Resumos CAL
Notas para exame.

Índice

1. Técnicas de Conceção
   1.1 Programação Dinâmica
   1.2 Algoritmos Gananciosos
   1.3 Algoritmos de Retrocesso
   1.4 Divisão e Conquista
   1.5 Funcionamento Correto
2. Grafos
   2.1 Conceitos
   2.2 Pesquisa e Ordenação
   2.3 Caminho mais curto
   2.4 Árvore de expansão mínima
   2.5 Conectividade
   2.6 Fluxo máximo em redes de transporte
   2.7 Fluxo de custo mínimo
   2.7 Circuito de Euler
   2.8 Carteiro Chinês
   2.9 Emparelhamento e casamentos estáveis
3. Strings
   3.1 Pesquisa exata
   3.2 Pesquisa aproximada
   3.3 Compressão de texto
4. Problemas NP-Completos
   4.1 Problemas de Decisão
   4.2 Classe de problemas NP
   4.3 Resolução de problemas
5. Resumo dos algoritmos
6. Kahoots

Técnicas de Conceção

Programação Dinâmica

Aplicabilidade e Abordagem

Problemas resolúveis recursivamente (solução é uma
combinação de soluções de subproblemas similares) ... Mas em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema).
Desta forma, procura-se evitar a resolução repetido de subproblemas sobrepostos.

Abordagem

1. Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos subproblemas (gastando memória!)
    - **Memoization** - guardar resultados em array ou hashmap, etc, de forma a evitar repetição de trabalho

2. Economizar memória, resolvendo subproblemas por ordem que minimiza nº de soluções parciais a memorizar (bottom-up, começando pelos casos base)

Problema de programação linear: problema de otimização em que a função objetivo e as restrições envolvem combinações lineares das variáveis de decisão (no caso geral não resolúvel em tempo polinomial).

Combinações nCk
Calculado em tempo de O(k(n-k))

Números de fibonacci
Calculado em tempo O(n) e espaço O(1)

Subsequência crescente mais comprida
Dada uma lista, encontrar a sequência crescente com maior tamanho.

- s~1~ , ..., s~n~ - sequência
- l~i~ – compr. da maior subseq. crescente de (s~1~ , ..., s~i~) terminando em s~i~
- p~i~ – predecessor de s~i~ nessa subsequência crescente
- l~i~ = 1 + max { l k | 0 < k < i ^ s~k~ < s~i~ } (max{} = 0)
- p~i~ = valor de k escolhido para o máx. na expr. de l~i~
- Comprimento final: max(l~i~)

	i	0	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

Resposta: (2, 3, 6)

Problema da mochila
Um ladrão tem mochila de capacidade x, e quer saber qual a combinação de itens que pode levar para maximizar o valor do roubo (sendo que cada item tem um valor e um peso).

Formalização como problema de programação linear

- Dados
    - capacidade da mochila - m
    - tamanhos dos itens 1 até n, s~i~
    - valores dos itens 1 até n, v~i~
- Variáveis de decisão
    - nº de cópias de cada item, x~i~
- Função objetivo

- Maximizar $\sum_{i=1}^{n} v_i x_i$

- Restrição

$$\sum_{i=1}^{n} s_i x_i \leq m$$

Formulação recursiva

- O valor máximo que se consegue colocar numa mochila de capacidade k ($\in \mathbb{N}$), usando itens $1,\ldots, i$ de tamanho $s_1,\ldots, s_i$ ($\in \mathbb{N}$) e valor $v_1,\ldots, v_i$ pode ser dado pela função:

Usando item i

$$f(i,k) = \begin{cases} 0, & se\ k = 0\ \vee i = 0 \\ v_i + f(i, k - s_i), & se\ \ s_i \leq k \wedge v_i + f(i,k-s_i) > f(i-1,k) \\ f(i-1,k), & noutros\ casos \end{cases}$$

Não usando item i

- O último item na solução ótima é dado pela função:

$$g(i,k) = \begin{cases} 0\ (nenhum), & se\ \ k = 0\ \vee i = 0 \\ i, & se\ \ s_i \leq k \wedge v_i + f(i,k-s_i) > f(i-1,k) \\ g(i-1,k), & noutros\ casos \end{cases}$$

- O valor ótimo é $f(n,m)$ c/ itens $g(n,m)$, $g(n,m\text{-}s_{g(n,m)})$, ...

Analisando a função f:
Em cada 'iteração' é preciso analisar se o item atual deve ser usado ou não. Por isso é preciso testar se o valor que resultaria do seu uso é superior ao que resultaria se não fosse usado (ver segunda condição).
Além disso, é preciso ter em atenção a capacidade máxima da mochila.

`Algoritmo

```
int f[m+1] = {0}; // iniciado c/ 0’s (i=0)
int g[m+1] = {0}; // iniciado c/ 0’s (i=0)
for (int i = 1; i <= n; i++ )
    for (int k = s[i]; k <= m; k++)
        if (v[i] + f[k-s[i]] > f[k]) {
            /*
            como k é percorrido por ordem
            crescente f[k-s[i]] já tem valor
            da iteração i
            */
            f[k] = v[i] + f[k-s[i]];
            g[k] = i;
    }
// impressão de resultados (valor e itens)
print(f[m]);
for(int k = m; k > 0 && g[k] > 0; k -= s[g[k]])
    print(g[k]);
```

Tempo: O(nm)
Espaço: O(m)

Algoritmos Gananciosos

Algoritmo que procura realizar a escolha ótima localmente, esperando que leve no final a um ótimo global.
Aplicável a problemas de otimização.

Exemplos

Solução ótima

- Problema do troco, desde que não haja falta de stock e o sistema de moedas esteja bem concebido
- Problema de escalonamento
- Árvores de expansão mínima
- Dijkstra, para cálculo do caminho mais curto num grafo
- Codificação de Huffman

Solução não ótima

- Problema da mochila

Problema do Escalonamento de Atividades
Dado um conjunto de atividades, escolher o máximo de atividades possíveis de se atender.

Estratégia gananciosa:

- ordenar atividades por hora de fim
- em cada passo, escolher primeira atividade da lista
- remover atividades incompatíveis (começam antes de a atividade atual acabar)
- continuar da mesma forma para as restantes

PseudoCódigo

```
A //conjunto de atividades
R //conjunto de atividades a atender
While A not empty
    a <- ai | earliest finishing time
    R <- R uniun {a}
    A <- A \ {aj pertence A | aj não é compatível com ai } (incl. ai )
EndWhile
Return R
```

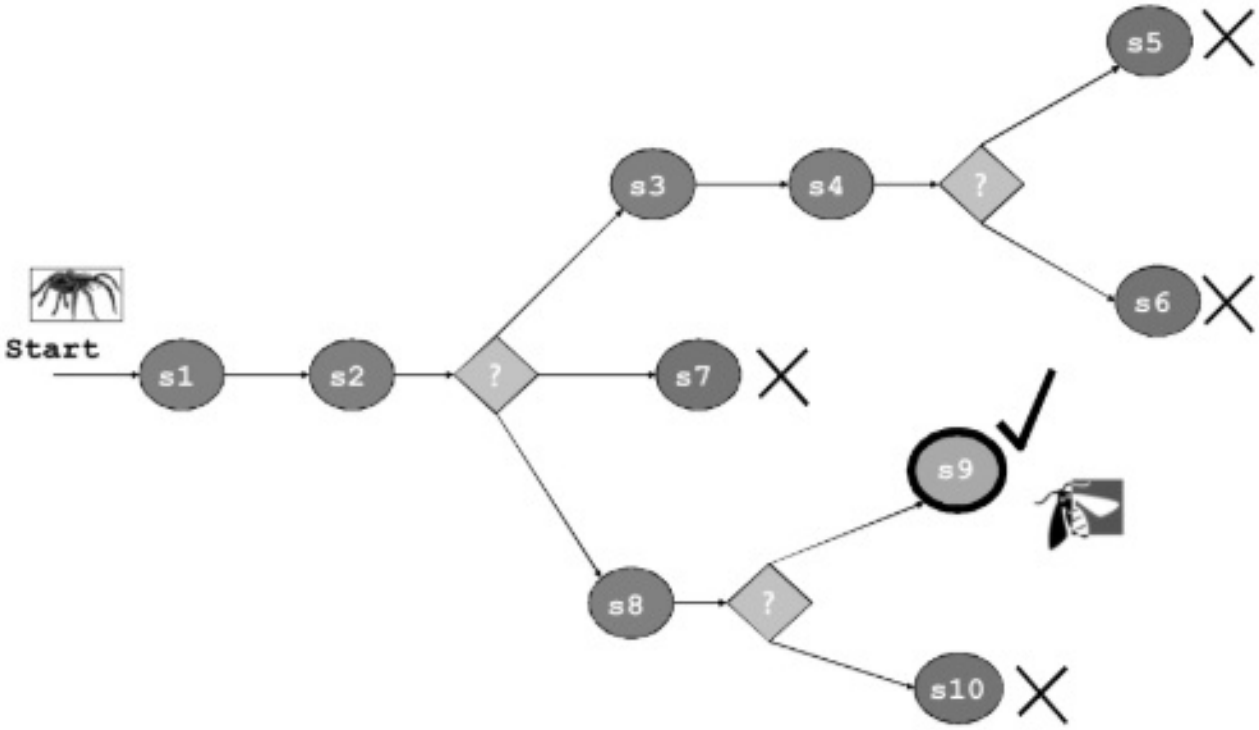
Prova de otimalidade
No exemplo e algoritmo dados sejam:

- A – conjunto inicial de atividades
- a – atividade selecionada com fim mais cedo
- I – conj. de atividades incompatíveis com a
- C – conj. de atividades restantes

Do conjunto {a} union I, só pode ser selecionada no máximo uma atividade (pois são mutuamente incompatíveis)
Desse conjunto, escolhemos uma, que é o máx. possível
A atividade escolhida (a) não tem incompatibilidade com as restantes (C), logo a escolha de a permite maximizar o no de atividades que se podem escolher de C

Algoritmos de Retrocesso

Contexto



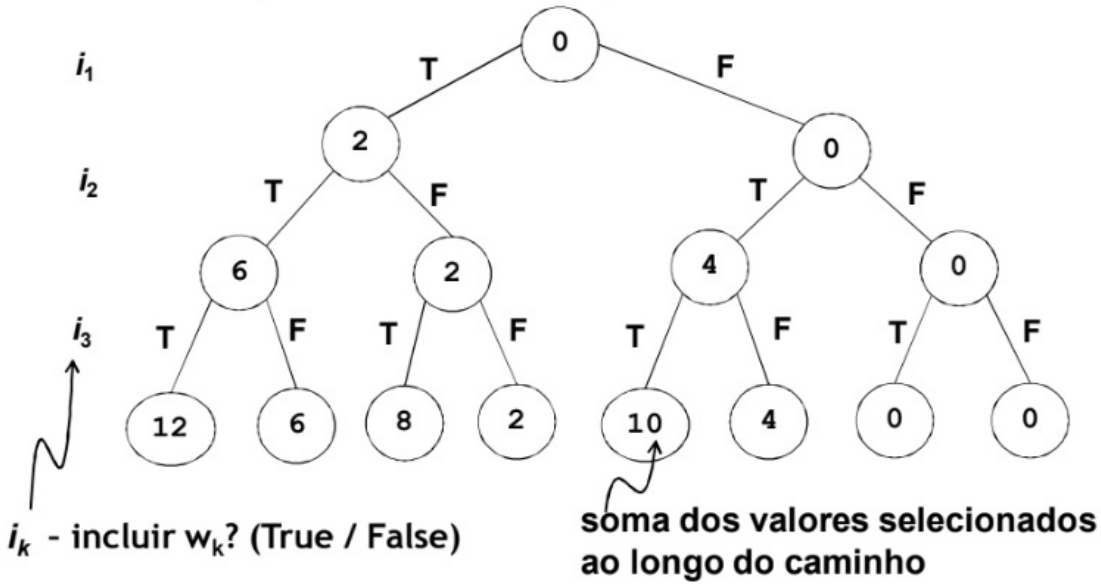
- Explorar um espaço de estados à procura de um estado-objetivo.
- Estado = estado e jogo, subproblema, solução parcial, etc.
- Sem algoritmos eficientes que levem diretamente ao objetivo.

Estratégia

- Ao chegar a um ponto de escolha (com vários estados seguintes), escolher uma das opções e prosseguir a exploração.
- Chegando a um "beco sem saída", retroceder até ao ponto de escolha mais próximo com alternativas por explorar, e tentar outra alternativa.

Árvore de espaço de estados binária

- 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.



Exemplos de aplicação

- Problema do troco com limitações de stock
- Sudoku
- 8 rainhas
- Labirintos

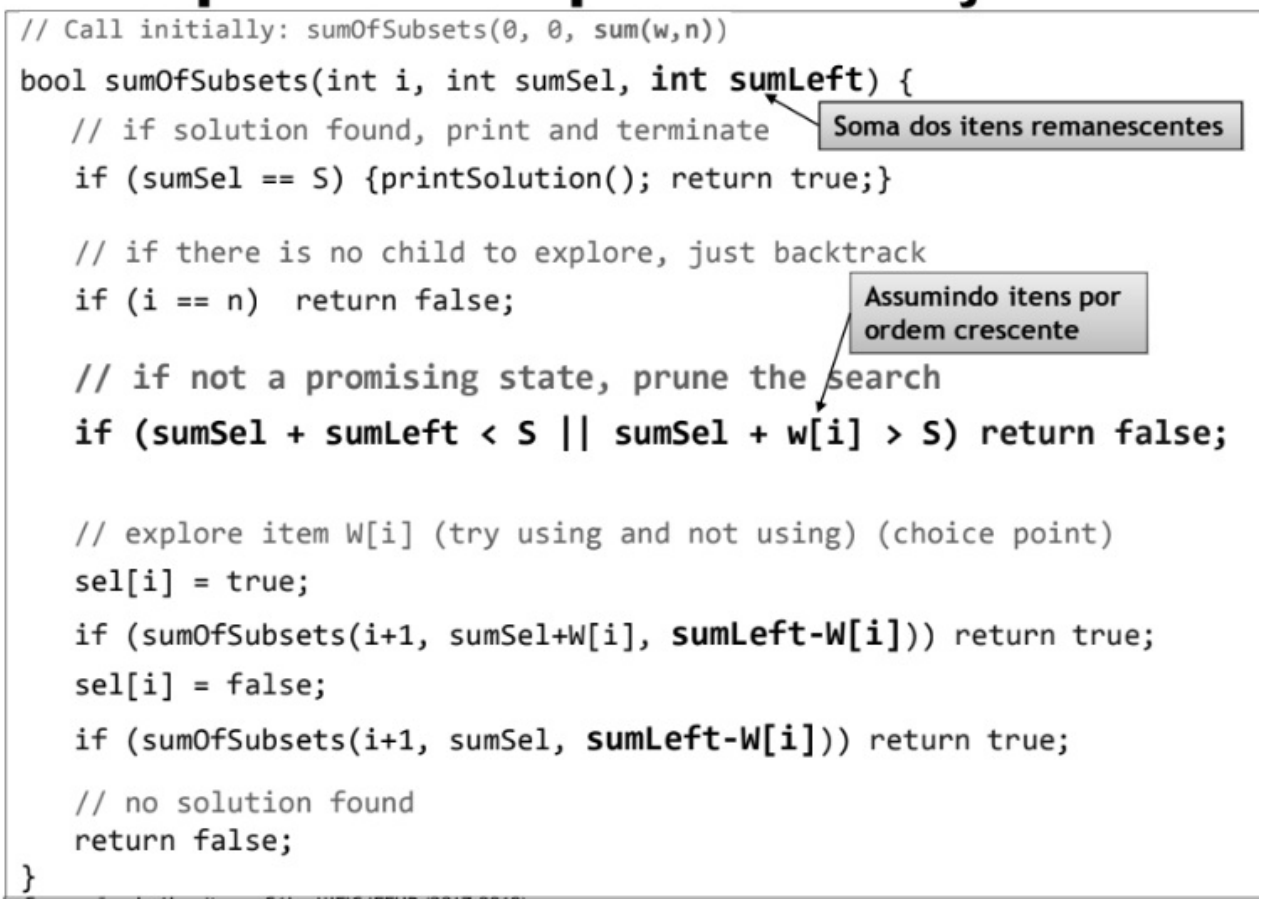
(No print, em i3, o penúltimo valor é 6, pois quando é True é adicionado o valor (6) ao anterior (0).

Eficiência temporal

- Tempo de execução no pior caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial.

Exemplo no problema da soma de subconjuntos:

Exemplo de implementação



Como se pode melhorar?

Poda da pesquisa (pruning)

- Interromper (podar) a pesquisa e retroceder em nós que garantidamente não levam a uma solução viável (chamados nós não promissores);
- No problema da soma de subconjuntos, pode-se podar a pesquisa quando:
    - a soma já selecionada é superior à soma a perfazer.
    - a soma ainda selecionável é inferior à soma a perfazer.
- A uma árvore de espaço de estados que contém apenas nós expandidos chama-se árvore de espaço de estados podada;

Outras otimizações

- Combinar com algoritmo ganancioso para procurar chegar mais rapidamente à solução (ou a uma boa solução quando se procura o ótimo) **Por exemplo**, no problema do troco ou da soma de subconjuntos, começar a pesquisa pelos valores mais elevados.
- Combinar com técnicas de memorização para evitar explorar repetidamente o mesmo nó, na presença de caminhos paralelos ou ciclos. **Por exemplo**, ao pesquisar num espaço de estados em forma de grafo, marcar os nós já visitados

- Em conjunto com técnica de poda da pesquisa, podem melhorar o desempenho mas podem continuar a existir casos patológicos com tempo exponencial.

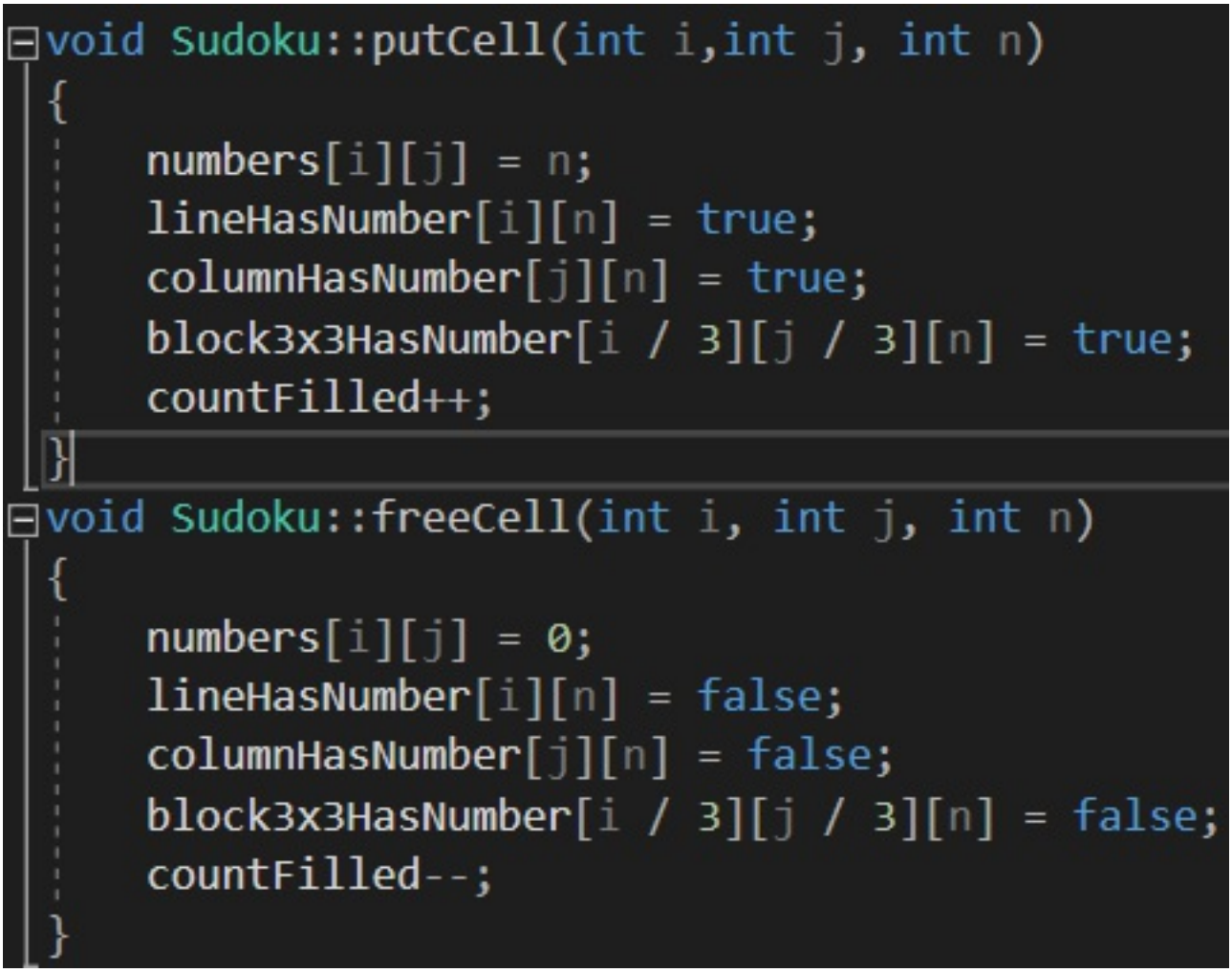
```
bool Sudoku::solveRec(int x, int y)
{
    if (isComplete())
    {
        return true;
    }

    for (int i = 1; i <= 9; i++)
    {
        if (lineHasNumber[x][i] == false &&
            columnHasNumber[y][i] == false &&
            block3x3HasNumber[x / 3][y / 3][i] == false)
        {
            putCell(x, y, i);

            pair<int, int> cell = getunsolvedCell();

            if (solveRec(cell.first, cell.second) == false)
            {
                freeCell(x, y, i);
            }
            else return true;
        }
    }
    return false;
}
```

```
41  bool Labirinth::findGoalRec(int x, int y) {
42      visited[x][y] = true;
43
44      if (labiринth[x][y] == 2)
45      {
46          return true;
47      }
48      if (labiринth[x - 1][y] != 0 && !visited[x - 1][y])
49      {
50          if (findGoalRec(x - 1, y)) return true;
51      }
52      if (labiринth[x + 1][y] != 0 && !visited[x + 1][y])
53      {
54          if (findGoalRec(x + 1, y)) return true;
55      }
56      if (labiринth[x][y+1] != 0 && !visited[x][y+1])
57      {
58          if (findGoalRec(x, y + 1))  return true;
59      }
60      if (labiринth[x][y-1] != 0 && !visited[x][y-1])
61      {
62          if (findGoalRec(x, y - 1)) return true;
63      }
64      else  return false;
65  }
66
67  bool Labirinth::findGoal(int x, int y){
68      this->initializeVisited();
69      return findGoalRec(x, y);
70  }
```



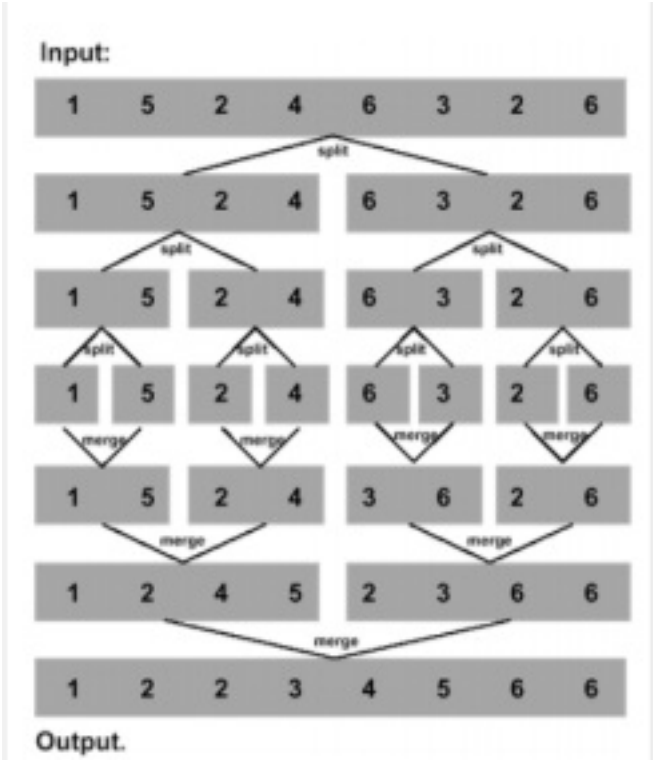
```
void Sudoku::putCell(int i,int j, int n)
{
    numbers[i][j] = n;
    lineHasNumber[i][n] = true;
    columnHasNumber[j][n] = true;
    block3x3HasNumber[i / 3][j / 3][n] = true;
    countFilled++;
}
void Sudoku::freeCell(int i, int j, int n)
{
    numbers[i][j] = 0;
    lineHasNumber[i][n] = false;
    columnHasNumber[j][n] = false;
    block3x3HasNumber[i / 3][j / 3][n] = false;
    countFilled--;
}
```

Algoritmos de Divisão e Conquista

- Dividir o problema em subproblemas que são instâncias mais pequenos do mesmo problema.
- Conquistar os subproblemas resolvendo-os recursivamente; se os subproblemas forem suficientemente pequenos, esolvem-se diretamente.
- Combinar as soluções dos subproblemas para obter a solução do problema original.
- Subproblemas devem ser disjuntos (senão, usar programação dinâmica);
- Para existir divisão, devem existir 2 ou mais chamadas recursivas;
- Dividir em subproblemas de dimensão similar para maior eficiência;
- Algoritmos adequados para processamento paralelo;

Exemplo: ordenação de arrays

- Mergesort: Ordenar 2 subsequências de igual dimensão e juntá-las.
T(n) = O(n log n), tanto no pior caso como no caso médio.



Algoritmo em pseudo-código

```
// Sorts sequence (array) A between indices p and r.
Merge-Sort(A, p, r)
    if p < r then
        q ← ⌊(p + r) / 2⌋
        Merge-Sort(A, p, q) ‖ Merge-Sort(A, q + 1, r)
        Merge(A, p, q, r)
```

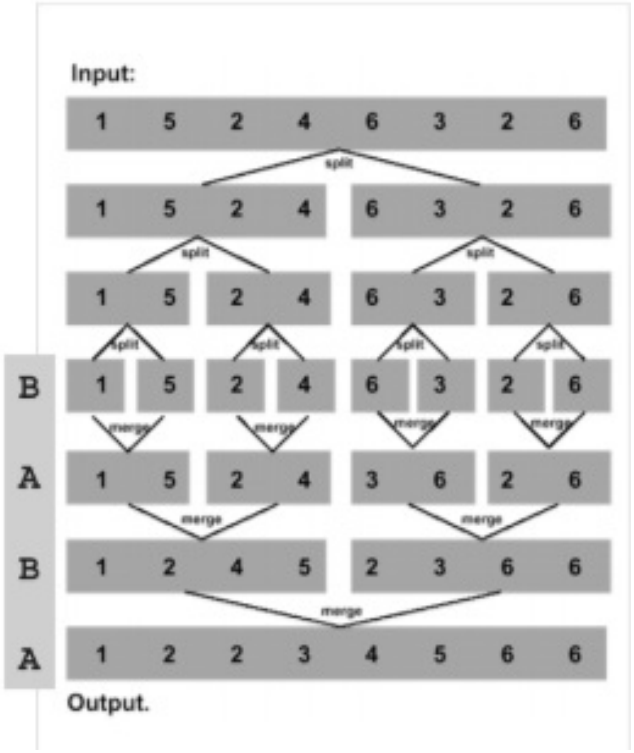
possibly in parallel

```
// Merges two sorted subsequences A[p..q] and A[q+1..r]
// into a single sorted subsequence A[p..r].
Merge(A, p, q, r)
    //Copy the subsequences into aux. memory with a sentinel
    L ← (A[p], …,A[q], ∞),  R ← (A[q+1], …,A[r], ∞)
   //Repeatedly take the smallest leftmost element of L and R
   i ← 1, j ← 1
   for k = p to r do
     if L[i] ≤ R[j] then A[k] ← L[i], i ← i+1
     else A[k] ← R[j], j ← j+1
```

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 log n) no melhor caso e no caso médio (com escolha aleatória de pivot)

Optimização da memória auxiliar

- Em vez de fazer cópias para memória auxiliar em cada `Merge`, cria-se inicialmente uma cópia (`B`) de `A`, e as operações de `Merge` vão alternadamente colocando os resultados em `A` e `B`
- O tempo gasto nestas cópias é reduzido de $\Theta(n \log)$ para $\Theta(n)$
- Tempo total pode ser reduzido até metade




```
// Sorts array A of size n.
Merge-Sort(A, n)
   Merge-Sort(A, copy(A), 1, n)
```

```
// Sorts array A between indices p and r, using aux. copy B.
Merge-Sort(A, B, p, r)
   if p < r then
      q ← ⌊(p + r) / 2⌋
      Merge-Sort(B, A, p, q) ‖ Merge-Sort(B, A, q + 1, r)
      Merge(A, B, p, q, r)
```

```
// Merges two sorted subsequences B[p..q] and B[q+1..r]
// into a single sorted subsequence A[p..r].
Merge(A, B, p, q, r)
  i ← p, j ← q + 1
  for k = p to r do
    if j > r ∨ i ≤ p ∧ B[i]≤B[j] then A[k] ← B[i], i ← i+1
    else A[k] ← B[j], j ← j+1
```

Processamento paralelo

- Com k processadores ou núcleos(cores), executando as chamadas recursivas em paralelo, pode-se ter um ganho de desempenho de até k vezes.
- Execução paralela é conseguida usando múltiplos threads.

Processamento paralelo em C++

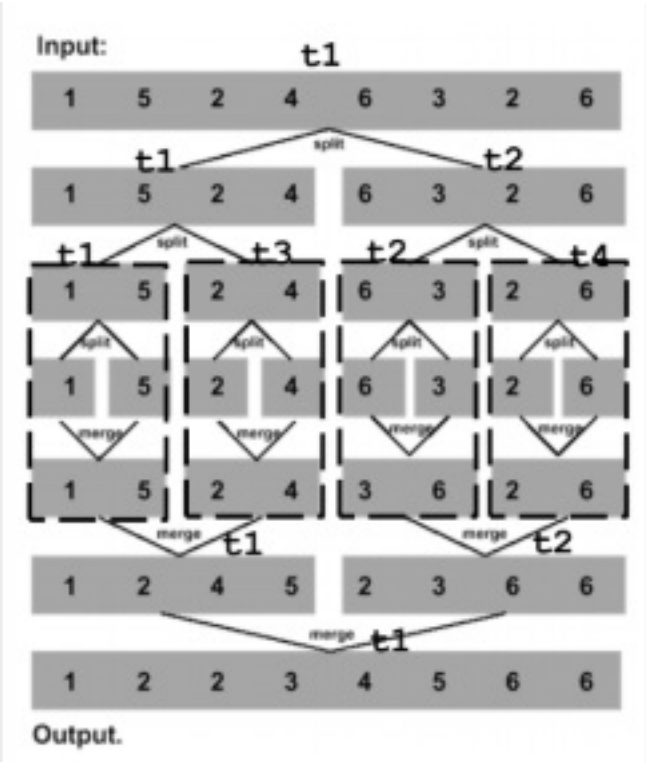
```
#include <thread>
template <typename T>
void MergeSort(T A[], T B[], int p, int r, int threads){
 if (p < r) {
  int q = (p + r) / 2;
  if (threads > 1) {
    std::thread t([=](){MergeSort(B, A, p, q, threads/2);});
    MergeSort(B, A, q + 1, r, threads /2);
    t.join();
  }
  else {
    MergeSort(B, A, p, q, 1);
    MergeSort(B, A, q + 1, r, 1)
  }
  Merge(A, B, p, q, r);
 }
}
```

Nº de *threads* a usar (inicialmente = nº de *cores*).

1) Lança 1ª chamada recursiva num novo *thread t* separado.

2) Executa 2ª chamada recursiva neste *thread*.

3) Espera que o outro *thread* termine.



Em suma
Programação Dinâmica

- Contexto: Problemas de solução recursiva.
- Objectivo: Minimizar tempo e espaço.
- Forma: Induzir uma progressão iterativa de transformações sucessivas de um espaço linear de soluções.

Algoritmos gananciosos:

- Contexto: Problemas de optimização (max. ou min.)

- Objectivo: Atingir a solução óptima, ou uma boa aproximação.
- Forma: tomar uma decisão óptima localmente, i.e., que maximiza o ganho (ou minimiza o custo) imediato.

Algoritmos de retrocesso:

- Contexto: problemas sem algoritmos eficientes (convergentes) para chegar à solução.

- Objectivo: Convergir para uma solução.
- Forma: tentativa-erro. Gerar estados possíveis e verificar todos até encontrar solução, retrocedendo sempre que se chegar a um “beco sem saída”.

Divisão e conquista:

- Contexto: Problemas passíveis de se conseguirem sub-dividir.
- Objectivo: melhorar eficiencia temporal.
- Forma: agregação linear da resolução de sub-problemas de dimensão semelhantes até chegar ao caso-base.

Funcionamento Correto

Muitos problemas podem ser especificados por um par:

- Entradas: Dados de entrada e restrições associadas (pré-condições).
- Saídas: Dados de saída e restrições associadas (pós-condições).

Correção parcial: se o algoritmo for executado com entradas que obedecem às pré-condições, então, se terminar, produz saídas que obedecem às pós-condições.

Correção total: se o algoritmo for executado com entradas que obedecem às pré-condições, então termina produzindo saídas que obedecem às pós-condições.

EXEMPLO:

- template <typename T>
  void sort(T a[ ], unsigned n)

- Pré-condições?
  - a != NULL
  - Operadores de comparação definidos para o tipo T

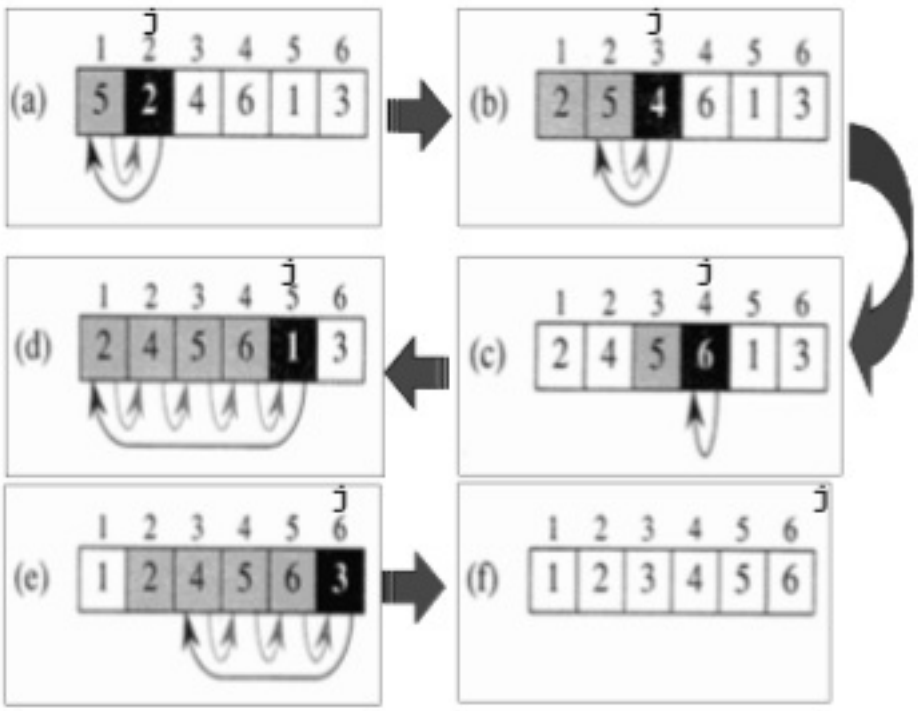
- Pós-condições?
  - Array “a” está ordenado, isto é a[0] <= a[1] <= … <= a[n-1]
  - Array final tem os mesmos elementos que o array inicial

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 - uma expressão booleana (nas variáveis do ciclo) ‘sempre verdadeira’ ao longo do ciclo, e mostrar que:
  - é verdadeira inicialmente, i.e., é implicada pela pré-condição;
  - é mantida em cada iteração, i.e., é verdadeira no fim de 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 uma variante do ciclo – uma função (nas variáveis do ciclo).

Insertion Sort:

Funcionamento:



Pseudo-código:

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Análise

- Invariante do ciclo principal [I(j)] ?

  A[1, …, j−1] contém os elementos originais, mas ordenados (j = 2, …, n+1)

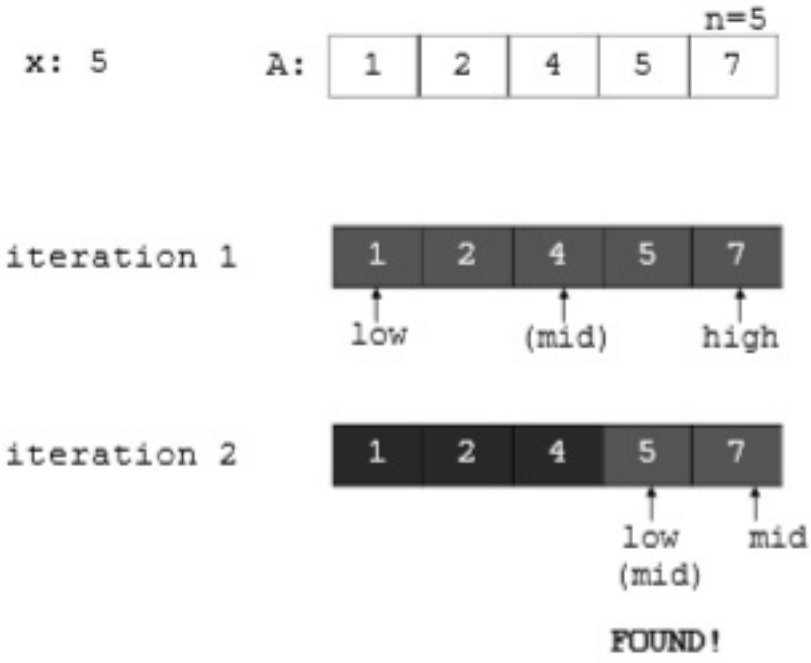
  - ✓ É válido inicialmente (j=2)
    - É óbvio que A[1,…,1] contém os elementos originais, mas ordenados
  - ✓ É mantido em cada iteração
    - Assume-se que o invariante se verifica no início da iteração
    - O alg. insere A[j] na posição certa em A[1, …, j] e incrementa j
    - Logo, no fim da iteração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

- Variante do ciclo principal [v(j)] ?

  `n + 1 − j  (j=2, …, 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

Binary Search:
Funcionamento



Pseudo-código:

```
BINARY-SEARCH(A, n, x)
    low ← 1
    high ← n
    while low ≤ high
        mid ← ⌊ (low + high) / 2⌋
        if x = A[mid] then
            return mid
        else if  x < A[mid] then
            high ← mid − 1
        else
            low ←  mid + 1
    return -1
```

Análise:

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

  `x` só pode existir na área de pesquisa, entre `low` e `high`

  - ✓ É válido 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`
  - ✓ No fim do ciclo, garante 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 [`v(low, high)`]?

  - ➢ Largura da área de pesquisa: `high - low + 1`

  - ✓ Inteiro, pois `low` e `high` são inteiros
  - ✓ Não negativo, pois no pior caso é `low = high + 1`
  - ✓ Estritamente decrescente, pois em cada iteração ou aumenta `low` ou diminui `high`

Grafos

Conceitos

Grafo F = (V,E)

V – vértices (ou nós)

E – arestas (ou arcos)

Se o par for ordenado, o grafo é dirigido (dígrafo)

Vertice w é adjacente a um vértice v sse (v,w) pertence a E **(neste sentido!)**

Grafo dirigido: Arestas têm sentido

Grafo não dirigido: Assume-se ambos os sentidos

Caminhos: Sequência de vértices tal que todas as transições sejam arestas válidas, o seu comprimento é igual ao número de arestas percorridas

Caminho Simples: todos os V distintos, exceto possivelmente o ultimo e primeiro

Ciclo: implica pelo menos uma aresta, se for não dirigido então as arestas terão de ser diferentes. **Anel:** Aresta (v,v) que pertence a E, comprimento 1 (raro)

DAG (Grafo acíclico dirigido): Para qualquer V v que se escolha não há ciclos.

Conetividade:

Grafo não dirigido é conexo sse houver um caminho a ligar qualquer par de vértices

Grafo dirigido é fortemente conexo se a partir de qualquer v chegas a todos os outros, fracamente conexo se todos os vértices tiverem conectados e não conexo caso tenha vértices "soltos"

Densidade: (Q é o "O" mas com limite superior e inferior)

Grafo denso - |E| = Q(V^2)

Grafo completo: existe uma aresta entre qualquer par de nós

Grafo esparso - |E| = Q(V)

Matriz de adjacências: (matriz com VxV elementos) – apropriado para grafos densos

a[u][v] = 1 sse (u,v) pertence a E, 0 otherwise

Lista de adjacências: típico para grafos esparsos

- para cada vértice, manter lista dos vértices adjacentes
- vetor de cabeças de lista, indexado pelos vértices
- espaço: O(|E| + |V|)
- pesquisa de adjacentes proporcional ao numero destes

Codificação: A nível prático é mais comum ter uma classe com vértices e arestas, onde os vértices guardam as arestas adjacentes e as arestas guardam o destino.

Esta abordagem permite que as arestas tenham um peso/nome associado.

Pesquisa e Ordenação

Pesquisa em profundidade (depth-first search)

- arestas são exploradas a partir do vértice mais recentemente descoberto ainda com arestas por visitar
- quando todas as arestas forem descobertas volta até ao ultimo vértice com arestas por explorar
- se ainda houverem vértices por descobrir, começar de novo a partir de um deles
- repetir o processo até todos os vértices serem visitados

Pseudo-código

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E}  (∀ v ∈ V)

DFS(G):
1.  for each v ∈ V
2.      visited(v) ← false
3.  for each v ∈ V
4.      if not visited(v)
5.          DFS-VISIT(G, v)

DFS-VISIT(G, v):
1.  visited(v) ← true
2.  pre-process(v)
3.  for each w ∈ Adj(v)
4.      if not visited(w)
5.          DFS-VISIT(G, w)
6.  post-process(v)
```

Numa aplicação concreta de DFS, há processamento a fazer num destes pontos

Pesquisa em largura (breadth-first search)

Descobre todos os vértices a que se pode chegar a partir da fonte, depois muda se a fonte etc.

```
BFS(G, s):
1.      for each v ∈ V do discovered(v) ← false
2.      Q ← ∅

3.      ENQUEUE(Q, s)
4.      discovered(s) ← true

5.      while Q ≠ ∅ do
6.          v ← DEQUEUE(Q)
7.          pre-process(v)
8.          for each w ∈ Adj(v) do
9.              if not discovered(w) then
10.                 ENQUEUE(Q, w)
11.                 discovered(w) ← true
12.         post-process(v)
```

Numa aplicação concreta de BFS, há processamento a fazer num destes pontos

NOTAS: Caminho na árvore BFS é sempre o mais curto. Caso a fila seja mudada para uma pilha, obtém-se um algoritmo iterativo de visita em profundidade.

Ordenação topológica (impossível se o grafo for cíclico)

Objetivo: ordenar vértices de um DAG tal que se existir uma aresta (v,w) então v aparece antes de w. (Pode haver mais do que uma solução)

Método:

- Descobrir um vértice sem arestas de chegada (indegree = 0)
- Imprimir o vértice
- Eliminá-lo e às arestas que saem dele
- Repetir o processo no grafo restante

Refinamento: atualizar variável "indegree" com o número de arestas a entrar no vértice. Estrutura de dados com todos os vértices com indegree = 0.

```
TOP-SORT( in G=(V,E),  out  T):
1.  for each v∈V do indegree(v) ← 0
2.  for each v∈V do for each w∈adj(v) do indegree(w) ← indegree(w) + 1
3.  C ← { } // Pode ser uma fila (Queue), pilha (Stack), etc.
4.  for each v∈V do if indegree(v) = 0 then C ← C ∪ {v}
5.  T ← [ ] // Pode ser uma lista  (LinkedList)
6.  while C ≠ {} do
7.      v ← remove-one(C)
8.      T ← T concatenado-com [v]
9.      for each w ∈ adj(v) do
10.         indegree(w) ← indegree(w) - 1
11.         if indegree(w) = 0 then C ← C ∪ {w}
12. if |T| ≠ |V| then Fail("O grafo tem ciclos")
```

Análise:

Se as inserções e eliminações em C forem feitas em tempo constante (fila FIFO) algoritmo é executado em tempo O(|V|+|E|).

- ciclo 9-11 é executado no máximo 1 vez por aresta
- operações em 4,7 e 11 são executadas no máximo uma vez por vértice
- inicialização leva um tempo proporcional ao tamanho do grafo

A ordenação topológica de um DAG dá-nos a ordem/sequência dos eventos do DAG.

Caminho mais curto

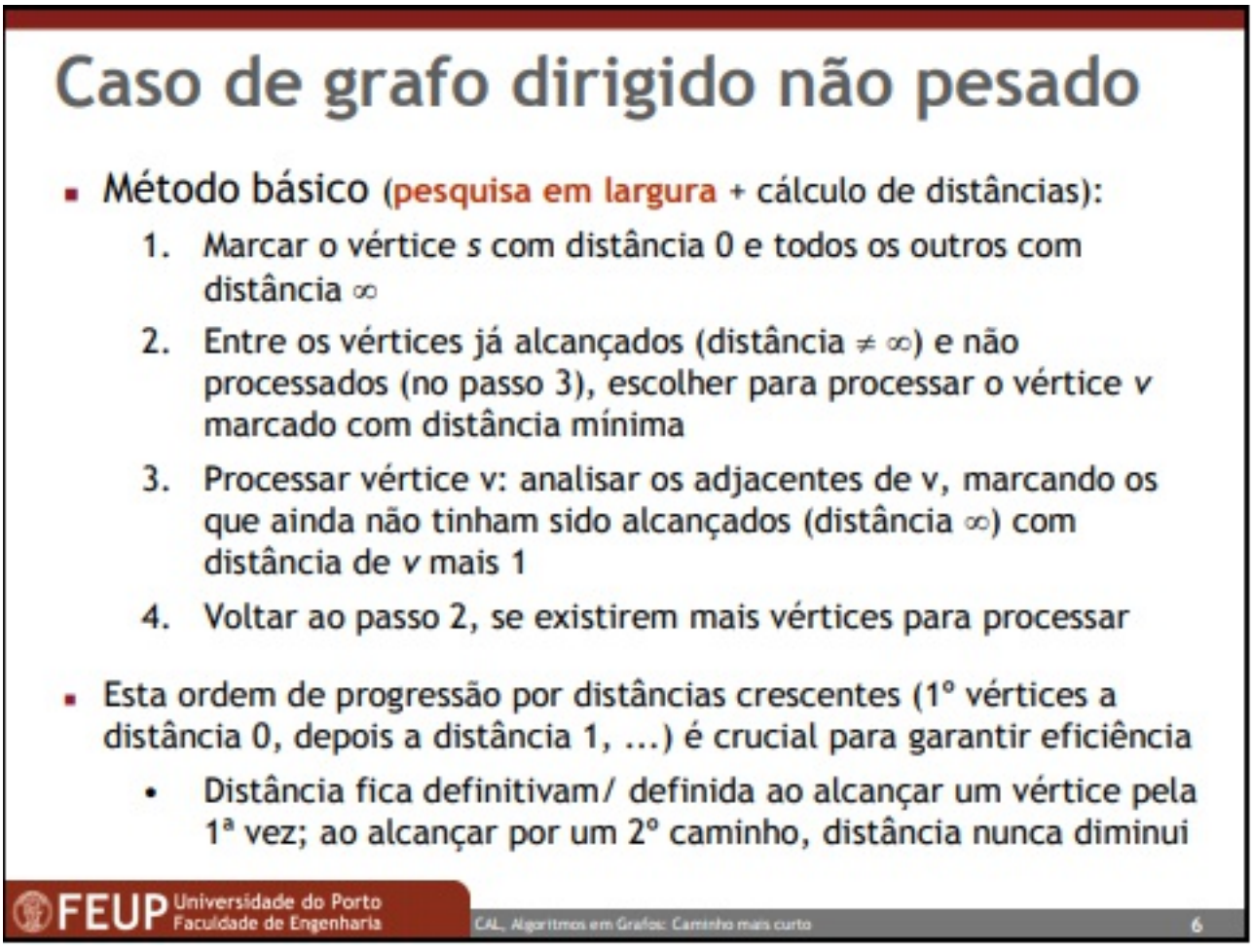
Caminhos mais curtos de um vértice para todos os outros:

- **Grafos dirigidos não pesados:** baseado em pesquisa em largura, O(|V|+|E|)
- **Grafos dirigidos pesados:** Dijkstra (algoritmo ganancioso) O((|V|+|E|) * log |V|)
- **Grafos dirigidos com arestas de peso negativo:** Bellman-Ford, programação dinâmica O(|E||V|)
- **Grafos dirigidos acíclicos:** baseados em ordenação topológica, O(|V| + |E|)

Variantes:

- **Caso base:** grafo dirigido, fortemente conexo, pesos >= 0
- **Grafo não dirigido:** = grafo dirigido, com pares de arestas simétricas
- **Grafo não conexo:** pode não haver caminho para alguns vértices (dist = inf)
- **Grafo não pesado:** arestas peso 1, existe algoritmo mais eficiente do que para o caso base
- **Arestas com pesos negativos:** algoritmo menos eficiente do que caso base, ciclos com peso negativo tornam o caminho mais curto indefinido.

Grafo dirigido não pesado



Estrutura de dados: fila FIFO, onde são inseridos novos vértices alcançados e é extraído próximo vértice a processar. Cada vértice contem dist (distância ao vértice inicial) e path(vértice anterior no caminho mais curto).

Pseudo-código

```
SHORTEST-PATH-UNWEIGHTED(G=(V,E), s):
1.    for each v ∈ V do
2.        dist(v) ← ∞
3.        path(v) ← nil
4.    dist(s) ← 0
5.    Q ← ∅
6.    ENQUEUE(Q, s)
7.    while Q ≠ ∅ do
8.        v ← DEQUEUE(Q)
9.        for each w ∈ Adj(v) do
10.           if dist(w) = ∞ then
11.               ENQUEUE(Q, w)
12.               dist(w) ← dist(v) + 1
13.               path(w) ← v
```

Tempo de execução:

O(|E| + |V|)

Espaço auxiliar:

O(|V|)

Grafo dirigido pesado

Caso de grafo dirigido pesado (pesos≥0)

- Método básico semelhante ao caso de grafo não pesado
- Distância obtém-se somando pesos das arestas em vez de 1
- Próx. vértice a processar continua a ser o de distância mínima
  - Mas já não é necessariamente o mais antigo ⇒ Obriga a usar **fila de prioridades** (com mínimo à cabeça) em vez duma fila simples
  - Mas pode ser necessário rever em baixa a distância de um vértice alcançado e ainda não processado (vértice na fila) ⇒ Obriga a usar **fila de prioridades alteráveis**
  - Nota: A ordem é crucial para garantir que a distância ao vértice de partida dos vértices já processados não é mais alterada, assumindo que não há pesos negativos (ver análise adiante)
- É um algoritmo **ganancioso**: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância)

Algoritmo de Dijkstra (adaptado)

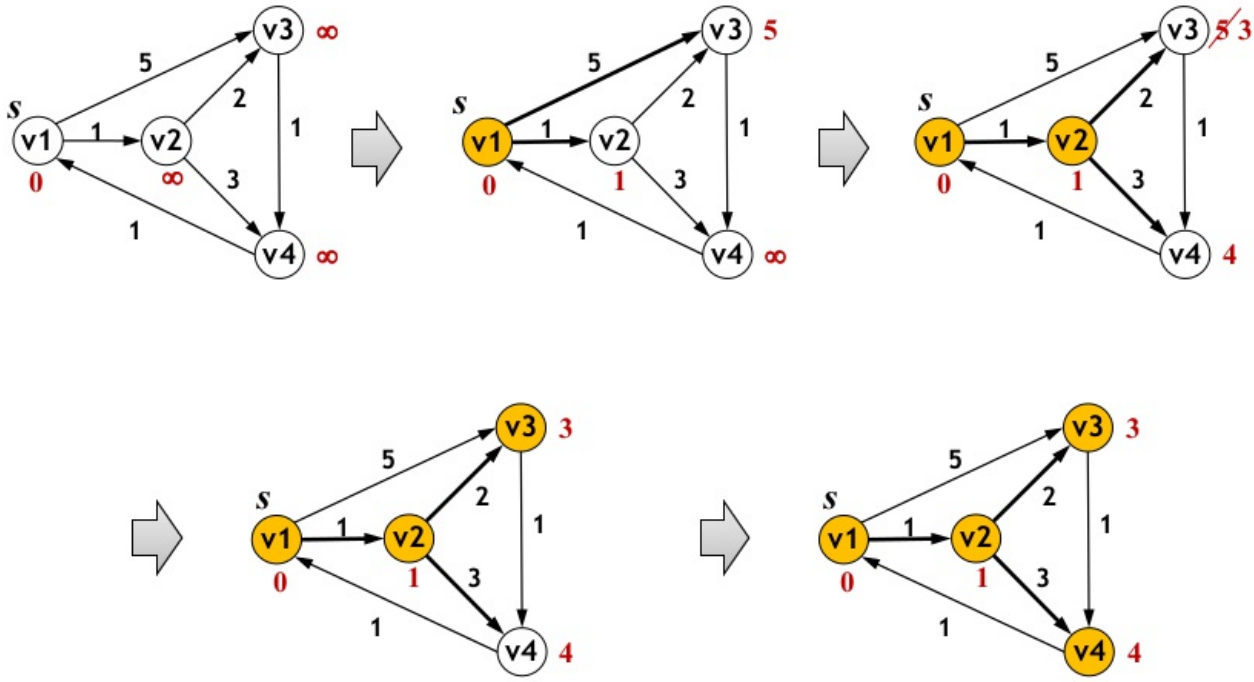
```
DIJKSTRA(G, s):   // G=(V,E), s ∈ V
1.    for each v ∈ V do
2.       dist(v) ← ∞
3.       path(v) ← nil
4.    dist(s) ← 0
5.    Q ← ∅   // min-priority queue
6.    INSERT(Q, (s, 0)) // inserts s with key 0
7.    while Q ≠ ∅ do
8.       v ← EXTRACT-MIN(Q) // greedy
9.       for each w ∈ Adj(v) do
10.         if dist(w) > dist(v) + weight(v,w) then
11.            dist(w) ← dist(v)+ weight(v,w)
12.            path(w) ← v
13.            if w ∉ Q then  // old dist(w) was ∞
14.               INSERT(Q, (w, dist(w)))
15.            else
16.               DECREASE-KEY(Q, (w, dist(w)))
```

Tempo de execução:
O((|V|+|E|) * log |V|)

Efetuar algoritmo 'a olho'
Em grafos não dirigidos, passar para dirigido, duplicando as arestas e adicionando sentidos.
Marcar vértice de partida com distância 0, e restantes com infinito.
Manter uma fila (FIFO), colocando o vértice inicial.
Começar por processar esse vértice, e atualizar a distância dos vértices adjacentes com o valor do peso da aresta respetiva.
A seguir, colocar na fila (pelo lado esquerdo) os vértices adjacentes, por ordem da distância, e retirar o vértice atual, processando sempre o vértice no lado direito da fila.
Ver arestas adjacentes a esse vértice, e verificar se as distâncias aos vértices de destino podem ser diminuídas, tendo em conta o caminho desde a origem: nesse caso, atualizar o valor delas.
Parar quando forem processados todos os vértices.

As distâncias finais representam o peso mínimo desde o vértice inicial até cada um dos outros vértices.

Evolução da marcação do grafo



Eficiência de Decrease-Key:

Fila de prioridades -> heap com mínimo à cabeça e tamanho = n (max. |V|).

Método naive: O(n)

Procurar no array objeto cuja chave se vai alterar (O(n)).

Subir/Descer o objeto na árvore até restabelecer o invariante da árvore (cada nó menor ou igual aos filho) (O(log n)).

Método melhorado: O(log n)

Cada objeto colocado na heap guarda o seu índice nele. Não é necessário o passo 1, logo o tempo total é O(log n). Quando se insere/move um objeto, o seu índice tem que ser atualizado).

Método otimizado: O(1)

Com Fibonacci Heaps consegue-se fazer Decrease-Key em tempo amortizado (O(1)).

Eficiência do algoritmo de Dijkstra

Tempo de execução é O(|V|+|E|+|V|*log*|V|+|*E*|log|V|)
ou
O((|V|+|E|)*log|V|).

- O(|V|*log|V|) – extração e inserção na fila de prioridades
    - Nº de extrações e inserções = |V|

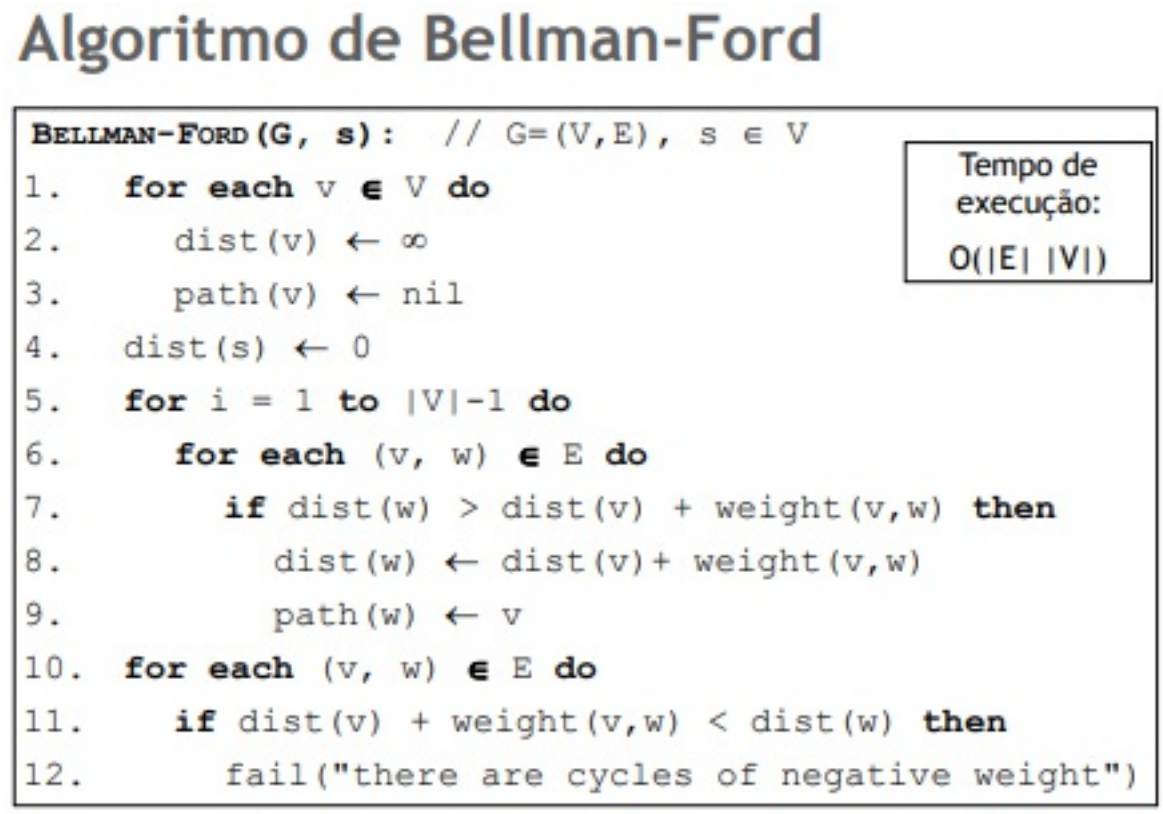
    - Cada extração/inserção pode ser feita em O(log n), tamanho máximo da fila é |V|

- O(|E|*log|V|) – Decrese-Key
    - Feita no máximo uma vez por aresta (|E|)
    - Cada extração/inserção pode ser feita em O(log n), tamanho máximo da fila é |V|

Pode ser melhorado para O(|V|*log|V|) com Fibonacci Heaps.

Arestas com peso negativo

Neste caso pode ser necessário processar cada vértice + que 1 vez. Se existirem ciclos com peso negativo, o problema não tem solução. Resolúvel em tempo O(|V||E|) pelo algoritmo de Bellman-Ford.



Grafos acíclicos

Simplificação do algoritmo 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 atualização das distâncias e caminhos numa só passagem
- Tempo de execução é o da ordenação topológica: O(|V|+|E|)

Caminho mais curto numa rede viária

Caminho mais curto entre dois vértices
Não se conhece algoritmo mais eficiente para resolver este problema do que a resolver o mais geral (de um vértice para todos os outros). Por isso, acha-se o caminho mais curto da origem para todos os outros, e seleciona-se depois o caminho da origem para o destino. Otimização: parar assim que chega a vez de processar o vértice de destino (ajuda para distâncias curtas, mas não para distâncias longas).

Método base (rede viária)

Rede viária -> Grafo dirigido:

- Vértices -> Interseções
- Arestas -> Vias (possivelmente sentido único)
- Pesos -> Distâncias, tempos, custos, etc.

Algoritmo de Dijkstra é a base para encontrar o caminho mais curto entre dois pontos s e t, parando-se a pesquisa quando o próximo nó a processar é o nó t. Uma vez que o algoritmo processa os vértices por distâncias crescentes ao vértice de partida, é inspecionado um círculo em torno de s de raio igual à distância entre s e t.

Otimizações

Os mapas de estradas são enormes, algoritmo de Dijkstra pode demorar muitos segundos ou minutos a encontrar o caminho mais curto em trajetos de longa distância. Otimizações que não exigem pré-processamento conseguem ganhos de desempenhos modestos (até 10x). Com pré-processamento, conseguem-se ganhos da ordem de 103 ou mesmo 106, reduzindo tempo de pesquisa para ms ou μs.

Pesquisa bidirecional

Executar o algoritmo de Dijkstra no sentido de s para t e em sentido inverso de t para s (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 d 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 d. Retornar a distância d e o caminho correspondente.

Área processada – metade da pesquisa unilateral. **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 (= algoritmo Dijkstra)
- pi~vt~ – estimativa por baixo da dist. min. de v a t (função potencial)

Em geral, não garante o ótimo. Em certos casos, garante o ótimo, por exemplo:

Pesos das arestas são distâncias em km e pi~vt~ é a distância em linha reta de v a t. Equivale a aplicar o algoritmo de Dijkstra com pesos de arestas modificados **w~uv~ = w~uv~ - pi~ut~ + pi~vt~**, somando-se no final pi~st~ à distância mínima obtida de s para t.

Pode ser combinado com pesquisa bidirecional. Speedup na prática é moderado.

Redes hierárquicas (highway networks)

Pré-processamento decompõe a rede em vários níveis hierárquicos. Analogia com mapa de estradas nacional e mapas de ruas locais. Uma aresta (u,v) é classificada automaticamente como ***highway edge*** se existe pelo menos um par de nós s e t da rede tal que:

- O caminho mais curto de s a t passa em (u,v);
- U está a mais de H nós de distância de s;
- V está a mais de H nós de distâncias de t.

H é um parâmetro configurável.

Pode ser aplicável a mais níveis (local, *highway*, *super-highway*)

Pesquisa é bidirecional e usa rede mais densa próximo de s e t e mais esparsa longe de s e t. A pesquisa realiza-se em tempo da ordem de 1ms. Exige pouco espaço adicional: um campo por aresta.

Nós de trânsito (*transit-node routing*)

Pré-processamento determina:

- **Nós de trânsito** – nós tal que o caminho mais curto entre quaisquer 2 nós da rede que não estão "muito perto" entre si passa pelo menos um dos nós de trânsito. Armazenam-se numa tabela as distâncias entre todos os pares de nós de trânsito
- **Nós de acesso** – para cada nó da rede, são os nós de trânsito mais próximos. Tipicamente há 10 nós de acesso por nó da rede. Armazenam-se numa tabela, para cada nó de rede, os nós de acesso e distâncias. Na verdade, determinam-se dois conjuntos de nós de acesso: nós de saída (*forward*, AF) e nós de entrada (*backward*, AB)

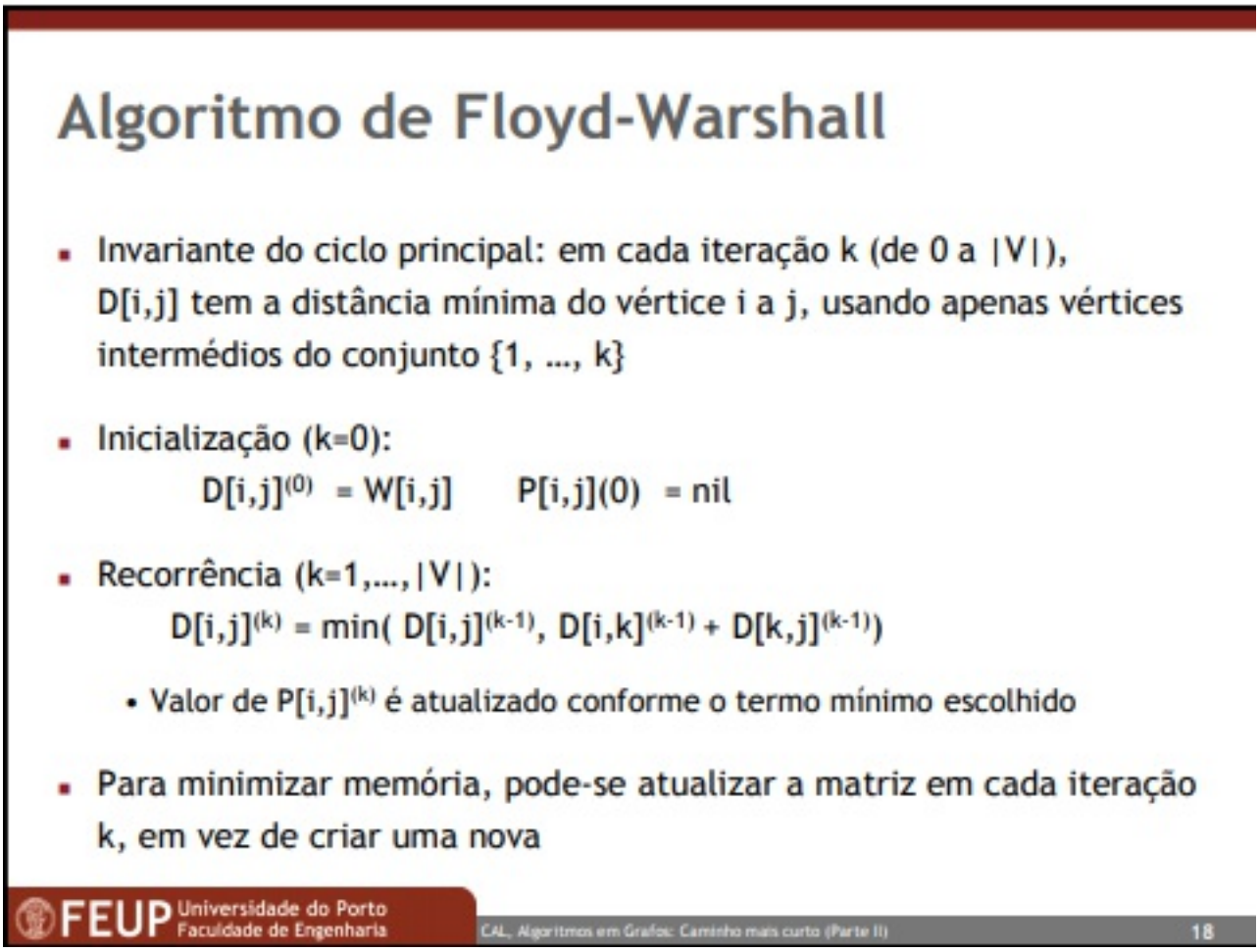
A pesquisa do caminho mais curto entre dois pontos afastados é reduzida a poucos *table lookups*, e realizada em tempo da ordem de 10 microsegundos (mas exige espaço de armazenamento adicional significativo)

- Obter os nós de acesso dos nós de partida (s) e chegada (t)
- Para cade para (nó de acesso inicial (u), nó de acesso final (v)), obter distância de s a t em 3 *table lookups*

Caminho mais curto entre todos os pares de vértices

Execução repetida do algoritmo de Dijkstra (ganancioso): O(|V| (|V| + |E|) log|V|), bom se o grafo for esparso (|E| - |V|).

Algoritmo de Floyd-Warshall, programação dinâmica: O(|V|3). Melhor que o anterior se o grafo for denso (|E| - |V|2). Mesmo em grafos pouco densos pode ser melhor porque o código é mais simples. Baseia-se em matriz de adjacências W[i,j] com pesos (infinito quando não há aresta; 0 quando i = j). Calcula matriz de distâncias mínimas D[i,j] e matriz P[i,j] de predecessor no caminho mais curto de i para j.



Árvore de Expansão Mínima

Árvore que liga todos os vértices do grafo usando arestas com um custo total mínimo.
Estudamos o caso do grafo não dirigido.

- Deve ser conexo.
- Árvore é grafo conexo acíclico.
- Número de arestas: |V| - 1

Algoritmo de Prim

Expandir a árvore por adição sucessiva de arestas e respectivos vértices.

Critério de seleção
Eescolher a aresta (u,v) de menor custo tal que u já pertence à árvore e v não (ganancioso)

Início
Um vértice qualquer

Idêntico ao algoritmo de Djikstra:
informação para cada vértice

- `dist(v)` é o custo mínimo das arestas que ligam a um vértice já na árvore
- `path(v)`é o último vértice a alterar `dist(v)`
- `known(v)` indica se o vértice já foi processado (i.e., já pertence à árvore)
- diferença na regra de actualização: após a selecção do vértice v, para cada w não

processado, adjacente a v, `dist(w) = min{ dist(w), cost(v,w) }`

Tempo de execução
O(|V|^2) sem fila de prioridade
O(|E| 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)

Se dois vértices pertencem à mesma árvore/conjunto, mais uma aresta entre eles provoca um ciclo (2 Buscas)

método:

- manter uma floresta, inicialmente com um vértice em cada árvore (há |V|)
- adicionar uma aresta (a próxima com menor peso) é fundir duas árvores
- quando o algoritmo termina há só uma árvore (de expansão mínima)

tempo no pior caso: O(|E| log|E|) ou O(|E| log|V|).

```
void kruskal() {
    int edgesAccepted = 0;

    PriorityQueue<Edge> h = readGraphIntoHeapArray();
    h.buildHeap();
    DisjSet<Vertex> s = new DisjSet(NUM_VERTICES);

    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);
        }
    }
}
```

Conetividade

Um grafo não dirigido é conexo sse uma pesquisa em profundidade a começar em qualquer nó visita todos os nós.

Pesquisa em profundidade

```
void dfs()
{
    for(Vertex v: vertexSet)
        v.visited = false;
    for(Vertex v: vertexSet)
        if(!visited)
            dfs(v);
            //v passa a ser raiz de arvore dfs
}

void dfs(Vertex v)
{
    v.visited = true;
    //fazer qualquer coisa c/ v aqui
    for(Edge e: v.adj)
        if(!e.dest.visited)
            dfs(e.dest)
    //ou aqui
}
```

Biconectividade e Pontos de Articulação

- Grafo conexo não dirigido é biconexo se não existe nenhum vértice cuja remoção torne o resto do grafo desconexo
- Pontos de articulação: vértices que, quando removidos, tornam o grafo desconexo: pelo menos um dos vértices restantes é inatingível a partir dos restantes.

Deteção de pontos de articulação

```
// Procura Pontos de Articulação usando dfs
// Contador global e inicializado a 1
void findArt( Vertex 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);
}
```

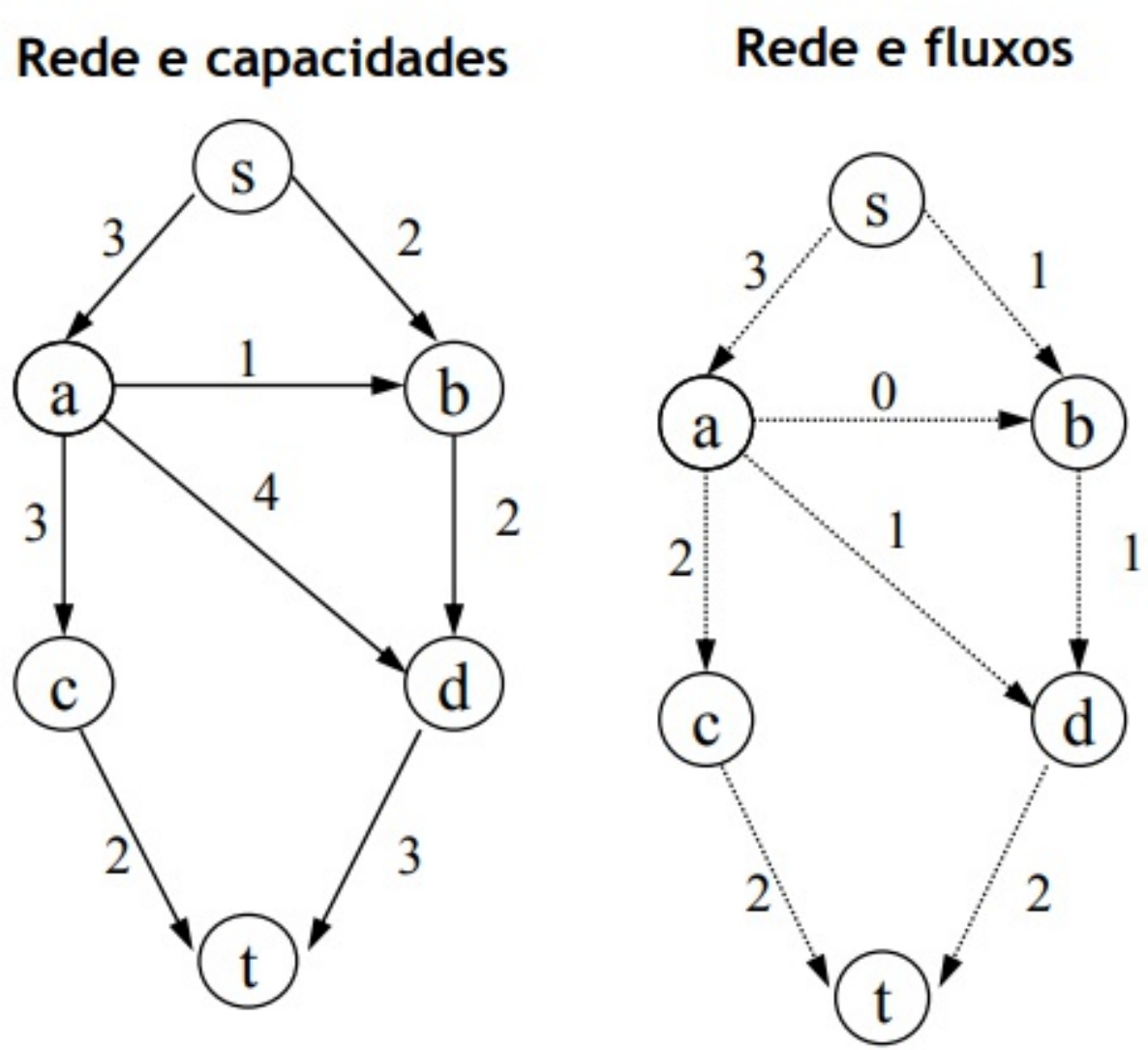
Em Grafos dirigidos

Procurar componentes fortemente conexos

- 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, i.e., a partir de um qualquer dos nós pode chegar-se a todos os outros

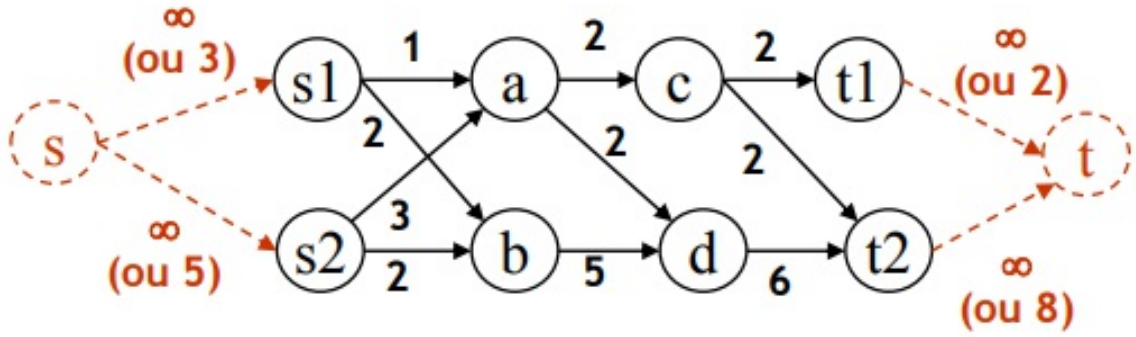
Fluxo máximo em redes de transportes

Redes de transporte



- S: fonte (produtor)
- T: poço (consumidor)
- Fluxo não pode ultrapassar a capacidade da aresta
- Soma do fluxo à entrada de um vértice intermédio tem que ser igual à soma do fluxo à saída

Múltiplas fontes ou poços podem ser reduzidas ao caso base



Problema: Encontrar fluxo máximo

Exemplos de aplicação:

- Rede de abastecimento de líquido ponto a ponto
- Tráfego entre dois pontos
- Emparelhamento máximo em grafos bipartidos (maximum bipartite matching)

Soluções

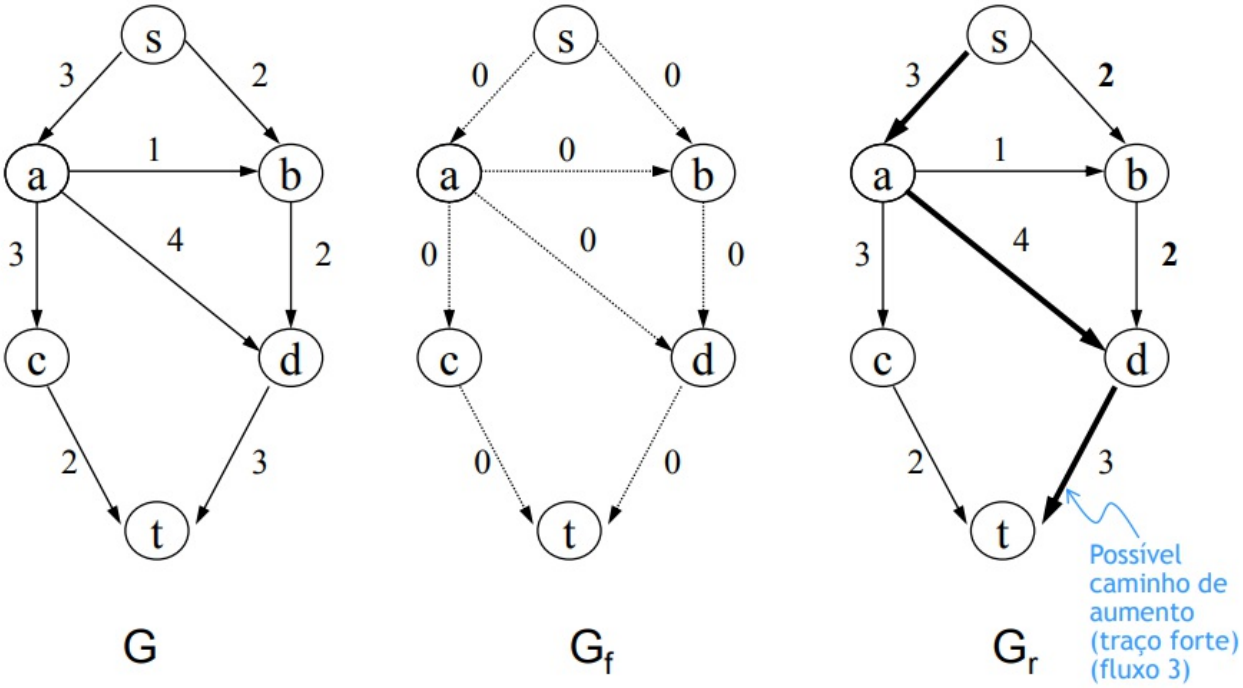
Algoritmo de Ford-Fulkerson

Três grafos:

G – Grafo base de capacidades

Gf – Grafo de fluxos (inicializado a 0, vai conter solução final)

Gr – Grafo de resíduos (inicializado igual a G)



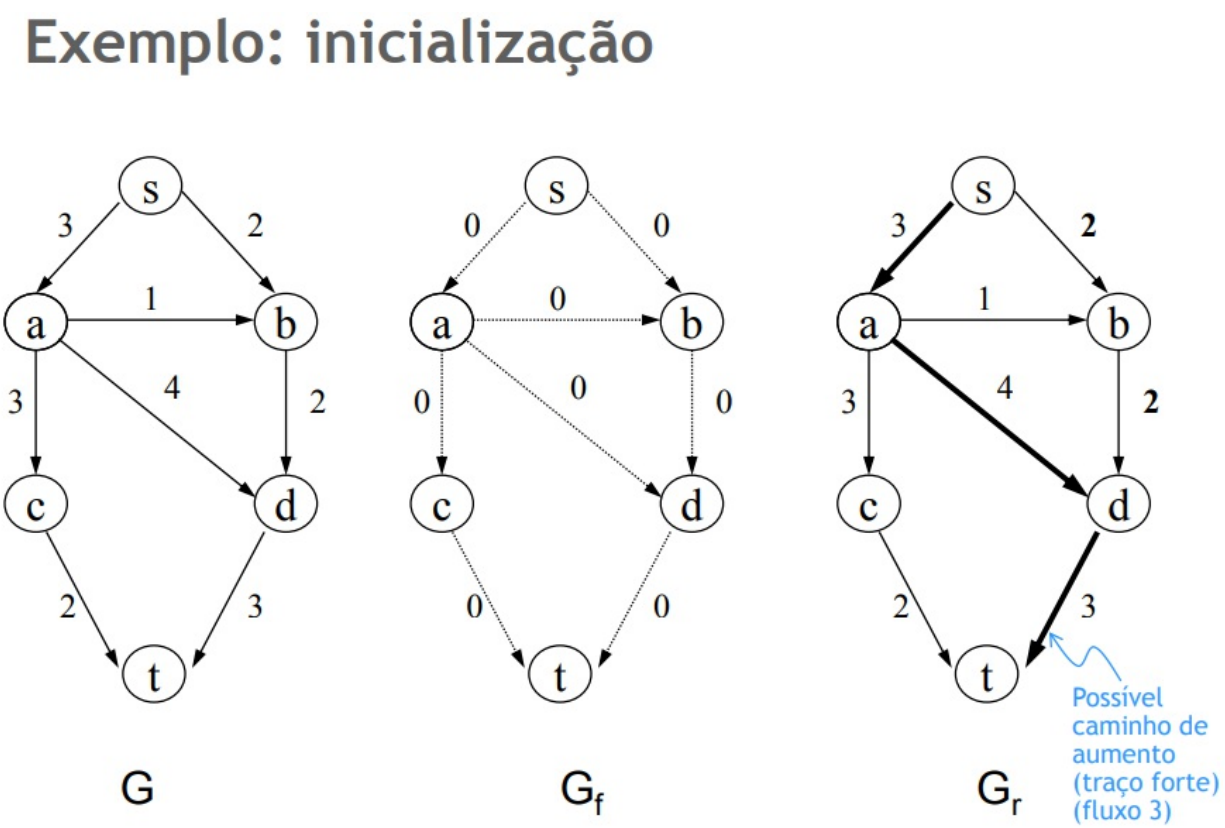
Algoritmo

Caminho de aumento é caminho entre s e t onde capacidade mínima das arestas é maior que 0.

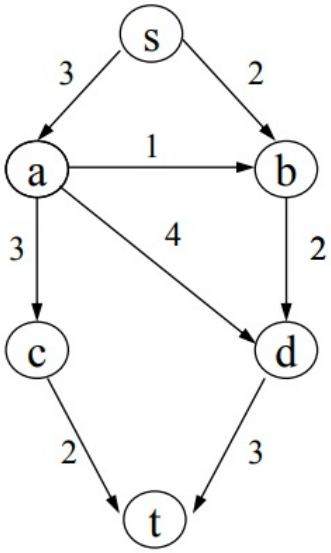
Enquanto existir um caminho entre s e t em Gr:

- Selecionar um caminho qq entre s e t em Gr (caminho de aumento)
- Determinar o valor mínimo de fluxo nesse caminho - seja f(a)
- Adicionar f(a) ao fluxo em Gf nos arcos correspondentes Se o sentido das arestas correspondentes for diferente, subtrair o valor em Gf com o de Gr.
- Atualizar Gr
    - Na direção do caminho tomado: reduzir o peso das arestas em f(a). Se o peso ficar a 0, a aresta desaparece.
    - Na direção oposta, aumentar o peso da aresta em f(a). Se a aresta não existir, cria-se.

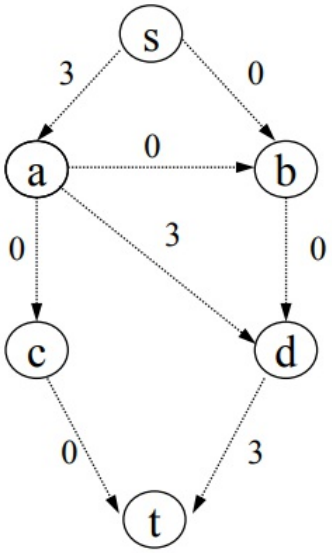
No final, o fluxo máximo é a soma do fluxo das arestas que chegam ao t em Gf.



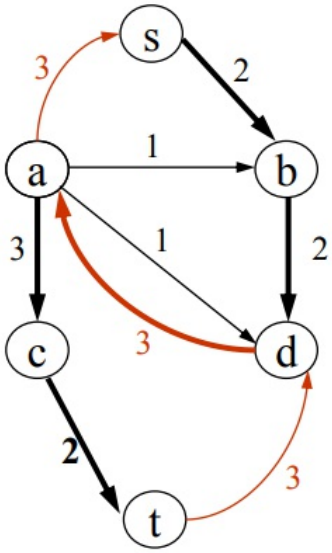
Exemplo: 1ª iteração



G



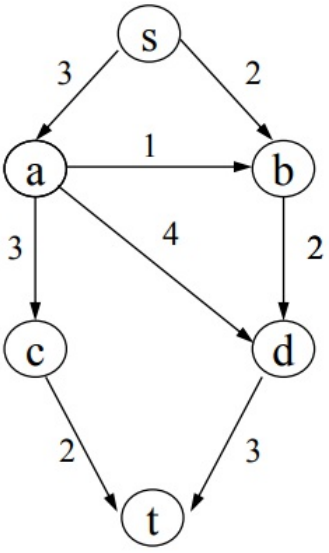
G_f



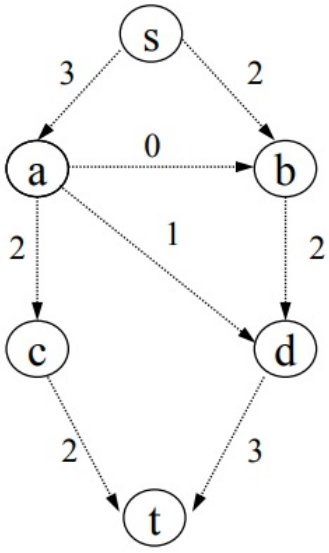
G_r

Se não tivesse as arestas em sentido inverso, parava aqui com solução não óptima! (Caminho de aumento com fluxo 2)

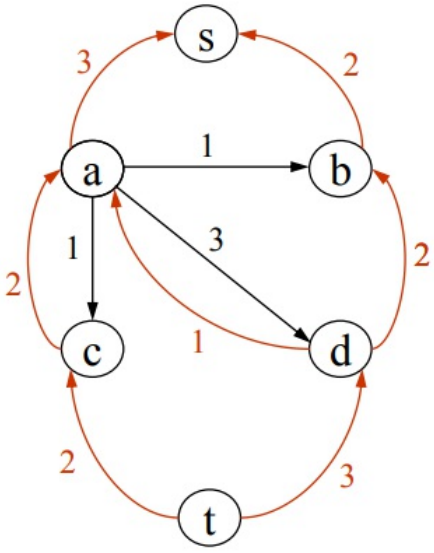
Exemplo: 2ª iteração



G



G_f



G_r

FIM
(valor do fluxo máximo = 5)

Análise do algoritmo:

- Cada iteração - O(|E|)

- Tempo de execução total – O(F * |E|) – mau

(|E| - número de arestas; F – fluxo máximo)

Algoritmo de Edmonds-Karp

- Melhoramento do algoritmo de Fords-Fulckerson
- Em cada iteração escolhe-se o caminho de aumento de comprimento mínimo (pesquisa em largura – O(|E|)
- Nº máximo de aumentos - |E| . |V|
- Tempo de execução – O(|V| |E|^2)

Implementação

Para efetuar os cálculos num único grafo, guardam-se:

• Em cada aresta:

- orig: apontador para vértice de origem
- dest: apontador para vértice de destino
- capacity: capacidade da aresta
- flow: fluxo na aresta

• Em cada vértice:

- outgoing: vetor de apontadores para arestas que saem do vértice
- incoming: vetor de apontadores para arestas dirigidas ao vértice
- visited: campo booleano usado na procura do caminho de aumento
- path: apontador para aresta anterior no caminho de aumento

• No grafo:

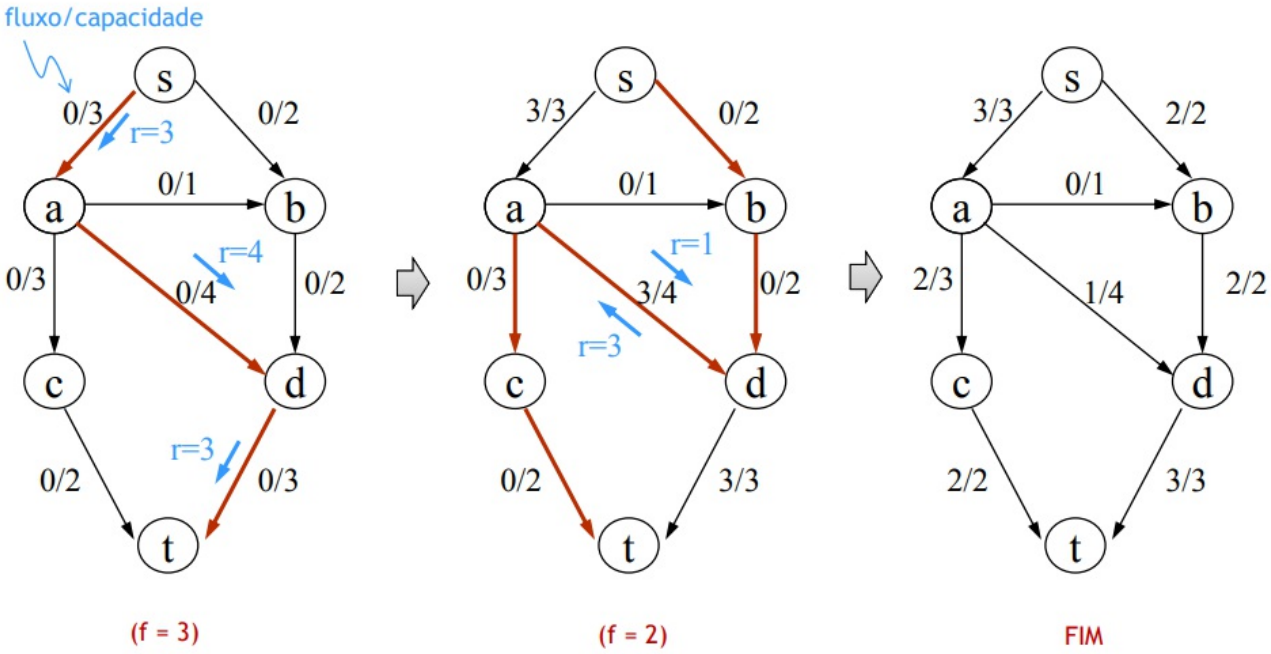
- vertexSet: vetor de apontadores para vértices

O grafo de resíduos é determinado “on the fly”

• Arestas percorridas no sentido normal têm resíduo = capacidade - fluxo

• Arestas percorridas no sentido inverso têm resíduo = fluxo

Exemplo




```
FordFulkerson(g, s, t):
1. ResetFlows(g)
2. tot ← 0
3. while FindAugmentationPath(g, s, t) do
4.     f ← FindMinResidualAlongPath(g, s, t)
5.     AugmentFlowAlongPath(g, s, t, f)
6.     tot ← tot + f
7. return tot
```

```
ResetFlows(g):
1. for v ∈ vertexSet(g) do
2.     flow(v) ← 0
```

```
FindAugmentationPath(g,s,t): // Edmonds-Karp (breadth-first)
1.   for v ∈ vertexSet(g) do
2.      visited(v) ← false
3.   visited(s) ← true
4.   Q ← ∅
5.   ENQUEUE(Q, s)
6.   while Q ≠ ∅ ∧ ¬ visited(t) do
7.      v ← DEQUEUE(Q)
8.      for e ∈ outgoing(v) do // direct residual edges
9.         TestAndVisit(Q, e, dest(e), capacity(e) - flow(e))
10.     for e ∈ incoming(v) do // reverse residual edges
11.        TestAndVisit(Q, e, orig(e), flow(e))
12.  return visited(t)
```

```
TestAndVisit(Q, e, w, residual):
1.  if ¬ visited(w) ∧ residual > 0 then
2.     visited(w) ← true
3.     path(w) ← e // previous edge in shortest path
4.     ENQUEUE(Q, w)
```

```
FindMinResidualAlongPath(g, s, t):
1.   f ← ∞
2.   v ← t
3.   while v ≠ s do
4.      e ← path(v)
5.      if dest(e) = v then // direct residual edge
6.         f ← min(f, capacity(e) - flow(e))
7.         v ← orig(e)
8.      else // reverse residual edge
9.         f ← min(f, flow(e))
10.        v ← dest(e)
11.  return f
```

```
AugmentFlowAlongPath(g, s, t, f):
1.   v ← t
2.   while v ≠ s do
3.      e ← path(v)
4.      if dest(e) = v then // direct residual edge
5.         flow(e) ← flow(e) + f
6.         v ← orig(e)
7.      else // reverse residual edge
8.         flow(e) ← flow(e) - f
9.         v ← dest(e)
```

Fluxo de Custo Mínimo

Problema: transportar uma certa quantidade F de fluxo da fonte (s) para o poço (t), com um custo total mínimo

- Arestas têm custo associado para além da capacidade
- Podem existir arestas de custo negativo

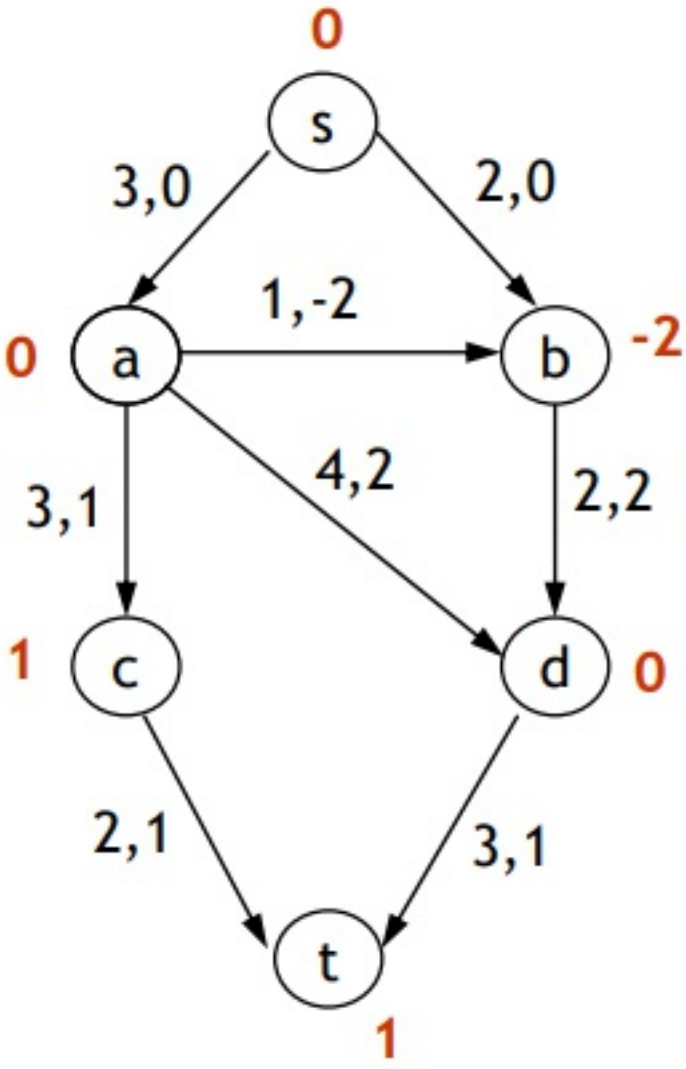
Soluções:

Algoritmo ganancioso (Ford-Fulkerson) – em cada iteração, escolhe-se o caminho de aumento de custo mínimo (funciona apenas se as redes não tiverem ciclos de custo negativo)

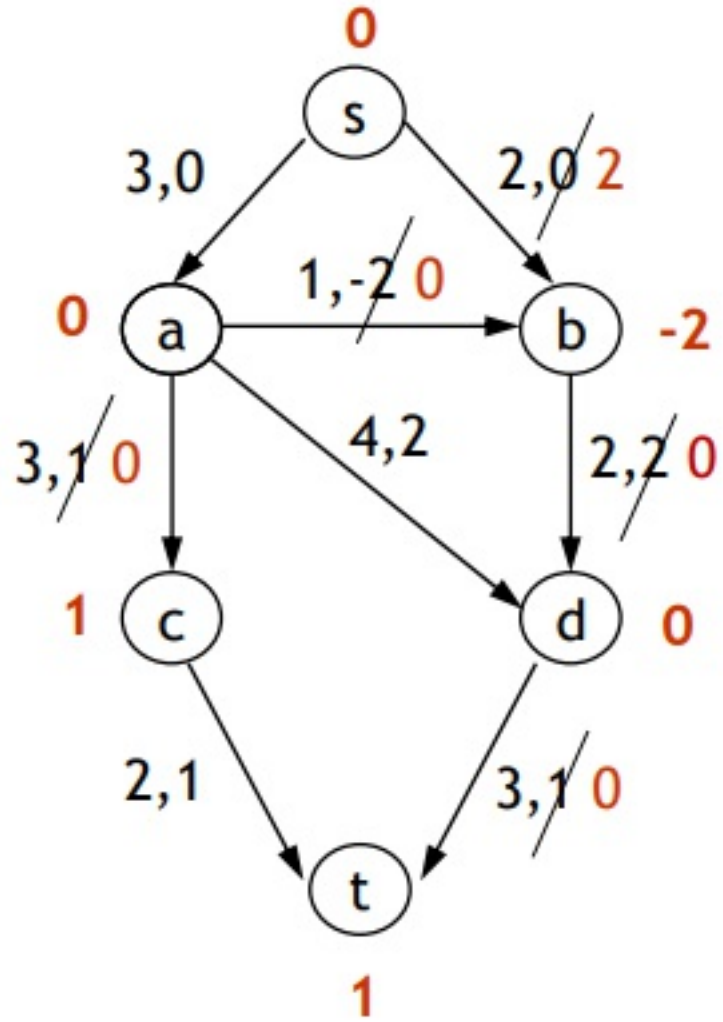
- Obriga o uso de algoritmos menos eficientes (Bellman-Ford O(|V| |E|)) na procura do caminho de custo mínimo
- Solução: Converte-se o grafo num equivalente, mas sem custos negativos

Conversão do grafo:

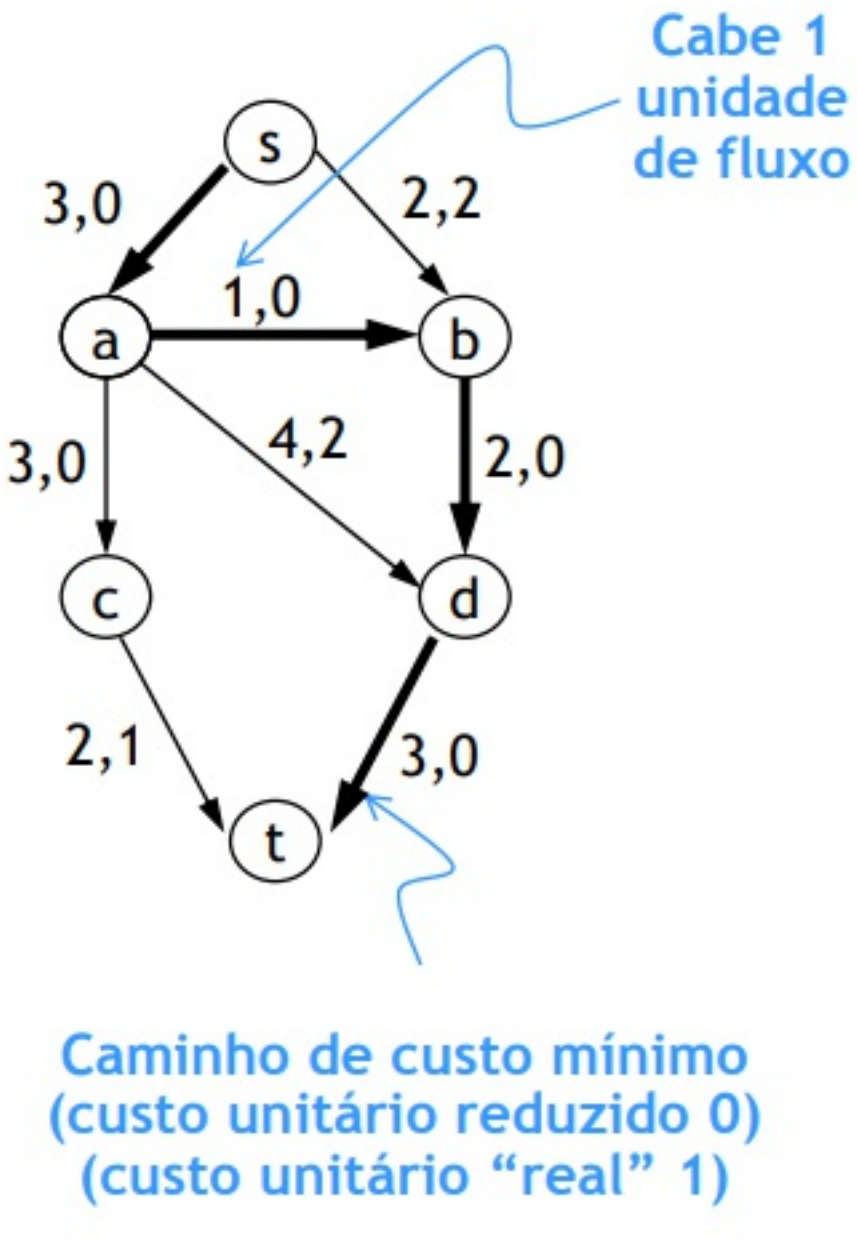
1 – Determinar o custo mínimo de S a todos os vértices (d(v))



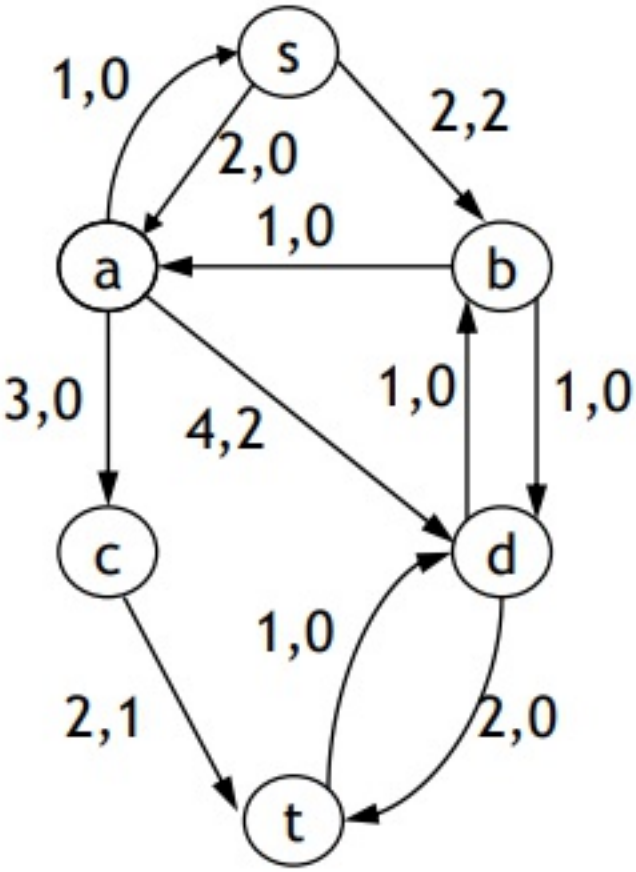
2 – Recalcular o custos das arestas tal que: w’(u,v) = w(u,v) + d(u) - d(v)



A seleção do caminho de custo mínimo pode ser agora realizada com pesquisa simples (DFS – tempo linear), visto que os caminhos de custo mínimo apenas percorrem arestas de custo recalculado 0.



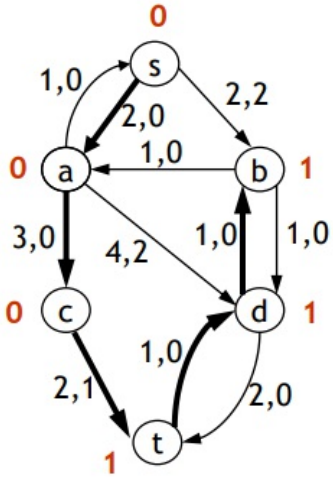
Aplicar caminho de aumento (não são inseridas arestas de custo negativo, porque o caminho percorrido tem sempre custo 0)

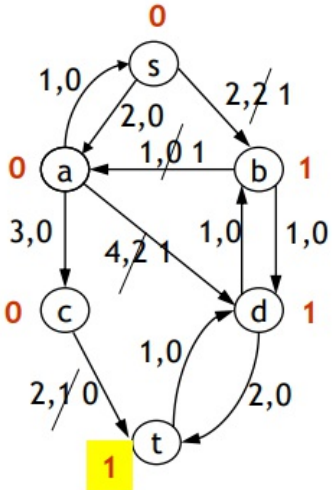


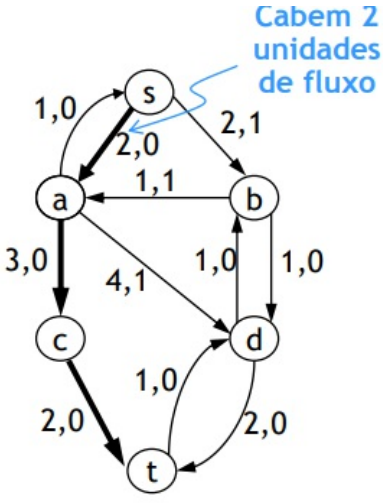
(quantidade atual do fluxo = 1)

(custo atual do fluxo = 1)

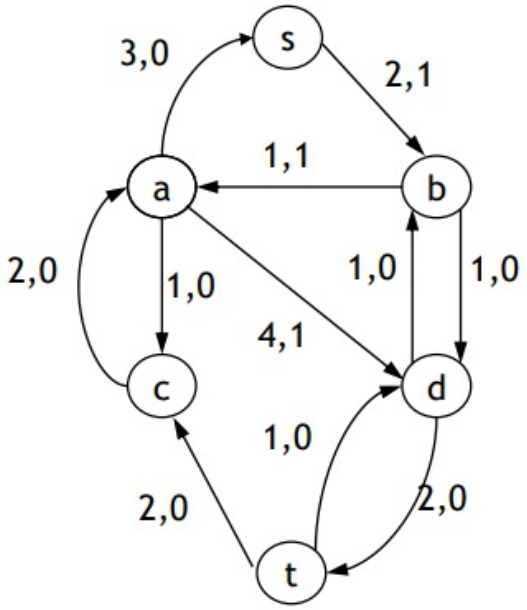
Quando já não existem caminhos de custo 0, reconverte-se o grafo







Repetição do processo



Quantidade atual de fluxo = 3 = pretendido => FIM

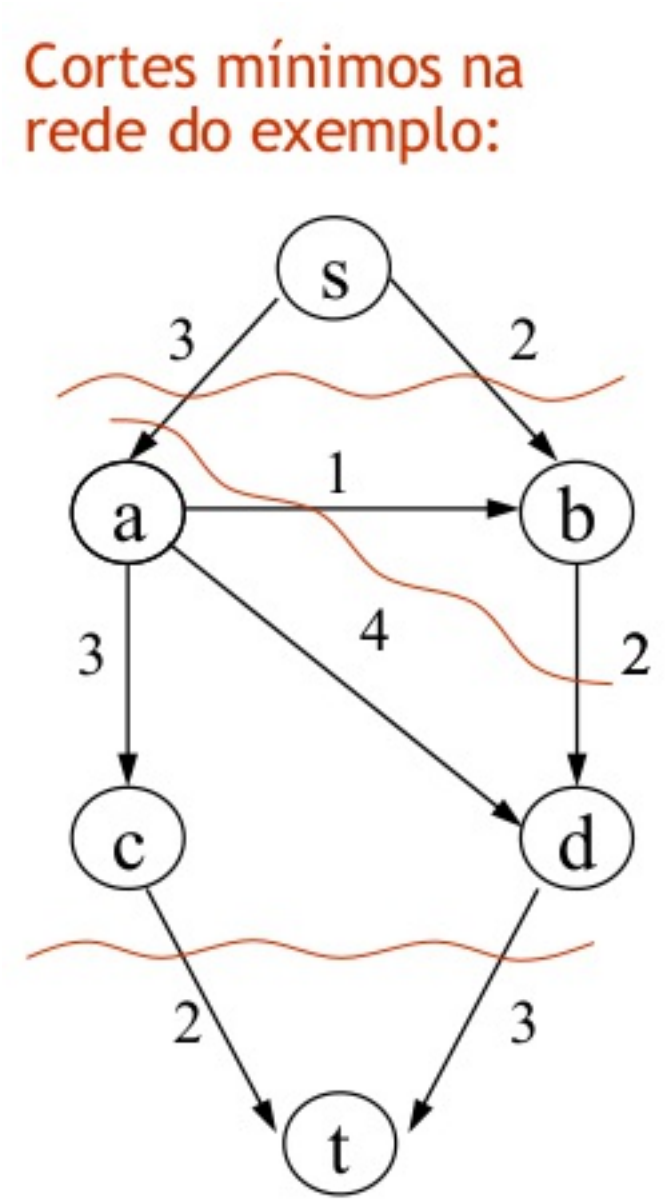
Custo atual do fluxo = 5

Eficiência temporal

- Primeira redução - O (|V| |E|) – Bellman-Ford
- Subsequentes reduções – O(|E| log|V|) – Dijkstra
- Tempo total – O(F |E| log|V|)

Dualidade entre fluxo máximo e corte mínimo

O valor do fluxo máximo numa rede de transporte é igual à capacidade do corte mínimo.
Um corte numa rede de transporte é uma partição dos vértices em dois grupos (S,T). A capacidade é a soma das capacidades das arestas dirigidas de S para T.
Um corte mínimo é um corte cuja capacidade é mínima.



Circuito de Euler

Caminho de Euler: caminho que visita cada aresta exatamente uma vez
Circuito de Euler: caminho de Euler que começa e acaba no mesmo vértice

Condições necessárias e suficientes

Grafo não dirigido

- contém circuito de Euler sse é conexo, e cada vértice tem nº de arestas incidentes (grau) par.
- contém caminho de Euler sse é conexo e todos os vértices menos dois têm grau par (estes vértices são os de início e fim do caminho).

Grafo dirigido

- contém circuito de Euler sse é fortemente conexo e cada vértice tem o mesmo grau de entrada e saída.

- contém caminho de Euler sse é fortemente conexo e todos os vértices menos dois têm o mesmo grau de entrada e de saída. A diferença dos graus desses dois vértices deve ser 1. Esses dois serão também o início, e o fim do caminho.

Encontrar circuito/caminho de Euler

1. Se caminho, começar com um dos vértices de grau ímpar, senão, encontrar vértice qualquer e efetuar pesquisa em profundidade a partir desse vértice.
    - visitar vértice, se tiver arestas incidentes não visitadas, escolher uma dessas arestas, marcá-la como visitada (ignorá-la daqui para a frente), e visitar vértice adjacente
2. Enquanto existirem arestas por visitar
    - Procurar primeiro vértice no caminho obtido até ao momento que possua aresta não percorrida
    - Lançar sub pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas)
    - Inserir o resultado no caminho principal

Tempo de execução: O (|E| + |V|)

Problema do carteiro chinês

Dado um grafo pesado conexo, encontrar um caminho com início e fim no mesmo vértice que atravessa cada aresta pelo menos uma vez.

- Chama-se problema do carteiro.
- Se for de peso mínimo, é percurso ótimo do carteiro Chinês.

Método para grafos não dirigidos

1. Achar todos os vértices de grau ímpar. Este nº tem que ser par obrigatoriamente! Se o nº de de vértices for 0, saltar para 6.
2. Achar os caminhos mais curtos e distâncias mínimas entre todos os pares dos vértices de grau ímpar (fazer tabela).
3. Construir grafo completo G' com os vértices de grau ímpar ligados entre si com arestas de peso igual aos calculados em 2.
4. Emparelhar os vértices de G' de modo a envolvê-los a todos, e a minimizar a soma das distâncias entre os vértices.
5. Para cada par, adicionar arestas duplicadas a G, de caminho mais curto entre os dois vértices. Seja G* o grafo resultante.
6. Achar um circuito de Euler em G*.

Grafos dirigidos

1. Para cada vértice, colocar informação sobre nº de arestas a entrar menos a sair - seja diff.
2. Determinar caminhos mais curtos de vértices com diff positivo para vértices com diff negativo. Representar distâncias num grafo bipartido G'.
3. Formular problema de emparelhamento óptimo como problema de fluxo máximo de custo mínimo e resolver.

4. Duplicar em G os caminhos mais curtos entre os vértices emparelhados em 3, e obter o circuito Euleriano.

Emparelhamento e Casamentos Estáveis

Emparelhamento

Conceito de Emparelhamento:

Seja um grafo não dirigido G = (V, E), um emparelhamento (M) em G é um conjunto de arestas que não contém mais do que uma aresta incidente no mesmo vértice (também chamado **conjunto de arestas independentes**).

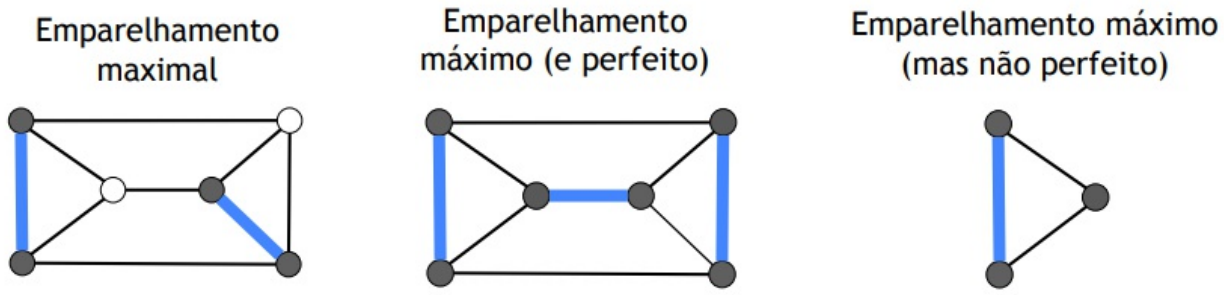
Caraterísticas de Emparelhamentos:

Emparelhamento maximal: Não pode ser aumentado.

Emparelhamento máximo: Tem tamanho máximo.

- Não é necessariamente único.
- É necessariamente maximal.
- O número v(G) é o tamanho do emparelhamento máximo.

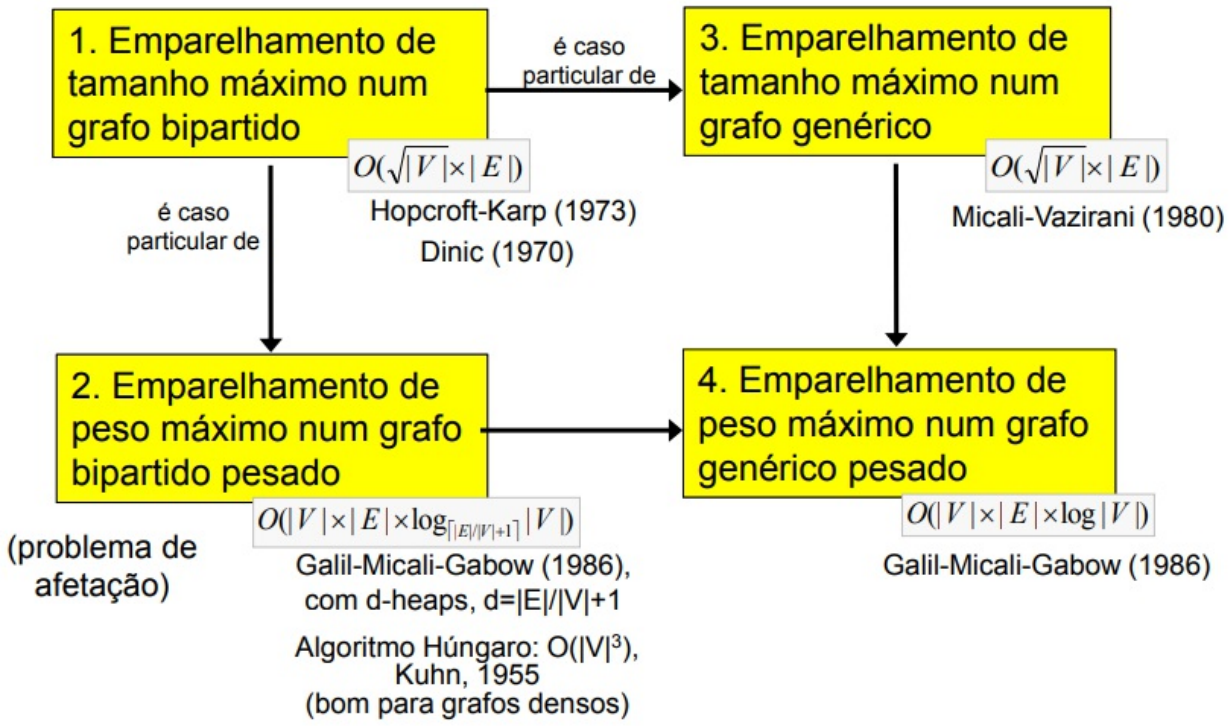
Emparelhamento perfeito: Inclui todos os vértices.



Grafo bipartido:

Grafo cujos vértices podem ser divididos em dois conjuntos disjuntos (nenhum elemento em comum) U e V, tal que todas as arestas de G ligam um vértice u de U a um vértice v de V.

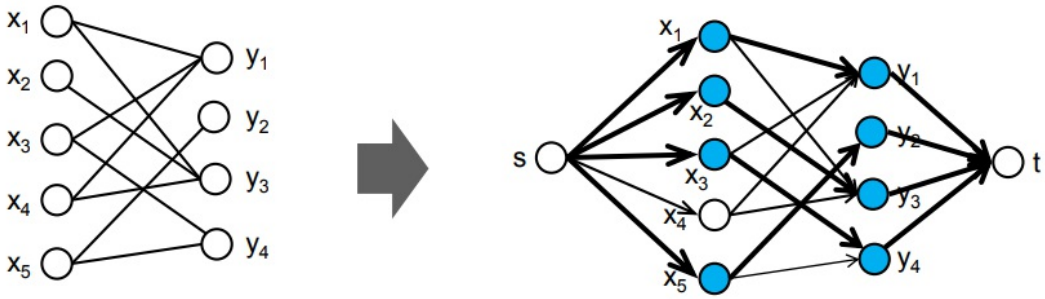
Problemas de Emparelhamento



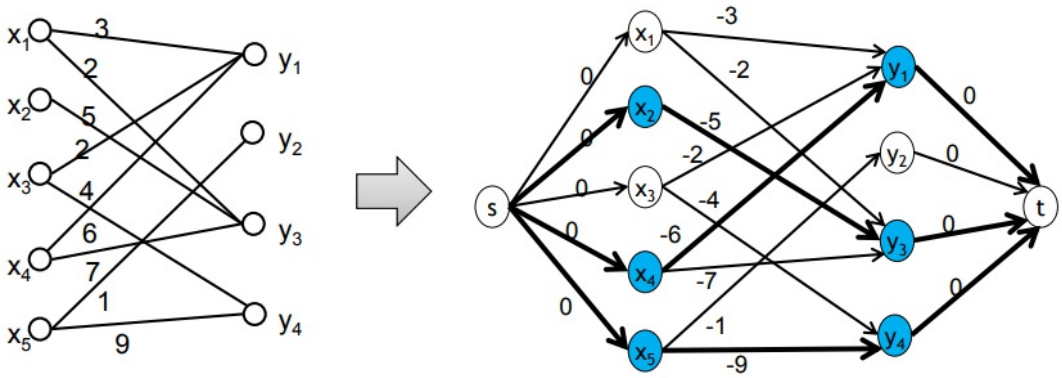
Redução a problemas em redes de transporte

Problemas de emparelhamento em grafos bipartidos são redutíveis a problemas em redes de transporte (com capacidades unitárias):

- **Emparelhamento de tamanho máximo -> fluxo máximo:**
    - Criar uma super origem e um super destino . Todas as arestas devem ter capacidade um e no final basta-nos calcular o fluxo máximo entre a super origem e o super destino.



- **Emparelhamento de peso máximo -> fluxo de custo mínimo (custo=-peso):**
    - Capacidades unitárias (logo só se mostram custos e não capacidades).
    - Custo do transporte = simétrico do peso do emparelhamento (origina arestas de custo negativo, mas não há ciclos).
    - Aplica-se método dos caminhos de aumento de custo mínimo; para-se quando o próximo caminho de aumento tem custo real >= 0



Casamentos Estáveis

Um emparelhamento (E) diz-se instável se e só se existir um par (h, m) não pertencente a E tal que, h prefere m à sua parceira em E e m também prefere h ao seu parceiro em E. Caso contrário, diz-se estável.

Algoritmo de Gale-Shapley (1962)

ALGORITMO DE GALE-SHAPLEY (1962)

Considerar inicialmente que todas as pessoas estão livres.
Enquanto houver algum homem h livre fazer:
    seja m a primeira mulher na lista de h a quem este ainda não se propôs;
    se m estiver livre então
        emparelhar h e m (ficam noivos)
    senão
        se m preferir h ao seu actual noivo h' então
            emparelhar h e m (ficam noivos), voltando h' a estar livre
        senão
            m rejeita h e assim h continua livre.
fim

Tempo de execução: O(n^2).

Listas de preferências incompletas:

Surgiu na colocação de internos em hospitais.
Neste caso, um emparelhamento é instável se e só se existir um candidato R e um hospital H tais que:

- H é aceitável para R e R é aceitável para H.
- R não ficou colocado ou prefere H ao seu atual hospital.
- H ficou com vagas por preencher ou prefere R a pelo menos um dos candidatos com que ficou.

Caso contrário, diz-se estável.

Algoritmo de Gale-Shapley com listas de preferências incompletas

ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS)

Considerar inicialmente que todos os internos estão livres.
Considerar também que todas as vagas nos hospitais estão livres.
Enquanto existir algum interno r livre cuja lista de preferências é não vazia
    seja h o primeiro hospital na lista de r;
    se h não tiver vagas
        seja r' o pior interno colocado provisoriamente em h;
        r' fica livre (passa a não estar colocado);
    colocar provisoriamente r em h;
    se h ficar sem vagas então
        seja s o pior dos colocados provisoriamente em h;
        para cada sucessor s' de s na lista de h
            remover s' e h das respectivas listas
fim

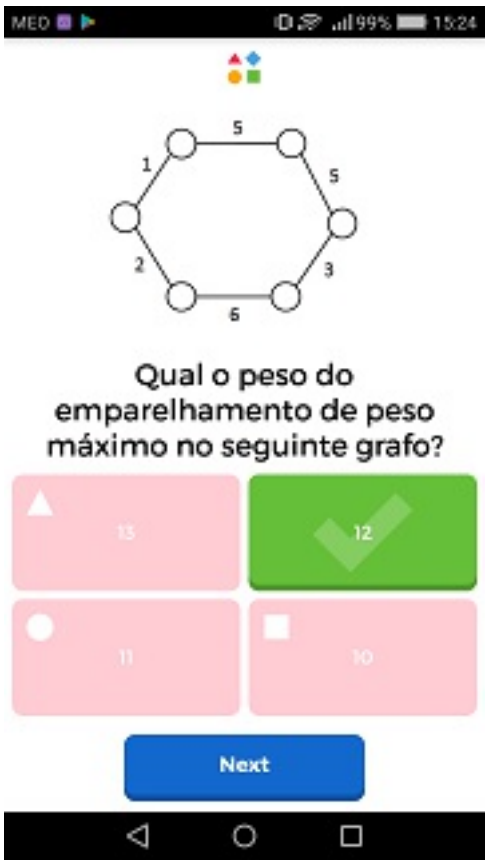
Tempo de execução: O(nº internos x nº hospitais).

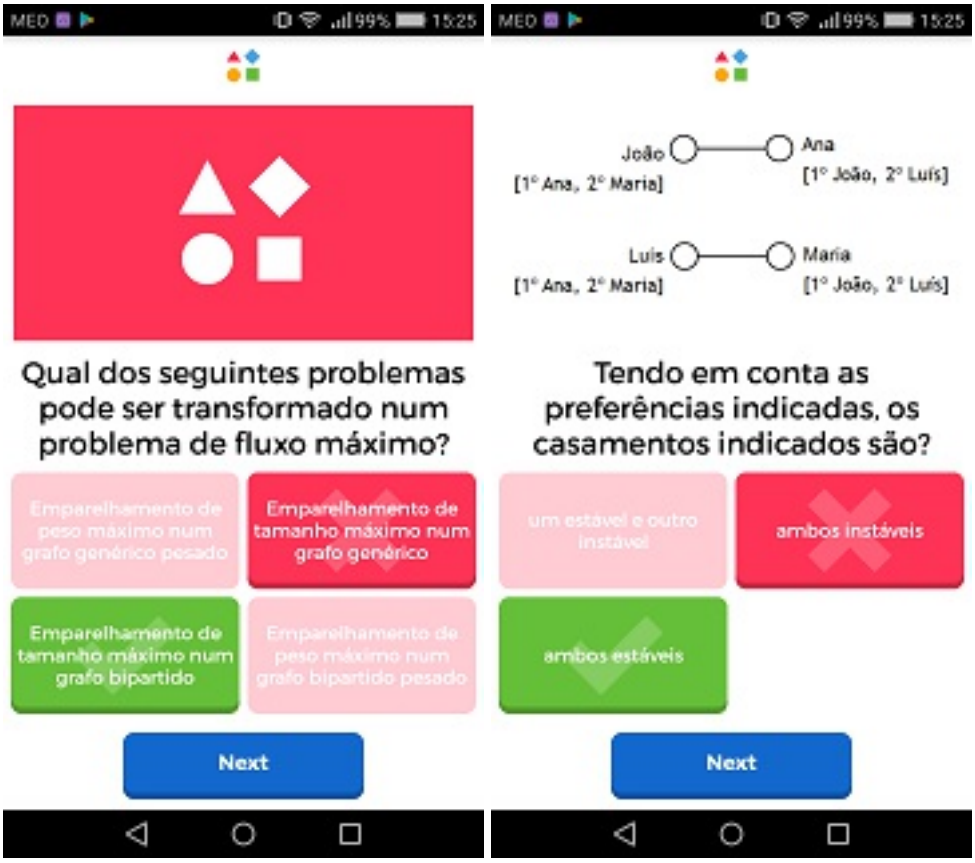
Kahoots


MED
99%
15:24
Um emparelhamentos de tamanho máximo é necessariamente ... (selecionar a afirmação verdadeira)
perfeito
não vazio
único
maximal
Next


MED
99%
15:24
G1
G2
G3
G4
Qual dos seguintes grafos não é bipartido?
G1
G2
G3
G4
Next


MED
99%
15:24
Qual é o tamanho do emparelhamento de tamanho máximo no seguinte grafo?
3
4
2
5
Next


MED
99%
15:24
5
1
5
2
3
6
Qual o peso do emparelhamento de peso máximo no seguinte grafo?
13
12
11
10
Next



Strings

Pesquisa Exata

Consiste em encontrar todas as ocorrências de um padrão T num texto T.

- Ocorrências são definidas pela deslocação em relação ao início do texto
- Ocorrências podem ser sobrepostas

Algoritmos de Knuth-Morris-Pratt

```
void KMPMatcher(string text, string pattern)
{

    int m = pattern.size();
    int n = text.size();
    int q = 0;
    vector<int> pi = ComputePrefixFunction(pattern);

    for (int i = 0; i < n; i++)
    {
        while (q > 0 && pattern.at(q) != text.at(i))
            q = pi.at(q - 1);

        if (pattern.at(q) == text.at(i))
            q++;

        if (q == m)
        {
            cout << "Pattern occurs with shift " << i - m << endl;
            q = pi.at(q-1);
        }
    }

}
```

Executa em O(|P|+|T|)
Usa uma técnica de pré-processamento: calcular uma função prefixo correspondente ao padrão.

Pré processamento do padrão

Função prefixo
Sendo P a pattern string.
Para calcular a função prefixo, começa-se sequencialmente a partir do início da string (q = 1)
Selecionamos o prefixo de comprimento q , e procuramos na string P, o maior prefixo que é um sufixo do prefixo selecionado, sendo que estes fixos têm que ser diferentes.

Exemplo:
P: ababaca
pre = prefixo

q = 1, PRE = 'a'.
Não encontramos em P um prefixo que seja sufixo de 'a', sem ser ele próprio: logo pi[1] = 0

q = 2, PRE = 'ab'.
Não encontramos em P um prefixo que seja sufixo de 'ab', sem ser ele próprio: logo pi[2] = 0

q = 3, PRE = 'aba'.
'a' é prefixo de P, e é também sufixo de PRE. Logo,
pi[3] = 1 (porque len(a)=1).

q=4, PRE = 'abab'
'ab' é prefixo de P, e é também sufixo de PRE. Logo, pi[4] = 2 (porque len(ab)=2).

q=5, PRE = 'ababa'
'a' é prefixo de P, e é também sufixo de PRE. Mas, 'aba' é prefixo de P, e é também sufixo de PRE. Como o comprimento deste é maior, pi[5] = 3

q=6, PRE = 'ababac'
Não encontramos em P um prefixo que seja sufixo de 'abababc', sem ser ele próprio: logo pi[6] = 0

q=7, PRE = 'ababaca'
'a' é prefixo de P, e é também sufixo de PRE. Logo, pi[7] = 1.

Algoritmo

```
vector<int> ComputePrefixFunction(string pattern)
{
    int m = pattern.size();
    vector<int> pi(m);
    int k = 0;

    for (int q = 1; q < m; q++)
    {
        while (k > 0 && pattern.at(k) != pattern.at(q))
            k = pi.at(k - 1);

        if (pattern.at(k) == pattern.at(q))
            k++;

        pi.at(q) = k;
    }

    return pi;
}
```

Pesquisa Aproximada

Distância de edição entre duas *strings*

P - pattern string, T - text string
A distância de edição é o menor número de alterações que permitem transformar T em P.
Podem ser:

- substituir carater por outro
- inserir carater
- eliminar carater

Exemplo:
P - Algoritmo
T - Algorithm
Substituir h por m, e m por o: EditDistance = 2.

Complexidade temporal: O(|P|*|T|)

Matriz de Programação Dinâmica.

Índices das strings a começar em 1. Na vertical, em cima, está P, e na horizontal, à esquerda, está T.
Inicializar primeira coluna e primeira linha com valores de 1 a size(P) ou size(T).

Para preencher as células restantes: começar na do canto superior esquerdo, e preencher a linha. De seguida, mudar de linha, começando no primeiro elemento, e assim sucessivamente.

O valor a colocar é o mínimo de:

- D[i-1, j]+1 -> cima
- D[i, j-1] + 1 -> esquerda
- D[i-1, j-1] + delta(T[i-1], P[j-1])

delta -> 1 se T[i-1] != P[j-1], senão 0

Algoritmo

```
int editDistance(string pattern, string text)
{
    int pSize = pattern.size();
    int tSize = text.size();
    int oldValue, newValue;
    vector<int> D(tSize + 1);

    for (int j = 0; j <= tSize; j++)
        D.at(j) = j;

    for (int i = 1; i <= pSize; i++)
    {
        oldValue = D.at(0);
        D.at(0) = i;

        for (int j = 1; j <= tSize; j++)
        {
            if (pattern.at(i - 1) == text.at(j - 1))
                newValue = oldValue;
            else
                newValue = 1 + min({oldValue, D.at(j), D.at(j - 1)});

            oldValue = D.at(j);
            D.at(j) = newValue;
        }

    }

    return D.at(tSize);
}
```

Compressão de Texto

Representação de carateres

Permitem representar 2^bits carateres diferentes
ASCII -> 7/8 bits
Unicode -> 16 bits
ISO -> 32 bits

Texto é simplesmente sequência de carateres. Logo pode ser represnetado pela sequência dos seus códigos

Keyword encoding

Substituir palavras muito comuns por carateres ou sequencias especiais.
As palavras são substituídas de acordo com uma tabela de frequências (ocorrências)

Exemplo:

Chave	Significado
%	carro
$	acidente
&	senhor
#	do

“No acidente estiveram envolvidos três carros. O carro do senhor António ficou destruído. O carro do senhor José não sofreu grandes danos no acidente. O carro do senhor Carlos... bom, depois do
acidente, nem se pode chamar aquilo um carro!”
241 bytes

“No $ estiveram envolvidos três carros. O % # & António ficou destruído. O % # & José não sofreu grandes danos no $. O % # & Carlos... bom, depois # $, nem se pode chamar aquilo um %!”
185bytes (76% do original)

Run-length encoding

Usado quando o mesmo padrão/letra surge muitas vezes seguidas.
Nesse caso, cada sequência é substituida por um marcador especial (*), o carater em questão, e número de vezes que aparece.
Exemplo:
AABBBBBBBBAMMKKKKKKKKKM ->
AA*B8AMM*K9M

Codificação constante

Tamanho do código
Sendo N o número de elementos do alfabeto:
ceil(log2(N))

Representação
Árvore binária, em que as folhas são os carateres.
Nas aresta da esquerda está 0, na da direita 1.
Percorrendo as arestas, encontra-se o código para cada carater.

Tendo a frequência dos carateres numa string, facilmente se encontra o tamanho do código total correspondente (ou custo de codificação): multiplicar frequência pelo tamanho do código do carater, e somar com os restantes.

Para encontrar um código para representar um alfabeto, basta concatenar os códigos de cada carater.

Codificação variável (algoritmo de Huffman)

Códigos de tamanho variável: os carateres mais frequentes têm código mais pequeno.
Usar árvore binária com símbolos nas folhas

Algoritmo

1. Inicialmente, floresta de árvores só com raíz, com os símbolos do alfabeto.
2. Cada folha tem a frequência associada, logo o peso de cada árvore é a soma dessas frequências.
   Escolher as duas árvores com pesos menores e torná-las sub-árvores de uma nova raíz.
   Empates resolvidos aleatoriamente.
3. Repetir passo anterior até restar 1 só árvore.

No final, para codificar, colocar 0 na aresta esquerda, 1 na direita, sucessivamente.
O custo de codificação calcula-se tal como antes, fazendo a soma das multiplicações da frequência relativa pelo tamanho do código.

Construção da árvore
Entrada: conjunto C de N carateres
Saída: Árvore de Huffman

```
n = len(C);
Q = C;
for(int i = 1; i < n; i++)
    CreateNode(z);
    x = z->left = ExtractMin(Q);
    y = z->right = ExtractMin(Q);
    f[z] = f[x] + f[y];
    Insert(Q,z);
return ExtractMin(Q);
```

Tempo de execução: O(N logN)

Problemas NP-Completos

Problemas de decisão

Problema cuja resposta deve ser SIM ou NÃO (ou derivados).
Problemas de otimização podem ser expressos como problemas de decisão:
Por exemplo, o problema “qual o menor número de cores que se pode utilizar para colorir um grafo?,” pode ser expresso como:
“Dado um grafo G e um inteiro k, é possível colorir G com k cores?”

A classe de problemas P é constituída por todos os problemas de decisão que podem ser resolvidos em tempo polinomial.

A classe de problemas NP

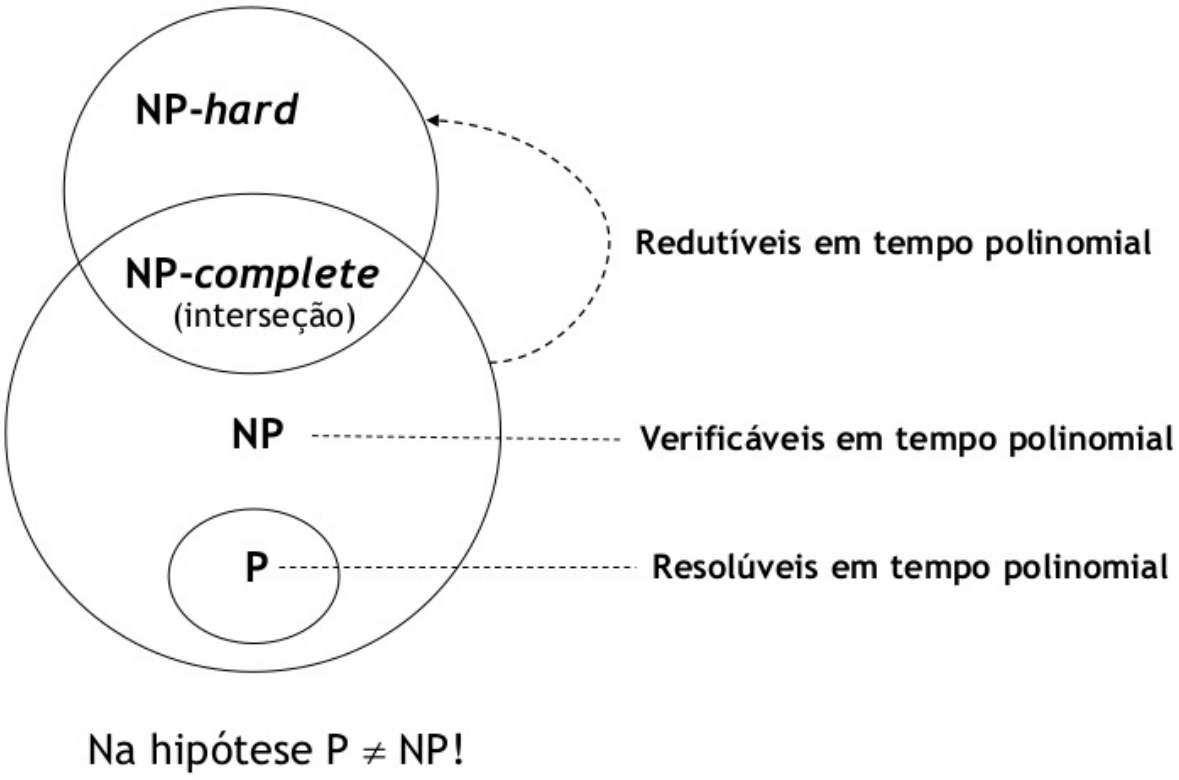
Todos os problemas que podem ser verificados por um algoritmo de tempo polinomial. Ou seja, dado um determinado problema, esse é NP, se for possível verificar em tempo polinomial se uma solução é correta.

Problemas NP-Completos

Problema de decisão A é NPC, se for NP sse todos os outros problemas NP podem ser redutíveis a A em tempo polinomial.
Atualmente, para provar que A é NPC, basta encontrar um problema A' NPC já conhecido e provar que A' é redutível a A em tempo polinomial

Problemas NP-Difíceis

Problema de decisão A é NPD sse todos os problemas NP podem ser redutíveis a A em tempo polinomial.



Provar que problema X não é P

- Selecionar problema Y não P.
- Provar que Y é redutível a X em tempo polinomial

Provar que problema X é NPC

- Provar que X é NP
- Selecionar problema Y que se sabe ser NPC
- Definir uma redução de tempo polinomial de Y em X
- Provar que dada uma instância de Y, tem solução sse X tem uma solução

Exemplos

Problemas P

- Problema do circuito Euleriano
- Problema do carteiro chinês
- Problema do fluxo máximo
- Problema da subsequência crescente mais comprida

Problemas NP

- Problema do circuito Hamiltoniano
    - Verificar se grafo não dirigido tem um ciclo que visita cada vértice exatamente uma vez
- Problema do caixeiro viajante (TSP)
    - Num grafo não dirigido com pesos não negativos, encontrar um ciclo de peso total mínimo que passa em todos os vértices.
    - Mesmo que: Dado um grafo não dirigido, com pesos inteiros nao negativos, existe um ciclo de peso <=k (k inteiro não negativo) que passa em todos os vértices?
    - Algoritmos:
        - Held-Karp: O(n^2 2^n)
        - Com desigualdade triangular, visita em pré-ordem de árvore de expansão mínima dá solução de peso que não excede 2x o ótimo
        - No mesmo pressuposto, algoritmo de Christofides, garante solução que não excede em 1.5x o ótimo
- Problema da cobertura de vértices
    - Uma cobertura de vértices é um subconjunto de vértices tal que toda a aresta do grafo é incidente em pelo menos um vértice desse subconjunto
- Problema do clique
    - Determinar se um grafo tem sub-grafo completo de tamanho máximo
- SAT
    - Determinar se uma expressão booleana é satisfazível (pode retornar verdadeiro)
- Problema dos subconjuntos
- Problema da Coloração de Grafos
    - Dado um grafo, colori-lo é definir rótulos, ou cores, para todos os vértices tal que não haja uma aresta em que os seus vértices tenham a mesma cor.
- Problema da paragem é NP-difícil, mas não NP-completo
- Conjunto Independente
    - Encontrar um subconjunto de vértices tal que não há dois vértices que partilham uma aresta.

Resolução de problemas

No geral, pedem para transformar problema de otimização em problema de decisão. Na alínea seguinte, deve-se usar os problemas sugeridos para provar que o problema dado é NP-Completo, sendo que temos que reduzir um dos problemas sugeridos ao do enunciado.

1. Dizer que problema é NP, e dar um exemplo de como verificar solução em tempo polinomial.
2. Reduzir problema sugerido ao dado.

Marcação de exames (2017 Normal)
Estudantes podem inscrever-se em vários cursos.
Todos os exames finais terão duração de 1 hora.
Determinar o número mínimo de slots de exame a fim de evitar que estudantes inscritos em vários cursos tenham exames sobrepostos.

Reformular este problema como problema de decisão

Determinar se é possível usar um número de slots <= k, a fim de evitar que estudantes inscritos em vários cursos tenham exames sobrepostos.

Verifique se há uma solução eficiente para este problema, explicando os passos da sua solução
Sugestão: usar Cobertura de vértices ou Coloração de Grafos

Os problemas NP são aqueles que podem ser verificados em tempo polinomial, ou seja, têm uma solução eficiente. Como nos dão a sugestão acima, pretende-se provar que este problema é NP-Completo.

1. Dizer que problema é NP, pois uma solução candidata pode ser verificada facilmente em tempo polinomial. Basta verificar se o nº de slots é efetivamente <= k e percorrer a lista de estudantes e verificar se algum estudante tem 2 exames marcados no mesmo slot.

2. Reduzir, por exemplo, o problema da Coloração de Grafos, em tempo polinomial, ao problema da marcação de exames.

   Se no PCG duas arestas não podem ter vértices com a mesma cor, é o mesmo que dizer que esses vértices são incompatíveis. Portanto, podemos admitir que num grafo, os vértices são os cursos, e as arestas são os estudantes -> onde os estudantes ligam cursos aos quais estão inscritos. Dessa forma, os cursos ligados pela mesma aresta são incompatíveis, por isso devem ter cores diferentes.

Problema do Clique (Recurso 2017)
Um clique de um grafo não dirigido é um subconjunto dos seus vértices, tal que, para quaisquer pares de vértices u e v neste subconjunto, existe uma aresta do grafo que liga os vértices u
e v. O problema de otimização consiste em encontrar um clique de tamanho máximo.

Problema de decisão
Dado um grafo não dirigido G=(V,E) e um k inteiro positivo, verificar se G tem um clique de tamanho >= k?

Verifique se há uma solução eficiente para este problema, explicando os passos da sua solução
Sugestão: usar Coloração de Grafos ou Conjunto Independente

1. Dizer que um clique candidato pode facilmente ser verificado em tempo polinomial.
2. Provar que é NP-Completo, reduzindo Conjunto Independente ao problema do clique. Começar por ilustrar um grafo, e fazer o seu complemento. Se encontrarmos o clique máximo deste grafo, então encontramos o maior conjunto independente do grafo original.

Resumo dos algoritmos

Algoritmos gananciosos

Aplicável a problemas de optimização. (maximização ou minimização)

Programação dinâmica

Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares)

... Mas em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema)

Abordagem:

1o) Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos subproblemas (gastando memória!)

2o) Economizar memória, resolvendo subproblemas por ordem que minimiza no de soluções parciais a memorizar (bottom-up, começando pelos casos base).

Retrocesso

Contexto geral de aplicação:

Explorar um espaço de estados à procura dum estado-objetivo
Estado = estado de jogo, subproblema, solução parcial, etc.
Sem algoritmos eficientes que levem directamente ao objetivo.

Exemplos: Problema do troco com limitações de stock, Sudoku, 8 Rainhas, Labirintos

GRAFOS

Pesquisa e Ordenação

Pesquisa em profundidade (depth-first search)

Pesquisa em largura (breadth-first search)

Ordenação topológica – algoritmo

O(|V|+|E|)

Aplicações: Grafos Acíclicos Dirigidos (DAG) -– grafo dirigido sem ciclos

Caminho mais curto

Caminhos mais curtos de um vértice para todos os outros

Caso de grafos dirigidos não pesados

- baseado em pesquisa em largura, O(|V| + |E|)

Caso de grafos dirigidos pesados

- Dijkstra, algoritmo ganancioso, O((|V| + |E|) log |V|)

Caso de grafos dirigidos com arestas de peso negativo

- Bellman-Ford, programação dinâmica, O(|E| |V|)

Caso de grafos dirigidos acíclicos

- baseado em ordenação topológica, O(|V| + |E|)

Caminho mais curto entre dois pontos numa rede viária

Método base

Pesquisa bidirecional

Pesquisa orientada

Redes hierárquicas (highway networks)

Nós de trânsito (transit-node routing)

Caminho mais curto entre todos os pares de vértices

Algoritmo de Floyd-Warshall O(V^3).

Algoritmos em Grafos: Fluxo Máximo em Redes de Transporte

Exemplos de aplicação

Rede de abastecimento de líquido ponto a ponto

Tráfego entre dois pontos

Emparelhamento máximo em grafos bipartidos (maximum bipartite matching).

Algoritmo de Ford-Fulkerson

Análise:

Se as capacidades forem números racionais, o algoritmo termina com o fluxo máximo

Se as capacidades forem inteiros e o fluxo máximo M

• Algoritmo tem a propriedade de integralidade: os fluxos finais são também inteiros

• Bastam M iterações (fluxo aumenta pelo menos 1 por iteração)

• Cada iteração pode ser feita em tempo O(|E|)

• Tempo de execução total: O(M |E|) – mau

Algoritmo de Edmonds-Karp

Tempo de execução: O(|V| |E|^2)

Algoritmos em grafos: árvore de expansão mínima (minimum spanning tree)

Algoritmo de Prim

- O(|V|^2) sem fila de prioridade
- O(|E| log |V|) com fila de prioridade

Algoritmo de Kruskal

tempo no pior caso O(|E| log |E|), dominado pelas operações na fila

como |E| ≤ |V|2 , log |E| ≤ 2 log |V|, logo eficiência é também O(|E| log |V|)

Algoritmos em Grafos: Conectividade

Grafos não dirigidos

Conetividade
Pesquisa em profundidade
Biconectividade e Pontos de Articulação
Algoritmo de detecção de pontos de articulação
Cálculo de Low(v) - O(|E| + |V|)

Grafos dirigidos

Componentes fortemente conexos
Árvore de expansão
Componentes fortemente conexos
Numeração em pós-ordem

Kahoots

Programação Dinâmica

- Cálculo Combinações nCk: O (n*k)
- Cálculo Sequência Fibonacci: O(n) tempo; O(1) espaço
- Cálculo com divisões por 2n: O(log(n))
- Programação dinâmica => Melhoramento de casos recursivos com mais do que uma chamada
- Programação dinâmica => evita a resolução repetida de subproblemas sobrepostos
- Memorization => melhora a complexidade temporal (em detrimento da complexidade espacial)

Fluxo Máximo

- Algoritmo de cálculo: Ford-Fulkerson

- Eficiência de Edmonds-Karp: O (|V| |E|2)
- Grafos envolvidos no algoritmo: capacidades, fluxos e resíduos
- Caminho custo mínimo com arestas de custo negativo: Bellman-Ford
- Eficiência de Bellman-Ford: O (|V| |E|)
- Algoritmo para fluxo de custo mínimo: Ford-Fulkerson com aumentos de peso mínimo

###Caminho de Euler

- Caminho de Euler: caminho que visita cada aresta de um grafo exatamente uma vez
- Grafo não dirigido contém caminho de Euler sse for conexo e todos menos dois vértices tem grau par
- Grafo dirigido contém caminho de Euler sse for conexo e cada vértice temo mesmo grau de entrada e saída
- Encontrar circuito de Euler: O (|E| + |V|)
- Percurso ótimo do carteiro chinês: caminho fechado, de peso mínimo, que atravessa cada aresta **pelo menos** uma vez

Emparelhamento e casamentos estáveis

- Emparelhamento associa a cada vértice um e só um outro vértice
- Emparelhamentos máximos (contêm o número máximo de vértices possível de emparelhar no grafo) é sempre maximal (não podem ser criadas novas associações nesse emparelhamento), mas o inverso não é verdade
- Emparelhamento de tamanho máximo num grafo bipartido pode ser transformado num problema de fluxo máximo

Pesquisa em strings

- Knuth-Morris-Pratt: O(|padrão| + |texto|)
- Knuth-Morris-Pratt: Pré-processamento – cálculo da função prefixo
- Distância de edição = nº de inserções/eliminações/substituições de caracteres

Compressão de texto

- Métodos de compressão: keyword encding, Huffman codes, Run-length enconding
- Codificação constante: cada símbolo é codificado com um número igual de bits
- Método de Huffman: códigos de tamanho variável, utilização de árvore binária, minimização do custo de codificação
- Método de Huffman: O(N2)

NP-Completos

- Problemas P – resolúveis em tempo polinomial
- Problemas NP – verificáveis em tempo polinomial
- Problema NP-difícil – todos os problemas NP são redutíveis nesse problema
- Exemplo de um problema não P – circuito Hamiltoniano
- Exemplo de um problema P – subsequência crescente mais comprida
- Para q X seja NP-completo, então existe Y tal que Y é NP-completo e Y é redutível em X em tempo polinomial (ou seja, X é NP e NP-difícil)
- Exemplo de um problema NP-difícil mas não NP-completo – Problema da paragem
- SAT => satisfação booleana