Relatório do Trabalho Problemas sobre Estruturas de Dados Básicas

Técnicas de Programação Avançada — IFES — Campus Serra

Alunos: Antônio Carlos Durães da Silva,
Carlos Guilherme Felismino Pedroni,
Lucas Gomes Fleger

Prof. Jefferson O. Andrade

04 de dezembro de 2019

Sumário

Lista de Códigos Fonte

Lista de Figuras

1 Introdução

A fim de minimizar a quantidade de código-fonte contida neste relatório, o grupo optou por remover comentários e converter algumas operações para operações inline, sendo assim, o código-fonte com maior ênfase em formatação e documentação estará em um repositório[1] da plataforma Github ou no diretório anexado juntamente a este relatório.

O(s) autor(es) de cada solução encontra-se no topo de cada código-fonte no repositório ou diretório. O diretório de cada problema também guarda a imagem que atesta a aceitação do Juiz Online.

2 Implementação do Trabalho

Este capítulo está divido em pequenas seções, onde o objetivo de cada seção é exibir o código fonte e uma breve introdução de como o problema foi solucionado.

2.1 Programação Dinâmica

2.1.1 #00108 – Maximum Sum

Após indagar-se sobre o problema de maximizar a soma dos elementos contidos em uma matriz e pesquisar sobre, descobrimos que há um algorítimo clássico que realiza essa operação para arrays ou lista de uma dimensão, o **algorítimo de Kadane**[2].

Após ler a matriz, cada coluna k em um array auxiliar é preenchida com o acumulado de todos elementos de índice k de linhas anteriores até a linha atual [linha 28 a 33]. Dessa forma, o array auxiliar com os acumulados de cada coluna poderá ser processado pelo método de Kadane [linha 35]. Um outro ponto chave é variar onde o índice k inicia e termina, foram testadas todas faixas para k.

Devido a natureza do problema, foi necessário modificar o método de Kadane para trabalhar com números negativos.

[1] `https://github.com/duraes-antonio/TPA_trab3`

[2] TAKAOKA, Tadao. Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication. Electronic Notes In Theoretical Computer Science, [s.l.], v. 61, p.191-200, jan. 2002. Elsevier BV. http://dx.doi.org/10.1016/s1571-0661(04)00313-5.

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>
#define MAX_N 100

using namespace std;

int n, max_soma, temp_soma, temp_vet[MAX_N], mat[MAX_N][MAX_N];

int kadane(int ind_ini_linha, int ind_fim_linha) {
    int max_atual = 0, max_geral = 0;

    for (int i = ind_ini_linha; i < ind_fim_linha; ++i) {
        max_atual = max_atual + temp_vet[i];
        if (max_atual < 0) max_atual = 0;
        if (max_atual > max_geral) max_geral = max_atual;
    }
    return max_geral > 0 ? max_geral : *min_element(temp_vet, temp_vet + n);
}

int main() {
    while(cin >> n && n > 0) {
        max_soma = temp_soma = 0;

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) cin >> mat[i][j];
        }
        for (int col_ini = 0; col_ini < n; ++col_ini) {
            fill(temp_vet, temp_vet + MAX_N, 0);

            for (int col_fim = col_ini; col_fim < n; ++col_fim) {
                for (int ind_lin = 0; ind_lin < n; ++ind_lin) {
                    temp_vet[ind_lin] += mat[ind_lin][col_fim];
                }
                temp_soma = kadane(0, n);
                if (temp_soma > max_soma) max_soma = temp_soma;
            }
        }
        cout << max_soma << endl;
    }
    return 0;
}
```

Código Fonte .1: Solução do problema #108 - Maximum Sum

2.1.2 #10684 – The jackpot

A ideia principal da solução é ler cada número de entrada e somá-lo em um acumulador, se o acumulador conter maior valor do que o máximo já registrado, o máximo é atualizado, senão, se o acumulador for negativo, então, ele é zerado.

```
1  #include <iostream>
2
3  using namespace std;
4
5  #define MAX_N   10000
6  #define MSG_SUC "The maximum winning streak is %d.\n"
7  #define MSG_FAL "Losing streak.\n"
8
9  int main() {
10     int acumulador, max_num, n_entrada, numeros[MAX_N];
11
12     while(cin >> n_entrada && n_entrada > 0) {
13         acumulador = max_num = -1;
14
15         for (int i = 0; i < n_entrada; ++i) {
16             cin >> numeros[i];
17             acumulador += numeros[i];
18
19             if(acumulador > max_num) max_num = acumulador;
20             else if(acumulador < 0) acumulador = 0;
21         }
22         (max_num > 0) ? printf(MSG_SUC, max_num) : printf(MSG_FAL);
23     }
24     return 0;
25 }
```

Código Fonte .2: Solução do problema #10684 – The jackpot

2.2 Algoritmos Gulosos

2.2.1 #12405 - Scarecrow

A essência dessa solução está em iterar sobre cada caractere de entrada até encontrar um ponto final (para o exercício, um solo com plantação). Como cada espantalho protege uma área linear de tamanho 3, após encontrar o ponto na posição **i** [linha 15], avança-se a iteração para o caractere de índice i + 3 e incrementa-se o número de espantalhos necessários para proteger a plantação [linha 16 e 17].

```cpp
#include <iostream>
#include <string>

using namespace std;

int main() {
    int n_testes, n_char, n_espants = 0;
    string entrada;
    cin >> n_testes;

    for (int n_caso = 0; n_caso < n_testes; ++n_caso, n_espants = 0) {
        cin >> n_char >> entrada;

        for (int i = 0; i < n_char; ++i) {
            if (entrada[i] == '.') {
                i += 2;
                ++n_espants;
            }
        }
        printf("Case %d: %d\n", n_caso + 1, n_espants);
    }
    return 0;
}
```

Código Fonte .3: Solução do problema #12405 - Scarecrow

2.2.2 #11100 – The Trip, 2007

Para este exercicio, sabemos que um pacote de tamanho normal não pode ser incluído no mesmo pacote de tamanho grande; portanto, o número de pacotes grandes necessários é o valor máximo do mesmo pacote de modelo na sequência original. A descrição do problema diz que qualquer tipo de saída pode ser satisfeita, logo, nossa idéia é colocar n pacotes em ordem. Depois disso, as primeiras respostas no array pacotes, após a classificação, são divididos respectivamente em pacotes maiores diferentes.

Por exemplo: 1 1 1 2 2 2 3 3 3 3, o número de pacotes necessários para 4 pacotes é 3 3 2 2. Primeiro pacote: 1 1 1 2 2 2 3 3 3 saída três das linhas 1 2 3 O segundo pacote: 1 1 1 2 2 2 3 3 3 três linhas de linhas de saída 1 2 3

Nota: na descrição requer que uma linha em branco seja impressa entre duas amostras. Nste caso, usamos uma variável de flag para marcar a primeira amostra sem gerar uma linha em branco, seguida por output uma linha em branco antes da saída.

```cpp
#include <iostream>
#include <cstdio>

using namespace std;

int n, resposta, flag, pacotes[10010], pacoteMaior[10010];

void saida(){
    int i, j, tmp;
    if (!flag) printf("\n");
    printf("%d\n", resposta);

    for (i = 0; i < resposta; i++) {
        printf("%d", pacotes[i]);
        for (j = 1; j < pacoteMaior[i]; j++)
            printf(" %d", pacotes[i + resposta * j]);
        printf("\n");
    }
}
void resolver() {
    int i, pos = 0, sum = 1, tmp;
    sort(pacotes, pacotes + n);
    resposta = 0;
    for (i = 1; i < n; i++) {
        if (pacotes[i] != pacotes[i - 1]) {
            if (resposta < sum) resposta = sum;
            sum = 1;
        }
        else sum++;
    }
    if (resposta < sum) resposta = sum;
    for (i = 0, tmp = n / resposta; i < resposta; i++) pacoteMaior[i] = tmp;
    for (i = 0, tmp = n % resposta; i < tmp; i++) pacoteMaior[i]++;
}

int main() {
    flag = 1;
    while (scanf("%d%*c", &n) && n) {
        for (int i = 0; i < n; i++) scanf("%d", &pacotes[i]);
        resolver();
        saida();
        if (flag) flag = 0;
    }
    return 0;
}
```

Código Fonte .4: Solução do problema #11100 – The Trip, 2007

2.3 Algoritmos em Grafos

2.3.1 #00280 – Vertex

A solução tem como base o uso de uma matriz de acessibilidade, em que cada aresta é representada pelo par linha e coluna ou número do vértice A e B. Após preencher a matriz e marcar os vértices relacionados diretamente [linha 23 a 25], realiza-se uma busca em profundidade para cada vértice a ser verificado [linha 31].

O método de busca em profundidade é responsável por verificar recursivamente quais vértices são acessíveis a partir do nó atual [linha 9 a 16].

Presume-se que todos vértices são inacessíveis para o vértice atual [linha 32] e para cada vértice acessível, decrementa-se o contador de inacessíveis [linha 34]. Por fim, a quantidade e os vértices propriamente ditos são impressos [linha 36].

```cpp
#include <iostream>
#define MAX_N 101

using namespace std;

int mat_acesseb[MAX_N][MAX_N];
bool acessaveis[MAX_N];

void busca_em_prof(int n_vertices, int vertice) {
    for (int i = 1; i <= n_vertices; ++i) {
        if (!acessaveis[i] and mat_acesseb[vertice][i]) {
            acessaveis[i] = true;
            busca_em_prof(n_vertices, i);
        }
    }
}

int main() {
    int n_vert, n_verif, vert_ini, vert, vert_inacess[MAX_N];

    while (cin >> n_vert, n_vert > 0) {
        fill(mat_acesseb[0], mat_acesseb[0] + MAX_N * MAX_N, 0);
        while (cin >> vert_ini, vert_ini)
            while (cin >> vert, vert)
                mat_acesseb[vert_ini][vert] = 1;
        cin >> n_verif;

        for (int i = 0; i < n_verif; ++i) {
            fill(acessaveis, acessaveis + MAX_N, false);
            cin >> vert;
            busca_em_prof(n_vert, vert);
            vert_inacess[vert] = n_vert;

            for (int j = 1; j <= n_vert; ++j) vert_inacess[vert] -= acessaveis[j];
            cout << vert_inacess[vert];
            for (int j = 1; j <= n_vert; ++j) if (!acessaveis[j]) cout << ' ' << j;
            cout << endl;
        }
    }
}
```

Código Fonte .5: Solução do problema #00280 – Vertex

2.3.2 #00459 – Graph Connectivity

A solução desse problema tem como ideia principal a utilização de dois arrays, um armazenando o grafo em si, a relação dos nós e o outro marcando se cada nó já foi verificado. É feito um loop que realiza a operação de busca em largura até que todos os nós sejam visitados. Como o busca em largura só termina quando todos os nós de um grafo do nó informado é visitado, temos a informação que o grafo que aquele nó passado foi todo percorrido e marcado como visitado, caso o loop que verifica se todos os nós foram visitados não se encerre, significa que ainda possui subgrafos naquele grafo, então é incrementado a quantidade de subgrafos e o busca em largura roda novamente. Isso ocorre até que todos os nós tenham sido visitados. no final temos a quantidade total de subgrafos que o grafo informado contém.

```
1  def buscLarg(no, visitadoF, grafoF):
2    visitadoF[no] = 1
3    for i in grafoF[no]:
4        if visitadoF[i] == 0: buscLarg(i, visitadoF, grafoF)
5
6  def main():
7      qtdExec = int(input().strip())
8      input()
9      while qtdExec > 0:
10         qtdExec = qtdExec - 1
11         nos = ord(input().strip()) - 64
12         grafo = [[] for x in range(nos+1)]
13         visitado = [0 for x in range(nos+1)]
14         subgrafo = 0
15         entrada = input().strip()
16         while entrada != '':
17             valorLetra1 = ord(entrada[0]) - 64
18             valorLetra2 = ord(entrada[1]) - 64
19             grafo[valorLetra1].append(valorLetra2)
20             grafo[valorLetra2].append(valorLetra1)
21             try:
22                 entrada = input().strip()
23             except(EOFError):
24                 break
25         for i in range(1,nos+1):
26             if visitado[i] == 0:
27                 buscLarg(i, visitado, grafo)
28                 subgrafo = subgrafo + 1
29         print(subgrafo)
30         if qtdExec > 0: print()
31 main()
```

Código Fonte .6: Solução do problema #00459 – Graph Connectivity

2.3.3 #00872 – Ordering

Por se tratar de um problema que apresenta uma sequência de saídas para um único grafo de entrada, decidiu-se criar uma função (**dfs_backtrack**) backtrack para percorrer o grafo e encontrar as combinações aceitas.

Se uma letra (vértice) **V** ainda não foi visitada e não possui vizinhos (vértices associados) visitados [linha 23], marca-se **V** como visitado [linha 24], e para cada vizinho de V, presume-se que o vizinho faz parte da solução e chama-se a função de busca passando a solução parcial atual [linha 25].

```
1  #include <iostream>
2  #include <algorithm>
3  #include <map>
4  #include <vector>
5  using namespace std;
6
7  vector<char> letras;
8  vector<string> respostas;
9  map<char, bool> visitados;
10 map<char, vector<char>> map_vizinhos;
11
12 bool sem_vizinhos_visitados(char vertice) {
13     for (char vert : map_vizinhos[vertice]) if(visitados[vert]) return false;
14     return true;
15 }
16 bool dfs_backtrack(const string &str_verts) {
17     bool resp_encontrada = str_verts.length() == letras.size();
18     if (resp_encontrada) {
19         respostas.push_back(str_verts);
20         return true;
21     }
22     for (char letra : letras) {
23         if(!visitados[letra] and sem_vizinhos_visitados(letra)) {
24             visitados[letra] = true;
25             resp_encontrada = dfs_backtrack(str_verts + letra) || resp_encontrada;
26             visitados[letra] = false;
27         }
28     }
29     return resp_encontrada;
30 }
```

Código Fonte .7: Função de busca em profundidade #00872 – Ordering

Na função principal, após a leitura de dados e esvazeamento das estruturas de dados, a lista de vértices é ordenada alfabeticamente [linha 15]. Para regra, adicione o vértice da direita na lista de vizinhos do vértice da esquerda [linha 18 e 19].

```
1  int main() {
2      int n_testes;
3      string linha;
4      char fim_linha;
5      cin >> n_testes;
6
7      while(n_testes-- and scanf("%c\n", &fim_linha) and getline(cin, linha)) {
8          letras.clear();
9          visitados.clear();
10         map_vizinhos.clear();
11         respostas.clear();
12
13         for (char simb : linha) if (simb != ' ') letras.push_back(simb);
14
15         sort(letras.begin(), letras.end());
16         getline(cin, linha);
17
18         for (int i = 0, tam = linha.size(); i < tam; i += 4) {
19             map_vizinhos[linha[i]].push_back(linha[i+2]);
20         }
21
22         if (dfs_backtrack("")) {
23             for (auto &resposta : respostas) {
24                 for (int j = 0, r_tam = resposta.length() - 1; j < r_tam; ++j) {
25                     cout << resposta[j] << ' ';
26                 }
27                 cout << resposta[resposta.length() - 1] << endl;
28             }
29         }
30         else cout << "NO\n";
31
32         if (n_testes) cout << endl;
33     }
34     return 0;
35 }
```

Código Fonte .8: Função principal #00872 – Ordering

2.3.4 #10034 – Freckles

A essência da solução está em percorrer os vértices ainda não visitados [linha 18], marcá-lo como visitado, armazenar sua distância até o vértice anterior em um acumulador [linha 19], calcular a distância até seus vizinhos [linha 20 a 21], selecionar o mais próximo [linha 23 e 24], elegê-lo como próximo a ser minimizado.

```cpp
#include <iostream>
#include <cmath>
using namespace std;

int n_casos, n_vert, vert_min, visitados[100];
pair<double, double> pts[100];
double x, y, minimo, soma_pesos, dists[100];

double calc_dist(pair<double, double> p1, pair<double, double> p2) {
    return sqrt(pow((p1.first - p2.first), 2) + pow((p1.second - p2.second), 2));
}

double dijkstra() {
    fill(dists, dists + n_vert, MAXFLOAT);
    fill(visitados, visitados + n_vert, 0);
    vert_min = soma_pesos = dists[0] = 0;

    while (!visitados[vert_min]++) {
        soma_pesos += dists[vert_min];
        for (int i = 0; i < n_vert; ++i)
            if (!visitados[i]) dists[i] = min(calc_dist(pts[vert_min], pts[i]), dists[i]);
        vert_min = 0, minimo = MAXFLOAT;
        for (int i = 0; i < n_vert; ++i)
            if (!visitados[i] and dists[i] < minimo) vert_min = i, minimo = dists[i];
    }
    return soma_pesos;
}

int main() {
    cin >> n_casos;

    while (n_casos--) {
        scanf("\n\n%d\n", &n_vert);
        for (int i = 0; i < n_vert; ++i) {
            cin >> x >> y;
            pts[i] = make_pair(x, y);
        }
        printf(n_casos ? "%.2f\n\n" : "%.2f\n", dijkstra());
    }
    return 0;
}
```

Código Fonte .9: Solução do problema #10034 – Freckles