



CENTRO DE INFORMÁTICA - UFPB

Etapa 2 - Parte 1

Relatório de Representação da Solução

Yuri da Costa Gouveia

11328072

João Pessoa, Abril de 2019

1. Descrição do Problema

Este relatório visa explicar a heurística construída para resolução do problema do caixeiro viajante. Na seção 2, será abordado o método utilizado para a construção inicial da solução. Já na seção 3, será descrita a lógica utilizada para alcançar uma solução de otimização utilizando movimentos de vizinhança.

2. Construção

Para a construção inicial do problema foi utilizado o método do vizinho mais próximo. Dado um ponto partida (inicialmente a posição 0 da matriz, no método desenvolvido) o de chegada é escolhido com base na distância entre o primeiro e os seus vizinhos, de modo que o ponto de chegada escolhido será o seu vizinho mais próximo. Esse procedimento termina quando todos os vértices são visitados e no fim, retorna-se para o ponto inicial.

```
procedimento ConstracaoGulosa( $g(\cdot), s$ );  
1   $s \leftarrow \emptyset$ ;  
2  Inicialize o conjunto  $C$  de elementos candidatos;  
3  enquanto ( $C \neq \emptyset$ ) faça  
4     $g(t_{melhor}) = \text{melhor}\{g(t) \mid t \in C\}$ ;  
5     $s \leftarrow s \cup \{t_{melhor}\}$ ;  
6    Atualize o conjunto  $C$  de elementos candidatos;  
7  fim-enquanto;  
8  Retorne  $s$ ;  
fim ConstracaoGulosa;
```

Figura 1: Pseudo-código da solução gulosa do vizinho mais próximo.

```
def vizinho_mais_proximo(pontoPartida, numNos, matriz, visitados, caminho, distancia):  
    minimo = sys.maxsize  
  
    for noChegada in range(0, numNos):  
        # Loop de escolha do vizinho mais próximo  
        if matriz[pontoPartida][noChegada] > 0 and matriz[pontoPartida][noChegada] < minimo and visitados[noChegada]==False:  
            minimo = matriz[pontoPartida][noChegada]  
  
    distancia += minimo  
    visitados[matriz[pontoPartida].index(minimo)] = True # Marca ponto como visitado  
    caminho.append(matriz[pontoPartida].index(minimo)) # Adiciona ponto aos caminhos  
  
    if len(caminho) < numNos:  
        # Recursividade para achar o caminho do vizinho mais próximo  
        return vizinho_mais_proximo(matriz[pontoPartida].index(minimo), numNos, matriz, visitados, caminho, distancia)  
    else:  
        return visitados, caminho, distancia
```

Figura 2: Implementação do vizinho mais próximo.

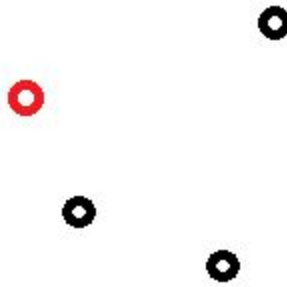


Figura 3: Representação de pontos a se percorrer pelo PCV.

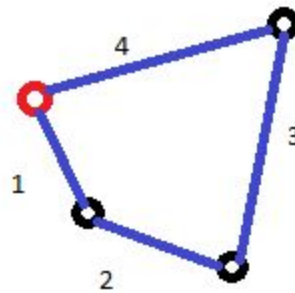


Figura 4: Representação da solução do vizinho mais próximo..

3. Movimentos de Vizinhança

Para os movimentos de vizinhança a seguinte lógica foi aplicada:

Dada uma solução inicial do problema do caixeiro viajante, faz-se uma troca entre duas arestas. Caso essa troca resulte em uma solução melhor que a solução inicial, toma-se essa nova solução como uma otimização, não sendo obrigatoriamente a solução ótima para o problema.

Essa abordagem de pegar a primeira otimização como solução foi feita devido ao fato do tempo para achar a melhor solução do caixeiro viajante. Para isso, seria necessário muito tempo e elevado gasto de processamento devido às inúmeras iterações que seriam realizadas.

Ainda assim, para um grande número de rotas, é provável que a execução do programa desenvolvido demore muito tempo. Para contornar esse problema um

critério de parada foi definido: se o laço do *for* é executado mais que cinco mil vezes, retorna-se a solução inicial de construção.

```

procedimento VND( $f(\cdot), N(\cdot), r, s$ )
1  Seja  $r$  o número de estruturas diferentes de vizinhança;
2   $k \leftarrow 1$ ; {Tipo de estrutura de vizinhança corrente}
3  enquanto ( $k \leq r$ ) faça
4      Encontre o melhor vizinho  $s' \in N^{(k)}(s)$ ;
5      se ( $f(s') < f(s)$ )
6          então
7               $s \leftarrow s'$ ;
8               $k \leftarrow 1$ ;
9          senão
10              $k \leftarrow k + 1$ ;
11      fim-se;
12 fim-enquanto;
13 Retorne  $s$ ;
fim VND;

```

Figura 5: Pseudo-código dos movimentos de vizinhança.

```

def vizinhanca(numNos, matrizAux, caminho, visitados, distancia):

    melhorCaminho = caminho
    melhorado = 0

    for i in range(1, len(caminho)-2):
        for j in range(i+1, len(caminho)):
            if j-i == 1: # Não muda nada
                continue
            novoCaminho = caminho[:]
            novoCaminho[i:j] = caminho[j-1:i-1:-1] # 2-opt swap
            novaDistancia = custo(np.array(matriz), np.array(novoCaminho))
            if novaDistancia < distancia: # Verifica se o novo caminho é melhor que o antigo
                melhorCaminho = novoCaminho
                melhorado = 0
                return caminho, distancia, melhorCaminho, novaDistancia #Retorna primeira melhora
            else:
                melhorado += 1
    if melhorado > 5000:
        return caminho, distancia, melhorCaminho, novaDistancia #Retorna solução sem melhora

```

Figura 6: Implementação dos movimentos de vizinhança.

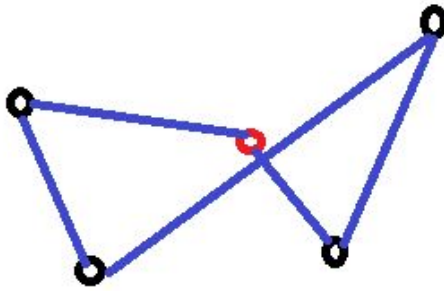


Figura 7: Representação de um caminho inicial.

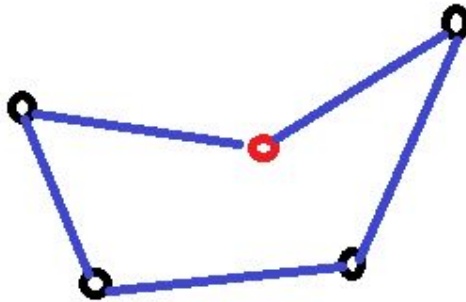


Figura 8: Representação de movimentos de vizinhança.