UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

SEGUNDO TRABALHO PRÁTICO DA DISCIPLINA DE PROJETO E ANÁLISE DE ALGORITMOS (BCC241)

Vinicius Gabriel Angelozzi Verona de Resende
&
Carlos Eduardo Gonzaga Romaniello de Souza

Professor: Dr. ANDERSON ALMEIDA FERREIRA

Relatório referente ao segundo trabalho prático da disciplina BCC241-11

Data: 06 de junho de 2022
Local: Ouro Preto – Minas Gerais – Brasil

Sumário

Resumo

Segundo trabalho prático da disciplina de Projeto e Análise de Algoritmos (BCC241)

Devido ao aumento constante de dados na internet ao redor do mundo, problemas que costumavam ser triviais começam a se tornar complexos, como por exemplo os problemas de otimização combinatorial que ficam mais difíceis conforme os dados aumentam. Por isso é necesário pesquisar e desenvolver métodos eficientes para resolvê-los. Com esse intuito, neste trabalho são analisados três técnicas de programação diferente para resolver um problema combinatorial clássico da literatura, o problema da mochila 0-1, ou *Knapsack problem*. Os algoritmos implementados utilizam abordagens de Programação Dinâmica Recursiva com memoização (DPR), Programação Dinâmica Iterativa (DPI) e a abordagem do *Branch-and-Bound*. Após análise de resultados obtidos, percebe-se que a abordagem *Branch* & *Bound* apresenta maior eficiência para grandes instâncias, isto levadno em consideração a solução inicial utilizada.

1 Introdução

O problema da mochila 0-1, também conhecido como Problema de Knapsack 0-1, é um problema de otimização combinatorial cujo enunciado é: dados números inteiro $I, W \geq 0$ e um conjunto $K = \{1, ..., I\}$, onde cada item $i \in K$ tem valor v_i e peso w_i, ambos positivos; encontre um subconjunto X de K tal que

$$\sum_{i \in X} v_i$$

é máximo atendendo a restrição sequinte:

$$\sum_{i \in X} w_i \leq W$$

Isto é, temos um problema de otimização cujo modelo é dado por 1.1 e 1.2 que consiste em escolher k itens de forma que a soma dos valores dos k itens seja máxima e cuja soma dos pesos dos itens não ultrapasse W.

$$max: \sum_{i=1}^{I} (v_i \times y_i) \tag{1.1}$$

sujeito a:

$$\begin{gathered} \sum_{i=1}^{I} (w_i \times y_i) \leq W \\ y_i \in \{0, 1\}, \forall i \in \{1, ..., I\} \end{gathered} \tag{1.2}$$

Este trabalho consiste em implementar e analisar três algoritmos na resolução do problema da mochila 0-1, cada um utilizando uma técnica diferente para a resolução deste. Dentre as técnicas temos *Programação Dinâmica com memoização (DPR)*, *Programação Dinâmica iterativa (DPI)* e Branch-and-Bound*(B&B)*. Para isso, os algoritmos foram desenvolvidos na linguagem de programação *C++* e cada um foi executado três vezes com instâncias de tamanhos dife rentes. Vale salientar que para cada tamanho, a instância é a mesma para todos os algoritmos.

Para recolher os dados a serem avaliados, foram criadas 3 instâncias diferentes na linguagem de programação *Julia*[1]. A primeira possui *30* itens com *50* unidades de capacidade, a segunda possui *30* itens com *100* unidades de capacidade e a terceira possui *30* itens com *10000* unidades de capacidade. Após todas as execuções, os dados foram

[1]Mais informações em: https://julialang.org/

coletados e armazenados em arquivos separados para que posteriormente fossem utilizados para a geração de gráficos na linguagem de programação *JavaScript*[2] com o intuito de facilitar a interpretação dos dados coletados.

Todos os arquivos gerados para a realização deste trabalho estão disponíveis no *GitHub*[3] e foram separados em pastas de acordo com o seu conteúdo. O diretório *src* possui as implementações dos algoritmos mencionados e o algoritmo gerador das instâncias. O diretório *data* possui os arquivos de input, output, o algoritmo gerador dos gráficos e os próprios gráficos. O diretório *bin* é o destino para todos os arquivos executáveis e compilados. Por fim, o diretório *scripts* possui arquivos em *shell-script* que executam todo o projeto.

Pelas execuções dos algoritmos, percebe-se que para instâncias pequenas os algoritmos são competitivos, no entanto, para instâncias maiores, como a terceira, o método DPI não apresenta bons resultados comparados aos outros dois. Além disso, percebe-se que o B&B apresenta melhores resultados para instâncias maiores levando em consideração a construção da solução inicial utilizada.

A fim de um melhor entendimento, este relatório se encontra subdividido em seções focadas a cada fase do projeto, desde o objetivo e organização dos dados de entrada à formação das soluções comparadas na seção de resultados. A distribuição esta feita da seguinte forma:

- Seção 2 encontra-se a descrição do método *DPR*;
- Seção 3 encontra-se a descrição do método *DPI*;
- Seção 4 encontra-se a descrição do método *B&B*;
- Seção 5 apresenta os resultados obtidos pelos algoritmos;
- Seção 6 apresenta uma análise final mais abrangente acerca da pesquisa.

[2]Mais informações em: https://developer.mozilla.org/pt-BR/docs/Web/JavaScript
[3]Código disponível em: https://github.com/vvarg-iinet/KnapsackProblemSolver

2 DPR

Programação Dinâmica recursiva com uso de memoização é uma técnica para solucionar problemas combinatoriais sem a necessidade de recomputar cálculos previamente feitos. Para tal, além de dividir o problema em subproblemas e resolver cada um separadamente, os resultados parcialmente adquiridos são guardados em uma tabela. Assim, antes de computar uma nova solução de um problema, verifica-se a existência do valor nesta tabela.

```
1  int solve(const Problem* instance, int nItems, int capacity) {
2      if (capacity == 0 || nItems == 0) return 0;
3
4      if (this->memo[nItems][capacity] == -1) {
5          Item* item = instance->getItem(nItems - 1);
6
7          if (item->weight > capacity) {
8              this->memo[nItems][capacity] = this->solve(instance, nItems - 1,
                   ↪  capacity);
9          } else {
10             int notInSolution = this->solve(instance, nItems - 1, capacity);
11             int inSolution = this->solve(instance, nItems - 1, capacity -
                   ↪  item->weight) + item->value;
12
13             if (notInSolution > inSolution) {
14                 this->memo[nItems][capacity] = notInSolution;
15             } else {
16                 this->memo[nItems][capacity] = inSolution;
17             }
18         }
19     }
20
21     return this->memo[nItems][capacity];
22 }
```

Para a implementação do DPR, é necessário dividir o problema em subproblemas que podem ser resolvidos recursivamente. Para isso, devemos criar um caso base para a parada da recursão. No caso do problema da mochila, temos a linha 2 representando o caso base. Além disso, como temos um problema de maximização, devemos analisar como teremos um lucro maior na mochila, analisando os 2 subproblemas a partir do atual. Inicialmente verificamos o subproblema que não possui o item atual na mochila, e então o subproblema que possui o item atual na mochila (linhas 10–11).

Em resumo esta é a versão recursiva, no entanto, para adaptar para a programação dinâmica temos a condição que verifica se a solução atual já foi computada. Esta solução é representada na linha 4.

Em outras palavras, seja $K(i, w)$ a função que calcula o maior lucro com i itens em uma mochila com capacidade w. Temos como definição recursiva do problema:

$$K(i,w) = \begin{cases} 0 \Leftrightarrow i = 0 \; ou \; w = 0 \\ K(i-1,w) \Leftrightarrow w_i > w \\ max\{K(i-1,w),\; K(i-1,w-w_i) + v_i\} \end{cases}$$

3 DPI

Já a abordagem iterativa utilizada preenche completamente uma matriz de resultados item a item para cada capacidade possível da mochila. Ou seja, dado uma matriz $M_{i,w}$ onde $i \in \{0..I\}$ e I é o número de itens e $w \in \{0..W\}$ onde W é a capacidade máxima da mochila, analísa-se qual é o maior valor entre o elemento $(M_{i-1,w})$ e $(M_{i-1,w-w_i} + v_i)$ onde w_i e v_i são o peso e valor do elemento i respectivamente.

Abaixo encontra-se a tradução para códgo da abordagem mencionada. Inicialmente temos um loop aninhado para percorrer a matriz M (linhas 2–3), seguido da verificação se o elemento cabe na mochila (linha 6) e em seguida da verificação de qual elemento da matriz é maior (linhas 9–10).

```
1  void solve(const Problem* instance) {
2      for (int i = 1; i <= instance->getSize(); i++) {
3          for (int j = 1; j <= instance->getCapacity(); j++) {
4              Item* item = instance->getItem(i - 1);
5
6              if (item->weight > j) {
7                  this->solution[i][j] = this->solution[i - 1][j];
8              } else {
9                  int notInSolution = this->solution[i - 1][j];
10                 int inSolution = this->solution[i - 1][j - item->weight] +
                   ↪  item->value;
11
12                 if (notInSolution > inSolution) {
13                     this->solution[i][j] = notInSolution;
14                 } else {
15                     this->solution[i][j] = inSolution;
16                 }
17             }
18         }
19     }
20
21     this->value = this->solution[instance->getSize()][instance->getCapacity()];
22 }
```

Em outras palavras, seja $M(i, w)$ o elemento da matriz M que representa o maior lucro com até os i primeiros itens em uma mochila com capacidade w. Temos que o problema

é:

$$M_{i,w} = \begin{cases} 0 \Leftrightarrow i = 0 \; ou \; w = 0 \\ M_{i-1,w} \Leftrightarrow w_i > w \\ max\{M_{i-1,w}, \; M_{i-1,w-w_i} + v_i\} \end{cases}$$

4 Branch & Bound

O *Branch & Bound* é um técnica de implementação de algoritmo que é usada para resolver problemas de otimização combinatorial. Esses problemas costumam ter uma complexidade exponencial, pois pode ser necessário explorar todas as possibilidades no pior caso. Essa técnica utiliza estratégias para diminuir o espaço de busca do algoritmo para poder otimizá-lo.

A implementação do algoritmo principal realizada nesse trabalho é a seguinte:

```
1  void solve(const Problem* instance, vector<int> items, int item, int value, int
   ↪  weight) {
2
3      if (this->isComplete(items)){
4
5          this->bestAssignment = items;
6          this->bestValue = value;
7
8      } else{
9          items[item] = 0;
10         if (this->isConsistent(instance, items) && this->isPromissing(instance,
           ↪  items, item, value, weight)){
11             this->solve(instance, items, item + 1, value, weight);
12         }
13
14         items[item] = 1;
15         if (this->isConsistent(instance, items) && this->isPromissing(instance,
           ↪  items, item, value, weight)){
16             this->solve(instance, items, item + 1, value +
               ↪  instance->getItem(item)->value, weight +
               ↪  instance->getItem(item)->weight);
17         }
18
19         items[item] = -1;
20     }
21
22 }
```

O algoritmo começa verificando se a solução atual é completa, em caso positivo a melhor solução é atualizada e caso contrário será necessário expandir a árvore de possibilidades. Para isso todos os estados possíveis para o item atual serão testados, porém é realizada uma verificação para assegurar que essa ramificação da árvore irá gerar soluções consistentes e promissoras. Caso não seja possível garantir essas condições, essa rami-

ficação não será explorada.

Para que seja possível executar o algoritmo é necessário uma solução inicial consistente e completa. Para isso foi desenvolvido um algoritmo para gerá-la de maneira gulosa que leva em consideração a proporção *valor* × *peso*. Os itens são ordenados de maneira decrescente de acordo com essa proporção e então inseridos na solução caso ela continue consistente. A implementação do algoritmo guloso realizada nesse trabalho é a seguinte:

```
void BranchBoundKnapsack::initialSolution(const Problem* instance) {

    vector<pair<double, int>> proportion;
    for (int i = 1; i <= instance->getSize(); i++){
        Item* item = instance->getItem(i - 1);
        pair<double, int> x;
        x.first = (double)item->value / (double)item->weight;
        x.second = i - 1;

        proportion.push_back(x);
    }

    this->valueProportion = proportion;
    sort(proportion.begin(), proportion.end(), greater<pair<double, int>>());

    int capacity = instance->getCapacity();

    for (int i = 1; i <= (int) proportion.size(); i++){
        int position = proportion[i - 1].second;
        Item* item = instance->getItem(position);

        if (capacity - item->weight >= 0){
            this->bestAssignment[position] = 1;
            this->weight += item->weight;
            this->bestValue += item->value;
            capacity -= item->weight;

        } else{
            this->bestAssignment[position] = 0;
        }
    }

}
```

5 Resultados

Como citado anteriormente, este trabalho consiste na análise dos três algoritmos apresentados anteriormente. Para cada execução foi analisado o tempo necessário para cada algoritmo resolver o problema da mochila. Vale ressaltar que as instâncias são iguais para todos os algoritmos o que fornece uma maior credibilidade para a comparação dos resultados. Os dados coletados para cada um dos algoritmos estão representados nos gráficos 5.1, 5.2, 5.3 onde o eixo x equivale a instância e o eixo y equivale ao tempo gasto em microsegundos para solucionar o problema.

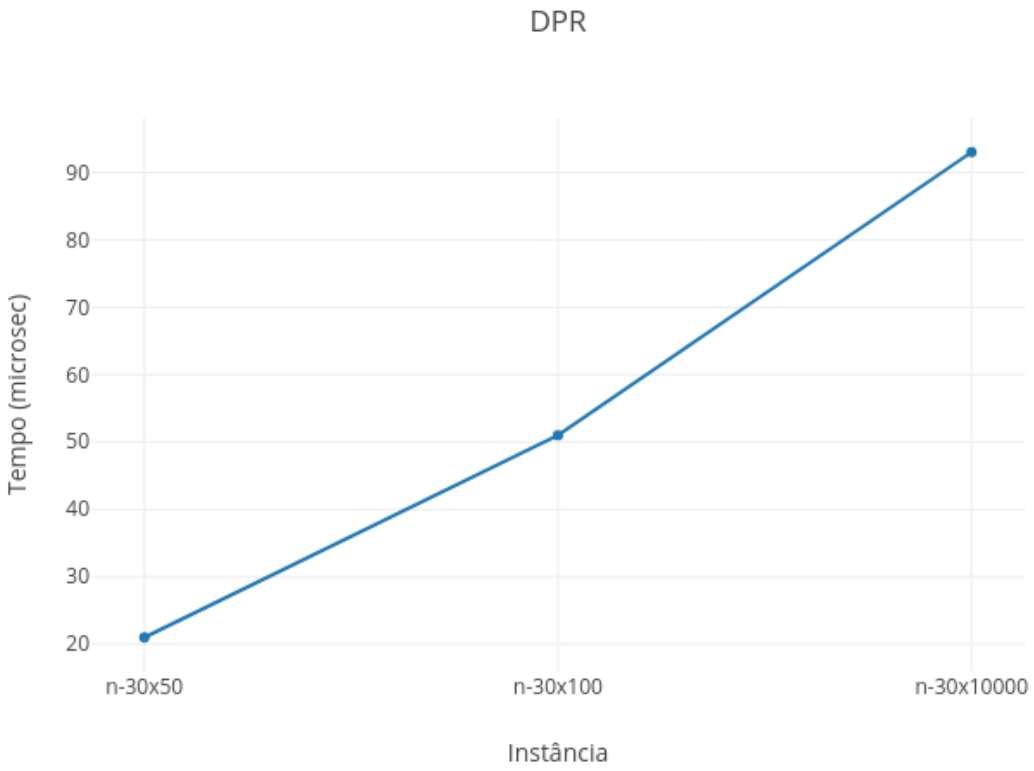

Figura 5.1: DPR

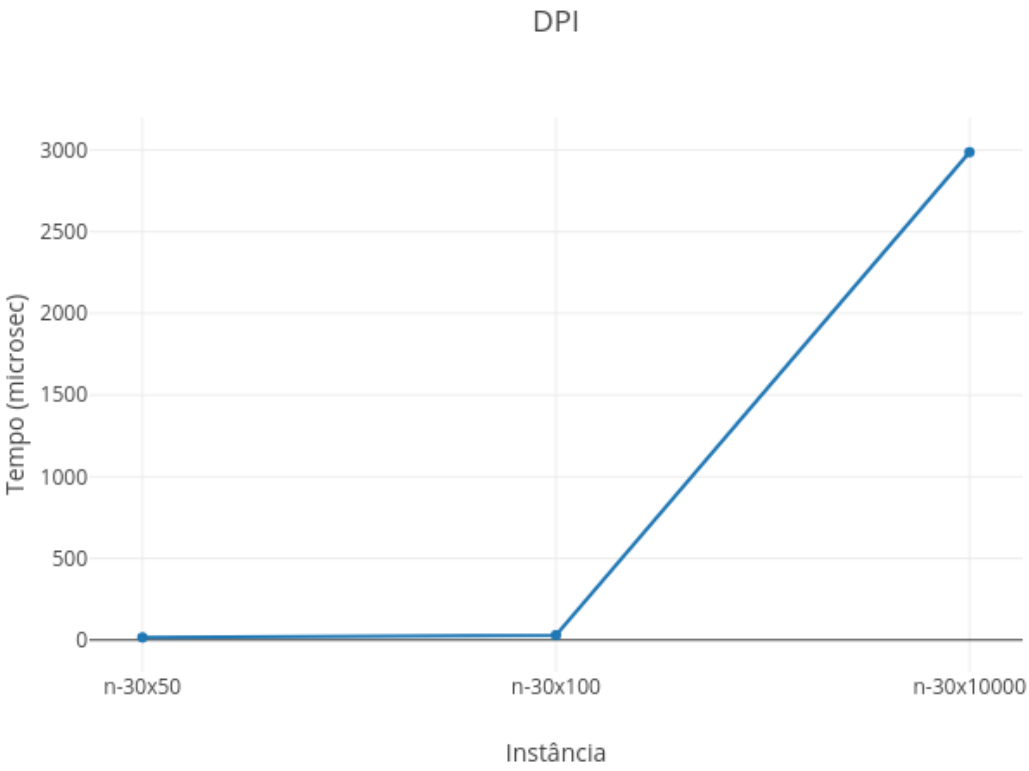

Figura 5.2: DPI

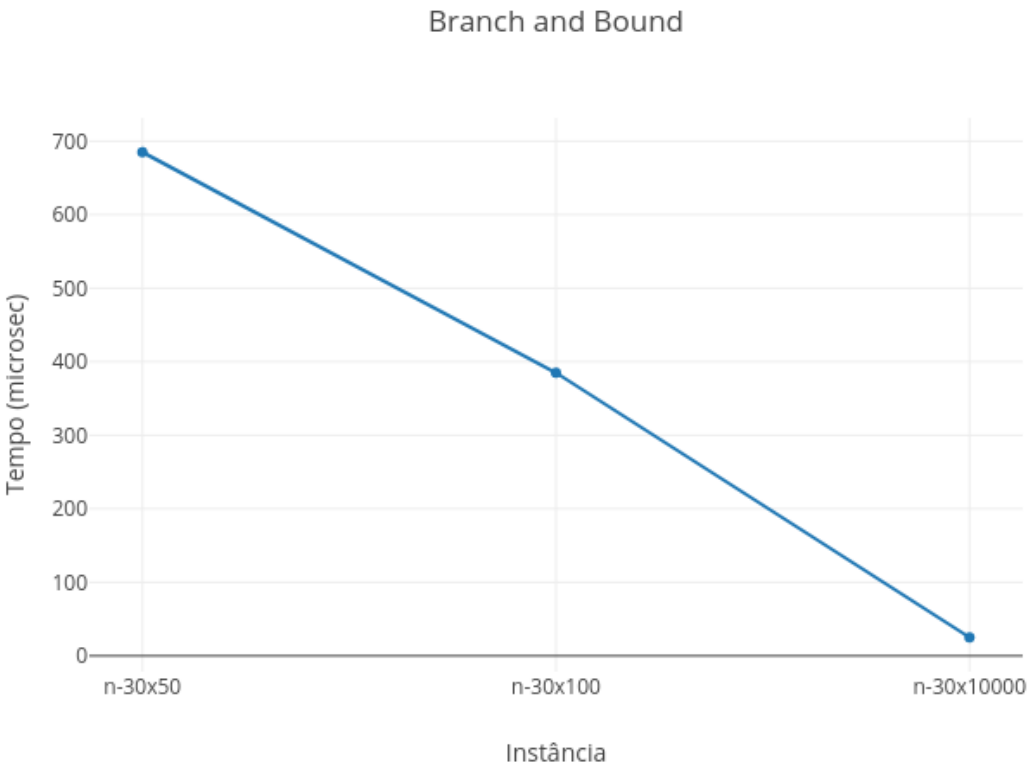

Figura 5.3: Branch and Bound

Para facilitar a análise entre os três algoritmos, o gráfico 5.4 foi gerado contendo os dados de todos eles. Com ele é possível verificar a maior eficiência da técnica *Branch & Bound* para problemas maiores. Em contra partida, o *DPR* e o *DPI* pioram sua eficiência conforme o problema cresce, porém são melhores que o *Branch & Bound* para problemas pequenos.

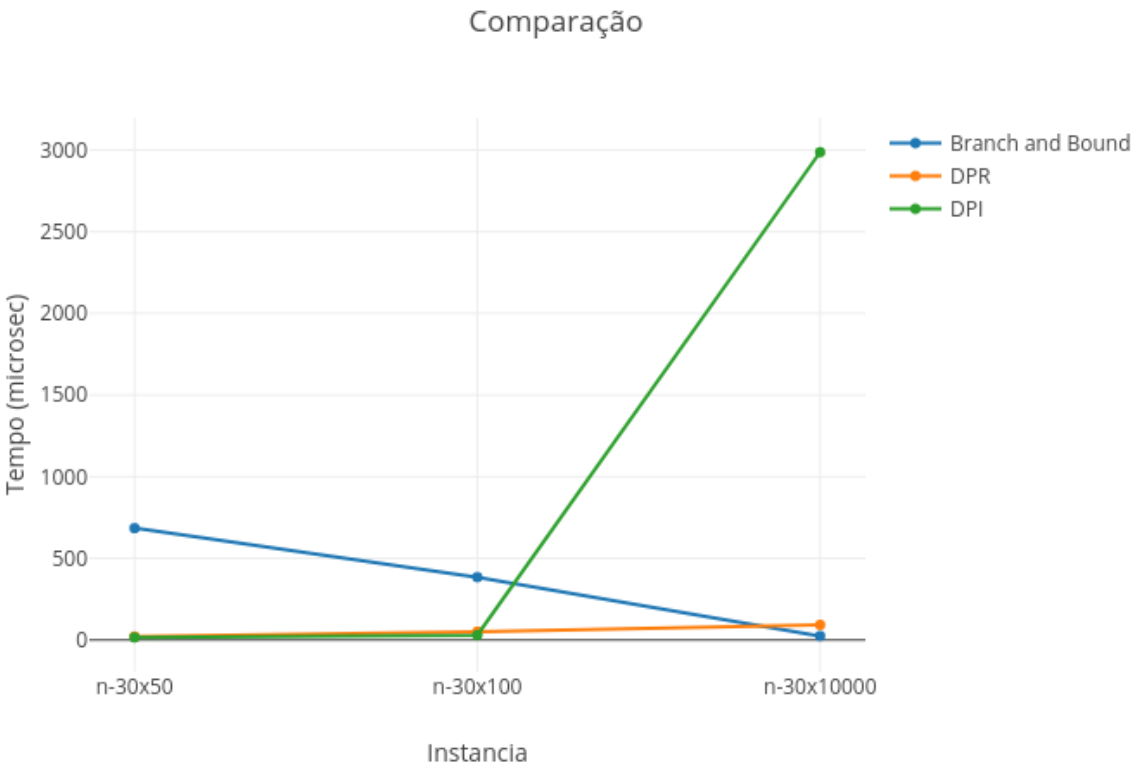

Figura 5.4: Branch and Bound X DPR X DPI

6 Conclusões

Através dos testes e das análises realizadas pode-se concluir que dos três algoritmos, o *Branch & Bound* possui o melhor desempenho para problemas grandes. O *DPR* e o *DPI* possuem um desempenho melhor que o *Branch & Bound* para problemas pequenos, porém a eficiência de ambos é inversamente proporcional ao tamanho do problema. Além disso, observa-se que o *DPI* possui uma péssima eficiência comparada aos outros dois algoritmos para instância grande, o que é o esperado dado sua complexidade $\theta(Wn)$.

Portanto pode-se concluir que a técnica a ser escolhida para resolver o problema depende do tamanho do mesmo, visto que a eficiência dos três algoritmos depende do tamanho das instâncias.

Referências Bibliográficas

Ziviani, N. *Projetos de Algoritmos com implementações em Pascal e C*; Cengage Learning, 2010.

ASCENCIO, A. F. G.; ARAÚJO, G. S. d. Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++. *São Paulo: Perarson Prentice Halt* **2010**, *3*.

Dasgupta, S.; Papadimitriou, C. H.; Vazirani, U. V. *Algorithms*; McGraw-Hill Higher Education New York, 2008.