione com ela.

5. Análise da Precisão

Algumas instâncias se mostraram tão grandes que os 30 minutos não foram o suficiente nem mesmo para construir o grafo. Nessa sessão estão apresentados os resultados obtidos por cada algoritmo sobre as instâncias em que cada algoritmo teve tempo para executar. Os gráficos dos resultados estão no notebook em anexo.

5.1. Twice-Around-the-Tree (TAtT)

O algoritmo TAtT terminou de executar em 73 das 78 instâncias.
Apesar desse algoritmo ter um fator-de-aproximação igual a 2, o fator de aproximação médio calculado para essas instâncias foi 1.3913441459839382, o que é bem melhor que o mínimo garantido teoricamente.

5.2. Christofides (Ctf)

O algoritmo Ctf terminou de executar em 72 das 78 instâncias. A única instância que foi possível para TAtT e impossível para Ctf foi fnl4461.
Esse algoritmo tem um fator-de-aproximação igual a 1.5, e o fator de aproximação médio calculado para essas instâncias foi 1.369239720909113, o que é melhor que o fator médio computado do TAtT e um pouco melhor que o mínimo garantido teoricamente.
Esse algoritmo não é inerentemente incapaz de calcular o valor aproximado do TSP para entradas com aproximadamente 4500 em tempo hábil, o desempenho apresentado aqui também é devido ao fato de ter compartilhado os 30m com a execução do TAtT. Como já foi falado antes, conforme o tamanho da entrada cresce, até mesmo o esforço para construir o grafo se torna relevante.

5.3. Branch-and-Bound (BaB)

A execução do algoritmo de BaB foi possível sobre 50 instâncias.
Esse baixo número é esperado, como será mostrado a seguir na análise de memória, a implementação do algoritmo gasta mais tempo pois a cada iteração tem que expandir todos as arestas de um vértice, ou seja, mesmo para encontrar o primeiro mínimo local o algoritmo BaB gasta um tempo quadrático. Além disso, o tempo em que o BaB executou foi o que sobrou após o TAtT e o Ctf executarem, e como já visto mesmo esses algoritmos polinomiais já começam a gastar um tempo considerável em instâncias da ordem de 4000

vértices.
O valor encontrado pelo BaB nessas 50 instâncias foi em média 1.2127426940335475 do valor ótimo, o que é melhor ainda que a média do algoritmo de Christofides.

6. Análise de memória

Para simplificar a análise de memória, cada vértice e aresta criado por um algoritmo será contado como 1.
O TAtT cria uma árvore geradora mínima, portanto ele cria n vértices e n-1 arestas.
O Ctf cria uma multígrafo de uma árvore geradora mínima e um matching de tamanho variável. Não há fórmula fechada para quantas arestas ele vai criar, portanto seu custo de memória foi computado em tempo de execução.
O BaB foi implementado por meio de uma chamada recursiva, para encontrar um mínimo local o BaB precisa de n+1 funções na pilha, a chamada que inicia o algoritmo e mais 1 chamada recursiva para cada um dos n elemento do ciclo hamiltoniano. Em cada uma dessas chamadas recursivas, são expandidas n-1 arestas, portando o algoritmo BaB como implementado gastou n^2-n+1 em memória.
A seguir, a tabela com tais resultados:

”’

Problema	Tamanho	Memória TAtT	Memória Ctf	Memória BaB
eil51	51	101	214	2551
berlin52	52	103	217	2653
st70	70	139	295	4831
eil76	76	151	322	5701
pr76	76	151	315	5701
rat99	99	197	414	9703
kroA100	100	199	420	9901
kroB100	100	199	416	9901
kroC100	100	199	420	9901
kroD100	100	199	419	9901
kroE100	100	199	423	9901
rd100	100	199	424	9901
eil101	101	201	429	10101
lin105	105	209	437	10921
pr107	107	213	449	11343
pr124	124	247	512	15253
bier127	127	253	539	16003
ch130	130	259	542	16771
pr136	136	271	560	18361
pr144	144	287	587	20593
ch150	150	299	624	22351
kroA150	150	299	634	22351
kroB150	150	299	634	22351
pr152	152	303	621	22953
u159	159	317	668	25123
rat195	195	389	815	37831
d198	198	395	832	39007
kroA200	200	399	844	39801
kroB200	200	399	841	39801
ts225	225	449	914	50401
tsp225	225	449	941	50401
pr226	226	451	938	50851
gil262	262	523	1104	68383
pr264	264	527	1095	69433
a280	280	559	1176	78121
pr299	299	597	1255	89103
lin318	318	635	1333	100807
linhp318	318	635	1333	100807
rd400	400	799	1689	159601
fl417	417	833	1735	173473
pr439	439	877	1828	192283
pcb442	442	883	1863	194923
d493	493	985	2070	242557
u574	574	1147	2414	328903
rat575	575	1149	2427	330051

Table 1. Tabela de Complexidade de Espaço - Pt1

Problema	Tamanho	Memória TAtT	Memória Ctf	Memória BaB
p654	654	1307	2680	427063
d657	657	1313	2760	430993
u724	724	1447	3047	523453
rat783	783	1565	3309	612307
pr1002	1002	2003	4239	X
u1060	1060	2119	4465	1122541
vm1084	1084	2167	4510	X
pcb1173	1173	2345	4905	X
d1291	1291	2581	5319	X
rl1304	1304	2607	5342	X
rl1323	1323	2645	5421	X
nrw1379	1379	2757	5836	X
fl1400	1400	2799	5966	X
u1432	1432	2863	6027	X
fl1577	1577	3153	6520	X
d1655	1655	3309	6899	X
vm1748	1748	3495	7272	X
u1817	1817	3633	7570	X
rl1889	1889	3777	7752	X
d2103	2103	4205	8525	X
u2152	2152	4303	8981	X
u2319	2319	4637	9548	X
pr2392	2392	4783	10073	X
pcb3038	3038	6075	12723	X
fl3795	3795	7589	15865	X
fnl4461	4461	8921	X	X
rl5915	5915	11829	24126	X
rl5934	5934	11867	24247	X
rl11849	11849	X	X	X
usa13509	13509	X	X	X
brd14051	14051	X	X	X
d15112	15112	X	X	X
d18512	18512	X	X	X

Table 2. Tabela de Complexidade de Espaço - Pt2

7. Análise de Tempo

7.1. Twice-Around-the-Tree (TAtT)

O algoritmo TAtT é sub-cúbico, pois ele roda o algoritmo de Prim que é $O(|E|log|V|)$ sobre um grafo completo, e como um grafo completo tem $|E| = (|V|^2 - |V|)/2$, então nesse caso o algoritmo de Prim roda em $O(|V|^2 log|V|)$. Ele também roda o dfs sobre uma árvore, nesse caso o algoritmo dfs é $O(|V|)$ e essa complexidade é dominada pelo algoritmo de Prim.

7.2. Christofides (Ctf)

O algoritmo Ctf é cúbico, pois envolve executar o algoritmo de Prim (cuja complexidade já foi estudada), o algoritmo dfs sobre um multigrafo ($O(|E| + |V|)$), e o algoritmo de Blossom ($O(|E| * |V|^2)$).
O algoritmo de Blossom domina a complexidade do Ctf e é a grande diferença entre o TAtT e o Ctf. Instâncias de quase 6000 vértices tiveram tempo para terminar enquanto o Ctf enquanto a fnl4461 não conseguiu, e isso provavelmente se deve ao algoritmo Blossom que tendo uma dificuldade maior de encontrar um matching extremo, afinal no pior caso o algoritmo é cúbico, mas em casos favoráveis ele pode terminar mais rápido.

7.3. Branch-and-Bound (BaB)

O algoritmo BaB é exponencial, e embora fosse esperado a princípio que ele encontrasse o ótimo para as menores instâncias em tempo hábil, isso não aconteceu.
Os motivos para isso podem ter sido a implementação que gastava tempo quadrático expandindo arestas (nem sempre úteis) em qualquer caminho entre a raiz e alguma folha da árvore de busca.
Mesmo rodando com tempo insuficiente, o algoritmo de BaB conseguiu encontrar valores melhores que os outros 2 algoritmos para as instâncias em que ele encontrou algum caminho hamiltoniano.
Como o algoritmo é quadrático para ir da raiz a uma folha qualquer, uma implementação mais eficiente dele é útil para qualquer instância em que TAtT ou Ctf conseguiram aproximar com facilidade.

8. Conclusões

Esse trabalho mostrou a comparação entre os resultados de diferentes algoritmos buscando o ótimo de um problema NP-Difícil (TSP).
Com esses resultados e uma instância do TSP em mãos, é possível começar a planejar qual algoritmo escolher para resolver TSP com base no tamanho da instância e na precisão requerida.
Alguns pontos importantes ao tentar melhorar o que foi feito aqui seriam otimizar o uso de memória do algoritmo Branch-And-Bound, mudar para uma biblioteca de grafos e uma linguagem mais focadas em eficiência, e dependendo do tamanho da instância, usar o algoritmo de Hopcroft–Karp($O(|E| * \sqrt{|V|})$ para matching máximo ao executar o Algoritmo de Christofides.