# I. Análise de Correcção de Programas

## Correcção Parcial (Algumas Regras em Lógica de Hoare)

Fortalecimento da pré-condição

$$\frac{R \Rightarrow P \qquad \{ P \}\ S\ \{ Q \}}{\{ R \}\ S\ \{ Q \}}\ (\text{Fort})$$

Enfraquecimento da pós-condição

$$\frac{\{ P \}\ S\ \{ Q \} \qquad Q \Rightarrow R}{\{ P \}\ S\ \{ R \}}\ (\text{Enfraq})$$

Atribuição-1

$$\frac{}{\{ P[x \backslash E] \}\ x := E\ \{ P \}}\ (\text{Atrib1})$$

Atribuição-2

$$\frac{P \Rightarrow (Q[x \backslash E])}{\{ P \}\ x = E\ \{ Q \}}\ (\text{Atrib2})$$

Sequência

$$\frac{\{ P \}\ S_1\ \{ R \} \qquad \{ R \}\ S_2\ \{ Q \}}{\{ P \}\ S_1 ; S_2\ \{ Q \}}\ (;)$$

Condicional

$$\frac{\{ P \wedge c \}\ S_1\ \{ Q \} \qquad \{ P \wedge \neg c \}\ S_2\ \{ Q \}}{\{ P \}\ \texttt{if}\ c\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \{ Q \}}\ (\text{if})$$

Ciclo-1

$$\frac{\{ I \wedge c \}\ S\ \{ I \}}{\{ I \}\ \texttt{while}\ c\ S\ \{ I \wedge \neg c \}}\ (\text{while}-1)$$

Ciclo-2

$$\frac{P \Rightarrow I \qquad \{ I \wedge c \}\ S\ \{ I \} \qquad (I \wedge \neg c) \Rightarrow Q}{\{ P \}\ \texttt{while}\ c\ S\ \{ Q \}}\ (\text{while}-2)$$

## Correcção Total

Para a Correcção Total aplicam-se as regras da Parcial, substituindo-se apenas os parênteses curvos por rectos. As excepções são os casos "Ciclo-1" e "Ciclo-2":

Ciclo-1

$$\frac{(I \wedge c) \Rightarrow V \geqslant 0 \qquad [\ I \wedge c \wedge (V = v_0)\ ]\ S\ [\ I \wedge (V < v_0)\ ]}{[\ I\ ]\ \texttt{while}\ c\ S\ [\ I \wedge \neg c\ ]}\ (\text{while}-1)$$

Ciclo-2

$$\frac{P \Rightarrow I \qquad (I \wedge c) \Rightarrow V \geqslant 0 \qquad [\ I \wedge c \wedge V = v_0\ ]\ S\ [\ I \wedge V < v_0\ ] \qquad (I \wedge \neg c) \Rightarrow Q}{[\ P\ ]\ \texttt{while}\ c\ S\ [\ Q\ ]}\ (\text{while}-2)$$

## Condições de Verificação

Premissas a provar quando se quer mostrar a validade de um ciclo:

1. **P => I**: Antes da execução do ciclo o invariante é verdadeiro
2. **I ∧ c => ( V≥ 0 )**: antes de cada iteração o variante é não negativo
3. **{ I ∧ c } S { I }**: assumindo que o invariante é válido antes de uma iteração do ciclo, ele continua válido depois dessa iteração
4. **( I ∧ ¬c ) => Q**: quando o ciclo termina a pós-condição é estabelecida.
5. **{ I ∧ c ∧ V = $v_0$ } S { V < $v_0$ }**: por cada iteração o valor do variante decresce.

3. e 5. podem resumir-se em: **{ I ∧ c ∧ V = $v_0$ } S { I ∧ V < $v_0$ }**

## Outras Notas

- Usar uma anotação antes de qualquer comando que não seja uma atribuição;
- Usar uma anotação imediatamente a seguir à condição de um ciclo.
- ((...)[2ª instr])[1ª instr]

# IIa. Contagem de Operações em Algoritmos Iterativos

## Somas de Séries

| | | |
|---|---|---|
| $\sum_{i=1}^{N-1} 1 = N-1$ | $\sum_{i=a}^{b} 1 = b - a + 1$ | |
| $\sum_{i=1}^{N} i = \frac{N(N+1)}{2}$ | $\sum_{i=k}^{N} i = \sum_{i=1}^{N} 1 + \sum_{i=1}^{N-1} i$ | $\sum_{i=a}^{b} i = \frac{(a+b)(b-a+1)}{2}$ |
| $\sum_{i=1}^{N} k = k \,.\, N$ | $\sum_{i=0}^{N} N = N(N+1)$ | $\sum_{i=1}^{N} N = N^2$ |
| $a^{(\log_a N)} = N$ | $\sum_{i=1}^{n} i x^i = x\frac{1 + n x^{n+1} - (n+1)x^n}{(1-x)^2}$ | |
| $\sum_{i=0}^{N} a^i = \frac{a^{(N+1)} - 1}{a-1}$ | $\sum_{i=a}^{n} b^i = \frac{b^{n+1} - b^a}{b-1}$ | $\sum_{i=1}^{n} b^i = b * \frac{b^n - 1}{b-1}$ |
| $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$ | $\sum_{i=1}^{N} \frac{1}{2^i} = 1 - \frac{1}{2^N}$ | $\sum_{i=1}^{N} \frac{i}{2^i} = 2 - \frac{N+2}{2^N}$ |
| $\sum_{i=a}^{b} k \,.\, t(i) = k \,.\, \sum_{i=a}^{b} t(i)$ | $\sum (f(i) + g(i)) = \sum f(i) + \sum g(i)$ | |

## Recorrências Típicas (e respectivo tempo de execução)

| | |
|---|---|
| T(n) = Θ(1) + T(n-1) | Θ(n) |
| T(n) = Θ(n) + T(n-1) | Θ(n^2) |
| T(n) = Θ(1) + T(n/2) | Θ(log n) |
| T(n) = Θ(n) + T(n/2) | Θ(n) |
| T(n) = Θ(n) + 2T(n/2) | Θ(n log n) |
| T(n) = Θ(1) + 2T(n-1) | Θ(2^n) |
| T(n) = Θ(1) + 2T(n/2) | Θ(n) |

## Outras Notas

- O tempo de execução dá sempre um de (do maior para o menor): $N^N$, $2^N$, $N^k$, …, $N^3$, $N^2$, N.log N, N, log N, 1.
- $f \in O(f)$ → f é limitado superiormente por f ("pior caso")
  $f \in \Omega(f)$ → f é limitado inferiormente por f ("melhor caso")
  $f \in \theta(f)$ → f é limitado por f

- $\Theta(N^2) =$, $O(N^2) \leq$, $\Omega(N^2) \geq$
- Propriedades das Notações Θ, O, Ω
  - Transitividade: f(n) = Θ(g(n)) e g(n) = Θ(h(n)) implica f(n) = Θ(h(n)) [válida também para O e Ω]
  - Reflexividade: f(n) = Θ(f(n)) [válida também para O e Ω]
  - Simetria f(n) = Θ(g(n)) sse g(n) = Θ(f(n))
  - Transposição f(n) = O(g(n)) sse g(n) = Ω(f(n))
- Funções Úteis em Análise de Algoritmos

  Uma função f(n) diz-se limitada:
  • polinomialmente se f(n) = $O(n^k)$ para alguma constante k.
  • polilogaritmicamente se f(n) = $O(\log_a^k n)$ para alguma constante k [notação: lg = $\log_2$].

  Algumas relações úteis:

$$\log_b n = O(\log_a n) \quad [a, b > 1]$$
$$n^b = O(n^a) \quad se\ b \leqslant a$$
$$b^n = O(a^n) \quad se\ b \leqslant a$$
$$\log^b n = O(n^a)$$
$$n^b = O(a^n) \quad [a > 1]$$
$$n! = O(n^n)$$
$$n! = \Omega(2^n)$$
$$lg(n!) = \Theta(n \lg n)$$

- Algoritmos de ordenação: http://www.youtube.com/playlist?list=PL89B61F78B552C1AB
- Counting sort: http://www.youtube.com/watch?v=3mxp4JLGasE&t=15m30s

# IIb. Análise Amortizada de Algoritmos

Trata-se do estudo do custo médio, relativamente a uma sequência de operações sobre uma estrutura de dados, de cada operação no pior caso, e não de uma análise de caso médio. Não envolve ferramentas probabilísticas nem assunções sobre os inputs.

A ideia chave é considerar o pior caso da sequência de N operações, em situações em que este é claramente mais baixo do que a soma dos tempos de pior caso das N operações singulares.

## Análise Agregada

- Mostrar que, para qualquer *n*, uma sequência de *n* operações executa no pior caso em tempo *T(n)*.
- Então o custo amortizado por operação é *T(n)/n*.
- Considera-se que todas as diferentes operações têm o mesmo custo amortizado (as outras técnicas de análise amortizada diferem neste aspecto).

## Método Contabilístico

- *Atribuir* custos amortizados $c_i$, possivelmente diferentes, às diversas operações, que podem não corresponder aos reais – superiores para algumas operações (acumulando *crédito*) e inferiores para outras (gastando *crédito acumulado*).

- Se o custo amortizado total de uma (qualquer) sequência de $n$ operações for um limite superior para o custo real, $\Sigma_i t_i \leq \Sigma_i c_i$, então o custo total amortizado fornece um limite para o tempo de pior caso da sequência, o que significa que os custos amortizados individuais de cada operação podem ser considerados válidos para efeitos de análise de pior caso.
- Uma forma de se garantir a condição anterior é assegurar o seguinte:
  *Em qualquer momento na execução da sequência de operações, o custo acumulado (ou "saldo") deve ser sempre não-negativo.*
- $B_{i+1} = B_i - C_{i+1} + \hat{C}_{i+1}$
  $B$ – *saldo* ; $C$ – *custo* ; $\hat{C}$ *custo amortizado*

## Método do Potencial

- O método contabilístico gera crédito individualmente, por cada operação efectuada.
- O método do potencial considera este crédito de forma global, para toda a estrutura de dados, com base numa *função de potencial* sobre os estados sucessivos da estrutura de dados.
- Seja $\Phi_i$ o potencial da estrutura de dados no estado $i$, ou seja depois de $i$ operações da sequência. O custo amortizado da operação $i$ é então dado por
  $$c_i = t_i - \Phi_{i-1} + \Phi_i$$
- A ideia é que o potencial da estrutura deve aumentar com operações de baixo custo, e diminuir com operações de alto custo.
- O cálculo do custo amortizado total é *telescópico*:
  $$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} t_i - \Phi_{i-1} + \Phi_i \\ &= (t_1 - \Phi_0 + \Phi_1) + (t_2 - \Phi_1 + \Phi_2) + (t_3 - \Phi_2 + \Phi_3) + \ldots + (t_n - \Phi_{n-1} + \Phi_n) \\ &= \Phi_n - \Phi_0 + \sum_{i=1}^{n} t_i\end{aligned}$$
- As seguintes condições são suficientes para garantir que $\Sigma_i c_i \geq \Sigma_i t_i$:
  $\Phi_0 = 0$ e $\Phi_i \geq 0$ para $i > 0$.
- Nestas condições o custo total amortizado fornece um limite superior para o tempo de pior caso da sequência, como desejado.

# III. Estruturas de Dados Eficientes

## Buffers

Queues

- Queue é um tipo de dados abstracto baseado no princípio de First In First Out (FIFO), ou seja, o primeiro elemento a sair é o primeiro a ser acrescentado.
- Operação “enqueue” → inserção de dados no conjunto.
- Operação “dequeue” → remoção de dados do conjunto.
- Operação “peek” ou “front” → devolve o valor da frente da queue, mas sem o remover.

Stacks

- Stack é um tipo de dados abstracto baseado no princípio de Last In First Out (LIFO), ou seja, o primeiro elemento a sair é o último a ser acrescentado.
- Operação "push" → inserção de dados no conjunto.
- Operação "pop" → remoção de dados do conjunto.
- Operação “top” ou "peek" → devolve o valor no topo da stack, mas sem o remover.

## Árvores Binárias

Árvore Binária de Pesquisa é uma estrutura de dados de árvore binária baseada em nós, onde todos os nós da sub-árvore esquerda possuem um valor numérico inferior ao nó raiz e todos os nós da sub-árvore direita possuem um valor superior ao nó raiz. As sub-árvores esquerda e direita devem também ser árvores binárias de procura e não devem existir nodos duplicados.

Numa árvore completa, todos os nós (excepto possivelmente um) são equilibrados: as alturas das suas duas sub-árvores são iguais, ou seja $|h_e - h_d| = 0$.

O tempo de execução:

- é Θ(log n) no caso da pesquisa binária num vector ordenado;
- numa BST em geral a operação de pesquisa executa em Ω(1) e em O(n);
- numa BST cuja altura seja logarítmica no número de nodos da árvore, a operação de pesquisa executa em Θ(log n);

## Heaps

Max/Min Heap

- Árvore binária em que a raíz é maior/menor que os filhos.
- Inserção é na 1ª posição livre.
- 2*p+1 (1º filho)
  2*p+2 (2º filho)
  (p-1)/2 (pai)
  p → posição onde está
- Remoção só na raíz, pega-se no último elemento do vector, põe-se no topo e reordena-se.

## AVL

- AVL → árvore binária de procura balanceada; elementos à esquerda menores que a raíz; elementos à direita maiores que a raíz.

- Nas árvores AVL cada nodo pode estar equilibrado, mas também desequilibrado para a esquerda ou para a direita, desde que a diferença entre as alturas não exceda 1, ou seja, $|h_e - h_d| \leq 1$.
- Este relaxamento permite que a altura da árvore permaneça logarítmica, pelo que o mesmo acontece com o tempo de pesquisa.
- O tempo de execução dos algoritmos de inserção e remoção, modificados por forma a efectuarem o necessário ajuste das árvores preservando sempre o invariante $|h_e - h_d| \leq 1$, é também logarítmico.
- As operações adicionais necessárias para a "auto-regulação" do equilíbrio dos nós (gestão de informação auxiliar e rotações) executam em tempo $\Theta(1)$.
- Tendo as árvores sempre altura logarítmica, as consultas são feitas em tempo $\Theta(\log n)$.
- As operações de inserção e remoção, que numa BST de altura logarítmica executam em tempo também logarítmico, não são perturbadas pelas operações adicionais, mantendo-se em $\Theta(\log n)$.

## Tabelas de Hash

- Generalização dos arrays para grande universo de chaves.
- Tendencialmente as operações executam em tempo constante $\Theta(1)$.
- Factor de carga ("load"); aconselhável limite < 0.8

$$\alpha = \frac{\text{nº de chaves inseridas}}{\textit{capacidade}}$$

- Se $\alpha > k$, fazer: duplicar a capacidade ("rehash"). $\Theta(1)$ em termos amortizados.
- Condicionamento + "hashing".
  h(x) = x % N, com N primo
  (exemplo de função de hash aceitável)
- Uma função de hash mapeia chaves para o domínio de um array concreto. Deve ser não injectiva. Se as chaves geradas não forem números naturais, devem ser previamente condicionadas, i.e. transformadas em números naturais, novamente de forma determinista e tão uniforme quanto possível.
- Altura da árvore: $\log_2 N$

Colisões

- Acontecem quando a função de hash calcula a mesma posição do array para duas chaves diferentes, principalmente quando a capacidade da tabela é muito menor que a cardinalidade do conjunto de chaves.
- Soluções:
  a) Open Addressing
     ↳ Inserir a chave noutra posição do array, usando um método que possa ser reproduzido ao efectuar consultas.
     ↳ Método mais natural *linear probing*: resolve-se a colisão inserindo na próxima posição livre do array.
     ↳ É essencial armazenar as chaves juntamente com os valores, uma vez que na operação de consulta não basta procurar a posição respectiva à chave, é necessário repetir o *linear probing* até esta ser encontrada.
     ↳ As posições do array devem ser inicializadas como "empty" e marcadas como "deleted" quando é removido um elemento da tabela.
     ↳ Problema: formação de clusters – aumento da probabilidade de inserção em posições subsequentes às já preenchidas. Pode ser evitado recorrendo a *quadratic probing*.
     ↳ A tabela deve ser redimensionada quando necessário, assegurando um factor de

carga pequeno.
    - ↳ Vantagens: se devidamente optimizado (ex. *quadratic probing*, redimensionamento dinâmico) é boa escolha porque não é penalizador em termos de espaço e é totalmente estático (vantagens em termos de caching).
    - ↳ Desvantagens: dificuldade em lidar com remoções.
- b) Closed Addressing (Chaining)
    - ↳ Alterar a estrutura de dados de forma a permitir mais que um par *chave* → *valor* na mesma posição do array. Através da criação de uma lista ligada cujo endereço inicial é guardado no array.
    - ↳ A consulta envolverá uma pesquisa neste conjunto de pares.
    - ↳ As listas ligadas em teoria podem ser $\alpha > 1$, mas não devem crescer indefinidamente, pois o tempo de execução deixa de ser tendencialmente constante.
    - ↳ A tabela deve ser redimensionada quando necessário, assegurando um factor de carga pequeno.
    - ↳ Vantagens: de programação simples.
    - ↳ Desvantagens: custo adicional de espaço relevante.

# IV. Algoritmos Fundamentais Sobre Grafos

Conceitos, Notas e Terminologia

- Grafo orientado: par (V,E) com V conjunto finito de vértices (ou nós) e E uma relação binária em V – o conjunto de arestas (ou arcos) do grafo.
- Grafo não orientado: o conjunto E é constituído por pares não-ordenados (conjuntos com 2 elementos) neste tipo de grafo. Assim, (i,j) e (j,i) correspondem ao mesmo arco.
- Um arco (i,i) designa-se por anel. Os anéis são interditos nos grafos não-orientados.
- i e j são, respectivamente, vértices de origem e destino do arco (i,j). j diz-se adjacente a i.
- A relação de adjacência pode não ser simétrica num grafo orientada.
- Grau de entrada (de saída) de um vértice num grafo orientado é o número de arcos com destino (origem) no vértice. O grau do vértice é a soma de ambos.
- A distância d(s,v) do vértice s ao vértice v define-se como caminho mais curto entre esses vértices.
- Um sub-caminho de um caminho é uma qualquer sua sub-sequência contígua.
- Um caminho diz-se simples se todos os seus vértices são distintos.
- Um ciclo é um caminho de comprimento $\geq 1$ com início e fim no mesmo vértice. Note-se que existe sempre um caminho de comprimento 0 de um vértice para si próprio, que não se considera ser um ciclo.
- Um grafo diz-se acíclico se não contém ciclos. Um grafo orientado acíclico é usualmente designado por DAG (Directed Acyclic Graph).
- Um grafo não-orientado diz-se ligado se para todo o par de vértices existe um caminho que os liga. Os componentes ligados de um grafo são os seus maiores sub-grafos ligados.
- Um grafo é fortemente ligado se, para todo o par de vértices m,n existem caminhos de m para n e de n para m. Os componentes fortemente ligados de um grafo são os seus maiores sub-grafos fortemente ligados.

Árvores

- Uma floresta é um grafo não-orientado acíclico.
- Uma árvore é um grafo não-orientado, acíclico, e ligado.
- Árvores com raíz: a escolha de um vértice arbitrário para raíz de uma árvore define noções

de descendentes de um vértice e de sub-árvore com raíz num vértice. Existe um caminho único da raíz para qualquer vértice. Uma árvore com raíz pode também ser vista como um caso particular de grafo orientado.
- Árvores Ordenadas: uma árvore ordenada é uma árvore com raíz em que a ordem dos descendentes de cada vértice é relevante.

Representação de Grafos em Computadores

Listas de adjacências
- Consiste num vector Adj de |V| listas ligadas. A lista Adj[i] contém todos os vértices j tais que $(i,j) \in E$, ie, todos os vértices adjacentes a i.
- A soma dos comprimentos das listas ligadas é |E|.
- Se o grafo for pesado (ie, se contiver informação associada aos arcos), o peso de cada arco pode ser incluído no vértice respectivo numa das listas ligadas.
- No caso de um grafo não-orientado, esta representação pode ser utilizada desde que antes se converta o grafo num grafo orientado, substituindo cada arco (não-orientado) por um par de arcos (orientados). A representação contém informação redundante e o comprimento total das listas é 2|E|.
- O espaço necessário de memória em qualquer dos casos é $\Theta(V+E)$.

Matriz de adjacências
- É uma representação estática, apropriada para grafos densos, em que |E| se aproxima de $|V|^2$.
- Tem dimensão |V| x |V|, $A = (a_{ij})$, com $a_{ij} = 1$ se $(i,j) \in E$; $a_{ij} = 0$ em caso contrário.
- Se o grafo for pesado, o peso de cada arco pode ser incluído na respectiva posição da matriz (em vez de 1).
- Se o grafo for não-orientado a matriz de adjacências é simétrica. É possível armazenar apenas o triângulo acima da diagonal principal.
- Vantagem em relação às listas de adjacências: é imediato verificar a adjacência de dois vértices ($\Theta(1)$ sem ter de percorrer uma lista ligada).
- Espaço de memória necessário é $\Theta(V^2)$ – independente do número de arcos.

Travessias em Grafos

Pesquisa em Largura ("Breadth-first Search")
→ Dado um grafo G = (V,E) e um seu vértice s, um algoritmo de pesquisa explora o grafo passando por todos os vértices alcançáveis a partir de s.
- O algoritmo de pesquisa em largura calcula a distância (menor número de arcos) de s a cada vértice;
- Produz uma árvore (sub-grafo de G) com raíz s e contendo todos os vértices alcançáveis a partir de s;
- Nessa árvore o caminho da raíz s a cada vértice corresponde ao caminho mais curto entre os dois vértices;
- Algoritmo para grafos orientados e não-orientados.

→ Algoritmo
- Utiliza a estratégia: todos os vértices à distância k de são visitados antes de qualquer vértice à disância k+1 de s.
- Pinta os vértices (inicialmente brancos) de cinzento ou preto. "Branco" ainda não foi descoberto, "cinzento" já foi visitado mas pode ter adjacentes ainda não descobertos, "preto" só tem adjacentes já descobertos. Os "cinzentos" correspondem à fronteira entre descobertos e não-descobertos.

- A árvore de travessia em largura é expandida atravessando-se a lista de adjacências de cada vértice cinzento u; para cada vértice adjacente v acrescenta-se à árvore o arco (u,v).
- Utiliza-se uma fila de espera FIFO.
- O sub-grafo dos antecessores de G é uma árvore de pesquisa em largura (APL).
- Adequado para utilização com representação por listas de adjacências.
- Tempo de execução Θ(V+E).

Pesquisa em Profundidade ("Depth-first Search")
→ Algoritmo

- Utiliza a estratégia: os próximos arcos a explorar têm origem no mais recente vértice descoberto que ainda tenha vértices adjacentes não explorados.
- Quando todos os adjacentes a um vértice v tiverem sido explorados, o algoritmo recua para explorar vértices com origem no vértice a partir do qual v foi descoberto.
- O grafo dos antecessores de G é uma floresta composta de várias árvores de pesquisa em profundidade (APP).
- O algoritmo faz ainda uma etiquetagem dos vértices com marcas temporais; para o instante em que o vértice é descoberto e quando todos os seus adjacentes são descobertos. O algoritmo não tem, no entanto, necessariamente que produzir essa etiquetagem.
- Tempo de execução Θ(V+E).
- Utiliza-se uma fila de espera LIFO.

Vídeo BFS/DFS→ http://www.youtube.com/watch?v=zLZhSSXAwxI

Árvores Geradoras Mínimas (MST - "Minimum Spanning Trees")

- Seja G = (V,E) um grafo não-orientado, ligado. Uma árvore geradora de G é um sub-grafo (V,T) acíclico e ligado de G;
- (V,T) é necessariamente uma árvore e liga todos os vértices de G entre si;
- Associe-se a cada arco um peso numérico;
- Árvores geradoras mínimas de G são aquelas para as quais o peso total é mínimo.

→ Algoritmo de Prim para Determinação de MSTs

- O algoritmo considera em cada instante da execução o conjunto de vértices dividido em três conjuntos distintos: (1) vértices da árvore construída at´ao momento (2) os vértices na orla, alcançáveis a partir dos anteriores (3) restantes vértices. Em cada passo selecciona-se um arco (com origem em 1 e destino em 2) para acrescentar à árvore. O vértice destino desse arco também é acrescentado.
- O algoritmo de Prim selecciona sempre o arco com menor peso nestas condições.
- Estrutura geral:

```
void MST((V,E)) {
   seleccionar vértice arbitrário x para início da árvore;
   while(existem vértices na orla) {
      seleccionar um arco de peso mínimo entre um vértice da
         árvore e um vértice na orla;
      acrescentar esse arco e o respectivo vértice-destino à
         árvore;
   }
}
```

- Grafo implementado por listas de adjacências.
- São mantidos vectores adicionais para o estado de cada vértice (indicando se está na árvore, na orla, ou nos restantes), para a construção da árvore, e para o peso dos arcos candidatos.

- É mantida uma lista ligada correspondente aos vértices na orla.
- A pesquisa e remoção do arco candidato de menor peso é feita por uma travessia da orla e consulta directa dos pesos dos arcos candidatos.
- Se se colocar na árvore os nós da orla e os respectivos arcos candidatos, a substituição de um arco candidato é feita alterando-se o vector parent e o vector de pesos dos arcos candidatos.
- Tempo de execução $\Theta(V^2+E)$.

Vídeo Algoritmo de Prim → http://www.youtube.com/watch?v=YyLaRffCdk4

Caminhos Mais Curto ("Shortest Paths")

- Algoritmo de Dijkstra para determinação de caminhos mais curtos → muito semelhante ao algoritmo de Prim para MST.
- Em cada passo selecciona um vértice da orla para acrescentar à árvore que vai construindo.
- O algoritmo vai construindo caminhos cada vez mais longos (ie, com peso cada vez maior) a partir de v, dispostos numa árvore e pára quando alcançar w.
- A diferença para o algoritmo de Prim é o critério de selecção do arco candidato.
- Arcos candidatos: (1) para cada vértice z na orla, existe um caminho mais curto v, $v_1,...v_k$ na árvore construída, tal que $(v_k,z) \in E$. (2) se existem vários caminhos v, $v_1,...v_k$ e arco $(v_k,z)$ nas condições anteriores, o arco candidato (único) de z será aquele para o qual $d(v_k,v) + w(v_k,z)$ for mínimo. (3) Em cada passo o algoritmo selecciona um vértice da orla para acrescentar à árvore. Este será o vértice z com valor $d(v_k,v)+w(v_k,z)$ mais baixo.
- Tempo de execução $\Theta(V\log(V) +E)$. (???)

Vídeo Algoritmo de Dijkstra → http://www.youtube.com/watch?v=8Ls1RqHCOPw

Variantes do Problema dos Caminhos Mais Curtos
Seja G um grafo orientado,

- Single-source Shortest Paths: determinar todos os caminhos mais curtos com origem num vértice v dado e destino em cada vértice de G. Pode ser resolvido por uma versão ligeiramente modificada do algoritmo de Dijkstra.
- Single-destination Shortest Paths: determinar todos os caminhos mais curtos com destino num vértice w dado e eorigem em cada vértice de G. Pode ser resolvido por um algoritmo de resolução do problema anterior, operando sobre um grafo obtido do original invertendo-se o sentido de todos os arcos.
- All-pairs Shortest Paths: determinar caminhos mais curtos entre todos os pares de vértices de G.

Estratégias Algorítmicas – Algoritmos "Greedy"

- Podem ser usados na resolução de problemas de optimização.
- Um algoritmo greedy efectua uma sequência de escolhas; em cada ponto de decisão do algoritmo esta estratégia elege a solução que "parece" melhor.
- Nem sempre esta estratégia resulta em soluções globalmente óptimas, pelo que é necessário provar que a estratégia é adequada.
- Para que um problema seja resolúvel por uma estratégia "greedy" é condição necessária que possua sub-estrutura óptima, ou seja: (1) é efectuada uma escolha, da qual resulta um (único) sub-problema que deve ser resolvido. (2) Essa escolha é efectuada localmente sem considerar soluções de sub-problemas – o novo sub-problema resulta da escolha efectuada; a escolha greedy não pode depender da solução do sub-problema criado. (3) Trata-se de um

método “top-down”: cada problema é reduzido a um mais pequeno por uma escolha greedy e assim sucessivamente.

Fecho Transitivo de um Grafo

- O objectivo é, dado um grafo orientado G = (V,E) com o conjunto de vértices V = {1,2,...,n}, descobrir se existe um caminho em G desde i até j para todos os pares de vértices i,j ∈ V. O fecho transitivo de G é definido como o grafo G* = (V, E*), onde E* = {(i,j) : existe um caminho desde o vértice i até o vértice j em G}.
- Algoritmo:

```
void TC(int A[][], int R[][], int n) {
   /* G representado por A, fecho transitivo em R */
   R = A; /* copia matriz... */
   for (i=1 ; i<=n; i++)
     for (j=1 ; j<=n ; j++)
         for (k=1 ; k<=n ; k++)
           if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

- Algoritmo de Warshall, mais eficiente, executa em tempo $\theta(|V|^3)$:

```
void TC(int A[][], int R[][], int n) {
   /* G representado por A, fecho transitivo em R */
   R = A; /* copia matriz... */
   for (k=1 ; k<=n ; k++)
     for (j=1 ; j<=n ; j++)
        for (i=1 ; i<=n; i++)
           if (R[i][k] && R[k][j]) R[i][j] = 1;
}
```

Programação Dinâmica

- Consiste na optimização de uma definição recursiva quando há margem para armazenamento de resultados intermédios, que se calculam “bottom-up”.
- Exemplo – sequência de Fibonacci. Basta calcular os valores sequencialmente e armazená-los num vector (tempo $\Theta(n)$), às custas de espaço adicional também em $\Theta(n)$.
- Exemplo – algoritmo de Floyd-Warshall.

# V. Problemas NP-Completos

Problemas de Optimização vs Problemas de Decisão

A resolução de um problema de optimização consiste na selecção da melhor solução para outro problema. Exemplo: árvore geradora mínima – opt, escolher a melhor solução (ie, de menor peso) para o problema da determinação de uma árvore geradora (qualquer).

A cada problema de optimização está normalmente associado um problema de decisão, isto é, um problema cuja solução é uma resposta sim/não. Exemplo: árvore geradora mínima – dec: dado um valor k, existirá alguma árvore geradora para G com peso $\leq$ k?

Classe de Problemas P

- Um algoritmo é limitado polinomialmente se tem comportamento no pior caso em O(P(n)), com P(n) um polinómio em n (a dimensão do input).
- Um problema diz-se limitado polinomialmente se existe um algoritmo limitado polinomialmente que o resolve.

- A *Classe P* é constituída pelos problemas de decisão limitados polinomialmente, e inclui todos os problemas "razoáveis" mas também problemas de difícil resolução (há polinómios de crescimento muito rápido).
- É certo que um problema que não pertença a P será de resolução praticamente impossível.
- A classe P é fechada por operações diversas (exemplo +. *).

Algoritmos Não-Determinísticos

- Algoritmos de aplicação teórica, usados para classificação de problemas. Tem duas fases:
  (1) Fase não determinística: e escrita algures (em memória) uma string arbitrária s. Em cada execução do algoritmo esta string pode ser diferente.
  (2) Fase determinística: o algoritmo lê agora s e processa-a, após o que pode suceder uma de três situações: (a) o algoritmo pára com resposta sim; (b) o algoritmo pára com resposta não; (c) o algoritmo não pára.

  A primeira fase pode ser vista como uma "tentativa de adivinhar" uma solução. A segunda fase verifica se este "palpite" obtido na primeira fase corresponde ou não a uma solução para o problema.
- Estes algoritmos distinguem-se dos tradicionais pelo facto de a execuções diferentes poderem corresponder outputs (sim/não) diferentes.
- A resposta de um algoritmo A não determinístico a um input x define-se como sendo "sim" sse existe uma execução de A sobre x que produz output "sim".
  A resposta "sim" de A a x corresponde então à existência de uma solução para o problema de decisão com input x.
- O número de passos de execução de um algoritmo ND corresponde à soma dos passos das duas fases.

Classe de Problemas NP (Nondeterministic Polynomial-bounded)

- Informalmente é constituída pelos problemas de decisão em que a verificação de soluções pode ser feita em tempo polinomial.
- Um algoritmo não-determinístico A diz-se limitado polinomialmente se existe um polinómio P(x) tal que para cada input (de dimensão n) para o qual a resposta de A seja "sim", existe uma execução do algoritmo que produz output "sim" em tempo O(P(x)) – ou seja, se existe uma solução, ela pode ser escrita e verificada em tempo polinomial.
- A classe NP é constituída por todos os problemas de decisão para os quais existe um algoritmo não-determinístico limitado polinomialmente.
- $P \subseteq NP$.
- A estratégia "força bruta" resolve os problemas NP em tempo exponencial.

Problemas NP-completos

- A classe de complexidade NP-completo é o subconjunto dos problemas de decisão em NP de tal modo que todo problema em NP se pode reduzir, com uma redução de tempo polinomial, a um dos problemas NP-completo.
- Para provar que P=NP (altamente improvável) bastaria provar que um qualquer problemas NP-completo pode ser resolvido por um algoritmo limitado polinomialmente (também improvável por P=NP ser improvável).
- Estratégias de resolução: escolher o mais eficiente dos algoritmos exponenciais; concentrar a escolha na análise do caso médio em vez de pior caso; um estudo dos inputs que ocorrem com mais frequência pode levar à escolha de um algoritmo que se comporte melhor para esses inputs; escolha pode depender mais de resultados empíricos do que de uma análise rigorosa.

Algoritmos de Aproximação ou Heurísticos

- Princípio: utilizar algoritmos rápidos (da classe P) que não produzem garantidamente uma solução óptima, mas sim próxima da solução óptima.
- Muitas heurísticas utilizadas são simples e eficientes, resultado apesar disso em soluções muito próximas da ptimalidade.
- A definição de "proximidade à solução óptima" depende do problema.
- Uma solução aproximada para, por exemplo, o problema do caixeiro viajante, não é uma solução que passa por "quase todos" os vértices do grafo, mas sim uma solução que passa por todos, e cujo peso é próximo do mínimo possível.
- Por outras palavras, as soluções óptimas devem sempre ser soluções propostas por alguma execução de um algoritmo não-determinístico para o problema.

- versão 2.3 - 2013/01/15