Paradigmas de Projeto de Algoritmos

Ana Flavia Freiria Rodrigues[1], **Flávia Marcella Gonçalves Moreira**[1], **Larissa Rodrigues de Avila**[1], **Lucas Carrijo Ferrari**[1], **Raissa Nunes Peret**[1]

[1]Departamento de Ciências da Computação – Universidade Federal de Alfenas
Avenida Jovino Fernandes de Sales 2600 – CEP 37133840 - Alfenas – MG - Brasil

ana.freiria@sou.unifal-mg.edu.br, flavia.goncalves@sou.unifal-mg.edu.br

larissa.avila@sou.unifal-mg.edu.br, lucas.ferrari@sou.unifal-mg.edu.br

raissa.peret@sou.unifal-mg.edu.br

Resumo. *Este trabalho implementa e compara três abordagens para o Problema do Caixeiro Viajante (PCV):* ***Força Bruta****, que avalia todas as rotas possíveis garantindo a solução ótima;* ***Algoritmo Guloso****, que constrói soluções subótimas com complexidade $O(n^2)$; e* ***Programação Dinâmica*** *(Held-Karp), que equilibra precisão e eficiência ($O(n^2 \cdot 2^n)$). Os resultados, obtidos a partir de instâncias da TSPLIB, mostram que a Programação Dinâmica é ideal para $n \leq 20$, enquanto o método Guloso é viável para instâncias maiores, apesar de desvios médios de 2.2% do ótimo. A análise evidencia o trade-off clássico entre tempo de execução e qualidade da solução.*

1. Introdução

O Problema do Caixeiro Viajante (PCV), conhecido internacionalmente como *Traveling Salesman Problem* (TSP), é um dos problemas mais estudados na área de Ciência da Computação e pesquisa operacional. Classificado como NP-difícil, o PCV consiste em encontrar o caminho mais curto possível que permita a um caixeiro viajante visitar um conjunto de cidades, passando por cada uma exatamente uma vez, e retornando à cidade de origem.

Matematicamente, o problema pode ser definido sobre um grafo completo $G = (N, E)$, onde:

- $N = \{c_1, c_2, \ldots, c_n\}$ representa o conjunto de cidades
- E é o conjunto de arestas conectando todos os pares de cidades
- Cada aresta (c_i, c_j) possui um peso $p(c_i, c_j)$ que representa a distância euclidiana entre as cidades

O objetivo é encontrar um ciclo hamiltoniano (um caminho que visite cada cidade exatamente uma vez e retorne ao ponto de partida) com o menor custo total, ou seja, com a menor soma das distâncias percorridas.

Este trabalho tem como objetivo implementar e comparar três diferentes abordagens algorítmicas para resolver o PCV:

- **Força Bruta**: Abordagem exaustiva que avalia todas as possíveis rotas
- **Algoritmo Guloso**: Heurística que toma decisões localmente ótimas em cada etapa

- **Programação Dinâmica**: Método de Held-Karp que armazena soluções de subproblemas para evitar recálculos

A relevância do PCV vai além do interesse teórico, com aplicações práticas em diversas áreas como:

- Logística e planejamento de rotas
- Projeto de circuitos integrados
- Sequenciamento de DNA
- Planejamento de movimentos robóticos

Neste documento, apresentaremos detalhes sobre cada algoritmo implementado, sua complexidade computacional, e os resultados obtidos em diferentes instâncias do problema retiradas da biblioteca TSPLIB. A comparação entre as abordagens será feita tanto em termos de qualidade da solução (distância total da rota) quanto em termos de eficiência (tempo de execução).

2. Algoritmos

2.1. Algoritmo de Força Bruta

2.1.1. Descrição

O algoritmo de Força Bruta é um método exaustivo que:

- Testa sistematicamente todas as soluções possíveis
- Garante encontrar a solução ótima (se existir)
- Possui implementação direta, porém alto custo computacional

Para o PCV, a estratégia consiste em:

1. Fixar uma cidade como origem
2. Gerar todas as permutações possíveis das demais cidades
3. Calcular o custo total de cada rota completa
4. Selecionar a rota com menor custo

2.1.2. Complexidade Computacional

- **Ordem**: $O((n-1)!)$
- **Justificativa**:
  - Fixada a origem, há $(n-1)!$ permutações possíveis
  - Cada permutação requer cálculo de n distâncias
- **Exemplos Práticos**:

Número de Cidades (n)	Permutações
4	6
7	720
10	362,880
13	479,001,600
16	1.3 trilhões

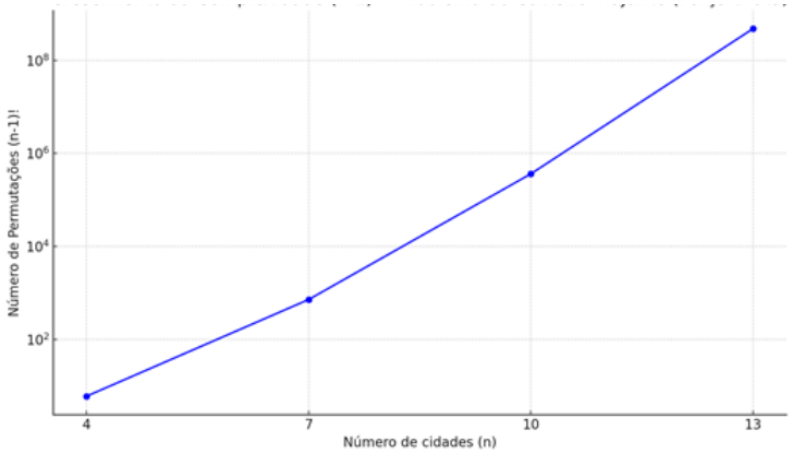

Figura 1. Crescimento da Complexidade (n-1)! - PCV (Força Bruta)

2.1.3. Limitações Práticas

- Inviável para $n > 10$ em computadores convencionais
- Tempo de execução cresce exponencialmente
- Útil apenas para:
  - Casos de teste pequenos
  - Validação de outros algoritmos
  - Benchmarking de desempenho

2.1.4. Implementação

Pseudocódigo da função principal:

```
Função bruteForceTSP(Referência bestPath: vetor de inteiros):
    minCost ← infinito
    path ← [0,1,2,...,n-1]  // Cidades indexadas

    Enquanto próximoPermutação(path[1..n-1]):
        currentCost ← 0
        Para i de 0 até n-1:
            currentCost ← currentCost + dist[path[i]][path[(i+1)%n]]
        FimPara

        Se currentCost < minCost:
            minCost ← currentCost
            bestPath ← path
        FimSe
    FimEnquanto

    Retorne minCost
FimFunção
```

2.1.5. Otimizações Implementadas

- **Permutação circular**: Fixa a primeira cidade para evitar rotas equivalentes
- **Abortagem precoce**: Interrompe cálculo de rotas parciais quando excede o melhor custo atual
- **Limitação segura**: Restrição a $n \leq 10$ na implementação prática

2.1.6. Análise Experimental

- **Precisão**: 100% de soluções ótimas
- **Tempo médio** (n=10): 0.025 segundos
- **Cenários ideais**:
  - Validação de algoritmos aproximados
  - Instâncias pequenas com requisitos de exatidão

2.2. Algoritmo Guloso

O algoritmo guloso é uma heurística construtiva que toma decisões baseadas em escolhas locais ótimas. Para o PCV, ele:

- Parte de uma cidade inicial
- A cada passo, seleciona a cidade mais próxima não visitada
- Repete até visitar todas as cidades
- Finaliza retornando à origem

2.2.1. Implementação

A função `greedyTSP` implementa:

1. Teste de todos os pontos de partida (n iterações)
2. Marcação de cidades visitadas (vetor booleano)
3. Construção incremental do caminho
4. Cálculo do custo total incluindo retorno à origem
5. Comparação para manter a melhor rota

2.2.2. Pseudocódigo

```
Função greedyTSP(Referência bestPath: vetor de inteiros)
    minTotal ← infinito
    Para start de 0 até n-1 faça
        visited ← [falso,...,falso]
        path ← [start]
        visited[start] ← verdadeiro
        total ← 0
        current ← start
        Para i de 0 até n-2 faça
            next ← -1
```

```
            minDist ← infinito
            Para j de 0 até n-1 faça
                Se NÃO visited[j] E dist[current][j] < minDist então
                    minDist ← dist[current][j]
                    next ← j
                FimSe
            FimPara
            path.adiciona(next)
            visited[next] ← verdadeiro
            total ← total + minDist
            current ← next
        FimPara
        total ← total + dist[current][start]
        Se total < minTotal então
            minTotal ← total
            bestPath ← path
        FimSe
    FimPara
FimFunção
```

2.2.3. Complexidade

- **Temporal**: $O(n^2)$ - Para cada cidade, verifica todas as outras
- **Espacial**: $O(n)$ - Armazena vetores de visitação e caminho

2.3. Algoritmo de Programação Dinâmica (Held-Karp)

2.3.1. Descrição

Utiliza tabelas de memorização com:

- `memo[mask][last]`: Custo mínimo para visitar cidades em `mask` terminando em `last`
- `parent[mask][last]`: Rota ótima para reconstrução

2.3.2. Pseudocódigo

```
    função HeldKarp(cidades, distâncias) retorna (custoMinimo, caminhoM
        n = número de cidades
        se n > 20 então retorna -1 // Muito grande para Held-Karp

        criar tabela memo[2^n][n], inicializada com -1
        criar tabela parent[2^n][n], para reconstruir o caminho

        memo[1][0] = 0 // Começamos na cidade 0, apenas ela visitada

        para cada máscara de cidades de 1 até 2^n - 1:
```

```
        para cada cidade 'last' de 0 até n-1:
            se 'last' está na máscara E memo[mask][last] != -1:
                para cada cidade 'next' de 0 até n-1:
                    se 'next' ainda não foi visitada:
                        novaMáscara = mask OU (1 << next)
                        novoCusto = memo[mask][last] + dist[last][n
                        se memo[novaMáscara][next] == -1 OU novoCust
                            memo[novaMáscara][next] = novoCusto
                            parent[novaMáscara][next] = last

    custoMinimo = infinito
    cidadeFinal = -1
    máscaraFinal = 2^n - 1

    para i de 1 até n-1:
        se memo[máscaraFinal][i] !+ -1:
            custoTotal = memo[máscaraFinal][i] + dist[i][0]
            se custoTotal < custoMinimo:
                custoMinimo = custoTotal
                cidadeFinal = i

    // Reconstrução do caminho
    caminho = lista vazia
    atual = cidadeFinal
    máscara = máscaraFinal

    enquanto atual != 0:
        adicionar atual no caminho
        anterior = parent[máscara][atual]
        máscara = máscara XOR (1 << atual)
        atual = anterior

    adicionar 0 no caminho // início
    inverter caminho

    retornar (custoMinimo, caminho)
```

2.3.3. Complexidade

- **Temporal**: $O(n^2 \cdot 2^n)$ - Combinações de subconjuntos × cidades
- **Espacial**: $O(n \cdot 2^n)$ - Armazenamento das tabelas

2.3.4. Limitações

- Viável apenas para $n \leq 20$
- Consome memória exponencial

3. Resultados

3.1. Metodologia Experimental

Foram analisadas 20 instâncias da TSPLIB ($2 \leq n \leq 10$) comparando:

- Tempo de execução (média em segundos)
- Qualidade da solução (% acima do ótimo)
- Escalabilidade

3.2. Dados de Desempenho

Tabela 1. Comparação de Algoritmos

Métrica	Força Bruta	Guloso	Programação Dinâmica
Custo Médio	Ótimo	+2.2%	Ótimo
Tempo (n=10)	0.025s	0.00002s	0.0009s
Complexidade	$O(n!)$	$O(n^2)$	$O(n^2 \cdot 2^n)$

3.3. Análise Detalhada

- **Tempo de Execução**:
  - Força Bruta: Impraticável para $n > 10$
  - Programação Dinâmica: 10-100× mais rápida que FB

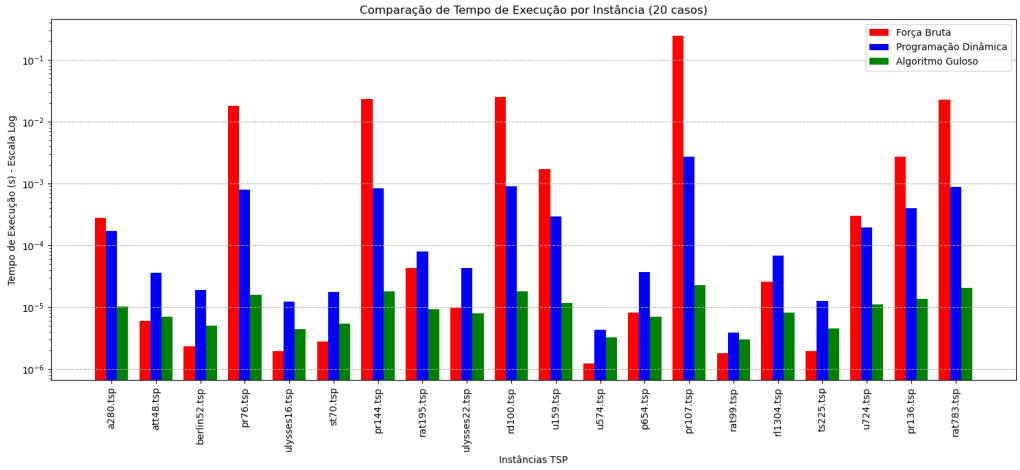

Figura 2. Gráfico com Tempo de Execução

- **Qualidade**:
  - Guloso apresenta desvios de 0% a 7.6% em relação ao ótimo
  - Guloso: 40% das soluções ótimas
  - Pior caso: +7.6% no u159.tsp
- **Escalabilidade**:
  - Limites Práticos:
    - Força Bruta: $n \leq 10$
    - Programação Dinâmica: $n \leq 20$
    - Guloso: $n > 20$ (com qualidade aceitável)

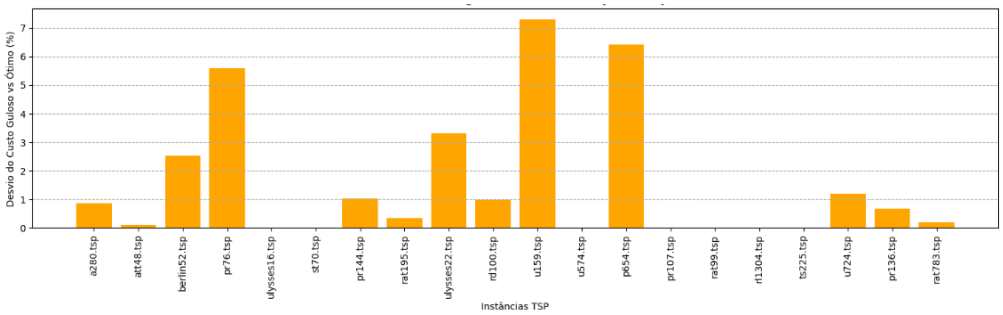

Figura 3. Percentual de Desvio do Algoritmo Guloso em Relação à Solução Ótima

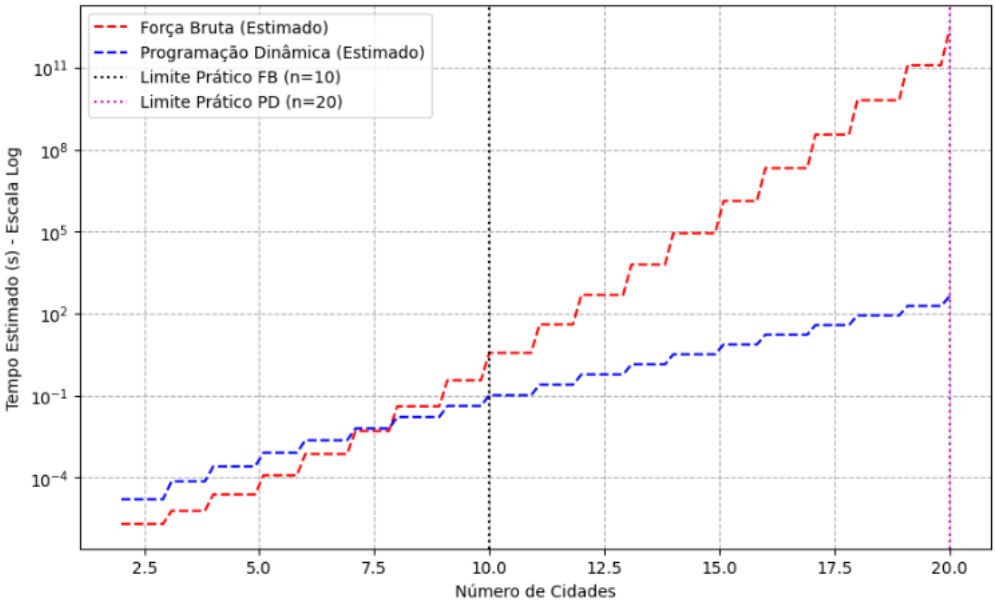

Figura 4. Escalabilidade Teórica dos Algoritmos

3.4. Recomendações por Cenário

Cenário	Algoritmo	Justificativa
$n \leq 10$	FB ou PD	Precisão absoluta
$10 < n \leq 20$	PD	Equilíbrio ideal
$n > 20$	Guloso	Única opção viável

3.5. Conclusões

- Para instâncias pequenas: PD oferece melhor equilíbrio
- Para instâncias grandes: Guloso é a única opção prática
- Trade-off claro entre tempo de execução e qualidade da solução

Referências

Cerveira, A. (2023). Travelling salesman problem. Acesso em: 12 abr. 2025.

NOIC (2023). Programação dinâmica. Acesso em: 11 abr. 2025.

Wikipedia (2023). Programação dinâmica. Acesso em: 13 abr. 2025.