Fluxos em Redes

prof. Fábio Luiz Usberti

MC521 - Desafios de Programação I

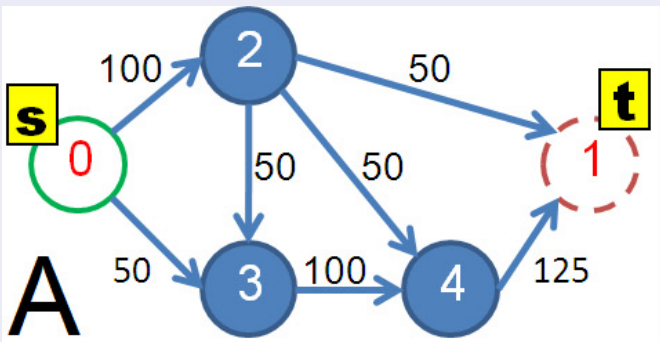
Instituto de Computação - UNICAMP - 2018

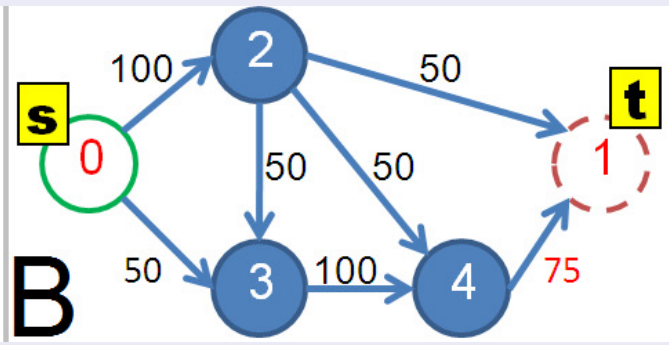
Sumário

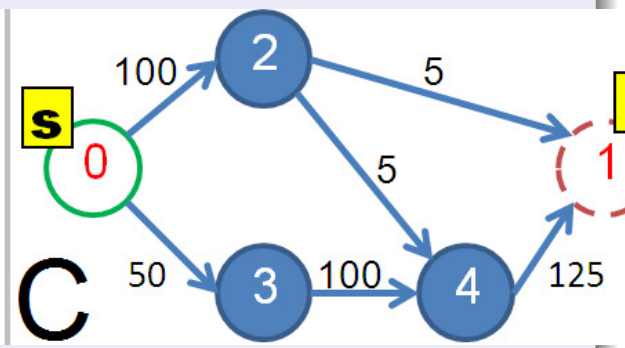
Introdução

- Problema de Fluxo Máximo ("Maximum-Flow Problem" – Max-Flow): Dado um grafo orientado G, com capacidades não-negativas w_{ij} nos arcos, e dois vértices s (origem) e t (destino). Qual o fluxo máximo que pode ser enviado de s para t respeitando as capacidades dos arcos?

Exemplos


s
0
100
2
50
t
1
50
50
50
3
100
4
125
A


s
0
100
2
50
t
1
50
50
50
3
100
4
75
B


s
0
100
2
5
1
5
50
3
100
4
125
C

Ford-Fulkerson

- O algoritmo de Ford-Fulkerson, para o problema Max-Flow, foi desenvolvido por Lester R. Ford e Delbert R. Fulkerson em 1956.
- Trata-se de um algoritmo onde em cada iteração busca-se por um caminho aumentante p, ou seja, um caminho de s para t que passa somente por arcos com capacidade positiva no grafo residual.
- O grafo residual é inicializado igual ao grafo original, mas assim que um fluxo escoa por um caminho, as capacidades residuais dos arcos pertencentes ao caminho são atualizadas.

Ford-Fulkerson

- Uma vez que um caminho aumentante p é encontrado, verifica-se a capacidade mínima f (gargalo) dos arcos que pertencem ao caminho.
- Em seguida, duas operações são realizadas:

1. Para cada arco de "avanço" (i, j), subtrair de sua capacidade residual res_{ij} o montante f ($res_{ij} = res_{ij} - f$).
2. Para cada arco de "retrocesso" (j, i), somar à sua capacidade residual res_{ji} o montante f ($res_{ji} = res_{ji} + f$).

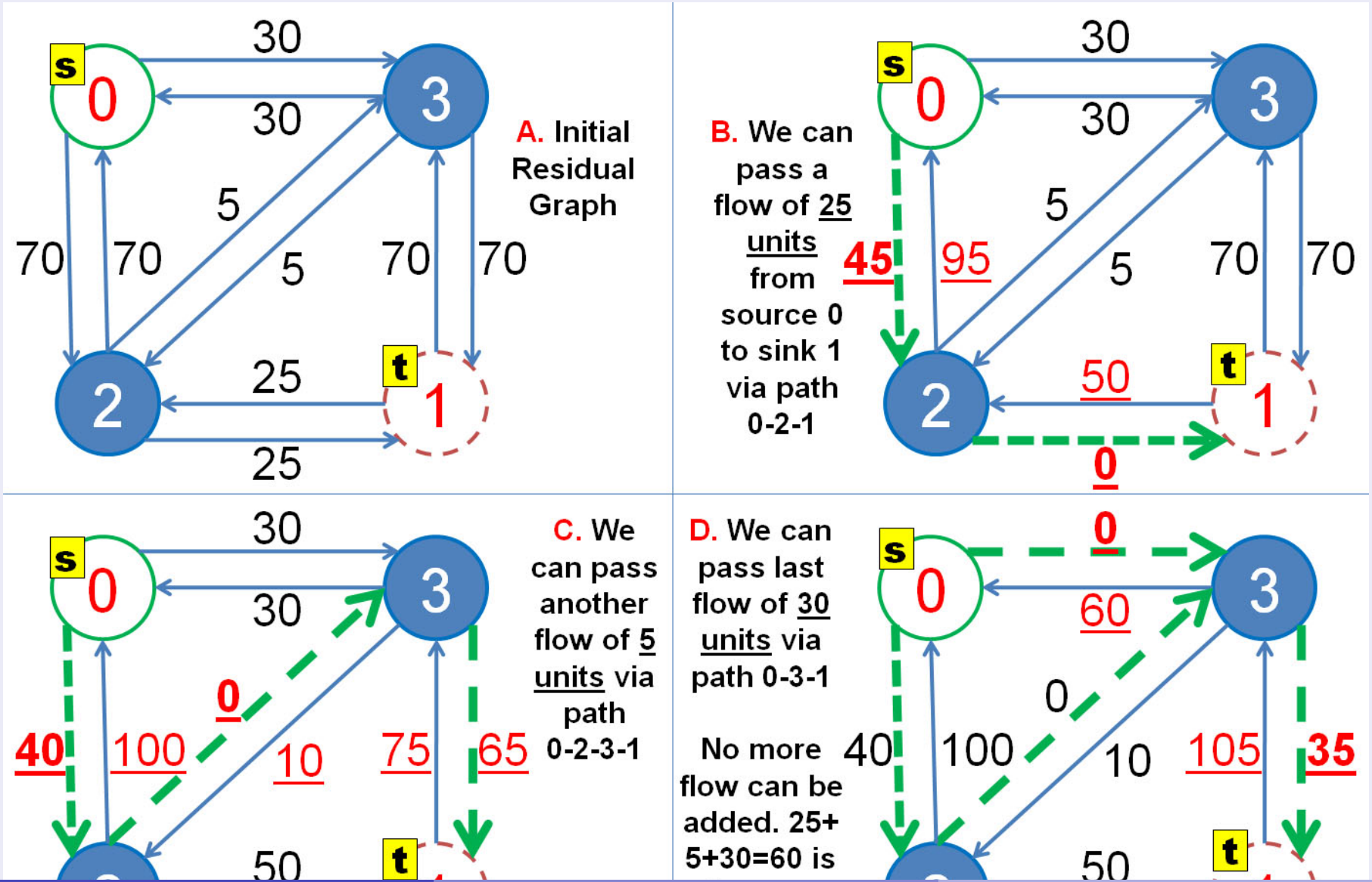
Ford-Fulkerson: Pseudo-código

```
1. setup directed residual graph with edge capacity = original graph weights
2. mf = 0
       // this is an iterative algorithm, mf stands for max_flow
3.     while (there exists an augmenting path p from s to t) {
3.1.       find f, the minimum edge weight along the path p
3.2.       decrease capacity of forward edges (e.g. i -> j) along path p by f
3.3.       increase capacity of backward edges (e.g. j -> i) along path p by f
3.4.       mf += f   // we can send a flow of size f from s to t, increase mf
       }
4.     output mf   // this is the max flow value
```

Ford-Fulkerson

- O motivo para diminuir a capacidade dos arcos de avanço é intuitivo, uma vez que estamos utilizando recursos do arco para escoar o fluxo.
- O motivo para aumentar a capacidade dos arcos de retrocesso não é tão evidente, mas trata-se de um passo importante para a corretude do algoritmo. Esse incremento da capacidade permite que os fluxos de futuras iterações possam repôr, ainda que parcialmente, as capacidades de arcos de avanço indevidamente utilizadas por fluxos de iterações anteriores.
- O algoritmo repete o processo até que não existam mais caminhos aumentantes de s para t, o que pelo teorema Max-Flow Min-Cut (vide Cormen et al.) implica que o fluxo é máximo.

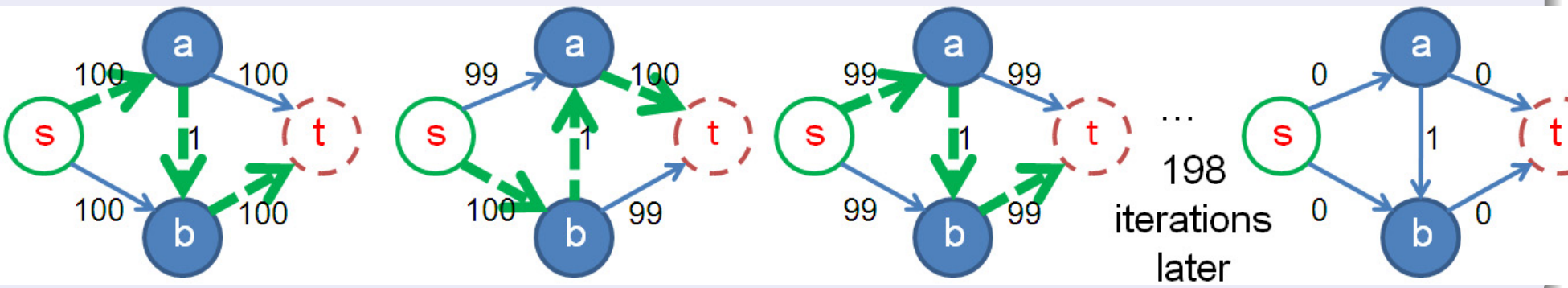
Ford-Fulkerson



Ford-Fulkerson

- Existem diferentes maneiras para encontrar um caminho aumentante $s - t$.
- O algoritmo de Ford-Fulkerson utiliza busca em profundidade (DFS), o que torna sua complexidade $O(f^* \times |E|)$, no pior caso, onde f^* é o valor do fluxo máximo.
- Essa complexidade decorre do fato de que, para certos grafos, cada iteração do algoritmo incrementa o fluxo de s para t em uma unidade.
- O tempo computacional do algoritmo de Ford-Fulkerson pode variar muito, uma vez que depende do valor do fluxo máximo f^*.
- Uma vez que em competições os casos de testes procuram explorar condições extremas das instâncias, a imprevisibilidade no tempo computacional do algoritmo torna-o uma solução pouco recomendável.

Ford-Fulkerson



Edmonds-Karp

- Um algoritmo de tempo polinomial $O(|V||E|^2)$ para o Max-Flow foi proposto por Jack Edmonds e Richard Karp em 1972.
- Esse algoritmo consiste em uma implementação aperfeiçoada do algoritmo de Ford-Fulkerson, que utiliza busca em largura (BFS) para encontrar o caminho aumentante mais curto (em número de arcos) entre s e t.
- É possível demonstrar que o algoritmo termina após $O(|V||E|)$ buscas de caminhos aumentantes (vide Cormen et al.).

Edmonds-Karp: Código-fonte (continua)

```
#define MAX_V 40 // enough for sample graph in Figure 4.24/4.25/4.26/UVa 259
#define INF 1000000000

int res[MAX_V][MAX_V], mf, f, s, t;                          // global variables
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {      // traverse BFS spanning tree from s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       // update
}

int main() {
  int V, k, vertex, weight;

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  AdjList.assign(V, vi());
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
      AdjList[i].push_back(vertex);
    }
  }
```

Edmonds-Karp: Código-fonte

```
  mf = 0;
  while (1) {                          // main loop
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;              // we change vi dist to bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {  // we use AdjList here!
        int v = AdjList[u][j];
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }

  printf("%d\n", mf);                              // this is the max flow value

  return 0;
}
```

Modelagem de Problemas

- Em competições, resolver problemas de fluxos em redes envolve os seguintes passos:

1. Reconhecer que um problema é de fato um problema de fluxos em redes.
2. Construir corretamente o grafo de fluxos.
3. Executar o algoritmo para o grafo de fluxos.

Modelagem de Problemas

Exemplo (UVa 259 – Software Allocation):
Dados do problema:

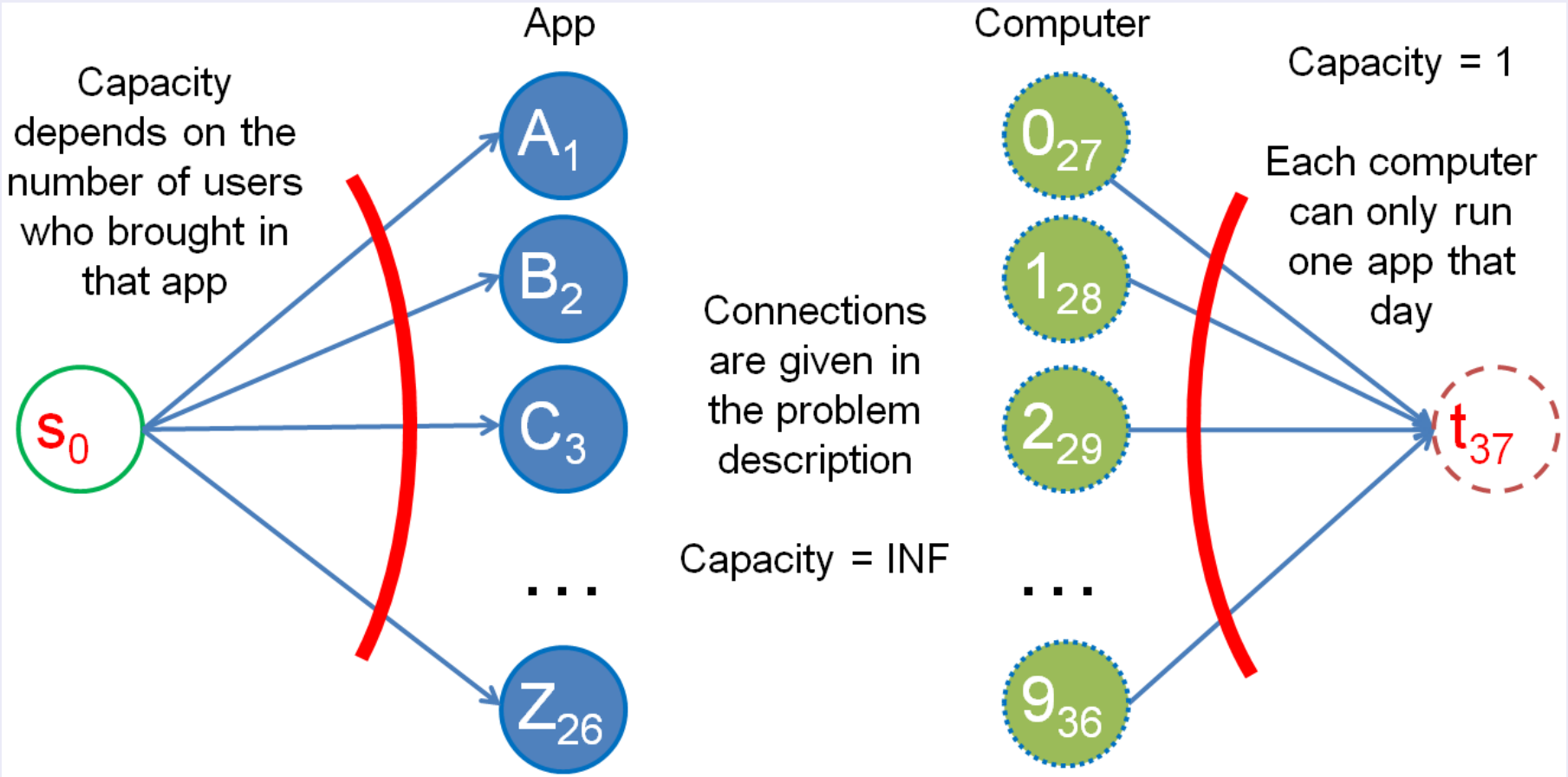
- 10 computadores, numerados de 0 a 9.
- 26 softwares, rotulados de A a Z.
- número de usuários (até 9) que precisam executar cada software.
- subconjunto de computadores onde cada software pode ser executado.
- cada computador pode executar um único software
- Problema: determinar se existe uma atribuição factível de softwares para os computadores?

Modelagem de Problemas

Exemplo (UVa 259 – Software Allocation):

- O grafo de resíduos é construído de modo que cada software e cada computador corresponde a um nó do grafo.
- Dois nós artificiais s e t são incluídos para serem a origem e o destino dos fluxos.
- Adiciona-se arcos do nó origem para cada software, cujas capacidades correspondem ao número de usuários que requerem execução do software.
- Adiciona-se arcos de cada computador para o nó destino, cujas capacidades são iguais a 1, dado que cada computador pode executar no máximo um software por vez.
- Quando um software i pode ser executado em um computador j, então adiciona-se o arco (i, j). A capacidade desse arco pode ser ∞ ou igual a 1 para este problema específico onde cada computador pode executar no máximo um software por vez.

Ford-Fulkerson



Modelagem de Problemas

Exemplo (UVa 259 – Software Allocation):

- Após a solução do Max-Flow, haverá um fluxo unitário de *i* para *j* quando o software *i* for atribuído para o computador *j*, .
- Todos os softwares estarão atribuídos se e somente se o valor do fluxo máximo for igual ao número requerido de execuções de software.

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.)