INSTITUTO FEDERAL
Sul de Minas Gerais

Campus
Poços de Caldas

Engenharia de Computação - 8º Período

Tópicos em Sistemas Inteligentes

Algoritmo ótimo / genético

Cristhian Cintra Barbosa

Sumário

Objetivo

Este documento foi desenvolvido na disciplina de Tópicos em Sistemas Inteligentes (TSI) no 2º semestre de 2022, tendo como objetivo facilitar a compreensão da análise e comparação dos algoritmos (ótimo e genético) para resolução do problema caixeiro viajante utilizando a linguagem JAVA.

2. Algoritmo ótimo

Tempo de execução (ms)

Algoritmo Ótimo tempo de execução		
Nº Cidades	Tempo execução (ms)	fitness
2	1	104
3	2	186
4	2	180
5	3	212
6	4	183
7	7	243
8	14	251
9	29	292
10	94	272
11	570	272
12	5142	282
13	64241	272
14	825564	267

(tabela 6)

Memória gasta (MB)

Algoritmo Ótimo memória gasta	
Nº Cidades	Memória gasta (MB)
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	5
10	6
11	6
12	6
13	6
14	8

(tabela 7)

3. Algoritmo genético

Para criação do algoritmo utilizou-se vários ArrayList contendo ArrayList do tipo Integer para fazer as manipulações necessárias nas seguintes funções. (figura 1 e 2).

```
static ArrayList<ArrayList<Integer>> populacaolimpa = new ArrayList<>();
static ArrayList<ArrayList<Integer>> populacao = new ArrayList<>();
static ArrayList<ArrayList<Integer>> filhos = new ArrayList<>();
static ArrayList<ArrayList<Integer>> pai = new ArrayList<>();
static ArrayList<ArrayList<Integer>> mae = new ArrayList<>();
static ArrayList<ArrayList<Integer>> elite = new ArrayList<>();
static ArrayList<ArrayList<Integer>> osmelhores = new ArrayList<ArrayList<Integer>>();
static ArrayList<ArrayList<Integer>> osalpha = new ArrayList<ArrayList<Integer>>();
```

(figura 1)

```
static ArrayList<Integer> criarpop(int tamanho) {...11 lines }

static void separapopulacao(int tamanhopop) {...15 lines }

static void cruzamento(int tamanhopop, int escolha) {...23 lines }

static void slot(int escolha, ArrayList<ArrayList<Integer>> vetor) {...11 lines }

static void mutacaoSwap(float taxaMutacao, int escolha, int tamanhopop) {...23 lines }

static void mutacaoInvert(float taxaMutacao, int escolha, int tamanhopop){...27 lines }

static void validarpop(int escolha) {...18 lines }

static void fitness(int graph[][], int escolha) {...22 lines }

static void Bubble(int escolha){...16 lines }

static ArrayList<Integer> createCopy(ArrayList<Integer> orginal) {...8 lines }

static void OrdenaMelhores(int escolha){...16 lines }
```

(figura 2)

criarpop() :

Um array auxiliar foi criado para receber os genes que irão representar o indivíduo. Cada gene é composto por vários valores (genomas). Depois esses valores são embaralhados e a função retorna um indivíduo completo.(figura 3).

```
static ArrayList<Integer> criarpop(int tamanho) {

    ArrayList<Integer> individuo = new ArrayList<>();

    for (int i = 0; i < tamanho; i++) {
        individuo.add(i);
    }
    Collections.shuffle(individuo);

    return individuo;
}
```

(figura 3)

separapopulacao() :

Função com o objetivo de separar o array da população em 2 auxiliares chamados de array PAI e MÃE.(figura 4).

```
static void separapopulacao(int tamanhopop) {

    int aux = (tamanhopop / 2);
    int aux2 = aux;

    for (int i = 0; i < aux; i++) {
        pai.add(populacao.get(i));
    }

    for (int i = 0; i < aux; i++) {
        mae.add(populacao.get(aux2));
        aux2++;
    }

}
```

(figura 4)

cruzamento() :

O cruzamento foi realizado pegando metade dos genomas do pai (lado esquerdo) e juntando com a metade da mãe (lado direito). Assim formando um novo indivíduo.(figura 5)

```
static void cruzamento(int tamanhopop, int escolha) {


    int genoma;
    int j = 0;

    for (int i = 0; i < tamanhopop / 2; i++) {

        ArrayList<Integer> novoindividuo = new ArrayList<>();

        for (int k = 0; k < escolha / 2; k++) {
            genoma = pai.get(j).get(k);
            novoindividuo.add(genoma);
        }
        for (int k = (escolha / 2); k < escolha; k++) {
            genoma = mae.get(j).get(k);
            novoindividuo.add(genoma);
        }
        filhos.add(novoindividuo);
        j++;

    }

}
```

(figura 5)

slot():

Adiciona ao final do array um espaço vazio para posteriormente ser guardado o fitness do indivíduo.(figura 6).

```
static void slot(int escolha, ArrayList<ArrayList<Integer>> vetor) {

    for (int j = 0; j < vetor.size(); j++) {

        if (vetor.get(j).size() == escolha) {
            vetor.get(j).add(0);
        }

    }

}
```

(figura 6)

mutacaoSwap() :

Com base na taxa de mutação foram escolhidos indivíduos aleatórios da população total para terem seus genomas trocados de posição de forma também aleatória.(figura 7).

```
static void mutacao(float taxaMutacao, int escolha, int tamanhopop) {

    populacao.addAll(filhos); // juntar pais e filhos no mesmo array

    int aux = (int) (tamanhopop * taxaMutacao);
    ArrayList<Integer> indmut = new ArrayList<>();

    for (int i = 0; i < aux; i++) {
        Random aleatorio = new Random();
        int valor = aleatorio.nextInt(escolha);
        indmut.add(valor);
    }

    for (int i = 0; i < indmut.size(); i++) {
        Random aleatoriol = new Random();
        int valorl = aleatoriol.nextInt(escolha); // quantidade de numeros que vao s
        int valor2 = aleatoriol.nextInt(escolha); // posicao do numero que vai ser r
        Collections.swap(populacao.get(indmut.get(i)), valorl, valor2);
    }


}
```

(figura 7)

mutacaoInvert():

Outra opção para mutação. Consiste em inverter totalmente o array do indivíduo. O mesmo é escolhido de forma aleatória.(figura 8).

```
static void mutacaoInvert(float taxaMutacao, int escolha, int tamanhopop){

    invert++;

    populacao.addAll(filhos); // juntar pais e filhos no mesmo array

    int aux = (int) (tamanhopop * taxaMutacao);
    ArrayList<Integer> indmut = new ArrayList<>();

    for (int i = 0; i < aux; i++) {
        Random aleatorio = new Random();
        int valor = aleatorio.nextInt(escolha);
        indmut.add(valor);
    }

    for (int i = 0; i < indmut.size(); i++) {

        Collections.reverse(populacao.get(indmut.get(i)));
        Collections.swap(populacao.get(indmut.get(i)), 0,escolha);
        int x=0;
        for (int j = 0; j < escolha-1; j++) {
            x++;
            Collections.swap(populacao.get(indmut.get(i)), j,x);
        }
    }

}
```

(figura 8)

validarpop() :

Depois da mutação devemos eliminar os indivíduos indesejados da população para prosseguir com avaliação de qualidade e posteriormente sua propagação por forma de cruzamento. Os indivíduos indesejados são aqueles que possuem o mesmo genoma repetido 2 ou mais vezes dentro do mesmo gene.(figura 9).

```
static void validarpop(int escolha){

    for (int aux = 0; aux < populacao.size(); aux++) {
            int flag = 0;
            for (int n = 0; n < escolha; n++) {
                for (int j = n + 1; j < escolha; j++) {

                    if (populacao.get(aux).get(n).equals(populacao.get(aux).get(j))) {
                        flag++;
                    }
                }
            }
            if (flag == 0) {
                populacaolimpa.add(populacao.get(aux));
            }
        }


}
```

(figura 9)

fitness() :

A qualidade do indivíduo será determinada por seu fitness que será medido analisando a matriz teste fornecida. Como no exemplo abaixo. (figura 10).

```
Matriz :
0 52 48 81

52 0 86 78

48 86 0 2

81 78 2 0
```

(figura 10)

O gene de cada indivíduo aqui representa um caminho com um peso que no fim da comparação com a matriz base um valor será atribuído a cada indivíduo da população. O programa está utilizando como base o arquivo Teste_100.txt para realizar os cálculos. (figura 11).

```
static void fitness(int graph[][], int escolha) {


    //int valorfinal = 0;

    for (int i = 0; i < populacaolimpa.size(); i++) {
        int valorfinal =0;
        int aux = 0;

        for (int j = 0; j < escolha - 1; j++) {
            //System.out.println(graph[populacao.get(i).get(j)][populacao.get(i).get(j + 1)]);
            aux = aux + graph[populacaolimpa.get(i).get(j)][populacaolimpa.get(i).get(j + 1)];
        }

        //System.out.println(graph[populacaolimpa.get(i).get(escolha - 1)][populacaolimpa.get(0).get(

        valorfinal = aux + graph[populacaolimpa.get(i).get(escolha - 1)][populacaolimpa.get(i).get(0)

        //System.out.println(populacaolimpa.get(i).get(escolha - 1)+" "+populacaolimpa.get(i).get(0)+

        populacaolimpa.get(i).set(escolha,valorfinal); //substituir uma pos e nao adicionar
        //populacaolimpa.get(i).add(valorfinal);
        //valorfinal = 0;
    }

}
```

(figura 11)

Bubble() :

Função com o objetivo de ordenar do menor para o maior cada indivíduo com base em seu valor fitness.Foi utilizado o algoritmo de ordenação bubble sort para realizar esta tarefa. (figura 12).

```
static void Bubble(int escolha){

    for (int i = 0; i < populacaolimpa.size(); i++) {
        for (int j = 0; j < populacaolimpa.size(); j++) {
            if (populacaolimpa.get(i).get(escolha) <= populacaolimpa.get(j).get(escolha)) {
                ArrayList<Integer> aux = populacaolimpa.get(i);
                populacaolimpa.set(i, populacaolimpa.get(j));
                populacaolimpa.set(j,aux);
            }
        }
    }


    elite.addAll(populacaolimpa);

}
```

(figura 12)

createCopy():

Faz uma cópia da referência na memória dos dados do array para não se perder e sobrescrever os dados nas próximas manipulações.(figura 13).

```
static ArrayList<Integer> createCopy(ArrayList<Integer> orginal) {

    ArrayList<Integer> copy = new ArrayList<Integer>();
    for (Integer s : orginal) {
        copy.add(s);
    }
    return copy;
}
```

(figura 13)

OrdenaMelhores():

Ordena o array final que contém apenas os melhores indivíduos de cada geração utilizando o algoritmo de ordenação bubble sort.(figura 14).

```
static void OrdenaMelhores(int escolha){

    for (int i = 0; i < osmelhores.size(); i++) {
        for (int j = 0; j < osmelhores.size(); j++) {
            if (osmelhores.get(i).get(escolha) <= osmelhores.get(j).get(escolha)) {
                ArrayList<Integer> aux = osmelhores.get(i);
                osmelhores.set(i, osmelhores.get(j));
                osmelhores.set(j,aux);
            }
        }
    }


    osalpha.addAll(osmelhores);

}
```

(figura 14)

No laço principal a população é criada em seguida adicionado um slot para o fitness. A população é separada em arrays pai e mãe e efetuado o cruzamento gerando os filhos.
Slots para o fitness são adicionados no array dos filhos. Duas opções para mutação foram adicionadas no código (swap e inversão). A população então é validada e calculado seu fitness logo após ordenada.(figura 15).

```
//EVOLUCAO
for (int i = 0; i < geracoes; i++) {

    //adicionando individuo no Array da população com tampop de tamanho
    for (int j = populacao.size(); j < tampop; j++) {

        populacao.add(criarpop(escolha));
    }

    slot(escolha, populacao);

    //separar população criando 2 array aux ( PAI e MAE )
    separapopulacao(tampop);

    cruzamento(tampop, escolha); // gerar filhos

    slot(escolha, filhos);



    //aplicando mutacao em individuos aleatorios do array populacao
    mutacaoSwap(taxaMutacao, escolha, tampop); //mutação por swap
    //mutacaoInvert(taxaMutacao, escolha, tampop); //mutação por inversão



    //seleção --> verificar numeros repetidos, eliminar, calcular o fitness e selecionar os % melhores
    //trabalhar com o Array populacaolimpa
    validarpop(escolha);
    fitness(graph, escolha);
    populacao.clear(); // limpando a população inicial para prox. gerações.




    //trabalhar com o Array elite
    Bubble(escolha);
```

(figura 15)

Depois da população sofrer todas essas manipulações ao fim teremos apenas um Array contendo todos indivíduos válidos e ordenados pelo fitness do melhor para o pior. Em seguida, a taxa de sobrevivência é aplicada sobre eles restando apenas os n% melhores. Assim irão se propagar na próxima geração com o objetivo de ajudar na criação de indivíduos cada vez melhores (menor fitness possível do conjunto).(figura 16).

```
    int sobreviventes = (int) (tampop * taxaSobrev);


    // passando os melhores para prox. gen
    for (int j = 0; j < sobreviventes; j++) {
        populacao.add(elite.get(j));
        osmelhores.add(createCopy(elite.get(j))); // salvando em outro lugar da memoria para nao perder referencia
    }

    populacaolimpa.clear();
    pai.clear();
    mae.clear();
    elite.clear();
    //System.out.println(i);
}
```

(figura 16)

Ao fim do código os ArrayList utilizados são limpos para poderem receber os novos indivíduos quantas vezes for necessário.

Algoritmo Genético mutação Swap tempo de execução		
Nº Cidades	Tempo execução (ms)	fitness
2	57381	104
3	35506	186
4	21442	180
5	16136	212
6	10450	183
7	9756	243
8	8639	251
9	8562	292
10	9684	272
11	9624	274
12	9238	301
13	9270	272
14	9229	365
15	10549	410
16	9754	388
100	19101	**3574**

(tabela 1)

Algoritmo Genético mutação Swap memória gasta	
Nº Cidades	Memória gasta (MB)
2	8
3	8
4	6
5	13
6	13
7	13
8	13
9	13
10	15
11	15
12	15
13	15
14	15
15	12
16	10
100	100

(tabela 2)

Algoritmo Genético mutação Inversão tempo de execução		
Nº Cidades	Tempo execução (ms)	fitness
2	56171	104
3	41588	186
4	22037	180
5	14115	212
6	10534	183
7	9328	243
8	8581	251
9	8340	292
10	9912	272
11	9195	276
12	9448	306
13	9211	320
14	9382	317
15	10549	428
16	11185	430
100	18844	**3719**

(tabela 3)

Algoritmo Genético mutação Inversão memória gasta	
Nº Cidades	Memória gasta (MB)
2	10
3	13
4	11
5	11
6	10
7	10
8	10
9	10
10	12
11	12
12	12
13	12
14	12
15	23
16	23
100	37

(tabela 4)

A mutação por swap apresentou ligeira vantagem no fitness se comparada a por inversão. Já para um número de nós elevados o resultado foi de swap[3574] inversão[3719]. (tabelas 1-4).
Para o teste acima, a seguinte configuração foi utilizada.

número de gerações : 10
tamanho da população : 8000
taxa de mutação : 0,6
taxa de sobrevivência : 0,1

O algoritmo genético teve melhor desempenho com nós maiores porém sofre com precisão podendo a cara Run apresentar resultados totalmente diferentes. Sendo ajustados nas configurações paramétricas. Sendo elas.

número de gerações : quantas vezes o algoritmo irá criar novos conjuntos de população.
tamanho da população : quantidade de indivíduos gerados. Quanto mais indivíduos, maior a chance de aparecer um de ótima qualidade para resolução do problema.
taxa de mutação : Quantidade de indivíduos presentes na população que irão sofrer alterações nos seus genes.
taxa de mortalidade : Quantidade de indivíduos que irão passar para próxima geração propagando seus genes.

Para rodar os testes da matriz 100x100 foram escolhidas as seguintes configurações paramétricas por apresentar um indivíduo com melhor fitness.

número de gerações : 600
tamanho da população : 1000
taxa de mutação : 0,6
taxa de sobrevivência : 0,1

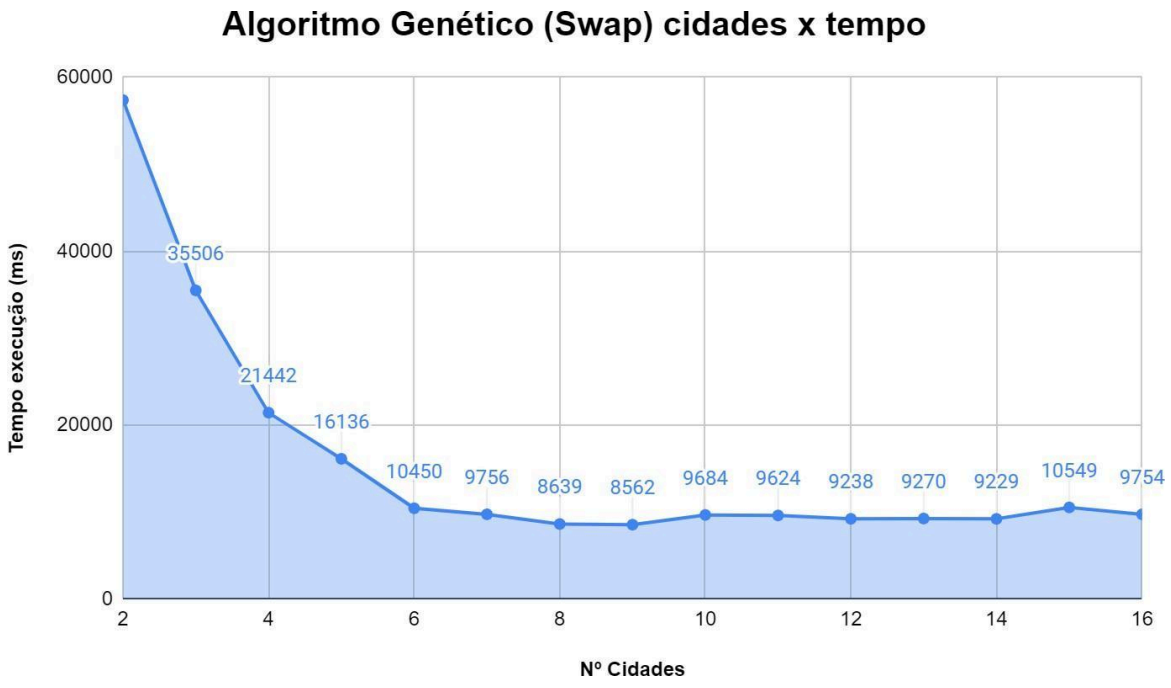
Essa combinação de configurações gerou o indivíduo com menor fitness nos testes realizados (3271).

Matriz 100x100	
Nº Run	Resultado
1	3410
2	3271
3	3593
4	3562
5	3518

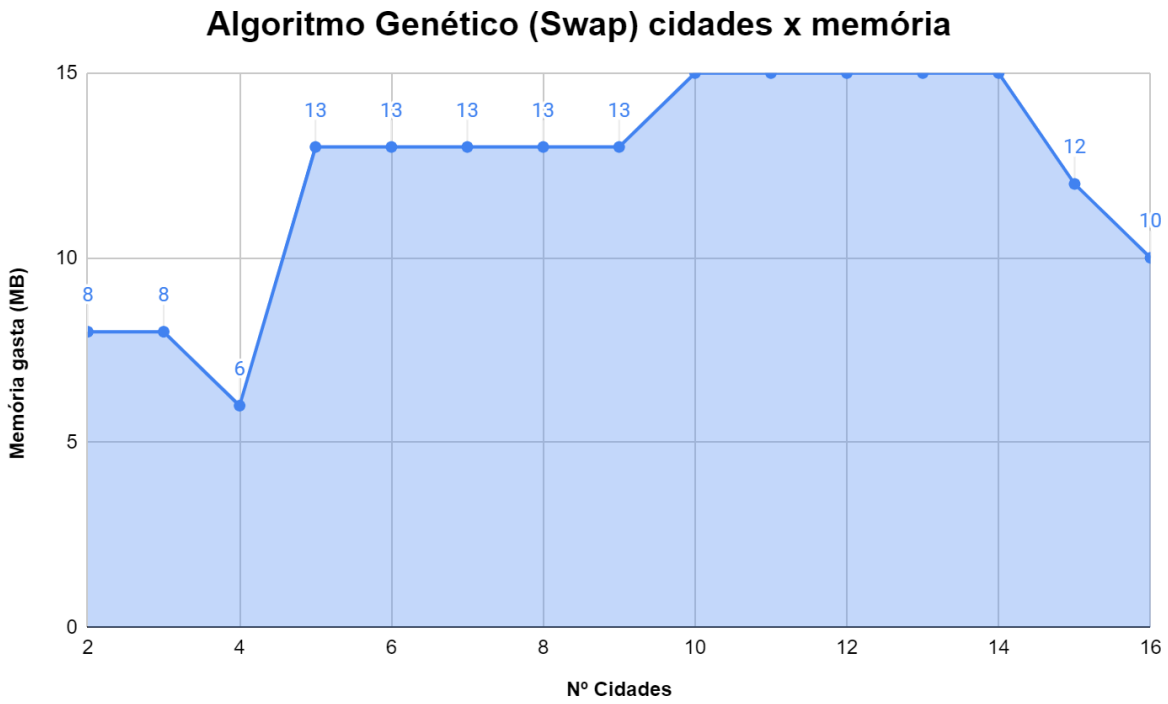
(tabela 5)

Demonstrando a variação de resultados obtidos com apenas 5 Runs em tempo médio de 12 minutos.(tabela 5).

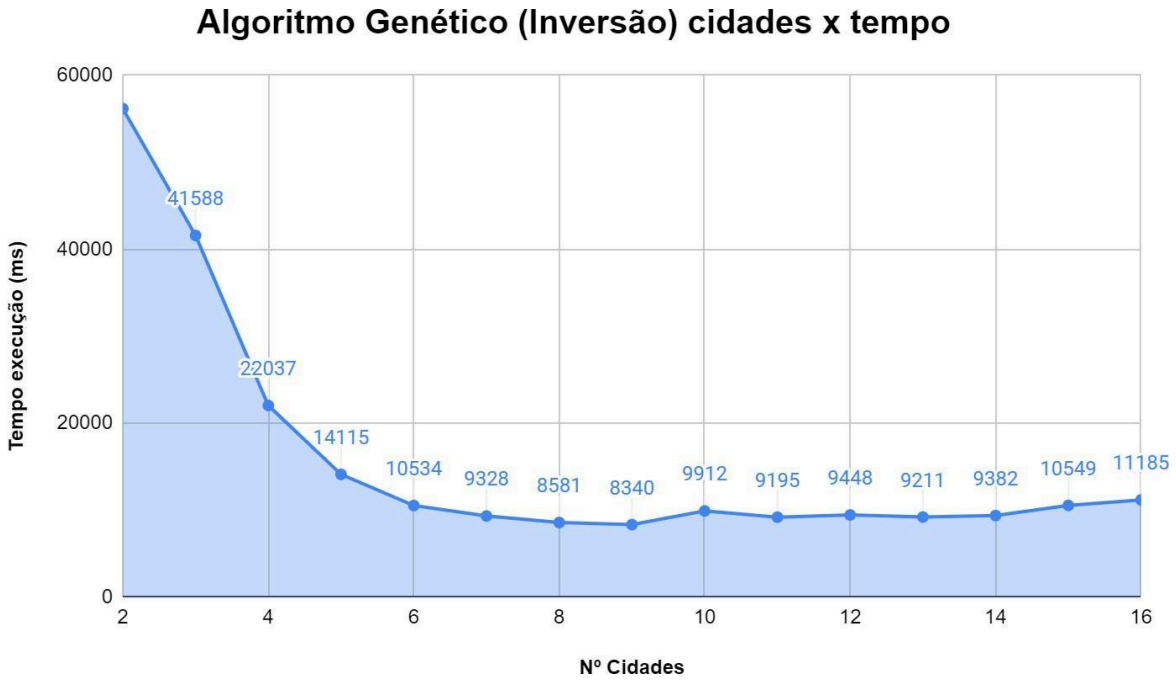
4. Algoritmo genético X Ótimo (tempo e consumo de memória)



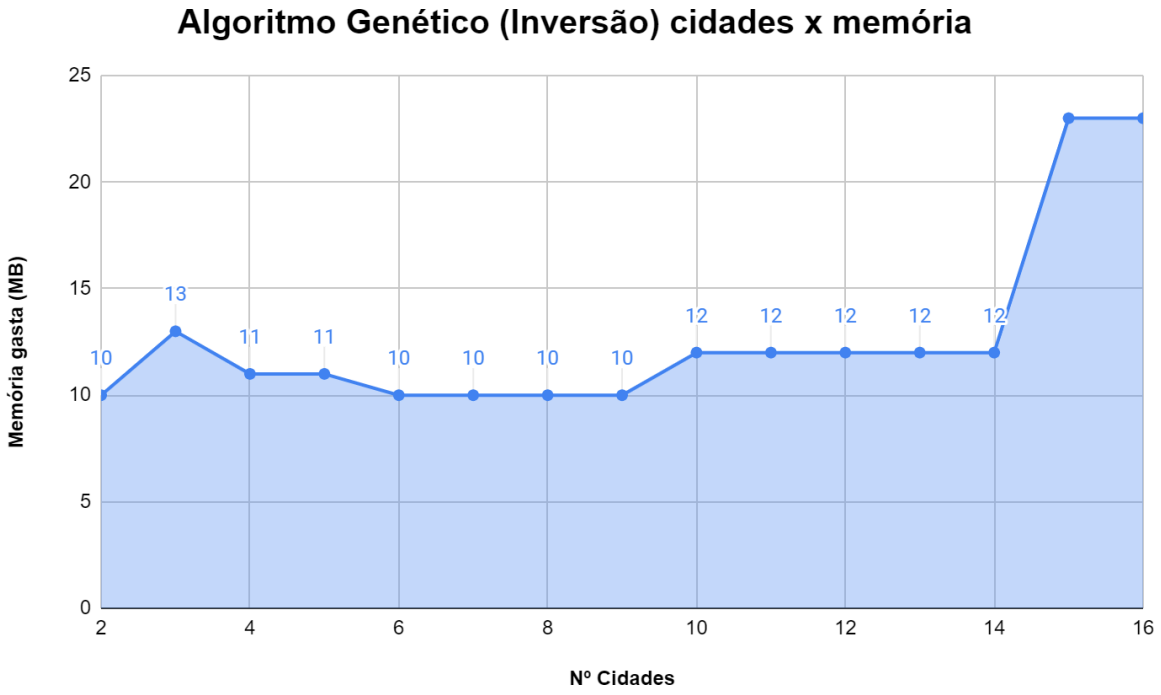
(figura 17)



(figura 18)

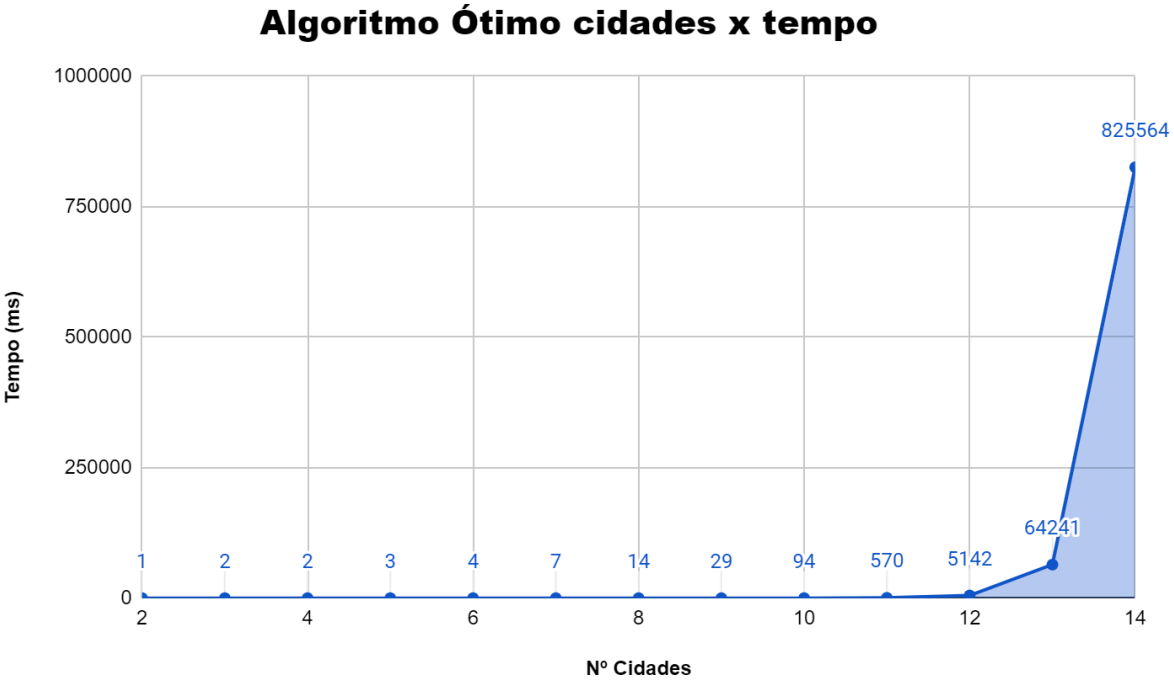


(figura 19)

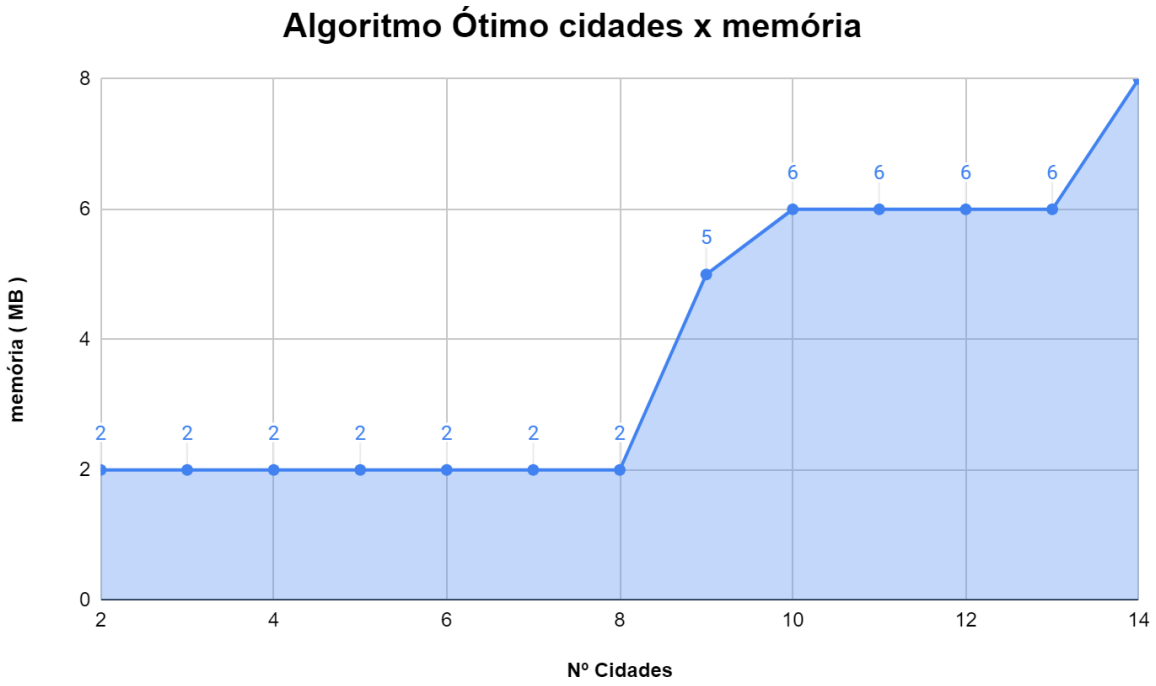


(figura 20)

Podemos observar que para poucas cidades o algoritmo gastou mais tempo. Conforme vamos aumentando a demanda ele vai se estabilizando. Em paralelo o consumo de memória vai aumentando conforme a carga de cálculos.(figura 17 ao 20).



(figura 21)



(figura 22)

Para o algoritmo ótimo percebe-se um desempenho melhor ao genético quando comparado com o tempo levado para realizar os cálculos com pequenas quantidades de cidades. Porém seu consumo de memória também vai aumentando conforme mais cálculos são exigidos.(figura 21 e 22).

Em testes realizados com o arquivo ótimo na matriz 100x100 disponibilizada no Google sala de aula o limite para as cidades do algoritmo ótimo foi de 14, já que ao escolher 15 cidades para realizar os cálculos o programa fica horas indeterminadas para retornar uma resposta.

Quando comparamos o máximo do algoritmo ótimo contra o genético cada um leva vantagem em um quesito porém a precisão do genético não é boa pois depende de aleatoriedade para retornar o resultado. Já o ótimo perde em velocidade porém ganha em precisão nos mostrando um caminho com o menor peso.

Run Algoritmo ótimo (figura 23):

```
Tempo gasto : 887297 ms
Memoria gasta : 8 MB

O custo total para o menor caminho foi de: 267
```

(figura 23)

Run Algoritmo genético (figura 24):

```
Configuracao :
Geracoes : 10
Populacao inicial : 10000
Taxa de Mutacao : 0.5
Taxa de sobrevivencia : 0.6
Memoria gasta : 18 MB
Tempo gasto : 8267 ms
melhor individuo :[1, 9, 6, 0, 5, 7, 10, 3, 2, 8, 13, 4, 11, 12, 405] posicao 0
Fitness = 405
```

(figura 24)

5. Algoritmo genético (Swap) (tempo e consumo de memória)

Resultado do algoritmo aplicado ao arquivo de 200 cidades disponibilizado no Google sala de aula.

==========
Configuracao :
==========
Mutacao utilizada : Swap
Geracoes : 700
Populacao inicial : 200
Taxa de Mutacao : 0.6
Taxa de sobrevivencia :
0.1 Memoria gasta : 178
MB Tempo gasto : 560495
ms
melhor individuo :[139, 26, 27, 192, 190, 104, 78, 81, 75, 189, 72, 117, 31, 131, 116, 112, 6, 63, 60, 73, 151, 48, 62, 15, 188, 163, 182, 135, 36, 168, 92, 68, 129, 159, 89, 23, 145, 167, 160, 169, 140, 87, 69, 38, 90, 24, 105, 120, 193, 103, 165, 12, 44, 177, 10, 113, 102, 198, 184, 185, 143, 79, 49, 141, 107, 53, 74, 101, 94, 5, 3, 33, 70, 173, 25, 58, 115, 124, 121, 28, 67, 57, 82, 180, 137, 98, 80, 191, 86, 179, 152, 118, 130, 41, 76, 170, 138, 56, 43, 11, 54, 111, 34, 52, 20, 4, 32, 109, 157, 153, 1, 2, 71, 29, 119, 146, 162, 95, 186, 42, 106, 22, 7, 96, 123, 176, 47, 66, 125, 61, 156, 144, 147, 100, 172, 17, 30, 51, 39, 18, 128, 50, 85, 148, 9, 16, 155, 46, 132, 174, 35, 99, 93, 14, 195, 110, 59, 164, 150, 91, 134, 127, 199, 65, 175, 133, 97, 64, 183, 142, 166, 161, 83, 55, 181, 88, 0, 171, 77, 84, 19, 136, 40, 197, 187, 108, 126, 149, 114, 154, 194, 8, 196, 158, 37, 45, 178, 21, 13, 122, 7812] posicao 0
Fitness = 7812