Análise e Síntese de Algoritmos

Resumo

Rafael Rodrigues

LEIC
Instituto Superior Técnico
2023/2024

Contents

1 Fundamentos **3**

1.1 Divide-and-Conquer . . . 3

1.2 Teorema Mestre . . . 3

1.2.1 Teorema Mestre Simplificado . . . 3

1.2.2 Teorema Mestre Generalizado . . . 3

2 Sorting and Order Statistics **4**

2.1 Amontoados . . . 4

2.1.1 Operações sobre Amontoados . . . 4

2.1.2 Algoritmo Heap-Sort . . . 4

2.1.3 Outras Operações . . . 4

4 Técnicas de Síntese de Algoritmos **5**

4.1 Programação Dinâmica . . . 5

4.2 Algoritmos Greedy . . . 6

4.2.1 Seleção de Atividades . . . 6

4.2.2 Algoritmo de Huffman . . . 6

4.3 Análise Amortizada . . . 7

5 Estruturas de Dados Avançadas **8**

5.1 Estrutura de Dados para Conjuntos Disjuntos . . . 8

5.1.1 Estruturas Baseadas em Listas . . . 8

5.1.2 Estruturas Baseadas em Árvores . . . 8

6 Algoritmos em Grafos **9**

6.1 Algoritmos Elementares . . . 9

6.1.1 Representação de Grafos . . . 9

6.1.2 Breadth-First Search . . . 9

6.1.3 Depth-First Search . . . 9

6.1.4 Ordenação Topológica 9
6.1.5 Componentes Fortemente Ligados 9
6.2 Árvores Abrangentes de Menor Custo 10
6.2.1 Algoritmo de Prim 10
6.2.2 Algoritmo de Kruskal 10
6.3 Caminhos Mais Curtos com Fonte Única 11
6.3.1 Algoritmo Dijkstra 11
6.3.2 Algoritmo Bellman-Ford 11
6.3.3 Caminhos mais curtos em DAGs 11
6.4 Caminhos Mais Curtos entre Todos os Pares 12
6.4.1 Algoritmo Johnson 12
6.5 Fluxos Máximos 13
6.5.1 Método Ford-Fulkerson 13
6.5.2 Algoritmo Edmonds-Karp 13
6.5.3 Emparelhamento Bipartido Máximo 13

7 Tópicos Adicionais **14**

7.1 Programação Linear 14
7.1.1 Formulações 14
7.1.2 Algoritmo Simplex 14
7.1.3 Dualidade 14
7.2 Completude-NP 15
7.2.1 Problemas Resolúveis em Tempo Polinomial 15
7.2.2 Problemas Verificáveis em Tempo Polinomial 15
7.2.3 Redutibilidade e Completude-NP 15

1 Fundamentos

Slides Aula 1 e 2

1.1 Divide-and-Conquer

- Dividir o problema num conjunto de sub-problemas do mesmo tipo.
- Resolver cada sub-problema (com uma chamada recursiva).
- Combinar as soluções dos sub-problemas para obter a solução do problema original.

1.2 Teorema Mestre

1.2.1 Teorema Mestre Simplificado

Se $T(n) = aT(n/b) + n^d$, $a \geq 1$, $b > 1$, $d \geq 0$, então:

$$T(n) = \begin{cases} O(n^d) & \text{se } d > \log_b a \\ O(n^d \log_b n) & \text{se } d = \log_b a \\ O(n^{\log_b a}) & \text{se } d < \log_b a \end{cases}$$

1.2.2 Teorema Mestre Generalizado

Se $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$, então:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^{\log_b a - \varepsilon}) \\ \Theta(n^{\log_b a} \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^{\log_b a + \varepsilon}) \end{cases}$$

2 Sorting and Order Statistics

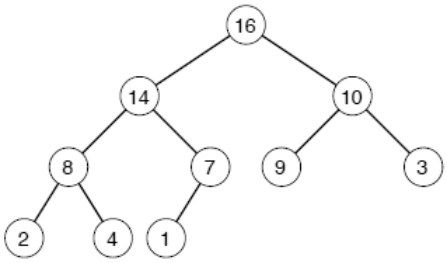
2.1 Amontoados

Slides Aula 5

Um **heap** é um vetor de valores (A) interpretado como uma árvore binária (essencialmente completa), em que $A[1]$ é a raíz da árvore e $A[Parent(i)] \geq A[i]$.

Relações entre nós da árvore:

- Parent$(i) = \lfloor i/2 \rfloor$
- Left$(i) = 2i$
- Right$(i) = 2i + 1$



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

2.1.1 Operações sobre Amontoados

- **Max-Heapify**($A,\ i$): $O(\log n)$
  - Transforma a árvore com raiz em i num amontoado, assumindo que as árvores com raiz em Left(i) e Right(i) são amontoados
- **Build-Max-Heap**(A): $O(n)$
  - Construir amontoado a partir de um vector arbitrário
  - Chamada seletiva de Max-Heapify

2.1.2 Algoritmo Heap-Sort

1. Extrair consecutivamente o elemento máximo de uma heap
2. Colocar esse elemento na posição (certa) do vector

- **Complexidade**: $O(n \log n)$

2.1.3 Outras Operações

- **Heap-Maximum**(A): $O(1)$
  - Devolve o maior elemento da heap
- **Heap-Extract-Max**(A): $O(\log n)$
  - Remove o maior elemento da heap
- **Heap-Increase-Key**($A,\ i,\ key$): $O(\log n)$
  - Incrementa o elemento i para o valor key
- **Max-Heap-Insert**($A,\ key$): $O(\log n)$
  - Insere key na heap

4 Técnicas de Síntese de Algoritmos

4.1 Programação Dinâmica

Slides Aula 3 e 4

Passos para a realização de um algoritmo baseado em programação dinâmica:

- Caracterizar **estrutura** de uma solução ótima
- Definir **recursivamente** o valor de uma solução ótima
- Calcular valor da solução ótima utilizando abordagem **bottom-up**
- Construir **solução** a partir da informação obtida

Características da Programação Dinâmica:

- Solução ótima do problema composta por soluções ótimas para sub-problemas
- Solução construtiva para evitar resolver repetidamente o mesmo problema
  - Solução recursiva resolve repetidamente os mesmos sub-problemas
- Reconstrução da solução ótima
- Memorização
  - Permite obter tempo de execução das soluções bottom-up, mas utilizando abordagem recursiva, **memorizando** resultados de sub-problemas já resolvidos

4.2 Algoritmos Greedy

Slides Aula 6

Caracteristicas Algoritmos Greedy

- **Propriedade da escolha greedy**
  - Ótimo (global) para o problema pode ser encontrado realizando escolhas locais ótimas (em programação dinâmica, esta escolha está dependente de resultados de sub-problemas)
- **Sub-estrutura ótima**
  - Solução ótima do problema engloba soluções ótimas para sub-problemas

4.2.1 Seleção de Atividades

A escolha greedy é optar sempre pela **próxima atividade que acabar mais cedo**.

4.2.2 Algoritmo de Huffman

1. Adicionar cada caracter à fila de prioridade, pela ordem ascendente de frequência.
2. Retirar os dois elementos com menor frequência da fila.
3. Somam-se os valores dos dois elementos, criando um novo nó com esse valor, que é inserido na fila.
4. Enquanto a fila não estiver vazia, voltar ao passo 2.
5. Numerar os nós de acordo com o enunciado.

Custo do código de uma árvore binário T, sobre um alfabeto Λ:

$$B(T) = \sum_{i \in \Lambda} f(i) \cdot d_T(i) \ , \ d_T \text{ corresponde à profundidade de } i \text{ na árvore binária}$$

A frequência pode corresponder a uma fração do número total de caracteres, neste caso:

$$\text{Total Bits} = B(T) \times \frac{\#\text{caracteres}}{\text{raiz da árvore } T}$$

4.3 Análise Amortizada

Slides Aula 7 e 8

Análise Amortizada - Determina o **custo médio** de uma **sequência de operações** sobre uma estrutura de dados.

Custo Amortizado - Dada uma sequência de n operações com uma complexidade total de $T(n)$, o custo amortizado de cada operação é $T(n)/n$

Métodos para Análise Amortizada:

- Método de análise agregada
  - Determina o custo amortizado de uma sequência de operações
  - Divide o custo total $T(n)$ pelo número de n operações
- Método do contabilista
  - Mais que um tipo de operação, cada operação tem custo **real** e custo **amortizado**
  - Operações baratas **pagam mais** para que operações caras **paguem menos**
- Método do potencial
  - Variação do método do contabilista, onde o crédito pago a mais por certas operações é mantido como **energia potencial** total que pode ser gasto em operações caras

5 Estruturas de Dados Avançadas

5.1 Estrutura de Dados para Conjuntos Disjuntos

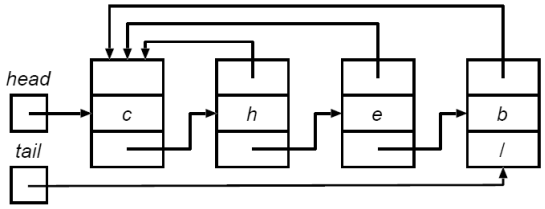
Slides Aula 14

Operações sobre Conjuntos Disjuntos:

- **Make-Set(x)** - Cria novo conjunto que apenas inclui o elemento x, que será o representante do conjunto.
- **Find-Set(x)** - Retorna o representante do conjunto que contém x.
- **Union(x, y)** - Realiza a união dos conjuntos que contêm x e y, respetivamente S_x e S_y.

5.1.1 Estruturas Baseadas em Listas

- Cada conjunto representado por uma lista ligada.
- Primeiro elemento é o representante do conjunto.
- Todos os elementos incluem apontador para o representante do conjunto.



Heurística União Pesada

Associa a cada conjunto o seu número de elementos, permitindo juntar a lista com menor número de elementos à lista com maior número de elementos em cada operação Union.

Complexidade: $O(m + n \log n)$ para m operações com n Union

5.1.2 Estruturas Baseadas em Árvores

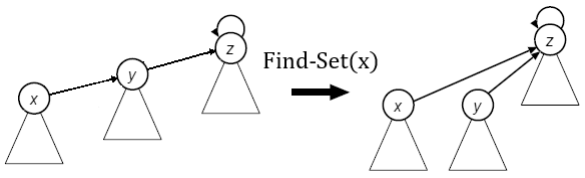
- Cada conjunto representado por uma árvore.
- Cada elemento aponta apenas para antecessor na árvore.
- Representante da árvore é a raiz, que antecede a si própria.

Heurística de União por Categoria

O representante de cada conjunto guarda a sua categoria (rank), que inicialmente é 0. O rank pode ser aumentado quando há uma união de dois conjuntos, onde se coloca a árvore com menor rank a apontar para árvore com maior rank. Utilizando apenas esta heurística a **complexidade** é $O(m \cdot \log n)$.

Heurística de Compressão de Caminhos

Em cada operação Find-Set coloca cada nó visitado a apontar diretamente para a raiz da árvore



Complexidade: $O(m \cdot \alpha(n)) \approx O(m)$ para m operações com n Union

6 Algoritmos em Grafos

6.1 Algoritmos Elementares

Slides Aula 9 e 10

6.1.1 Representação de Grafos

Dado um grafo $G = (V, E)$, um **caminho** p é uma sequência $\langle v_0, v_1, \ldots, v_k \rangle$ tal que para todo o i, $0 \leq i \leq k-1, (v_i, v_{i+1}) \in E$.

- Se existe um caminho p de u para v, então v diz-se **atingível** a partir de u usando p.
- Um **ciclo** num grafo $G = (V, E)$ é um caminho $\langle v_0, v_1, \ldots, v_k \rangle$, tal que $v_0 = v_k$.
- Um grafo dirigido $G = (V, E)$ diz-se **acíclico** (ou Directed Acyclic Graph (DAG)) se não tem ciclos.

6.1.2 Breadth-First Search

Dados $G = (V, E)$ e vértice fonte s, o algoritmo BFS explora sistematicamente os vértices de G para descobrir todos os vértices atingíveis a partir de s.

Árvore Breadth-First - é um sub-grafo de G, inclui todos os vértices atingíveis a partir de s e os arcos que definem a relação predecessor da BFS.

6.1.3 Depth-First Search

DFS

6.1.4 Ordenação Topológica

Uma **ordenação topológica** de um DAG $G = (V, E)$ é uma ordenação de todos os vértices tal que se $(u, v) \in E$ então u aparece antes de v na ordenação.

Complexidade: $O(V + E)$

6.1.5 Componentes Fortemente Ligados

Dado um grafo dirigido $G = (V, E)$ um **componente fortemente ligado** (ou Strongly Connected Component (SCC)) é um conjunto máximo de vértices $U \subseteq V$, tal que para quaisquer u, $v \in U$, u é atingível a partir de v, e v é atingível a partir de u.

Complexidade: $O(V + E)$

6.2 Árvores Abrangentes de Menor Custo

Slides Aula 13

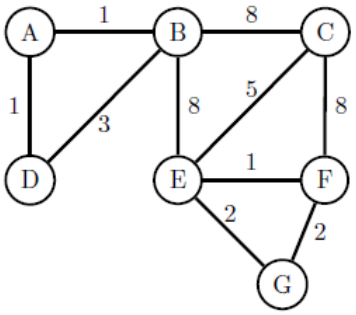
- Um grafo não dirigido $G = (V, E)$, diz-se **ligado** se para qualquer par de vértices existe um caminho que liga os dois vértices
- Dado um grafo não dirigido $G = (V, E)$, ligado, uma **árvore abrangente** é um sub-conjunto acíclico $T \subseteq E$, que liga todos os vértices
- O tamanho da árvore é $|T| = |V| - 1$
- Uma **árvore abrangente de menor custo** é uma árvore abrangente T em que a soma dos pesos dos seus arcos é minimizada.

$$\min w(T) = \sum_{(u,v)\in T} w(u, v)$$

6.2.1 Algoritmo de Prim

1. Escolher um vértice raiz r (que passa a ser a árvore) e adicionar os restantes vértices à fila por explorar.
2. Procurar o arco adjacente de menor peso que liga a árvore a um vértice não explorado. Adicionar esse arco à árvore e remover o vértice da fila por explorar.
3. Cada vértice guarda o seu antecessor, $\pi[v]$, e o peso do arco que o liga à árvore, $key[v]$.
4. Voltar ao segundo passo até não haver vértices por explorar.

- **Complexidade**: $O(E \cdot \log V)$



	A	B	C	D	E	F	G
$key[]$	0	1	8	1	5	1	2
$\pi[]$	Nil	A	B	A	C	E	E

6.2.2 Algoritmo de Kruskal

1. Cria uma floresta de árvores, em que cada vértice é uma árvore. (**Make-Set**)
2. Percorre as arestas do grafo pela ordem ascendente de peso.
3. Para cada aresta verifica se os seus vértices pertencem a árvores diferentes. (**Find-Set**)
4. Se sim, une as duas árvores. (**Union**)

- **Complexidade**: $O(E \cdot \log V)$

  Esta complexidade é obtida recorrendo às estruturas de dados adequadas, referidas no Capítulo 5

6.3 Caminhos Mais Curtos com Fonte Única

Slides Aula 11

- Dado um grafo $G = (V, E)$, dirigido, com uma função de pesos $w : E \to \mathbb{R}$, define-se o **peso de um caminho** p, onde $p = \langle v_0, v_1, \ldots, v_k \rangle$, como a soma dos pesos dos arcos que compõem p:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- O **peso do caminho mais curto** de u para v é definido por:

$$\delta(u, v) = \begin{cases} min\{w(p) : u \to_p v\} & \text{se existe caminho de u para v} \\ \infty & \text{se não existe} \end{cases}$$

- Um **caminho mais curto** de u para v é qualquer caminho p tal que $w(p) = \delta(u, v)$.

6.3.1 Algoritmo Dijkstra

1. Mantém conjunto de vértices S com pesos dos caminhos mais curtos já calculados
2. A cada passo seleciona vértice $u \in V - S$ com menor estimativa do peso do caminho mais curto
3. Insere o vértice u em S e relaxa os arcos que saem de u

- Só permite **pesos não negativos**.
- **Complexidade**: $O((V + E) \log V)$

6.3.2 Algoritmo Bellman-Ford

1. Relaxa todas as arestas do grafo $V - 1$ vezes, com vista a atualizar gradualmente a estimativa de custo associado a cada vértice.
2. Percorre mais uma vez todas as arestas, de modo a verificar se há algum ciclo negativo.

- Permite **pesos negativos** e identifica **ciclos negativos**.
- **Complexidade**: $O(VE)$

6.3.3 Caminhos mais curtos em DAGs

1. Ordena os vértices do grafo pela ordem topológica.
2. Relaxa os vértices pela ordem obtida anteriormente.

- Só permite grafos **acíclicos**.
- **Complexidade**: $O(V + E)$

6.4 Caminhos Mais Curtos entre Todos os Pares

Slides Aula 12

Para encontrar o caminho mais curto entre **todos os pares** de vértices de um grafo podemos aplicar o algoritmo de Dijkstra a cada vértice, sendo a limitação desta abordagem o facto deste algoritmo não suportar arcos com peso negativo.

6.4.1 Algoritmo Johnson

O algoritmo de Johnson combina os algoritmos de Dijkstra e de Bellman-Ford, para fazer a **re-pesagem dos arcos** do grafo, solucionando a limitação anteriormente mencionada.

- Aplicar o algoritmo de Bellman-Ford: $O(VE)$

  Dado um grafo $G = (V, E)$ a re-pesagem de Johnson devolve um grafo $\hat{G} = (V, \hat{E})$, onde $\hat{E}$ corresponde a arcos "equivalentes" aos de E, porém sem arcos negativos.

  1. Adicionar um vértice s, ao grafo, e conectá-lo com arcos de peso 0 a todo o vértice v de G.
  2. Aplicar o algoritmo de Bellman-Ford ao grafo para descobrir o peso do caminho mais curto de s a cada vértice (**alturas de Johnson**, h).
  3. Calcular o peso $\hat{w}$ de cada arco em $\hat{G}$: $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$
  4. Remover s e os respetivos arcos de modo a obter $\hat{G}$.

- Aplicar o algoritmo de Dijkstra a cada vértice: $O(V(V + E) \log V)$
- **Complexidade Total**: $O(V(V + E) \log V)$

6.5 Fluxos Máximos

Slides Aula 15 e 17

Rede de Fluxo - é um grafo $G = (V, E)$ dirigido, em que cada arco (u, v) tem capacidade $c(u, v) \geq 0$. O grafo é ligado, logo todos os vértices de G pertencem a um caminho da **fonte** s para o **destino** t.

Rede Residual - de um grafo G com fluxo f é um grafo $G_f = (V, E_f)$, em que cada arco (u, v) tem uma **capacidade residual** $c_f(u, v) = c(u, v) - f(u, v) > 0$, ou seja, o fluxo que sobra de u para v.

6.5.1 Método Ford-Fulkerson

6.5.2 Algoritmo Edmonds-Karp

6.5.3 Emparelhamento Bipartido Máximo

7 Tópicos Adicionais

7.1 Programação Linear

Slides Aula 18

Procura otimizar (minimizar ou maximizar) função linear (objetivo) sujeita a um conjunto de restrições (desigualdades) lineares.

- Qualquer solução que satisfaça o conjunto de restrições designa-se por **solução exequível**.
  A formulação diz-se **não exequível** se não existir solução exequível.
- A cada solução exequível corresponde um valor (custo) da função objetivo.
- O conjunto de soluções exequíveis é designado por **região exequível**, ou simplex.
- A **solução ótima** encontra-se num vértice do simplex.
  Se a formulação é exequível, mas sem solução ótima, diz-se **não limitada**.

7.1.1 Formulações

Forma Standard

Forma Slack

7.1.2 Algoritmo Simplex

7.1.3 Dualidade

7.2 Completude-NP

Slides Aula 19 e 20

7.2.1 Problemas Resolúveis em Tempo Polinomial

7.2.2 Problemas Verificáveis em Tempo Polinomial

7.2.3 Redutibilidade e Completude-NP

Slides Correspondentes

Slides Aula 1 e 2 3

Slides Aula 5 4

Slides Aula 3 e 4 5

Slides Aula 6 6

Slides Aula 7 e 8 7

Slides Aula 14 8

Slides Aula 9 e 10 9

DFS 9

Slides Aula 13 10

Slides Aula 11 11

Slides Aula 12 12

Slides Aula 15 e 17 13

Slides Aula 18 14

Slides Aula 19 e 20 15