PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Diferentes estratégias de solução para o problema do caixeiro viajante*

Different solution strategies for the traveling salesman problem

João Rene Teixeira Filho[1]
Pedro Diogenes[2]

*Artigo apresentado como trabalho da materia Projeto e Analise de Algoritmos no segundo semestre de 2019.

[1]Aluno do Programa de Graduação em Ciência da Computação, Brasil – joao.filho.1092201@sga.pucminas.br.
[2]Aluno do Programa de Graduação em Ciência da Computação, Brasil – pedro.lara@sga.pucminas.br.

1 INTRODUÇÃO

O problema do caixeiro viajante consiste em visitar todas as cidades dadas exatamente uma vez, com a menor distância possível. Ou seja, achar o Ciclo Hamiltoniano de menor custo. Este é um problema NP-difícil, baseado na necessidade cada vez maior de uma otimização da logística de transporte, reduzindo a distância necessária, tempo e custos.

Nesse artigo iremos implementar soluções com quatro paradigmas diferentes de programação para resolver o problema: Força bruta, Branch and Bound, Programação Dinâmica e Algoritmos Genéticos. Após a implementação será feita a análise de complexidade de cada solução.

Serão realizados testes com um conjunto de entradas pré-determinadas para cada abordagem escolhida. Os resultados obtidos serão demonstrados, assim como tempo de execução com o objetivo de comparar cada estratégia escolhida.

2 IMPLEMENTAÇÃO

Nesta sessão será especificada a implementação de cada programa. Serão detalhadas as estruturas utilizadas, o funcionamento das principais funções e o formato de entrada e saída de dados.

Os programas foram escritos em conjunto utilizando a IDE Atom que permite a edição do código em tempo real por mais de uma pessoa. Os programas foram executados e testados usando os sistemas operacionais Ubuntu 18, macOS Mojave versao 10.14.5, e Windows 10, e foram compilados com o compilador G++ 9.2. As implementações foram feitas no estilo pair programming, onde tentamos pensar juntos nas melhores soluções para cada paradigma.

A entrada de dados se dá por meio do terminal ou de um arquivo de entrada, a primeira linha da entrada é o numero N de cidades, e as linhas subsequentes são as coordenadas X/Y de cada cidade em ordem. Na tabela 1 observa-se um exemplo de entrada.

Tabela 1 – Entrada de dados

5
100 100
200 200
300 300
400 400
500 500

A saída de todos programas é a menor distancia possível, o caminho que gera essa distancia e o tempo de execução do algoritmo.

2.1 Força Bruta

O algoritmo de força bruta testa todas as permutações de cidades para depois comparar qual o melhor caminho a ser tomado.

O método principal do programa recebe as entradas, cria um vetor de pares ordenados das cidades, e cria uma matriz de adjacência com as respectivas distancias entre as cidades utilizando a função *calcDist*. As distâncias entre as mesmas cidades são inicializadas como Infinito, para não interferir no cálculo da menor distância.

Tabela 2 – Matriz de adjacência (Estrutura de Dados)

Cidades	Cidade 1	Cidade 2	Cidade 3	Cidade 4	Cidade 5
Cidade 1	INF	141,421	282,842	424,264	565,685
Cidade 2	141,421	INF	141,421	282,842	424,264
Cidade 3	282,842	141,421	INF	141,421	282,842
Cidade 4	424,264	282,842	141,421	INF	141,421
Cidade 5	565,685	424,264	282,842	141,421	INF

Com uma implementação simples e intuitiva é usada uma variável menorDist com o maior inteiro possível para comparação da distancia do caminho. Então é chamado o método bruteForce que faz a permutação das cidades, calculando a distancia resultante. Após isso é comparado se a distancia encontrada é menor que a menorDist, se sim é salvo o caminho atual e atribuído a distancia atual a menorDist.

Método bruteForce

```
void bruteForce(int a[], int n, double menorDist){
  //orgazina o vector
  sort(a, a+n);
  double somaDist = 0; // inicia soma das distancias
  do{
    for (int i = 0; i < n; i++) {
        //soma as distancias do caminho
          somaDist += m[a[i]][a[i+1]];
    }
    if(somaDist < menorDist){
      // se essa soma for menor que a menor atual,
      // salva ela como menor e salva o caminho dela
      menorDist = somaDist;
      copyPath(a, n);
    }
    somaDist = 0; //reseta a soma
  } while (next_permutation(a + 1, a + n));
      //proxima permuta  o

  // procedimento para imprimir resposta
  cout << menorDist << endl;
  for(int i = 0; i < shorterPath.size() - 1; i++){
    cout << shorterPath[i] + 1 << " ";
  }
  cout << endl;
}
```

2.2 Branch and Bound

No método Branch and Bound, para cada vértice (nesse caso, uma cidade na nossa matriz de adjacência) calcula-se um limite inferior da melhor solução possível no momento em que essa cidade for percorrida. Se o limite inferior da solução utilizado a cidade atual for pior do que o melhor limite inferior calculado até então, ignoramos essa cidade no passo atual da solução completa. Portanto, podemos perceber que essa solução funciona de maneira similar à força bruta. A partir de um limite inferior inicial, tentaremos colocar cada cidade como a próxima (similar à geração de permutações do força bruta). Entretanto, caso essa cidade gerar uma solução que seja pior que a melhor solução encontrada até então, nós a descartamos, e consequentemente descartamos todas as permutações de cidades consecutivas utilizando a atual como base. A parte mais difícil do paradigma é calcular o limite inferior para cada passo do problema.

Nossa entrada de dados é a mesma entrada do algoritmo de força bruta (Veja Tabela 1) e é montada a matriz de adjacência com a distancia entre as cidades (Veja Tabela 2). Para calcular o limite inferior inicial, devemos considerar que qualquer caminho achado será maior que $\frac{1}{2} \sum sc$, onde sc = Soma do custo dos dois caminhos de menor peso adjacentes a cidade atual (DESCONHECIDO,).

Iniciamos o código com o calculo do limite inferior:

Calcular o limite inferior

```
void TSP(int adj[][100]) {
    int caminhoAtual[N+1];

    // calcula o limite inferior atual
    // usando a formula formula 1/2 * (soma da menor aresta +
    // segunda menor aresta) para toda aresta.
    int limite_atual = 0;
    memset(caminhoAtual, -1, sizeof(caminhoAtual));
    memset(visited, 0, sizeof(caminhoAtual));
    for (int i=0; i<N; i++)
        limite_atual += (menorAresta(adj, i) +
                         segundaMenor(adj, i));

     //arredonda o limite atual
    limite_atual = (limite_atual&1)? limite_atual/2 + 1 :
                                     limite_atual/2;

    visited[0] = true;
    caminhoAtual[0] = 0;
    TSPRec(adj, limite_atual, 0, 1, caminhoAtual);
}
```

O programa então calcula o limite de cada cidade recursivamente, ignorando a cidade caso o limite dela seja maior que o menor calculado até então.

Calcular o limite inferior

```
limite_atual -= ((segundaMenor(adj, caminhoAtual[level-1]) +
                                menorAresta(adj, i))/2);

// limite_atual + pesoAtual eh o limite inferior atual
if (limite_atual + pesoAtual < respostaFinal) {
    caminhoAtual[level] = i;
    visited[i] = true;


    TSPRec(adj, limite_atual, pesoAtual, level+1,
           caminhoAtual);
}
```

2.3 Programacao Dinamica

O algorítmo da programação dinâmica utiliza de uma matriz auxiliar para gravar resultados de calculos já executados, afim de acelerar a execução dos passos seguintes através de sua reutilização. Esse método utiliza de uma máscara de bits, que possui o tamanho N, sendo N o número de cidades. Essa máscara é utilizada para marcar quais cidades já foram visitadas, por exemplo: 0001 significa que a primeira entre 4 cidades já foi visitada. Portanto, entre as execuções recursivas do código, temos 2^N possibilidades de visitas de cidades.

Gerando a máscara de bits

```
int baseCase = (1 << n) - 1;
//gera um caso base, exemplo: 1111 para 4 cidades
//significando que as 4 cidades foram visitadas.
```

Executamos então recursivamente, passando como parâmetros a máscara de bits do caso base, a cidade atual, e a máscara de bits atual (que é inicializada com 1, indicando 0...1 - a primeira cidade foi visitada). O código calcula recursivamente a distancia entre todas as cidades, para cada cidade, descartando aquelas que façam com que a distancia seja maior que a menor distancia calculada até então.

Execução recursiva

```
for(int prox = 0; prox < n; prox++){
    //verifica se a proxima cidade ja foi visitada
    if((mask&(1<<prox)) == 0){
        //resposta parcial = distancia entre o vertice atual e o
        //proximo, e a menor distancia calculada recursivamente
        int parcial = m[atual][prox] +
            TSPutil((mask|(1<<prox)), prox, bc, caminho_parcial);
        if(parcial < ans) ans = parcial;
    }
}
```

O código para quando a máscara de bits for igual à do caso base, significando que todos as cidades foram visitadas. Ele então retorna a distancia da cidade final até a inicial.

Caso base

```
if(mask==baseCase){
    //se a mascara for igual ao caso base
    //retorna a distancia entre a ultima cidade visitada e a inicial
    return m[atual][0];
}
```

2.4 Algoritmo Genetico

O algorítmo utilizado para análise do procedimento genético neste problema foi desenvolvido por Marcos Castro (CASTRO, 2014). Algorítmos Genéticos são técnicas de otimização baseadas na seleção natural, que é a mudança de um elemento ao passar de um longo período de tempo, devido a adaptação. Na natureza, os indivíduos "mais fortes"são os que tem a maior chance de sobreviver e de se reproduzir, gerando assim indivíduos de uma próxima geração que são ainda mais fortes e saudáveis, pois vieram de pais considerados fortes. Os algorítmos genéticos funcionam a partir dessa mesma lógica. Gera-se um conjunto de soluções aleatórias, e "casam-se"as soluções mais próximas das corretas para gerar um novo conjunto de soluções que (provavelmente) será mais correto que o conjunto anterior (SABAN; WALTUCH, 2012).
O algorítmo começa aleatóriamente gerando um conjunto de soluções (uma população). Então, dois mais "pais"são selecionados para formar novas soluções. A seleção dos pais é feita de acordo com a "força"da solução. Quanto mais cabível uma solução é, maior as chances de ela ser utilizada para um casamento. Posteriormente, as soluções geradas passam por um processo de mutação até alguma condição específica ser satisfeita. O processo então se repete até achar a solução mais próxima possível do problema.

Abaixo segue um pseudocódigo da solução genética para o Caxeiro Viajante:

Pseudocódigo

```
gerarPopulacao(n); //gera n solucoes aleatorias para o problema
while(//solucao final nao encontrada){
  for(int i = 0; i < n; i++){
      avaliarForca(i); //avalia a forca da solucao i
  }

  x, y = selecionarPais(i); //seleciona 2 pais mais fortes
  z = casarPais(x, y); //gera um conjunto z de filhos dos 2 pais
  mutacao(z); //faz uma mutacao nos elementos do conjunto z
  newPop = accept(z); //gera uma nova populacao
                      //com os melhores filhos apos a mutacao
}
```

3 ANÁLISE DE COMPLEXIDADE

Nesta sessão sera demonstrada a analise de complexidade do melhor e do pior caso dos códigos.

3.1 Forca Bruta

O algoritmo de forca bruta gera todas a permutações de caminhos a serem percorridos. Para cada possibilidade é calculado a distancia entre cada cidade do caminho gerado, e então compara-se essa distância com a menor distância encontrada até o momento. Caso a distancia da permutação atual seja a menor no momento ela é salva e o algoritmo parte para a próxima. Por isso a disposição de dados do melhor e o pior caso é igual, já que são testadas todas permutações. A operação mais relevante nesse caso é o calculo das permutações que é $(n-1)!$. Também há o custo de de calcular a distancia do caminho percorrido que é $n!$, pois para cada caminho devemos calcular a distancia das $n-1$ arestas, assim como a distancia da ultima para a primeira aresta. Portanto podemos concluir que a complexidade da força bruta, em qualquer caso, é:

$$O(n!)$$

Ressalta-se ainda que a **operação relevante** é a troca de elementos no vetor (mecanismo utilizado para gerar as permutações).

3.2 Branch and Bound

A complexidade do Branch and Bound depende diretamente da maneira de calcular o limite inferior. No Branch and Bound caso não houver nenhum corte de vértice, a complexidade permanece a mesma do algoritmo de Força Bruta, que é $O(n!)$. Contudo na pratica essa situação é extremamente rara, e há uma chance muito grande de cortar vértices, o que melhora a performance do algorítmo.

No melhor caso, calcula-se apenas o resultado do primeiro caminho encontrado (pois esse seria o melhor, e portanto todos os caminhos consecutivos seriam cortados). Temos assim:

n (custo para achar a menor distancia)

$+$

$(n-2!)$ (custo para achar as distâncias parciais a partir da permutação inicial)

O que resultaria, novamente, em uma complexidade de: $O(n!)$, mesmo para o melhor caso. A **operação relevante** para esse cálculo é a troca de elementos no vetor (mecanismo utilizado para gerar as permutações).

Temos também o método para calcular o limite inferior de um vértice, que faz apenas **uma soma**, mas faz essa soma para todos os vértices, tendo assim uma complexidade de n, o que não influenciaria de qualquer maneira na complexidade do programa ao todo.

3.3 Programação Dinâmica

No método de programação dinâmica, temos, a partir de um único vértice inicial, 2^n possibilidades de visitação das cidades, demonstrado pela máscara de bits, conforme abaixo:

0001 - apenas a primeira cidade visitada. Caminho atual: 1

0011 - primeira e segunda cidades visitadas. Caminho atual: 1 -> 2

0101 - primeira e terceira cidades visitadas. Caminho atual: 1 -> 3

...

Assim em diante, até verificar todas as possibilidades de visitação. Entretanto, para cada uma dessas cidades, verificamos qual a cidade seguinte que nos daria o melhor resultado. Temos assim, $n * 2^n$ sub-problemas. Cada um deles executa em tempo linear, resultando finalmente em uma complexidade de:

$$O(n^2 * 2^n)$$

A **operação relevante** desse código é a soma das distâncias parciais, que é feita recursivamente, 1 vez por sub-problema.

3.4 Algoritmo Genético

O algoritmo genético é uma meta-heurística(BIANCHI et al., 2009) inspirado no processo de seleção natural. Tem se m gerações possíveis, e quanto maior esse valor, maior a chance de achar a solução otimizada. Para cada geração são feitas diversas seleções, que rodam em tempo linear $O(n)$, diversas recombinações, também $O(n)$, 1 mutação, e diversas avaliações, novamente $O(n)$. A complexidade resultante é:

$$O(n * m)$$

4 TESTES

Para realizar os testes foi usado um arquivo de entrada com coordenadas das cidades de acordo com o valor de N cidades, e também escrito um script para automatizar os testes. A saída de todas execuções foram gravadas em arquivos separados para cada paradigma. Os testes foram realizados no sistema operacional Ubuntu.

System : Kernel: 4.10.0-38-generic x86 64 (64 bit)
Desktop: Unity 7.4.0 Distro: Ubuntu 16.04 xenial

Machine : System: ASUSTeK (portable) product: GL502VMK v: 1.0
Mobo: ASUSTeK model: GL502VMK v: 1.0
Bios: American Megatrends v: GL502VMK.308 date: 05/03/2019

CPU : Quad core Intel Core i7-7700HQ (-HT-MCP-) cache: 6144 KB
clock speeds: max: 3800 MHz
1: 3477 MHz
2: 3488 MHz
3: 3623 MHz
4: 3484 MHz
5: 3441 MHz
6: 3501 MHz
7: 3505 MHz
8: 3691 MHz

A saída dos programas esta no padrão, menor distancia percorrida, ordem das cidades visitadas e tempo gasto para achar a solução.

Tabela 3 – Saída dos programas

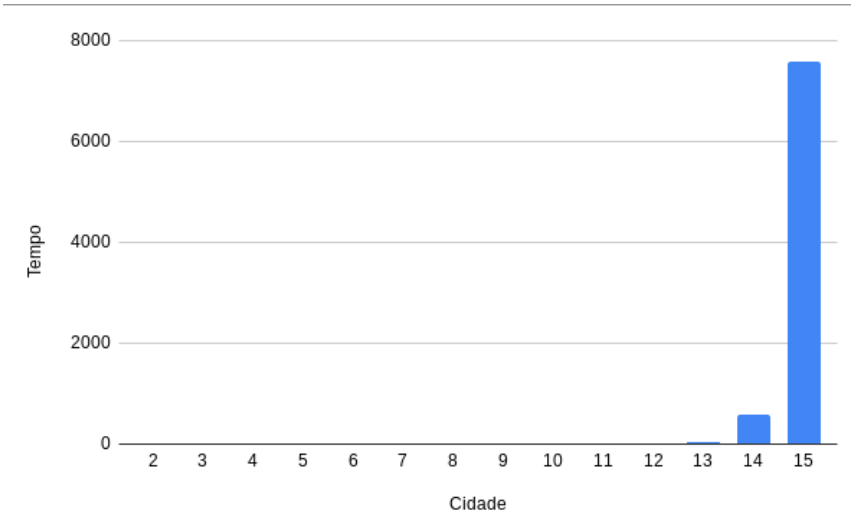
Menor distancia: 3394.11 Cidades visitadas : 1 2 3 5 6 8 9 11 12 13 10 7 4 Tempo: 45.9293

Para cada número de cidades, foram feitas 20 execuções do programa, e foi calculada a média aritmética dos tempos dessas execuções.

4.1 Forca Bruta

Utilizando o método Força Bruta obtivemos as seguintes médias de tempo para cada número de cidades:

Nº de cidades	Tempo médio de 20 execuções
2 Cidades	0 segundos
3 Cidades	0 segundos
4 Cidades	0 segundos
5 Cidades	0 segundos
6 Cidades	0 segundos
7 Cidades	0 segundos
8 Cidades	0 segundos
9 Cidades	0 segundos
10 Cidades	0 segundos
11 Cidades	0,3 segundos
12 Cidades	3,4 segundos
13 Cidades	44,95 segundos
14 Cidades	584,7 segundos
15 Cidades	7582,5 segundos



4.2 Branch and Bound

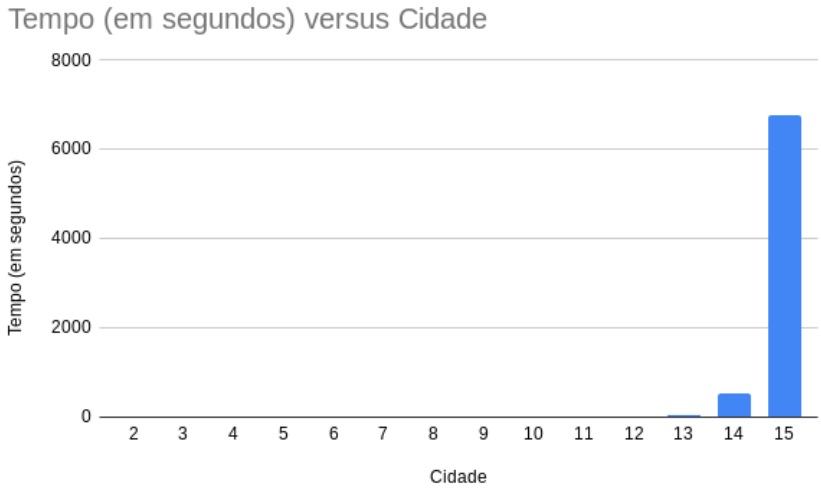
Utilizando o método Branch and Bound obtivemos as seguintes médias de tempo para cada número de cidades:

Nº de cidades	Tempo médio de 20 execuções
2 Cidades	0 segundos
3 Cidades	0 segundos
4 Cidades	0 segundos
5 Cidades	0 segundos
6 Cidades	0 segundos
7 Cidades	0 segundos
8 Cidades	0 segundos
9 Cidades	0 segundos
10 Cidades	0 segundos
11 Cidades	0 segundos
12 Cidades	0,05 segundos
13 Cidades	0,1 segundos
14 Cidades	0,55 segundos
15 Cidades	1,85 segundos
16 Cidades	5,7 segundos
17 Cidades	22,6 segundos
18 Cidades	82,95 segundos
19 Cidades	356,05 segundos
20 Cidades	1163,3 segundos
21 Cidades	2453,4 segundos

Devemos observar que o branch and bound não caiu em seu pior caso em nenhum momento. Adicionalmente, por ser mais eficiente em seu caso médio, ele só começou a estourar com 6 entradas a mais que o método Força Bruta. Vale ressaltar que nos nossos arquivos de teste, o branch and bound sempre achava a solução ótima na primeira iteração, e por isso foi significantemente mais rápido. Num caso normal, ele seria um meio termo entre a programação dinamica e a força bruta.

4.3 Programação Dinâmica

Nº de cidades	Tempo médio de 20 execuções
3 Cidades	0 segundos
4 Cidades	0 segundos
5 Cidades	0 segundos
6 Cidades	0 segundos
7 Cidades	0 segundos
8 Cidades	0,0004 segundos
9 Cidades	0,0029 segundos
10 Cidades	0,0347 segundos
11 Cidades	0,3391 segundos
12 Cidades	3,9703 segundos
13 Cidades	42,846 segundos
14 Cidades	536,24 segundos
15 Cidades	6770,1 segundos



4.4 Algoritmo Genético

Nº de cidades	Tempo médio de 20 execuções
3 Cidades	0,006 segundos
4 Cidades	0,008 segundos
5 Cidades	0,012 segundos
6 Cidades	0,016 segundos
7 Cidades	0,016 segundos
8 Cidades	0,020 segundos
9 Cidades	0,023 segundos
10 Cidades	0,028 segundos
11 Cidades	0,033 segundos
12 Cidades	0,041 segundos
13 Cidades	0,044 segundos
14 Cidades	0,058 segundos
15 Cidades	0,063 segundos

O algoritmo genético foi de longe o mais rápido, contudo devemos levar em consideração que ele é uma heurística e que a chance de acerto depende proporcionalmente das gerações e inversamente proporcional a quantidade de cidades.

5 CONCLUSÃO

A complexidade para resolver o problema do caixeiro viajante é exponencial. Ele é considerado um problema NP-difícil, em maquinas nao determinísticas também não seria possível encontrar uma solução em tempo polinomial para este problema. Foram implementadas e demonstradas algumas abordagens para o problema, cada uma com sua particularidade. Em algumas somente trocando um pouco a logica conseguimos melhores resultados, em outros casos devemos abrir mão de outros recursos como espaço de memoria para melhorar a execução.

A solução dinâmica não é simples de ser entendida, mas é a com menor complexidade de tempo. Contudo há uma grande utilização de memoria para o armazenamento das informações necessárias. O algoritmo de força bruta é bem simples, porém apresenta a pior complexidade. A estrategia de branch and bound em seu pior caso é igual a de força bruta, porém no melhor caso se torna muito mais eficiente se tornando uma das melhores opções. Por sua vez a solução genética é uma meta-heurística e pode não gerar a solução ótima em certos casos, e sim uma aproximação.

Caso seja conhecida a disposição de entrada dos dados pode ate mesmo ser feito alguma operação para otimizar ainda mais a solução do problema. Assim como em casos que não se precisa da otimização, mas sim de um valor bem próximo do ideal podemos utilizar heurísticas que tem um custo computacional bem menor.

REFERÊNCIAS

BIANCHI, Leonora et al. A survey on metaheuristics for stochastic combinatorial optimization. **Natural Computing**, v. 8, n. 2, p. 239–287, Jun 2009. ISSN 1572-9796. Disponível em: <https://doi.org/10.1007/s11047-008-9098-4>.

CASTRO, MARCOS. **Genetic TSP**. [S.l.]: GitHub, 2014. <https://github.com/marcoscastro/tsp_genetic>.

DESCONHECIDO, Autor. **Optimal Solution for TSP using Branch and Bound**. Disponível em: <http://lcm.csa.iisc.ernet.in/dsa/node187.html>.

SABAN, Asher; WALTUCH, Liat. Genetic algorithm for the traveling salesman problem. p. 8–9, 2012.