Problema do Caixeiro Viajante

e a Solução com Held-Karp

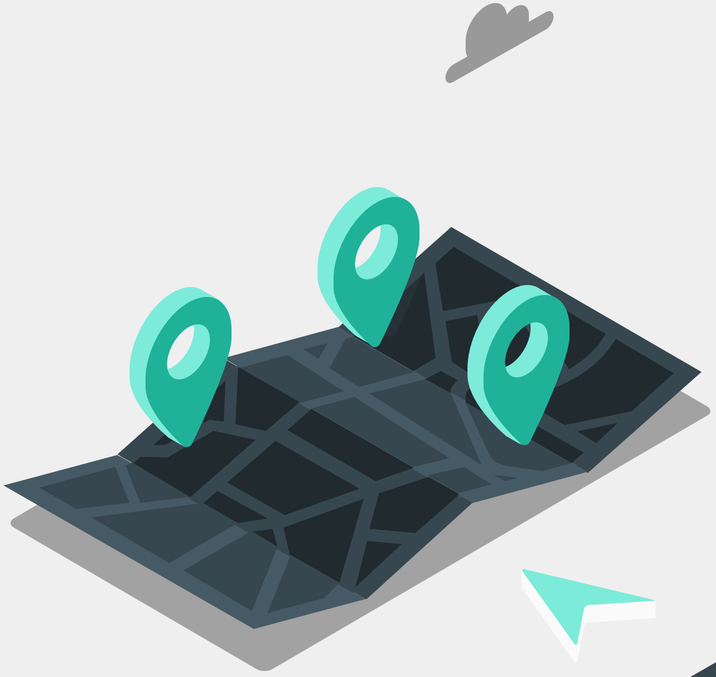
Docente: Dr. Leonardo Nogueira Matos
Discentes: Alisson Freitas & Erik Barbosa

AGENDA

1. DEFINIÇÃO
2. ENTRADA DO PROBLEMA
3. CLASSIFICAÇÃO
4. RESOLUÇÃO HELD-KARP

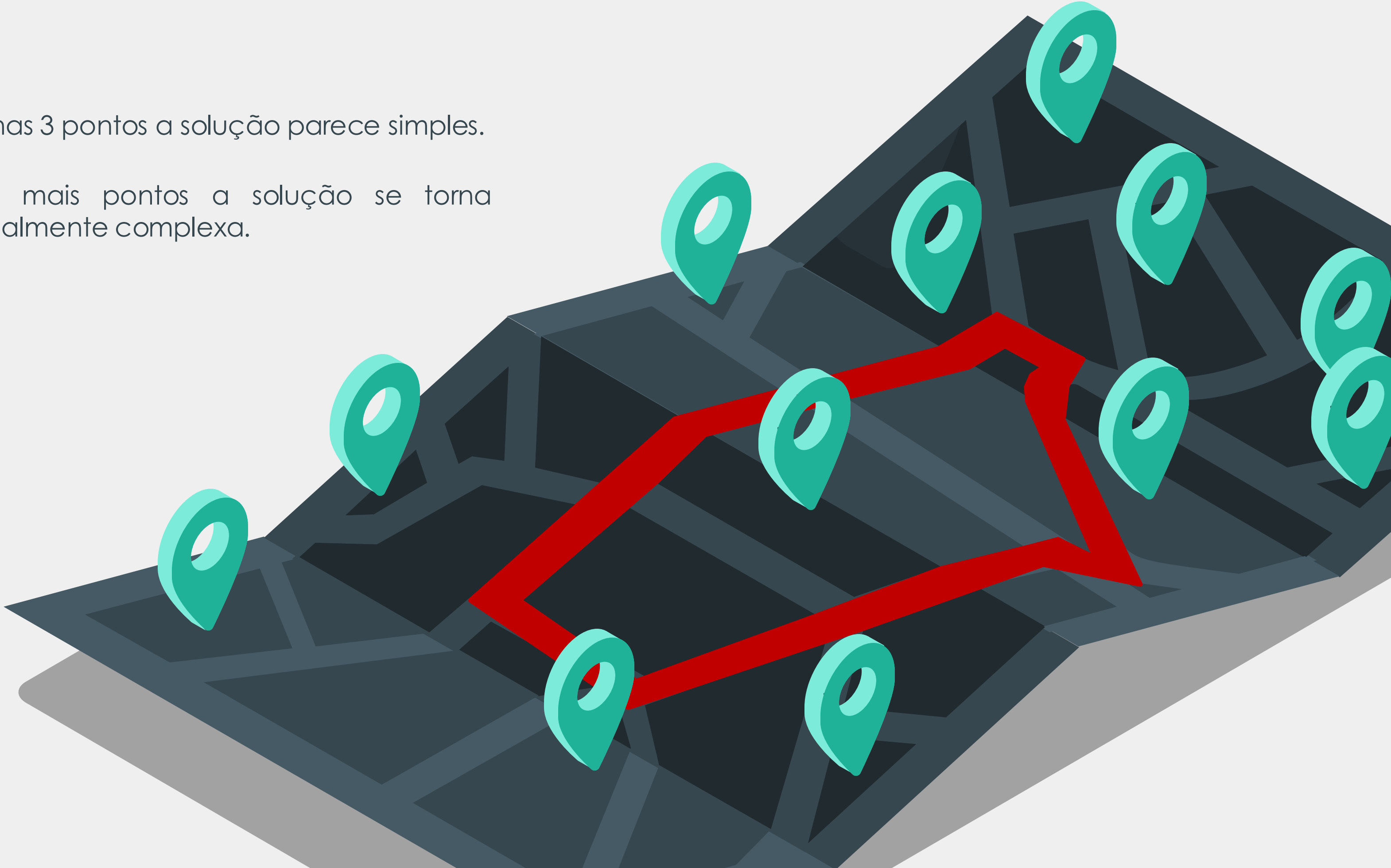
DEFINIÇÃO DO PROBLEMA

O Problema do Caixeiro-Viajante (PCV) tenta determinar a menor rota para percorrer uma série de cidades, visitando-as uma única vez.



Com apenas 3 pontos a solução parece simples.

Mas com mais pontos a solução se torna exponencialmente complexa.



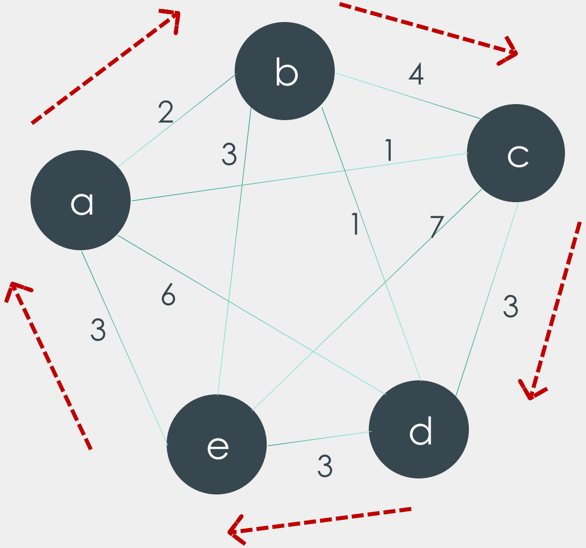
ENTRADA DO PROBLEMA

Utilizando a teoria dos grafos, cada cidade é identificada com um nó.

As rotas que ligam cada par de nós são identificadas como arestas.

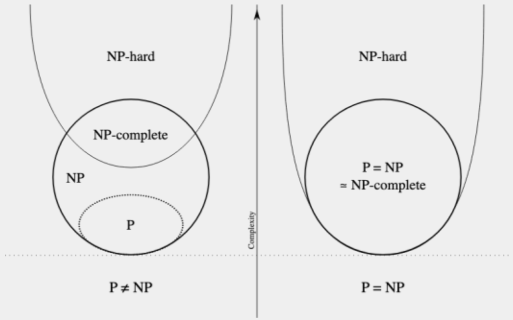
A cada uma destas arestas estarão associadas as distâncias (ou custos) correspondentes.

Uma viagem que passe por todas as cidades corresponde a um ciclo Hamiltoniano. A distância do ciclo é o somatório das distâncias das arestas presentes no mesmo.



Por exemplo, a distância do ciclo destacado com **setas vermelhas tracejadas C = {a;b;c;d;e}** possui custo (ou distância) de **15**

CLASSIFICAÇÃO DO PROBLEMA



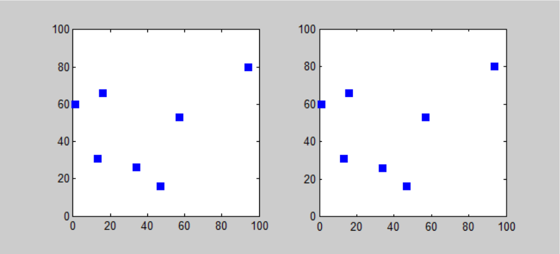
Problema de Decisão (Sim ou Não)

- A Pergunta: *"Existe uma rota com custo menor ou igual a K K?"*
- Classificação: NP-Completo
- Por que?
    - Está em NP: Dada uma solução, é fácil verificar a soma (O(n) $O(n)$).
    - É NP-Difícil: Tão complexo quanto qualquer outro problema chave (ex: SAT).

Problema de Otimização (Encontrar Valor)

- A Pergunta: *"Qual é a rota de menor custo absoluto?"*
- Classificação: NP-Hard (NP-Difícil)
- Por que?
    - Não é NP-Completo: A classe NP é restrita a perguntas de "Sim/Não".
    - Superset: É *pelo menos* tão difícil quanto a decisão. Se você resolve a otimização, resolve a decisão automaticamente.

SOLUÇÃO DE FORÇA BRUTA



Para determinar qual a solução ótima é necessário calcular todas combinações de ciclos possíveis. Resultando em uma em complexidade no tempo:

$$O((n-1)!)$$

Porém é possível afirmar que o custo de qualquer rota é igual ao inverso da mesma. Não é necessário calcular novamente o mesmo intervalo. resultando em uma em complexidade no tempo:

$$O\left(\frac{(n-1)!}{2}\right)$$

SOLUÇÃO DE HELD-KARP

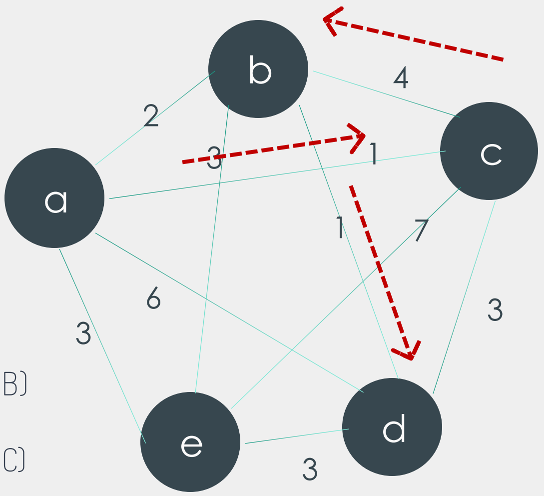
O algoritmo de Held-Karp é uma técnica utilizada para resolver o Problema do Caixeiro Viajante de forma exata. Em vez de tentar todas as permutações possíveis , **ele utiliza Programação Dinâmica para reduzir drasticamente o número de cálculos necessários.**

O PROBLEMA E A INTUIÇÃO

- **Held-Karp:** A ideia central é que **caminhos ótimos são compostos de subcaminhos ótimos.**

- **Exemplo intuitivo:** Imagine que você quer encontrar o menor caminho **visitando as cidades {A,B,C,D}** começando em A e terminando em D. Para chegar a D da melhor forma, você precisa ter vindo de B ou de C.

    - Se você veio de B, o custo é: (Melhor caminho visitando {A,C} terminando em B) + Distância B→D
    - Se você veio de C, o custo é: (Melhor caminho visitando {A,B} terminando em C) + Distância C→D



O algoritmo "memoriza" os melhores custos para subconjuntos menores, para não precisar recalculá-los.

O ALGORITMO – PASSO I : INICIALIZAÇÃO

- Entrada: Uma matriz de adjacência que representa os vértices que serão trafegados e o custo das arestas que os ligam, usaremos a seguinte entrada como exemplo:

```
# 0: A, 1: B, 2: C, 3: D
distancias = [
    [0, 10, 15, 20], # Distâncias de A
    [10, 0, 35, 25], # Distâncias de B
    [15, 35, 0, 30], # Distâncias de C
    [20, 25, 30, 0]  # Distâncias de D
]
```

- Saida: Uma tupla com custo mínimo representado por um inteiro e o caminho encontrado representado por uma lista de vértices

```
#(custo_minimo, caminho)
saida = (80, [0, 2, 3, 1, 0])
```

O ALGORITMO – PASSO I : INICIALIZAÇÃO

- **Passo 1:** Calculamos o custo de ir da **cidade 0 para cada cidade k** e armazenamos essa informação
- **Estrutura de Dados Usada:** Um HashMap (Dicionário) onde:
  - **Chave:** (bitMask, last_city)
  - **Valor:** (custo_acumulado, cidade_anterior)

```
# --- PASSO 1: Inicialização (Conjuntos de tamanho 2) ---
# Calculamos o custo de ir da cidade 0 para cada cidade k.
# A máscara (1 << k) | 1 significa: o bit k está ligado E o bit 0 está ligado.
for k in range(1, n):
    C[(1 << k) | 1, k] = (dists[0][k], 0)
```

```
C = {
# Chave: (Máscara em Binário, Última Cidade) : Valor: (Custo, Cidade Pai)

# Máscara 0011: Cidades {0, 1} visitadas. Termina em 1.
(0b0011, 1): (10, 0),

# Máscara 0101: Cidades {0, 2} visitadas. Termina em 2.
(0b0101, 2): (15, 0),

# Máscara 1001: Cidades {0, 3} visitadas. Termina em 3.
(0b1001, 3): (20, 0)
}
```

O ALGORITMO – PASSO 2 : CUSTO DAS DEMAIS COMBINAÇÕES

- Passo 2: Iterar sobre tamanhoss de subconjuntos crescentes. Começamos procurando caminhos que passam por 2 cidades intermediarias, depois 3, até N

```
for subset_size in range(2, n):
    # itertools.combinations gera todos os subconjuntos de cidades (excluindo a 0)
    for subset in itertools.combinations(range(1, n), subset_size):
        # Criar a bitmask para esse subconjunto
        bits = 0
        for bit in subset:
            bits |= 1 << bit

        # Adicionar a cidade inicial (0) à máscara
        bits |= 1

        # Para cada cidade 'k' neste subconjunto, qual é o menor custo para terminar nela?
        for k in subset:
            prev_mask = bits & ~(1 << k) # Remove k da máscara atual

            res = []
            # 'm' é a cidade visitada imediatamente antes de 'k'
            for m in subset:
                if m == 0 or m == k:
                    continue

                # Custo = Custo para chegar em m (com máscara anterior) + dist(m, k)
                res.append((C[(prev_mask, m)][0] + dists[m][k], m))

            # Armazena o mínimo e quem foi o pai (m) para reconstrução
            C[(bits, k)] = min(res)
```

Exemplo Prático (Trace)

Imagine 4 cidades: 0, 1, 2, 3. Estamos na iteração onde:

- **`subset`** = **`{1, 2}`** (Tamanho 2)
- Máscara **`bits`** representa **`{0, 1, 2}`**.

Caso A: Queremos terminar na cidade 2 (k=2)

Definir o Passado: Se terminamos em 2, removemos 2 do conjunto.

    - **`prev_mask`** representa **`{0, 1}`**.

Buscar o Pai (m): Quem sobrou no conjunto **`{1, 2}`** além do 2? Apenas o 1.

    - Então **m** só pode ser 1.

Calcular:

    - Olhamos na tabela **C**: Qual o custo de **`{0, 1}`** terminando em 1? (Digamos que seja 10).
    - Somamos a distância 1 para 2 (Digamos que seja 5).
    - Total = 15.

Salvar: **`C[({0,1,2}, 2)] = (15, 1)`**.

- Nota: itertools.combinations gera as combinações de cidades sem que precisemos escrever loops aninhados complexos. Se temos cidades {1, 2, 3}, ela gera (1, 2), (1, 3), (2, 3) quando pedimos tamanho 2.

O ALGORITMO – PASSO 3 : VOLTAR PARA O INICIO

- Passo 3: Agora temos o custo de visitar todas as cidades terminando em qualquer k. **Precisamos adicionar o custo de k para 0.**

```
all_bits = (1 << n) - 1
res = []
for k in range(1, n):
    res.append((C[(all_bits, k)][0] + dists[k][0], k))

opt_cost, parent = min(res)
```

Imagine que temos 3 cidades (0, 1, 2). A máscara cheia é **111 (7)**. A tabela **C** já tem:
Custo para visitar **{0, 1, 2}** terminando em **1: C[(7,1)]=50**
Custo para visitar **{0, 1, 2}** terminando em **2: C[(7,2)]=60**
Distâncias de retorno:

- **1→0 custa 10.**
- **2→0 custa 5.**

O Passo 3 faz:

- Testa terminando em 1: 50+10=60
- Testa terminando em 2: 60+5=65

Resultado: O mínimo é 60. O melhor caminho termina em 1 antes de voltar.

O ALGORITMO – PASSO 4 : RECONSTRUIR O CAMINHO

- Passo 4: Reconstruir o Caminho (Backtracking)

```python
path = []
# Começamos do fim para o começo, mas sabemos que o fim volta para 0
curr_bit = all_bits
curr_node = parent # O último nó antes de voltar para 0

# O ciclo termina em 0
path.append(0)

# Reconstrói de trás para frente
for i in range(n - 1):
    path.append(curr_node)
    new_bit = curr_bit & ~(1 << curr_node)
    _, curr_node = C[(curr_bit, curr_node)]
    curr_bit = new_bit

# Adiciona o ponto de partida
path.append(0)

# Inverte para mostrar na ordem correta (0 -> ... -> 0)
path.reverse()

return opt_cost, path
```

- Suponha 4 cidades. Caminho real: 0 → 1 → 2 → 3 → 0.
- Início (Passo 3): **`parent = 3`**. Máscara = **`1111`** (todas).
  - Lista: **`[0]`** (o zero final).
- Iteração 1 (Nó 3):
  - Adiciona 3 na lista. Lista: **`[0, 3]`**.
  - Remove 3 da máscara. Nova máscara: **`0111`** ({0, 1, 2}).
  - Consulta **`C[(1111, 3)]`**. O pai armazenado lá é 2.
  - Novo nó atual: 2.
- Iteração 2 (Nó 2):
  - Adiciona 2 na lista. Lista: **`[0, 3, 2]`**.
  - Remove 2 da máscara. Nova máscara: **`0011`** ({0, 1}).
  - Consulta **`C[(0111, 2)]`**. O pai armazenado lá é 1.
  - Novo nó atual: 1.
- Iteração 3 (Nó 1):
  - Adiciona 1 na lista. Lista: **`[0, 3, 2, 1]`**.
  - Remove 1 da máscara. Nova máscara: **`0001`** ({0}).
  - Consulta **`C[(0011, 1)]`**. O pai armazenado lá é 0.
  - Novo nó atual: 0.
- Fim do Loop.
  - Adiciona o 0 inicial. Lista: **`[0, 3, 2, 1, 0]`**.
  - Reverse: **`[0, 1, 2, 3, 0]`**.

O ALGORITMO – COMPLEXIDADE

Por que usar Held-Karp se ele ainda parece trabalhoso?

- Complexidade de Tempo: $O(n^2 2^n)$
- Complexidade de Espaço: $O(n2^n)$

Comparação :

- Para n=20 cidades:
  - Força Bruta (n!): 2.4×10^{18} operações (Bilhões de anos).
  - Held-Karp ($n^2 2^n$): Aprox. 4.2×10^8 operações (Segundos ou minutos).

REFERÊNCIAS

https://www.youtube.com/watch?v=-JjA4BLQyqE

https://www.youtube.com/watch?v=GiDsjIBOVoA

INTRODUCTION TO ALGORITHMS – Cormen,

Obrigado!