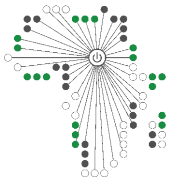


Universidade Virtual Africana

INFORMÁTICA APLICADA: ITI 3300

DESIGN E ANÁLISE DE ALGORITMOS

Arlete Maria Vilanculos

Prefácio

A Universidade Virtual Africana (AVU) orgulha-se de participar do aumento do acesso à educação nos países africanos através da produção de materiais de aprendizagem de qualidade. Também estamos orgulhosos de contribuir com o conhecimento global, pois nossos Recursos Educacionais Abertos são acessados principalmente de fora do continente africano.

Este módulo foi desenvolvido como parte de um diploma e programa de graduação em Ciências da Computação Aplicada, em colaboração com 18 instituições parceiras africanas de 16 países. Um total de 156 módulos foram desenvolvidos ou traduzidos para garantir disponibilidade em inglês, francês e português. Esses módulos também foram disponibilizados como recursos de educação aberta (OER) em oer.avu.org.

Em nome da Universidade Virtual Africana e nosso patrono, nossas instituições parceiras, o Banco Africano de Desenvolvimento, convido você a usar este módulo em sua instituição, para sua própria educação, compartilhá-lo o mais amplamente possível e participar ativamente da AVU Comunidades de prática de seu interesse. Estamos empenhados em estar na linha de frente do desenvolvimento e compartilhamento de recursos educacionais abertos.

A Universidade Virtual Africana (UVA) é uma Organização Pan-Africana Intergovernamental criada por carta com o mandato de aumentar significativamente o acesso a educação e treinamento superior de qualidade através do uso inovador de tecnologias de comunicação de informação. Uma Carta, que estabelece a UVA como Organização Intergovernamental, foi assinada até agora por dezenove (19) Governos Africanos - Quênia, Senegal, Mauritânia, Mali, Costa do Marfim, Tanzânia, Moçambique, República Democrática do Congo, Benin, Gana, República da Guiné, Burkina Faso, Níger, Sudão do Sul, Sudão, Gâmbia, Guiné-Bissau, Etiópia e Cabo Verde.

As seguintes instituições participaram do Programa de Informática Aplicada: (1) Université d'Abomey Calavi em Benin; (2) Université de Ougagadougou em Burkina Faso; (3) Université Lumière de Bujumbura no Burundi; (4) Universidade de Douala nos Camarões; (5) Universidade de Nouakchott na Mauritânia; (6) Université Gaston Berger no Senegal; (7) Universidade das Ciências, Técnicas e Tecnologias de Bamako no Mali (8) Instituto de Administração e Administração Pública do Gana; (9) Universidade de Ciência e Tecnologia Kwame Nkrumah em Gana; (10) Universidade Kenyatta no Quênia; (11) Universidade Egerton no Quênia; (12) Universidade de Addis Abeba na Etiópia (13) Universidade do Ruanda; (14) Universidade de Dar es Salaam na Tanzânia; (15) Universite Abdou Moumouni de Niamey no Níger; (16) Université Cheikh Anta Diop no Senegal; (17) Universidade Pedagógica em Moçambique; E (18) A Universidade da Gâmbia na Gâmbia.

Bakary Diallo

O Reitor

Universidade Virtual Africana

Créditos de Produção

Autor

Arlete Maria Vilanculos Ferrão

Par revisor(a)

José Luis Sambo

UVA - Coordenação Académica

Dr. Marilena Cabral

Coordenador Geral Programa de Informática Aplicada

Prof Tim Mwololo Waema

Coordenador do módulo

Jules Degila

Designers Instrucionais

Elizabeth Mbasu

Benta Ochola

Diana Tuel

Equipa Multimédia

Sidney McGregor

Barry Savala

Edwin Kiprono

Kelvin Muriithi

Victor Oluoch Otieno

Michal Abigael Koyier

Mercy Tabi Ojwang

Josiah Mutsogu

Kefa Murimi

Gerisson Mulongo

Direitos de Autor

Este documento é publicado sob as condições do Creative Commons

Http://en.wikipedia.org/wiki/Creative_Commons

Atribuição http://creativecommons.org/licenses/by/2.5/

O Modelo do Módulo é copyright da Universidade Virtual Africana, licenciado sob uma licença Creative Commons Attribution-ShareAlike 4.0 International. CC-BY, SA

Apoiado por

Projeto Multinacional II da UVA financiado pelo Banco Africano de Desenvolvimento.

Índice

Prefácio 2

Créditos de Produção 3

Aviso de direitos autorais 4

Supporté par 4

Descrição Geral do Curso 8

Pré-requisitos . . . 8

Materiais . . . 8

Recursos da Internet . . . 9

Objetivos do Curso . . . 9

Unidades . . . 10

Avaliação . . . 10

Calendarizaçã . . . 11

Leituras e outros Recursos . . . 12

Unidade 0: Diagnóstico 15

Introdução à Unidade . . . 15

Objetivos da Unidade . . . 15

Actividades de Aprendizagem . . . 16

Atividades 1: Estruturas de controlo . . . 16

Introdução 16

Detalhes da atividade . . . 16

Fluxo sequêncial 16

Conclusão 19

Avaliação da Unidade . . . 19

Instruções 19

Critérios de Avaliação . . . 20

Leituras e Outros Recursos . . . 20

Unidade 1. Análise Assintótica do Tempo de Execução de Algoritmos 21

Introdução à Unidade . . . 21

Objetivos da Unidade 21

Actividades de Aprendizagem 22

Actividades 1: Notação Big O (O, Ω e Θ) 22

Introdução 22

Detalhes da actividade. 22

Classes de comportamento assimptótico 27

Conclusão 29

Avaliação 29

Actividades 2: Análise de programas recursivos 30

Introdução 30

Detalhes da actividade. 30

Análise de funções recursivas 30

Actividades 3: Análise de programas iterativos 32

Introdução 32

Detalhes da actividade. 32

Identidades úteis envolvendo somatórios 32

Conclusão 33

Avaliação 33

Resumo da Unidade 33

Avaliação da Unidade 34

Leituras e outros Recursos 35

Unidade 2. Técnica de Desenho de Algoritmos 36

Introdução à Unidade 36

Objetivos da Unidade 36

Actividades de Aprendizagem 37

Actividade 1: Pesquisa Exaustiva (força bruta). 37

Detalhes da actividade. 37

Actividade 2: Divisão e Conquista 41

Introdução 41

Detalhes da atividade 41

Ordenação por intercalação 42
Conclusão: 44
Actividades 3: Programação dinâmica **45**
Introdução 45
Detalhes da actividade. **45**
Avaliação 45
Conclusão 47
Avaliação 47
Resumo da Unidade **48**
Avaliação da Unidade **48**
Instruções 48
Avaliação Sistemática **49**
Critérios de Avaliação 49
Leituras e outros Recursos **49**
Unidade 3. Algoritmos em Grafos **50**
Introdução à Unidade **50**
Objetivos da Unidade **50**
Actividades de Aprendizagem **51**
Actividades 1: Representação de Grafo **51**
Introdução 51
Detalhes da actividade: **51**
Conclusão 53
Actividade 2: Pesquisa em largura e pesquisa em profundidade **54**
Introdução 54
Detalhes da actividade. **54**
Avaliação 54
Actividade 3: Árvores de cobertura mínima **56**
Introdução 56
Detalhes da actividade. **56**
Avaliação 56

Algoritmos para encontrar árvores de cobertura mínima 57

Conclusão 58

Avaliação 59

Resumo da Unidade 59

Avaliação Sistemática 60

Critérios de Avaliação 60

Avaliação da Unidade 60

Unidade 4. Estruturas de Dados Especializados 61

Introdução à Unidade 61

Objetivos da Unidade 61

Actividades de Aprendizagem 62

Actividades 4.1: Filas de prioridade 62

Introdução 62

Detalhes da atividade 62

Implementação de uma fila de prioridade em vetores 63

Conclusão 64

Actividades 2: Árvores binárias de pesquisa 65

Introdução 65

Detalhes da actividade. 65

Avaliação 65

Avaliação 70

Conclusão 70

Actividades 3: Conjuntos disjuntos 71

Introdução 71

Detalhes da actividade. 71

Implementação através de Listas Encadeadas 71

Conclusão 72

Avaliação 73

Resumo da Unidade 73

Avaliação da Unidade 73

Instruções 73

Avaliação Sistemática 74

Critérios de Avaliação 74

Leituras e outros Recursos 74

Resumo da Unidade 75

Avaliação da Curso 75

Instruções 75

EXAME FINAL 1 75

EXAME FINAL 2 76

Avaliação Sistemática 77

Critérios de avaliação 77

Leituras e outros Recursos 77

Descrição Geral do Curso

Bem-vindo(a) a Design e Análise de Algoritmos

No módulo de Análise e Design de Algoritmos, você irá ganhar familiaridade com princípios fundamentais de design e análise de algoritmos e demonstrar competências na análise de complexidade de algoritmos e compreensão das principais classes de complexidade. É objectivo deste curso que você adquira competências de desenho e análise de algoritmos eficientes, técnicas de pesquisa exaustiva, estruturas de dados especializadas e ainda a programação dinâmica e amortização. Com os conhecimentos adquiridos neste módulo, você poderá utilizá-los no campo profissional ao fazer a aplicação de algoritmos genéricos a problemas concretos.

Pré-requisitos

- Introdução à programação estruturada
- Matemática discreta
- Álgebra linear
- Programação Orientada a Objectos
- Algoritmos e Estruturas de Dados
- Introdução à Estatística e probabilidade
- Fundamentos de Organização e Arquitetura Informática

Materiais

Os materiais necessários para completar este curso incluem:

- Udi Manber, Introduction to Algorithms: A Creative Approach.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms (3rd edition). The MIT Press, 2009.
- J. Kleinberg and E. Tardos, Algorithm Design, Addison-Wesley, 2005.
- R. Motwani and P. Raghavan, Randomized Algorithms. Cambridge U. Press, 1995
- Christos H. Papadimitriou, Kenneth Steiglitz, Kenneth Steiglitz. Combinatorial Optimization: Algorithms And Complexity. Courier Dover Publications, 1998
- Celes, Waldemar, Cerqueira, Renato. Rangel, José Lucas.Introdução à Estruturas de Dados e Algoritmos, 7ª edição. 2004, Elsevier Editora.
- Edelweis, Nina, Galante, Renata. Estrutura de Dados , Porto Alegre. Bookman, 2009.
- Szwarcfiter, J., Markenzon, L., Estrutura de Dados e Algoritmos, 3ª Edição Rio Janeiro, LTC 2010.

- Michael T. Goodrich and Roberto Tamassia. Algorithm Design. Wiley, 2002.
- Steven S. Skiena. The Algorithm Design Manual (2nd edition). Springer, 2008.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms (3rd edition). The MIT Press, 2009
- Jon Kleinberg and Éva Tardos. Algorithm Design. Addison-Wesley, 2006.
- Michael T. Goodrich and Roberto Tamassia. Algorithm Design. Wiley, 2002.
- Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. W. H. Freeman and Company, 1979.
- Steven S. Skiena and Miguel A. Revilla. Programming Challenges: The Programming Contest Training Manual. Springer-Verlag, 2003.

Recursos da Internet

- http://people.cs.vt.edu/~shaffer/Book/C++3e20130328.pdf
- http://www.unl.pt/guia/2007/fct/UNLGI_getUC?uc=8154
- http://www.dcc.fc.up.pt/~pribeiro/aulas/daa1516

Outros recursos incluem sebentas elaboradas pelo docente da cadeira, materiais em PowerPoint, não se esquecendo que a internet também pode ser uma fonte de consulta e leitura.

Objetivos do Curso

Após concluir este curso, o(a) aluno(a) deve ser capaz de:

- Demonstrar competência na área de técnicas de concepção e análise de algoritmos eficientes;
- Saber escolher e aplicar uma técnica de desenho de algoritmos adequada à resolução de um problema concreto;
- Calcular a complexidade de algoritmos;
- Saber escolher, comparar e conceber e utilizar estruturas de dados adequadas ao problema a resolver.
- Demonstrar conhecimentos em estruturas de dados avançadas como árvores de busca binária, tabelas (harsh tables) e grafos;
- Aplicar as estruturas de dados que permitam implementar os algoritmos fundamentais de grafos com eficiência;
- Aplicar as principais técnicas avançadas de desenho de algoritmos, nomeadamente, programação dinâmica e amortização;
- Demonstrar competências no conhecimento classes de complexidade e alguns dos problemas em aberto;

Unidades

Unidade 0: Diagnóstico

Com esta unidade, pretende-se que você faça a revisão da matéria relacionada com os conteúdos que serão discutidos neste módulo, como é o caso de estruturas de dados e programação estruturada.

Unidade 1: Análise assintótica do tempo de execução de algoritmos

Nesta unidade pretende-se que você adquira competências para a análise de algoritmos. Analisar algoritmos por outras palavras significa fazer uma estimativa do tempo que o algoritmo leva a ser executado, tendo em consideração que o tempo é determinado por vários factores tais como, o tamanho do problema, a natureza do algoritmos, etc.

Unidade 2: Técnicas de Desenho de Algoritmos

O objectivo desta unidade definir uma forma de criar uma medida de comparação entre diferentes algoritmos que resolvem um mesmo problema, de modo que se possa avaliar a solução viável para um determinado problema.

Unidade 3: lgoritmos de grafos

Nesta unidade discutiremos um tipo de dados não linear denominado grafo, bem como a sua representação na memória e as várias operações e algoritmos sobre eles.

Unidade 4: Algumas estruturas de dados especializadas

Neste unidade, discutir-se-á o conceito de fila de prioridade, que é uma estrutura de dados que se assemelha a uma fila de um banco por exemplo, onde se estabelece uma prioridade para idosos, gestantes, portadores de deficiências, etc. Para além das filas de prioridade, também discutir-se-á os conceitos de conjuntos disjuntos e árvores binárias de pesquisa.

Avaliação

Em cada unidade encontram-se incluídos instrumentos de avaliação formativa a fim de verificar o progresso do(a)s aluno(a)s.

No final de cada módulo são apresentados instrumentos de avaliação sumativa, tais como testes e trabalhos finais, que compreendem os conhecimentos e as competências estudadas no módulo.

A implementação dos instrumentos de avaliação sumativa fica ao critério da instituição que oferece o curso. A estratégia de avaliação sugerida é a seguinte:

1	Avaliação do tempo de estudo	20%
2	Avaliação do trabalho independente	30%
3	Exame final	50%

Calendarizaçã

Unidade	Temas e Atividades	Estimativa do tempo
0	Diagnóstico 1.1 Comandos de decisão e de iteração	5 horas
1	Análise assintótica do tempo de execução de algoritmos 1.1.Notação Big O, teta e ômega 1.2. Análise de programas recursivos 1.3. Análise de programas iterativos	15 horas
2	Técnica de desenho de algoritmos 2.1.Pesquisa exaustiva (força bruta) 2.2. Divisão e conquista 2.3. Programação dinâmica	45 horas
3	Algoritmos em grafos 3.1. Representação de grafos 3.2. Pesquisa em largura e pesquisa em profundidade 3.3. Árvores de cobertura mínima	30 horas
4	Algumas estruturas de dados especializadas 4.1. Filas de prioridade 4.2. Conjuntos disjuntos 4.3. Árvores binárias de pesquisa	25 horas

Leituras e outros Recursos

As leituras e outros recursos deste curso são:

Unidade 0

Leituras e outros recursos obrigatórios:

- Stroustrup, B. (2014). Programming Principles and Practice Using C and C++. Addison Wesley ISBN0321992784
- Balagurusamy, E. (2008). Programming in ANSI C. (4ª ed). New Delhi: Tata Mc-Graw-Hill

Unidade 1

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- Nivio Ziviani, Projecto de Algoritmos com Implementação em Java e C++.. 2006. Editora Thomson, ISBN 8522105251
- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C.
- Introduction to Algorithms, 3ª edição, MIT Press, 2009.
- SZWARCFITER, J. L. e MARKENZON, L. Estruturas de Dados eseus Algoritmos, LTC, 1994.
- ZIVIANI, N. Projeto de Algoritmos: com implementações em Pascal e C, 2ª edição, Cengage Learning, 2009

Leituras e outros recursos opcionais:

- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Rocha, A. M. A, Estruturas de Dados e Algoritmos em C, 2008 FCA Editora Informática. Coleção:Tecnologias de Informação
- SEYMOUR LIPSCHUTZ, G. A. VIJAYALAKSHMI PAI. Data Structures. Tata McGraw-Hill Publishing Company Limited. New Delhi, 2006.

Unidade 2

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- Nivio Ziviani, Projecto de Algoritmos com Implementação em Java em C++. 2006. Editora Thomson, ISBN 8522105251

Leituras e outros recursos opcionais:

- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Adam Drozdek, Data Structures and Algorithms in Java, 2ª edição.

Unidade 3

Leituras e outros recursos obrigatórios:

- SAMS Teach Yourself Data Structures And Algorithms In 24 hours. 1999
- Concise Notes on Data Structures and Algorithms, Edições Ruby. Christopher Fox. 2012
- Data Structures Using C++, Second Edition, D.S. Malik, 2010

Leituras e outros recursos opcionais:

- MARK ALLEN WEISS. Data Structures and Algorithms Analysis (2nd Ed)1994
- Robert Sedgewick, Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", 3rd Edition, 2001

Unidade 4

Leituras e outros recursos obrigatórios:

- V. Das, Principles of Data Structures and Algorithms using C and C++, 2008.
- Robert Sedgewick, Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", Parts 1-4, 2012.
- Adam Drozdek, Data Structures and Algorithms in Java, 2ª edição.

Leituras e outros recursos opcionais:

- Robert Sedgewick, Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms", 3rd Edition, Addison Wesley, 2001.
- ROCHA, A. Estrutura de dados e algoritmos em C, FCA Lisboa 2008
- Carvalho A, Exercicios de Java; Algoritmia e Programação Estruturada, FCA, Lisboa 2013
- http://delta.cs.cinvestav.mx/~adiaz/anadis/Analisis.pdf

Unidade 0: Diagnóstico

Introdução à Unidade

A presente unidade visa verificar os conhecimentos prévios do aluno em matéria de processamento de dados, manipulação de funções e procedimentos, bem ainda a manipulação de matrizes. Para além destes conhecimentos, o aluno deve demostrar conhecimentos na escrita de algoritmos, utilizando as estruturas de seleção, controle e de repetição e, poder transformar os mesmos em uma linguagem de programação.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Demonstrar capacidades introdutórias de programação estruturada, escrevendo algoritmos sequenciais para resolver problemas que envolvem entrada, processamento e saída de dados.
- Aplicar funções e procedimentos no desenvolvimento de algoritmos;
- Traduzir algoritmos em código fonte executável em linguagem de programação estruturada;

Termos-chave

Estrutura sequencial: é um conjunto de instruções no qual cada instrução será executada de forma sequencial, isto é de cima para baixo e da direita para esquerda.

Estruturas de repetição: Utilizadas quando se deseja executar um determinado conjunto de instruções por um número definido ou indefinido de vezes, enquanto um determinado estado prevalecer ou até que seja alcançado um determinado estado.

Estruturas de controlo: permitem ao programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores.

Matrizes: coleção de um ou mais objetos, do mesmo tipo, armazenados em endereços adjacentes de memória.

Algoritmo: sequência finita de etapas não ambíguas e nem redundantes que conduzem a resolução de um problema.

Funções: blocos de execução internos a um programa, chamados sub-programas. São avaliados e retornam algum valor ao programa.

Actividades de Aprendizagem

Atividades 1: Estruturas de controlo

Introdução

Numa linguagem de programação, os comandos de fluxo indicam a ordem em que o processamento é efetuado. Nesta atividade, discutiremos como forma de revisão, o fluxo sequêncial, comandos de seleção ou decisão e de repetição ou iteração.

Detalhes da atividade

Fluxo sequêncial

A estrutura sequencial de um programa, como o nome sugere, é a estrutura mais simples na execução de instruções de um programa. Pois, as instruções são executadas na ordem em que aparecem, de cima para baixo e da direita para a esquerda, sem nenhum desvio.

Exemplo: Um programa em linguagem de programação C, para demonstrar o uso das expressões e sua avaliação

```
main()

{

float a, b, c, x, y, z;

a=9; b=12; c=3;

x=a-b/3 +c*2-1;

y=a-b/(3+c)* (2-1);

z= a-b/(3+c)* 2)-1;

printf(“x=%f\n”, x);
```

```
printf("y=%f\n", y);

printf("z=%f\n", z);

}
```

Comando de seleção ou decisão

Com as instruções de salto ou desvio pode-se fazer com que o programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores. (Balagurusamy, 2008)

As principais estruturas de decisão são:

if

Sintaxe:

```
            If (expressao teste)
                  {
                  bloco de instruções;
                  }
                  instruções;
```

If… else

Sintaxe:

if(expressão teste)

```
          {
            bloco de instruções; /*executado quando  verdadeiro*/
           }
      else
          {
            bloco de instruções; /*executado quando  falso*/
           }
            instruções;
```

Case

Sintaxe:

```
switch(expressao)
{
case valor1: instruções
```

```
break;

case valor2: instruções

break;

………..

Defaut:

Default block;

}

Instruções x;
```

Comandos de repetição ou iteração

Estes comandos são utilizados quando se pretende que um determinado conjunto de instruções seja executado um número definido ou indefinido de vezes, ou enquanto um determinado estado prevalecer ou até que seja alcançado. (Balagurusamy, 2008)

As principais estruturas de repetição são:

While …Loop

Sintaxe:

```
while( expressão )

 instrução
```

do … While

```
do

{

instrução

}

While (expressão);
```

For

Sintaxe:

```
        for(inicializacao; Condicao_teste; incremento/decremento)

{

corpo do ciclo;

}
```

Conclusão

Nesta unidade denominada de diagnóstico, o estudante teve a oportunidade de rever os conteúdos aprendidos nas disciplinas antecedentes que são a base para o entendimento da disciplina de Análise e Design de Algoritmos. O entendimento das estruturas de decisão, repetição, é muito importante para a resolução de vários problemas diários pois com as instruções de salto e de desvio possibilita fazer com que o programa proceda de uma ou outra maneira, de acordo com as decisões lógicas tomadas em função dos dados ou resultados anteriores.

Avaliação da Unidade

Teste Diagnóstico

Instruções

As seguintes questões estão relacionadas com o que aprender nas disciplinas de princípios de programação e introdução à informática aplicada. Responda as seguintes questões com clareza, dando exemplos se for necessário.

1. Elabore uma síntese da história da linguagem de programação em C.
2. Elabore um quadro comparativo de Instruções em C e respectivos exemplos.
3. Escreva um programa para calcular a media de 7 avaliações e, determine com base na média as condições de excluído, admitido ou dispensado.
4. Escreva um programa com uso de matrizes, que imprima a tabela de multiplicação.
5. Escreva um programa modular interativo usando funções que lêm os valores de 3 lados de um triângulo e mostrar se é área ou perímetro de acordo com o pedido do usuário. Dados 3 lados:

   Perímetro = a+b+c

   Área = $\sqrt{(s-a)(s-b)(s-c)}$

   Onde s=(a+b+c)/2

Critérios de Avaliação

Pergunta	Pontuação (máximo 10 valores)
1	1.5
2	1.5
3	2
4	2.5
5	2.5

Leituras e Outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de “Leituras e Outros Recursos do curso”.

- http://www.mheducation.com

Unidade 1. Análise Assintótica do Tempo de Execução de Algoritmos

Introdução à Unidade

Um critério importante na avaliação de algoritmos é o tempo que demoram e o modo como esse tempo varia quando o tamanho das instâncias cresce. Para que a comparação de eficiência faça sentido, procura-se uma estimativa do tempo de execução que seja função do comprimento dos dados mas independente da máquina, do sistema operativo, da linguagem em que o algoritmo é implementado, da versão do compilador.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Demonstrar competência em análise da complexidade de algoritmos e compreensão de algumas classes de complexidade
- Calcular a complexidade de algoritmos com base na complexidade amortizada das funções auxiliares;
- Demonstrar competências no conhecimento classes de complexidade e alguns dos problemas em aberto;

Termos-chave

Complexidade de Algoritmos: Consiste na quantidade de esforço necessário para a sua execução, expressa em função das operações fundamentais, as quais variam de acordo com o algoritmo, e em função do volume de dados.

Tempo de execução: Representa o número de vezes que determinada operação considerada relevante é executada e não representa tempo diretamente.

Análise assintótica do algoritmo: caracteriza a complexidade de tempo numa função de tamanho n.

Notação Big O: denota limites superiores em relação ao tempo de execução dos algoritmos, entretanto, utilizar apenas essa análise do limite superior não é suficiente.

Notação Ω: é o menor tempo de execução em uma entrada de tamanho n.

Notação Θ: baseia-se no maior tempo de execução sobre todas as entradas de tamanho n.

Actividades de Aprendizagem

Actividades 1: Notação Big O (O, Ω e Θ)

Introdução

Para problemas de tamanho pequeno, escolher um algoritmo não é um problema crítico, porque a análise de algoritmos é realizada para valores grandes de tamanho n. Estuda-se o comportamento assintótico das funções de custo (comportamento de suas funções de custo para valores grandes de n). Portanto, o comportamento assintótico de f(n) representa o limite do comportamento do custo quando n cresce. (Ziviane, 2006)

Detalhes da actividade

Seja A um algoritmo para um problema P. A quantidade de tempo que A consome para processar uma dada instância de P depende da máquina usada para executar A. Mas o efeito da máquina se resume a uma constante multiplicativa. Se A consome tempo t numa determinada máquina, consumirá tempo 2t numa máquina duas vezes mais lenta e t/2 numa máquina duas vezes mais rápida. Para eliminar o efeito da máquina, considera-se o consumo de tempo de A ignorando as constantes multiplicativas. Existem três notações principais na análise assintótica de funções: (Ziviane, 2006)

A notação O - grande

A notação Ω - ómega

A notação Θ - teta

Procura-se exprimir o consumo de tempo de um algoritmo de modo que não depende da linguagem de programação, nem dos detalhes de implementação, nem do computador utilizado. Para conseguir-se isto é preciso introduzir uma maneira de comparar as funções utilizando a dependência entre o consumo de tempo de um algoritmo e o tamanho de sua entrada. Essa comparação só leva em conta a “velocidade de crescimento” das funções.

Assim, ela despreza fatores multiplicativos (pois a função 〚2n〛^2 , por exemplo, cresce tão rápido quanto) e despreza valores pequenos do argumento (a função 〚2n〛^2 cresce mais rápido que , embora seja menor que quando é pequeno). Esta forma de comparação de funções se designa assintótica. A comparação é feita de três formas nomeadamente de "≥ ", de ≤", e de "=".

Analisando a expressão n+10 ou n^2+1, pensa-se automaticamente em valores pequenos para n isto é, mais próximos de zero. No entanto, a análise de algoritmos funciona de outra forma, ignora os valores pequenos e toma em conta os valores maiores de . Para esses valores, as funções: n^2, 9999n^2,n^2/1000,n^2+100n, etc., crescem todas com a mesma velocidade e portanto, são equivalentes. Portanto, a área de matemática interessada somente em valores grandes de n, é chamada assintótica. Na análise assintótica, as funções são classificadas em "ordens"; todas as funções de uma mesma ordem são denominadas equivalentes. (de Sousa, 2012)

Notação O grande (Big O)

Segundo um trabalho realizado na universidade de São Paulo na disciplina de Análise de Algoritmos (www.ime.usp.br/~pf/analise_de_algoritmos/aulas/Oh.html), convém restringir a atenção a funções assintoticamente não negativas, ou seja, as funções f tais que para todo n suficientemente grande. Mais explicitamente: f é assintoticamente não negativa se existe no tal que f(n)0 para todo n>no. A notação O define um limite superior para a função, por um factor constante. Diz-se que se existirem constantes positivas c e tais que, para nno o valor de f(n) é menor ou igual a Deste modo, diz-se que é um limite assintótico superior para

Matematicamente escreve-se:

Operações com a notação O grande

f(n)=O(g(n)), c>0 e n_0 f(n)cg(n),nno

Escreve-se f(n)=O(g(n)) para denotar que g(n) domina assintoticamente f(n) e lê-se é de ordem no máximo g(n).

Exemplos:

1.Seja f(n)=〚(n+1)〛^2

Então f(n) é 〚O(n)〛^2 quando n_0=1 e c=4 uma vez que (〚n+1)〛^2   para n 1.

2.Seja n2+10n = O(n2) considerando que f(n)=O(g(n)) e f(n)cg(n)

Prova: se n10 então n2+10n2n2

n2+10n2n2 para todo o n10

Para saber se 2 e 10 são bons valores ou não para c e n então queremos nn ^2 +10ncn2 dividindo por n2, teremos 1+10nc

se n10 então1+10n2 logo, basta escolher os seguintes valores c2   n10c ≥ 2

Suponha que f(n)=2n2+3n+4eg(n)=n2.

Observe que 2n2+3n+42n2+3n2+4n2=9n2 desde que n1

Então f(n) g(n) para todo n1

Portanto, f(n)=O(g(n))   f(n)=O(g(n)) e f(n)cg(n)

Exemplo de aplicação:

$$
\begin{aligned}
f(n) &= O(f(n)) \\
c \times O(f(n)) &= O(f(n)) \quad c = constante \\
O(f(n)) + O(f(n)) &= O(f(n)) \\
O(O(f(n)) &= O(f(n)) \\
O(f(n)) + O(g(n)) &= O(max(f(n), g(n))) \\
O(f(n))O(g(n)) &= O(f(n)g(n)) \\
f(n)O(g(n)) &= O(f(n)g(n))
\end{aligned}
$$

Fonte: Tabela1: Operações com O grande. Fonte: (Ziviane, 2006)

A regra de soma O(f(n))+O(g(n))

Suponha três trechos cujos tempos de execução são O(n), O(n2) e O(n log n).

O tempo de execução dos dois primeiros trechos é O(max(n,n2)) que é O(n2).

O tempo de execução de todos os trechos é O(max(n2, nlog n)) que é O(n2).

O produto de log n+k+O(1n) por n+O(n) é nlog n+kn+O(nlog n)

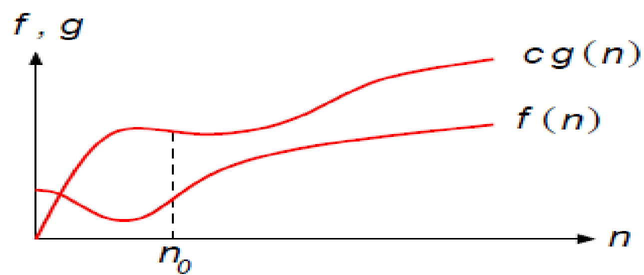
As terminologia de classes mais comuns de funções são:

- Logarítmica - O(log n)
- Linear - O(n)
- Quadrática - O(n2)
- Polinomial – O(nk) , k1
- Exponencial – O(an), com a>1

Função	Designação
c	Constante
log n	Logaritmo
log2n	Logarítmo quadrado
n	Linear
n log n	n log n
n2	Quadrática
n3	Cúbica
2n	Exponencial

Tabela 2: Classes de complexidade mais comuns

Gráfico de f(n)= O(g(n))



$$f(n) = O(g(n))$$

Gráfico 1: Notação O grande. Fonte: (Ziviane, 2010)

Notação Theta

A notação theta limita a função por fatores constantes.

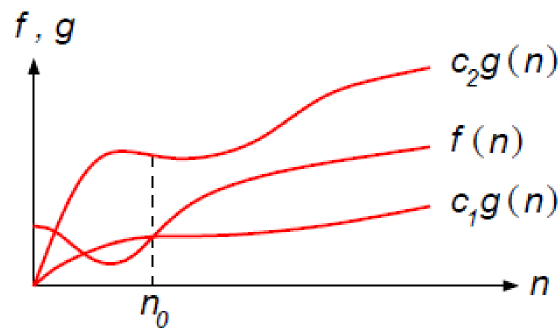
Escreve-se f(n) = (g(n)), se existirem constantes positivas c1, c2 e n0

tais que para n n0, o valor de f(n) está sempre entre c1g(n) e c2g(n)

inclusive.

Neste caso, pode-se dizer que g(n) é um limite assintótico firme (em inglês,

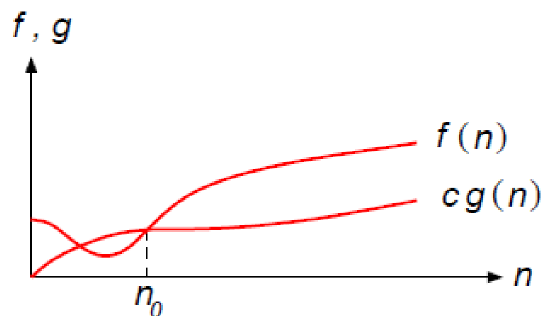
asymptotically tight bound) para f(n)



$$f(n) = \Theta(g(n))$$

Gráfico 2: Notação Theta. Fonte: (Ziviane, 2010)

Notação Omega



$$f(n) = \Omega(g(n))$$

Gráfico 3: Notação Ômega. Fonte: (Ziviane, 2010)

Comparação de programas

A avaliação de programas pode ser feita pela comparção das funções de complexidade, não considerando as constantes de proporcionalidade. Um programa com tempo de execução O(n) é melhor que outro com tempo O(n⟦^2⟧) . No entanto, as constantes de proporcionalidade podem alterar esta consideração.

Exemplo: um programa leva 100n unidades de tempo para ser executado e outro leva ⟦2n⟧^2. Qual dos dois programas é melhor?

Depende do tamanho do problema.

Para n<50, o programa com tempo 2n⟦^2⟧ é melhor do que o que possui tempo 100n

Para problemas com entrada de dados pequena é preferível usar o programa cujo tempo de execução é ⟦O(n)⟧^2. Entretanto, quando cresce, o programa com tempo de execução ⟦O(n)⟧^2 leva muito mais tempo que o programa .

Classes de comportamento assimptótico

Complexidade Constante: f(n)=O(1)

O uso do algoritmo independe do tamanho de . As instruções do algoritmo são executadas um número fixo de vezes.

Complexidade Logarítmica: f(n)=O(log n)

Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande.

Dado um logaritmo de base 2.

Para n=1000,log_2 10

Para n=1000000,l〖og〗_2 20

Exemplo: algoritmo de pesquisa binária

Complexidade Linear: f(n)=O(n)

Em geral, um pequeno trabalho é realizado sobre cada elemento de entrada.

Esta é a melhor situação possível para um algoritmo que tem que processar ou produzir elementos de entrada/saída.

Cada vez que n dobra de tamanho, o tempo de execução também dobra.

Exemplos:

Algoritmo de pesquisa sequencial.

Algoritmo para teste de linearidade de um grafo.

Complexidade Linear Logarítmica: f(n) = O(n log n)

Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois agrupando as soluções. Caso típico dos algoritmos baseados no paradigma divisão-e-conquista.

Seja 2 a base do logaritmo:

Para n = 1 000 000, 〖log 〗_2 20 000 000.

Para n = 2 000 000, 42000000..

Exemplo:

Algoritmo de ordenação MergeSort.

Complexidade Quadrática f(n) = O(n2)

Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro do outro.

Para n = 1 000, o número de operações é da ordem de 1 000 000. Sempre que n dobra o tempo de execução é multiplicado por 4. Os algoritmos deste tipo são úteis para resolver problemas de tamanhos relativamente pequenos.

Exemplos:

Algoritmos de ordenação simples como seleção e inserção.

Complexidade Cúbica f(n) = O(n^3)

Os algoritmos de complexidade cúbica geralmente são úteis apenas

para resolver problemas relativamente pequenos. Para n = 100, o número de operações é da ordem de 1 000 000. Sempre que n dobra o tempo de execução é multiplicado por 8.

Exemplo:

Algoritmo para multiplicação de matrizes.

Complexidade Exponencial: f(n) = O(2^n)

Algoritmos desta ordem de complexidade não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa a força bruta para resolvê-los. Para n = 20, o tempo de execução é cerca de 1 000 000. Sempre que n dobra o tempo de execução fica elevado ao quadrado.

Exemplo:

Algoritmo do Caixeiro Viajante

Complexidade Exponêncial: f(n) = O(n!)

Um algoritmo de complexidade O(n!) é dito ter complexidade exponencial, apesar de O(n!) ter comportamento muito pior do que O(2n) . Geralmente ocorrem quando se usa força bruta na solução do problema.

Seja n=20, temos que 20! = 2432902008176640000 é um número com 19 dígitos.

Se n=40 temos um número com 48 dígitos.

Quadro comparativo de funções de complexidade

Função de custo	Tamanho n					
	10	20	30	40	50	60
n	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
n^2	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
n^3	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
n^5	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
2^n	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 s
3^n	0,059 s	58 min	6,5 anos	3855 s	10^8 s	10^{13} s

Tabela 3: Comparação de funções de complexidade (Ziviane, 2010)

Conclusão

Deve-se preocupar com a eficiência de algoritmos quando o tamanho de n for grande. A eficiência assintótica de um algoritmo descreve a eficiência relativa dele quando n torna-se grande. Portanto, para comparar 2 algoritmos, determinam-se as taxas de crescimento de cada um: o algoritmo com menor taxa de crescimento rodará mais rápido quando o tamanho do problema for grande.

Avaliação

1. O que significa dizer que uma função g(n) é O(f(n))?

2. Indique se as afirmativas a seguir são verdadeiras ou falsas e justifique a sua resposta:

   2^(n+1) = O(2^n).

   2^2n = O(2^n).

   É melhor um algoritmo que requer 2^n passos do que um que requer 〖10n〗^5 passos.

   f(n) = O(u(n)) e g(n) = O(v(n)) => f(n) + g(n) = O(u(n) + v(n))

   f(n) = O(u(n)) e g(n) = O(v(n)) => f(n) - g(n) = O(u(n) - v(n))

3. Suponha um algoritmo A e um algoritmo B com funções de complexidade de tempo a(n) = 〖n 〗^2– n + e b(n) = 49n + 49, respectivamente. Determine quais são os valores de n pertencentes ao conjunto dos números naturais para os quais A leva menos tempo para executar do que B.

Actividades 2: Análise de programas recursivos

Introdução

A recursividade é uma forma interessante de resolver problemas por meio da divisão dos problemas em problemas menores de mesma natureza. Se a natureza dos sub-problemas é a mesma do problema, o mesmo método usado para reduzir o problema pode ser usado para reduzir os sub-problemas e assim por diante.

Detalhes da actividade

As regras gerais de análise de algoritmos permitem-nos analisar programas e com estruturas de controle convencionais.

```
void Burbuja( int A[], int n)
{
   int  i, j, temp;
   for( i = 0; i < n-1; i++ )                         ⎫
      for( j = n-1; j >= i+1; j-- )      ⎫            ⎪
         if( A[j-1] > A[j] ) {  ⎫        ⎪            ⎬ O(n)
            temp = A[j-1];      ⎬ O(1)   ⎬ O(n)       ⎪
            A[j-1] = A[j];      ⎪        ⎪            ⎪
            A[j] = temp;        ⎭        ⎭            ⎭
         }
}
```

Figura 2: algoritmo de bolha

A análise de um algoritmo recursivo é feita da seguinte maneira: Em cada procedimento recursivo, associa-se uma função de tempo desconhecida T(n) ou ondem n corresponde o tamanho do problema. Sendo então possível obter-se uma recorrência para T(n), isto é, uma equação de T(n)em termos de T(k) , para vários valores de T(k), tais que, T(n)=... T(k)....

Análise de funções recursivas

Um exemplo mais clássico de funções recursivas é o factorial de um número.

long factorial (long n) if (n>=0) return 1 else return n*factorial (n-1)

T(n)={ c+T(n-1) se n >0

d   se n<= 0

Se n>2 então T(n)= c+ T(n-1)= c+c+T(n-2)= 2c +T(n-2)

Se n > 3 então T(n)= 2c +T(n-2)= 2c+c+T(n-3)= 3c+T(n-3)

De um modo geral, se n > i   então: T(n)= ic+ T(n-i)

Finalmente se então T(n)=(n-1)c+T(1)=(n-1)c +d com esta demonstração se pode concluir que: T(n) é O(n).

Resolução de Recorrências:

Para analisar o consumo de tempo de um algoritmo recursivo é necessário resolver uma recorrência. Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores anteriores da mesma função. (www.ime.usp.br)

Por exemplo, $T(n) = T(n-1) + 3n + 2$ é uma recorrência que dá o valor de $T(n)$ em relação a $T(n-1)$.

Que valores se podem ter de n? Supondo que n tome valores (2,3,4,5,…). Uma recorrência é um algoritmo recursivo que calcula uma função a partir de um valor inicial. Tomando $T(1) = 1$ como valor inicial.

Uma recorrência é satisfeita por muitas funções diferentes, uma para cada valor inicial, mas todas essas funções são, em geral, do mesmo tipo.

As funções que interessam são, via de regra, definidas no conjunto dos números naturais mas podem ser definidas em outros conjuntos (como os naturais maiores que 99, as potências inteiras de 2, as potências inteiras de 1½, os racionais maiores que ½, etc.).

Em suma resolver uma recorrência é encontrar uma fórmula fechada que dê o valor da função diretamente em termos do seu argumento, sendo que a fórmula fechada é uma combinação de polinômios, quocientes de polinômios, logaritmos, exponenciais, etc.

Exemplo:

Seja $T(n) = T(n-1) + 3n + 2$ e suponha que n pertence ao conjunto {2,3,4,…}. Existirá uma infinidade de funções T que satisfazem a recorrência, como sendo a tabela abaixo:

Mas também existe outra função que satisfaz a recorrência:

De modo mais geral, é evidente que para cada número i existe uma (e uma só) função T definida sobre {1,2,3,4,…} que tem valor inicial $T(1) = i$ e satisfaz a recorrência. O ideal é conseguir uma fórmula fechada para a recorrência. Nosso primeiro método consiste em adivinhar e depois verificar por indução.

Para o valor inicial $T(1) = 1$, a solução da recorrência de recorrência é $T(n) = 3n^2/2 + 7n/2 - 4$.

Então:

$$F(n-1) + 3n + 2$$

$$3(n-1)^2/2 + 7(n-1)/2 - 4 + 3n + 2$$

$$(3n^2 - 6n + 3 + 7n - 7 - 8 + 6n \;\; 4)/2$$

$$(3n^2 + 7n - 8)/2$$

$$3n^2/2 + 7n/2 - 4 \,,$$

A fórmula (1.2) está confirmada! (Como a fórmula correta foi adivinhada?) Isso não interessa no momento, mas pode-se dar uma pista: suspeitamos que f(n) é da forma an^2+bn+c e usei a tabela de valores de F(n) para calcular a, b e c.)

Actividades 3: Análise de programas iterativos

Introdução

Um programa iterativo utiliza comandos chamados laços de repetição para o controle do fluxo de execução, como o comando for ou while por exemplo, que dependem de uma condição ser verdadeira ou falsa para iniciar ou interromper a repetição.

Detalhes da actividade

A complexidade de algoritmos iterativos é expressa através de somatórios, para encontrar a complexidade siga os seguintes passos:

- Escolha uma unidade para medir o tamanho da entrada,
- Identifique a operação básica do algoritmo,
- Expresse o número de execuções da operação básica por um somatório,
- Ache uma forma fechada para o somatório,

De (4) derive a ordem de grandeza do algoritmo

Identidades úteis envolvendo somatórios

$$\sum_{i=l}^{u} cx_i = c\sum_{i}^{u} x_i$$

$$\sum_{i=l}^{u}(x_i \pm y_i) = \sum_{i=l}^{u} x_i \pm \sum_{i=l}^{u} y_i$$

$$\sum_{i=l}^{u} 1 = (u - l + 1)$$

$$\sum_{i=1}^{u} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{u-1} i = 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

Exemplo: Determine a complexidade, no pior caso, do algoritmo abaixo que verifica se um array possui elementos distintos entre si.

```
Elementos distintos (A, n)

for i=1 to n-1

        for j=i+1  to

              if A[i]==A[j]

                     return false

return true
```

- Tamanho da entrada = número de elementos do array A.
- Operação básica = comparação (linha 4).
- Contagem de operações:

$$C(n) := \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} [n-(i+1)+1] = \sum_{i=1}^{n-1} (n-i)$$
$$:= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n \sum_{i=1}^{n-1} 1 - \frac{n(n-1)}{2}$$
$$:= n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Conclusão

A presente atividade visava fazer uma análise de programas recursivos e iterativos. A ilustração de exemplos de programas recursivos e iterativos, possibilitou ao aluno a fazer a diferenciação entre algoritmos recursivos e iterativos.

Avaliação

1. Calcule a soma de todos os valores de um array de reais, usando a recursividade
2. Usando iteratividade encontre o maior elemento de um array de inteiros.

Resumo da Unidade

Na presente unidade trabalhamos os conceitos de complexidade de um algoritmo e técnicas de análise um algoritmo. A noção de complexidade de algoritmos foi abordada, as notações big O, Ômega e Theta, complexidade de algoritmos recursivos e o conceito de algoritmos ótimos.

Chegamos a conclusão que um algoritmo recursivo nem sempre é o melhor caminho para se resolver um problema. No entanto, a recursividade muitas vezes torna o algoritmo mais simples. Vários exemplos ilustrativos também foram mostrados.

Avaliação da Unidade

1. Analise criticamente a seguinte função recursiva, cujo objetivo é de encontrar um elemento máximo de um vetor v[0..n-1]. (3 valores)

int maximoR1 (int n, int v[])

```
{
   int x;
   if (n == 1) return v[0];
   if (n == 2) {
      if (v[0] < v[1]) return v[1];
      else return v[0];
   }
   x = maximoR1 (n-1, v);
   if (x < v[n-1]) return v[n-1];
   else return x;
}
```

2. Que tipo de problemas um algoritmo recursivo é capaz de resolver? (2 valores)
3. O tempo de execução de um algoritmo A é descrito pela recorrência T(n) = 7T(n/2)+n2. Um outro algoritmo B tem um tempo de execução descrito pela recorrência F(n) = aF(n/4)+n2. Qual é o maior valor inteiro de a tal que B é assintoticamente mais rápido que A? (3 valores)
4. Determine a notação assintótica O para as funções seguintes: (1 valor X 4)

   f(x)=⟦3x⟧^2+2x+5

   f(x)=2x+logx+1

   f(x)=542

   f(x)=n(n-1)/2
5. Coloque em ordem crescente as seguintes funções: (1.5 valores X 2)

   2^n,n!,10000,nlog(n)

   nlog(n),n+n^2+n^3,√n

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso

- http://www.inf.puc-rio.br
- http://homepages.dcc.ufmg.br
- http://www.caelum.com.br
- http://www2.dcc.ufmg.br/livros/algoritmos

Unidade 2. Técnica de Desenho de Algoritmos

Introdução à Unidade

Um problema fica bem caracterizado quando responde ás questões tais como quais são os possíveis dados de entrada, quais são os resultados esperados e quando um resultado é uma resposta aceitável para um dado?

Nesta unidade, iremos discutir os algoritmos gulosos e programação dinâmica. Estes métodos baseiam-se na ideia de decomposição de problemas complexos em outros mais simples, cujas soluções são combinadas para fornecer uma resposta ao problema original, á semelhança do método de divisão e conquista.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Escolher e aplicar uma técnica de desenho de algoritmos adequada à resolução de um problema concreto;
- Aplicar as principais técnicas avançadas de desenho de algoritmos, nomeadamente, programação dinâmica e amortização;

Termos-chave

Pesquisa exaustiva (força bruta): É uma das técnicas de resolução de problemas que consiste em gerar todas as possíveis soluções de um problema e verificar qual delas é de fato a solução procurada.

Divisão e conquista: Este método reflete a estratégia militar de dividir o exército adversário para vencer cada uma das partes facilmente.

Programação Dinâmica: é uma técnica algorítmica, normalmente usada em problemas de optimização, que é baseada em guardar os resultados de subproblemas, em vez de os recalcular.

Actividades de Aprendizagem

Actividade 1: Pesquisa Exaustiva (força bruta)

Introdução

Em ciência de computação, força bruta é um algoritmo trivial mas de uso geral que consiste em enumerar todas as soluções possíveis e verificar se cada uma satisfaz o problema. Possui uma implementação muito simples, e encontra sempre uma solução se ela existir. Entretanto, o seu custo computacional é proporcional ao número de possíveis soluções, que, em problemas reais, tende a crescer exponencialmente. Portanto, a força bruta é tipicamente usada em problemas cujo tamanho é limitado, ou quando há uma heurística usada para reduzir o conjunto de soluções para uma espaço aceitável. Também pode ser usado quando a simplicidade da implementação é mais importante que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem em sérias consequências.

Detalhes da actividade

A técnica de força bruta é uma técnica de programação mais fácil de se aplicar. Apesar desta técnica raramente gerar algoritmos eficientes, é uma importante técnica para análise de algoritmos e é aplicável a uma ampla variedade de problemas. Pode ser empregada em diversas situações comuns porém importantes, como encontrar o maior elemento de uma lista ou somar N números. É uma das alternativas válidas quando se deseja resolver um problema "pequeno" através de um algoritmo "simples e direto". Para alguns problemas, a "força bruta" gera algoritmos razoáveis, práticos e sem limitações no tamanho da instância (Ex.: String matching, multiplicação de matrizes).

Qual a forma mais direta de encontrar um elemento em uma lista? Uma das soluções mais utilizadas é a utilização do algoritmo de busca sequêncial. A busca sequencial é a forma mais simples de busca, é aplicável a uma tabela organizada como um vetor ou como uma lista encadeada. É uma técnica mais simples que existe e consiste em percorrer registro por registro em busca da chave.

Exemplo: Algoritmo de busca sequencial em um vetor A, com n posições (0 até n -1), sendo x a chave procurada:

```
for (i=0; i<n; i++)

if (A[i]==x)

return(i); /*chave encontrada*/

else

return(-1); /*chave não encontrada*/
```

Uma maneira de tornar o algoritmo mais eficiente é usar um sentinela que consiste em adicionar um elemento de valor x no final da tabela, pois garante que o elemento procurado será encontrado, o que elimina uma expressão condicional, melhorando a performance do algoritmo:

```
A[n]=x;

for (i=0; x!=A[i]; i++);

if (i < n) return(i); /*chave encontrada*/

else

return(-1); /*chave não encontrada*/
```

Algoritmo 1: Pesquisar o elemento Elem no vector V de tamanho tam

```
k ← -1 // significa que Elem ainda não foi encontrado em V

i ←  0 // índice dos elementos do vector V

Enquanto (i < tam) e (k = -1) Fazer

Se (V[i] = Elem) então

k ← i

senão

Se (V[i] < Elem) então

i ← i + 1

senão

k ← -2; // significa que Elem não está em V

Se (k ≥ 0) então
```

Elem encontra-se na posição k

senão

Elem não se encontra em V

Algoritmo 2: Pesquisar o elemento Elem no vector V de tamanho tam

int PesquisaSequencial (int Elem, int V[], int tam)

```
{
int i = 0;
while ( (i < tam) && (V[i] < Elem) )
i = i + 1;
if ( (i < tam) && (Elem == V[i]) )
return (i);
else
return (-1);
}
```

Tipicamente uma solução por busca exaustiva ou força bruta é composta por duas funções, nomeadamente, uma que gera todas as possíveis soluções e outra que verifica se a solução gerada atende ou não ao problema.

Um dos problemas que se podem encontrar com a busca exaustiva é da possibilidade de existência de um número muito grande de soluções.

Algoritmo 1: Pesquisar o elemento Elem no vector V de tamanho tam

```
k← -1 // significa que Elem não foi encontrado em V
i←0
Enquanto (i < tam) e (k = -1) Fazer
Se (V[i] = Elem)  então
 k← i
senão
i←i + 1
Se (k = -1) então
```

Elem não se encontra em V

senão

Elem encontra-se na posição k

Algoritmo 2: int PesquisaExaustiva (int Elem, int V[], int tam)

```
{
int i = 0, k = -1; // k = posição onde se encontra Elem em V
while ( (i < tam) && (k == -1) )
if (Elem == V[i])
k = i;
else
i = i + 1;
if (k == -1)
 return (-1);
}
```

Exemplo de modelação de busca pelo maior elemento de um vetor como uma busca exaustiva, da seguinte forma:

Soluções possíveis: cada elemento do vetor

Verificação: designar de MAX a melhor solução encontrada até o momento e de x, a solução que está sendo verificada no momento. Se x for igual a MAX, então MAX passa a valer X.

Complexidade de busca sequencial

A utilização do vetor apresenta várias desvantagens tais como, o seu tamanho é fixo, o que pode desperdiçar ou faltar espaço. A alternativa ao uso de vetores é a utilização de listas. Numa busca sequencial, se o registro for o primeiro, existirá apenas uma comparação, se for o último, existirão n comparações. Se for igualmente provável que o argumento apareça em qualquer posição da tabela, em média: (n+1)/2 comparações. Se a busca for mal sucedida, haverá n comparações, logo a busca sequencial no pior caso será O(n).

Actividade 2: Divisão e Conquista

Introdução

O paradigma Divisão e Conquista consiste em dividir o problema a ser resolvido em partes menores, encontrar soluções para as partes, e então combinar as soluções obtidas em uma solução global. ZIVIANI (2007) cita "o uso do paradigma para resolver problemas nos quais os sub problemas são versões menores do problema original geralmente leva a soluções eficientes e elegantes, especialmente quando é utilizado recursivamente".

A Divisão e Conquista empregam modularização de programas e frequentemente conduz a um algoritmo simples e eficiente. Esta técnica é bastante utilizada em desenvolvimento de algoritmos paralelos, onde os sub problemas são tipicamente independentes um dos outros, podendo assim serem resolvidos separadamente.

Detalhes da atividade

O método de desenvolvimento de algoritmos por divisão e conquista reflete a estratégia

militar de dividir o exercito adversário para vencer cada uma das partes facilmente. Dado um problema, de tamanho n, o método divide-o em k instancias disjuntas ($1<k\leq n$).

que corresponde a k subproblemas distintos. Cada subproblema é resolvido separadamente e então combina as soluções parciais para a obtenção da solução da instancia original. Em geral, os subproblemas resultantes são do mesmo tipo do problema original e neste caso, a aplicação sucessiva do método pode ser expressa naturalmente por um algoritmo recursivo, isto é, em um algoritmo denominado x, que tenha um dado de entrada de tamanho n, usamos o próprio x, k vezes, para resolver as subentradas menores do que n.

Segundo FIGUEIREDO (2011), a técnica de Divisão e Conquista consistem em 3 passos:

- Divisão: dividir a instância do problema original em duas ou mais instância menores, considerando-as como sub problemas.
- Conquista: resolver cada sub problema recursivamente.
- Combinação: combinar as soluções encontradas em cada sub problema, compondo uma solução para o problema original.

Vantagens

- Indicado para aplicações que tem restrição de tempo.
- É de fácil implementação.
- Simplifica problemas complexos.

Desvantagens

- Necessidade de memória auxiliar.
- Repetição de Subproblemas.
- Tamanho da pilha (número de chamadas recursivas e/ou armazenadas pode causar estouro de memória).

Algumas aplicações

- Multiplicação de inteiros longos.
- Menor distância entre pontos.
- Ordenação rápida (quicksort) e por intercalação (mergesort).
- Pesquisa em árvore binária.

Ordenação por intercalação

Algoritmo de Ordenação por intercalação (MergeSort) é outro exemplo de algoritmo do tipo divisão e conquista, é um algoritmo de ordenação por intercalação ou segmentação.

A ideia básica é a facilidade de criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo divide a sequência original em pares de dados, ordena-as; depois as agrupa em sequências de quatro elementos, e assim por diante, até ter toda a sequência dividida em apenas duas partes.

Como o algoritmo do Merge Sort usa a recursividade, em alguns problemas esta técnica não é muito eficiente devido ao alto consumo de memória e tempo de execução. Apresenta-se em seguida os passos do algoritmo:

- Dividir uma sequência em duas novas sequências.
- Ordenar, recursivamente, cada uma das sequências (dividindo novamente, quando possível).
- Combinar (merge) as subsequências para obter o resultado final.
- A desvantagem deste algoritmo é precisar de uma lista (vector) auxiliar para realizar a ordenação, ocasionando em gasto extra de memória, já que a lista auxiliar deve ter o mesmo tamanho da lista original.

Exemplo ilustrativo:

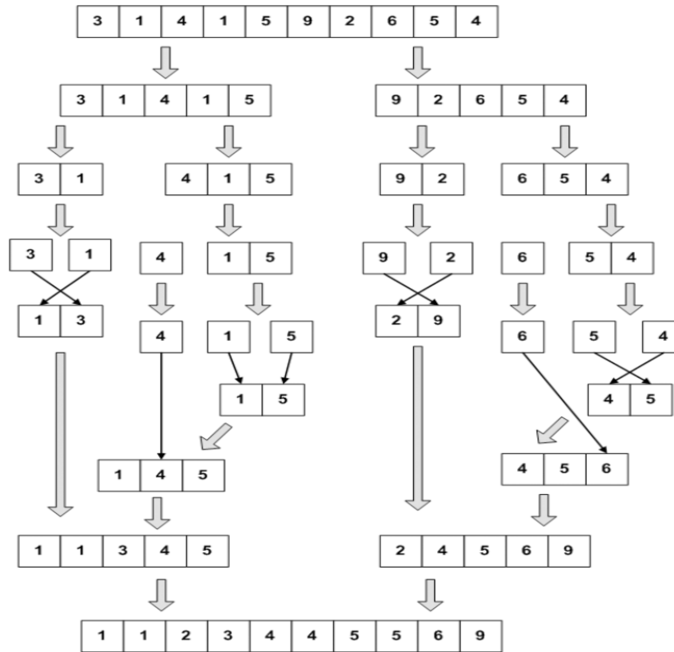


Figura 3 – Técnica de Divisão e Conquista (FIGUEIREDO, 2011)

Algoritmo MERGESORT (L, ini, fim)

ENTRADA: um vetor L e as posições ini e fim

SAÍDA: o vetor L em ordem crescente da posição ini até a posição fim

inicio

```
se ini < fim

meio = ( ini + fim) /2 // divisão inteira

se ini < meio

MERGESORT(L, ini, meio)

se (meio + 1) < fim

MERGESORT(L, meio + 1, fim)

MERGE(L, ini, meio, fim)

fim {MERGESORT}
```

PROCEDIMENTO MERGE (L, ini, meio, fim)

ENTRADA: inteiros: ini, meio, fim; Vetor L[ini..fim] // L[ini..meio] = primeira série ordenada

// L[meio+1..fim] = segunda série ordenada

SAÍDA: Registro L com uma única série ordenada // L[ini..fim] = série intercalada /ordenada

Inicio

```
i = i n i ,  k = 1,  j = meio + 1
enquanto ( i ≤ meio e j ≤ f im )
se ( L[ i ] ≤ L[j ] )
S[k] = L[i ]
i = i + 1
senão
S[k] = L[j ]
j = j + 1
k = k + 1
// fim se
// fim enquanto
se ( i > meio )
p = j
q = f im
senão
p = i
q = meio
para i = p até q
S[k] = L[i ]
k = k + 1
/ / fim para
L[ini . . f im] = S[1. . ( f im-ini+1) ]
retorna (L)
f im {MERGE}
```

Conclusão:

A maior parte dos algoritmos de ordenação funciona pela comparação dos dados que estão sendo ordenados. Determina-se um pedaço de dados e denota-se por chave e, é a partir da chave que os dados são comparados e consequentemente ordenados. Os algoritmos de ordenação normalmente são julgados por sua eficiência. Neste caso, refere-se a eficiência, como sendo a eficiência algorítmica à medida que o tamanho de dados de entrada cresce. A maior parte dos algoritmos em uso tem uma eficiência algorítmica de O (n 2) ou de O(n *log (n)).

Avaliação

1. Explique como funciona o método de desenvolvimento de algoritmos denominado de "Divisão e conquista"?

2. O método de desenvolvimento de algoritmos "divisão e conquista" pode ser utilizado para desenvolver algoritmos recursivos? Porque?

Actividades 3: Programação dinâmica

Introdução

A Programação Dinâmica (PD) pode ser caracterizada como um processo sequencial de tomada de decisões, onde uma decisão ótima global pode ser obtida através da otimização de sub-problemas (ou ótimos locais) individuais, seguindo o "princípio de optimalidade de Bellman" (DIAS et al, 2010).

Detalhes da actividade

A programação dinâmica resolve um problema, dividindo-o em sub-problemas menores, solucionando-os e combinando soluções intermediárias, até resolver o problema original. A particularidade desta metodologia diz respeito à divisão do problema original: o problema é decomposto somente uma vez e os sub-problemas menores são gerados antes dos sub-problemas maiores. Dessa forma, esse método é claramente ascendente (problemas recursivos são descendentes). Esta metodologia tem bastante aplicação na multiplicação de matrizes e em projeto de sistemas confiáveis. (www.ft.unicamp.br)

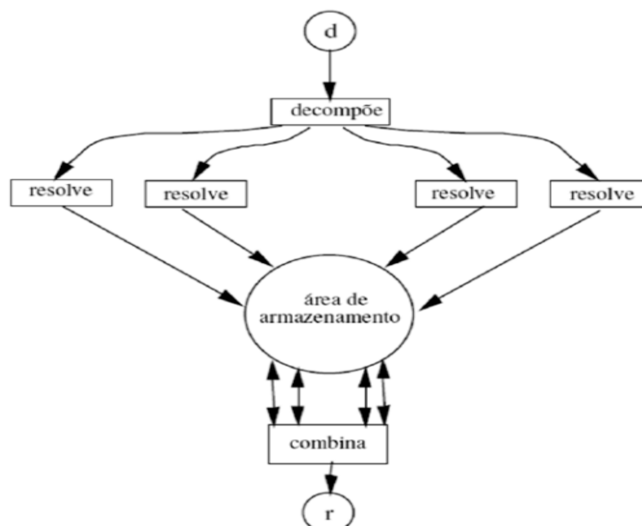


Figura 4: Estrutura Geral da Programação Dinâmica. (Munari e Augusto, 2007)

Na inicialização, a entrada é decomposta em partes mínimas para as quais são obtidas respostas directas, a cada iteração vai aumentando tamanho das partes e obtendo respostas correspondentes a partir das já geradas, até que as partes atingem o tamanho da entrada original.

Quando então sua solução é recuperada e o processo finalizado.Considere um problema de multiplicação de cadeia de matrizes. O problema trata de cálculo do produto: $M = M1 \times M2 \times M3 \times ...Mn$

Um algoritmo de multiplicação de uma matriz $p \times q$ por outra $q \times r$ derivado da fórmula de multiplicação de matrizes requer $p \times q \times r$ multiplicações de elementos. Utilizando o conceito de programação dinâmica e sabendo-se que a multiplicação de matrizes é associativa, esse número reduz bastante.

Considere o exemplo: (entre parênteses é dada a dimensão da matriz)

$M = M1\ (100 \times 3) \times M2\ (3 \times 10) \times M3\ (10 \times 50) \times M4\ (50 \times 30) \times M5\ (30 \times 5)$

Calcular M por meio de $\{[(M1 \times M2) \times M3] \times M4\} \times M5$ resulta em 218000 operações.

Agrupando-se da forma $M1 \times \{M2 \times [(M3 \times M4) \times M5]\}$, a quantidade de operações reduz para 18150. A questão está em determinar a sequência ótima de multiplicações, cujo número de possibilidade de agrupamentos é grande.

Um algoritmo de programação dinâmica decompõe o problema em partes menores, guarda valores intermediários de produtos (na diagonal de uma matriz) e, por último, combinaria a saída dessas multiplicações.

Algoritmo do produto de matrizes em programação dinâmica.Algoritmo PD_Produto_Matrizes

ENTRADA: um vetor P contendo as ordens das n matrizes

SAÍDA: matrizes M e S contendo a solução

inicio

```
para i =1 até n
M[i , i ]=0
para L= 2 até n
para i= 1 até n-L+1
j = i + L - 1
M[i , j ] = ∞
para k= i até j -1
q = M[ i ,k] + M[k+1, j ] + P[i ]*P[k]*P[j ]
se q < M[ i , j ]
M[i , j ] = q
```

```
S[i , j ] = k

return M[1,n] , S

fim {PD_Produto_Matrizes}
```

Vantagens da Programação Dinâmica

- Pode ser utilizada num grande número de problemas de otimização discreta.
- Não necessita de muita precisão numérica.
- Útil para aplicar em problemas que exigem teste de todas as possibilidades.

Desvantagens

- Necessita de grande espaço de memória
- A complexidade espacial pode ser exponencial

Exemplo – Escreva um algoritmo (ou uma sequência de passos) para resolver o problema do troco. Esse problema consiste em: dado um montante X qualquer, que corresponde ao troco devido, e sabendo que existem moedas de 50 centavos, 1, 5 e 10 meticais deve-se pagar o troco com a menor quantidade possível de moedas.

Conclusão

A importância da programação dinâmica reside em resolver um problema através da divisão em pequenos problemas, de modo a facilitar a sua resolução, uma vez que o tamanho do problema também foi reduzido. Ela constitui uma ferramenta muito poderosa para resolver problemas de otimização em itens ordenados da esquerda para a direita.

Avaliação

1. Qual a melhor maneira de se fazer o produto entre matrizes.
2. Escrever um algoritmo para achar o produto de menor custo entre n matrizes, usando programação dinâmica.

Resumo da Unidade

Os resultados mostraram que a eficiência das técnicas e o tempo e recursos demandados estão intrinsecamente relacionados aos parâmetros de configuração de entrada. Comparando as técnicas de Divisão e Conquista e Programação Dinâmica percebe-se que no caso da instância com maior número de itens houve o estouro de recursos computacionais, sugerindo que a técnica deve ser utilizada de forma controlada.

Avaliação da Unidade

Verifique a sua compreensão!

Instruções

Nas perguntas que se seguem, deverá responder todas elas com clareza:

1. Encontre o maior e o menor elemento de um array de inteiros, A[1…n], onde n >=1, utilizando a programação dinâmica.
2. Encontrar o índice de um elemento k em uma lista ordenada (assumindo que ele exista).
3. Explique de que forma o de algoritmo "divisão e conquista" pode ser utilizado para desenvolver algoritmos recursivos. Dê exemplo.
4. Faça uma análise comparativa entre os algoritmos divisão e conquista e ordenação por intercalação.
5. Considere um array linear ordenado alfabeticamente. Usando a ordenação linear, quantas comparações são necessárias para encontrar K, P e G?

A	C	D	F	G	J	K	O	P
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8

Avaliação Sistemática

Critérios de Avaliação

Pergunta	Pontuação (máximo 10 valores)
1	3.0
2	1.5
3	1.5
4	2.5
5	1.5

Leituras e outros Recursos

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso.

- http://www.inf.puc-rio.br
- http://homepages.dcc.ufmg.br
- http://www.caelum.com.br
- www.dcomp.sor.ufscar.br
- www.dcc.fc.up.pt

Unidade 3. Algoritmos em Grafos

Introdução à Unidade

Existem muitas aplicações em ciência de computação que necessitam de considerar conjunto de conexões entre pares de objetos:

Situações tais como se existirá ou não um caminho para ir de um objeto a outro seguindo as conexões, qual é a menor distância entre um objeto e outro objeto, quantos outros objetos podem ser alcançados a partir de um determinado objeto, são modelados com uso do tipo abstrato de dados denominado grafo. Nesta unidade, iremos discutir o tipo de dados abstratos denominado grafo.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Demonstrar conhecimentos em estruturas de dados avançadas tais como grafos;
- Implementar os algoritmos fundamentais de grafos;

Termos-chave

Grafo: consiste num conjunto de nós (ou vértices) e num conjunto de arcos (ou arestas).

Pesquisa em largura: estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos vizinhos (que pode ser vazio).

Pesquisa em profundidade: estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos adjacentes (que pode ser vazio).

Árvores de cobertura mínima: Uma árvore de cobertura mínima de um grafo não dirigido G=(V,E) é um sub-grafo de G que é uma árvore e contém todos os vértices de G.

Actividades de Aprendizagem

Actividades 1: Representação de Grafo

Introdução

Os grafos representam um modelo fundamental em computação, surgindo em problemas de caminhos (por exemplo, caminho mínimo ou máximo numa rede de transportes), representação de redes (por exemplo, de computadores, de tráfego rodoviário), descrição de sequência de programas, desenho de circuitos integrados, representação de relações (por exemplo, ordenação, emparelhamento), análise sintática de linguagens (árvores sintáticas). (Tomás, 2013)

Detalhes da actividade:

Um grafo é uma representação gráfica de elementos de dados e das conexões entre alguns destes itens. A árvore é um caso particular de grafo, onde as conexões entre os elementos não são circulares.

A escolha da estrutura de dados certa para a representação de grafos tem um enorme impacto no desempenho de um algoritmo. Existem duas representações usuais, nomeadamente a matriz de adjacências e listas de adjacências. (Lipschutz, Pai, 2008)

Exemplos de grafos temos diagramas de organizações, mapas rodoviários, redes de transporte, redes de comunicação, etc.

Definição: Grafo (graph) é uma tripla ordenada (V, E, f), onde: V é um conjunto não-vazio de vértices (ou nós), onde E é um conjunto de arestas (ou arcos) e f é uma função que associa cada aresta a um par não-ordenado (x, y) de vértices chamados extremos de a.

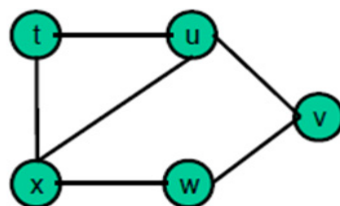
Dado um grafo G = (V, E) , a matriz de adjacências M é uma matriz de ordem |V|x|V|, tal que:

|V| = número de vértices

M[i,j] = 1, se existir aresta de i a j

M[i,j] = 0, se NÃO existir aresta de i a j

Esquemas de grafos



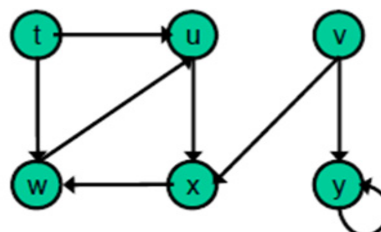


Figura 5: Grafo não orientado e grafo orientado

Exemplo: Do grafo seguinte, qual a sua matriz de adjacências:

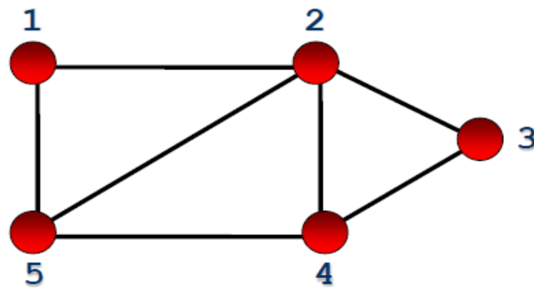


Figura 7: Grafo não direcionado

Em resposta a pergunta acima, temos:

M =

1	2	3	4	5	
0	1	0	0	1	1
1	0	1	1	1	2
0	1	0	1	0	3
0	1	1	0	1	4
1	1	0	1	0	5

vértices

Grafo não direcionado
➔Matriz simétrica

Se o grafo for direccionado, M[i,j] deve indicar ou não a presença de uma aresta divergente de i e convergente em j, ou seja i:=j

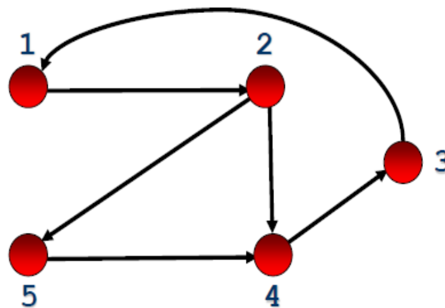


Figura 8: Grafo direccionado

Uma possível resposta seria uma matriz assimétrica seguinte:

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	1
3	1	0	0	0	0
4	0	0	1	0	0
5	0	0	0	1	0

Figura 9: Matriz assimétrica

Se o grafo for valorizado, –M[i,j] deve conter o peso associado com a aresta.

Se não existir uma aresta entre i e j, então é necessário utilizar um valor que não possa ser usado como peso (como o valor 0 ou negativo, por exemplo).

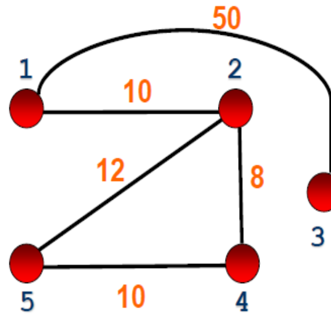


Figura 10: Matriz valorizada

Possível resposta:

	1	2	3	4	5
1	0	10	50	-1	-1
2	10	0	-1	8	12
3	50	-1	0	-1	-1
4	-1	8	-1	0	10
5	-1	12	-1	10	0

Operações em Grafos

- Criar grafo vazio
- Inserir aresta
- Retirar aresta
- Calcular o grau de um vértice (de entrada/saída, se for dígrafo)
- Obter lista de vértices adjacentes a um determinado vértice

Conclusão

Nesta actividade discutimos o conceito de grafo, sua representação, bem como as operações que podem ser realizadas em grafos.

Avaliação

1. Construa uma representação geométrica do grafo G = (V,E), onde:

   V = {1,2,3,4,5,6}

   E = {(1,3), (1,4), (1,5), (2,3),(2,4),(2,5),(3,5),(4,5)}

2. Os amigos João, Pedro, Antônio, Marcelo e Francisco sempre se encontram para botar conversa fora e às vezes jogar dama, xadrez e dominó. As preferências de cada um são as seguintes: João só joga xadrez; Pedro não joga dominó; Antônio joga tudo; Marcelo não joga xadrez e dominó e Francisco não joga nada.

   a. Represente através de um grafo bipartido G=(V,E) todas as possibilidades de um amigo jogar com os demais. Defina V e E.

Actividade 2: Pesquisa em largura e pesquisa em profundidade

Introdução

A pesquisa em largura ou breadth-first search é uma estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos vizinhos que pode ser vazio.

Detalhes da actividade

A pesquisa em largura quando é aplicada a partir de um elemento de origem s, começa por visitar os vizinhos de s, a seguir os vizinhos dos vizinhos de s que ainda não tenham sido visitados, e sucessivamente. Deste modo, esta pesquisa corresponde à visita de um grafo em que os nós representam os elementos do conjunto e os ramos definem a relação de vizinhança. A visita a partir de um nó é feita por ordem crescente de distância: na iteração k do processo, são visitados os nós v que estão a distância k da origem s, sendo a distância dada pelo número de ramos do caminho mais curto de s até v.

Segue abaixo os algoritmos apresentados na forma de pseudocódigo. À esquerda a função BSF_Visit(s, G) para efectuar a busca de s. Á direita uma função análoga BSF_Visit(s, G, Q) para visitar todo o grafo.

```
BFS_VISIT(s, G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    visitado[s] ← true;
    Q ← MKEMPTYQUEUE();
    ENQUEUE(s, Q);
    Repita
        v ← DEQUEUE(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                ENQUEUE(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QUEUEISEMPTY(Q) = true );

ESCREVECAMINHO(v, pai)
    Se pai[v] ≠ NULL então
        ESCREVECAMINHO(pai[v], pai);
        escrever(v);
```

```
BFS(G)
    Para cada v ∈ G.V fazer
        visitado[v] ← false;
        pai[v] ← NULL;
    Q ← MKEMPTYQUEUE();
    Para cada v ∈ G.V fazer
        Se visitado[v] = false então
            BFS_VISIT(v, G, Q);

BFS_VISIT(s, G, Q)
    visitado[s] ← true;
    ENQUEUE(s, Q);
    Repita
        v ← DEQUEUE(Q);
        Para cada w ∈ G.Adjs[v] fazer
            Se visitado[w] = false então
                ENQUEUE(w, Q);
                visitado[w] ← true;
                pai[w] ← v;
    até (QUEUEISEMPTY(Q) = true );
```

Tabela 3: Pesquisa em largura (www.dcc.fc.up.pt)

No algoritmo acima, assumimos que que as variáveis pai[] e visitado[] são globais, embora as pudéssemos ter considerado também como parâmetros da função (o que pode fazer sentido em algumas aplicações).

Pesquisa em profundidade

À semelhança da pesquisa em largura, a pesquisa em profundidade (depth-first search) é uma estratégia genérica de análise exaustiva de um conjunto em que cada elemento tem um conjunto de elementos adjacentes (que pode ser vazio).

Quando aplicada a partir de um nó origem s, começa por visitar um dos adjacentes de s, a seguir um adjacente desse adjacente de s que ainda não tenha sido visitado, e sucessivamente. Quando um nó já não tiver mais adjacentes por visitar, a pesquisa prossegue com a análise de outro adjacente do pai desse nó (que ainda não tenha sido visitado). De seguida apresentamos algoritmo DFS(G) para visita integral do grafo, na versão recursiva.

DFS(G)

Para cada v ∈ G.V fazer

isitado[v]←false;

pai[v]←NULL;

Para cada v ∈ G.V fazer

se visitado[v]=false então

DFS_Visit(v,G);

DFS_Visit(v, G)

visitado[v]←true;;

Para cada w ∈ G.Adjs[v] fazer

se visitado[w]=false então

pai[w]←v;

DFS_Visit(w,G);

Procura a partir de um nó origem. Para determinar apenas os nós acessíveis de um nó origem s, podemos adaptar DFS VISIT para incluir a inicialização das variáveis visitado[ ] e pai[ ], como se fez para BFS_VISIT.

Avaliação

1. Dado um o grafo cujas arestas estão a seguir é representadas por suas listas de adjacência. 0-1 0-2 1-3 1-4 1-5 3-6 3-7 4-7 5-7. Suponha que uma busca em profundidade visita os vértices na seguinte ordem: 0 1 5 7 4 3 6 2. Em que os ordem aparecem os vizinhos de cada vértice nas listas de adjacência?
2. Considere o dígrafo definido pelos arcos 0-1 1-2 1-3 3-4 3-5 1-6 0-7 7-8 7-9. Faça uma busca dfs. Em que ordem os vértices são visitados?

Actividade 3: Árvores de cobertura mínima

Introdução

Na presente atividade iremos discutir o conceito de árvore de cobertura mínima que é um conceito da área de grafos.

As aplicações de árvore geradora mínima são várias tais como numa instalação de linhas telefônicas (ou elétricas) entre um conjunto de cidades utilizando a infra-estrutura das rodovias com o menor uso de material. Outros problemas como análise de clusters e armazenamento de informações, dentre outros podem ser tratados por esta modelagem que possui eficientes algoritmos como Kruskal e Prim.

Detalhes da actividade

Definição: Seja G=(V,E) um grafo conexo onde cada aresta e possui um peso p(e). Denomina-se peso de uma árvore geradora T=(V,ET) de G a soma dos pesos das arestas ET. Uma árvore geradora mínima é a árvore geradora de G que possui peso mínimo dentre todas as árvore geradoras de G.

Como encontrar uma árvore de cobertura?

- Se o grafo G não tem ciclos, G é uma árvore de cobertura.
- Se G tem ciclo, é necessário remover recursivamente as arestas até achar uma árvore, mantendo o grafo conectado

Num grafo, o custo de um sub-grafo (de um grafo não dirigido ponderado) é a soma dos pesos de suas arestas sendo que, uma árvore de cobertura mínima de um grafo não dirigido ponderado é uma árvore de cobertura com menor custo.

Algoritmo Genérico

A ← ∅

enquanto A não é uma árvore de cobertura

faça encontre uma aresta (u,v) que é segura para A

A ← A ∪ {(u,v)}

return A

Algoritmos para encontrar árvores de cobertura mínima

Kruskal: Encontra a aresta segura e adiciona a árvore de cobertura que está sendo formada. A nova aresta não deve necessariamente ter um dos vértices na árvore que está sendo formada. Ou seja, uma floresta pode existir antes da árvore de cobertura mínima ter sido encontrada. (Tomás, 2013)

Prim: Encontra a aresta segura e um dos vértices necessariamente deve pertencer a árvore que está sendo construída. Sempre existe uma única árvore parcial. (Tomás, 2013)

Algoritmo de Kruskal

A←∅

Para cada vértice v ∈ V[G]

faça Make-Set(v)

Ordene as arestas de E (ordem crescente por peso w)

∀ edge (u,v) ∈ E, (considerando a ordem)

Se Find-Set(u) ≠ Find-Set(v)

então A ← A ∪ {(u,v)}

Union(u,v)

return A

Nota: O algoritmo de Kruskal parte de uma floresta em que cada vértice de V está isolado numa árvore sem ramos. Em cada iteração, seleciona um ramo para ligar duas árvores e, consequentemente, quando tiver inserido |V| - 1 ramos, a floresta estará reduzida a uma árvore única.

Algoritmo de Prim

T←∅

B← um nó arbitrário de V

enquanto B≠V faça

Determine { u, v) de menor comprimento tal que u∈V\B e v∈B

T-T U {u,v}

B=B U {u}

fim enquanto

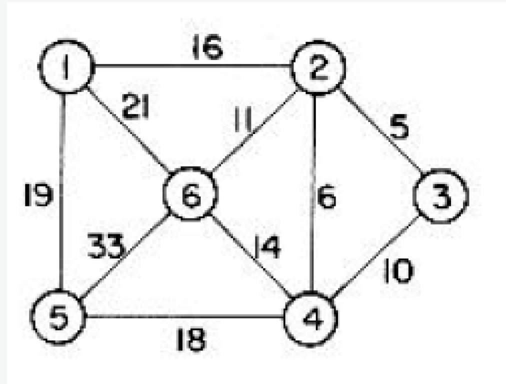
Nota: No algoritmo de Prim, a árvore é construída a partir de um vértice qualquer e, em cada iteração, escolhe um dos vértices que está mais próximo dos vértices que na sub-árvore já construída incluídos e liga esse vértice à árvore.

Conclusão

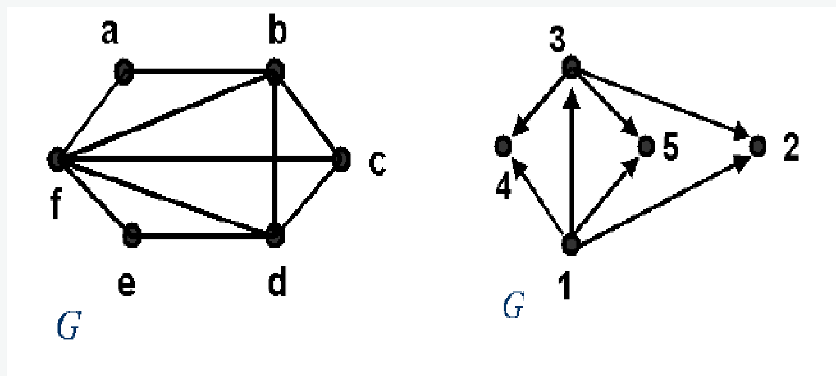
As árvores geradoras mínimas ou árvores de cobertura mínima são largamente utilizadas em problemas de redes com o objetivo de reduzir os custos. Para além disso, a aplicação de árvores geradoras mínimas é notória no campo

Avaliação

1. Dado o grafo abaixo, determine a árvore geradora mínima usando o algoritmo de Prim ou o algoritmo de Kruskal. Apresente o desenvolvimento de sua solução.



2. Utilizando matriz de adjacência, representa os seguintes grafos.



Resumo da Unidade

O conceito de grafo discutido nesta unidade representa um modelo fundamental em computação, surgindo em problemas de caminhos (por exemplo, caminho mínimo ou máximo numa rede de transportes. Os algoritmos de busca em profundidade tem como missão percorrer um único caminho do grafo até onde ele possa chegar (isto é, até visitar um nó sem sucessores ou um nó cujos sucessores já tenham sido visitados, enquanto que o percurso em largura pode ser usado em algumas aplicações também idênticas às do percurso em profundidade mas também pode ser usado para determinar se um grafo não-orientado é conexo e para identificar os componentes conexos do grafo, pode também ser usado para determinar se um grafo é cíclico.

Avaliação da Unidade

Verifique a sua compreensão!

1. Apresente um grafo, com no mínimo 5 vértices. Apresente suas matrizes de adjacência e de incidência. Mostre exemplos de:

   percurso

   caminho (simples)

   trajeto (trilha)

   ciclo

2. Mostre que se $f(n) = O(g(n))$ então $O(f(n)) + O(g(n)) = O(g(n))$.

3. Mostre que para toda árvore geradora mínima $T \subset G$ existe uma ordenação nas arestas de G tal que T é a árvore devolvida pelo algoritmo de Kruskal.

4. Dê exemplos de aplicação de busca em largura e em profundidade.

Avaliação Sistemática

Critérios de Avaliação

Pergunta	Pontuação (máximo 10 valores)
1	3
2	1.5
3	3
4	2
5	2

Unidade 4. Estruturas de Dados Especializados

Introdução à Unidade

As árvores são estruturas de dados extremamente úteis em diversas aplicações, tais como, bases de dados de grandes dimensões, determinação do caminho mais curto entre dois computadores de uma rede, etc. A forma como elas estão organizadas hierarquicamente permite o desenvolvimento de diversas aplicações utilizando-se algoritmos relativamente simples, recursivos e de eficiência bastante razoável.

Na presente unidade iremos discutir uma das mais importantes estruturas de dados na ciência de computação que são as árvores binárias de pesquisa. Para além deste conceito, discutir-se-á também as filas de prioridade e conjuntos disjuntos.

Objetivos da Unidade

Após a conclusão desta unidade, deverá ser capaz de:

- Demonstrar conhecimentos em estruturas de dados avançadas como árvores de busca binária, conjuntos disjuntos
- Implementar as filas de prioridade na resolução de um dado problema

Termos-chave

Filas de prioridade: É uma coleção de elementos tais que a cada elemento é atribuído uma prioridade.

Conjuntos Disjuntos: É uma coleção de conjuntos dinâmicos disjuntos em que cada conjunto é identificado por um representante denominado membro do conjunto.

Árvores binárias de pesquisa: é uma árvore ordenada em que qualquer elemento a pesquisar é único (com a mesma chave de pesquisa).

Actividades de Aprendizagem

Actividades 4.1: Filas de prioridade

Introdução

O conceito de filas de prioridade é encontrado principalmente no campo da programação de computador. Na vida real, podemos ver as filas de prioridade por exemplo para idosos, gestantes, etc. A execução de tarefas é feita por ordem de prioridade. Uma fila de prioridades é uma estrutura de programação crucial para um sistema de gestão de recursos.

Detalhes da atividade

No quotidiano , encontramos as filas de prioridade em diversas aplicações. O seguintes exemplos, são os mais comuns como nas filas do banco existe um esquema de prioridade em que clientes preferenciais, idosos ou mulheres grávidas, pessoas portadoras de deficiência, possuem a mais alta prioridade de atendimento relativamente aos demais clientes Em filas de atendimento para análise de sinais em sistemas de controle, sensores prioritários são adicionados a fila de atendimento com uma maior prioridade que sensores secundários, o que garante o atendimento de sinais prioritários a frente dos demais. (www.facom.ufu.br)

Estrutura de uma fila de prioridade

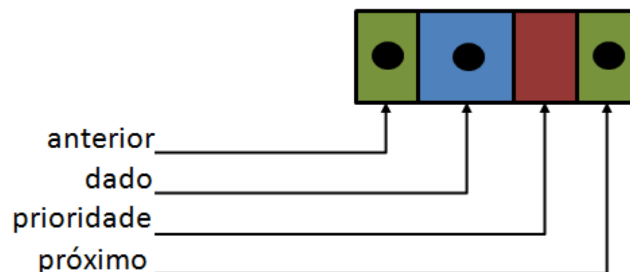


Figura 11: Estrutura de uma fila de prioridade (www.facom.ufu.br)

Em computação, existem dois tipos de filas de prioridade, a saber, a fila de prioridade ascendente e fila de prioridade descendente. (Tenenbaum, 1995).

Uma fila de prioridade ascendente consiste num conjunto de elementos no qual, os tais elementos podem ser adicionados de forma arbitrária. Isto é, a partir do qual apenas o menor item pode ser removido.

Por outro lado, a fila de prioridade descendente assemelha-se a fila de prioridade ascendente, mas só que permite a remoção do maior elemento.

Os elementos de uma fila de prioridade podem ser estruturas complexas, que podem ser classificadas por um ou vários campos. Como pode perceber, não é necessário que os elementos de uma fila de prioridade sejam números ou caracteres que facilmente possam ser comparados. (Tenenbaum, 1995).

Exemplo: as listas telefónicas são constituídas por apelidos, nomes, endereços e números de telefone, e são classificadas pelo apelido.

Operações

Em filas de prioridade prevê-se duas operações básicas que são de facto uma extensão das operações básicas de uma fila normal, nomeadamente, Inserir com prioridade e remover um elemento de mais alta prioridade.

Adicionalmente, versões mais avançadas de filas de prioridade podem requerer ainda outras operações como modificar a prioridade de um dado elemento, número de elementos na fila e testar a existência de elementos de mesma prioridade.

Implementação de uma fila de prioridade em vetores

Das actividades anteriores vimos que, pilha e fila podem ser implementadas num vetor de modo que cada inserção ou eliminação compreenda o acesso a um único elemento do vetor. Nas filas de prioridade isso não é possível. (Tenenbaum, 1995).

Dados n elementos de uma fila de prioridade (fp) mantidos nas posição 0 a n - 1 de um vetor fq.elementos, de tamanho maxfq, e que fq.rear seja igual à primeira posição vazia do vetor, n. Desse modo, fpqinsert(fq,x) será uma operação como mostra o algoritmo seguinte:

```
se (fq.rear >= maxfq)

{

escreva("fila de prioridade cheia");

exit(1);

}      /* fim se */

fq.items[fq.rear] = x;

fq.rear++;
```

Nota: Este método de adicionar um elemento na fila, não permite a manutenção de elementos classificados no vetor.

Este método não oferece grandes problemas no caso de inserções apenas. A dificuldade surge quando queremos realizar as operações de remoção na fila de prioridade ascendente. Neste caso teríamos que considerar duas questões. Primeiro, para localizar o menor elemento, todo elemento do vetor precisa ser examinado. Portanto, uma eliminação exigiria o acesso a cada elemento da fila de prioridade.

Segundo, se o elemento a remover estiver em qualquer posição da fila de prioridade como no meio por exemplo, como eliminar? Vimos que em filas e pilhas, as operações de eliminação envolvem a remoção de um elemento de uma das duas extremidades e não exigem nenhuma busca. A operação de remoção na fila de prioridade, sob essa implementação, requer tanto a busca do elemento a ser removido, assim como a remoção propriamente dita.

Existem várias soluções para esse problema, nenhuma delas totalmente satisfatória. Uma das soluções é encontrar um indicador especial de "vazio" e introduzir numa posição eliminada. Esse indicador pode ser um valor inválido como elemento (por exemplo, -1 numa fila de prioridade de números não-negativos), ou um campo separado pode estar contido em cada elemento do vetor para indicar se ele está vazio.

A operação de inserção ocorre como anteriormente, mas, quando fq.rear alcança maxfq, os elementos do vetor são compactados no início do vetor, e fq.rear é redefinida com um a mais que o número de elementos. (Tenenbaum, 1995).

Algortimo de remoção do primeiro elemento na fila de prioridade

- Set item:=info[start] /**este procedimento guarda os dados no primeiro nó da fila
- Apagar o 1° elemento da lista
- Computar o item
- exit

Algortimo de adicionar um item com prioridade N na fila de prioridade

- Percorrer a lista até encontrar o nó X que contém um número prioritário maior que N.
- Inserir o elemento em frente do nó X
- Se o tal nó não for encontrado, insira o elemento no final da lista
- exit

Conclusão

A filas de prioridade são utilizados para o processamento de múltiplas tarefas, como é o caso de tarefas individuais de funcionamento de um computador. Para além disso, O computador ou o usuário pode, então, aplicar prioridades numéricas para essas tarefas. Tarefas de alta prioridade são executados primeiro. Uma hierarquia de prioridade estrita nem sempre pode determinar qual tarefa é atribuída ao lado, de modo que mesmo as tarefas de baixa prioridade será concluída mesmo se eventualmente há sempre tarefas de maior prioridade de espera. Outras vezes, uma tarefa pode ser removido da fila, então atribuída uma prioridade mais elevada e transferido se passar muito tempo

Avaliação

1. Demonstre como uma sequência de inserções e remoções de uma fila representada por um vetor linear pode provocar estouro ao tentar inserir um elemento numa fila vazia.

2. Demonstre como classificar um conjunto de números de entrada usando uma fila de prioridade e as operações fqinsert, fqmindelete e empty.

Actividades 2: Árvores binárias de pesquisa

Introdução

As árvores são estruturas de dados extremamente úteis em diversas aplicações, como por exemplo, bases de dados de grandes dimensões, determinação do caminho mais curto entre dois computadores de uma rede, etc. A forma como elas estão organizadas hierarquicamente permite o desenvolvimento de diversas aplicações utilizando-se algoritmos relativamente simples, recursivos e de eficiência bastante razoável. Árvore é um tipo abstrato de dados que armazena elementos de uma forma hierárquica.

Detalhes da actividade

A árvore é um conjunto finito de nós, com informação e que têm uma relação entre si do tipo pai-filho. Se a árvore não é vazia, ou seja, se tem nós, então:

- existe um nó especial – a raiz – que não tem pai
- todo o nó da árvore (excepto a raiz) tem um único pai
- um nó pode ter 0, 1 ou mais filhos

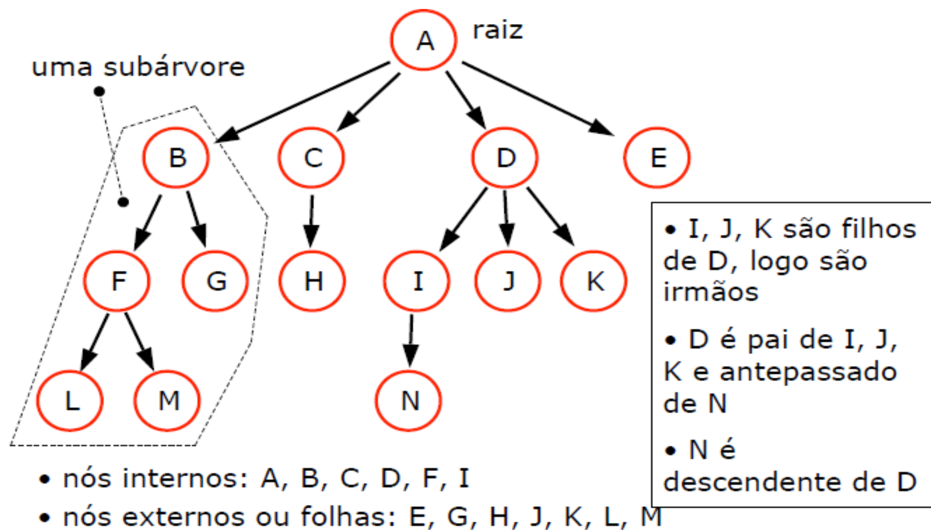


Figura 12: Estrutura de uma Árvore Fonte: (Lipschurtz, 2006)

Árvore binária ordenada tal que:

- todo o nó tem no máximo dois filhos
- um filho é esquerdo ou direito
- na ordem, o filho esquerdo antecede o filho direito

As operações de adicionar no caso de árvores binárias são:

- retornar o filho esquerdo
- retornar o filho direito
- determinar se tem filho esquerdo ou não
- determinar se tem filho direito ou não

Árvore Binária de Pesquisa T (ABP) ou Árvore Binária de Busca

Para uma árvore binária ser considerada uma árvore binária de pesquisa, os valores que vão sendo armazenados nesta devem obedecer à chamada condição de pesquisa. Essa condição será que o valor de um nó deve ser sempre maior que o valor de qualquer nó contido na subárvore esquerda deste e menor que o valor de qualquer nó na sua subárvore direita.

É tal que T= f e a árvore é dita vazia ou seu nó raiz contém uma chave e:

- Todas as chaves da subárvore esquerda são menores que a chave da raiz.
- Todas as chaves da subárvore direita são maiores que a chave raiz.
- As subárvores direita e esquerda são também Árvores Binárias de Busca.

Operações em árvores binárias de pesquisa

As operações básicas em uma Árvore binária de pesquisa são:

- Inserção
- Remoção
- Pesquisa: busca um elemento de chave k na árvore binária de pesquisa com raiz em root. Devolve o nó onde está o elemento, se este existir; senão devolve NULL.

Inserção: A inserção consiste num processo de adicionar um elemento numa árvore.

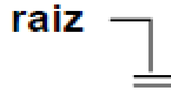
Princípio de funcionamento:

- Procure um "endereço" para inserir o novo nó, começando a procura a partir do nó-raiz
- Para cada nó-raiz de uma sub-árvore, compare; se o novo nó possui um valor menor do que o valor nó raiz (vai para sub-árvore esquerda), ou se o valor é maior que o valor no nó-raiz (vai para sub-árvore direita)

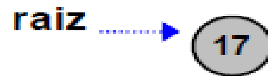
Exemplo:

Considere a inserção do conjunto de números, na sequência 17, 99, 13, 1, 3, 100, 400

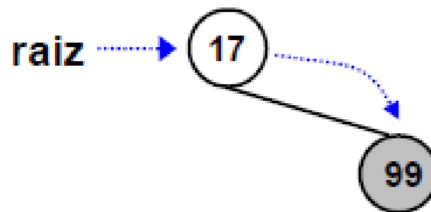
No início a árvore binária de pesquisa está vazia!



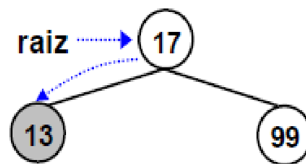
O número 17 será inserido tornando-se o nó raiz;



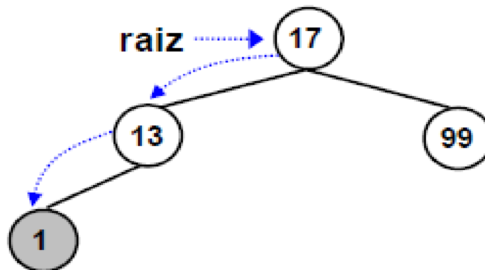
A inserção do 99 inicia-se na raiz. Compara-se 99 com 17. Como 99 > 17, 99 deve ser colocado na sub-árvore direita do nó contendo 17 (sub-árvore direita, inicialmente, nula);



A inserção do 13 inicia-se na raiz. Compara-se 13 com 17. Como 13< 17, 13 deve ser colocado na sub-árvore esquerda do nó contendo 17. Já que o nó 17 não possui descendente esquerdo, 13 é inserido na árvore nessa posição.



5. Para inserir o valor 1, repete-se o procedimento. 1<17, então será inserido na sub-árvore esquerda. Chegando nela, encontra-se o nó 13, 1<13, então ele será inserido na sub-árvore esquerda de 13.



Em suma este processo de comparação entre o novo elemento e a razão repete-se até que todos os elementos estejam inseridos.

Remoção de um nó numa ABP: Procura-se o nó que contém o elemento a eliminar e o nó pai desse elemento. A forma em que o nó e apagado depende de números de filhos que o nó contém. Desta maneira, para a remoção de um nó em uma árvore binária, devem ser considerados três casos nomeadamente:

Caso 1: nó é folha. O nó pode ser retirado sem problema;

A remoção do nó 1, decorre facilmente pois não requer ajustes posteriores. Os nós 30, 80 e 100 também podem ser removidos sem problemas!

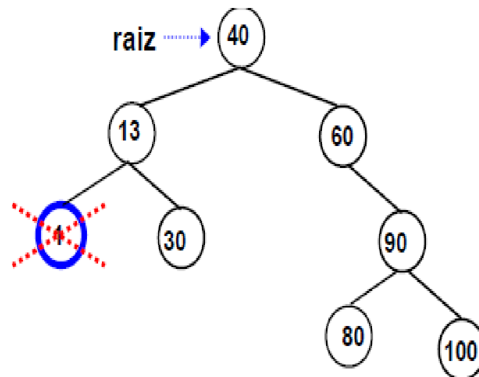


Figura 13: Remoção de um nó folhaFonte: (Lipschurtz, 2006)

Caso 2: o nó possui uma sub-árvore (esq./dir). O nó-raiz da sub-árvore (esq./dir.) "ocupa" o lugar do nó retirado. Como ele possui apenas a sub-árvore direita, o nó contendo o valor 90 pode "ocupar" o lugar do nó removido.

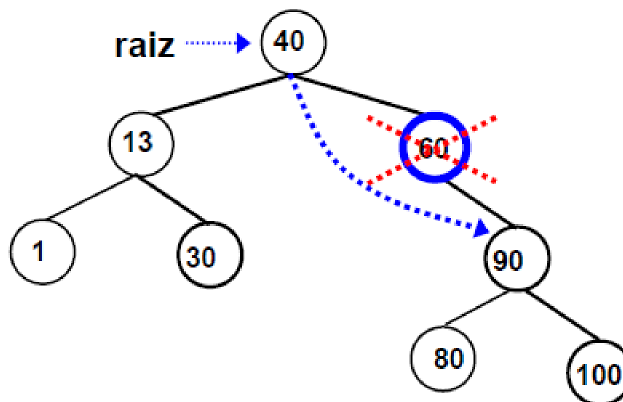


Figura 14: Remoção do nó com um filho Fonte: (Lipschurtz, 2006)

Caso 3: o nó possui duas sub-árvores. O nó contendo o menor valor da sub-árvore direita pode "ocupar" o lugar; ou o maior valor da sub-árvore esquerda pode "ocupar" o lugar. Neste caso, existem 2 opções: O nó com valor 30 pode "ocupar" o lugar do nó-raiz, ou o nó com valor 60 pode "ocupar" o lugar do nó-raiz.

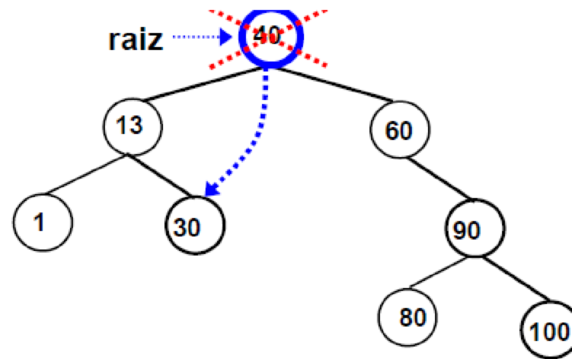


Figura 15: Remoção de nó raiz Fonte: (Lipschurtz, 2006)

Este caso também se aplica ao nó 90. O nó com valor 80 pode "ocupar" o lugar do nó-raiz, ou o nó com valor 100 pode "ocupar" o lugar do nó-raiz.

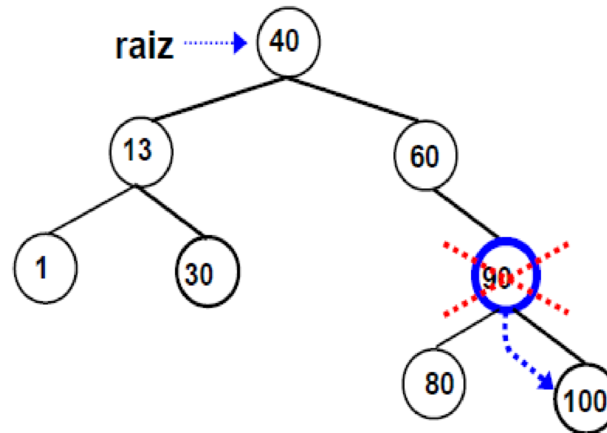


Figura 16: Remoção de nó com dois filhos Fonte: (Lipschurtz, 2006)

É importante notar que uma vez definidas as regras de escolha do nó substituto, ela deve ser a mesma para todas as operações de remoção!

Algoritmo de remoção numa árvore de pesquisa binária

```
p = tree;

q = null;

I* procura o nó com chave key, define p de modo a apontar */

/* para o nó e q para seu pai, caso exista algum. */

while (p .'= null && k(p) != key )

{

q = p;

p = {key < k(p)) ? left(p) : right(p);

}

/* fim while

}
```

Busca numa árvore binária

Comece a busca a partir do nó-raiz;

Para cada nó-raiz de uma sub-árvore compare:

Se o valor procurado é menor que o valor no nó-raiz (continua pela sub-árvore esquerda), ou se o valor é maior que o valor no nó-raiz (sub-árvore direita). Caso o nó contendo o valor pesquisado seja encontrado, retorne o nó, caso contrário retorne nulo.

Exemplo: Seja uma árvore binária como descrita abaixo, suponha que estamos a procura da chave 3.

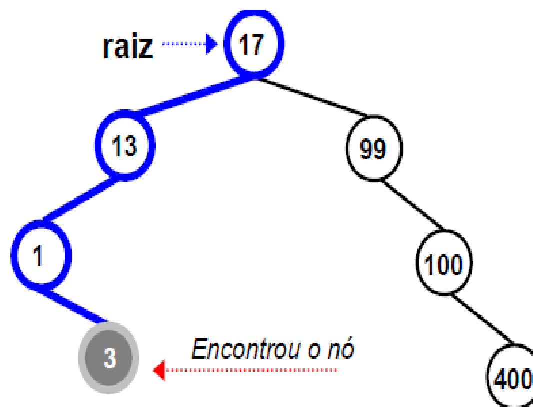


Figura 17: Pesquisa de um nó Fonte: Lipschurtz, 2006

A complexidade do algoritmo de pesquisa é determinado pela altura da árvore no pior caso.

Conclusão

A utilização de árvore de busca binária sobre um vetor traz vantagem pois uma árvore permite que as operações de inserção, remoção e de pesquisa, sejam executadas com eficiência.

Avaliação

1. Prove que toda árvore de busca binária de n nós não tem a mesma probabilidade (presumindo-se que os itens sejam inseridos em ordem aleatória) e que as árvores balanceadas são mais prováveis do que as árvores geradas em listas sequenciais.
2. Escreva um algoritmo para eliminar um nó de uma árvore binária que substitua o nó por seu predecessor em ordem e não por seu sucessor em ordem.

Actividades 3: Conjuntos disjuntos

Introdução

Aplicações que envolvem agrupamento de n elementos distintos em uma coleção de conjuntos disjuntos são feitas com o uso de conjuntos disjuntos. As operações de procurar o conjunto a que pertence um elemento e de unir dois conjuntos são as principais temas que iremos discutir nesta actividade. (www.das.ufsc.br)

Detalhes da actividade

Coleção de conjuntos dinâmicos disjuntos: S = {s1, s2, . . . , sk };

Cada conjunto é identificado por um representante (um elemento do conjunto);

As operações em conjuntos disjuntos são:

Make Set(x): cria um novo conjunto cujo único elemento é apontado por x.

x não pode pertencer a outro conjunto da coleção.

Union(x, y): executa a união dos conjuntos que contêm x e y, digamos Sx e Sy , em um conjunto único.

$S_x \cap S_y = \emptyset$;

O representante de $S_x \cup S_y$ é um elemento de S.

Find(x): retorna um ponteiro para o representante único do conjunto que contém .

Implementação através de Listas Encadeadas

Para implementar uma estrutura de dados de conjuntos disjuntos de forma mais simples, consiste em representar cada um dos conjuntos em forma de lista encadeada. Sendo que, o primeiro elemento da lista é designado representante do conjunto.

Exemplo:

S={a,d,e}

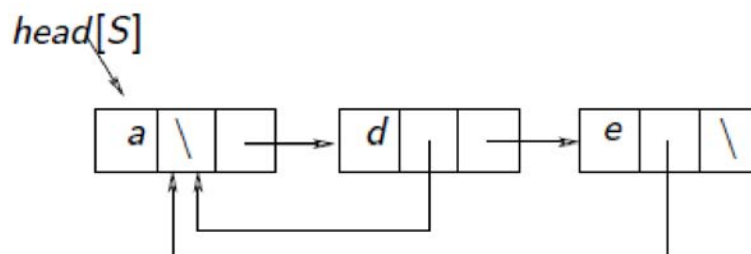


Figura 18: Estrutura de lista encadeada Fonte: www.das.ufsc.br consultado em 10/02/2016

Para a representação ilustrada acima: FindSet ∈ O(1) e MakeSet ∈ O(1)

Para implementar a operação união, usa-se uma forma mais simples como a seguir se ilustra:

Union(x, y), é adicionar a lista de x no fim da lista de y O representante do conjunto é o elemento que era originalmente o representante de y. No entanto, temos de atualizar o ponteiro para o representante de todos os elementos de x

Não é difícil construir uma instância e sequência de operações para mostrar que Union executa em tempo médio de θ(m).

Instância exemplo:

Exercícios:

Seja n = dm/2e + 1 o número de operações MakeSet:

Seja q = m - n = bm/2c - 1 o número de operações Union

Suponha que tenhamos x1,..., x_n objetos

Então executamos uma sequência de operações, conforme a tabela abaixo.

Operação	Número de objectos Observados
Make Set(x1)	1
Make Set(x2)	1
.....	
.....	
Make Set(xn)	1
Union(x1, x2)	1
Union(x2, x3)	2
Union(x3, x4)	3
......	....
......	
Union(xq−1, xq)	q − 1

Conclusão

Em computação, uma estrutura de dados conjunto-disjunto é uma estrutura de dados que considera um conjunto de elementos particionados em vários subconjuntos disjuntos.

Avaliação

1. Escreva o pseudocódigo para makeSet(x), union(x, y) e findSet(x) usando a representação de lista ligada. Suponha que cada objeto x tenha um atributo rep apontando para o representante do conjunto que contém x, e que cada conjunto S tem atributos inicio, fim e tamanho (que é igual ao comprimento da lista).

Resumo da Unidade

A fila de prioridade funciona de forma diferente que a fila padrão, em que os elementos são ordenados pela ordem de chegada. A fila de prioridade determina a ordem dos elementos realizando uma comparação entre os seus itens para verificar que elemento apresenta maior prioridade. Discutimos também o árvores binárias de pesquisa e a suas aplicações, concluímos que elas representam uma boa solução quando há necessidade de representar tipos ordenados de acordo com certas regras inerentes a esse tipo.

Avaliação da Unidade

Verifique a sua compreensão!

Instruções

Nas perguntas que se seguem, deverá responder todas elas com clareza:

1. Implemente uma fila de prioridade ascendente e suas operações, fqinsert, fqmindelete e empty.
2. Demonstre que é possível obter uma árvore de busca binária na qual existe uma única folha, mesmo se os elementos da árvore não forem inseridos em ordem estritamente ascendente ou descendente.
3. A árvore fornecida pelo algoritmo de Kruskal é única, ou seja, o algoritmo sempre fornecerá a mesma árvore geradora mínima?

Avaliação Sistemática

Critérios de Avaliação

Pergunta	Pontuação (máximo 10 valores)
1	4.0
2	3.0
3	3.0

Leituras e outros Recursos

- Carvalho, M. A. G. Tópicos Especias de Informática, 2004. Campinas, Brasil.
- Moreira, F. V. C & Viana, G. V. R.Técnicas de Divisão e Conquista e de Programação Dinâmica para a resolução de Problemas de Otimização. Revista Científica da Faculdade Lourenço Filho - v.8, n.1, 2011
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. 2ª Ed. The MIT Press, 2001.
- Algoritmos : teoria e prática 1 Thomas H. Cormen ... [et a[]; tradução da segunda edição [americana] Vandenberg D. de Souza. - Rio de Janeiro : Elsevier, 2002 -Reimpressão.
- Halim, S, Halim, F. Competitive Programming. Lulu, 2010
- Horowitz, E., Sahni, S., Rajasekaran, S. Computer Algorithms. Computer Science Press, 1997.

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso

- http://www.inf.puc-rio.br
- http://homepages.dcc.ufmg.br
- http://www.caelum.com.br
- http://www.ft.unicamp.br/~magic/analisealgo/apoalgoritmos_ceset_magic.pdf
- www.flf.edu.br/revista-flf.edu/volume08/Vol8_Artigo1.pdf
- Wikipedia: http://en.wikipedia.org/wiki/Disjointset_data_structure

Resumo da Unidade

Na presente unidade, a discussão começou com a apresentação de conceitos básicos inerentes a introdução da disciplina tais como a resolução de exercícios sobre os conteúdos tratados nas disciplinas antecedentes a esta, que constituem os pré-requisitos para cursar a disciplina de Análise e Design de Algoritmos. Na segunda unidade abordamos a complexidade de algoritmos e sua eficiência. Na terceira unidade abordamos várias técnicas de análise e de design de algoritmos, de seguida na quarta unidade trabalhamos com a algoritmos em grafos, onde abordamos os temas pesquisa em profundidade e em largura e também visto o tema sobre a sua importância. O algoritmo de busca por exemplo pode tirar proveito da ordenação dos dados. A operação de busca é tão frequente em aplicações que diversas estruturas de dados são projetadas especificamente para oferecer suporte a essa operação.

Avaliação da Curso

Instruções

Das perguntas que se seguem, leia e responda-as de forma clara e concisa, mostrando exemplos sempre que lhe for necessário.

EXAME FINAL 1

1. Dois algoritmos A e B possuem complexidade n5 e 2n,respectivamente. Qual dos dois utilizaria A ou B. Em qual caso? Exemplifica. (2.0 valores)
2. Para duas funções g(n) e f(n) temos que f(n)="teta"(n) se somente se f(n)= Ω(g(n)) e f(n)=O(g(n)). Explique o teorema acima. (2.0 valores)
3. Escreva algoritmos para percorrer elementos na árvore binária nas seguintes sequências:
   a. Pós Ordem (1.5 valores)
   b. Pré Ordem (1.5 valores)
   c. Em Ordem (1.5 valores)
4. Desenhe uma árvore binária ordenada de pesquisa composta por alfabeto português. (2.0 valores)
5. Desenvolver os algoritmos ordenação por seleção em linguagem C. Executar os programas com listas contendo 100, 200 e 400 elementos na seguinte situação inicial:
   a. Lista inicial aleatória (1.5 valor)
   b. Lista inicial ascendente (1.5 valor)
   c. Lista inicial descendente (1.5 valor)

6. Exemplifique representações geométricas de grafos completos Kn (n = 1,2,3,4 e 5)

   a.Quantas arestas possui um grafo completo Kn ? (2.5 valores)

   b.Calcule o total de arestas para n = 1,2,3,4 e 5 (2.5 valores)

EXAME FINAL 2

1. Escreva as propriedades da notação assimptótica. (2.0 valores)
2. Descreva a complexidade do algoritmo de pesquisa binária. (2.0 valores)
3. Calcule a complexidade para os algoritmos que se seguem:

   a. (2.0 valores)

```
for (int i = 1; i < n; i++)
{
min = i;
for (int j = i; j< m; j++)
{
if a[j] < a[min]
{
min = j;
x = a[min];
a[min] = a[i];
a[i] = x;
}}}
```

   b. (1.5 valores)

```
for ( i = 1; j< n; i++)
for ( j = 1; j< =i; j++)
soma=soma+1;
```

4. Escreva um algoritmo de inserção eficiente para uma árvore de busca binária inserir um novo registro cuja chave não existe na árvore. (2.5 valores)
5. Se tivesse que resolver um problema de pesquisa numa lista de n números, como poderia usar a vantagem do facto de que a lista por ordenar já é conhecida? Dê respostas separadas para:

a. A lista é representada por um array. (2.5 valores)

b. A lista é representada por listas ligadas. (2.5 valores)

c. Compare a complexidade de tempo envolvida na análise de ambos algoritmos. (2.5 valores)

6. Escreva o algoritmo Mergesort e explique através de exemplos o uso da técnica de divisão e conquista. (3.0 valores)

Avaliação Sistemática

Critérios de avaliação

O exame final tem a pontuação máxima de 20 valores, valendo 40% do total da avaliação do curso. A duração do mesmo é de 90 minutos.

Leituras e outros Recursos

- Carvalho, M. A. G. Tópicos Especias de Informática, 2004. Campinas, Brasil.
- Moreira, F. V. C & Viana, G. V. R.Técnicas de Divisão e Conquista e de Programação Dinâmica para a resolução de Problemas de Otimização. Revista Científica da Faculdade Lourenço Filho - v.8, n.1, 2011

As leituras e outros recursos desta unidade encontram-se na lista de Leituras e Outros Recursos do curso

- http://www.inf.puc-rio.br
- http://homepages.dcc.ufmg.br
- http://www.caelum.com.br
- http://www.ft.unicamp.br/~magic/analisealgo/apoalgoritmos_ceset_magic.pdf
- www.flf.edu.br/revista-flf.edu/volume08/Vol8_Artigo1.pdf

Sede da Universidade Virtual africana

The African Virtual University Headquarters

Cape Office Park

Ring Road Kilimani

PO Box 25405-00603

Nairobi, Kenya

Tel: +254 20 25283333

contact@avu.org

oer@avu.org

Escritório Regional da Universidade Virtual Africana em Dakar

Université Virtuelle Africaine

Bureau Régional de l'Afrique de l'Ouest

Sicap Liberté VI Extension

Villa No.8 VDN

B.P. 50609 Dakar, Sénégal

Tel: +221 338670324

bureauregional@avu.org

CC BY SA 2017 UVA