Pontifícia Universidade Católica do Rio de Janeiro

Algoritmos de Busca Gulosa e Simulated Annealing para resolução do Problema de Mínima Latência

Bruna Paracat - 1611896
Conrado Costa - 1710234

Sumário

Introdução

O Problema de Mínima Latência (PML) é uma das mais populares variantes do Problema do Caixeiro Viajante (Travelling Salesman Problem). A solução de ambos problemas é foco de muitas pesquisas da área de inteligência artificial, pois, por sua alta complexidade, para resolvê-los não se pode utilizar de simples algoritmos.

O PML consiste em, dado um vértice de origem pertencente a um grafo, construir um circuito hamiltoniano que minimize os tempos de espera para cada vértice, isto é, o tempo percorrido desde a saída da origem até a chegada em cada vértice. Portanto, o problema se encaixa na série de otimização através do mínimo entre as soluções.

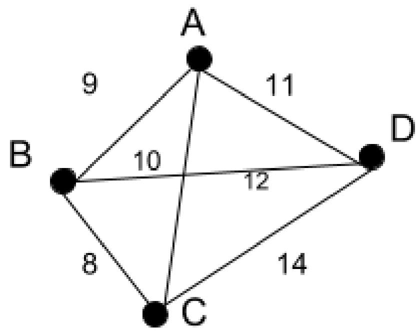
Neste projeto, aplicou-se a heurística de busca gulosa para gerar uma solução inicial, tendo em vista a simplicidade de implementação do algoritmo e a necessidade de gerar rapidamente um ponto de partida razoável.

A fim de refinar a solução obtida, aplicou-se um algoritmo que faz uso da meta-heurística *simulated annealing* que, em suma, gera diversas soluções e as compara sempre buscando melhora mas, aceitando de acordo com uma probabilidade baixa soluções piores. Ao finalizar resgata a melhor solução encontrada até então.

1 Definição do Problema

Sabendo que a solução possui o formato de uma rota a ser seguida, antes de implementar qualquer aplicação é necessário estudar o espaço de possibilidades que o problema de mínima latência está contido, isto é, quais ações podem ser efetuadas para se encontrar uma solução.

Estabeleceremos que inicialmente conhecemos um grafo, conexo, não orientado, completo de n vértices. Qualquer grafo assim construído, pode ser representado como uma matriz onde cada campo possui o valor da aresta entre dois vértices.



	A	B	C	D
A	0	9	10	11
B	9	0	8	12
C	10	8	0	14
D	11	12	14	0

1.1 Exemplo de grafo e sua matriz equivalente

Para contextualização do problema, considera-se cada vértice uma cidade e suas as arestas, os tempos de viagem uma origem a um destino.

1.1 Componentes do problema

- Estado Inicial: Em uma cidade qualquer *x.*
- Estado Final: Em uma cidade qualquer *y ,* tendo percorrido todos as outras cidades do grafo.
- Ações Possíveis: Ir para uma cidade não visitada.
- Espaço de Estados: O número de estados finais possíveis, varia de acordo com a quantidade n de cidades do grafo estudado em *n!* .
- Custo: Tempo de espera de cada cidade ou tempo de viagem até uma cidade *y* partindo de uma cidade *x* .

2 Metodologia

O projeto foi desenvolvido na linguagem de programação C++ , esta foi escolhida devido a vasta coleção de classes já implementadas que agilizam a programação.

Para a construção de soluções finais, foram implementados dois métodos de busca que seguem estruturas algorítmicas tradicionais.

A primeira, chamada “busca gulosa” é classificada como uma função heurística, pois utilizando informações conhecidas sobre o problema, constrói uma solução a partir de iterações partindo de um ponto inicial. Possui uma lógica simples e não obtém resultados eficazes para problemas envolvendo um número grande de vértices.

Portanto, foi implementado o algoritmo de “simulated annealing”. Este, faz o uso de uma estratégia considerada meta-heurística que, em suma, é a prática de utilizar combinações aleatórias e conhecimento de soluções já obtidas para assim guiar a busca e construir novas soluções evitando cair em problemas de mínimos locais.

As instâncias utilizadas para teste estão disponíveis online e consistem em matrizes que expressam as distâncias entre dois vértices (como demonstrado imagem 1.1).

Para melhor organização, o projeto foi encapsulado em 2 módulos:

- *Leitura.cpp* - referente a leitura dos arquivos de instâncias e carregamento de informações na RAM (preenchimento da matriz de cidades e tempos de viagem).
- *main.cpp* - sequência de execução da aplicação e funções referentes aos algoritmos implementados.

2.1 Leitura

Antes de abordarmos a questão dos algoritmos implementados, é importante observar como foram organizados os dados de cada instância testada. Seja **n** o número de cidades descritas num arquivo texto (vide imagem abaixo), cada uma terá **n** pares **$<j, d_{ij}>$**, sendo **j** a cidade da qual **i** dista **d_{ij}**. Nesta variante do problema, **$d_{ij} = d_{ji}$** $\forall$ i,j $\in$ [0,n). Tem-se, também, que **d_{ij}** $\forall$ i=j.

Foi criado um vetor de **n** posições, e cada uma dessas posições, em si, é um vetor de pares de números inteiros. Cada posição dispõe de n pares, totalizando assim uma matriz com nxn elementos <cidade, distância>. Deste modo, os dados ficam melhor organizados e mais fáceis de serem acessados.

```
NAME: 11cidades
TYPE: TSP
COMMENT: 11 cities
DIMENSION: 11
EDGE_WEIGHT_TYPE: EXPLICIT
EDGE_WEIGHT_FORMAT: LOWER_DIAG_ROW
DISPLAY_DATA_TYPE: TWOD_DISPLAY
EDGE_WEIGHT_SECTION
   0   8   0  39  45   0  37  47   9   0  50  49  21  15   0  61  62  21
  20  17   0  58  60  16  17  18   6   0  59  60  15  20  26  17  10   0
  62  66  20  25  31  22  15   5   0  81  81  40  44  50  41  35  24  20
   0 103 107  62  67  72  63  57  46  41  23   0
EOF
```

2.1 Exemplo de instância com matriz de 11 cidades

```
11
<0,0>  <1,8>  <2,39>  <3,37>  <4,50>  <5,61>  <6,58>  <7,59>  <8,62>  <9,81>  <10,103>
<0,8>  <1,0>  <2,45>  <3,47>  <4,49>  <5,62>  <6,60>  <7,60>  <8,66>  <9,81>  <10,107>
<0,39>  <1,45>  <2,0>  <3,9>  <4,21>  <5,21>  <6,16>  <7,15>  <8,20>  <9,40>  <10,62>
<0,37>  <1,47>  <2,9>  <3,0>  <4,15>  <5,20>  <6,17>  <7,20>  <8,25>  <9,44>  <10,67>
<0,50>  <1,49>  <2,21>  <3,15>  <4,0>  <5,17>  <6,18>  <7,26>  <8,31>  <9,50>  <10,72>
<0,61>  <1,62>  <2,21>  <3,20>  <4,17>  <5,0>  <6,6>  <7,17>  <8,22>  <9,41>  <10,63>
<0,58>  <1,60>  <2,16>  <3,17>  <4,18>  <5,6>  <6,0>  <7,10>  <8,15>  <9,35>  <10,57>
<0,59>  <1,60>  <2,15>  <3,20>  <4,26>  <5,17>  <6,10>  <7,0>  <8,5>  <9,24>  <10,46>
<0,62>  <1,66>  <2,20>  <3,25>  <4,31>  <5,22>  <6,15>  <7,5>  <8,0>  <9,20>  <10,41>
<0,81>  <1,81>  <2,40>  <3,44>  <4,50>  <5,41>  <6,35>  <7,24>  <8,20>  <9,0>  <10,23>
<0,103>  <1,107>  <2,62>  <3,67>  <4,72>  <5,63>  <6,57>  <7,46>  <8,41>  <9,23>  <10,0>
 Solucao inicial:
0 --> 1 --> 2 --> 3 --> 4 --> 5 --> 6 --> 7 --> 8 --> 9 --> 10 -->
```

2.2 Como a matriz de 11 cidades fica carregada na memória e logo abaixo exemplo de uma solução inicial construída

2.2 Avaliação da Solução

Neste trabalho, uma solução é dita como um caminho. Em questões de implementação, esse caminho foi definido como um vetor contendo as trajetória que se deve seguir. Para se avaliar o custo de uma solução (a latência total), deve-se calcular os tempos de viagem entre as cidades subsequentes mantendo-se salvo o tempo de viagem total até então.

A função de avaliação recebe um vetor com o caminho a ser seguido e utiliza a matriz carregada na memória com todos os tempos de viagem entre cidades para avaliar o custo. Cada vez que se passa por um vértice verifica-se na matriz principal a distância deste para o próximo e soma-se a uma variável “caminho”. Para calcular a latência total, basta somar o caminho percorrido a uma variável “latenciaTotal”.

Pseudocódigo:

i ← 0;
enquanto i < quantidade de cidades **faça**
    se esta cidade é a última **então**
        parar execução;
    fim se

caminho ← caminho + MatrizDistancias[linha i][coluna i+1].distância;
latenciaTotal ← latenciaTotal + caminho;
fim enquanto

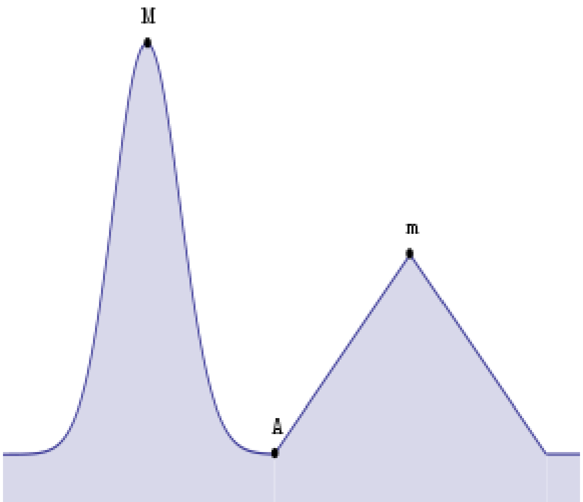
2.3 Busca Gulosa

2.3.1 Conceito e aplicações

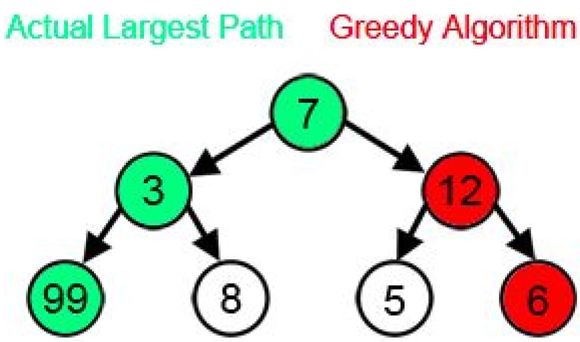
Um algoritmo guloso, *greedy algorithm* em inglês, caracteriza-se por sempre fazer uma escolha ótima de maneira local, a fim de tentar chegar à melhor solução. Em alguns problemas, essa abordagem funciona perfeitamente, sendo um algoritmo fácil e, por vezes, muito poderoso. Porém, para a maioria das instâncias do problema abordado neste relatório, o algoritmo não funciona de forma ótima globalmente, uma vez que o PML trata-se de um NP-Difícil.

Apesar de não resolver o problema, o algoritmo guloso, ainda assim, pode ser utilizado na solução. Geralmente, um algoritmo mais robusto necessita de um ponto de partida, uma solução inicial e muitas vezes, a saída é construir um caminho que seja um ótimo local, ou seja, o melhor resultado entre as soluções vizinhas.

Dessa maneira, o algoritmo guloso entra em cena e fornece, em tempo polinomial, uma solução satisfatória para um começo. A partir daí, outros algoritmos podem ser implementados para promover caminhos entre soluções vizinhas, a fim de se evitar mínimos locais e tentar explorar outras vizinhanças.



2.3 O Algoritmo guloso achará o melhor valor local (m), porém o máximo global (M) não será encontrado.



2.4 O exemplo simples acima mostra que nem sempre o algoritmo guloso fornecerá a melhor solução.

2.3.2 O algoritmo

Com as distâncias já armazenadas (**vector < vector < pair <int, int> > >**), o próximo passo é ordenar, da menor para a maior distância, as **n** duplas de cada uma das **n** cidades.

Uma vez a ordenação pronta, a busca gulosa começa. Coloca-se a cidade de índice 0 como origem do caminho (depósito). Para evitar que uma mesma cidade seja contabilizada mais de uma vez, é necessário marcar, num vetor booleano, aquelas que já foram visitadas.

A partir de uma cidade, o vetor de distâncias é percorrido e, já que está ordenado, é capaz de informar a cidade mais próxima não visitada. Todo o processo é feito até não restar nenhuma cidade para ser visitada. O algoritmo termina com a contabilização da volta da última cidade para a origem.

Pseudocódigo:

```
globais: n (numero de cidades), solucao_inicial (vetor vazio para a solução inicial;
Procedimento busca_gulosa(matriz_distancias)
cidades_visitadas[n];
cidade_origem←0;
push_back cidade_origem para solucao_inicial;
end ← 0;
enquanto (não end) faça
        para i ← 1, …, n faça
                se (não cidades_visitadas[matriz_distancias[cidade_origem][i]) então
                        push_back i para solucao_inicial;
                        cidades_visitadas[i] ←1;
                        cidade_origem = matriz_distancias[cidade_origem][i];
                        quebra do ciclo;
                fim se
        fim para
        end ← 1;
        para i ← 0, ... , n faça
                se (não cidades_visitadas[i])
                        end ← 0;
                        quebra do ciclo;
                fim se
        fim para
fim enquanto
pushback 0 para soluçao_inicial;
retorna solução inicial;
fim busca_gulosa
```

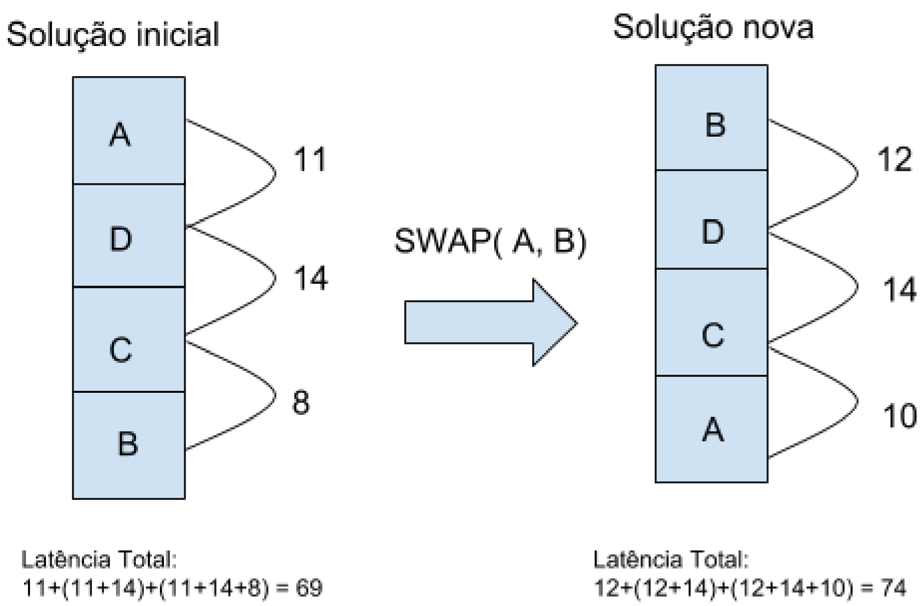
2.4 Simulated Annealing

Antes de entender o funcionamento do algoritmo é preciso saber alguns conceitos relativos à estruturas meta-heurísticas.

2.4.1 Vizinhança de SWAP

Vizinhança é uma operação aplicada a uma solução conhecida que como resultado gera uma nova solução. Existem diversos tipos de vizinhanças aplicáveis mas para este projeto somente utilizou-se a SWAP.

A SWAP consiste em selecionar dois elementos de uma solução e trocá-los de lugar. Neste caso, quando construímos um vetor que contém a ordem de visitação das cidades (a rota), ao trocar duas cidades de lugar, a latência total calculada da nova solução será diferente da original pois termos que recalcular as distâncias das cidades anteriores e sucessoras às cidades trocadas. O exemplo a seguir ilustra a situação utilizando os valores da imagem 1.1.



2.5 Exemplo de SWAP aplicado à uma solução do PML

Portanto, viu-se o SWAP como uma maneira simples e eficaz de obter-se novas soluções.

2.4.2 O algoritmo

Sendo assim, prossegue-se para a implementação do simulated annealing.
Pseudocódigo:

```
Procedimento simulated_annealing()
temperatura ← 10000
numero_de_iteracoes ← 30
fator_de_esfriamento ← 1
contador_de_repeticoes ←0

repita
        solução_corrente ← melhor_solução;
        contador_de_repeticoes ← 0;
        repita
                solução_nova ← nova solução utilizando vizinhança SWAP();
                delta ← avaliação(solução_nova) - avaliação(solução_corrente);
                se delta < 0 então
                        solução_corrente ← solução_nova;
                se não, então
                        solução_corrente ← solução_nova (de acordo com a probabilidade);
                fim se
                se avaliação(solução_corrente) < avaliação (melhor_solução) então
                        melhor_solução ← solução_corrente
                fim se
                contador_de_repetições  ← contador_de_repetições + 1
        até contador_de_repeticoes < numero_de_iteracoes

        temperatura ← temperatura - fator_de_esfriamento
até temperatura < 1
retorna melhor_solucao
```

A probabilidade mencionada trata-se do cálculo do valor $e^{-delta/temperatura}$. Sabe-se que para entrar neste cálculo delta será obrigatoriamente maior que 0 (devido a operação condicional *“**se** delta < 0”*), portanto ao definir “-delta” calcularemos o equivalente a $1/e^{delta/temperatura}$, ou seja, provavelmente um número muito pequeno.

No programa implementado o algoritmo aceitará a piora se $1/e^{delta/temperatura}$ for maior que um número aleatório muito pequeno gerado. Portanto, a probabilidade de aceitação diminui à medida que a temperatura diminui ou o delta aumenta, mas não se torna nula devido ao fator de aleatoriedade do número gerado.

Antes de definir-se tal condição, foram testadas várias maneiras diferentes de aceitação do movimento de piora, optou-se por este caminho pois assim que se encontrou os melhores resultados.

Também devido ao fator de aleatoriedade, o algoritmo não garante que a melhor solução final encontrada será sempre a mesma para toda execução, sendo assim, deve-se efetuar uma série de testes para cada instância avaliada.

3 Resultados Obtidos

Como dito, o problema de mínima latência é conhecido ao redor do mundo entre aqueles que estudam algoritmos de inteligência artificial, portanto , é possível avaliar a qualidade das soluções obtidas comparando-as com as melhores soluções já obtidas por especialistas (também conhecidas como “BKS”, *best known solution*). Visto que o algoritmo de simulated annealing lida com números aleatórios, a saída de cada teste pode ser diferente a cada execução. Para isto foram feitos vários testes em cada instância. Nota-se que para as instâncias com números menores de vértices os resultados são constantes pois o número de iterações elevadas no algoritmo garante a chegada no “ótimo” possível para o programa.

Instância: dantzig42.tsp

Número de vértices: 42
BKS: 12528
Tempo de Execução BKS: 0.17 (s)

Melhor solução obtida: 12761
Tempo de Execução: 2.1 (s)
Percentual de erro para BKS: 1,86%

Outros resultados obtidos: 12873
Tempo de Execução Médio: 1.96 (s)

Instância: gr48.tsp

Número de vértices: 48
BKS: 102378
Tempo de Execução BKS: 0.31 (s)

Melhor solução obtida: 110109
Tempo de Execução: 1.76 (s)
Percentual de erro para BKS: 7,55%

Outros resultados obtidos: 113992
Tempo de Execução Médio: 2.0 (s)

Instância: brazil58.tsp

Número de vértices: 58
BKS: 512361
Tempo de Execução BKS: 0.55 (s)

Melhor solução obtida: 591168
Tempo de Execução: 2.03 (s)
Percentual de erro para BKS: 15,38%

Outros resultados obtidos: 600705, 592365
Tempo de Execução Médio: 2.07 (s)

Instância: gr120.tsp

Número de vértices: 120
BKS: 363454
Tempo de Execução BKS: 9.54 (s)

Melhor solução obtida: 379015
Tempo de Execução: 4.61 (s)
Percentual de erro para BKS: 4,28%

Outros resultados obtidos: 390569, 380631 , 379715, 380668 , 379119
Tempo de Execução Médio: 4.66 (s)

Instância: pa561.tsp

Número de vértices: 561
BKS: 658870
Tempo de Execução BKS: 1155.32 (s)

Melhor solução obtida: 767429
Tempo de Execução: 18.68 (s)
Percentual de erro para BKS: 16,48%

Outros resultados obtidos: 786699, 767852 , 770081
Tempo de Execução Médio: 18.5 (s)

4 Conclusão

Como foi visto, em nenhum momento conseguiu-se obter a melhor solução possível. No entanto, considerando o tempo delegado ao projeto e a experiência dos programadores, os resultados foram satisfatórios. Durante as etapas de programação, vários obstáculos foram encontrados, a maioria referente ao algoritmo de simulated annealing.

Uma das grandes questões abordadas pelo grupo foi como definir o percentual de aceitação dos movimentos de piora. Foi notado que, ao escolher um percentual muito alto, o algoritmo falhava, pois aceitava excessivamente movimentos de piora e o mesmo acontecia com um percentual muito baixo. Vários valores foram testados, além de tentativas de impor novas condições na avaliação dos "deltas", mas no final escolheu-se seguir o modelo tradicional do algoritmo.

Outro problema ocorreu na definição do "fator de resfriamento". Sabe-se que, para melhores resultados, a temperatura deve decair lentamente. Mas, se colocado um fator muito baixo, este decaimento pode se tornar demasiadamente lento, a ponto de comprometer o tempo de execução do programa. O mesmo pode ser dito para a escolha de uma temperatura inicial: se for muito baixa, o programa rodará menos vezes e, ao mesmo tempo, não chegar a obter um resultado melhor; mas se muito alta, o programa pode-se tornar muito lento. Levando em consideração que as instâncias a serem testadas possuíam tamanhos muito variados, para estes parâmetros foram escolhidos valores que tornassem a saída de todos os testes um resultado razoável, mesmo que para as instâncias menores o número de iterações necessário para se atingir uma solução melhor seja menor.

Finalmente, é importante observar que o tempo de execução do programa varia em cada computador de acordo com a velocidade de processamento da CPU, Sendo assim, para testes efetuados em computadores diferentes os tempos de execução obtidos provavelmente serão diferentes.