Trabalho Prático 2
Soluções para Problemas Difíceis

Rita R B. de Lima[1]

[1]Universidade Federal de Minas Gerais (UFMG)

ritaborgesdelima@dcc.ufmg.br

Abstract. *This work addresses practical aspects of algorithms that solve difficult problems. Specifically, an exact solution, based on Branch and Bound, and two approximate solutions to the Traveling Salesman Problem, Twice Around the Tree and Christofides' algorithm.*

Resumo. *O trabalho implementado aborda aspectos práticos de algoritmos para solucionar problemas difíceis. Especificamente, uma solução exata, baseada em Branch and Bound, e duas soluções aproximadas para o Problema do Caixeiro Viajante, Twice Around the Tree e algoritmo de Christofides.*

1. Introdução - O Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante retrata a necessidade de um vendedor de visitar n cidades percorrendo a menor distância possível. Modelando as cidades e suas respectivas distâncias em um grafo ponderado $G = (v, e)$ onde cada vértice é uma das n cidades e suas respectivas distâncias são representadas pelos pesos das arestas, é possível dizer que o vendedor deseja realizar um Ciclo Hamiltoniano, visitando cada cidade apenas uma vez e terminando o percurso na cidade inicial.

Podemos atribuir um custo total para esse respectivo caminho que consiste da soma de cada uma das e arestas utilizadas. O vendedor deseja percorrer a menor distância possível, logo quer percorrer o caminho de menor custo. [Cormen et al. 1990]

O Caixeiro Viajante é um problema de otimização NP-difícil. Neste trabalho serão abordados algoritmos aproximativos que buscam uma solução próxima em tempo polinomial além de uma solução ótima utilizando a técnica *Branch and Bound* que por sua vez possui como pior caso complexidade exponencial igual a implementação força bruta, contudo na prática esta consiga se sair de forma mais eficiente.

Todos os algoritmos foram implementandos por meio de *Python 3.7* utilizando as bibliotecas: *Time, Pandas, Numpy, Networkx e Matplotlib*. O código pode ser acessado pelo seguinte link:
`https://github.com/RitaRez/Travelling-Salesman-Problem`.

2. Implementação

2.1. A classe grafo

Para a manipulação de grafos e algoritmos relacionados a biblioteca *Networkx* foi utilizada. O módulo usado para constuir a estrutura grafo foi *soft random geometric graph*, este recebe os seguintes parâmetros:

- O número de nós aleatórios a serem gerados, quantidade que na implementação do trabalho variou de 2^4 até 2^{10}.
- Uma distância que define até quando duas cidades estão conectadas. Como na modelagem do trabalho o vendedor pode ir de uma cidade para qualquer outra, essa distância é definida como maior do que o mapa.
- A função de probabilidade que define se duas cidades estão conectadas, como todas as cidades devem ser conectadas essa é definida como constante em um.
- Uma *seed* para a geração aleatória do grafo. Esta é sempre mantida como um para que o mesmo grafo possa ser gerado para ser utilizado nos diferentes algoritmos e nas diferentes métricas.

O módulo por sua vez retorna uma grafo onde cada nó é composto por um índice único e uma tupla representando as coordenadas x, y da cidade. Todas as cidades são conectadas entre si por arestas e os pesos destas são calculados em seguida utilizando as funções de distância euclidiana e manhattan.

2.2. Twice Around the Tree

O primeiro algoritmo aproximativo implementado para o trabalho foi o algoritmo *Twice Around the Tree*, este funciona da seguinte maneira: primeiro é construída uma árvore geradora mínima do grafo, a partir de um vértice arbitrário deste, caminhe ao redor da árvore com uma busca em profundidade registrando todos os vértices. Elimine da lista de vértices obtida todas as ocorrências repetidas do mesmo vértice, exceto o vértice inicial e final. Os vértices restantes na lista representam o passeio aproximado e o custo do caminho pode ser obtido iterando sobre cada um destes vértices acumulando a soma das arestas de cada par adjacente de vértices da lista.

Para implementar o algoritmo duas funções da biblioteca *Networkx* foram utilizadas. Uma delas *minimum spanning tree* que recebe como parâmetro o grafo e o algoritmo a ser utilizado, que no caso é o algoritmo de Prim e retorna a árvore geradora mínima. Para caminhar em profundidade sobre esta árvore a função *dfs preorder nodes* é utilizada. Esta por sua vez recebe a árvore geradora mínima e o nó inicial como parâmetro e retorna a lista de vértices na ordem percorrida. Assim para calcular o custo do percurso iteramos sobre esta incrementando uma variável custo inicialmente nula com a aresta formada pelo vértice atual e o próximo vértice da lista.

2.3. Christofides

O segundo algoritmo aproximativo implementado para o trabalho foi o algoritmo de *Christofides*, este funciona da seguinte maneira: primeiro computamos uma ávore geradora mínima T, do grafo, em seguida adicionamos cada vértice de grau ímpar da árvore em uma lista de vértices, l, depois geramos um subgrafo induzido, I do grafo original com os nós da lista l. Computamos o matching perfeito de peso mínimo M, do subgrafo I. Por fim geramos um multigrafo G' formado com os vértices de G e as arestas de M e T, computamos o circuito euleriano em G' e eliminamos vértices duplicados.

Para implementar o algoritmo quatro funções da biblioteca *Networkx* foram utilizadas. Uma delas *minimum spanning tree* que recebe como parâmetro o grafo e o algoritmo a ser utilizado, que no caso é o algoritmo de Prim e retorna a árvore geradora mínima. Em seguida para cada um dos vértices da árvore computada usamos a função *degree* para computar os vértices de grau impar. Depois, chamamos a função subgraph para o grafo original usando como parâmetro a lista de nós de grau impar. Esta função retorna um subgrafo induzido que por sua vez é utilizado como parâmetro para outra função da biblioteca *Networkx*, a função *min weight matching*. Criamos um multigrafo com a classe *MultiGraph* e adicionamos as arestas da árvore e as encontradas no matching. Por fim usamos a função de busca em profundidade da biblioteca, *dfs preorder nodes* que retorna uma lista de vértices representando a ordem por qual cada vértice será visitado. Calculamos a soma dos custos das arestas dos pares de vértices adjacentes da lista e retornamos este custo total.

2.4. Branch and Bound

O último algoritmo implementado foi uma solução ótima utilizando a técnica *Branch and Bound*, a solução funciona da seguinte maneira: Calculamos um limite inferior somando as distâncias das duas cidades mais próximas de cada uma das n cidades e dividimos essa soma por dois. Assim podemos calcular cada uma das possibilidades de circuito no grafo, contudo deixando de calcular uma certa instancia caso essa resulte em um valor menor do que nosso limite inferior. Caso encontremos um circuito de custo menor que o limite inferior, este último assume o valor do menor custo calculado na atual instância. [Levitin 2012]

Para implementar o algoritmo inicialmente para cada um dos vértices verificamos as duas cidades mais próximas a este, isto é, as que formam as arestas de menor peso. Armazenamos estes pesos em um vetor de tuplas nomeado closest. Também somamos estes pesos para inicializar nosso primeiro limite inferior. Em seguida chamamos uma função recursiva que computa o custo do circuito para cada uma das cidades ainda não visitadas, caso este custo seja maior que o limite inferior atual retornamos, caso contrário continuamos o cálculo. Caso estejamos no último nível, ou seja não existem mais cidades para serem visitadas, retornamos o resultado.

3. Experimentos

3.1. Tempos para cada instância

Tempo de Execução (em segundos)						
	Twice Around the Tree		*Christofides*		*Branch and Bound*	
2^i	Euclidean	Manhattan	Euclidean	Manhattan	Euclidean	Manhattan
16	0.00656	0.00045	0.00300	0.00175	4.89871	3.28805
32	0.00184	0.00121	0.00637	0.00429	NA	NA
64	0.00746	0.00543	0.03905	0.03060	NA	NA
128	0.02565	0.02301	0.23443	0.22894	NA	NA
256	0.11285	0.17691	2.12471	1.71487	NA	NA
512	0.77872	0.67023	15.02844	15.79019	NA	NA
1024	4.10071	4.13423	118.72057	128.08117	NA	NA

3.2. Custo da solução de cada algoritmo

Custo da solução de cada algoritmo						
	Twice Around the Tree		*Christofides*		*Branch and Bound*	
2^i	Euclidean	Manhattan	Euclidean	Manhattan	Euclidean	Manhattan
16	5.11530	4.76666	4.12092	5.42677	3.80295	4.52721
32	6.78848	8.95759	7.27664	8.69663	NA	NA
64	7.92138	9.90764	9.25765	10.28145	NA	NA
128	12.26242	15.07830	12.36543	15.86461	NA	NA
256	15.94390	20.23065	16.72676	20.85476	NA	NA
512	22.36349	28.35046	23.08387	28.32556	NA	NA
1024	31.45198	39.23458	31.72484	40.97122	NA	NA

4. Conclusões

4.1. Diferenças de tempo e custo de acordo com as métricas de distância

Em todos os experimentos foi possível observar como a distância euclidiana retorna resultados menores, logo melhores para cada instância. Essa diferença de qualidade inclusive cresce de forma proporcional a quantidade de nós do grafo. Contudo, também foi possível observar uma diferença de tempo na execução das duas métricas, com a métrica de distância Manhattan tendo tempos ligeiramente menores.

4.2. Diferenças de tempo e custo dos algoritmos

Analisando o custo da solução dos algoritmos aproximativos não houve diferença palpável entre ambos. Em alguns casos os resultados de *Christofides* foram melhores e em outros Twice Around the Tree se saiu melhor. Entretanto, foi possível observar uma diferença significativa entre estes em função do tempo de execução. Também foi possível observar uma diferença ainda maior entre os algoritmos aproximativos e a solução ótima retornada pelo algoritmo que utiliza *Branch and Bound*, ainda que não tenha sido possível fazer uma comparação com múltiplas instâncias, afinal a execução do Branch and Bound para $n \geq 2^5$ ultrapassou a marca de dez minutos. Essa última conclusão já era esperada considerando que os algoritmos aproximativos possuem complexidade polinomial.

4.3. Diferenças de custo em relação ao tamanho da instância

Uma última diferença que também ficou visível nos resultados expostos foi como o custo total do caminho aumenta de forma proporcional ao número de nós. É interessante ressaltar que o espaço onde estes nós estão dispostos permanece o mesmo para cada uma das instâncias, no caso o quadrado unitário em R2.

4.4. Conclusões finais

Por meio da implementação do trabalho e análise dos experimentos nas instâncias variando de 2^4 até 2^{10} foi possível observar as diferenças de desempenho no tempo de execução e na qualidade de resposta de algoritmos aproximativos e algoritmos que retornam a solução ótima. Também foi possível observar como diferentes métricas no cálculo de distâncias de cada cidade interfere no resultado final.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (1990). Introduction to algorithms. pages 1027–1032 and 1012–1013. The MIT Press, 3rd edition.

Levitin, A. (2012). *Introduction to the Design and Analysis of Algorithms*. Pearson, 3rd edition.