Paradigmas de Projeto de Algoritmos - O Problema do Caixeiro-Viajante

Davi Azarias do Vale Cabral[1], **João Antonio Lassister Melo**[1], **João Antonio Siqueira Pascuini**[1], **Renan Catini Amaral**[1], **Thallysson Luis Teixeira Carvalho**[1], **Vinicius Ribeiro da Silva do Carmo**[1]

[1]Departamento de Ciências da Computação – Universidade Federal de Alfenas
Avenida Jovino Fernandes de Sales 2600 – CEP 37133840 - Alfenas – MG - Brasil

davi.cabral@sou.unifal-mg.edu.br,

joao.lassister@sou.unifal-mg.edu.br,

joaoantonio.pascuini@sou.unifal-mg.edu.br,

renan.amaral@sou.unifal-mg.edu.br,

thallysson.carvalho@sou.unifal-mg.edu.br,

viniciusribeiro.carmo@sou.unifal-mg.edu.br

Resumo. *Este trabalho tem como objetivo apresentar três algoritmos que encontrem soluções para o problema do caixeiro-viajante, cada um baseado em um diferente paradigma de projeto de algoritmos: os paradigmas de força bruta, guloso e divisão-e-conquista. Os três algoritmos apresentados serão utilizados na solução de instâncias do problema e avaliados de acordo com seu desempenho.*

1. Introdução

O problema do caixeiro-viajante pode ser definido informalmente da seguinte forma: um caixeiro-viajante deve visitar todas as cidades em um determinado conjunto e retornar para a cidade de partida, de modo que nenhuma cidade seja visitada mais de uma vez e o caminho total percorrido por ele seja o menor possível.

Formalmente, podemos tratar esse problema como uma busca por um ciclo hamiltoniano em um grafo completo $G = (C, A)$ para um conjunto de cidades $C = \{c_1, c_2, ..., c_n\}$ de tamanho n, um conjunto de arestas $A = \{(c_i, c_j) | c_i, c_j \in C \wedge c_i \neq c_j\}$ e uma função $p : A \rightarrow \mathbb{R}^+$ que dê um peso para cada aresta.

Um ciclo hamiltoniano H no grafo G é uma lista de arestas de G $((c_1, c_i), (c_i, c_j), (c_j, c_k), ..., (c_z, c_1))$ tal que cada cidade aparece exatamente duas vezes na lista, sendo a primeira vez como a segunda cidade na aresta e como a primeira cidade na aresta seguinte. A única exceção é c_1, a cidade inicial, que deve aparecer como a primeira cidade na primeira aresta e como a segunda cidade na última aresta.

Para resolver o problema do caixeiro-viajante, devemos encontrar o ciclo hamiltoniano H no grafo G que faça a soma dos pesos p das arestas em H ser a menor possível.

O problema do caixeiro-viajante é sabidamente NP-Completo, portanto não conhecemos nenhum algoritmo determinístico que o solucione em tempo polinomial. Na

seção seguinte, apresentaremos um algoritmo que computa a solução ótima para qualquer instância do problema, embora seu tempo de execução se torne rapidamente intratável conforme aumentamos a quantidade de cidades. Apresentaremos também dois algoritmos que encontram caminhos que não resolvem o problema na sua otimalidade, isto é, o ciclo hamiltoniano encontrado nem sempre será o menor possível, mas que, em contrapartida, são capazes de lidar com instâncias maiores em tempo hábil.

Por fim, os três algoritmos serão testados e avaliados de acordo com seu tempo de execução e proximidade à solução ótima ao resolver instâncias de diferentes tamanhos.

2. Algoritmos

2.1. Força bruta

Algoritmos de força bruta testam todas as possibilidades de resposta para encontrar a melhor. Em particular, para o problema do caixeiro-viajante, devemos computar o custo de todos os ciclos hamiltonianos no grafo G.

---

algoritmo melhorCaminho(c_a, C)

  se C.tamanho == 1
    entao retorna p(c_a, c_1)
  fim-se

  menor <- ∞

  para $c_i \in C$

    menor <- **min**(menor, melhorCaminho($c_i, C \setminus \{c_i\}$) + p(c_a, c_i))

  fim-para

  retorna menor

fim-algoritmo

A função recursiva apresentada acima computa qual o menor caminho para visitar todas as cidades no conjunto C, que contém todas as cidades ainda não visitadas, a partir de c_a, passada como parâmetro, até retornar a c_1, a cidade de início global.

Ela começa testando o caso base, em que há somente uma cidade a ser visitada, c_1. Neste caso, o menor caminho será ir diretamente de c_a a c_1.

Caso haja mais de uma cidade a ser visitada, o algoritmo faz um laço $\forall c_i \in C$ para somar o peso de ir de c_a a c_i ao menor custo de ir de c_i a c_1, removendo c_i do conjunto de cidades a serem visitadas e passando-a como a cidade atual em uma chamada recursiva.

A abordagem acima garante que a função retornará o peso do menor ciclo hamiltoniano existente, uma vez que ela verifica todos. Essa garantia, contudo, vem às custas

de uma grande complexidade computacional. A função de recorrência deste algoritmo será a seguinte:

- Para C de tamanho $n = 1$, a função terá um custo constante, dado pelo teste e o cálculo de $p(c_a, c_1)$. Assim, $T(1) = 1$.
- Para C de tamanho $n > 1$, a função terá o custo de percorrer os n elementos do conjunto em um laço, chamando-se recursivamente a cada iteração com uma nova entrada subtraída de somente um elemento. Assim, $T(n) = n * T(n-1)$.

Desdobrando essa função recursiva, teremos que $T(n) = n!$. Isto é, este algoritmo tem complexidade de tempo $O(n!)$.

2.2. Algoritmo guloso

Algoritmos gulosos avaliam o estado atual do problema e avançam fazendo a escolha que parece mais adequada no momento. Para alguns problemas, esta abordagem garante que a solução ótima será encontrada. Todavia, para outros problemas, como os NP-Completos, e o problema do caixeiro-viajante em particular, isso não é verdade, e um algoritmo guloso muitas vezes nos dará um caminho pior do que o ótimo, embora com complexidade temporal muito menor, como veremos a seguir.

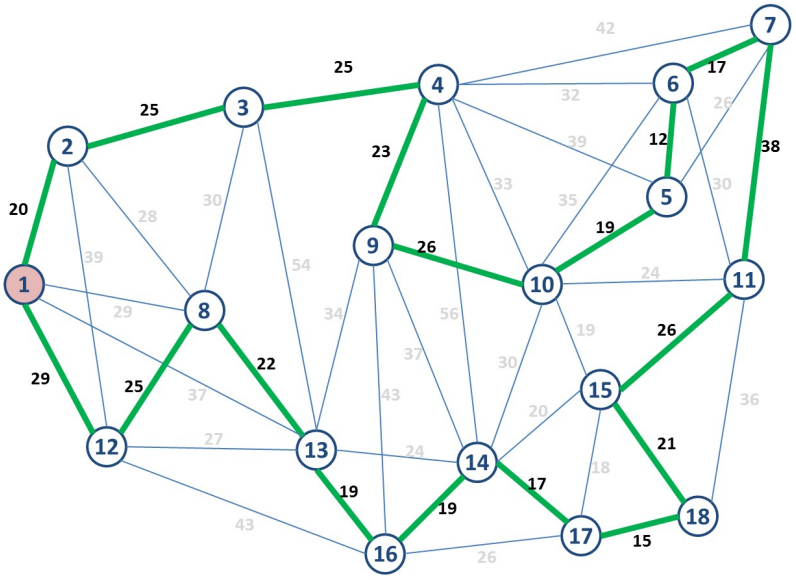

Figura 1. Solução gulosa para um grafo de 18 vértices

Uma possível aplicação desse paradigma para o problema do caixeiro-viajante seria a seguinte:

```
algoritmo caminhoGuloso(c_1, C)

    caminhoTotal <- 0
    c_a <- c_1

    enquanto C.tamanho > 1
        menorPeso <- ∞
        para c_i ∈ C\{c_a}
            se p(c_a, c_i) < menorPeso entao
                vizinhoMaisProx <- c_i
                menorPeso <- p(c_a, c_i)
            fim-se
        fim-para
        Caminho.insere(c_a, vizinhoMaisProx)
        caminhoTotal <- caminhoTotal + menorPeso
        C <- C\{c_a}
        c_a <- vizinhoMaisProx
    fim-enquanto

    Caminho.insere(c_a, c_1)
    Caminho.peso <- caminhoTotal + p(c_a, c_1)

    retorna Caminho

fim-algoritmo
```

Partindo da cidade inicial c_1, o algoritmo itera sobre as demais cidades do conjunto buscando aquela cuja distância da cidade atual seja a menor, remove a cidade atual do conjunto C e parte para a cidade mais próxima, reiniciando o laço, que se repetirá até que haja somente uma cidade a ser visitada. Por fim, a função retorna a soma de todas as distâncias escolhidas mais o custo de retornar da cidade restante do conjunto à cidade inicial.

Os laços aninhados mostrados acima, aliados ao fato de que o conjunto é subtraído de 1 a cada iteração, indicam-nos que esse algoritmo deve ter complexidade $O(n^2)$, isto é, seu tempo de execução nunca será maior que o tempo de analisar n vezes cada elemento de uma entrada de tamanho n.

Note que, além de calcular o peso do caminho encontrado, a função também constrói e retorna o ciclo hamiltoniano na lista **Caminho**. Isso nos será útil para o algoritmo apresentado a seguir.

2.3. Divisão e conquista

Algoritmos de divisão e conquista desmembram o problema em vários subproblemas que são semelhantes ao problema original, mas de menor tamanho, resolvem os subproblemas recursivamente e depois combinam essas soluções com o objetivo de criar uma solução

para o problema original.

O paradigma de divisão e conquista envolve três passos em cada nível da recursão:

- **Divisão** do problema em determinado número de subproblemas que são instâncias menores do problema original.
- **Conquista** os subproblemas, resolvendo-os recursivamente. Porém, se os tamanhos dos subproblemas forem pequenos o bastante, basta resolver os subproblemas de maneira direta.
- **Combinação** das soluções dadas aos subproblemas na solução para o problema original.

[Cormen et al. 2012]

Para o problema do caixeiro-viajante, seguiremos o seguinte algoritmo para as etapas de divisão e conquista:

```
algoritmo divisaoEConquista(C)

    se C.tamanho <= 5
        //c_1 eh uma cidade inicial qualquer
        entao retorna caminhoGuloso(c_1, C)
    fim-se

    caminhoEsq <- divisaoEConquista(C.metadeEsq)
    caminhoDir <- divisaoEConquista(C.metadeDir)

    retorna combinaCaminhos(caminhoEsq, caminhoDir)

fim-algoritmo
```

Para um conjunto de cidades de tamanho 5 ou menor, o problema é resolvido diretamente utilizando o algoritmo guloso mostrado acima.

Caso o conjunto de cidades tenha um tamanho maior que 5, a função chama-se a si mesma duas vezes passando uma metade do conjunto para cada chamada recursiva.

Uma vez que um nível da recursão tem dois caminhos prontos, ela chama a função **combinaCaminhos**, que recebe dois caminhos distintos, unindo-os em um caminho único.

```
algoritmo combinaCaminhos(cam1, cam2)

    menorSoma <- ∞
    para (a_i, a_j) ∈ cam1
        para (b_k, b_l) ∈ cam2
            cost1 <- p(a_i, b_k) + p(a_j, b_l) - p(a_i, a_k) - p(b_k, b_l)
            cost2 <- p(a_i, b_l) + p(a_j, b_k) - p(a_i, a_k) - p(b_k, b_l)
            menorSoma <- min(menorSoma, cost1, cost2)
        fim-para
    fim-para

    se menorSoma.caso1
        entao novoCaminho <- (a_1, ..., a_i) ∪ (b_k, ..., b_l) ∪ (a_j, ..., a_1)
        senao novoCaminho <- (a_1, ..., a_i) ∪ (b_l, ..., b_k) ∪ (a_j, ..., a_1)
    fim-se

    novoCaminho.peso <- cam1.peso + cam2.peso + menorSoma
    retorna novoCaminho

fim-algoritmo
```

O algoritmo busca a melhor forma de desconectar duas cidades para unir os dois caminhos. Para cada par de arestas, dado um estado inicial, há duas formas de fazer isso:

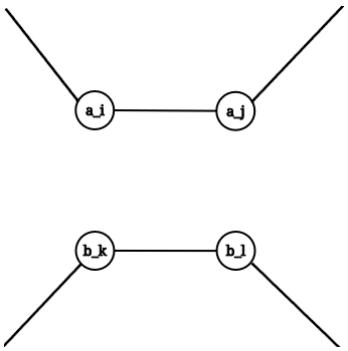

Figura 2. Início

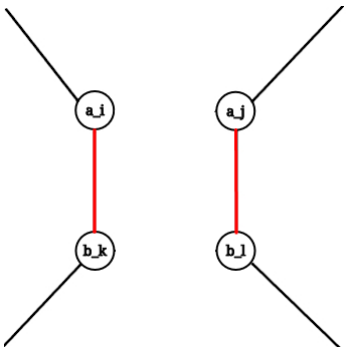

Figura 3. Caso 1

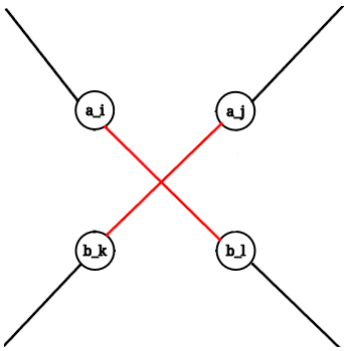

Figura 4. Caso 2

Cabe a essa função unir os dois caminhos de modo que o peso do novo caminho seja minimizado.

Uma vez que nem a forma de tratar o caso base (pelo algoritmo guloso) nem a forma de unir os subcaminhos garantem otimalidade de solução (uma vez que esta une os caminhos pela remoção de uma única aresta), também não podemos esperar que esse algoritmo vá encontrar o caminho ótimo para toda instância do problema.

Como a fase inicial de divisão divide a instância em 2 subproblemas com a metade do tamanho, a solução do caso base pelo algoritmo guloso fornece um custo constante

(neste caso, porque ele é invocado somente quando $n \leq 5$) e a combinação dos caminhos oferece um custo de $O(n^2)$ (o que pode ser constatado pelos laços aninhados), a função de recorrência que analisa a complexidade desse algoritmo será $T(n) = 2T(n/2) + n^2$. Isso nos leva ao caso 3 do teorema mestre:

$$n^2 = \Omega(n^{log_2 2+\epsilon}), \epsilon = 1$$

Ou seja, caso $2(n/2)^2 \leq cn^2$, para alguma constante $c > 1$ e n suficientemente grande, então $T(n) = \Theta(n^2)$.

$$(\exists c > 1)(2(n/2)^2 \leq cn^2)$$
$$(\exists c > 1)(2(n^2/4) \leq cn^2)$$
$$(\exists c > 1)((n^2/2) \leq cn^2)$$
$$(\exists c > 1)(1/2 \leq c)$$

A condição é verdadeira, logo a complexidade é $\Theta(n^2)$.

3. Resultados

Os algoritmos descritos e analisados acima foram implementados em linguagem C e anexados a este PDF. Para executá-los, foram utilizadas instâncias retiradas do site *http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/*.

Como já observado anteriormente, a execução do algoritmo de força bruta torna-se rapidamente intratável, portanto, optamos por alterar as instâncias submetidas a ele a um tamanho de no máximo 14, para que os resultados pudessem ser obtidos em tempo hábil. Contudo, a partir do resultado obtido dessas instâncias e da análise de complexidade do algoritmo, somos capazes de estimar quanto tempo ele levaria para resolver instâncias de um tamanho n qualquer.

Tabela 1. Experimentação do algoritmo de força bruta

	Tempo de execução
Instância com 6 cidades	0.000027 segundos
Instância com 7 cidades	0.000100 segundos
Instância com 8 cidades	0.000572 segundos
Instância com 9 cidades	0.004575 segundos
Instância com 10 cidades	0.045673 segundos
Instância com 11 cidades	0.299534 segundos
Instância com 12 cidades	2.746395 segundos
Instância com 13 cidades	33.908393 segundos
Instância com 14 cidades	267.532962 segundos

Tabela 2. Tempo de execução de cada instância

	Algoritmo Guloso	Divisão e Conquista	Força bruta (estimativa)
Instância ”eil51.tsp”	0.000019 segundos	0.000022 segundos	$\approx 1.51 * 10^{50}$ anos
Instância ”berlin52.tsp”	0.000018 segundos	0.000026 segundos	$\approx 7.86 * 10^{51}$ anos
Instância ”st70.tsp”	0.000031 segundos	0.000037 segundos	$\approx 1.17 * 10^{84}$ anos
Instância ”pr76.tsp”	0.000032 segundos	0.000043 segundos	$\approx 1.83 * 10^{95}$ anos
Instância ”eil76.tsp”	0.000037 segundos	0.000042 segundos	$\approx 1.83 * 10^{95}$ anos
Instância ”kroA100.tsp”	0.000058 segundos	0.000070 segundos	$\approx 9.07 * 10^{141}$ anos
Instância ”kroC100.tsp”	0.000060 segundos	0.000070 segundos	$\approx 9.07 * 10^{141}$ anos
Instância ”kroD100.tsp”	0.000058 segundos	0.000073 segundos	$\approx 9.07 * 10^{141}$ anos
Instância ”rd100.tsp”	0.000058 segundos	0.000069 segundos	$\approx 9.07 * 10^{141}$ anos
Instância ”eil101.tsp”	0.000057 segundos	0.000070 segundos	$\approx 9.16 * 10^{143}$ anos
Instância ”lin105.tsp”	0.000053 segundos	0.000079 segundos	$\approx 1.32 * 10^{146}$ anos
Instância ”ch130.tsp”	0.000089 segundos	0.000112 segundos	$\approx 6.30 * 10^{203}$ anos
Instância ”ch150.tsp”	0.000114 segundos	0.000144 segundos	$\approx 5.55 * 10^{246}$ anos
Instância ”u159.tsp”	0.000101 segundos	0.000155 segundos	$\approx 2.87 * 10^{266}$ anos
Instância ”ts225.tsp”	0.000171 segundos	0.000300 segundos	$\approx 1.34 * 10^{417}$ anos
Instância ”tsp225.tsp”	0.000206 segundos	0.000322 segundos	$\approx 1.34 * 10^{417}$ anos
Instância ”a280.tsp”	0.000320 segundos	0.000469 segundos	$\approx 6.20 * 10^{547}$ anos
Instância ”pcb442.tsp”	0.000716 segundos	0.001123 segundos	$\approx 1.26 * 10^{962}$ anos
Instância ”pr1002.tsp”	0.002951 segundos	0.005944 segundos	$> 10^{2551}$ anos
Instância ”pr2392.tsp”	0.016237 segundos	0.042381 segundos	$> 10^{6984}$ anos

Tabela 3. Peso do caminho encontrado

	Algoritmo Guloso	Divisão e Conquista	Peso do caminho ótimo
Instância ”eil51.tsp”	514	971	426
Instância ”berlin52.tsp”	8981	17855	7542
Instância ”st70.tsp”	806	2073	675
Instância ”pr76.tsp”	153462	174012	108159
Instância ”eil76.tsp”	712	1393	538
Instância ”kroA100.tsp”	26856	97058	21282
Instância ”kroC100.tsp”	26327	89918	20749
Instância ”kroD100.tsp”	26950	87773	21294
Instância ”rd100.tsp”	9941	30970	7910
Instância ”eil101.tsp”	825	1732	629
Instância ”lin105.tsp”	20362	27329	14379
Instância ”ch130.tsp”	7575	25707	6110
Instância ”ch150.tsp”	8194	29324	6528
Instância ”u159.tsp”	54669	63239	42080
Instância ”ts225.tsp”	4828	8686	3916
Instância ”tsp225.tsp”	152494	293488	3916
Instância ”a280.tsp”	3148	3922	2579
Instância ”pcb442.tsp”	61984	157617	50778
Instância ”pr1002.tsp”	315597	410321	259045
Instância ”pr2392.tsp”	461207	549857	378032

Os resultados nos mostram que o algoritmo guloso implementado é superior à divisão e conquista tanto em tempo de execução quanto em proximidade ao caminho ótimo. Embora sua margem de erro aumente conforme aumentamos o tamanho da instância, ela permanece razoavelmente pequena.

O algoritmo de divisão e conquista, por sua vez, demonstrou uma margem de erro muito maior com uma latência de execução ligeiramente maior. Perceba como sua margem de erro não parece variar uniformemente em função do tamanho da instância. Investiguemos o motivo.

Na descrição do algoritmo oferecida na seção anterior, vimos como, na etapa de divisão, o conjunto de cidades é dividido ao meio sem preocupação em agrupar as cidades por proximidade. Isso faz com que o desempenho do programa dependa da forma como o conjunto foi apresesentado na instância de leitura.

Uma estratégia possível para aumentar o desempenho desse algoritmo seria realizar uma ordenação do conjunto durante a fase de divisão, de modo que cidades geograficamente mais próximas fiquem no mesmo subconjunto.[Darwish e Talkhan 2014]

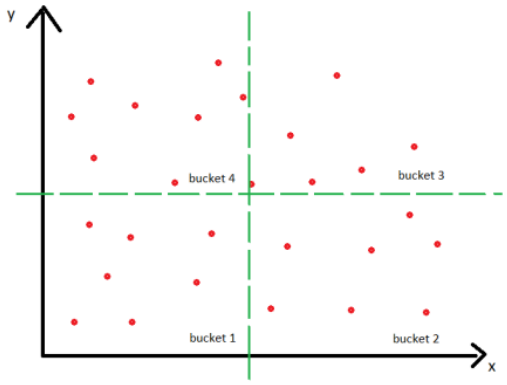

Figura 5. Exemplo de ordenação das cidades em buckets

Concluímos que um algoritmo de força bruta, ainda que exato, torna-se incapaz de resolver o problema em tempo razoável quando o tamanho da instância cresce ligeiramente. O algoritmo guloso, por sua vez, mostrou uma complexidade de tempo polinomial e com resultados próximos ao que sabemos ser o ótimo, embora sua margem de erro aumente conforme aumentamos as instâncias. O algoritmo de divisão e conquista, ao menos na implementação aqui apresentada, tem uma margem de erro grande demais e está altamente sujeito ao formato das instâncias, e o artigo de Darwish e Talkhan citado nas referências a seguir é sugerido para dar ideias de como melhorá-lo.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2012). *Algoritmos - Teoria e Prática*. GEN LTC, 3rd edition.

Darwish, H. A. and Talkhan, I. (2014). Reduced complexity divide and conquer algorithm for large scale tsps. *International Journal of Advanced Computer Science and Applications*.