Relatório do Trabalho de Ordenação e Estatísticas de Ordem

Técnicas de Programação Avançada — Ifes — Campus Serra

Alunos: Douglas Bolis,Jadson Pereira e Guilherme Bodart

Prof. Jefferson O. Andrade

29 de setembro de 2018

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

Este trabalho consiste na resolução de um conjunto de problemas envolvendo algoritmos gulosos, programação dinâmica e algoritmo em grafos vistos na disciplina de Tópicos Avançados de Programação.

Todos os problemas propostos neste trabalho estão disponíveis no site UVa Online Judge.

2 Implementação do Trabalho

2.1 Programação Dinâmica

2.1.1 UVa 00108 – Maximum Sum

Figura 1: Mensagem de Aceitação do *UVa 00108 – Maximum Sum*.

O problema consiste em resolver em uma retângulo, matriz, achar a maior soma possível de um sub-retângulo, sub-matriz, que é chamado de sub-retângulo máximo, um sub-retângulo é qualquer sub-matriz de tamanho 1x1 ou superior.

A entrada do problema consiste em um N, que define o tamanho da matriz N*N, seguidos de N*N números que preenche a matriz.

A saída do problema consiste em um número que é a maior soma possível de algum sub-retângulo.

Primeiramente foi feito em python a solução não dinâmica do problema, consiste em uma solução O(n^6), porém o problema precisa de uma solução no máximo de O(n^4), a solução inicial foi usando o 4 for principais, estes 4 loops são feitos para percorrer a matriz um por um, e para cada elemento ir criando um sub-retângulo com a posição inicial 0 até posição final y, porém assim será feito somas repetidas, porque para cada 0 até y, será feito uma soma, no entanto, haverá somas repetidas. (O programa em python com O(n^6) está sendo enviado junto nos códigos)

Para a verdadeira solução foi feita usando uma lista que contém as somas pré-calculadas de cada posição da matriz, até aquele ponto, por exemplo a posição matriz[1][1] tem a soma das posições [0][0],[0][1],[1][0],[1][1], menos a posição de [0][0], já que é interseção de [0][1] e [1][0] e é somada duas vezes, esse pré-cálculo tem um custo de O(n^2). O resultado é a soma de todas as sub-matrizes começando do 0 até um elemento x.

Depois é feito uma avaliação de todos os pontos iniciais e finais possíveis, usando os 4 for, O(n^4). Como já temos uma lista das somas sub-matrizes do ponto inicial até um elemento x, para calcularmos um valor de sub-matriz de [2][2] até [3][3], precisamos da sub-matriz até [3][3] e retirar as sub-matriz que não fazem parte do que queremos, no caso de [0][0] até [1][3] e de [0][0] até [3][1],onde os valores já estão calculados na lista de somas, porém como antes, é retirada duas vezes a interseção das duas, por isso precisamos somar ela novamente. Com isso iremos passar por todas as sub-matrizes, calcular suas somas e saber qual é a maior.

```cpp
#include <cstdio>
#include <stack>
using namespace std;
int main() {
    int listaSoma[100][100];
    int soma = 0;
    int maxSoma;
    int dim;
    scanf("%d",&dim);
    for(int n=0;n<dim;n++){
        for(int m=0;m<dim;m++){
            scanf("%d",&listaSoma[n][m]);
            if(m>0){
                listaSoma[n][m] = listaSoma[n][m] + listaSoma[n][m-1];
            }
            if(n>0){
                listaSoma[n][m] = listaSoma[n][m] + listaSoma[n-1][m];
            }
            if(m>0 && n>0){
                listaSoma[n][m] = listaSoma[n][m] - listaSoma[n-1][m-1];
            }
        }
    }
    maxSoma = -1000;
    for(int i=0;i<dim;i++){
        for(int j=0;j<dim;j++){
            for(int lin=i;lin<dim;lin++){
                for(int col=j;col<dim;col++){
                    soma = listaSoma[lin][col];
                    if(i>0){
                        soma = soma - listaSoma[i-1][col];
                    }
                    if(j>0){
                        soma = soma - listaSoma[lin][j-1];
                    }
                    if(i>0 && j>0){
                        soma = soma + listaSoma[i-1][j-1];
                    }
                    if(soma > maxSoma){
                        maxSoma = soma;
                    }
                }
            }
        }
    }
    printf("%d\n",maxSoma);

}
```

Código Fonte 1: Solução do problema do maximum sum - uva 00108

2.1.2 UVa 10684 – The Jackpot

#	Problem	Verdict	Language	Run Time	Submission Date
24281523	10684 The jackpot	Accepted	PYTH3	0.360	2019-12-04 17:03:49

Figura 2: Mensagem de Aceitação do *UVa 10684 - The Jackpot*.

O problema consiste em desenvolver um programa que identifique o máximo de ganho possível em uma sequência de apostas, uma aposta é uma quantidade de dinheiro que se ganha ou se perde.

Analisando Código Fonte 2, a primeira informação a saber é a quantidade de apostas a serem analisadas(linha 3 a 7), a entrada de dados podem conter as apostas em diferentes linhas, para tratar isso, com o auxilio de uma lista vazia, nós inserimos as apostas até ter todas em mãos (linha 8 a 15).

Com as apostas em mãos, é feito a varredura da lista de apostas, com o auxilio de duas variáveis para acumular a soma das apostas (acc1) e para registrar o valor do máximo de ganho possível (acc), em cada iteração é feito uma soma do acumulador com o atual valor, como as apostas podem ser negativas e positivas, em caso da soma resultar em um valor negativo o acumulador é zerado, caso contrário, é verificado se o acumulador com a soma do atual número é maior que o acumulador, caso seja, é porque esse valor é o máximo de ganho possível atual (linha 15 a 24).

O programa é finalizado caso a entrada de dados seja igual a 0, onde não há apostas a serem lidas, para cada sequência de apostas lidas e processadas, o saída de dados é o máximo de ganho possível, caso seja um número maior que zero, caso contrário é retornado a mensagem "Losing Streak"(linha 26 a 30).

```
while True:
    try:
        nNumeros = input()
        if(nNumeros != ''):
            nNumeros = int(nNumeros)
            if(nNumeros == 0):
                break
            values = []
            while(nNumeros != 0):
                txt = input()
                lstNum = txt.split()
                tam = len(lstNum)
                for num in lstNum:
                    values.append(int(num))
                nNumeros -= tam
            acc = 0
            acc1 = 0              #Acumulador
            for i in range(len(values)):
                acc1 += values[i]
                if(acc1<0):
                    acc1 = 0
                else:
                    if(acc1>acc):
                        acc = acc1

            if(acc <=0):
                print("Losing streak.")
            else:
                print("The maximum winning streak is "+str(acc)+".")
            acc = 0

    except EOFError:
        break
```

Código Fonte 2: Solução do problema do the jackpot - uva 10684

2.2 Algoritmos Gulosos

2.2.1 UVa 11100 – The Trip, 2007

Figura 3: Mensagem de Aceitação do *UVa 11100 – The Trip, 2007.*

O problema consiste em saber em quantas sacolas podem ser carregadas todas as sacolas que eles recebem dos patrocinadores, sendo que uma sacola menor cabe em outra maior, e que tem que tentar equilibrar a divisão para não ficar com quantidades de sacolas por sacolas muito diferente.

A entrada é um N dizendo quantas sacolas tem em seguidas com N números dizendo o tamanho de cada sacola, não necessariamente estará em alguma ordem. E termina caso na hora de ler o N, o mesmo seja 0.

Comecei fazendo esta questão em python, porém ao terminar a questão com tudo certo, o problema não foi aceito no UVa com o Run Time Error, como não conseguir achar onde era, passei para C++ e foi aceito. Enviarei os dois códigos, python e c++.

Neste problema a pré-escolha era saber quantas sacolas iriam ter no final. O melhor jeito de dividir, já que uma sacola de tamanho x não cabe em outra sacola de tamanho x, é ver qual tamanho mais se repete e usar ele como parâmetro. Neste problema existe várias soluções, pelo que o próprio enunciado diz, então é possível fazer uma outra divisão, como por exemplo, raiz do número total de sacolas, mas teria que tratar alguns problemas mais a frente.

Como o número máximo de repetições pode ser igual ao números de sacolas, caso use alguma outra solução, como a raiz, teria que tratar este problema e ir "criando"outras listas com as sacolas. Com a solução da divisão usando a quantidade máxima de repetições não é preciso tratar nenhum problema futuramente.

A solução consiste em ler tudo, colocar a lista de tamanho em ordem, ver a quantidade que mais se repete e a partir dai ir para a solução. Criei uma matriz que tem as dimensões sendo (qtdRepetidasMax+2 * (n/qtdRepetidasMax)+2) preenchidas com -1, a primeira é a quantidade de sacolas, e a segunda é quantas sacolas terá dentro de umas mesma sacola. (+2 em cada dimensão para poder usar o -1 adiante).

O loop da linha 31, é a solução real do problema, onde eu leio a lista de tamanho e vou colocando um por um na matriz, sendo que usei duas variáveis, 'i' e 'k', para percorrer a matriz, e como a quantidade de sacolas é a qtdRepetidasMax, o i é somado até ele, e depois reinicia e o k é somado cada vez que reinicia.

Após isso só é preciso mostrar na tela, mas como o vetor criado tem tamanho maior que do realmente foi criado, o "lixo", e o tamanho dos vetores dentro dos vetores não são todos iguais, foi colocado o -1 para saber quando uma linha termina para passar para a próxima.

```cpp
#include <cstdio>
#include <stack>
#include <vector>
#include  <bits/stdc++.h>
using namespace std;

int main() {
    int n;    int lista[10005];
    while(scanf("%d",&n) && n){
        for (int i=0;i<n;i++){
            scanf("%d",&lista[i]);
        }
        sort(lista, lista+n);
        int qtdRepetidasMax = 1; int qtdRepetidas = 1;

        for(int i=0;i<n-1;i++){
            if(lista[i]==lista[i+1]){
                qtdRepetidas = qtdRepetidas + 1;
            }
            else{
                if(qtdRepetidas>=qtdRepetidasMax){
                    qtdRepetidasMax = qtdRepetidas;
                }
                qtdRepetidas = 1;
            }
        }
        if(qtdRepetidas>=qtdRepetidasMax){qtdRepetidasMax =qtdRepetidas;}

        vector< vector<int> >
        ↪  mochilas(qtdRepetidasMax+2,vector<int>((n/qtdRepetidasMax)+2,-1));
        int i=0; int k=0;
        for (int j=0;j<n;j++){
            mochilas[i][k] = lista[j];
            i = i + 1;
            if(i>=qtdRepetidasMax){i=0;k=k+1;}
        }
        printf("%d\n",qtdRepetidasMax);
        for (int i=0;i<qtdRepetidasMax;i++){
            int j=0;
            while(mochilas[i][j]!=-1){
                if(mochilas[i][j+1]==-1){printf("%d",mochilas[i][j]);}
                else{printf("%d ",mochilas[i][j]);}
                j = j + 1;
            }
            printf("\n");
        }
    }
}
```

Código Fonte 3: Solução do problema do the trip, 2007 - uva 11100

2.2.2 UVa 12405 – Scarecrow

O problema do Scarecrow consiste em saber a quantidade miníma de espantalhos no campo para proteger a área de cultivo dos corvos. Nesse problema, um ponto (.) indica um ponto de cultivo, um hash indica uma região infértil.

Analisando Código Fonte 4, inicialmente se lê a quantidade de casos (linha 1), em seguida é lido a quantidade de campo, representado pela variável 'n' e depois o texto de tamanho n, representado pela variável field (linha 7 a 9).

Com isso em mãos, é feito a varredura do texto, verificando se a posição atual é um ponto de cultivo ou não, caso seja - é adicionado um espantalho no campo ao lado e se anda 3 campos, pelo fato de cada espantalho proteger o campo da direita e o campo da esquerda da sua atual localidade, caso não seja um ponto de cultivo(linha 13 a 16), caso não seja - não é feito.

#	Problem	Verdict	Language	Run Time	Submission Date
24265491	12405 Scarecrow	Accepted	PYTH3	0.010	2019-11-30 17:48:12

Figura 4: Mensagem de Aceitação do *UVa 12405 – Scarecrow*.

```
1  nCases = input()
2  case = 1
3  while(True):
4      if(case == nCases): break
5      try:
6          n = input()
7          n = int(n)       # Numero de ca
8          field = input()
9          scarecrow = 0
10         i=0
11         while(i<n):
12             if(field[i]=='.'):
13                 scarecrow+=1
14                 #Cobre três celulas <left> e <right>
15                 i+=3
16             else:
17                 i+=1
18         print("Case "+str(case)+': '+str(scarecrow))
19         case += 1
20
21     except EOFError:
22         break
```

Código Fonte 4: Solução do problema do scarecrow - uva 12405

2.3 Algoritmos em Grafos

2.3.1 UVa 00280 – Vertex

O problema *Vertex* propõe a procura em alguns grafos direcionados por vértices inacessíveis a partir de um determinado vértice inicial.

Um grafo direcionado é representado por n vértices onde $1 \leq n \leq 100$, numerado consecutivamente $1...n$, e uma série de arestas p -> q que conectam o par de nós p e q em uma única direção.

Um vértice r é alcançável a partir de um vértice p se existe uma aresta p -> r, ou se existe algum vértice q para o qual q é alcançável a partir de p e r é alcançável a partir de q. E um vértice r é inacessível a partir de um vértice p se r não for acessível a partir de p.

E para resolução desse problema foi desenvolvido o Código Fonte 5 e analisando o código, inicialmente lemos e armazenamos os dados de entrada das estruturas dos grafos como o número de vértices e as arestas direcionadas*(linhas 14 a 45 da primeira página do código)*. As arestas são armazenadas em uma matriz onde as linhas são os vértices de origem e as colunas são os vértices de destino e se um determinado vértice i possui uma aresta para outro vértice j a posição $[i]j]$ da matriz é setada com *true(linha 43)*.

Com os dados em mãos vamos para a resolução do problema e para isso foi desenvolvido a classe *Vertex* com o método *solve* para resolver o problema*(linha 30 da segunda página do código)*.

Como a identificação de vértices inacessíveis é a partir de um vértice inicial, então foi desenvolvido o método *walkGraph(linhas 6 a 14 da terceira página do código)* na classe *Vertex* que mapeia em uma *array* os vértices acessíveis ao vértice inicial.

Com o caminho mapeado é realizado uma verificação para os vértices que não foram identificados no mapeamento dos vértices, e se o vértice não estiver na lista temos um outro *array* para armazenar os vértices inacessíveis, que na primeira posição é para o cálculo da quantidade de vértices inacessíveis e as posições subsequente são armazenados os vértices inacessíveis*(linhas 34 a 41 da segunda página do código)*.

E por fim é impresso como saída do código o número de vértices inacessíveis na grafo a partir de um vértice inicial e os vértices inacessíveis encontrados.

#	Problem	Verdict	Language	Run Time	Submission Date
24300100	280 Vertex	Accepted	JAVA	1.440	2019-12-09 15:34:19

Figura 5: Mensagem de Aceitação do *UVa 00280 – Vertex*.

```java
import java.util.Scanner;
import java.util.StringTokenizer;

class Main {
    private final Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        Main main = new Main();
        main.run();
    }

    public void run() {
        // Cache para armazenar as linhas que contenham os números de vértices
        ↪ nos grafos.
        String line;

        while ((line = scanner.nextLine()) != null) {
            // Número de vértices no grafo.
            int vertices = Integer.parseInt(line);

            if (vertices == 0) {
                break;
            }

            boolean [][] graph = new boolean [vertices][vertices];

            // Identificando os vértices que possuem arestas direcionadas.
            while (true) {
                StringTokenizer st = new StringTokenizer(scanner.nextLine());
                int verticeSrc = Integer.parseInt(st.nextToken()) - 1;

                // Identificando o fim da definição das arestas direcionadas.
                if (verticeSrc == -1) {
                    break;
                }

                while (st.hasMoreTokens()) {
                    int verticeDest = Integer.parseInt(st.nextToken()) - 1;
                    if (verticeDest == - 1) {
                        break;
                    }

                    // Pegando os vértices de destino e definindo a aresta
                    ↪ entre origem e destino como true.
                    graph[verticeSrc][verticeDest] = true;
                }
            }

            // Pegando os vértices para iniciar as buscas.
            StringTokenizer st = new StringTokenizer(scanner.nextLine());
            int verticesResearch = Integer.parseInt(st.nextToken());
```

```java
        for (int i = 0; i < verticesResearch; i++) {
            // Pegando o vértice inicial para iniciar as buscas.
            int verticeStart = Integer.parseInt(st.nextToken()) - 1;

            Vertex vertex = new Vertex(vertices, graph, verticeStart);
            vertex.solve();

            System.out.println();
        }
    }
}

class Vertex {

    int [] inaccessibleVertices;
    boolean [][] graph;
    boolean [] walked;
    int vertices;
    int vertice;

    Vertex(int vertices, boolean [][] graph, int vertice) {
        this.walked = new boolean [vertices];
        this.graph = graph;
        this.vertices = vertices;
        this.vertice = vertice;
        // Posição 0 conterá a quantidade de vértices inacessíveis.
        this.inaccessibleVertices = new int [vertices + 1];
    }

    public void solve() {
        // Caminhando através do vértice inicial para mapear os vértices que
        ↪  possui acesso.
        walkGraph(vertice);

        for (int i = 0; i < vertices; i++) {
            if (!walked[i]) {
                // Contabilizando a quantidade de vértices inacessíveis.
                inaccessibleVertices[0]++;
                // Armazenando o número do vértice que é inacessível.
                inaccessibleVertices[inaccessibleVertices[0]] = i + 1;
            }
        }

        // Imprimindo os vértices que o inicial não acessa.
        for (int i = 0; i <= inaccessibleVertices[0]; i++) {
            System.out.print(inaccessibleVertices[i]);
            if (i < inaccessibleVertices[0]) {
                System.out.print(" ");
            }
        }
    }
```

```
1           /**
2            * Caminha pelo grafo identificando os vértices acessíveis pelo vértice
    ↪  inicial.
3            *
4            * @param start Vértice inicial.
5            */
6           public void walkGraph (int start) {
7               for (int i = 0; i < graph.length; i++) {
8                   if (graph[start][i] && !walked[i]) {
9                       // Se o vértice é acessível e ainda não foi sinalizado como
                        ↪  caminhado recebe true.
10                      walked[i] = true;
11                      walkGraph(i);
12                  }
13              }
14          }
15
16      }
17  }
```

Código Fonte 5: Solução do problema do *UVa 00280 – Vertex*

2.3.2 UVa 00459 – Graph Connectivity

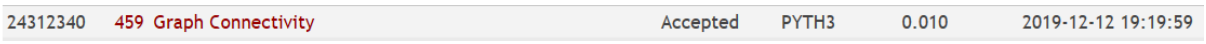

Figura 6: Mensagem de Aceitação do *UVa 00459 - Graph Connectivity*.

O problema consiste em um gráfico G formado a partir de um grande número de nós conectados, a questão do problema é saber quantos subgrafos máximos existem em um determinado grafo. Um subgrafo conectado é máximo se não houver nós e arestas no gráfico original que possam ser adicionados ao subgrafo e ainda o deixar conectado.

A entrada começa com um número, que diz a quantidade de testes a seguir, depois uma linha vazia, depois uma linha com um único caractere, de A-Z maiúsculo, em seguida de N duplas de caracteres, de A-Z maiúsculos, até uma próxima linha vazia.

A saída consiste apenas na quantidade de subgrafos máximos surgidos a partir dos dados de entrada, mais uma linha vazia a baixo.

Para resolver foi feito um dicionário com as chaves de A-Z, com seus respectivos números de 1-26, que foi usado apenas para formar inicialmente uma lista de listas contendo valores de A-Z em cada uma das listas de dentro, por exemplo se o valor inicial for 'I', vai criar uma lista assim: [[], ['A'], ['B'], ['C'], ['D'], ['E'], ['F'], ['G'], ['H'], ['I']]; uma lista vazia no inicio para o 'A' ser 1, 'B' ser 2 e assim por diante.

Depois de criado a lista com todos os nós daquele caso, vai começar a ler os nós que estão ligados, AB, CD, GI, etc. Para cada par de nó lido, é procurado na lista o valor de cada um nó e em qual posição que eles estão com a função find(), caso a posição deles seja diferente, usa a função extend(), que faz união da lista x com a lista y, e depois é deletada a lista y, já que extend() apenas pega os valores da lista y e adiciona na lista x, não transforma as duas em uma, e em seguida e lista outra duplas de nós e repete este processo até achar uma linha vazia, e no final é mostrado o tamanho da lista-1, por causa da lista vazia criada no início.

```
def find(lista,a):
    for i in range(len(lista)):
        for j in range(len(lista[i])):
            if a==lista[i][j]:
                return i
    return 0

def main():
    num = int(input())
    dic = {'A':1,'B':2,'C':3,'D':4,'E':5,
            'F':6,'G':7,'H':8,'I':9,'J':10,
            'K':11,'L':12,'M':13,'N':14,'O':15,
            'P':16,'Q':17,'R':18,'S':19,'T':20,
            'U':21,'V':22,'W':23,'X':24,'Y':25,'Z':26}
    for z in range(num):
        if z == 0:
            vazio = input()
        no = input()
        lista = [[]]
        x=1
        for i in dic:
            lista.append([])
            lista[x].append(i)
            if i==no:
                break
            x = x + 1

        a = input()
        while(a!=""):
            x = find(lista,a[0])
            y = find(lista,a[1])
            if x != y:
                lista[x].extend(lista[y])
                del lista[y]
            try:
                a = input()
            except:
                break
        if z<num-1:
            print(len(lista)-1)
            print()
        else:
            print(len(lista)-1)
    return 0
if __name__ == "__main__":
    main()
```

Código Fonte 6: Solução do problema do *UVa 00459 – Graph Connectivity*