

Universidade Estadual de Maringá

Relatório Problema do Caixeiro

João Pedro Paes Landim Alkamim
Sarah Anduca de Oliveira

Modelagem e Otimização Algorítmica

11 de Abril de 2022

Sumário

1	Introdução	2
2	Desenvolvimento	2
2.1	Linguagem	2
2.2	Descrição do Problema	2
2.3	Descrição do Algoritmo	3
2.3.1	Construtivos	3
2.3.2	Melhoramento	4
3	Resultados	4
3.1	Configuração da máquina para testes	4
3.2	Tabela comparativa de tempos de execução	4
4	Conclusão	5

1 Introdução

Este relatório diz respeito ao trabalho da matéria de modelagem de algoritmos que teve como objetivo o entendimento e implementação de algoritmos heurísticos de construção e melhoramento para o problema de Traveling Salesman, ou Caixeiro Viajante.

2 Desenvolvimento

2.1 Linguagem

Utilizamos a linguagem Python, na versão 3.9.1, para implementação dos algoritmos.

2.2 Descrição do Problema

O problema do Caixeiro Viajante consiste em encontrar o caminho de menor custo ao percorrer um trajeto em um grafo completo, como é apresentado na formulação matemática abaixo, apresentada por Miller–Tucker–Zemlin (1960).

$$\begin{aligned} \min Z &= \sum_{i=1}^n \sum_{j=1}^n c_{ij} x_{ij} \\ \text{s.a} \quad &\sum_{i=1}^n x_{ij} = 1, j = 1, 2, \dots, n \\ &\sum_{j=1}^n x_{ij} = 1, i = 1, 2, \dots, n \\ &u_i - u_j + nx_{ij} \leq n - 1 \quad \forall 2 \leq i \neq j \leq n \\ &x_{ij} \in \{0, 1\} \quad \forall i, j \end{aligned}$$

Neste caso o trabalho foi baseado em resolver o Caixeiro Viajante utilizando quatro heurísticas, duas de construção e duas de melhoramento. Métodos heurísticos são maneiras de resolver problemas que não garantem a solução ótima ou se quer resolver o problema, mas em geral produzem soluções próximas das ideais.

Para a solução do Caixeiro Viajante utilizamos duas classificações de algoritmos heurísticos: construtivos e de melhoria. Algoritmos construtivos criam

soluções a partir do zero e seguindo um conjunto de regras para a construção da solução do problema:

- a escolha do ciclo ou nó inicial da solução – *inicialização*;
- um critério de escolha do próximo elemento a juntar à solução – *seleção*;
- a escolha da posição onde esse novo elemento será inserido – *inserção*;

Já as de melhoria são auto explicativas, recebem uma solução já encontrada e tentam melhorar o resultado o máximo possível com o menor número de interações necessárias.

Vale ressaltar que o qualquer que seja o algoritmo heurístico construtivo ou de melhoria não garantem a solução ótima, mas podem vir a acertá-la ou chegar muito próximo do resultado exato.

2.3 Descrição do Algoritmo

Representamos os grafos pela classe *Graph*, com os seguintes atributos:

vertex (array): lista dos vertices do grafo

size (int): numero de vertices do grafo

Com método de setVertex com parametros id, x, y, que adiciona vertices no array vertex.

A classe *Edges* que inicializa com x e y sendo os pontos do vértice, e *weight* sendo o peso. O método *setEdge* para atribuir valores a novas arestas.

2.3.1 Construtivos

Vizinho mais próximo Para a implementação do Vizinho Mais Próximo utilizamos a função *nearestNeighbor* que recebe o grafo como parâmetro. A função armazena um vértice aleatório que inicializará o caminho a ser percorrido, feito isso, o algoritmo percorre todo o grafo verificando se o vertice atual já foi visitado, se não, ele compara o peso das arestas entre o vértice aleatório e o vertice atual e o peso mínimo encontrado até então, se for menor que o mínimo, o mínimo recebe o atual e o último vértice é armazenado. Feito isto com todos os vértices não visitados, a função retorna o peso de todas as arestas.

Este algoritmo tem um custo de execução de $O(n^2)$.

Inserção mais próxima No algoritmo de inserção mais próxima temos a função *nearestInsertion* que recebe o grafo como o parâmetro. A função

armazena um vértice aleatório e insere na lista *path*. O próximo vértice mais próximo é inserido nesta lista também. Enquanto todos os vértices não tenham sido visitado, um vértice aleatório é escolhido da lista *path* para que se obtenha o mais próximo dele com a função *nearestVertex*. Ao encontrar, a função *insertInPath* é utilizada para a melhor inserção deste vértice em *path*. Este algoritmo tem o custo de execução de $O(n)$.

2.3.2 Melhoria

2-opt A implementação do algoritmo 2 Opt foi feita na função *twoOpt* que recebe o conjunto de todas as arestas de um grafo já montado, percorrendo toda a lista de arestas verificando de duas em duas se são adjacentes, se não, verifica se o peso das arestas com a configuração atual é maior do que o peso com as arestas trocadas, se sim, faz o *swap* entre as duas arestas, promovendo a melhoria do grafo para um caminho de menor peso. Este algoritmo tem um custo de execução de $O(n^2)$.

3-opt A ideia do 3 Opt é a mesma da do 2 Opt, porém ao invés de comparar de duas em duas arestas, iremos comparar de três em três, sendo assim teremos sete configurações possíveis para o *swap*. Tanto no caso do 2 Opt quanto no 3 Opt após o *swap* é feito a inversão do grafo, pois a direção muda conforme a ligação das arestas entre si. Este algoritmo tem um custo de execução de $O(n^3)$.

3 Resultados

3.1 Configuração da máquina para testes

- Processador Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
- Gráficos GTX 1060 6GB
- Memória RAM 16GB
- SSD 240GB
- Sistema Operacional Windows 10

3.2 Tabela comparativa de tempos de execução

Abaixo se encontra a tabela com todos os resultados dos casos de PCV adquirida no *moodle* da matéria de Modelagem e Otimização de Al-

goritmo. Vale ressaltar que apenas o caso: *pr1002* tem uma comparação entre os dois algoritmos de melhoramento, 2-opt e 3-opt. Os demais testes têm somente o resultado do primeiro, por conta do outro ser um algoritmo com o tempo de execução $O(n^3)$, necessitando de muito tempo para executar este algoritmo (mais de dez horas).

Todos os testes foram feitos usando o o algoritmo construtivo Vizinho mais próximo.

Caso	MS	Alg1	$GAP_1\%$	Alg2	$GAP_2\%$
pr1002.tsp	259.045	280.914	8,44%	268.168	3,52%
fnl4461.tsp	182.566	196.377	7,56%	-	-
brd14051.tsp	469.385	506.855	7,98%	-	-
d15112.tsp	1.573.084	1.699.046	8,00%	-	-
d18512.tsp	645.238	695.505	7,79%	-	-
pla7397.tsp	23.260.728	24.584.356	5,69%	-	-
pla33810.tsp	66.048.945	71.409.872	8,11%	-	-
pla85900.tsp	142.382.641	151.366.083	6,30%	-	-

Tabela 1: Tabela de Comparação dos Resultados

4 Conclusão

Com tudo o que fora apresentado e trabalhado por nós, pudemos notar o que de fato é heurística na prática e que na maioria das vezes, apesar de se aproximar muito do resultado desejado, não garante a resposta correta, mas melhora a saída de forma significativa, ainda mais se combinadas heurísticas construtivas com de melhoramento. Além disso, não necessariamente uma heurística tem tempo computacional ótimo, como o exemplo do 3Opt, com custo de execução de $O(n^3)$, beirando o computacionalmente possível.

No mais, notamos o quanto heurísticas auxiliam na resolução de problemas, pois apesar de não devolver o resultado ótimo, chega muito próximo à ele com algoritmos simples com custo computacionalmente possível, podendo implementar soluções de problemas NP-Difícil, como a do Caixeiro Viajante.