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

Trabalho de Projeto e Análise de Algoritmos - Caixeiro Viajante*

Luiz Gustavo Bragança dos Santos[1]
Pedro Henrique Silva Xavier[2]

*Artigo apresentado ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como pré-requisito para obtenção do título de Bacharel em Ciência da Computação.

[1]Aluno, Ciência da Computação, Brasil, luiz.braganca@sga.pucminas.br.
[2]Aluno, Ciência da Computação, Brasil, phsxavier@sga.pucminas.br.

Sumário

Lista de Figuras

1 INTRODUÇÃO

O Problema do Caixeiro Viajante (PCV) trata-se de, dado um grafo *G*, determinar o circuito de menor custo que passa por todos os vértices somente uma vez, ou seja encontrar um circuito Hamiltoniano mínimo de *G*. O PCV é um problema de otimização NP-Complexo, o qual só pode sê-lo, se e somente se, existir um problema NP-Completo *L*, tal que seja Turing-redutível em tempo polinomial para *H*, sendo um dos problemas de otimização mais amplamente estudados, abordado por centenas de artigos. A origem do problema é desconhecida e alguns matemáticos já estudavam formulações desde o século XIX.

O objetivo desse trabalho é estudar as implementações de algoritmos que resolvem o PCV utilizando os seguintes paradigmas:

- Força Bruta;
- *Branch and Bound*;
- Programação Dinâmica;
- Algoritmo Genético;

Para cada paradigma apresentado, faremos a análise de complexidade das implementações. Serão analisados somente os métodos que interagem diretamente com o PCV, ou seja, somente a busca pelo circuito Hamiltoniano mínimo será considerada. Será citada a operação mais relevante e a configuração de entrada para melhor e pior caso.

2 APLICAÇÃO

Neste trabalho será abordado um escopo específico do PCV, baseado na analogia de percorrer cidades. A primeira linha da entrada consiste em um inteiro *n* indicando o número de cidades (vértices) a serem visitadas. As *n* linhas seguintes informam as coordenadas para as cidades respectivamente.

Exemplo de entrada (quatro cidades):

```
1 4
2 100  100
3 900  100
4 900  900
5 100  900
```

Nesta aplicação as arestas que ligam os vértices são bidirecionais, ou seja, todas as estradas existentes no mapa são de mão dupla. O peso das arestas corresponde a distância euclidiana entre as cidades. O grafo gerado para analisar o problema é sempre um grafo completo, logo, todas as cidades estão conectadas.

A saída do programa consiste em duas linhas contendo, a primeira com a soma das distâncias, que seria o peso total das arestas percorridas e a segunda linha correspondente a uma sequência de números, que indicam a ordem em que as cidades foram visitadas.

Exemplo de saída (correspondente ao exemplo de entrada a cima):

```
1 3200
2 1  2  3  4
```

Além disso, uma análise dos tempos de execução de cada algoritmo apresentando o tempo médio, desvio padrão, e intervalo de confiança variando o tamanho do problema. Os casos de teste serão gerados aleatoriamente, via código e os dados citados são calculados durante a execução.

3 FORÇA BRUTA

O Paradigma Força Bruta, também conhecido como exaustão, consiste numa técnica trivial de solução de problemas que visa testar todas as combinações possíveis para um problema, no caso todos os caminhos até encontrar o circuito Hamiltoniano mínimo.

3.1 Implementação

A implementação da técnica de força bruta foi feita recursivamente onde, partindo do vértice 1 percorremos todos os caminhos possíveis fazendo uma permutação entre as arestas do vértice atual, assim que um circuito é formado verificamos se ele é o melhor caso.

O código abaixo corresponde à inicialização da execução criando todas as variáveis auxiliares necessárias para encontrar o circuito hamiltoniano mínimo para a instância de *Graph* em questão:

```
1  vector<int> Graph::bruteForce() {
2
3      time_t inicio, fim;
4      vector<int> cidades;
5      vector<int> resposta;
6      cidades.push_back(0);
7      this->best = INFINITE;
8
9      inicio = clock();
10     resposta = bruteForceR(0, 0, 0, cidades, cidades);
11     fim = clock();
12     timeT = fim - inicio;
13
14     return resposta;
15 }
```

A permutação ocorre no trecho de código abaixo. A em cada vértice testamos todas as arestas que alcançam vértices ainda não contidos no caminho formado até agora, tentando formar um circuito.

```
1  vector<int> Graph::bruteForceR(int a, double res, double bestR, vector<int>
       ↪ cidades,  vector<int> bestC) {
2
3      for (int i = 0; i < n; i++) {
4          if (!isIn(i, cidades)) {
5              res += adj[a][i];
6              cidades.push_back(i);
7              bestC = bruteForceR(i, res, bestR, cidades, bestC);
```

```
8                bestR = res;
9
10               if (cidades.size() == n && (bestR + adj[0][cidades[cidades.size
                     ↪ () - 1]]) < this->best) {
11                   this->best = bestR + adj[0][cidades[cidades.size() - 1]];
12                   bestC = cidades;
13               }
14
15               res -= adj[a][i];
16               cidades.pop_back();
17           }
18       }
19       return bestC;
20
21   }
```

3.2 Análise de Complexidade

Consideramos a operação relevante a manipulação do vetor correspondente ao caminho, ou seja tanto a operação de inserir no vetor (*push_back*) quanto a de retirar do vetor (*pop_back*). O método recursivo *bruteForceR()* tem a sua complexidade dada pela equação de recorrência:

$$\begin{cases} f(n) = (n-1) * f(n-1) + 1 \\ f(1) = 2 \end{cases}$$

A equação de recorrência pode ser reduzida para:

$$\Gamma(n) + e\Gamma(n, 1)$$

Sabendo que $\Gamma(n)$ é equivalente a $\Gamma(n) = (n-1)!$ concluímos que a ordem de complexidade do algoritmo de exaustão é $O(n!)$, ou seja o algoritmo tem tempo fatorial.

3.3 Testes

Para o algoritmo de Força Bruta, foram feitos vários testes. A entrada padrão do programa, variando 4 a 12, de dois em dois, conforme o exemplo:

```
1 4
2 100  100
3 900  100
4 900  900
5 900  100
6
7 Custo:  2731.37
```

```
Caminho: 1 2 4 3
Tempo Gasto: 0.000117 s

[...]

6
4 23
4 24
4 25
4 26
4 27
4 28

Custo: 5
Caminho: 1 2 3 4 5 6
Tempo Gasto: 0.002672 s

[...]

8
6 91
6 12
6 13
6 200
6 15
6 26
6 17
6 18

Custo: 376
Caminho: 1 2 3 5 7 8 6 4
Tempo Gasto: 0.048630 s

[...]

10
50 21
10 22
30 23
10 24
10 25
120 26
10 27
15 28
10 29
90 30

Custo: 207.594
Caminho: 1 6 10 8 9 7 5 4 2 3
```

```
57 Tempo Gasto: 3.433317 s
58
59 [...]
60
61 12
62 12 11
63 12 12
64 12 13
65 12 14
66 12 15
67 12 16
68 12 17
69 12 18
70 12 19
71 12 20
72 12 21
73 12 22
74
75 Custo: 22
76 Caminho: 1 2 3 4 5 6 7 8 9 10 11 12
77 Tempo Gasto: 555.886201 s
```

4 *BRANCH AND BOUND*

Branch and Bound (B&B) é um paradigma de programação que, semelhante ao Força Bruta, enumera todas as soluções possíveis para uma otimização e testa até encontrar a solução ótima, mas diferenciando-se da exaustão, o algoritmo B&B verifica a cada passo se a solução ainda é viável. No caso do PCV o B&B, a cada aresta inserida no caminho verifica se o circuito mínimo encontrado é menor que o caminho que está sendo gerado.

4.1 Implementação

A implementação do *Branch and Bound* foi feita inspirando-se largamente na implementação por exaustão, a única diferença real entre um algoritmo e o outro é a verificação do caminho gerado, comparando com o melhor caminho encontrado até o momento, para descartar a solução caso ela deixe de ser viável.

O código abaixo corresponde à inicialização da execução criando todas as variáveis auxiliares necessárias para encontrar o circuito hamiltoniano mínimo para a instância de *Graph* em questão:

```
1  vector<int> Graph::branchBound(){
2      time_t inicio, fim;
3      vector<int> cidades;
4      vector<int> resposta;
5      cidades.push_back(0);
6      this->best = INFINITE;
7
8      inicio = clock(); //comeca o tempo
9      resposta = branchBoundR(0, 0, 0, cidades, cidades);
10     fim = clock(); //para o tempo
11     timeT = fim - inicio;
12
13     return resposta;
14 }
```

A permutação é feita recursivamente no método abaixo, juntamente com a validação com a solução atual:

```
1  vector<int> Graph::branchBoundR(int a, double res, double bestR, vector<int
       ↪ >cidades, vector<int> bestC){
2      for(int i = 0; i < n; i++){
3          if (!isIn(i, cidades)){
4              res += adj[a][i];
5              cidades.push_back(i);
6
```

```
7               if(res < this->best){ // validacao
8                   bestC = branchBoundR(i, res, bestR, cidades, bestC);
9                   bestR = res;
10
11                  if(cidades.size() == n && (bestR + adj[0][i]) < this->best)
                        ↪ {
12
13                      this->best = bestR + adj[0][i];
14                      bestC = cidades;
15                  }
16              }
17              res -= adj[a][i];
18              cidades.pop_back();
19          }
20      }
21      return bestC;
22  }
```

Como se vê no código acima, a semelhança entre o força bruta é grande,

4.2 Análise de Complexidade

Consideramos a operação relevante a manipulação do vetor correspondente ao caminho, ou seja tanto a operação de inserir no vetor (*push_back*) quanto a de retirar do vetor (*pop_back*). O método recursivo *bruteForceR()* tem a sua complexidade dada pela equação de recorrência:

4.3 Testes

Para o algoritmo de Branch and Bound, foram feitos vários testes. A entrada padrão do programa, variando 4 a 12, de dois em dois, conforme o exemplo:

```
4
100 100
900 100
900 900
900 100

Custo: 2731.37
Caminho: 1 2 4 3
Tempo Gasto: 0.000230 s

[...]

6
4 23
4 24
4 25
4 26
4 27
4 28

Custo: 5
Caminho: 1 2 3 4 5 6
Tempo Gasto: 0.001875 s

[...]

8
6 91
6 12
6 13
6 200
6 15
6 26
6 17
6 18

Custo: 376
Caminho: 1 2 3 5 7 8 6 4
Tempo Gasto: 0.031438 s

[...]

```

```
10
50 21
10 22
30 23
10 24
10 25
120 26
10 27
15 28
10 29
90 30

Custo: 207.595
Caminho: 1 6 10 8 9 7 5 4 2 3
Tempo Gasto: 0.678667 s

[...]

12
12 11
12 12
12 13
12 14
12 15
12 16
12 17
12 18
12 19
12 20
12 21
12 22

Custo: 22
Caminho: 1 2 3 4 5 6 7 8 9 10 11 12
Tempo Gasto: 4.004898 s
```

5 PROGRAMAÇÃO DINÂMICA

5.1 Implementação

A implementação da técnica de Programação Dinâmica foi feita recursivamente onde, partindo do vértice 1 é gerada uma árvore de combinações que é visitada recursivamente. O algoritmo divide essa árvore em vários subproblemas, onde cada um é solvido separadamente e retorna o seu valor para um outro problema assim sucessivamente até voltar a cidade inicial. Foi usado o conceito de máscara, onde cada cidade ramificada da árvore recebe uma máscara (id), dessa maneira podemos controlar se um determinado subproblema já foi computado evitando assim um recálculo do mesmo e evitando o cálculo desnecessário, por exemplo a distância da cidade 0 (máscara 0001) para ela mesmo.

O código da Programação Dinâmica teve a implementação base diferente dos demais apresentados nesse artigo, inicialmente é setado um valor máximo para as estruturas de suporte, são elas, a matriz G utilizada para guardar o resultado mínimo do caminho, matriz P utilizada para guardar o caminho que será realizado para o custo mínimo, matriz ADJ utilizada para fazer as ligações de aresta de um vértice no outro e dois vetores auxiliares para o armazenamento das coordenadas vetor x e y. O próximo passo é realizar a leitura da entrada e preencher a matriz ADJ.

O código abaixo inicia a busca do menor caminho para o usuário:

```
1  void TSP(){
2
3      int i,j;
4      //g(i,S) caminho mais curto entre S e i
5      for(i = 0;i < n;i++)
6          for(j = 0;j < npow ;j++)
7                  g[i][j] = p[i][j]= -1;
8
9      for(i = 0;i < n;i++)
10         g[i][0] = adj[i][0];//g(i,nullset)= aresta entre il
11
12     int resultado = distanciaSubProblemas(0,npow-2);
13
14     printf("Custo : %d\n",resultado);
15
16     printf("Caminho:\n1 ");
17     getpath(0,npow-2);
18     printf("1\n");
19
20 }
```

O resultado é obtido pelo método *distanciaSubProblemas()*, a cidade inicial é removida e então o método começa a verificar os vértices não visitados de forma recursiva até encontrar

o último e gera um valor de distância do primeiro vértice a ele e depois verifica se o caminho até o momento é o menor. Isso torna o nosso método um TD (*top-down*)

```
1  int distanciaSubProblemas(int start,int set){
2
3      int masked, mask, temp, i;
4      int resultado = INT_MAX; //guarda o minimo
5
6      if(g[start][set]!=-1)// verifica se tem subproblema que ja foi
           ↪ resolvido
7
8          return g[start][set];
9
10     for(i = 0; i < n;i++){
11             // npow agora e -1 porqque comecamos do vertice 0
12             //remove um vertice
13             mask = (npow-1) - ( 1 << i );
14
15             masked = set&mask;
16
17             if(masked != set)//caso um mesmoc conjunto seja gerado, entao a
                   ↪  mascara tem q ser diferente.
18             {
19                 temp = adj[start][i] + distanciaSubProblemas(i,masked);//
                       ↪ calcula o conjunto com o vertice ja removido
20
21                 if(temp < resultado)
22
23                     resultado = temp,p[start][set] = i;
24             }
25         }
26         return g[start][set] = resultado;//return minimum
27 }
```

O caminho é obtido pelo método *getPah()*, que também é recursivo e se orienta pela máscara de cada vértice:

```
1  void getpath(int start,int set){
2
3      if(p[start][set]==-1) return;//reached null set
4      int x = p[start][set] ;
5      int mask = (npow-1)-(1<<x);
6      int masked = set&mask;//remove p from set
7      printf("%d ",(x + 1));
8      getpath(x,masked);
9  }
```

5.2 Análise de Complexidade

Consideramos como operação relevante a atribuição de valores na matriz. Para um conjunto de tamanho n, consideramos n-2 subconjuntos de tamanho n-1, de forma que todos os subconjuntos não possuam n neles. Existe no máximo $O(n * 2^n)$ subproblemas portanto no algoritmo usado temos $O(n^2 * 2^n)$ como complexidade de tempo

5.3 Testes

Para o algoritmo de Programação Dinâmica, foram feitos vários testes. A entrada padrão do programa, variando 4 a 12, de dois em dois, conforme o exemplo:

```
4
100 100
900 100
900 900
900 100

Custo: 2731
Caminho: 1 2 4 3 1
Tempo Gasto: 0.000131 s

[...]

6
4 23
4 24
4 25
4 26
4 27
4 28

Custo: 10
Caminho: 1 2 3 4 5 6 1
Tempo Gasto: 0.000182 s

[...]

8
6 91
6 12
6 13
6 200
6 15
6 26
6 17
```

```
6 18

Custo: 376
Caminho: 1 2 3 5 7 8 6 4 1
Tempo Gasto: 0.000351 s

[...]

10
50 21
10 22
30 23
10 24
10 25
120 26
10 27
15 28
10 29
90 30

Custo: 227
Caminho: 1 2 4 5 7 9 8 3 10 6 1
Tempo Gasto: 0.001253 s

[...]

12
12 11
12 12
12 13
12 14
12 15
12 16
12 17
12 18
12 19
12 20
12 21
12 22

Custo: 22
Caminho: 1 2 3 4 5 6 7 8 9 10 11 12 1
Tempo Gasto: 0.005295 s
```

6 ALGORITMO GENÉTICO

6.1 Implementação

A implementação da técnica da técnica do Algorítimo Genético foi feita estabelecendo inicialmente uma população a qual é inicializada por um conjunto aleatório de rotas que são geradas.Posteriormente as melhores rotas desta população são escolhidas para um cruzamento(*crossover*) gerando novas rotas filhas, as rotas filhas serão possivelmente melhores que as rotas anteriores, pais. O cruzamento é realizado através da escolha de um ponto aleatório das rotas pais e assim os filhos são gerados a partir do ponto aleatório de corte.

```
1  Population* GeneticAlgorithm::crossoverPopulation(Population* population)
2  {
3      // definir dados
4      int x;
5      // criando uma nova populacao
6      Population* crossoverPopulation = new Population(population->getRoutes
           ↪ ().size(), getInitialRoute());
7
8      for(x = 0; x < NUMB_OF_ELITE_ROUTE; x++)
9      {
10         set(x, crossoverPopulation->getRoutes(), population->getRoutes().at
               ↪ (x));
11     }// fim for
12
13     for(x = NUMB_OF_ELITE_ROUTE; x < crossoverPopulation->getRoutes().size
           ↪ (); x++)
14     {
15         Route* route1 = selectTournamentPopulation(population)->getRoutes()
               ↪ .at(0);
16
17         Route* route2 = selectTournamentPopulation(population)->getRoutes()
               ↪ .at(0);
18
19         set(x, crossoverPopulation->getRoutes(), crossoverRoute(route1,
               ↪ route2));
20     }// fim for
21     return crossoverPopulation;
22 }// fim crossoverPopulation()
23
24
25 Route* GeneticAlgorithm::crossoverRoute(Route* route1, Route* route2)
26 {
27     // definir dados
28     int x;
29
```

```
30      Route* crossoverRoute = new Route(getInitialRoute().size());
31      Route* tempRoute1 = route1;
32      Route* tempRoute2 = route2;
33
34      if(gerarAleatorio() < 5)
35      {
36          tempRoute1 = route2;
37          tempRoute2 = route1;
38      }// fim if
39
40      for(x = 0; x < crossoverRoute->getCities().size()/2; x++)
41      {
42          set(x, crossoverRoute->getCities(), tempRoute1->getCities().at(x));
43      }// fim for
44
45      return fillNullsInCrossoverRoute(crossoverRoute, tempRoute2);
46  }// fim crossoverRoute()
```

Temos no programa classes que auxiliam no estabelecimento da população assim como o gerenciamento dessas para aplicação do cruzamento gerando assim rotas filhas assim como temos também uma classe que auxilia a construção das rotas:

```
1   Population::Population(int populationSize, vector<City*> cities)
2   {
3       for(int i = 0; i < populationSize; i++)
4       {
5           routes.push_back(new Route(cities));
6       }// fim for
7   }// fim construtor
8
9
10  vector<Route*> Population::getRoutes()
11  {
12      return this->routes;
13  }// fim getRoutes()
14
15
16  /**
17   * Compara se uma rota e' melhor que outra para a ordenacao.
18   * @param route1
19   * @param route2
20   * @return true or false
21   */
22  bool compare(Route* route1, Route* route2)
23  {
24      return(route1->getFitness() > route2->getFitness());
25  }// fim compare()
```

Rotas

```
....
Route::Route(vector<City*> citiesP)
{
    this->isFitnessChanged = true;
    this->fitness = 0.0;

    // copia do vetor citiesP para o vetor cities
    cities.reserve(citiesP.size());
    copy(citiesP.begin(),citiesP.end(),back_inserter(cities));

    // devo fazer um shuffle aqui, embaralhar o vector de Rotas
    // ****** SHUFFLE ******

    // seed aleatoria para o random
    random_shuffle ( cities.begin(), cities.end(), myrandom );
    // ****** FIM SHUFFLE ******
}// fim construtor


/**
  * Calcula a distancia de um caminho
  */
double Route::calculateTotalDistance()
{
    int citiesSize = this->cities.size();

    double sum = 0;

    for(int x = 0; x < citiesSize; x++)
    {
        int cityIndex = x;

        if(cityIndex < citiesSize - 1)
        {
            sum += cities.at(x)->measureDistance(cities.at(cityIndex + 1));
        }// fim if
    }// fim for

    sum += cities.at(0)->measureDistance(cities.at(citiesSize-1));

    return sum;
}// fim calculateTotalDistance()
...
```

6.2 Análise de Complexidade

A análise de complexidade foi feita considerando a operação crucial do algoritmo genético, o *crossover*) que é essencial para estabelecer rotas filhas temos então a complexidade de tempo em $(O(N))$

6.3 Testes

Para o algoritmo de Algoritmo Genético, foram feitos vários testes. A entrada padrão do programa, variando 4 a 12, de dois em dois, conforme o exemplo:

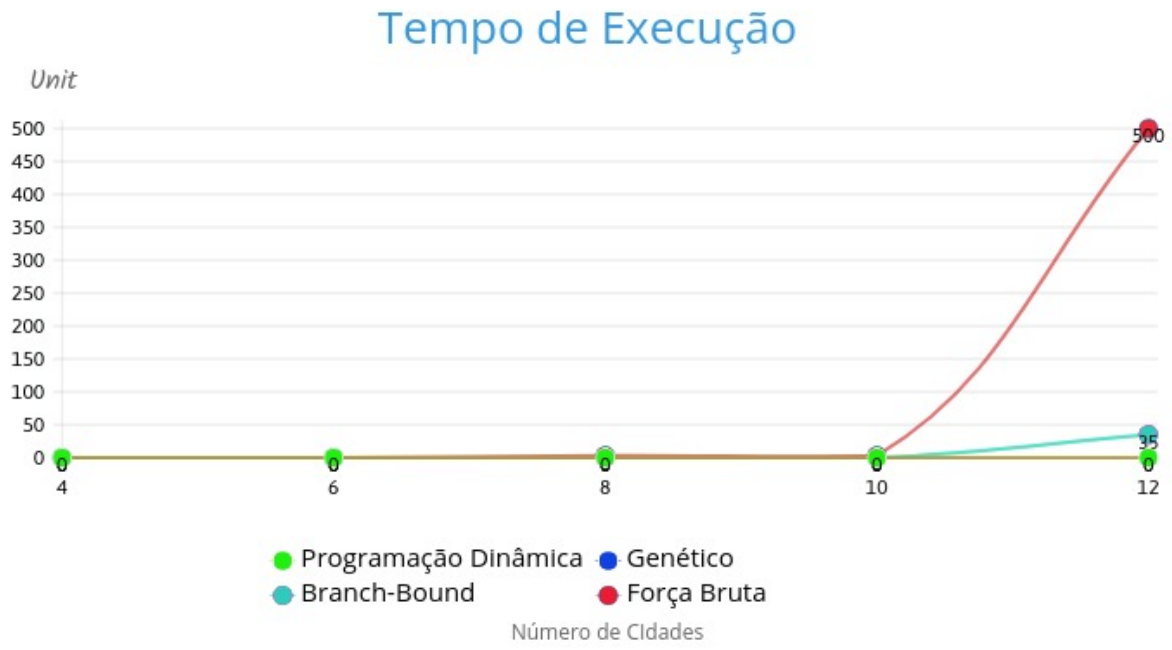
```
4
100 100
900 100
900 900
900 100

Custo: 2731.37
Caminho: D C A B
Tempo Gasto: 0.018143 s

[...]

6
4 23
4 24
4 25
4 26
4 27
4 28

Custo: 10
Caminho: E B A C D F
Tempo Gasto: 0.025192 s

[...]

8
6 91
6 12
6 13
6 200
6 15
6 26
6 17
6 18

Custo: 376
Caminho: A F D C E G H B
Tempo Gasto: 0.019633 s

[...]

10
```

```
50 21
10 22
30 23
10 24
10 25
120 26
10 27
15 28
10 29
90 30

Custo: 467.775
Caminho: E F G I A D H J C B
Tempo Gasto: 0.021004 s

[...]

12
12 11
12 12
12 13
12 14
12 15
12 16
12 17
12 18
12 19
12 20
12 21
12 22

Custo: 50
Caminho: H K E I D B A L G C F J
Tempo Gasto: 0.033759 s
```

7 CONCLUSÃO

Como podemos observar com o gráfico abaixo, o algoritmo mais demorado é o Força Bruta, quando possui 10 cidades, por possuir uma complexidade de O(*n!*), quanto mais entradas possuir mais tempo levará para terminar, pois ele vai testando todos caminhos, o que difere dos outros algoritmos em questão. E o mais rápido de todos é o Algoritmo Genético, gastando menos tempo do que o Programação Dinâmica.

Figure 1 - Comparação entre o tempo de execução



Source: Elaborado pelos autores.

8 ANEXOS

Os programas estão devidamente incluídos em suas pastas com os seus devidos nomes, seguido a lista mencionada na introdução. Dentro da pasta "Forca-Bruta", possui o arquivo forca-bruta.cpp e a sua biblioteca, esta contida na pasta de mesmo nome, "biblioteca"com o nome de Graph.h, os quais possuem a implementação do algoritmo e o seu arquivo de cabeçalho, respectivamente.

Dentro da pasta "Branch-and-Bound", possui o arquivo branch-and-bound.cpp e a sua biblioteca, esta contida na pasta de mesmo nome, "biblioteca"com o nome de Graph.h, os quais possuem a implementação do algoritmo e o seu arquivo de cabeçalho, respectivamente.

Dentro da pasta "Programacao-Dinamica", possui o arquivo programacao-dinamica.cpp, o qual possui a implementação do algoritmo.

E por fim, dentro da pasta "Algoritmo-Genetico", possui vários arquivos os quais ajudam a implementar o algoritmo, como City.cpp, Driver.cpp, GeneticAlgorithm.cpp, Population.cpp e Route.cpp, e suas respectivas bibliotecas, .h, ficam na pasta "bibliotecas".

Para gerar os casos de testes, foi implementado um arquivo chamado gerador.sh, o qual gera números aleatórios. Após gerá-los, colocamos no arquivo teste.txt, o qual possui 10 testes para cada N cidades.

Referências

AGUIAR, Marilton Sanchotene de. Análise formal da complexidade de algoritmos genéticos. 1998.

GOUVÊA, Elizabeth Ferreira et al. **Uma análise experimental de abordagens heurísticas aplicadas ao problema do caixeiro viajante**. 2006. Dissertação (Mestrado) — Universidade Federal do Rio Grande do Norte.

LACERDA, Estéfane GM de; CARVALHO, ACPLF de. Introdução aos algoritmos genéticos. **Sistemas inteligentes: aplicaçoes a recursos hıdricos e ciências ambientais**, v. 1, p. 99–148, 1999.

LAPORTE, Gilbert. The traveling salesman problem: An overview of exact and approximate algorithms. **European Journal of Operational Research**, Elsevier, v. 59, n. 2, p. 231–247, 1992.

TUTORIALS, ZA Software Development. **Traveling Salesman Problem (TSP) By Genetic Algorithms - JAVA 8 Tutorial**. 2018. Disponível em: <https://www.youtube.com/watch?v=Z3668A0zLCM>.

ZIVIANI, Nivio et al. **Projeto de algoritmos: com implementações em Pascal e C**. [S.l.]: Thomson, 2004. v. 2.