Algoritmos Gananciosos (Greedy)

- Aplica heurística de solução, realizando uma escolha ótima local.
- Aplicável a problemas de otimização
- Em diversos problemas, a otimização local garante a otimização global ← sub Estrutura ótima.
- Características: conjunto de candidatos; função de seleção que escolhe o melhor candidato a ser incluído na solução; função de viabilidade que determina se o candidato pode ou não fazer parte da solução; função objetivo que atribui um valor a uma solução; função solução que determina se chegou a solução completa

Troco de moedas

- Sistema de moedas canónico: algoritmo gananciosa encontra sempre uma solução ótima para o problema do troco (com stock ilimitado)
- Extrair a moeda de valor mais alto que não excede o montante em falta.
- Sendo C = {1, c2, …, cn} as denominações do sistema de moedas, se o sistema não for canónico, o menor contra exemplo situa-se na gama $c_3+1<x<c_{n-1}+c_n$, basta fazer pesquisa exaustiva nessa gama para determinar se é canónico.

Escalonamento de atividades

- **Problema**: dado um conjunto de atividades, encontrar um subconjunto com o maior numero de atividades não sobrepostas
- **Input**: conjunto A de n atividades, a1, a2, …, an com si = instante de inicio e fi = instante de fim
- **Ouptut**: subconjunto R com o numero máximo de atividades compatíveis.
- Ordenar as atividades numa ordem especifica (fim mais cedo ou inicio mais tardio); escolher a melhor opção; descartar as incompatíveis com a escolhida; proceder da mesma forma.
- Sendo A – conjunto inicial de atividades; a – atividade selecionada com fim mais cedo; I – conjunto de atividades incompatíveis com a; C – conjunto de atividades Restantes. Do conjunto $\{a\}\cup I$ só pode ser selecionada no máximo uma atividade pois são mutuamente incompatíveis, que com outro critério de ordenação poderia haver mais. Desse conjunto escolhemos uma, que é o máximo possível. A atividade escolhida não tem incompatibilidade com as restantes logo a escolha de a permite maximizar o nº de atividades que se podem escolher de C.
- **Problema 2**: sequenciar tarefas minimizando o tempo médio de conclusão. Método de ordenação usado é escolher tarefas mais curtas primeiro.

Backtracking

- Algoritmos de tentativa e erro.
- Explorar um espaço de estados a procura de um estado-objetivo;
- Ao chegar a um ponto de escolha, escolher uma das opções; chegando a um "beco sem saída", retroceder ate ao ponto de escolha mais próximo com alternativas por explorar, tentar outra alternativa.
- Problema do troco com limitações de stock; sudoku; 8 rainhas; labirintos.
- Uma representação é árvore de espaço de estados: A raiz representa o inicio (0 escolhas); Nós ao nível 1 representam a primeira escolha; caminhos da raiz até às folhas representam soluções candidatas.
- Outra representação é uma árvore n-ária em que, em cada nível, se escolhe o próximo valor a incluir. Cada nó representa uma solução candidata (total 2^n)
- Tempo de execução no piro caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial.
- Variantes: Encontrar uma solução; encontrar todas as soluções (não para a exploração); encontrar a melhor solução (variante de encontrar todas as soluções com possibilidade de pruning quando a solução atual não leva a uma melhor do que as já encontradas).

Pruning

- Interromper (podar) a pesquisa e retroceder em nos que garantidamente não levam a uma solução viável (chamados nos não promissores)

Soma de Subconjuntos

- **Problema**: dado um conjunto (ou multi conjunto) $W=\{w_1,\dots,w_n\}$ de inteiros positivos e soma S a perfazer, encontra um subconjunto R de W com soma S.
- Uma possibilidade é uma árvore binária em que, em cada nível k, se decide da inclusão ou não do valor w_k . As folhas representam as soluções candidatas (soma das inclusões).
- As folhas da árvore binária representam os possíveis subconjuntos de W, em número 2^n . O numero de nos da árvore é sensivelmente o dobro $2^{(n+1)}-1$. No pior caso $T(n)=O(2^n)$.
- **Pruning**: a soma já selecionada é superior à soma a perfazer ou a soma ainda selecionável é inferior à soma a perfazer.

Divide And Conquer

- **Dividir** o problema em subproblemas que são instâncias mais pequenas do mesmo problema. **Conquistar** os subproblemas resolvendo-os recursivamente ou diretamente. **Combinar** as soluções.
- Subproblemas devem ser disjuntos (usar programação dinâmica). Dividir em subproblemas de dimensão similar para maior eficiência. Para existir divisão devem existir 2 ou mais chamadas recursivas.
- Normalmente desempenho ótimo com n threads = n cores, hyper threading n ótimo é 2 * n cores.

Merge-Sort

- Ordenar 2 subsequências de igual dimensão e juntá-las.
- $T(n)=O(n\cdot\log(n))$ tanto no pior caso como no caso médio.
- $S(n)=n$
- Otimizar com memoria auxiliar, insertion sort com n < 20, processamento paralelo.

Quicksort

- Ordenar elementos menores e maiores que pivot, concatenar.
- $T(n)=O(n^2)$ no pior caso (1 elemento menor, restantes maiores)
- $T(n)=O(n\cdot\log(n))$ no melhor caso e no caso médio (com escolha aleatória do pivot)
- $S(n)=1$

Calculo de x ^ n

- Resolução iterativa com n multiplicações: $T(n)=O(n)$

$$x^n = \begin{cases} 1, \text{ se } n=0 \\ x, \text{ se } n = 1 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}}, \text{ se } n \text{ par} > 1 \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}}, \text{ se } n \text{ ímpar} > 1 \end{cases}$$

- Resolução com divisão e conquista com numero de multiplicações reduzido para $\log(n)$
- $T(n)=O(\log(n))$ $S(n)=O(\log(n))$
- Classificação como divisão e conquista não e consensual por apenas haver uma chamada recursiva (subproblemas idênticos).

Pesquisa binária (Binary Search)

- $T(n)=O(\log(n))$
- Classificação como divisão e conquista não consensual por um dos 2 subproblemas ser vazio.

Dynamic Programming

- Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares), mas que resolução direta duplicaria trabalho com resoluções repetidas dos mesmos subproblemas.
- Duas abordagens: economizar tempo (evitar repetir trabalho) memorizando as soluções parciais dos subproblemas (gastando memoria); economizar memoria, resolvendo subproblemas por ordem que minimiza o numero de soluções parciais a memorizar (bottom-up, começando pelos casos base, mais simples).

Cálculo de número de combinações $^{n}C_{k}$

$^{n}C_{k}$	k=0	k=1	K=2	K=3	K=4	k=5
n=0	1					
n=1	1	1				
n=2	1	2	1			
n=3	1	3	3	1		
n=4	1	4	6	4	1	
n=5	1	5	**10**	10	5	1

- No exemplo $n=5$ e $k=2$
- Abordagem bottom-up. Apenas necessário manter uma lista de resultados no percurso da solução. Começar na primeira coluna com k = 0, e continuar até chegar à ultima coluna. Usado para conservar **memória**.
- $j_{max}=n-k, 0\leq j\leq j_{max}$ logo $j_{max}=3, 0\leq j\leq 3$
- $T(n,k)=O(k\cdot(n-k))$
- $S(n,k)=O(n-k)$

- Para economizar tempo, basta aplicar a técnica de memorização (**memoization**), com array ou hash map.

Problema da mochila

- **Problema**: Encontrar combinação de itens de vários tamanhos e valores que maximiza o valor total guardando-os numa mochila de capacidade limitada. Assumindo capacidades inteiras e numero de itens ilimitado.
- Calcular a melhor combinação para todas as mochilas de capacidade 1 até M (capacidade da mochila). Começar por considerar que só se pode usar o item 1 (primeira iteração), depois os itens 1 e 2, etc., e finalmente todos os itens de 1 a N (numero de itens) (ultima iteração).

◆ Caso base (i = 0; k = 1, ..., M): $\mathrm{cost}[k]^{(0)}=0$ $\mathrm{best}[k]^{(0)}=0$

◆ Caso recursivo (i = 1, ..., N; k = 1, ..., M) :

$$\mathrm{cost}[k]^{(i)}=\begin{cases}\mathrm{val}[i]+\mathrm{cost}[k-\mathrm{size}[i]]^{(i)}, \text{ se } \begin{cases}\mathrm{size}[i]\leq k\\ \wedge\\ \mathrm{val}[i]+\mathrm{cost}[k-\mathrm{size}[i]]^{(i)}>\mathrm{cost}[k]^{(i-1)}\end{cases}\\ \mathrm{cost}[k]^{(i-1)}, \text{ no caso contrário}\end{cases}$$

$$\mathrm{best}[k]^{(i)}=\begin{cases}i, \text{ no primeiro caso acima (usa o item } i)\\ \mathrm{best}[k]^{(i-1)}, \text{ no segundo caso acima (não usa o item } i)\end{cases}$$

Encher o resto

Permite usar repetidamente o item *i*
(senão, escrevíamos *i-1*)

- **Dados**: N – numero de itens (com numero ilimitado de cada item); $\mathrm{size}[i], 1\leq i\leq N$ - tamanho inteiro do item i; $val[i], 1\leq i\leq N$ - valor do item i; M – capacidade da mochila.
- **Dados de trabalho, no fim de cada iteração i de 0 a N**: $\mathrm{cost}[k], 1\leq k\leq M$ - melhor valor que se consegue com mochila de capacidade k, usando apenas itens de 1 a i; $\mathrm{best}[k], i\leq k\leq M$ - último

item selecionado para obter o melhor valor com mochila de capacidade k, usando apenas itens de 1 a i.

- **Dados de saída**: $\text{cost}[M]$ - melhor valor que se consegue com mochila de capacidade M; $\text{best}[M], \text{best}[$ M - size[best[m]]] - itens selecionados.
- $T(N, M) = O(N \cdot M) \quad S(N, M) = O(M)$

Números de Fibonacci

- Memorizar os dois últimos elementos da sequência para calcular o seguinte.

Subsequência crescente mais comprida

- Exemplo: Sequencia S = (9, 5, 2, 8, 7, 3, 1, 6, 4); Subsequência crescente mais comprida (elementos não necessariamente contíguos): (2, 3, 4) ou (2, 3, 6).
- Formulação: $s_1, \ldots, s_n$ - sequência, l_i - comprimento da maior subsequência crescente de ($s_1, \ldots, s_i$), p_i - predecessor de s_i nessa subsequência crescente.
- $l_i = 1 + max\{l_k | 0 < k < i \wedge s_k < s_i\}$ p_i = valor de k escolhido para o máximo na expressão anterior.
- Comprimento $max(l_i)$

	i	1	2	3	4	5	6	7	8	9
Sequência	si	9	5	2	8	7	3	1	6	4
Tamanho	li	1	1	1	2	2	2	1	3	3
Predecessor	pi	-	-	-	2	2	3	-	6	6

- $T(n) = O(n^2) \quad S(n) = O(n)$

Troco de moedas

Consiste em escolher usar a moeda em i ou não usar e escolher a moeda em i-1

			k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8=m
i=0	v0=-	$C_{0,k}$	0	-	-	-	-	-	-	-	-(ou ∞)
		$P_{0,k}$	-	-	-	-	-	-	-	-	-(ou 0)
i=1	v1=1	$C_{1,k}$	0	1	2	3	4	5	6	7	8
		$P_{1,k}$	-	1	1	1	1	1	1	1	1
i=2	v2=4	$C_{2,k}$	0	1	2	3	1	2	3	4	2
		$P_{2,k}$	-	1	1	1	2	2	2	2	2
i=3	v3=5	$C_{2,k}$	0	1	2	3	1	1	2	3	2
		$P_{2,k}$	-	1	1	1	2	5	5	5	2

- $C_{i,0}$ = **0;** $C_{0,k} = \infty$ **(se** k>0); $P_{0,k} = P_{i,0}$ = indefinido (ou 0)
- $C_{i,k} = C_{i-1,k}$, e $P_{i,k} = P_{i-1,k}$ para i = 1, ..., n; k = 1, ..., v_i-1
- $C_{i,k} = \min(C_{i-1,k}, 1+C_{i,k-v_i})$ para i = 1, ..., n; k = v_i, .., m
- $P_{i,k} = P_{i-1,k}$ ou *i*, conforme se escolhe 1° ou 2° arg. de min
- Cardinal final: $C_{n,m}$ Solução final: $v_{P_{n,m}}$, $v_{P_{n,m-v_{P_{n,m}}}}$, ...

- $T(n, m) = O(n \cdot m)$
- $S(n, m) = O(m)$
- m – valor a trocar.
- v0, …, vn – moedas.
- n – numero de moedas.
- $C_{i,k}$ – número mínimo de moedas que se consegue trocar k com moedas até i.
- $P_{i,k}$ - valor da última moeda usada para alcançar o mínimo.

Correção de Algoritmos

	Análise estática (teórica)	Análise dinâmica (experimental)
Eficiência temporal e espacial	complexidade assintótica	testes de desempenho; *profiling*
Funcionamento correto	prova ou argumentação sobre correção	testes pontuais ou aleatórios (*)

- Para provar que um algoritmo resolve corretamente um problema, precisamos de uma especificação rigorosa do problema e de um descrição rigorosa do algoritmo.
- **Entradas**: Dados de entrada e restrições associadas (**pré-condições**)
- **Saídas**: Dados de saída e restrições associadas (**pós-condições**), sendo que objetivos de maximização e minimização podem ser reduzidos a restrições.
- **Correção parcial:** se o algoritmo for executado com entradas que obedecem às pré-condições, então, **se terminar,** produz **saídas corretas,** ou seja, obedecem às pós-condições.
- **Correção total:** se o algoritmo for executado com entradas que obedecem às pré-condições, então **termina**, produzindo saídas que obedecem as pós-condições.

Square Root

- Pré-condições: $x \geq 0$; Pós-condições: $RESULT \cdot RESULT = X \wedge RESULT \geq 0$

Binary Search

- Pré-condições: Array ordenado; $a \neq NULL$; Operadores de ordenação definidos para o tipo T; Pós-condições: $(0 \leq RESULT < n \wedge a[RESULT] = x) \vee (RESULT = -1 \wedge x \notin a)$

Invariantes e variantes de ciclos

- A maioria dos algoritmos são iterativos, com um ciclo principal.
- Para provar que um ciclo está correto, temos de encontrar um **invariante do ciclo** que é um a expressão booleana sempre verdadeira ao longo do ciclo e mostrar que é verdadeira **inicialmente** ou seja é implicada pela pré-condição; é **mantida** em cada iteração, ou seja, é verdadeira no fim da cada iteração, assumindo que é verdadeira no início da iteração; quando o ciclo **termina**, garante (implica) a pós-condição.
- Para provar que um ciclo termina, temos de encontrar um **variante do ciclo** – uma função (nas variáveis do ciclo) **inteira**; **positiva**; **estritamente decrescente**.

Insertion Sort

Invariante do ciclo principal [I(j)] ?

- A[1, …, j-1] contém os elementos originais, mas ordenados (j = 2, …, n + 1).
- É valido **inicialmente** (j = 2), pois é óbvio que A[1..1] contém os elementos originais, mas ordenados.
- É **mantido** em cada iteração: Assume-se que o invariante se verifica no início da iteração; O algoritmo insere A[j] na posição certa em A[1..j] e incrementa j; Logo, no fim da iteração (com novo j), verifica-se o invariante.
- No **fim do ciclo** (j = n + 1) garante-se a pós-condição: Invariante refere-se a A[1..n] ou seja toda a array; Logo, implica trivialmente a pós-condição, pois é coincidente.

Variante do ciclo principal [v(j)] ?

- $n+1-j, (j=2, \ldots, n+1)$
- Inteiro, pois n e j são inteiros; Não negativo, pois o valor máximo de j é n+1; Estritamente decrescente, pois j é sempre incrementado.
- Logo o algoritmo está correto e termina (correção total).

Binary Search

Invariante do ciclo principal [I(low, high)] ?

- x só pode existir na área de pesquisa entre low e $high$
- É valido **inicialmente** (low = 1, high = n), pois a área de pesquisa é todo o array.
- É **mantido** em cada iteração: Uma vez que se assume que o array está ordenado; quando se recua $high$, excluem-se apenas elementos > x; quando se avança low, excluem-se apenas elementos < x. Não se exclui nunca o x.
- No **fim do ciclo** garante-se a pós-condição: Se o ciclo é interrompido ($A[mid]=x$), garante-se a cláusula em que se encontra x; Se o ciclo for até ao fim, a área de pesquisa fica vazia, o que, pelo invariante, implica que x não existe em A.

Variante do ciclo principal [v(low, high)] ?

- $high-low+1$ ← largura da área de pesquisa
- Inteiro, pois low e $high$ são inteiros; Não negativo, pois no pior caso $low=high$; Estritamente decrescente, pois em cada iteração ou aumenta-se o low ou diminui-se $high$.
- Logo tem correção total.
-

Graph Theory

Grafo $G=(V,E)$

- V – conjunto de **vértices** (ou nós); E – conjunto de **arestas** (ou arcos); cada aresta é um par de vértices (v,w) $v,w \in V$; se o par for ordenado, o grafo é dirigido, ou **digrafo**; um vértice w é **adjacente** a um vértice v se e só se $(v,w) \in E$; num grafo não dirigido com aresta (v,w) e logo (w,v) , w é adjacente a v e v adjacente a w.

Caminhos

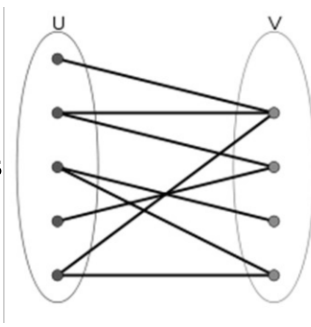
- **Caminho –** sequência de vértices $v_1,\ldots,v_n$ tais que $(v_i,v_{i+1}) \in E, 1 \leq i < n$; comprimento do caminho é o número de arestas, n – 1; se n = 1, caminho reduz-se a 1 vértice, comprimento 0;
- **Caminho simples –** todos os vértices distintos, exceto possivelmente o primeiro e o ultimo.

Ciclos

- **Ciclo (ou circuito) –** caminho de $comprimento \geq 1$ com $v_1 = v_n$; Num grafo não dirigido, requer-se que as arestas sejam diferentes
- **Anel –** caminho $v,v \Rightarrow (v,v) \in E$, comprimento 1; raro

Tipos de grafos

- **Grafo acíclico dirigido (DAG – Directed Acyclic Graph)** - Grafo dirigido sem ciclos. Para qualquer vértice v, não há nenhuma ligação dirigida começando e acabando em v.
- **Grafo simples** - Grafo sem arestas paralelas (várias adjacências, para o mesmo par de vértices) nem anéis.
- **Grafo pesado –** As arestas são etiquetadas com um peso (distância, custo, …)
- **Grafo bipartido** – Conjunto de vértices é partido em dois subconjuntos $V_1 e V_2$. Arestas ligam vértices de diferentes partições.



Conectividade

- Grafo não dirigido é **conexo** se e só se houver um caminho a ligar qualquer par de vértices.
- Digrafo com a mesma propriedade: **fortemente conexo**, se para todo $v,w \in V$ existir em G um caminho de v para w, assim como de w para v.
- Digrafo **fracamente conexo**: se o grafo não dirigido subjacente (correspondente) é conexo.

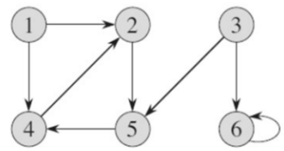
Densidade

- **Denso -** $|E| \sim O(V^2)$; **Esparso** $|E| \sim O(V)$

Representações

Matriz de adjacências

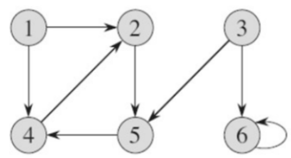
- Apropriada para grafos densos
- Elementos da matriz podem ser os pesos
- Grafos não dirigidos: matriz simétrica.

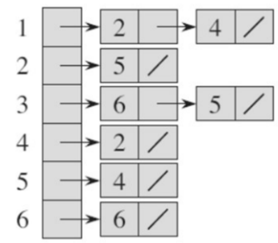


	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Lista de adjacências

- Espaço é $O(|E|+|V|)$
- Pesquisa de adjacentes em tempo proporcional ao número destes.
- Estrutura típica para grafos esparsos.
- Grafos não dirigidos: lista com dobro do espaço.





Pesquisa em profundidade (depth-first)

- Arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tenha arestas a sair dele.
- Quando todas as arestas de v forem exploradas, retorna para explorar arestas que saíram do vértice a partir do qual v foi descoberto.
- Se se mantiverem vértices por descobrir, um deles é selecionado como a nova fonte e o processo de pesquisa continua a partir dai.
- Todo o processo é repetido ate todos os vértices serem descobertos.
- Deteção de ciclos: quando um vértice adjacente está na stack de visita.
- No pior caso: $T(V,E)=O(|V|+|E|)$ $S(V,E)=O(|V|)$

Pesquisa em largura (breadth-first)

- Dado um vértice fonte s, explora-se sistematicamente o grafo descobrindo todos os vértices a que se pode chegar a partir de s (vértices adjacentes)
- Só depois é que se passa para outro vértice.
- Cria árvore de expansão em largura, com raiz s.
- Para qualquer vértice v atingível a partir de 2, o caminho na árvore BFS é o caminho mais curto no grafo (com menor número de arestas)
- No pior caso: $T(V,E)=O(|V|+|E|)$ $S(V,E)=O(|V|)$

Ordenação Topológica

Problema

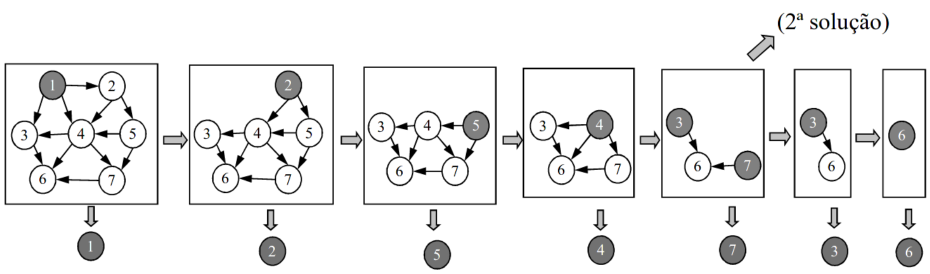
- Ordenar os vértices de um DAG tal que, se existe uma aresta (v, w) no grafo, então v aparece antes de w. Intuitivamente, dispor as setas todas no mesmo sentido; Impossível se o grafo for cíclico; Pode existir mais do que uma ordenação.

Método baseado em DFS

- Na DFS de um DAG, a pós-ordem de visita dá uma ordenação topológica inversa. No entanto o método não é genérico, pois algumas ordenações topológicas válidas não podem ser obtidas desta forma.

Método geral

- Descobrir um vértice sem arestas de chegada (indegree = 0); Imprimir/Guardar o vértice; Eliminá-lo, assim como as arestas que dele saem; Repetir o processo no grafo restante.



Refinamento da ordenação

- Simular eliminação atualizando indegree (número de arestas que chegam a v, partindo de vértices por visitar) dos vértices adjacentes; Memorizar numa estrutura auxiliar vértices por imprimir com indegree = 0 (C - conjunto de vértices por visitar cujo indegree é 0).
- Se o resultado tiver um número de vértices diferente do grafo original, então o grafo tem ciclos.

Análise do algoritmo

- Se as inserções e eliminações em C forem efetuadas em tempo constante, o algoritmo pode ser executado em tempo $O(|V|+|E|)$; o corpo do ciclo de atualização de indegree é executado no máximo uma vez por aresta; as operações de inserção e remoção na fila são executados no máximo uma vez por vértice; a inicialização leva um tempo proporcional ao tamanho do grafo.

Shortest path

- Dado um grafo pesado G = (V, E) e um vértice s, obter o caminho mais "curto" (de peso total mínimo de s para cada um dos outros vértices em G.

Variantes

- **<u>Caso base:</u>** grafo dirigido, fortemente conexo, pesos >= 0

- **<u>Grafo não dirigido</u>**: Mesmo que grafo dirigido com pares de arestas simétricas.
- **<u>Grafo não conexo</u>**: Pode não existir caminho para alguns vértices, ficando distancia infinita
- **<u>Grafo não pesado</u>**: Mesmo que peso 1 (mais curto = com menos arestas); Existe algoritmo mais eficiente para este caso do que para caso base.
- **<u>Arestas com pesos negativos</u>**: Existe algoritmo menos eficiente para este caso do que para o caso base; Ciclos com peso negativo tornam o caminho mais curto indefinido.

Grafo dirigido não pesado (BFS)

- Método básico: pesquisa em largura (BFS) + calculo de distancias:
- Marcar o vértice s com distancia 0 e todos os outros com distancia infinita; Entre os vértices já alcançados (distancia != infinito) e não processados (no próximo passo), escolher para processar o vértice v marcado com distancia mínima; Processar vértice v: analisar os adjacentes de v, marcando os que ainda não tinham sido alcançados (distancia infinita) com distancia v + 1; Voltar ao passo 2, se existirem mais vértices para processar.
- Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar, garante-se a ordem de progressão pretendida.
- Associa-se a cada vértice a seguinte informação: $dist$ - distancia ao vértice inicial (definida/definitiva ao alcançar um vértice pela primeira vez; ao alcançar um segundo caminho, distancia nunca diminui); $path$ - vértice antecessor no caminho mais curto.
- No pior caso: $T(V,E)=O(|V|+|E|)$ $S(V,E)=O(|V|)$

Grafo dirigido pesado (pesos >= 0) (Dijkstra)

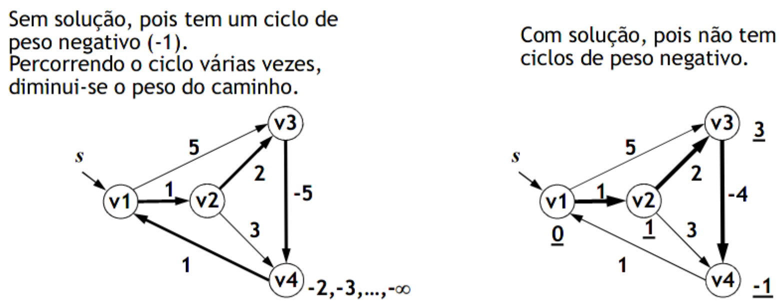
- Método básico semelhante ao caso do grafo não pesado; Distância obtém-se somando pesos das arestas em vez de 1.
- Próximo vértice a processar continua a ser o de distância mínima mas já não é necessariamente o mais antigo o que obriga a usar uma fila de prioridades (com mínimo à cabeça) em vez de uma fila simples. No entanto pode ser necessário rever a distancia de um vértice alcançado e ainda não processado (vértice na fila) o que obriga a usar uma fila de prioridades alteráveis (com uma implementação de DECREASE-KEY). Isto é necessário para garantir que a distancia ao vértice de partida dos vértices já processados não é mais alterada.
- É um algoritmo **ganancioso**: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância).
- No pior caso: $T(V,E)=O((|V|+|E|)\cdot\log|V|)$ ou $T(V,E)=O(|V|+|E|\cdot\log|V|)$ com o uso de Fibonacci heap; $S(V,E)=O(|V|)$
- $O(|V|\cdot\log|V|)$ na extração e inserção na fila de prioridades; $O(|E|\cdot\log|V|)$ - DECREASE-KEY, feito no máximo |E| vezes (uma vez por cada aresta).

Eficiência de DECREASE-KEY

- Fila de prioridades implementada com um heap
- Método naive O(n): Procurar sequencialmente no array o objeto cuja chave se quer alterar
- Método melhorado O(log n): Cada objeto colocado no heap guarda a sua posição (índice) no array, logo não é necessário procurar o objeto no array; Introduz um overhead mínimo nas inserções e eliminações (atualizar o índice no objeto).
- Método otimizado: O(1) com o use de Fibonacci heaps.

Arestas com peso negativo sem ciclos negativos (Bellman-Ford)

- Pode ser necessário processar cada vértice mais do que uma vez. Se existirem ciclos com peso negativo, o problema não tem resolução. Não existindo ciclos com peso negativo, o problema é resolúvel em em tempo $O(|E| \cdot |V|)$ pelo algoritmo de Bellman-Ford.



- Em cada iteração i (i de 1 até $|V|-1$), o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até i arestas (e possivelmente alguns mais longos) (**invariante do ciclo principal**). Como no máximo o caminho mais comprido, sem ciclos, tem $|V|-1$ arestas, basta executar no máximo $|V|-1$ iterações do ciclo principal para assegurar que todos os caminhos mais curtos são encontrados.
- No final é executada mais uma iteração para ver se alguma distância pode ser melhoradas; se for o caso, significa que há um caminho mais curto com $|V|$ arestas o que só pode acontecer se existir pelo menos um ciclo de peso negativo.
- É um caso de **programação dinâmica** (utilização de resultados anteriores, casos mais simples).

Grafos acíclicos

- Simplificação do algoritmos de Dijkstra: Processam-se os vértices por **ordem topológica**; Suficiente para garantir que um vértice processado jamais pode vir a ser alterado, pois não há arestas novas a entrar; Pode-se combinar a ordenação topológica com a atualização das distancias e caminhos numa só passagem
- $T(V,E)=O(|V|+|E|)$

Caminho mais curto entre dois vértices

- Não se conhece algoritmo mais eficiente a resolver este problema do que a resolver o mais geral (de um vértice para todos os outros).

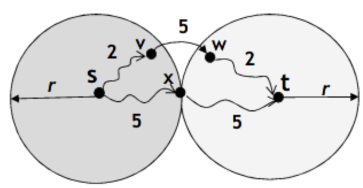
- Otimização: parar assim que chega a vez de processar o vértice de destino.

Dijkstra

- Uma vez que o algoritmo processa os vértices por distancias crescentes ao vértice de partida, é inspecionado um circulo em torno de *s* de raio igual à distância entre *s* e *t*.

Pesquisa bidirecional

- Executar o algoritmo de Dijkstra no sentido de s para t e em sentido inverso de *s* para *t* (no grafo invertido), alternando entre um e outro. Terminar quando se vai processar um vértice *x* já processado na outra direção (podendo o caminho mais curto passar por *x* ou não).
- Manter a distância μ do caminho mais curto conhecido entre s e t: ao processar uma aresta (v, w) tal que *w* já foi processado na outra direção, verificar se o correspondente caminho *s*-*t* melhora μ (ao processar o vértice v, logo não termina).
- Retornar a distância μ e o caminho correspondente.



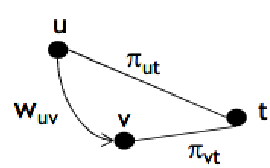
Área processada ~ $2\pi r^2$, em vez de ~ $4\pi r^2$ na pesquisa unidirecional.

Ganho (*speedup*) ~2x

Pesquisa orientada

- **Algoritmo A***: escolher para processar o vértice v com valor **mínimo de** $d_{sv}+\pi_{vt}$, parando quando se vai processar o vértice t. d_{sv} - distância mínima conhecida de *s* a *v* (como no algoritmo de Dijkstra); π_{vt} - **estimativa por baixo da distância mínima de *v* a *t*** (função potencial).
- Em geral, não garante o ótimo. Em certos casos, garante o ótimo, por exemplo quando pesos em arestas são distâncias em km e π_{vt} é a distância Euclidiana (em linha reta) de *v* a *t*.
- Equivale a aplicar o algoritmo de Dijkstra com pesos das arestas modificados $w'_{uv}=w_{uv}-\pi_{ut}+\pi_{vt}$, somando-se no final π_{st} .
- Pode ser combinado com pesquisa bidirecional.

- Pela desigualdade triangular, garante-se $\pi_{ut} \le w_{uv} + \pi_{vt}$, logo $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt} \ge 0$.



- O peso ao longo de um caminho $(s, v_1, v_2, \ldots, v_k)$, fica igual ao do grafo original, acrescido de $\pi_{st} - \pi_{v_k t}$ (pois os potenciais intermédios cancelam-se)

- Logo, escolher o vértice *v* com menor $d_{sv} + \pi_{vt}$ no grafo modificado (A*), é o mesmo que escolher o vértice com menor d_{sv} no grafo original (Dijkstra)

Redes hierárquicas (highway networks)

- Pré-processamento decompõe a rede em vários níveis hierárquicos (local, highway, super-highway)

Caminho mais curto entre todos os pares de vértices

- Execução repetida do algoritmo de Dijkstra (ganancioso) $O(|V|\cdot(|V|+|E|)\cdot\log|V|)$ - bom se o grafo for esparso $|E|\sim|V|$.
- **Algoritmo de Floyd-Warshall**, programação dinâmica: $O(|V^3|)$
  - Melhor que o anterior se o grafo for denso $|E|\sim|V^2|$; Baseia-se em **matriz de adjacências** $W[i,j]$ com pesos.
  - Invariante: em cada iteração k (de 0 a $|V|$), D[i,j] tem a distância mínima do vértice i a j, usando apenas vértices intermédios do conjunto {1, …, k}

▪ Inicialização (k=0):

$D[i,j]^{(0)} = W[i,j] \qquad P[i,j](0) = nil$

▪ Recorrência (k=1,…,|V|):

$D[i,j]^{(k)} = \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)})$

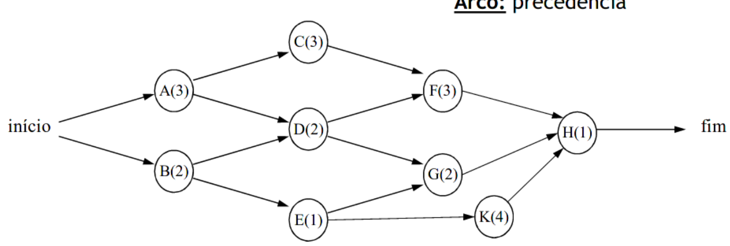
- Valor de $P[i,j]^{(k)}$ é atualizado conforme o termo mínimo escolhido

Aplicação a gestão de projetos

Grafo Nó-Atividade / Grafo Nó-Evento

DAG

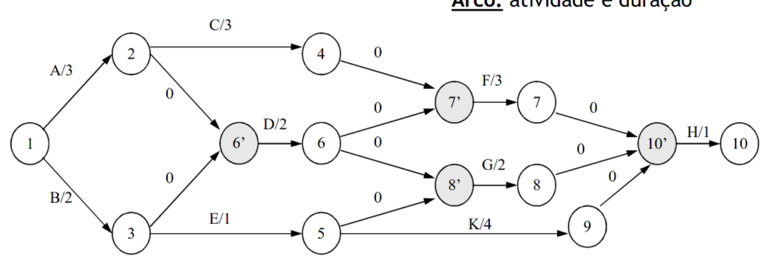
Nó: atividade e duração associada
Arco: precedência



Qual a duração total mínima do projeto?

Que atividades podem ser atrasadas e por quanto tempo (sem aumentar a duração do projeto)?

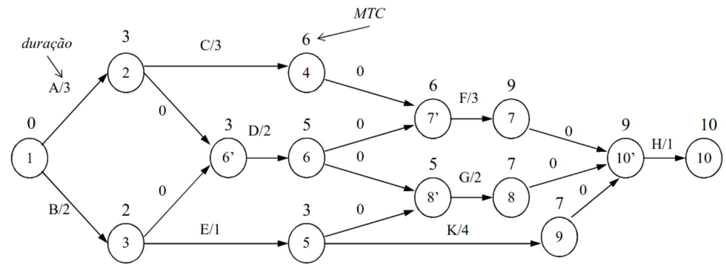
Nó: evento - completar atividade
Arco: atividade e duração



Introduzem-se nós e arcos extra para garantir precedências no caso de atividades com mais que uma antecessora

Menor Tempo de Conclusão

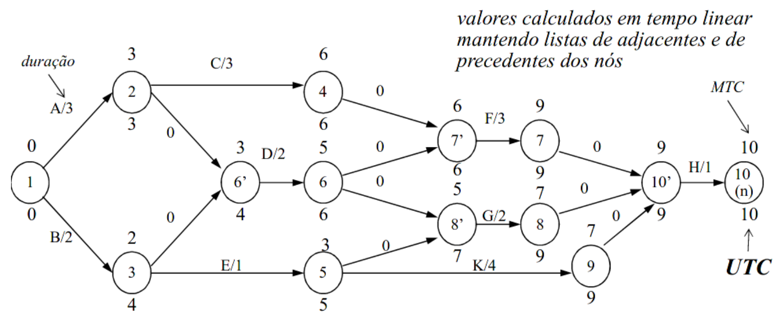
- Caminho mais comprido do evento inicial ao nó de conclusão da atividade: adaptar algoritmo de caminho mais curto para grafos acíclicos (ordem topológica).



- $MTC(1)=0 \;\; ; \;\; MTC(w)=max\{MTC(v)+c(v,w) \mid (v,w) \in E\}$

Último Tempo de Conclusão

- O tempo mais tarde que uma atividade pode terminar sem comprometer as que se lhe seguem (usando uma ordem topológica inversa)



- $UTC(n)=MTC(n) \;\; ; \;\; UTC(v)=min\{UTC(w)-c(v,w) \mid (v,w) \in E\}$

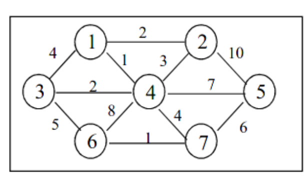
Folga nas atividades

- $folga(v,w)=UTC(w)-MTC(v)-c(v,w)$
- **Caminho Crítico:** só atividades de folga nula (há pelo menos 1).

Árvore de expansão mínima (minimum spanning trees)

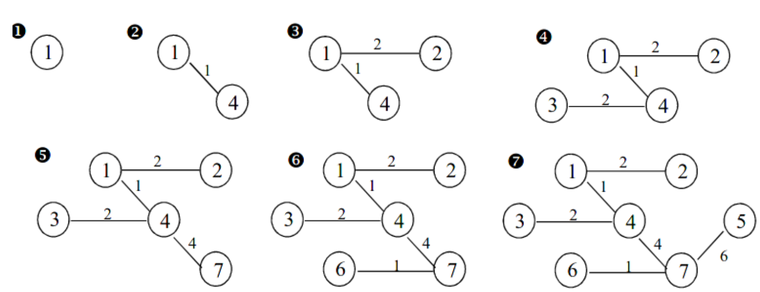
- Árvore que liga todos os vértices do grafo usando arestas com um custo total mínimo. É um grafo conexo acíclico, com um número de arestas $= |V|-1$
- Só pode ser aplicado a grafos não dirigidos. O grafo tem de ser conexo.
- Minimizar o tamanho total da rede.

Algoritmo de Prim



última tabela

v	known	dist	path
1	1	0	0
2	1	2	1
3	1	2	4
4	1	1	1
5	1	6	7
6	1	1	7
7	1	4	4



- Expandir a árvore por adição sucessiva de arestas e respetivos vértices. Critério de seleção: escolher a aresta (u, v) de menor custo tal que u já pertence à árvore e v não (ganancioso). Tem inicio num vértice qualquer.
- Idêntico ao algoritmo de Dijkstra para o caminho mais curto. Para cada vértice é guardado o custo mínimo das arestas que ligam a um vértice já na árvore - $dist(V)$; o ultimo vértice a alterar $dist(V)$ - $path(V)$; um indicador se o vértice já foi processado, ou seja, se já pertence à árvore - $known(V)$

- Após a seleção do vértice v, para cada w não processado, adjacente a v, $dist(w)=min\{dist(w),cost(v,w)\}$ ao invés de $dist(w)=min\{dist(w),dist(V)+cost(v,w)\}$
- $O(|V^2|)$ - sem fila de prioridade; $O(|E|\cdot\log|V|)$ - com fila de prioridade.

Algoritmo de Kruskal

- Analisar as arestas por ordem crescente de peso e aceitar as que não provocarem ciclos (ganancioso). Manter uma floresta, inicialmente com um vértice em cada árvore (há $|V|$). Adicionar uma aresta é fundir duas árvores. Quando o algoritmo termina há só uma árvore.
- Aceitar arestas implica utilizar algoritmos de união e de procura de conjuntos disjuntos que são representados como árvores. Se dois vértices pertencem à mesma árvore/conjunto, mais uma aresta entre eles provoca um ciclo. Se são de dois conjuntos disjuntos, aceitar a aresta é aplicar uma união.

```
while(edgesAccepted < NUM_VERTICES -1 ) {
      Edge e = h.deleteMin();     // e = (u,v)
      SetType uset = s.find(u);
      SetType vset = s.find(v);
      if (uset != vset) {
            edgesAccepted++;
            s.union(uset, vset);
      }
}
```

- A seleção de arestas é melhor feita com uma fila de prioridade em tempo linear e ordenar pelo peso.
- No pior caso $O(|E|\cdot\log|E|)$ devido às operações na fila e como $|E|\leq V^2$, $\log|E|\leq 2\cdot\log|V|$, logo a eficiência também é $O(|E|\cdot\log|V|)$.

Conectividade (Connectivity)

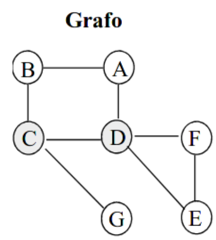
Grafo não dirigido

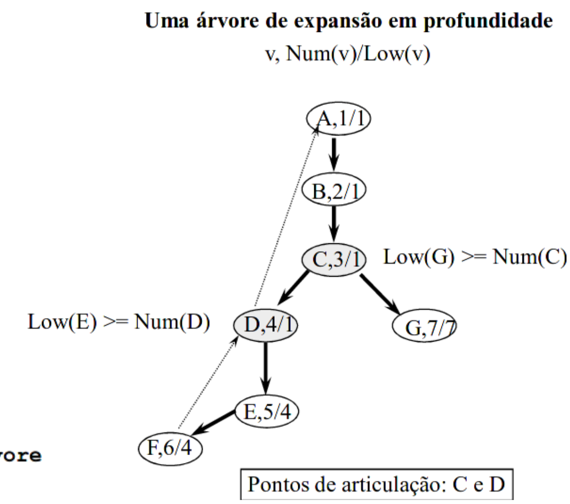
- Um grafo não dirigido é conexo se e só se uma pesquisa em profundidade, a começar em um qualquer vértice, visita todos os vértices do grafo.

Biconectividade e Pontos de Articulação

- Grafo conexo não dirigido é **biconexo** se não existe nenhum vértice cuja remoção torne o resto do grafo desconexo (**Ponto de Articulação**).
- **Algoritmo de deteção de pontos de articulação**: Início num vértice qualquer. Pesquisa em profundidade, numerando os vértices ao visitá-los - $Num(v)$, em pré-ordem (antes de visitar adjacentes. Para cada vértice v, na árvore de visita em profundidade, calcular $Low(v)$: **o menor número de vértice que se atinge com zero ou mais arestas na árvore e possivelmente uma aresta de retorno** (em qualquer ponto da árvore de DFS, quão próximo se pode chegar à raiz). Vértice v é ponto de articulação se tiver um filho *w* tal que $Low(w)\geq Num(v)$ (a partir de um descendente *w* de *v* o mais próximo da raiz nunca será menor que o nível onde está localizado o vértice v). A raiz é ponto de articulação se e só se tiver mais que um filho na árvore.

- $Low(v)$ é mínimo de: $Num(v)$; o menor $Num(w)$ de todas as arestas (v, w) de retorno; o menor $Low(w)$ de todas as arestas (v, w) (adjacentes) da árvore.
- Na visita de profundidade, inicializa-se $Low(v)=Num(v)$ antes de visitar adjacentes. Vai-se atualizando o valor depois da visita a cada adjacente.
- $O(|E|+|V|)$



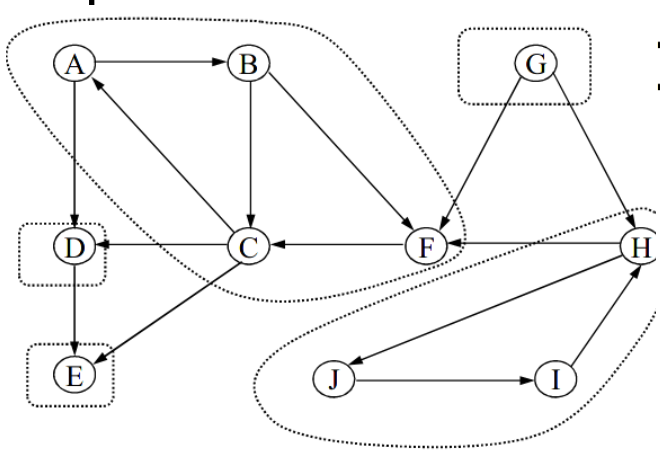



```
v.visited = true;
v.low = v.num = counter++;
for each w adjacent to v
     if( !w.visited ) {                // ramo da árvore
          w.parent = v;
          findArt(w);
          v.low = min(v.low, w.low);
          if(w.low >= v.num )
               System.out.println(v, "Ponto de articulação");
     }
     else
        if ( v.parent != w )           //aresta de retorno
          v.low = min(v.low, w.num);
```

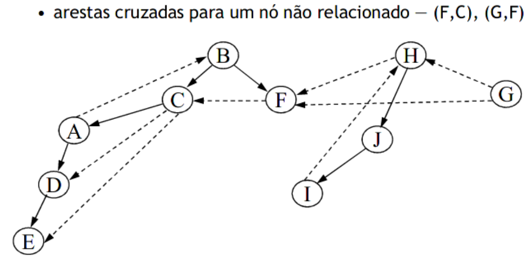
Se adjacente for pai direto, nada acontece.

Grafo dirigido

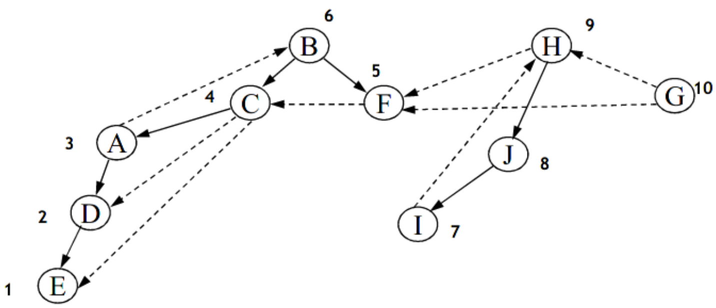
Componentes fortemente conexos

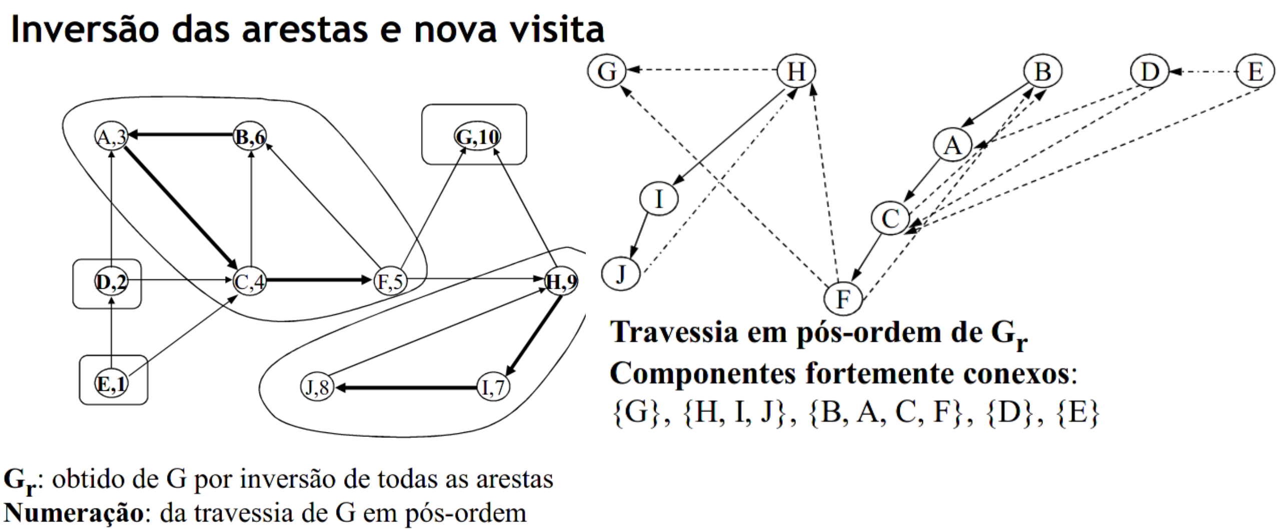


- Pesquisa em profundidade induz uma árvore/floresta de expansão
- Para além das arestas genuínas da árvore, há arestas para vértices já marcados
  - arestas de retorno para um antepassado – (A,B), (I,H)
  - arestas de avanço para um descendente – (C,D), (C,E)
  - arestas cruzadas para um nó não relacionado – (F,C), (G,F)



- **Método**: Pesquisa em profundidade no grafo G determina floresta de expansão, numerando vértices em pós-ordem (ordem inversa de numeração em pré-ordem). Inverter todas as arestas de G (grafo resultante é Gr). Segunda pesquisa em profundidade, em Gr, começando sempre pelo vértice de numeração mais alta ainda não visitado. Cada árvore obtida é um componente fortemente conexo, ou seja, **a partir de um qualquer dos nós pode chegar-se a todos os outros**.



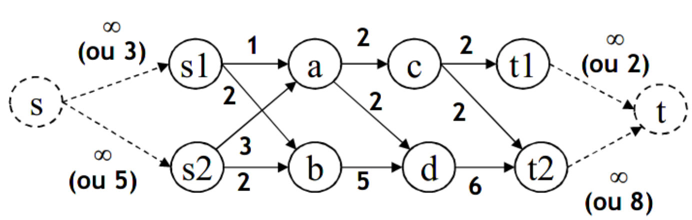


Fluxo máximo em Redes de Transporte

- Modelar fluxos conservativos entre dois pontos através de canais com capacidade limitada.
- s : fonte (produtor); t: poço (consumidor); fluxo não pode ultrapassar a capacidade da aresta; soma dos fluxos de entrada num vértice intermédio é igual à soma dos fluxos de saída (conservativo).
- Por vezes as arestas têm custos associados (custo de transportar uma unidade de fluxo).

Redes com múltiplas fontes e poços

- Se a rede tiver custos nas arestas, as arestas adicionadas têm custo 0.



Problema do fluxo máximo

- Encontrar um fluxo de valor máximo (fluxo total que parte de s / chega a t)