UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

ESTRUTURAS DE DADOS I

ABORDAGEM DE FORÇA BRUTA PARA O PROBLEMA DO CAIXEIRO VIAJANTE

MATHEUS DE CASTRO SINISCARCHIO -

MARCUS VINÍCIUS MEDEIROS PARÁ - 11031663

FRANCISCO ALGUMA COISA -

SÃO CARLOS/SP

2018/01

SUMÁRIO

1. Introdução

Problemas de otimização são modelos matemáticos que consistem encontrar um soluções que obedeçam um conjunto de restrições e que tornem um determinada função, nesse conjunto solução, máxima ou mínima. O domínio da função objetivo é $(x_1x_2...x_n)^T \in R^n$, onde n é a quantidade de variáveis envolvidas; e as soluções que obedecem às restrições são chamadas de soluções factíveis.

Nesse trabalho, será abordada a classe de problemas de otimização conhecida como problemas de caixeiro viajante, cuja descrição é dada na seção 3. Para resolvê-lo, iremos usar a abordagem de força bruta, que não é a solução usual, por ser menos eficiente na maioria dos casos. No entanto, o objetivo principal desse trabalho é implementar uma estrutura de dados que se adeque à formulação do problema dado e permita que as operações necessárias sejam realizadas de forma eficiente. É válido notar que, mesmo quando as operações sobre os dados podem ser realizadas eficientemente, não necessariamente o problema é resolvido de forma eficiente.

2. Abordagens de força bruta

Uma das alternativas para resolver problemas de otimização é calcular o valor da função objetivo para todas as soluções factíveis e escolher a solução em que a função objetivo tenha o menor valor possível. Abordagens desse tipo, que buscam todas as soluções factíveis, são chamadas de força bruta. Na seção 4, será analisado o desempenho desse tipo de abordagem para o problema em questão.

3. Problemas de caixeiro viajante

A classe de problemas do caixeiro-viajante é um conjunto de problemas clássicos na área de otimização não-linear cujo estudo foi importante no desenvolvimento de soluções de problemas envolvendo grafos. Originalmente, envolve um caixeiro-viajante que tem o objetivo de visitar um conjunto de cidades percorrendo a menor distância possível. Três restrições são impostas: o caixeiro não pode passar pela mesma cidade duas vezes, deve visitar todas as cidades, e deve finalizar seu percurso na mesma cidade em que começou.

4. Modelagem do problema

Para a modelagem do problema, podemos enumerar as cidades de 1 a n e definir distância para ir da cidade i até a cidade j como c_{ij}. É lógico que $c_{ij} = c_{ji}$, pois dadas duas cidades, a distância de i até j é a mesma de j para i. Além disso, para simlificação do problema, iremos supor que podemos viajar de qualquer cidade para qualquer cidade, ou seja $\exists c_{ij} \geq 0, \forall i \neq j$ e $i, j \in 1, ..., n$.

Notemos que a quantidade de cidades é conhecida de antemão, e que, se conhecermos c_{ij}, também conhecemos c_{ji}. Portanto, podemos inferir que uma solução adequada pode ser sequencial dinâmica, para alocarmos em tempo de execução a memória necessária. Também, não é necessário armazernar c_{ij} ao mesmo tempo que c_{ji}, o que permite a economia de memória. O acesso a determinada cidade é a operação mais frequente, enquanto a inserção é realizada apenas no início do problema e remoções só ocorrem ao fim.

Com essas informações, podemos descrever o conjunto de cidades por uma lista sequencial, em que o i-ésimo cada elemento representa a cidade i. Cada elemento i também contém uma lista, em que o seu j-ésimo elemento representa a distância até a cidade j. A figura 1 ilustra esse modelo.

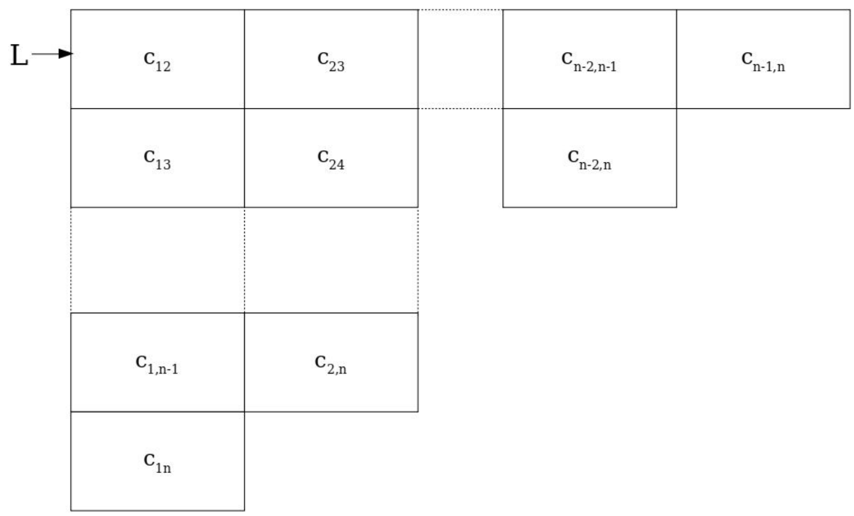

Figura 1: Representação gráfica de uma lista generalizada sequencial.

Com os custos armazenados em memória, resta o problema de calcular os custos de cada uma das soluções viáveis. A abordagem utilizada consiste em gerar todas as permutações possíveis das n cidades para a cidade inicial escolhida pelo usuário. Cada permutação representa uma solução viável e nos permite calcular o seu custo a partir dos custos armazenados c_{ij}. O menor custo e sua solução correspondente são guardados durante a execução do algoritmo e, toda vez que um novo mínimo é encontrado, a variável é atualizada.

Esse modelo seria inadequado se o custo de ir da cidade i para a cidade j fosse diferente de ir da cidade j para a cidade i. Além disso, ele parte da hipótese que é possível ir de qualquer cidade i para qualquer cidade $j \neq i$, o que dificilmente ocorre em casos reais. Ainda é custoso remover uma cidade, que exigiria a realocação de memória da lista principal, de todas as suas sublistas e, em cada sublista deslocar os seus elementos. Assim, teriamos $O(n)$ deslocamentos em cada uma das $n - 1$ sublistas e, portanto, a remoção seria $O(n^2)$.

Por outro lado, a indexação dos elementos permite o acesso de qualquer um deles a custo constante $O(1)$, que é a única operação de fato realizada após a inicialização da lista.

5. Algoritmo para gerar permutações

O algoritmo utilizado para gerar as permutações é o mostrado abaixo. Em resumo, a cada iteração i ele troca o primeiro elemento com o i-ésimo elemento, permuta o vetor a direita recursivamente, e troca novamente o i-ésimo elemento com o primeiro elemento novamente.

Algoritmo 1: Algoritmo de Permutação

Entrada: $vetor, inferior, superior$

Função permuta(vetor, inferior, superior) **início**
    $superior \leftarrow tamanho$;
    se $inferior == superior$ **então**
        calculaCusto($vetor$);
    fim
    $i \leftarrow 0$;
    Para $i \in \{inferior, ..., superior\}$
        troca($vetor[inferior]$, $vetor[i]$);
        permuta($vetor$, $inferior + 1$, $superior$);
        5 mm troca($vetor[inferior]$, $vetor[i]$);
    fim;
fim

Para calcularmos a sua complexidade, podemos notar que a cada chamada recursiva, ele realiza um loop de $n - 1$ passos, onde o n é o tamanho do vetor desde a posição $inferior$ até $superior$ e é decrementado a cada iteração. E, toda vez que chega à condição de parada, é necessário calcular o custo do vetor, que é uma operação $O(n)$. Então, são realizadas $O(n!)$ chamadas recursivas e cada uma delas implica em $O(n)$ operações para calcular o custo do vetor. Portanto, o algoritmo de permutação, em conjunto com o cálculo dos custos de cada vetor é $O(n \cdot n!)$.

Na prática, ainda temos que armazenar a solução com menor custo e seu respectivo custo. No pior caso, as soluções são verificadas em ordem decrescente de custo e devemos trocar o vetor armazenado sempre. A cópia de um vetor em C foi implementada copiando elemento a elemento, o que é $O(n)$. Isso torna a ordem do algoritmo realizado no fim das chamadas recursivas $O(2n) = O(n)$. A complexidade não é alterada, mas o fim da recursão leva o dobro de passos.

5.1. Detalhes sobre a implementação

O algoritmo foi implementado em C e compilado com o gcc 7.3.0 (Ubuntu 18.04). O arquivo principal é o ”CaixeiroTeste.c”. Os headers criados foram ”caixeiro.h”e ”combinatoria.h”. Os arquivos que contém a implentação das funções utilizadas são ”caixeiro.c”e ”combinatoria.c”. Para a compilação, foram utilizados os seguintes comandos:

```
$ gcc -g -c CaixeiroTeste.c -o CaixeiroTeste.o
$ gcc -g -c caixeiro.c -o  caixeiro.o
$ gcc -g -c combinatoria.c -o combinatoria.o
$ gcc CaixeiroTeste.o caixeiro.o combinatoria.o -o CaixeiroTeste
$ gcc ./CaixeiroTeste seu_arquivo.txt
```

6. Tempo de execução

Em testes, foi medido o tempo de execução do algoritmo utilizado para calcular o menor custo com auxílio da biblioteca *time.h* . Abaixo, seguem os gráficos descrevendo a medição do tempo em função da quantidade de cidades do problema. À esquerda temos medições de 4 até 7 cidades, e à direita, de 4 até 10 cidades.