# Procura Exaustiva

03/12/2025

# Ficheiro ZIP

- Está disponível no Moodle um ficheiro ZIP de suporte aos tópicos de hoje

- Módulo para gerar sucessivas permutações
- Módulo para gerar sucessivos subconjuntos

- Exemplos de aplicação da estratégia de procura exaustiva

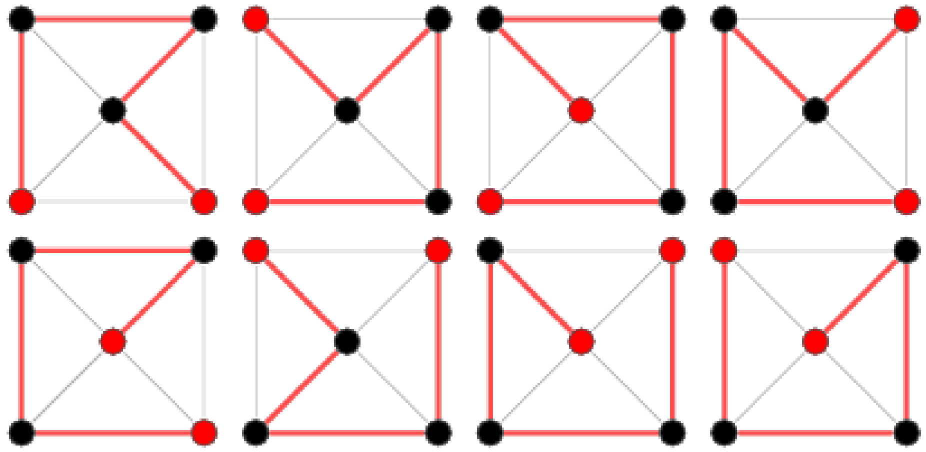
# Sumário

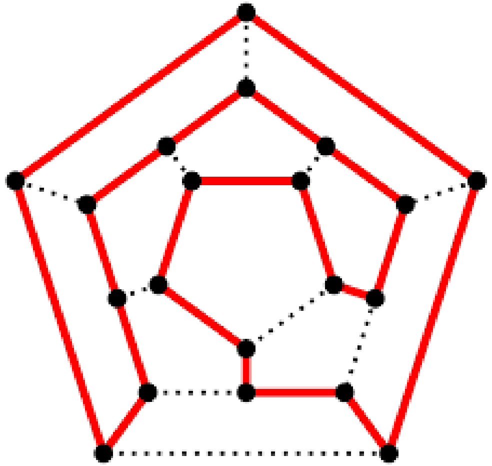
- Problemas Combinatórios
- Procura Exaustiva
- O Problema do Caixeiro Viajante (“The Traveling Salesman Problem”)
- O Problema da Soma de Subconjuntos (“The Subset Sum Problem”)
- O Problema da Mochila (“The 0-1 Knapsack Problem”)
- Geração de Quadrados Mágicos
- Sugestões de leitura

# Caminhos e Ciclos 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]

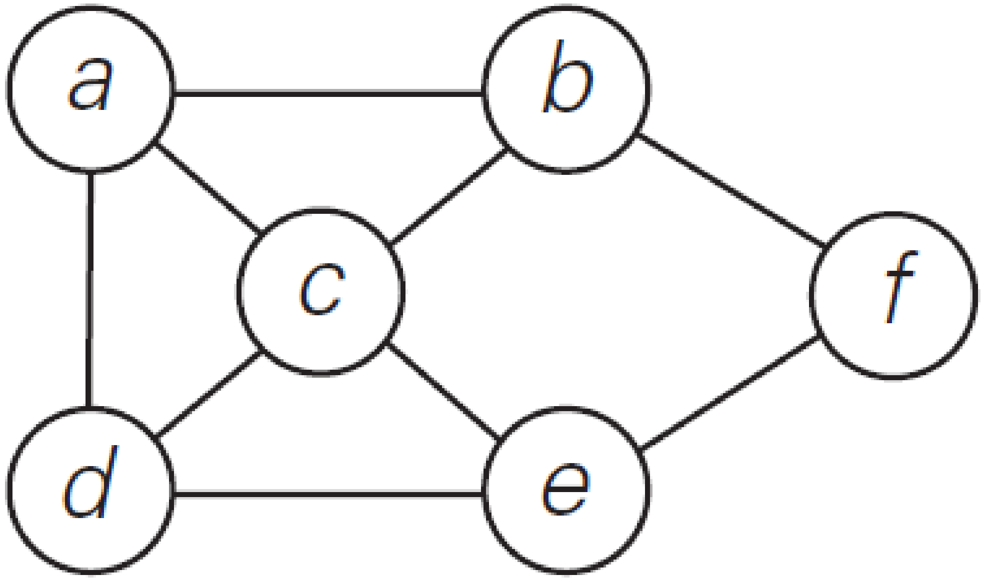

[Wikipedia]

# Problema do Ciclo Hamiltoniano

- Dado um grafo/grafo orientado G(V,E), G tem um Ciclo Hamiltoniano ?
- **Problema Combinatório de Decisão**
  - Resposta: SIM ou NÃO
- Formulação simples, mas "difícil" de resolver, qualquer que seja o grafo
  - Elevado esforço computacional, mesmo para instâncias não muito grandes !!
- Problema NP-Completo
  - Classe que contém alguns dos "famosos" e "difíceis" problemas de decisão

# Problema do Ciclo Hamiltoniano

- 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 candidatas 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 ← gerar a primeira solução candidata
enquanto ( c é candidata ) faz
  se ( c é uma solução válida )
  então imprimir (c)
  c ← gerar a próxima solução candidata, se existir

# Procura Exaustiva

- 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
  - No pior caso, implica processar todas as soluções candidatas

- **MAS,** tempo proporcional ao número de soluções candidatas
  - Explosão combinatória !
  - Só praticável para instâncias “muito pequenas” !!

- Como tornar a procura mais rápida ?

# Procura Exaustiva – Maior rapidez ?

- Reduzir a dimensão do espaço de procura
  - Usar análise/heurísticas para reduzir o número de soluções candidatas

- 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 – Traveling Salesperson Problem

# TSP – 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**
  - Conjunto finito de soluções candidatas
  - Determinar a (uma) solução ótima
  - Podem existir soluções ótimas alternativas


[Wikipedia]

# TSP – O Problema do Caixeiro Viajante

- Modelar o problema usando um grafo G com as distâncias entre cidades associadas às arestas

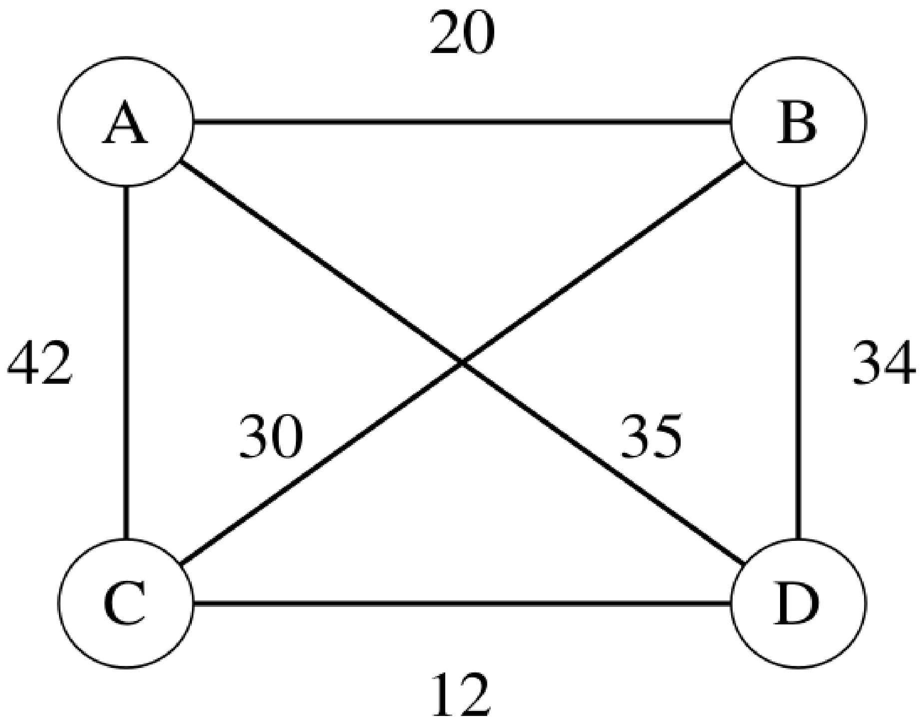
- Determinar o ciclo Hamiltoniano mais curto definido em G
  - Ciclo de menor custo / distância
  - Atravessa (uma só vez) cada um dos vértices

- Problema NP-difícil !!

# TSP – O Problema do Caixeiro Viajante

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

# TSP – O Problema do Caixeiro Viajante




[Wikipedia]

- Qual é a solução ?

# TSP – 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…

# permutation.h – Gerar permutações

```c
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 */
```

# Tarefa – Problema do Caixeiro Viajante

- Implementar o algoritmo de procura exaustiva

# Soma de Subconjuntos – The Subset Sum Problem

# O Problema da Soma de Subconjuntos

- **Problema de Decisão**
- Dado um conjunto A com n números inteiros positivos
- Dado um número inteiro positivo S
- Existe um subconjunto de elementos cuja soma seja igual a S ?
  - SIM / NÃO

- Problema combinatório
- NP-Completo !!

# O Problema da Soma de Subconjuntos

- **Problema de procura** – Qual é o subconjunto ?
- Encontrar, caso exista, um subconjunto de $A = \{a_1, \ldots, a_n\}$
- Cuja 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) ?

# binarycounter.h – Gerar subconjuntos

```c
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 */
```

# 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

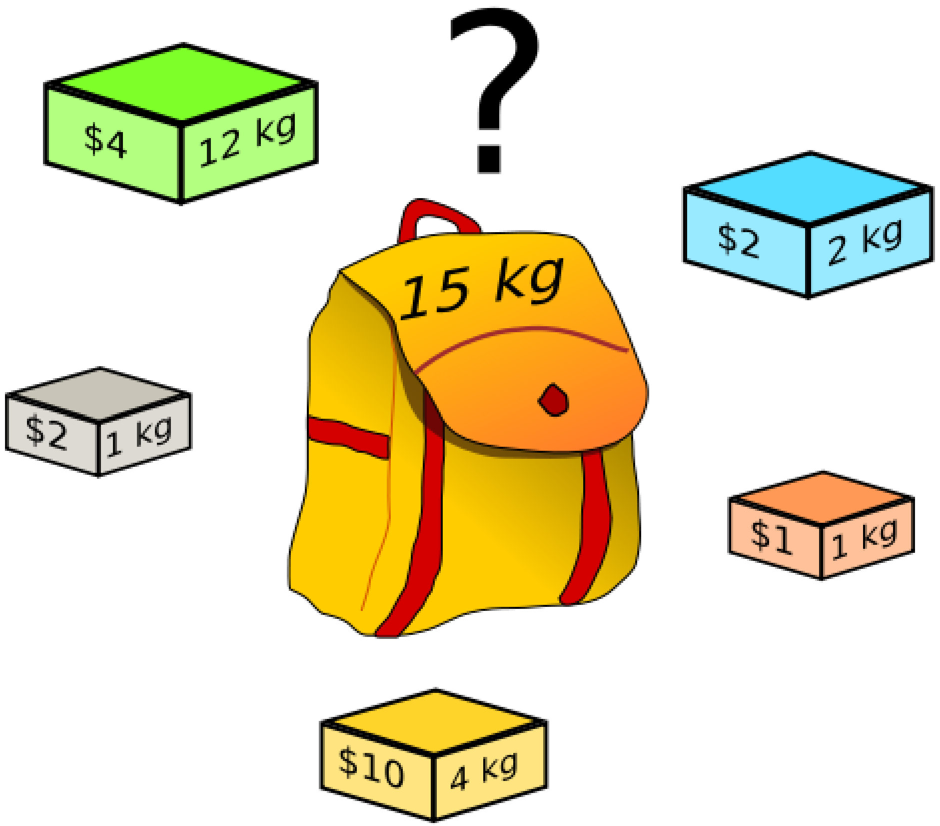
```c
for (subSetIndex = 1; subSetIndex < numSubSets; subSetIndex++) {
  sumElements = 0;

  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);
  }
}
```

- Gerar o próximo subconjunto
- Verificar a soma do valor dos seus elementos

# O Problema da Mochila
# – The 0-1 Knapsack Problem

# 0-1 Knapsack – O Problema da Mochila

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




[Wikipedia]

# 0-1 Knapsack – O Problema da Mochila

- Dados n itens
  - Com peso $\mathbf{w_1, w_2, \ldots, w_n}$
  - Com valor $\mathbf{v_1, v_2, \ldots, v_n}$
- Uma mochila de capacidade **W**

- Qual é o (um) subconjunto mais valioso de itens, que cabe na mochila ?

- **Problema de optimização combinatória**
  - NP-difícil !!

# 0-1 Knapsack – O Problema da Mochila

- Como formular ?

$$\max \sum x_i v_i$$

$$\text{sujeito a} \quad \sum x_i w_i \leq W$$

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

# 0-1 Knapsack – 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 / um subconjunto mais valioso que cabe na mochila

# 0-1 Knapsack – 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 ?

# 0-1 Knapsack – 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

# int* knapsackSearch(...)

```c
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 */
```

```c
/* 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);
```

- No início da procura, a "melhor solução" é o conjunto vazio

# 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];
    }
  }
```

- Gerar o próximo subconjunto
- Verificar se os itens cabem na mochila
- E qual é o seu valor total

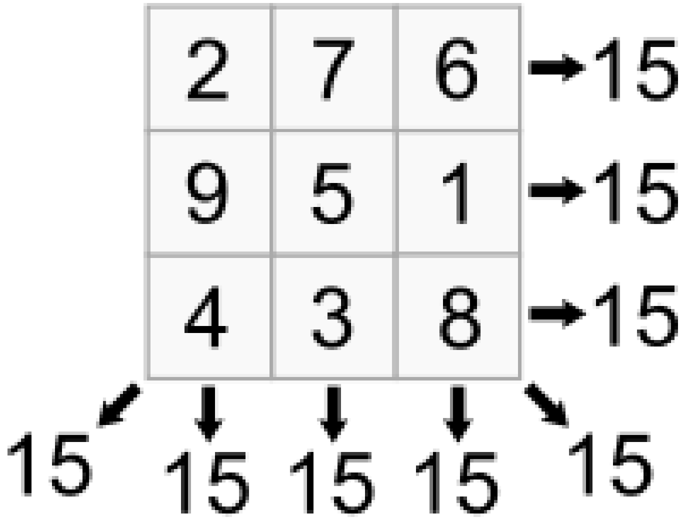
# Conseguimos melhorar a solução corrente ?

```
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 !! */
}
```

# Quadrados Mágicos

# Quadrado Mágico

- Matriz quadrada (n x n)
- Com os elementos 1, 2, 3, … , $n^2$
- A soma dos elementos de cada linha é igual à soma dos elementos de cada coluna
- E igual à soma dos elementos das duas diagonais principais




[Wikipedia]

- Quantos quadrados mágicos existem, para um dado n ?
- Representar os elementos da matriz num array 1D

# Quadrados Mágicos

- Como fazer ?
  - Gerar cada uma das permutações do conjunto de $n^2$ itens
  - Para cada permutação, verificar se satisfaz as restrições
    - Soma das linhas/colunas/diagonais principais
  - E imprimir cada permutação admissível, i.e., que define um quadrado mágico

# Procurar

- A primeira permutação
- Verificar se é um quadrado mágico
- Obter a próxima permutação

```c
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);
}
```

# Validar – É um quadrado mágico ?

```c
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));
```

- Para registar a soma de cada uma das linhas, das colunas e das duas diagonais principais

# Validar

```
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];
    }
  }
}
```

- Somar os elementos de cada uma das linhas e de cada uma das colunas
- Somar os elementos das duas diagonais principais

# Validar

- Verificar a igualdade
- Libertar a memória alocada

```c
/* 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;
  }
}
```

```c
  /* 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;
}
```

# Sugestões de Leitura

# Sugestões de leitura

- J. Murtagh, “*The Map Color Conundrum*”, in Scientific American, Dec. 2024

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