Algoritmos e Estruturas de Dados I

Trabalho I – Caixeiro Viajante

Enzo Nunes Sedenho - 13671810

Gabriel da Costa Merlin - 12544420

Mateus Bernal Leffeck - 13673318

2° Semestre

Outubro– 2022

Universidade de São Paulo

Correção:

Foram enviados dois trabalhos no run.codes e para correção final o correto é o do discente Enzo Nunes Sedenho - NUSP 13671810

- Objetivo:

O objetivo do trabalho é encontrar a menor rota para que um viajante possa deslocar-se entre um número X de cidades com a distância sendo a menor possível, para isso nosso programa em C utiliza a força bruta, ou seja, testa todas as combinações para encontrar a melhor.

Para realizar esse objetivo temos que usar algumas bibliotecas da linguagem C, sendo elas:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <assert.h>
```

- **Parte 1: Modelagem da solução:**

Para elaborar a solução para o PCV, o grupo optou pela utilização de uma lista encadeada dinâmica.

A principal vantagem que justifica essa escolha é o fato de que a lista encadeada nos permite percorrer a lista facilmente, e acessar um NO inserido em qualquer posição (o que não acontece com pilhas ou filas, uma vez que você acessa sempre o NO inicial ou final). Essa facilidade é essencial para a solução por força bruta do PCV, como explicado posteriormente no tópico de implementação, uma vez que nos permite filtrar o NO desejado armazenando apenas seu índice, sem que seja necessária a manipulação da estrutura em si. Além disso, por ser uma lista dinâmica, podemos trabalhar com uma quantidade de memória que não seja previamente definida, no caso do problema, o programa funciona para qualquer que seja o número de cidades fornecido.

Já como desvantagens da estrutura adotada, podemos citar a inflexibilidade para percorrer a lista, uma vez que não temos um ponteiro apontando para o NO anterior em cada NO (não se trata de uma lista duplamente encadeada). Ademais, a lista dinâmica simplesmente encadeada que utilizamos no problema não é generalizada, e portanto não apresenta profundidade.

- Parte 2: Explicação da lógica do problema:

Para solucionar o problema através de uma abordagem por força bruta, utilizamos um algoritmo de “backtracking”, uma vez que todas as rotas possíveis podem ser encontradas permutando-se as cidades que apresentam conexões entre si.

Assim, implementamos uma função (explicada na parte 3) que, partindo de uma cidade de origem, inicia um laço de repetição que vai de 1 até o número de cidades criadas. A cada realização do laço a função chama a si mesma novamente, dessa forma, partindo de uma cidade de origem, o primeiro laço criaria n rotas distintas (em que n é o número total de cidades que são conectadas à origem, criando assim uma ramificação para cada valor entre 1 e o número de cidades), na sequência, todas essas rotas distintas passariam pela função novamente, chamariam novamente a função, adicionando mais uma cidade à rota, e gerando mais ramificações para o caminho seguido pelo caixeiro.

Esse processo chega ao fim quando uma rota completa é formada, ou seja, quando o número de cidades visitadas ao longo das chamadas da recursão é igual ao número de cidades criadas, nesse caso, o caixeiro volta para a origem, e o programa julga a rota realizada. Caso a distância da rota em questão seja menor até o momento, a rota que foi completa na última chamada assume o posto de melhor rota. Finalmente, a recursão é encerrada e rota imediatamente anterior passa pelos mesmos passos descritos anteriormente.

E assim segue a lógica até que não reste mais nenhuma rota para ser avaliada.

- Parte 3: Explicação do código:

- Estruturas utilizadas (pcv.h e pcv.c):

● Struct no_t (NO):

A struct no_t é formada por dois inteiros, o índice que representa o número da cidade (id), uma flag chamada “visitado” que tem como função indicar se a cidade já foi visitada durante a construção de uma rota, uma vez que a mesma cidade não pode ser visitada mais de uma vez.

A Struct também possui um ponteiro de inteiro para armazenar a distância de uma cidade às outras (distância do NO atual até a cidade na posição "índice – 1” do vetor). Por fim, temos um ponteiro para NO que aponta a próxima cidade da lista.

● Struct lista_pcv_t (LISTA_PCV):

Essa struct apresenta dois ponteiros da struct NO, um para o início e outro para o fim da lista, além de 3 inteiros: um para guardar o número total de cidades (n_cidades), outro para o valor da menor distância (menor_distancia), correspondente à distância encontrada na melhor rota, e outro para armazenar a distância atual (distancia_atual).

Além disso, temos dois ponteiros de inteiros para serem usados como vetores, sendo o “rota_atual” para armazenar o percurso feito até o momento, e “menor_rota” para guardar a rota que corresponde à menor distância percorrida.

- Funções da resolução do PCV (backtracking.h e backtracking.c):

● void forca_bruta(LISTA_PCV *rota, int n_cidades):

A principal utilidade desta função é inicializar os dados da lista para iniciar o backtracking.

Recebe como parâmetro a lista e o número de cidades. De início verificamos se a lista de rotas é diferente de NULL com a função assert.

Em caso positivo criamos um ponteiro de inteiro para receber a rota atual com a função “lista_get_rotal_atual”. Inicializamos a “menor_distancia” como INT_MAX e a primeira posição da rota atual como o índice da cidade de origem. Feito isso, chamamos a função “forca_bruta_recursao”.

Após a resolução da função “forca_bruta_recursao”, imprimimos a menor_rota da lista na tela.

● void forca_bruta_recursao(LISTA_PCV *rota, int n_visitadas, int n_cidades):

Esta é a principal função para a solução (força bruta) do PCV, seu objetivo é permutar todas as rotas possíveis, avaliando as distâncias percorridas em cada uma.

Criamos dois NOs, “origem” e “no_atual”, estes têm como função, respectivamente, guardar o primeiro NO da lista e o último visitado (obtido através do vetor “rota_atual”) o vetor “rota_atual” recebe a rota atual da lista.

No primeiro “if” o programa verifica se todos os NOs foram visitados, ou seja, se o número de cidades visitadas é igual ao número total de cidades e se o NO atual tem conexão com a origem. Em caso positivo, o último índice do vetor “rota_atual” recebe a cidade de origem e adicionamos ao campo “distancia_atual” a distância do NO atual ao NO de origem. Com isso, o segundo “if” verifica se a distância da atual rota é menor que a menor distância encontrada até o momento, caso seja, atribuímos a rota atual ao campo “menor_rota” da lista, e a distância atual ao campo “menor_distancia”. Ao fim do segundo “if”, subtraímos a distância da cidade de origem para a cidade final (para que o cálculo das próximas distâncias não seja prejudicado) e encerramos a recursão.

A função agora testa todas as rotas possíveis através de um algoritmo de “backtracking”. Para isso, inicia um for com a variável “j” indo de 1 até o número de cidades. A cada realização do for, o NO “prox_no” recebe o NO correspondente ao índice “j”, a variável "distância" recebe a distância entre “no_atual” e “prox_no”.

Para continuar a criar a rota, o programa verifica 3 informações: Se “prox_no” ainda não foi visitado e é diferente da origem, e se “distancia” é diferente de 0 (caso a distância seja 0, as cidades não tem conexão uma com a outra). Em seguida, somamos a variável “distancia” ao campo “distancia_atual” da lista, definimos “prox_no” como visitado e adicionamos seu índice à posição atual do vetor “rota_atual”.

Na sequência, a função se chama novamente, incrementando o número de cidades visitadas. Finalmente, após a recursão ser encerrada, voltamos a flag de “visitado” de cada NO para 0 e decrementamos as distâncias somadas ao longo da recursão.

- main.c:

Na main inicializamos as variáveis utilizadas para preencher nossa lista.

Começamos lendo a quantidade de cidades e a cidade de origem do viajante, após isso criamos o primeiro NO (origem) e inserimos ele na lista.

Na sequência lemos o arquivo de entrada (até EOF) de acordo com o formato especificado pelo enunciado do problema. Ao ler duas cidades e uma distância, o programa verifica a existência de cada cidade e, caso ela não exista, criamos um novo NO relativo ao índice lido. Em seguida, setamos a distância entre as duas cidades.

Após preencher todos os NOs e a lista, vamos para a busca da melhor rota via força bruta. Ao fim do programa, liberamos a memória alocada para a lista.

Parte 4 – Análise de complexidade

Tempo de execução

A parte principal do Problema do Caixeiro Viajante feito através de força bruta é o teste de todas as possibilidades de caminhos possíveis a serem feitos. De acordo com os princípios da análise combinatória, se queremos saber de quantas formas diferentes n objetos podem estar dispostos, precisamos utilizar a permutação.

Assim, para uma dada cidade de origem, a função busca visitar todas as cidades distintas ainda não visitadas, portanto, se chamaria novamente (n – 1) vezes, já que uma cidade foi visitada. Em seguida, para cada uma dessas diferentes chamadas, a função se chamaria novamente (n – 2) vezes, já que agora duas cidades foram visitadas, e assim por diante.

Generalizando esse raciocínio para uma forma matemática, podemos equacionar o número de chamadas como sendo:

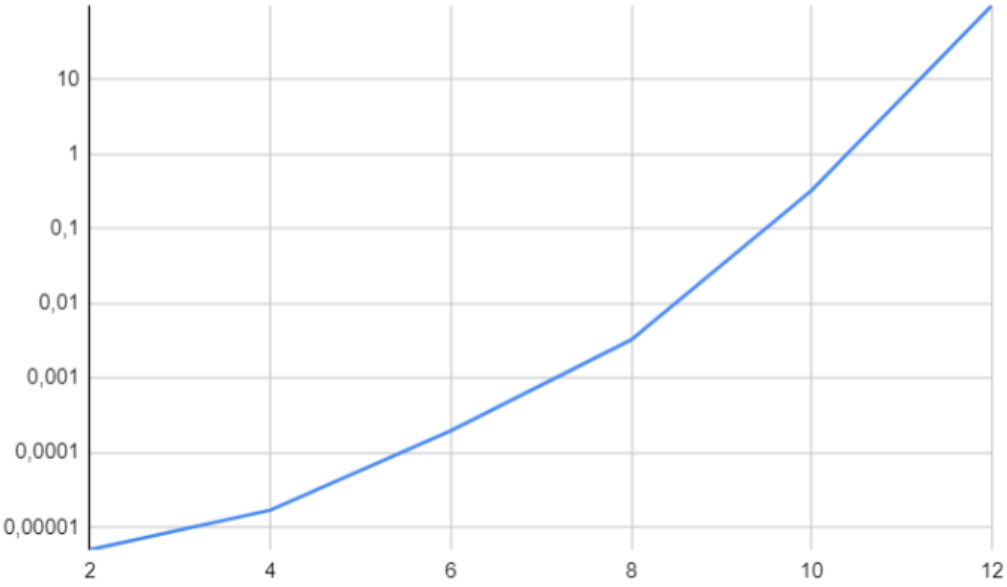
n * (n – 1) * (n – 2) * … * 2 * 1 = n!, Para qualquer n >= 2.

Assim, se queremos visitar n cidades, elas poderão estar dispostas de n! formas diferentes, o que configura uma complexidade de O(n!) para o backtracking. Em nossa implementação, possuímos funções de busca com complexidade O(n) e funções de retorno com complexidade O(1).

Dessa forma, utilizando a notação Big O, a solução força bruta para o PCV apresenta complexidade de O(n!).

Obs.: Para maximizar o tempo de execução, em todos os casos abaixo, todas as cidades são conectadas com todas as cidades

`Tempo de backtracking: 0.000005` 2 Cidades	`Tempo de backtracking: 0.000017` 4 Cidades
`Tempo de backtracking: 0.000396` 6 Cidades	`Tempo de backtracking: 0.013954` 8 Cidades
`Tempo de backtracking: 0.326950` 10 Cidades	`Tempo de backtracking: 42.094032` 12 Cidades



Gráfico

Obs.: Para evidenciar o incremento no tempo de execução, o gráfico omite o tempo para n = 12, uma vez que a escala ficaria muito distorcida, o que não ajudaria a enxergar a variação do eixo Oy.

Parte 5 – Otimização do código:

Para otimizar o código, foram feitas duas alterações:

● Função main: Foi declarada a variável “menor_dist_lida”, a qual foi inicializada com MAX_INT.

Agora, durante a etapa de leitura do arquivo, sempre que a distância lida for menor que “menor_dist_lida”, armazenamos a distância lida na variável. Dessa forma, ao fim da leitura, armazenamos a menor distância presente na rota.

● Forca_bruta_recursão (backtracking.h e backtracking.c):

Na versão mais otimizada, inserimos uma comparação, a qual é realizada logo no início da função. Essa comparação confere se a distância atual da rota, somada à menor distância registrada na leitura, é maior ou igual à distância realizada na melhor rota até o momento (menor distância dentre os caminhos possíveis encontrados até o momento).

Caso a comparação seja verdadeira, isso significa que, independente de qual seja a próxima cidade visitada, a rota já não é mais vantajosa, e portanto pode ser descartada. Assim, caso a comparação se confirme, o programa simplesmente utiliza um “return”, e a rota é descartada. O restante do código é idêntico à versão força bruta.

Obs.: Como na versão otimizada algumas rotas são descartadas, a resposta pode não bater com a do run codes, já que podem haver rotas com as mesmas distâncias.

Parte 5.1 – Análise de complexidade da versão otimizada:

Tempo de execução:

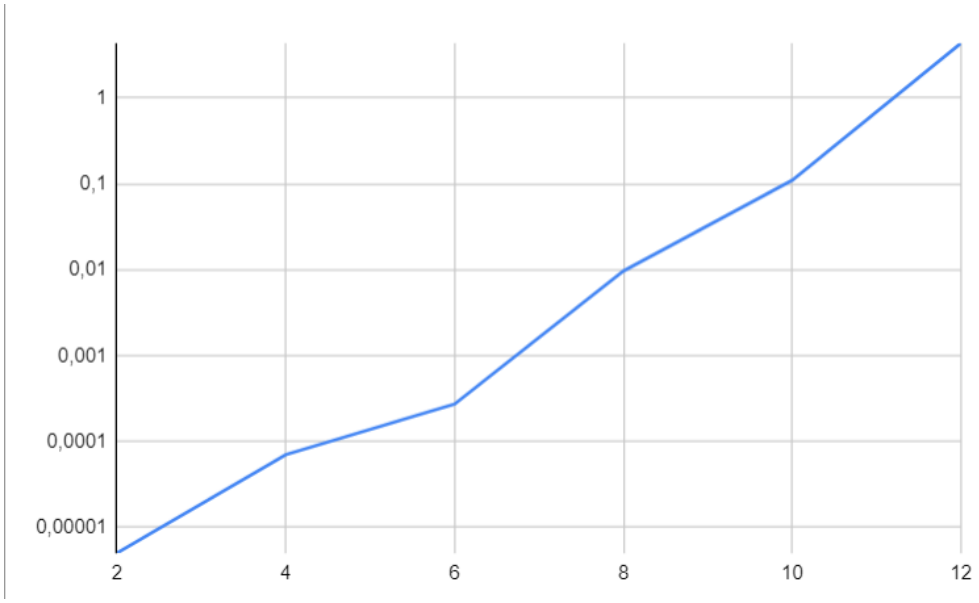
`Tempo de backtracking: 0.000005` 2 Cidades	`Tempo de backtracking: 0.000007` 4 Cidades
`Tempo de backtracking: 0.000272` 6 Cidades	`Tempo de backtracking: 0.009638` 8 Cidades
`Tempo de backtracking: 0.109683` 10 Cidades	`Tempo de backtracking: 4.357811` 12 Cidades

Para a análise de complexidade, levaremos em consideração o pior caso de sucesso do programa. Esse contexto ocorre quando a condicional inserida na otimização nunca se mostra verdadeira, ou seja, quando a próxima rota é mais vantajosa que a anterior em todas as etapas da recursão.

Nesse caso, perceba que ao fim da solução, todas as rotas terão sido permutadas, e portanto não haverá vantagem na complexidade quando analisamos o pior caso, configurando assim uma complexidade de O(n!), ídem à versão força bruta.

No entanto, é preciso compreender que o caso citado anteriormente é muito improvável. Usualmente, é esperado que a condicional inserida elimine muitas rotas durante a execução do programa, diminuindo consideravelmente a complexidade (vide os tempos de execução acima). Dessa forma, concluímos que embora a complexidade de ambos os programas seja a mesma para o pior caso, o caso médio da versão otimizada apresenta significativa vantagem.

Gráfico



Obs.: Para evidenciar o incremento no tempo de execução, o gráfico omite o tempo para n = 12, uma vez que a escala ficaria muito distorcida, o que não ajudaria a enxergar a variação do eixo Oy.