

Grafos V

Joaquim Madeira

04/06/2020

Sumário

- Recap
- Determinação de Ciclos Hamiltonianos (“Hamilton Tour”)
- Procura Exaustiva
- O Problema do Caixeiro Viajante (“The Traveling Salesman Problem”)
- O Problema da Mochila (“The 0-1 Knapsack Problem”)
- Geração de Quadrados Mágicos
- O Problema da Soma de Subconjuntos (“The Subset Sum Problem”)
- Sugestão de leitura

Let's
RECAP

Recapitulação

Problemas de Otimização Combinatória

- Conjunto finito de soluções
- De entre todas as soluções, determinar **a(uma) solução ótima**
 - Podem existir **soluções ótimas alternativas**
- Árvore dos **caminhos mais curtos** com origem num vértice s
 - Algoritmo de Dijkstra
- MST – Árvore geradora de **custo total mínimo**
 - Algoritmo de **Kruskal**
 - Algoritmo de **Prim**

Algoritmos Vorazes / “Greedy”

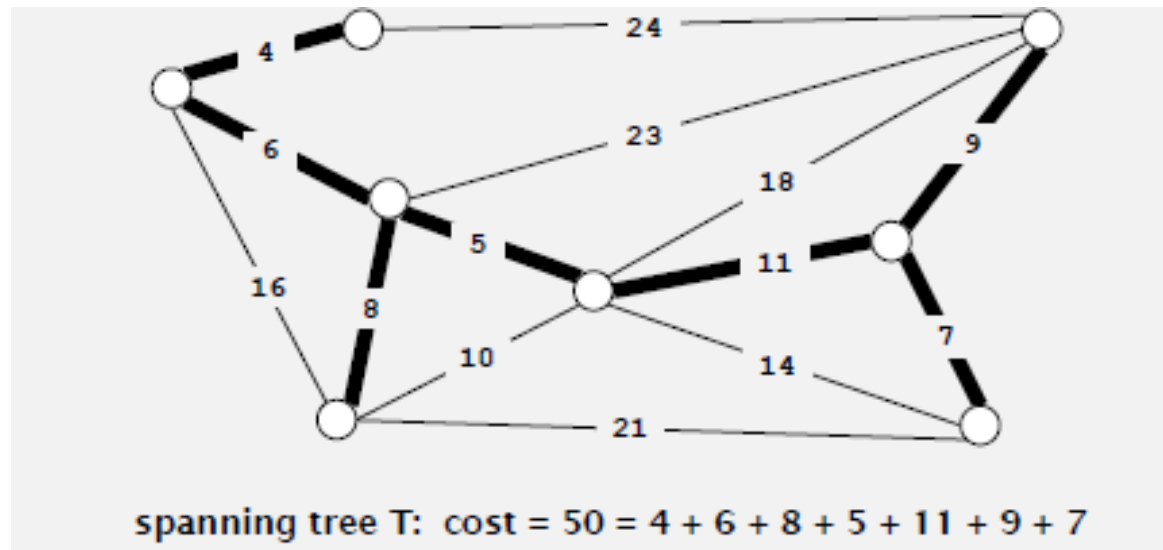
- Construir a solução **passo a passo**
- Efetuando uma sucessão de **escolhas localmente ótimas**
- E que são **irreversíveis**

- Usar uma **PRIORITY QUEUE** baseada numa **HEAP** binária
- Obter o próximo vértice / aresta sem grande esforço computacional

- Há **outras estruturas de dados** que se podem usar
- A **ordem de complexidade** do algoritmo depende dessa escolha

Árvore Geradora de Custo Mínimo

- Determinar a (uma) árvore geradora de **custo total mínimo**
 - Soma dos pesos associados às arestas da árvore
 - Assegurar a **conectividade** entre qualquer par de nós com o **menor custo**

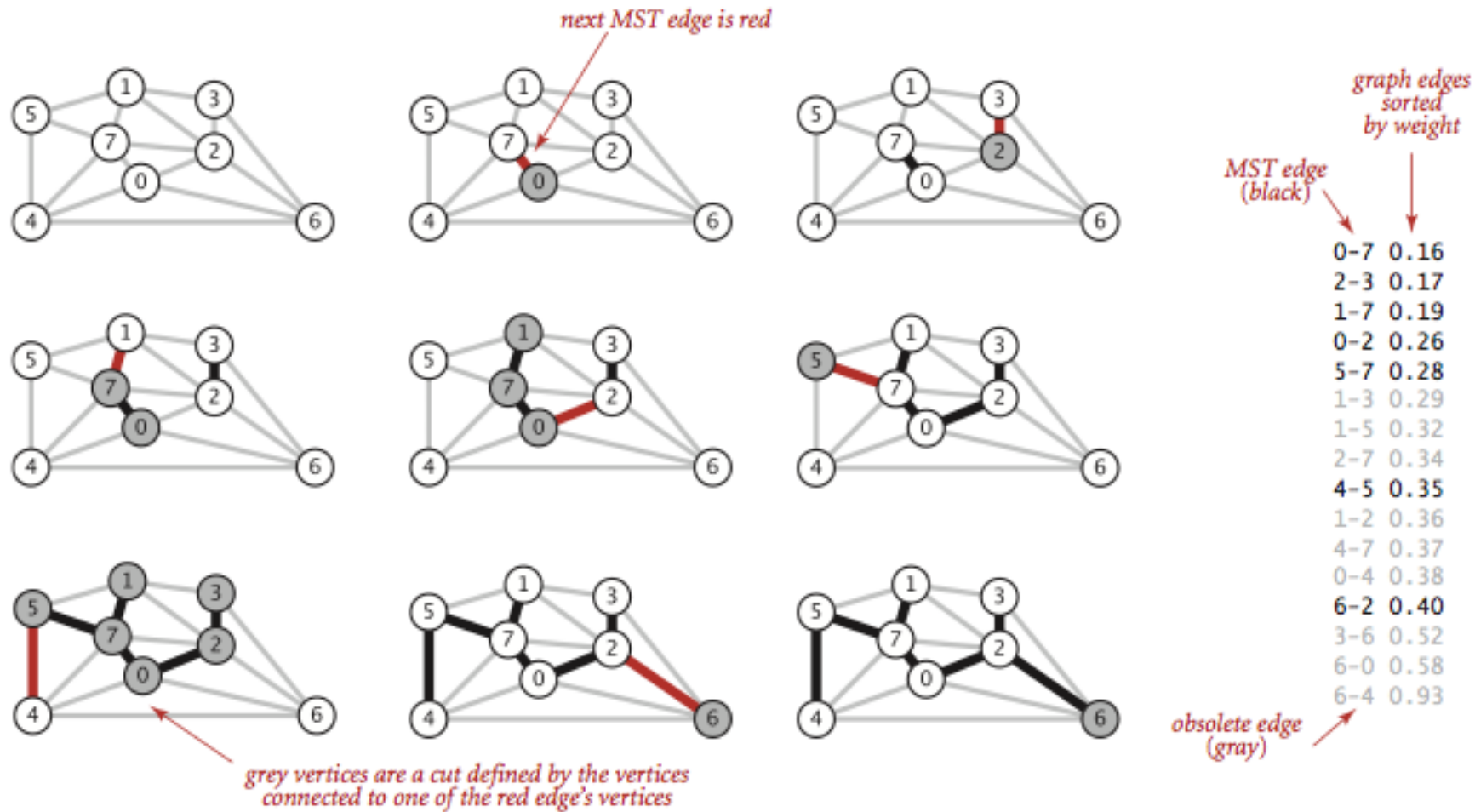


[Sedgewick & Wayne]

Algoritmo de Kruskal

- Ordenar as arestas de acordo com o custo associado
- Começar com uma **floresta** de **árvores**, cada uma com **um só vértice**
- Sucessivamente adicionar uma **aresta de menor custo** que não origina um ciclo
 - **Reunião** de duas árvores
 - Como verificar que não se forma um **ciclo** ?

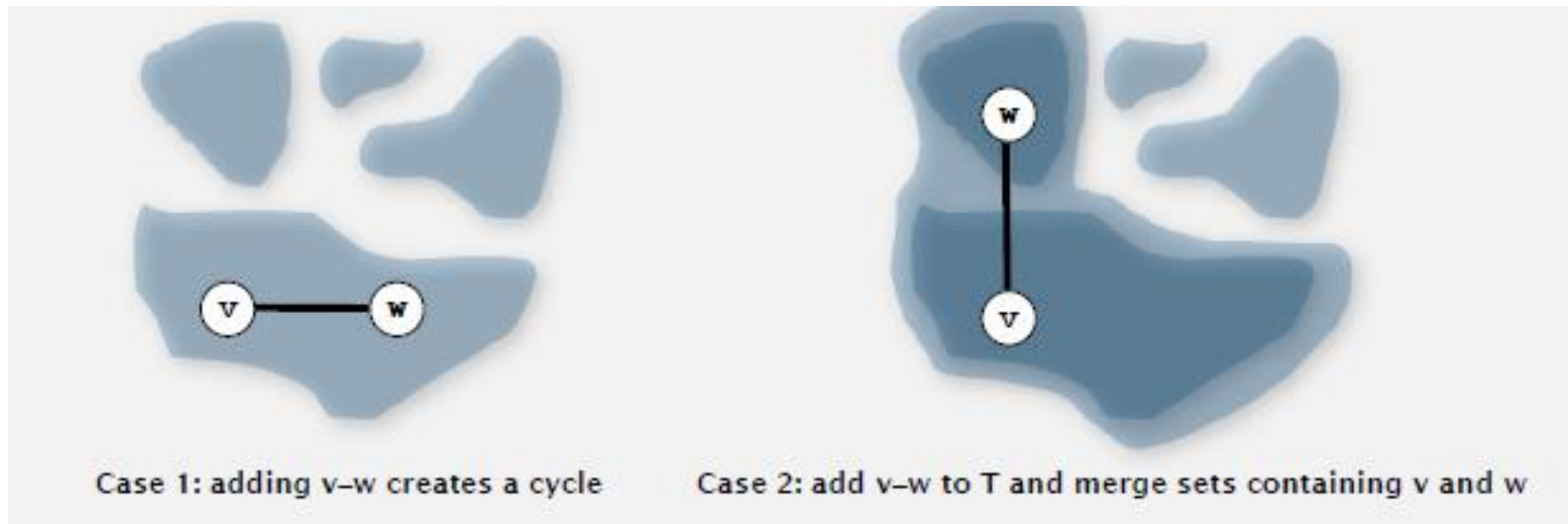
Algoritmo de Kruskal



[Sedgewick & Wayne]

Como verificar que não se forma um ciclo ?

- Estrutura de dados **UNION-FIND**

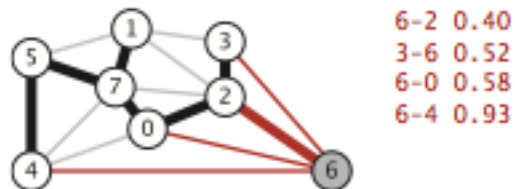
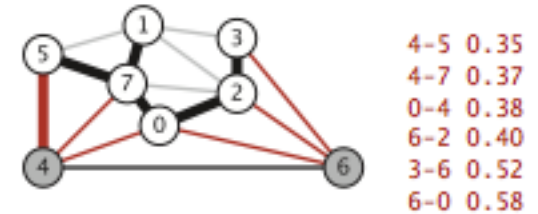
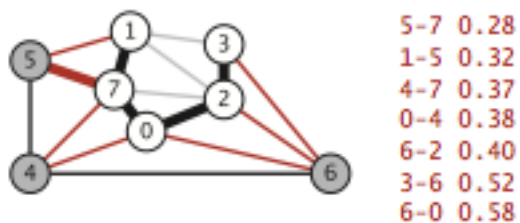
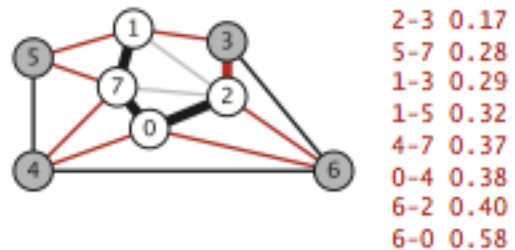
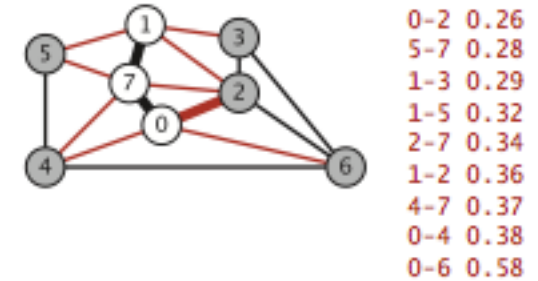
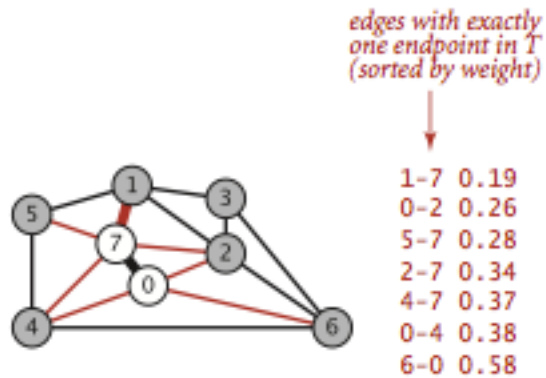
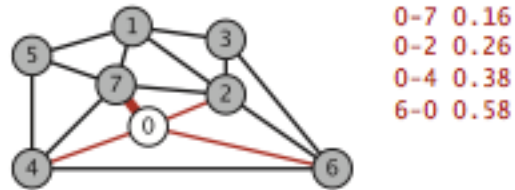


[Sedgewick & Wayne]

Algoritmo de Prim

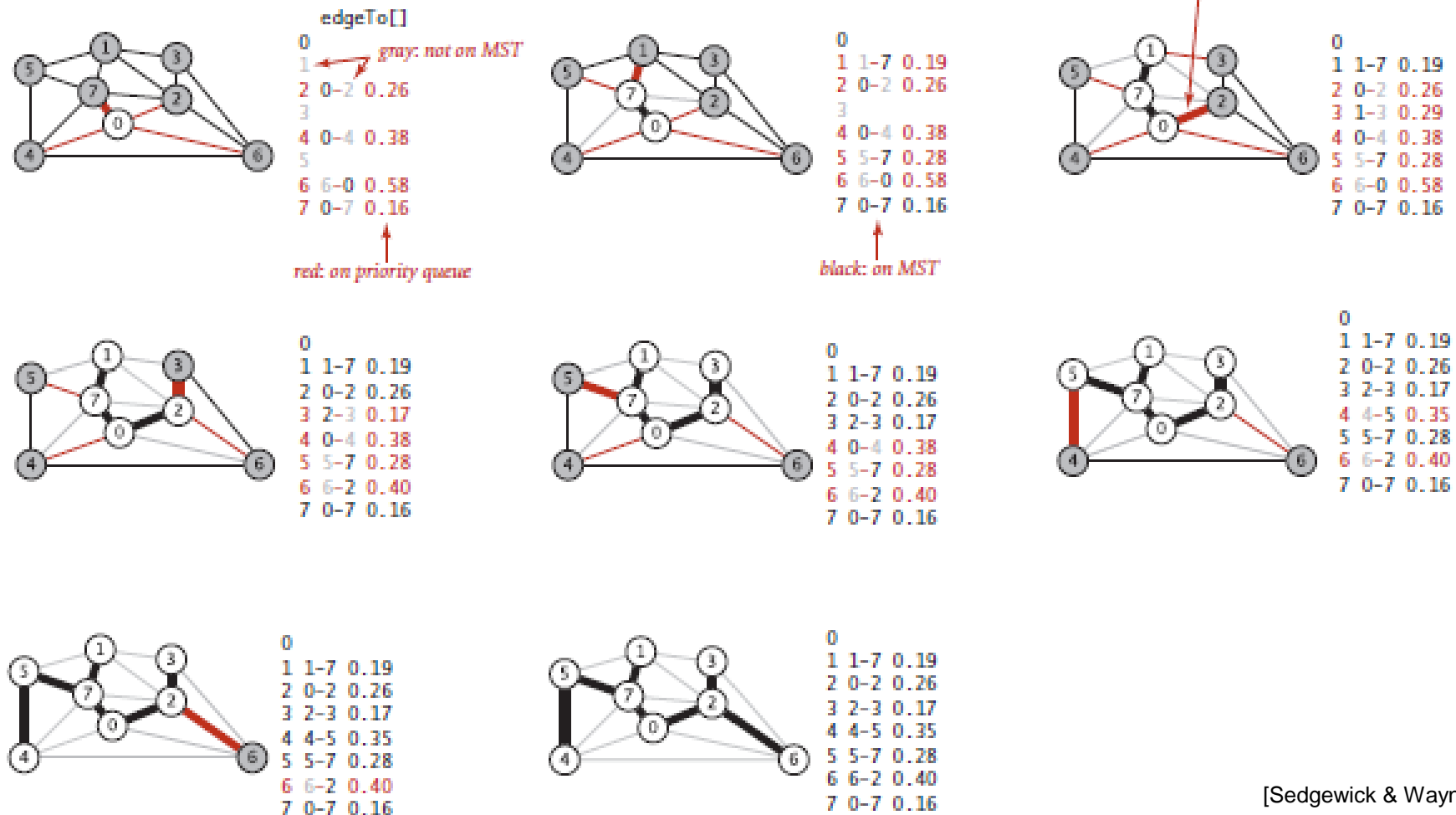
- Começar com **uma árvore** T , com **um só vértice** de G
- Sucessivamente adicionar uma aresta a T : uma aresta mais curta com (apenas) um dos seus vértices em T
 - Não se cria um ciclo !!
- Manter o conjunto de **arestas candidatas**
- Usar uma **PRIORITY-QUEUE**

Algoritmo de Prim



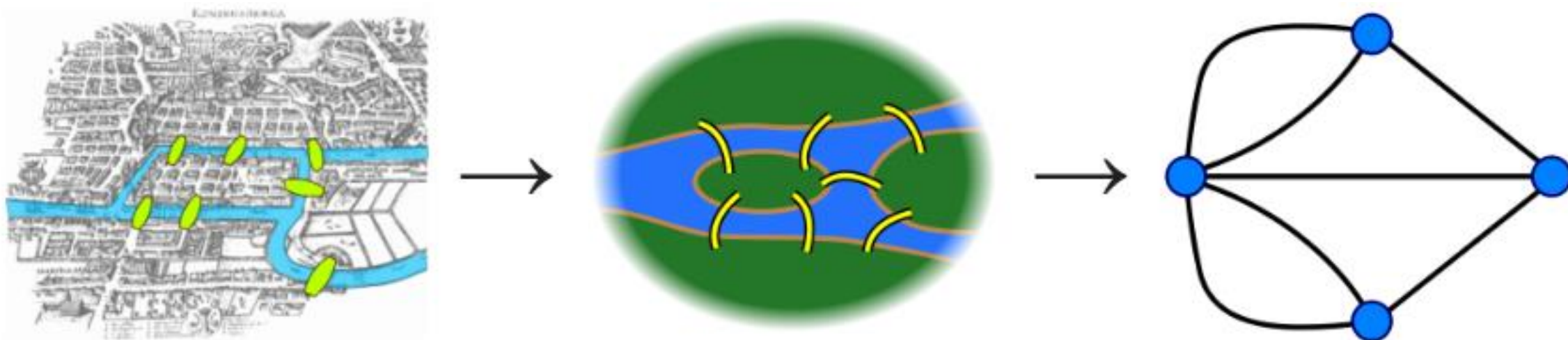
[Sedgewick & Wayne]

Estratégia alternativa



[Sedgewick & Wayne]

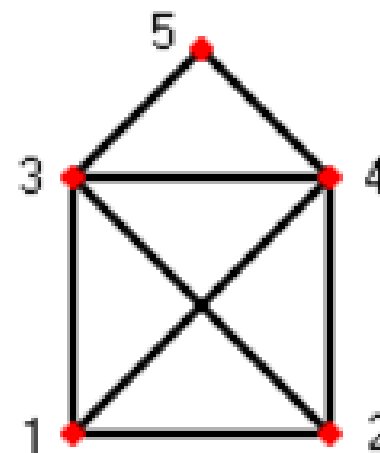
Problema das Pontes de Königsberg (1736)



[Wikipedia]

Caminho Euleriano / Circuito Euleriano

- Grafo / Grafo orientado
 - **Caminho** que contém uma única vez cada uma das arestas de um grafo
 - **Circuito** que contém uma única vez cada uma das arestas de um grafo
 - Qual é a **sequência de arestas** ?
-
- Há algum caminho Euleriano ?
 - Há algum circuito Euleriano ?



[Wikipedia]

Algoritmo de Fleury (1883)

- **Assegurar** que cada vértice tem grau par
- v = escolher um **qualquer vértice inicial**
- Em v , escolher a **próxima aresta (v, w)** da solução

Condição : a escolha de (v, w) não torna o grafo desconexo, a menos que seja a única escolha possível – **ponte / istmo** ?

Adicionar (v, w) à solução

$v = w$

Apagar (v, w)

Apagar v , se é agora um vértice isolado

Problemas ?

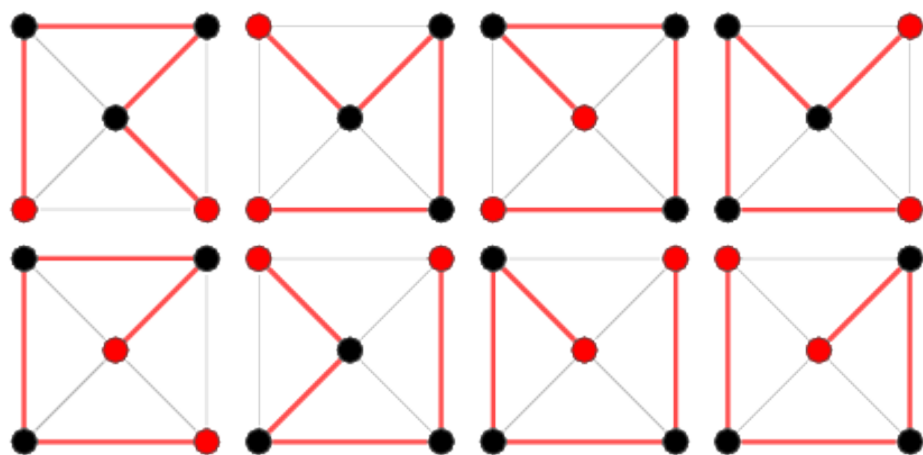
- Trabalhar sobre uma **cópia** do grafo dado
- Como verificar se uma aresta é um **istmo / ponte** e não pode ser apagada sem tornar o grafo desconexo ?
- Remover tentativamente a aresta e verificar se os outros vértices continuam a ser **alcançáveis**
 - Sucessivas **travessias em profundidade**, por exemplo
- Há algoritmos alternativos mais eficientes...

Caminhos e Circuitos

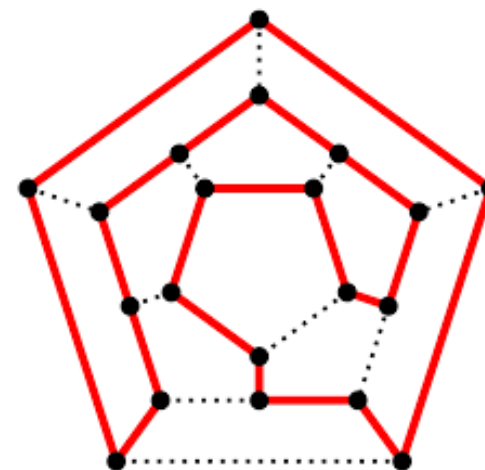
Hamiltonianos

Caminho Hamiltoniano / Ciclo Hamiltoniano

- Grafo / Grafo orientado
- **Caminho** que contém uma única vez **cada um dos vértices** de um grafo
- **Ciclo** que contém uma única vez **cada um dos vértices** de um grafo



[Mathworld]



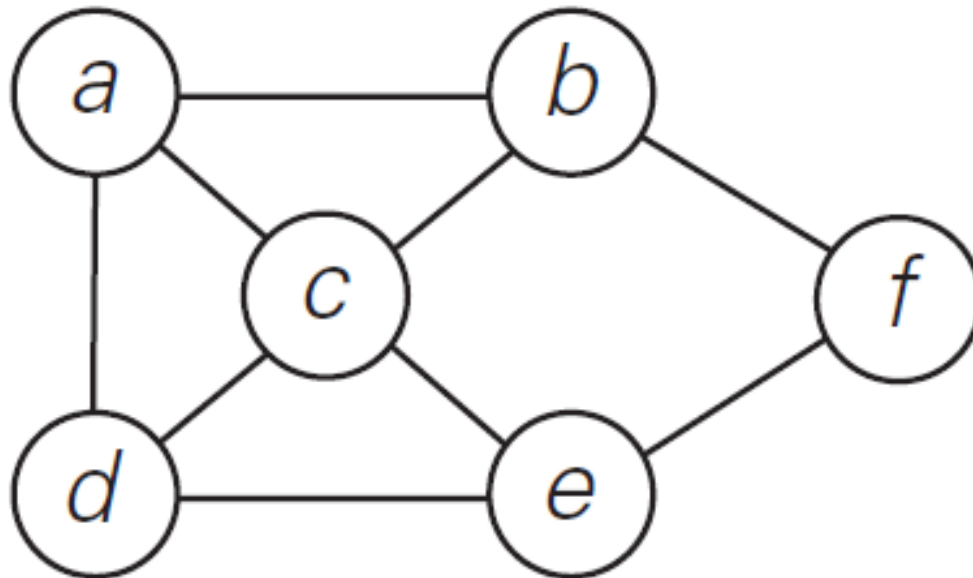
[Wikipedia]

Problema do Ciclo Hamiltoniano

- Dado um grafo grafo orientado $G(V,E)$, G tem um ciclo hamiltoniano ?
- Problema de Decisão
 - Resposta: SIM ou NÃO
- Problema NP-Completo
 - Veremos com o mais cuidado na próxima semana...
- Formulação simples, mas “difícil” de resolver, qualquer que seja o grafo
 - Contrastar com o Problema do Circuito Euleriano !!

Tarefa

- Este grafo tem um **Ciclo Hamiltoniano** ?
- **Como fazer** ?



[Levitin]

Procura Exaustiva

Procura Exaustiva

- Estratégia de força-bruta aplicada a **problemas combinatórios**
 - I.e., há um conjunto finito de soluções admissíveis
- Algoritmo
 - Enumerar **todas** as possíveis **soluções candidatas**
 - Verificar se cada uma **satisfaz** as restrições do problema
 - Se necessário, **escolher uma solução** do conjunto de soluções admissíveis
- Como assegurar que foram **verificadas todas as soluções candidatas** ?

Procura Exaustiva

- Algoritmo básico
 - $c \leftarrow$ gerar a primeira solução candidata
 - enquanto (c é candidata) faz
 - se (c é uma solução válida)
 - então imprimir (c)
 - $c \leftarrow$ gerar a próxima solução candidata, se existir
- Podemos parar após
 - Encontrar a primeira solução válida
 - Encontrar um dado número de soluções válidas
 - Testar um dado número de soluções candidatas
 - Gastar uma dada quantidade de tempo de CPU

Procura Exaustiva

- Características
 - Muitas vezes é simples de implementar
 - Irá **sempre** encontrar uma solução, caso exista (**?!?**)
- MAS, tempo proporcional ao número de soluções candidatas
 - **Explosão combinatória !**
 - Só praticável para instâncias “muito pequenas” **!!**
- Como tornar a procura **mais rápida** ?

Maior rapidez ?

- **Reduzir** a dimensão do espaço de procura
 - Usar análise / heurísticas para reduzir o número de soluções candidatas
 - E.g., o problema das n-damas
- **Reordenar** o espaço de procura
 - Útil quando procuramos uma só solução
 - O tempo de execução depende da ordem pela qual as soluções candidatas são testadas
 - Testar primeiro as soluções mais promissoras !!

O Problema do Caixeiro Viajante

– Travelling Salesperson Problem

O Problema do Caixeiro Viajante

- Determinar o **caminho mais curto** que atravessa **n cidades**
- MAS, visitando cada cidade uma só vez !
- E retornando à cidade inicial !
- Problema de **otimização** combinatória



[Wikipedia]

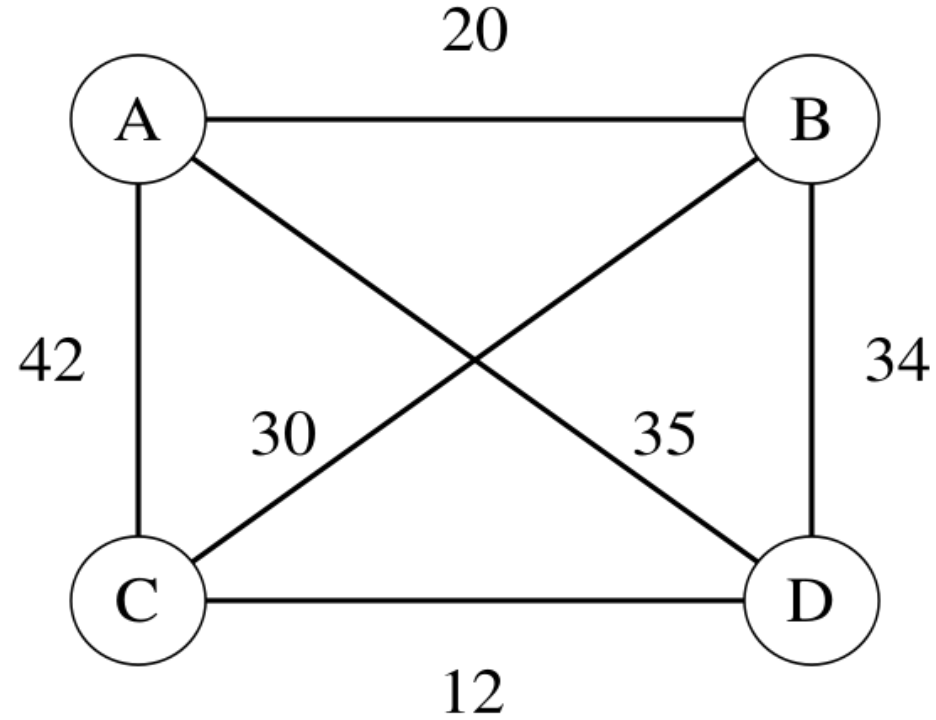
O Problema do Caixeiro Viajante

- Modelar o problema usando um grafo G com as distâncias entre cidades associadas às arestas
- Determinar o **ciclo Hamiltoniano mais curto** definido em G
 - Ciclo de menor custo / distância
 - Atravessa (uma só vez) cada um dos vértices
- Problema **NP-difícil** !!

O Problema do Caixeiro Viajante

- Ciclo Hamiltoniano
 - Sequência de $(n + 1)$ vertices adjacentes
 - O primeiro vértice é o ultimo vértice !
- Como fazer ?
 - Escolher um vértice qualquer como vértice inicial
 - Gerar as $(n - 1)!$ permutações possíveis dos vertices intermédios
 - Para cada um dos ciclos, calcular o seu custo / distância
 - E guardar o ciclo mais económico / mais curto

O Problema do Caixeiro Viajante



[Wikipedia]

- Qual é a **solução** ?

O Problema do Caixeiro Viajante

- Questões
 - Como armazenar o grafo ?
 - O grafo é **completo** ?
 - Como gerar todas as **permutações** ?
- Desempenho computacional
 - **$O(n!)$**
 - A procura exaustiva só pode ser aplicada a instâncias muito pequenas **!! Alternativas ?**
 - São possíveis pequenos melhoramentos

permutations.h

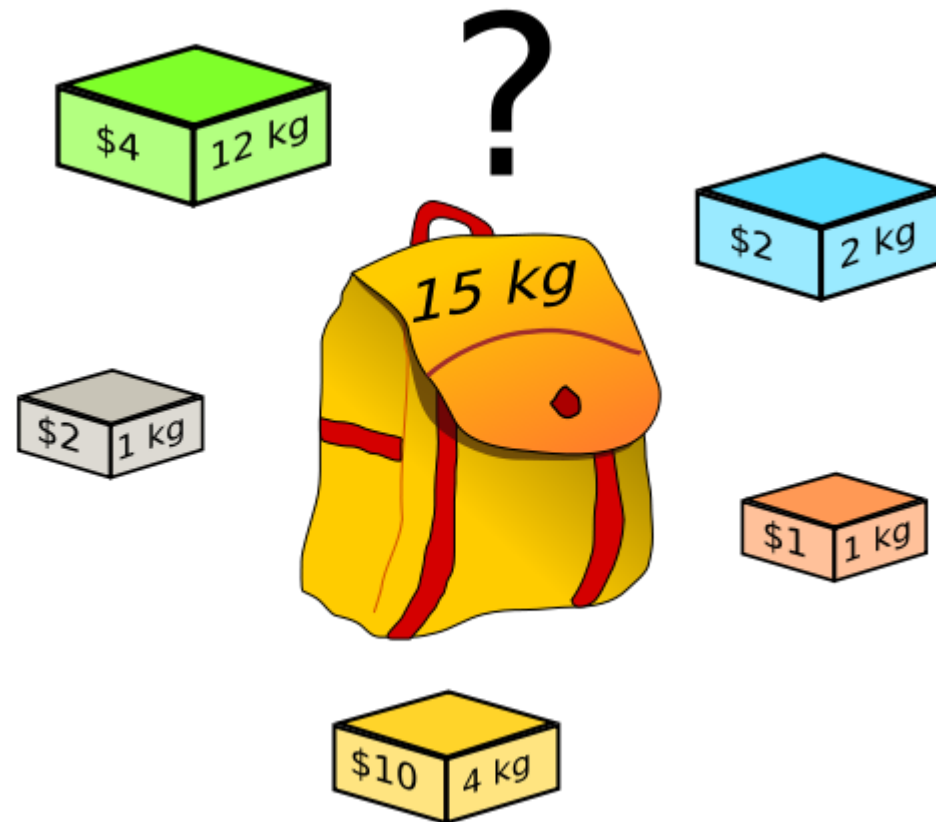
```
int* createFirstPermutation(int n);  
/* Cria o array de permutacoes com dimensao n, sendo a primeira permutacao  
 * 123456...n */  
  
void copyPermutation(int* original, int* copy, int n);  
/* Copia a permutacao actual */  
  
void destroyPermutation(int** p);  
/* Destroi o array de permutacoes */  
  
void printPermutation(int* p, int n);  
/* Imprime a permutacao actual */  
  
int nextPermutation(int* v, int n);  
/* Cria a permutacao seguinte */
```


O Problema da Mochila

- The 0-1 Knapsack Problem

O Problema da Mochila

- Determinar o **subconjunto mais valioso de itens**, que cabe na mochila



[Wikipedia]

O Problema da Mochila

- Dados **n itens**
 - Com **peso** w_1, w_2, \dots, w_n
 - Com **valor** v_1, v_2, \dots, v_n
- Uma mochila de **capacidade** W
- Qual é o (um) **subconjunto mais valioso de itens**, que cabe na mochila ?
- Problema **NP-difícil** !!

O Problema da Mochila

- Como formular ?

$$\text{max } \sum x_i v_i$$

sujeito a $\sum x_i w_i \leq W$

com $x_i \text{ in } \{0, 1\}$

O Problema da Mochila

- Como fazer ?
 - Gerar os 2^n subconjuntos do conjunto de n itens
 - Para cada um dos subconjuntos, calcular o seu peso total
 - Subconjunto admissível?
 - E guardar o subconjunto mais valioso

O Problema da Mochila

- Mochila de capacidade $W = 10$
- 4 itens
 - Item 1 : $w = 7$; $v = \$42$
 - Item 2 : $w = 3$; $v = \$12$
 - Item 3 : $w = 4$; $v = \$40$
 - Item 4 : $w = 5$; $v = \$25$
- Solução ótima ?

O Problema da Mochila

- Questões
 - Como gerar todos os subconjuntos ?
 - A ordem é importante ?
- Desempenho computacional
 - $O(2^n)$
 - A procura exaustiva só pode ser aplicada a instâncias muito pequenas !!
 - Alternativas ?
 - Soluções exatas vs. aproximadas

binarycounter.h

```
int* createBinCounter(int size);
/* Cria o contador binário com dimensão size, inicializado a zeros */

void copyBinCounter(int* original, int* copy, int size);
/* Copia o contador actual */

void destroyBinCounter(int** binCounter);
/* Destroi o contador */

void printBinCounter(int* binCounter, int size);
/* Imprime o contador */

int increaseBinCounter(int* binCounter, int size);
/* Incrementa o contador */
```


int* knapsackSearch(...)

```
int* knapsackSearch(float* weight, float* value, int n, float capacity) {  
    /* Gerar todos os sub-conjuntos dos indices do array de items */  
    /* Aproveitar a representacao binaria para os gerar ! */  
    /* Verificar, para cada um, o valor da soma dos pesos e dos valores */  
}
```

```
/* O numero de sub-conjuntos e 2^n */  
  
int numSubSets = (int)pow(2.0, n);  
  
/* Nao se testa o (sub-)conjunto vazio */  
  
int* binaryCounter = createBinCounter(n);  
  
int* currentBestSol = createBinCounter(n);
```

Iterar sobre os subconjuntos de itens

```
for (subSetIndex = 1; subSetIndex < numSubSets; subSetIndex++) {  
    sumWeights = 0;  
    sumValues = 0;  
  
    increaseBinCounter(binaryCounter, n);  
  
    for (i = 0; i < n; i++) {  
        if (binaryCounter[i] && ((sumWeights += weight[i]) > capacity)) {  
            break; /* Eficiencia --- Testar tambem sem este break !! */  
        }  
        if (binaryCounter[i]) {  
            sumValues += value[i];  
        }  
    }  
}
```

Conseguimos melhorar a solução corrente ?

```
if (sumValues > maxSumValues) {  
    maxSumValues = sumValues;  
  
    copyBinCounter(binaryCounter, currentBestSol, n);  
  
    /* Listar as sucessivas melhores solucoes */  
  
    solutionKnapsack(subSetIndex, weight, value, n, capacity, binaryCounter);  
}  
  
/* Poderia listar tambem eventuais solucoes alternativas !! */  
}
```

Soma de Subconjuntos

- The Sub-Set Sum Problem

O Problema da Soma de Subconjuntos

- Decision Problem
- Given a set A with n positive integer elements
- Given a positive integer value S
- Is there a subset of elements whose sum is equal to S ?
 - YES / NO
- Combinatorial problem
- NP-Complete !!

O Problema da Soma de Subconjuntos

- Um **problema de procura**
- Encontrar um **subconjunto** de um conjunto dado $A = \{a_1, \dots, a_n\}$ de **n números inteiros positivos**
- Cujas **soma** é igual a um dado **número inteiro positivo S**
- Exemplo
 - $A = \{1, 2, 5, 6, 8\}$ e **$S = 9$**
 - **Duas soluções** : $\{1, 2, 6\}$ e $\{1, 8\}$
- Outra instância
 - $A = \{3, 5, 6, 7\}$ e **$S = 15$**
 - Solução(ões) ?

void subsetSumSearch(...)

```
void subsetSumSearch(int* a, int size, int sum) {  
    // Gerar todos os sub-conjuntos dos indices do array  
    // Verificar, para cada um, o valor da soma dos elementos  
    // Aproveitar a representacao binaria para os gerar !  
}
```

```
// O numero de sub-conjuntos e 2^n  
  
int numSubSets = (int)pow(2.0, size);  
  
// Nao se testa o (sub-)conjunto vazio  
  
int* binaryCounter = createBinCounter(size);
```

Iterar sobre os subconjuntos de índices

```
for (subSetIndex = 1; subSetIndex < numSubSets; subSetIndex++) {  
    sumElements = 0;  
  
    increaseBinCounter(binaryCounter, size);  
  
    for (i = 0; i < size; i++) {  
        if (binaryCounter[i] && ((sumElements += a[i]) > sum)) {  
            break; /* Eficiencia --- Testar tambem sem este break !!*/  
        }  
    }  
  
    // Listar todas as solucoes encontradas  
  
    if (sumElements == sum) {  
        solutionFound(sum, a, size, binaryCounter);  
    }  
}
```


Quadrados Mágicos

Quadrado Mágico

- Matriz quadrada ($n \times n$)
- Com os elementos $1, 2, 3, \dots, n^2$
- A **soma** dos elementos de cada **linha** é igual à soma dos elementos de cada **coluna**
- É igual à soma dos elementos das duas **diagonais principais**

2	7	6	→15	
9	5	1	→15	
4	3	8	→15	
↙15	↓15	↓15	↓15	↘15

[Wikipedia]

Procurar

```
void magicSquaresSearch(int size) {  
    /* Gerar todas as permutacoes dos elementos do array */  
    /* Verificar, para cada uma, se se trata de um quadrado magico */  
  
    int sum;  
    int permutationIndex = 1;  
    int* p;  
    p = createFirstPermutation(size);  
  
    do {  
        if ((sum = isMagicSquare(p, size))) {  
            printf(" *** Permutation %d is a magic square of sum %d :\n\n",  
                permutationIndex, sum);  
            printMagicSquare(p, size);  
        }  
        permutationIndex++;  
    } while (nextPermutation(p, size));  
  
    destroyPermutation(&p);  
}
```

Válida ?

```
int isMagicSquare(int* a, int size) {  
    int i, j;  
    int sum;  
    int n = (int)sqrt(size);  
  
    int* sumRow = (int*)calloc(n, sizeof(int));  
    int* sumColumn = (int*)calloc(n, sizeof(int));  
    int* sumDiag = (int*)calloc(2, sizeof(int));
```

Válida ?

```
/* Summing the elements */  
  
for (i = 0; i < n; i++) {  
    for (j = 0; j < n; j++) {  
        sumRow[i] += a[j + i * n];  
        sumColumn[j] += a[j + i * n];  
  
        if (i == j) {  
            /* Main diagonal */  
            sumDiag[0] += a[j + i * n];  
        }  
  
        if ((i + j) == (n - 1)) {  
            /* The other diagonal */  
            sumDiag[1] += a[j + i * n];  
        }  
    }  
}
```

Válida ?

```
/* Checking the diagonals */  
  
if (sumDiag[0] != sumDiag[1]) {  
    free(sumRow);  
    free(sumColumn);  
    free(sumDiag);  
    return 0;  
}  
  
sum = sumDiag[0];  
  
/* Checking the rows */  
  
for (i = 0; i < n; i++) {  
    if (sumRow[i] != sum) {  
        free(sumRow);  
        free(sumColumn);  
        free(sumDiag);  
        return 0;  
    }  
}  
}
```

```
/* Checking the columns */  
  
for (i = 0; i < n; i++) {  
    if (sumColumn[i] != sum) {  
        free(sumRow);  
        free(sumColumn);  
        free(sumDiag);  
        return 0;  
    }  
}  
  
free(sumRow);  
free(sumColumn);  
free(sumDiag);  
  
return sum;  
}
```

Sugestão de Leitura

Sugestão de leitura

- A. Levitin, “*Design and Analysis of Algorithms*”, 3rd. Ed., Pearson, 2012
 - Chapter 3