Estruturas de Dados Avançadas

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

Instituto de Computação - UNICAMP - 2018

Sumário

Introdução

- Um grafo $G(V, E)$ consiste em uma abstração matemática de um conjunto de objetos e da interligação entre esses objetos.
- Os objetos são representados por um conjunto de vértices V, enquanto os elos de ligação entre dois objetos são representados pelo conjunto de arestas E.
- Existem diversas estruturas de dados para representar grafos computacionalmente. Elas são mais ou menos eficientes dependendo das operações e consultas que serão realizadas.
- A seguir são descritas três estruturas de dados para grafos:

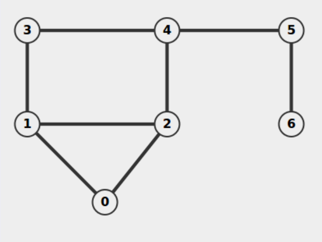
1. Matriz de adjacências
2. Lista de adjacências
3. Lista de arestas

Matriz de adjacências

- Quando o número de vértices de um grafo é conhecido, é possível construir uma matriz de conectividade a partir de um vetor bidimensional (`int AdjMat[V][V]`), denominado matriz de adjacências
- Para grafos não-ponderados, o valor de `AdjMat[i][j]` terá um valor não-nulo (geralmente igual a 1) se existe uma aresta entre os vértices i e j, e valor 0 caso contrário.
- Para grafos ponderados, onde cada aresta ij possui um peso w_{ij}, tem-se que `AdjMat[i][j]` terá um valor igual a w_{ij} se existe uma aresta entre os vértices i e j, e valor 0 caso contrário.

Grafos

Exemplo



Adjacency matrix							
	0	**1**	**2**	**3**	**4**	**5**	**6**
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Matriz de adjacências

- A matriz de adjacência é uma boa estrutura quando se deseja obter a conectividade de um grafo pequeno e denso.
- Essa estrutura não é recomendada para grafos grandes e esparsos devido ao elevado uso de memória $O(|V|^2)$. Na prática, o uso dessa estrutura é inviável para grafos com $|V| > 1000$.
- Outra desvantagem dessa estrutura está no fato de requerer tempo $O(|V|)$ para enumerar as arestas incidentes a um nó.

Lista de adjacências

- Uma lista de adjacências normalmente toma a forma de um vetor de vetores de pares. A declaração de uma lista de adjacências utilizando a C++ STL pode ser feita como:

```
typedef pair<int, int> ii; // data type shortcut
typedef vector<ii> vii;    // data type shortcut
vector<vii> AdjList;
```

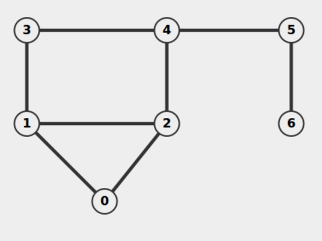
- Em uma lista de adjacências, cada elemento do vetor corresponde a um vértice associado ao seu conjunto de arestas incidentes, sendo este conjunto representado por um vetor de pares.

Lista de adjacências

- Cada par armazenado no vetor armazena duas informações de uma aresta: o nó adjacente e o peso.
- Uma lista de adjacências consegue enumerar eficientemente a adjacência de um nó.
- Essa é a estrutura recomendada para a maioria dos problemas de programação que utilizam grafos.

Grafos

Exemplo



Adjacency list			
0 :	1	2	
1 :	0	2	3
2 :	1	4	0
3 :	1	4	
4 :	3	2	5
5 :	4	6	
6 :	5		

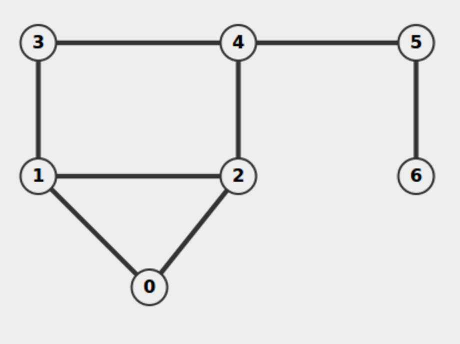
Lista de arestas

- Uma lista de arestas normalmente toma a forma de um vetor de triplas. A declaração de um lista de arestas utilizando a C++ STL pode ser feita como:

```
typedef pair<int, int> ii; // data type shortcut
vector< pair<int, ii> > EdgeList;
```

- Em uma lista de arestas, cada elemento corresponde a uma aresta do grafo, normalmente ordenado sob algum critério.
- Trata-se de uma estrutura de dados útil para certos problemas, como o algoritmo de Kruskal que determina árvores geradoras de custo mínimo.
- Não é uma boa representação de grafos para problemas que envolvem a enumeração das arestas incidentes a um certo vértice.

Exemplo



Edge list		
0 :	0	1
1 :	1	2
2 :	3	1
3 :	3	4
4 :	4	2
5 :	4	5
6 :	5	6
7 :	2	0

Estruturas de dados para grafos. Continua...

```
// Try this input for Adjacency Matrix/List/EdgeList
// Adj Matrix
//   for each line: |V| entries, 0 or the weight
// Adj List
//   for each line: num neighbors, list of neighbors + weight pairs
// Edge List
//   for each line: a-b of edge(a,b) and weight
/*
6
  0  10   0   0 100   0
 10   0   7   0   8   0
  0   7   0   9   0   0
  0   0   9   0  20   5
100   8   0  20   0   0
  0   0   0   5   0   0
6
2 2 10 5 100
3 1 10 3 7 5 8
2 2 7 4 9
3 3 9 5 20 6 5
3 1 100 2 8 4 20
1 4 5
7
1 2 10
1 5 100
2 3 7
2 5 8
3 4 9
4 5 20
4 6 5
*/
```

Estruturas de dados para grafos. Continua...

```
#include <cstdio>
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;

int main() {
  int V, E, total_neighbors, id, weight, a, b;
  int AdjMat[100][100];
  vector<vii> AdjList;
  priority_queue< pair<int, ii> > EdgeList;   // one way to store Edge List

  scanf("%d", &V);                            // we must know this size first!
                        // remember that if V is > 100, try NOT to use AdjMat!
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      scanf("%d", &AdjMat[i][j]);

  printf("Neighbors of vertex 0:\n");
  for (int j = 0; j < V; j++)                                    // O(|V|)
    if (AdjMat[0][j])
      printf("Edge 0-%d (weight = %d)\n", j, AdjMat[0][j]);
```

Estruturas de dados para grafos.

```
  scanf("%d", &V);
  AdjList.assign(V, vii()); // quick way to initialize AdjList with V entries of vii
  for (int i = 0; i < V; i++) {
    scanf("%d", &total_neighbors);
    for (int j = 0; j < total_neighbors; j++) {
      scanf("%d %d", &id, &weight);
      AdjList[i].push_back(ii(id - 1, weight));     // some index adjustment
    }
  }

  printf("Neighbors of vertex 0:\n");
  for (vii::iterator j = AdjList[0].begin(); j != AdjList[0].end(); j++)
    // AdjList[0] contains the required information
    // O(k), where k is the number of neighbors
    printf("Edge 0-%d (weight = %d)\n", j->first, j->second);

  scanf("%d", &E);
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &weight);
    EdgeList.push(make_pair(-weight, ii(a, b))); // trick to reverse sort order
  }

  // edges sorted by weight (smallest->largest)
  for (int i = 0; i < E; i++) {
    pair<int, ii> edge = EdgeList.top(); EdgeList.pop();
    // negate the weight again
    printf("weight: %d (%d-%d)\n", -edge.first, edge.second.first, edge.second.second);
  }

  return 0;
}
```

Estruturas de dados para grafos.

Saida:

```
Neighbors of vertex 0:
Edge 0—1 (weight = 10)
Edge 0—4 (weight = 100)
Neighbors of vertex 0:
Edge 0—1 (weight = 10)
Edge 0—4 (weight = 100)
weight: 5 (4—6)
weight: 7 (2—3)
weight: 8 (2—5)
weight: 9 (3—4)
weight: 10 (1—2)
weight: 20 (4—5)
weight: 100 (1—5)
```

Algoritmo Union-Find

- A estrutura de dados Union-Find para conjuntos disjuntos consegue em tempo quase-constante $O(\alpha(n))$, onde $\alpha(n)$ é a função inversa de Ackermann:

1. determinar a qual conjunto um elemento pertence.
2. verificar se dois elementos pertencem ao mesmo conjunto.
3. unir dois conjuntos disjuntos.

- Essas operações não são realizadas de modo eficiente pela C++ STL, pois gastam um tempo $O(n)$.
- Dentre outras aplicações, essa estrutura pode ser utilizada para determinar uma árvore geradora de um grafo não-orientado.

Algoritmo Union-Find

- A ideia principal da estrutura Union-Find consiste em escolher um elemento representante (pai) para representar cada conjunto.
- Para atingir esse objetivo, a estrutura Union-Find cria uma estrutura tal que os conjuntos são representados por uma floresta.
- Cada árvore corresponde a um conjunto e sua raiz determina o representante do conjunto. Portanto, o representante de um elemento pode ser obtido simplesmente seguindo a cadeia de nós pais até a raiz da árvore.

Algoritmo Union-Find

- Na estrutura Union-Find são armazenados para cada elemento `i` o índice do elemento pai `p[i]` e a altura `rank[i]` (na verdade, um limitante superior da altura) da árvore enraizada em `i`.
- A estrutura de dados é inicializada tal que cada elemento é seu próprio conjunto, logo `p[i] = i` e `rank[i] = 0`.
- Na união de dois conjuntos, o representante de um conjunto passa a ser o pai do representante do segundo conjunto. Isso é equivalente a mesclar duas árvores da estrutura Union-Find.

Conjuntos Disjuntos

Algoritmo Union-Find

- Para aumentar a eficiência, a estrutura Union-Find utiliza a heurística de "união por ranque" (`rank`). Essa heurística utiliza a informação sobre a altura da árvore de modo que o conjunto com o maior valor de `rank` seja o novo pai do conjunto de menor valor de `rank`, minimizando desse modo o valor de `rank` da árvore resultante.
- Se os valores de `rank` dos dois conjuntos forem iguais, um dos conjuntos é escolhido arbitrariamente para ser o novo pai e o valor de `rank` da árvore resultante é incrementado.

Conjuntos Disjuntos

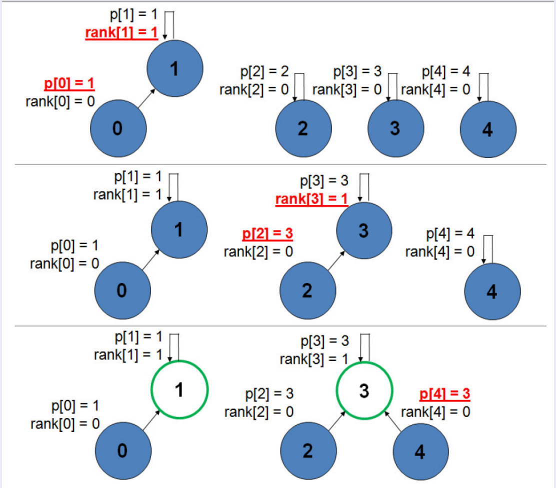
Algoritmo Union-Find

- Outra técnica que aumenta significativamente a eficiência das operações sobre a estrutura Union-Find é denominada compressão de caminhos.
- Sempre que um representante (raiz) de um conjunto é buscado seguindo-se a cadeia de pais de um determinado elemento, é possível atualizar o pai de cada elemento da cadeia para apontar diretamente para o elemento representante. Desse modo, chamadas subsequentes de `findSet(i)` nos elementos cujos pais foram atualizados vão encontrar seus representantes em apenas um passo recursivo.

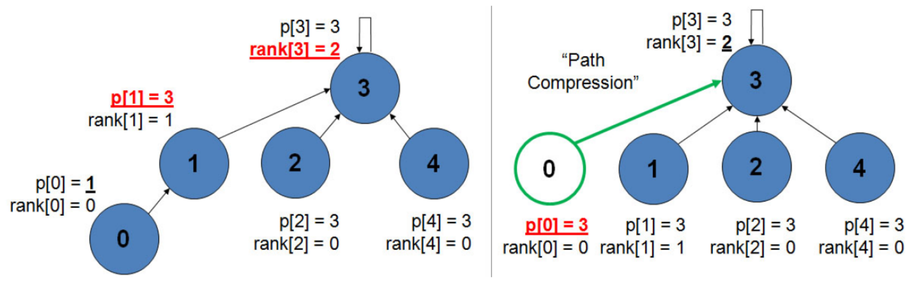
Algoritmo Union-Find

- Apesar da mudança na estrutura da árvore com a técnica de compressão de caminhos, a constituição dos conjuntos se mantém.
- Note que devido à compressão de caminhos, os valores de `rank` não representam exatamente a altura da árvore, mas continuam sendo limitantes superiores válidos.

Exemplo


p[1] = 1
rank[1] = 1
p[0] = 1
rank[0] = 0
1
0
p[2] = 2
rank[2] = 0
p[3] = 3
rank[3] = 0
p[4] = 4
rank[4] = 0
2
3
4
p[1] = 1
rank[1] = 1
p[0] = 1
rank[0] = 0
1
0
p[3] = 3
rank[3] = 1
p[2] = 3
rank[2] = 0
3
2
p[4] = 4
rank[4] = 0
4
p[1] = 1
rank[1] = 1
p[0] = 1
rank[0] = 0
1
0
p[3] = 3
rank[3] = 1
p[2] = 3
rank[2] = 0
3
p[4] = 3
rank[4] = 0
2
4

Exemplo


p[3] = 3
rank[3] = 2
p[1] = 3
rank[1] = 1
3
1
2
4
0
p[0] = 1
rank[0] = 0
p[2] = 3
rank[2] = 0
p[4] = 3
rank[4] = 0
p[3] = 3
rank[3] = 2
"Path Compression"
3
0
1
2
4
p[0] = 3
rank[0] = 0
p[1] = 3
rank[1] = 1
p[2] = 3
rank[2] = 0
p[4] = 3
rank[4] = 0

Estruturas de dados Union-Find. Continua...

```
#include <cstdio>
#include <vector>
using namespace std;

typedef vector<int> vi;

// Union-Find Disjoint Sets Library written in OOP manner, using both path compression and union by rank heuristics
class UnionFind {                                          // OOP style
private:
  vi p, rank, setSize;                       // remember: vi is vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```

Conjuntos Disjuntos

Estruturas de dados Union-Find.

```
int main() {
  printf("Assume that there are 5 disjoint sets initially\n");
  UnionFind UF(5); // create 5 disjoint sets
  printf("%d\n", UF.numDisjointSets()); // 5
  UF.unionSet(0, 1);
  printf("%d\n", UF.numDisjointSets()); // 4
  UF.unionSet(2, 3);
  printf("%d\n", UF.numDisjointSets()); // 3
  UF.unionSet(4, 3);
  printf("%d\n", UF.numDisjointSets()); // 2
  printf("isSameSet(0, 3) = %d\n", UF.isSameSet(0, 3)); // will return 0 (false)
  printf("isSameSet(4, 3) = %d\n", UF.isSameSet(4, 3)); // will return 1 (true)
  for (int i = 0; i < 5; i++) // findSet will return 1 for {0, 1} and 3 for {2, 3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i, UF.findSet(i), i, UF.sizeOfSet(i));
  UF.unionSet(0, 3);
  printf("%d\n", UF.numDisjointSets()); // 1
  for (int i = 0; i < 5; i++) // findSet will return 3 for {0, 1, 2, 3, 4}
    printf("findSet(%d) = %d, sizeOfSet(%d) = %d\n", i, UF.findSet(i), i, UF.sizeOfSet(i));
  return 0;
}
```

Estruturas de dados Union-Find.

Saida:

```
Assume that there are 5 disjoint sets initially
5
4
3
2
isSameSet(0, 3) = 0
isSameSet(4, 3) = 1
findSet(0) = 1, sizeOfSet(0) = 2
findSet(1) = 1, sizeOfSet(1) = 2
findSet(2) = 3, sizeOfSet(2) = 3
findSet(3) = 3, sizeOfSet(3) = 3
findSet(4) = 3, sizeOfSet(4) = 3
1
findSet(0) = 3, sizeOfSet(0) = 5
findSet(1) = 3, sizeOfSet(1) = 5
findSet(2) = 3, sizeOfSet(2) = 5
findSet(3) = 3, sizeOfSet(3) = 5
findSet(4) = 3, sizeOfSet(4) = 5
```

Árvore de Segmentos

Introdução

- Uma árvore de segmentos ("segment tree") é uma estrutura de dados que pode responder de modo eficiente a uma consulta de intervalo.
- Uma consulta de intervalo consiste, por exemplo, em encontrar o índice do menor elemento em um intervalo $[i..j]$ de um vetor. Essa consulta de intervalo, em particular, é conhecida como consulta de mínimo em um intervalo ("Minimum Range Query" – RMQ).

Array	Values	18	17	13	19	15	11	20
A	Indices	0	1	2	3	4	5	6

Introdução

- Há diversas maneiras de resolver um RMQ: um algoritmo trivial consiste em iterar sobre os elementos de i até j e reportar o índice com o menor valor. Isso implica um tempo $O(n)$ por consulta, o que pode ser inviável quando muitas consultas são realizadas e o valor de n é elevado.
- Uma maneira de resolver o RMQ em $O(\log n)$, é utilizando uma árvore de segmentos, que organiza os dados em uma árvore binária.

Introdução

- Uma árvore de segmentos pode ser representada com um vetor (`vector<int> st`), assim como os heaps.
- O índice 1 representa a raiz (índice 0 é ignorado) e os filhos esquerdo e direito de um índice p possuem índices $2p$ e $2p+1$, respectivamente.
- Um valor `st[p]` corresponde à solução do RMQ para a subárvore (segmento) associada ao índice p.

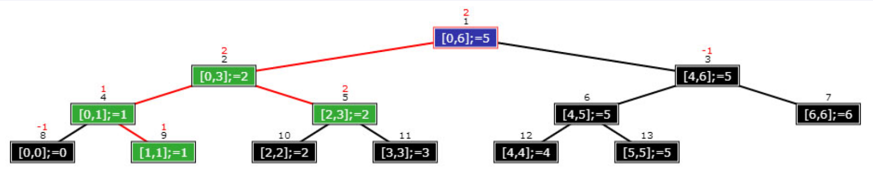
Árvore de Segmentos

Introdução

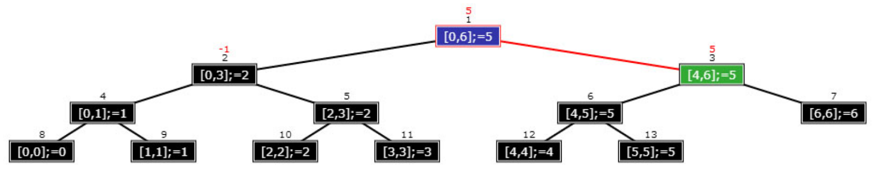
- A raiz da árvore de segmentos representa o segmento $[0, n-1]$.
- Um segmento $[L, R]$ armazenado no índice p é particionado em dois subsegmentos $[L, (L+R)/2]$ e $[(L+R)/2+1, R]$ nos filhos esquerdo e direito de p, respectivamente.
- Desse modo, a árvore de segmentos é construída recursivamente, comparando o menor valor do segmento à esquerda e à direita para atualizar o valor de `st[p]` do segmento pai.
- O caso base ocorre na folha da árvore, onde $L = R$ e `st[p] = L`.
- A árvore possui no máximo $2n$ nós e sua construção requer tempo $O(n)$.

Exemplo

Buscando $RMQ[1,3]$



Buscando $RMQ[4,6]$



Consulta de Mínimo em um Intervalo. Continua...

```
#include <cmath>
#include <cstdio>
#include <vector>
using namespace std;

typedef vector<int> vi;

class SegmentTree {          // the segment tree is stored like a heap array
private: vi st, A;             // recall that vi is: typedef vector<int> vi;
  int n;
  int left (int p) { return p << 1; }      // same as binary heap operations
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {                          // O(n)
    if (L == R)                              // as L == R, either one is fine
      st[p] = L;                                        // store the index
    else {                                 // recursively compute the values
      build(left(p) , L               , (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R          );
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }
```

Consulta de Mínimo em um Intervalo. Continua...

```
  int rmq(int p, int L, int R, int i, int j) {                    // O(log n)
    if (i >  R || j <  L) return -1; // current segment outside query range
    if (L >= i && R <= j) return st[p];                  // inside query range

     // compute the min position in the left and right part of the interval
    int p1 = rmq(left(p) , L                 , (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R          , i, j);

    if (p1 == -1) return p2;   // if we try to access segment outside query
    if (p2 == -1) return p1;                               // same as above
    return (A[p1] <= A[p2]) ? p1 : p2; }            // as as in build routine

public:
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();                      // copy content for local usage
    st.assign(4 * n, 0);              // create large enough vector of zeroes
    build(1, 0, n - 1);                                     // recursive build
  }

  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }    // overloading

};
```

Consulta de Mínimo em um Intervalo.

```
int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };         // the original array
  vi A(arr, arr + 7);                               // copy the contents to a vector
  SegmentTree st(A);

  printf("              idx    0, 1, 2, 3, 4,  5, 6\n");
  printf("              A is {18,17,13,19,15, 11,20}\n");
  printf("RMQ(1, 3) = %d\n", st.rmq(1, 3));             // answer = index 2
  printf("RMQ(4, 6) = %d\n", st.rmq(4, 6));             // answer = index 5
  printf("RMQ(3, 4) = %d\n", st.rmq(3, 4));             // answer = index 4
  printf("RMQ(0, 0) = %d\n", st.rmq(0, 0));             // answer = index 0
  printf("RMQ(0, 1) = %d\n", st.rmq(0, 1));             // answer = index 1
  printf("RMQ(0, 6) = %d\n", st.rmq(0, 6));             // answer = index 5

  return 0;
}
```

Consulta de Mínimo em um Intervalo.

Saida:

```
                idx     0, 1, 2, 3, 4,  5, 6
                A is {18,17,13,19,15, 11,20}
RMQ(1, 3) = 2
RMQ(4, 6) = 5
RMQ(3, 4) = 4
RMQ(0, 0) = 0
RMQ(0, 1) = 1
RMQ(0, 6) = 5
```

Referências

1. S. Halim e F. Halim. Competitive Programming 2, Second Edition Lulu (www.lulu.com), 2011. (IMECC – 005.1 H139c)
2. S. S. Skiena, M. A. Revilla. Programming Challenges: The Programming Contest Training Manual, Springer, 2003.
3. T.H. Cormen, C.E. Leiserson, R.L.Rivest e C. Stein. Introduction to Algorithms. 2nd Edition, McGraw-Hill, 2001. (no. chamada IMECC – 005.133 Ar64j 3.ed.)
4. U. Manber. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989. (no. chamada IMECC – 005.133 Ec53t 2.ed.)