UFC – UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE SOBRAL

CURSO DE ENGENHARIA COMPUTAÇÃO

INTELIGÊNCIA COMPUTACIONAL

PROFESSOR: JARBAS JOACI

Avaliação Final

FRANCISCO GILSON PEREIRA ALMEIDA FILHO – 401066

Sobral - CE
2020.1

SUMÁRIO

1. O problema do caixeiro viajante.

O problema do caixeiro viajante é um problema combinatorial que tem por definição conceitualmente padrão o seguinte: um caixeiro viajante deve visitar todas as cidades da sua área uma única vez e retornar ao ponto de partida. Dado que existe um custo de transporte entre todas as cidades, deseja-se encontrar o percurso com o menor custo possível. O espaço de busca deste problema é um conjunto de permutações de n cidades e este espaço tem um tamanho de n! permutações possíveis.

No arquivo do trabalho disponibilizado pelo professor foi defino o seguinte:

1.ª Crie um algoritmo genético para o problema do caixeiro viajante representado por um grafo completo não direcionado de 14 vértices (cidades) cuja matriz de adjacência, que representa as distâncias entre as cidades, é:

Fonte: PDF do trabalho disponibilizado pelo professor.

Cidades	01	02	03	04	05	06	07	08	09	10	11	12	13	14
01	0	1	2	4	6	2	2	3	3	5	6	1	4	5
02		0	3	2	1	3	6	3	4	4	2	4	4	4
03			0	1	3	3	2	3	4	1	3	5	5	6
04				0	5	1	4	2	3	4	4	8	2	2
05					0	2	1	6	5	2	3	4	2	2
06						0	3	1	2	3	5	7	3	4
07							0	2	1	2	5	2	4	3
08								0	5	5	1	5	3	6
09									0	1	4	4	5	3
10										0	5	4	4	2
11											0	4	2	1
12												0	1	3
13													0	1
14														0

Imagem 1: Matriz que define a distância entre as cidades.

O algoritmo deve exibir o melhor caminho encontrado e o seu custo de percurso. Use crossover de mapeamento parcial e mutação por permutação de elementos.

Com base no que foi fornecido, foi desenvolvido um algoritmo genético em SCILAB 6.1.0, que encontra um possível melhor caminho e o custo do caminho, que pode melhorar ou diminuir sua precisão com base no tamanho da população e quantidade de gerações utilizadas.

2. Código Fonte.

2.1. Declaração da matriz de distâncias.

Iniciantemente foi declarado a matriz de distâncias, com uma pequena alteração para facilitar a sua leitura durante o código. A matriz foi espelhada na parte inferior (Imagem 2).

Fonte: Autor.

```
//A matriz foi espelha na parte inferior para facilitar a leitura
Distancia =    [ 0 1 2 4 6 2 2 3 3 5 6 1 4 5;
                 1 0 3 2 1 3 6 3 4 4 2 4 4 4;
                 2 3 0 1 3 3 2 3 4 1 3 5 5 6;
                 4 2 1 0 5 1 4 2 3 4 4 8 2 2;
                 6 1 3 5 0 2 1 6 5 2 3 4 2 2;
                 2 3 3 1 2 0 3 1 2 3 5 7 3 4;
                 2 6 2 4 1 3 0 2 1 2 5 2 4 3;
                 3 3 3 2 6 1 2 0 5 5 1 5 3 6;
                 3 4 4 3 5 2 1 5 0 1 4 4 5 3;
                 5 4 1 4 2 3 2 5 1 0 5 4 4 2;
                 6 2 3 4 3 5 5 1 4 5 0 4 2 1;
                 1 4 5 8 4 7 2 5 4 4 4 0 1 3;
                 4 4 5 2 2 3 4 3 5 4 2 1 0 1;
                 5 4 6 2 2 4 3 6 3 2 1 3 1 0;];
```

Imagem 2: Declaração da matriz com os valores de distâncias das cidades.

2.2. Implementação da função de avaliação.

De forma resumida, essa função é responsável por calcular a somatório das distâncias das cidades de um determinado percurso. Foi utilizado uma estrutura de repetição *for* para a realização da somatória da primeira até a última cidade, e ao final da função, adicionado o valor da última cidade para a primeira (Imagem 3).

Fonte: Autor.

```
//Função de avaliação, somatorio das distancias entre as cidades
function [Nota]=Aval(Ind)
    Nota = 0
    for j=1:13
        Nota = Nota + Distancia(Ind(j),Ind(j+1))
    end
    //Distancia entre a ultima cidade e a primeira
    Nota = Nota + Distancia(Ind(14),Ind(1))
endfunction
```

Imagem 3: Função de avaliação(Somatório das distâncias).

2.3. Implementação da função do crossover de mapeamento parcial.

Para facilitar a explicação dessa função irei dividi-la em duas partes, na parte 1 (Imagem 4), inicialmente os filhos são clonados a partir dos pais, logo após, é feito a seleção dos pontos de corte (P1 e P2) de maneira aleatória, é importante ressaltar que o P1 não pode ser maior do que P2, considerando isso optei por fazer um sorteio de (0-13) nos pesos, e adiante, fiz um tratamento usando um *if* para caso o P1 ser maior que P2 os valores serem invertidos, também é feito um incremento do P2 para caso seja sorteado o mesmo valor, por exemplo, se ambos forem sorteados com o valor 13, o P2 incrementa +1 e assim o P2 continuara maior do que P1.

Fonte: Autor.

```
//Crossover de mapeamento parcial
function [Filho1,Filho2]=Cross(Pai1,Pai2)

    //Inicialmente os filhos sao clones dos pais
    Filho1 = Pai1
    Filho2 = Pai2

    //Selecao pontos de corte

    P1 = ceil(rand()*13)//Ponto de corte [1,14]
    P2 = ceil(rand()*13)//Ponto de corte [1,14]

    //Garante que P1 < P2
    if P1 > P2 then
        aux = P2
        P2 = P1
        P1 = aux
    end
    P2 = P2 + 1
```

Imagem 4: Parte 1 da função de crossover de mapeamento parcial.

Na segunda parte da função, foi usado algumas estruturas de repetição que vão garantir que não haja repetição de cidades nos filhos utilizando o mapeamento de genes do ponto de corte (Imagem 5).

Fonte: Autor.

```
//Garante que não haja repetição de cidades nos filhos, atraves do mapeamento
//Filho1
for i = 1:14
    j=P1
    while(j<=P2)
        if Filho1(i) == Filho1(j) & ((i<P1)|(i>P2))  then
        Filho1(i) = Filho2(j)
        j=P1-1
        end
    j=j+1
    end
end
```

Imagem 5: Parte 2 da função de crossover de mapeamento parcial.

2.4. Implementação da função de mutação por permutação de elementos.

Inicialmente foi declarado a chance de mutação, que foi definido com 0.5%, logo após, são selecionados os índices dos elementos da permutação (P1 e P2), adiante, ele permuta os elementos, isso é, faz a troca de suas posições (Imagem 6).

Fonte: Autor.

```
//Mutacao por permutação de elementos
function [Ind_M]= Mutacao(ind)
    Ind_M = ind

    if rand()*100 <= 0.5 then //Da uma chance de 0.5%

    //Seleciona os elementos da permutacao
    P1 = ceil(rand()*14)
    P2 = ceil(rand()*14)

    //Permuta os elementos selecionados
    aux = Ind_M(P1)
    Ind_M(P1) = Ind_M(P2)
    Ind_M(P2) = aux
    end
endfunction
```

Imagem 6: Função de mutação por permutação de elementos.

2.5. Implementação da função de seleção por sorteios.

Inicialmente é declarado o número de participantes do torneio, que foi definido como 20% do tamanho da população, logo após, são feitos os torneios com base no tamanho da população. Os participantes são selecionados aleatoriamente, após isso, suas notas são calculadas e o melhor é escolhido como um dos pais, esse processo é repetido n vezes com base no tamanho da população.

Fonte: Autor.

```
//Selecao dos pais utilizando o torneio
function [Pais]= Selecao(Populacao)
    NP = 0.2*Tam // Numero de participantes do torneio
    Pais = Populacao;
    Participantes = Populacao(1:NP,1:14);
    Notas_T = Notas(1:NP)
    for i=1:Tam
        for j=1:NP//Seleciona Numero de participantes aleatoriamente
            Participantes(j,1:14) = Populacao(ceil(rand()*Tam),1:14)
        end

        for j=1:NP
            Notas_T(j) = Aval(Participantes(j,1:14))
        end

        [val pos] = min(Notas_T)
        Pais(i,1:14)=Participantes(pos,1:14);
    end
endfunction
```

Imagem 7: Função de seleção por sorteio.

2.6. Código principal.

Inicialmente declarando como vazio as matrizes da população, pais e filhos. Após isso é declarado o tamanho da população, que foi definido no código como 100 (valores maiores exige mais poder computacional). Adiante, zera o vetor de notas e começa o sorteio sem repetição das cidades para definir a população inicial.

Fonte: Autor.

```
//Geração da primeira populacao
//Vetores que serão utilizados para armazenar a populacao, Pais e Filhc
Populacao=[]
Pais=[]
Filhos=[]

Tam = 100;//Tamanho da população
Notas=zeros(1,Tam)

//Sorteio sem repetição das cidades
for i=1:Tam//Gerando com Tam individuos
    C = []; //Vetor auxiliar vazio
    sel = 14; //Indice com o valor da quantidade de cidades
    cidades = [1 2 3 4 5 6 7 8 9 10 11 12 13 14]
        while(sel>0)//Seleciona uma das 14 cidades e em seguida a elimi
        R = ceil(rand()*sel);//Seleciona um numero entre 1 e sel(inicia
        C = [C cidades(R)]//Adiciona a cidade no vetor auxiliar C
        cidades(R)=[]//Remove a cidade selecionada do vetor cidade
        sel=sel-1;//Decrementa o valor de sel
        end
    Populacao = [Populacao; C]//Adiciona o vetor C em Populacao
end
```

Imagem 8: Parte 1 do código principal.

Aqui, foi definido o número de gerações, também definido como 100 (valores maiores exige mais poder computacional). E chegamos no for principal do algoritmo, primeiramente os indivíduos são avaliados, os pais são selecionados utilizando a função de seleção por torneios (tópico 2.5), após isso é realizado o cruzamento para gerar os filhos através do crossover de mapeamento parcial (tópico 2.3), cada filho tem chance de sofrer mutação (tópico 2.4), e por fim, a população é substituída pelos filhos e o processo se repete a cada geração.

Fonte: Autor.

```
NG=100//Numero de gerações

for n=1:NG
    //Avalicacao dos individuos
    for i=1:Tam
        Notas(i) = Aval(Populacao(i,1:14));
    end

    //Selecao dos pais
    Pais = Selecao(Populacao)

    //Cruzamento
    for i=1:2:Tam
        [Filhos(i,1:14) Filhos(i+1,1:14)] = Cross(Pais(i,1:14),Pais(i+1,1:14)) ;
    end

    //Mutacao
    for i=1:Tam
        Filhos(i,1:14) = Mutacao(Filhos(i,1:14));
    end

    Populacao = Filhos
end
```

Imagem 9: Parte 2 do código principal.

Ao final da última geração o indivíduo(percurso) com menor/melhor nota(custo) é selecionado e ambos são exibidos.

Fonte: Autor.

```
[val pos]=min(Notas)

disp("Melhor caminho:")

disp(Populacao(pos,1:14))

disp("Custo do melhor caminho:")

disp(val)
```

Imagem 10: Parte 3 do código principal.

3. Resultados Encontrados.

-Alguns resultados com testes realizados utilizando tamanho de população e número de gerações diferente em cada teste, vale ressaltar que quanto maior esses valores, mais tempo o algoritmo demorara para executar (Imagens 11,12,13,14 e 15).

Fonte: Autor.

```
"Melhor caminho:"

 3.   4.   14.   13.   11.   8.   6.   10.   9.   2.   5.   7.   12.   1.

"Custo do melhor caminho:"

 23.
```

Imagem 11: Teste com tamanho da população = 50 e número de gerações = 50.

Fonte: Autor.

```
 "Melhor caminho:"

  2.   5.   13.   11.   14.   10.   9.   7.   12.   1.   8.   6.   4.   3.

 "Custo do melhor caminho:"

  22.
```

Imagem 12: Teste com tamanho da população = 100 e número de gerações = 100.

Fonte: Autor.

```
"Melhor caminho:"

 11.   2.   1.   12.   13.   14.   5.   7.   3.   10.   9.   6.   4.   8.

"Custo do melhor caminho:"

 19.
```

Imagem 13: Teste com tamanho da população = 200 e número de gerações = 200.
Fonte: Autor.

```
"Melhor caminho:"

 11.   2.   1.   12.   13.   8.   6.   4.   3.   10.   9.   7.   5.   14.

"Custo do melhor caminho:"

 18.
```

Imagem 14: Teste com tamanho da população = 300 e número de gerações = 300.

Fonte: Autor.

```
 10.   9.   7.   12.   13.   14.   11.   8.   1.   2.   5.   6.   4.   3.

"Custo do melhor caminho:"

 18.
```

Imagem 15: Teste com tamanho da população = 400 e número de gerações = 400.

Devido ao caráter aleatório de alguns parâmetros do algoritmo, nem sempre o valor ótimo será atingido. Mas sempre ele encontrará um bom resultado.

Obs: Foi definido o tamanho da população = 100 e número de gerações = 100, no meu computador com esses valores, o algoritmo demora alguns segundos para executar.

Referências

Ajuda do Scilab. Disponível em: https://help.scilab.org/docs/5.5.0/pt_BR/index.html. Acesso em: 27.10.2020.

Algoritmos Genéticos. Disponível em: https://www.maxwell.vrac.puc-rio.br/3725/3725_4.PDF. Acesso em: 27.10.2020.