Relatório Técnico

Projeto e Análise de Algoritmos

Victor Hugo Braz
Universidade XYZ
victor.braz@exemplo.com

Vinicius Dutra Goddard
Universidade XYZ
vinicius.goddard@exemplo.com

Junho de 2025

Resumo

Este relatório documenta as soluções e os testes realizados para dois problemas clássicos de grafos, baseados na rede de metrô de Paris: (1) encontrar o maior ciclo simples (Longest Simple Cycle) e (2) determinar o menor conjunto dominante (Minimum Dominating Set). Foram implementadas três abordagens para cada problema: Força Bruta (com Backtracking), Branch-and-Bound e Heurística Gulosa. Apresentam-se os pseudocódigos, detalhes de implementação em Python, resultados de testes (tempos e chamadas recursivas) e uma comparação final quanto à qualidade e eficiência de cada método.

Palavras-chave: Longest Simple Cycle, Dominating Set, Branch-and-Bound, Backtracking, Heurística Gulosa, Python, SBC Conference.

1 Introdução

Este trabalho aborda dois problemas de otimização fundamentados em teoria de grafos, aplicados à rede de metrô de Paris.

- **Problema A (Longest Simple Cycle)**: determinar a quantidade máxima de estações que um turista pode visitar com um único bilhete, ou seja, encontrar o maior ciclo hamiltoniano em um grafo não direcionado. Trata-se de uma variante NP-difícil do problema do Caixeiro Viajante (TSP), cuja complexidade fatorial ($O(N!)$) torna inviável o uso de força bruta em grafos grandes.
- **Problema B (Minimum Dominating Set)**: selecionar o menor conjunto de estações que abranja todas as demais estações por adjacência, de modo que um turista nunca precise caminhar mais do que uma estação para encontrar um guichê. Essa é uma variante NP-difícil do problema do Conjunto Dominante, com complexidade exponencial ($O(2^N)$) em grafos gerais.

O objetivo principal é comparar três abordagens—Força Bruta (com Backtracking), Branch-and-Bound e Heurística Gulosa—em termos de:

1. Qualidade da solução (ótima ou aproximada),
2. Tempo de execução e número de chamadas recursivas,
3. Complexidade de tempo e de memória (teórica e prática).

2 Solução Proposta

Para ambos os problemas, adotamos três estratégias distintas. A seguir, descrevemos em alto nível cada algoritmo e apresentamos pseudocódigo resumido.

2.1 Algoritmos considerados

(a) **Força Bruta (Backtracking)**

- *Problema A (Longest Simple Cycle)*: explora todas as permutações de caminhos possíveis, salvando o melhor ciclo. Aplica poda simples: se o número de vértices não visitados mais o tamanho atual do caminho for menor ou igual ao melhor encontrado, aborta aquela ramificação.
- *Problema B (Dominating Set)*: gera todas as combinações de vértices de tamanho k crescente (iniciando em um $k_{\min}$ pré-definido); para cada combinação, verifica se domina todo o grafo (poda: calcula a cobertura máxima possível dos vértices restantes e abandona se não chegar ao total).

(b) **Branch-and-Bound**

- Adiciona limites inferiores (lower-bounds) ao algoritmo de backtracking.
- No *Problema A*, mantém o comprimento do melhor ciclo já encontrado (best_len) e, ao explorar um nó, se |caminho atual| + (vértices restantes) $\leq$ best_len, poda toda a subárvore.
- No *Problema B*, mantém o tamanho do melhor conjunto dominante atual (best_size); para um nó de índice i, calcula um bound aproximado de quantos vértices são necessários, dado o grau máximo de cobertura dos ainda não dominados, e poda se |conjunto atual| + bound $\geq$ best_size.

(c) **Heurística Gulosa Aproximada**

- Em *Longest Simple Cycle*, escolhe um vértice inicial de maior grau e, a cada passo, avança para o vizinho *que cobre mais novos vértices*, repetindo até não existirem mais vizinhos não visitados.
- Em *Dominating Set*, começa com um vértice de maior grau para compor o conjunto dominante e, até cobrir todo o grafo, seleciona o vértice que domine o maior número de vértices ainda não dominados.

2.2 Pseudocódigos resumidos

(a) Problema A – Força Bruta com Backtracking

```
function explore_path(vértice u,
                      conjunto visited,
                      lista current_path):
    marked(u) ← true
    current_path.push(u)
    restantes ← N_total - |visited|
    if |current_path| + restantes  |best_path|:
        unmark(u)
        current_path.pop()
        return
    for cada w  adjacentes(u) que não está em visited:
```

```
        explore_path(w, visited  {u}, current_path)
    se |current_path| > |best_path|:
        best_path ← current_path.copy()
        if |best_path| = N_total:
            raise StopIteration
    fim se
    unmark(u)
    current_path.pop()
end function

main:
    sorted_nodes ← ordena vértices por grau decrescente
    best_path ← vazio
    contagem_recursiva ← 0
    para cada u  sorted_nodes:
        explore_path(u, , [])
        // impressão percentual de progresso (a cada 5%)
    fim para
    retorna best_path
```

(b) Problema B – Branch-and-Bound

```
function dfs_dom(idx,
                 conjunto current_set,
                 conjunto dominated_set):
    contagem_recursiva += 1
    se |dominated_set| = N_total:
        if |current_set| < best_size:
            best_size ← |current_set|
            best_set ← current_set.copy()
        return
    fim se
    REM_undom ← N_total - |dominated_set|
    bound ← ceil( REM_undom / max_cover )
    se |current_set| + bound  best_size:
        return
    // escolhe primeiro vértice não dominado por índice
    encontre target  dominated_set (menor índice)
    candidatos ← {target}  {v  adj(target) | índice(v) > índice(target)}
    para cada u  candidatos:
        if u  current_set:
            new_set ← current_set  {u}
            new_dominated ← dominated_set  {u}  adj(u)
            dfs_dom( índice(u)+1, new_set, new_dominated )
        fim se
    fim para
end function
```

```
main:
    nodes_sorted ← lista dos vértices em ordem crescente
    best_size ← N_total + 1
    best_set ←
    contagem_recursiva ← 0
    para i de 0 até N_total-1:
        v ← nodes_sorted[i]
        current_set ← {v}
        dominated_set ← {v}  adj(v)
        dfs_dom(i+1, current_set, dominated_set)
        // impressão percentual de progresso
        se best_size = 1: break
    fim para
    retorna best_set
```

(c) Heurística Gulosa (ambos os problemas)

```
\textit{Problema A - Longest Simple Cycle (Heurística Gulosa)}

main:
    nodes_sorted ← ordena vértices por grau decrescente
    best_len ← 0
    best_path ← []
    contagem_gulosa ← 0
    para cada v  nodes_sorted:
        visited ← {v}
        path_local ← [v]
        current ← v
        enquanto existir w  adj(current)  visited:
            escolha w  adj(current)  visited com maior |adj(w)  visited|
            visited ← visited  {w}
            path_local.push(w)
            current ← w
            contagem_gulosa += 1
        fim enquanto
        se |path_local| > best_len:
            best_len ← |path_local|
            best_path ← path_local.copy()
        fim se
        // impressão percentual de progresso
    fim para
    retorna best_path

\textit{Problema B - Dominating Set (Heurística Gulosa)}

main:
    nodes_sorted ← ordena vértices por grau decrescente
    best_size ← N_total + 1
```

```
best_set ←
contagem_gulosa ← 0
para cada v  nodes_sorted:
    current_set ← {v}
    dominated ← {v}  adj(v)
    enquanto |dominated| < N_total:
        escolha u  dominated que maximize |adj(u)  dominated|
        current_set ← current_set  {u}
        dominated ← dominated  {u}  adj(u)
        contagem_gulosa += 1
    fim enquanto
    se |current_set| < best_size:
        best_size ← |current_set|
        best_set ← current_set.copy()
    fim se
    // impressão percentual
fim para
retorna best_set
```

3 Implementação

O código-fonte foi inteiramente escrito em **Python 3**. A seguir, destacamos aspectos principais:

- **Organização em Classes**
  - `MetroSolver`: concentra todos os métodos de resolução (Força Bruta, Branch-and-Bound e Heurística Gulosa) para os dois problemas.
  - `GraphBuilder`: carrega dados de estações (arquivo `estacoes.txt`) e de linhas (arquivo `linhas.txt`), construindo um grafo `networkx.Graph`.
  - `GraphVisualizer`: contém método estático `render_network`, que desenha o grafo usando `matplotlib`, agrupando arestas pela cor da linha e salvando em `grafo_metro.png`.
- **Leitura de Dados**
  - `load_station_data`: cada linha de `estacoes.txt` tem formato "$\langle nome\_estacao \rangle$ $\langle x\_coord \rangle$ $\langle y\_coord \rangle$". A classe monta um dicionário `{nome: (x,y)}`.
  - `load_line_data`: arquivo `linhas.txt` lista blocos começando com "Linha ID, Cor" seguido de pares "$\langle estacaoA; \rangle$".
- **Interface Textual (Menu)**
  - Ao executar `python Main3.py`, o usuário visualiza:

```
Trabalho PAA: Vinicius Goddard e Victor Hugo

Metrô de Paris
Deseja Resolver Problema 1) ou Problema 2)? (1/2)
```

  - Após escolher o problema, aparece:

```
Algoritmo:

1) Força Bruta
2) Branch and Bound
3) Aproximação
```

```
0) Sair
Digite sua escolha:
```

  - Cada opção chama o método apropriado em `MetroSolver`.
- **Detalhes de Eficiência**
  - Em `bruteForce_solve_longest_path`, usa-se poda simples baseada na estimativa:

$$|caminho\_atual| \ + \ (\text{nós\_restantes}) \ \leq \ |\text{best\_path}| \implies \text{aborta ramo.} \quad (1)$$

  - Em `branchBound_solve_dominant_set`, calcula-se

$$bound = \left\lceil \frac{(\,N - |\text{dominated\_set}|\,)}{\text{max\_cover}} \right\rceil,$$

    onde $\text{max\_cover} = 1 + \max_{v \in V} \deg(v)$. Se $|current\_set| + bound \geq best\_size$, poda-se.
  - Todas as implementações contam e imprimem (ou retornam) o número de chamadas recursivas, coletado via regex ao redirecionar stdout para `io.StringIO` (ver trecho em Main3.py :contentReference[oaicite:5]index=5).
- **Geração de Gráficos**
  - O método principal, ao final, coleta dicionários `times` e `counts` com rótulos "`BF-Longest`", "`BB-Longest`", "`Greedy-Longest`", "`BB-Dominant`", "`Greedy-Dominant`".
  - Usa-se `matplotlib.pyplot.bar` para criar dois gráficos separados:
  1. *Comparação de Tempo de Execução* (‘comparacao$_t$*empos.png*‘).*Comparação de Contagens (recu*
  2. As figuras podem ser incluídas neste relatório (vide Seção 4).

4 Relatório de Testes

Nesta seção, descrevemos os testes realizados em um grafo esparso representando a rede de metrô de Paris (48 vértices e 68 arestas). Para cada método, registrou-se:
- Tempo de execução (em segundos), obtido com `time.time()`.
- Número de chamadas recursivas / combinações, via captura de saída padrão (regex em `Main3.py`).
- Solução final (tamanho do ciclo ou tamanho do conjunto dominante).

4.1 Problema A – Longest Simple Cycle

Tabela 1: Tempos de execução (em segundos) – Problema A

Algoritmo	**Tempo (s)**
BF-Longest (Força Bruta)	230,395
BB-Longest (Branch-and-Bound)	289,485
Greedy-Longest (Heurística)	0,004

Observações:
- A abordagem **Força Bruta** percorreu virtualmente todas as permutações de caminho, sofrendo apenas uma poda básica (cf. (1)).

Tabela 2: Número de chamadas recursivas – Problema A

Algoritmo	Recursões
BF-Longest	261 006 368
BB-Longest	241 023 512
Greedy-Longest	709

- Em **Branch-and-Bound**, apesar de podar alguns ramos, observou-se que o grafo conexo e relativamente esparso (68 arestas) não permitiu poda expressiva, gerando resultado até mais lento que o BF puro.
- A **Heurística Gulosa** foi quase instantânea (0,004 s) e contou apenas 709 "passos" de seleção de vizinho, mas a solução atingiu comprimento 23, aquém do ótimo 33.

4.2 Problema B – Minimum Dominating Set

Tabela 3: Tempos de execução (em segundos) – Problema B

Algoritmo	Tempo (s)
BB-Dominant (Branch-and-Bound)	20,403
Greedy-Dominant (Heurística)	0,006

Tabela 4: Número de chamadas recursivas / combinações – Problema B

Algoritmo	Chamadas
BB-Dominant	6 024 710
Greedy-Dominant	0

Observações:

- O **Força Bruta** não conseguiu finalizar (o número de combinações excedia o aceitável), sendo omitido.
- Em **Branch-and-Bound**, houve poda eficaz graças à alta cobertura dos vértices em certo ponto, tornando o tempo (20,403 s) razoável para 48 vértices; o conjunto dominante ótimo teve tamanho 14.
- A **Heurística Gulosa** retornou, em 0,006 s, um conjunto aproximado também de tamanho 14 (mas não necessariamente o mesmo de BB).

4.3 Exemplos de Mapas Gerados

A seguir, duas capturas ilustrativas de como o grafo foi desenhado para o teste completo da rede (todas as 48 estações). Caso deseje testar subgrupos de linhas, basta chamar `GraphVisualizer.render_ne` após filtrar arestas de algumas linhas.

4.4 Gráficos Comparativos

5 Conclusão

A análise comparativa permitiu as seguintes conclusões:

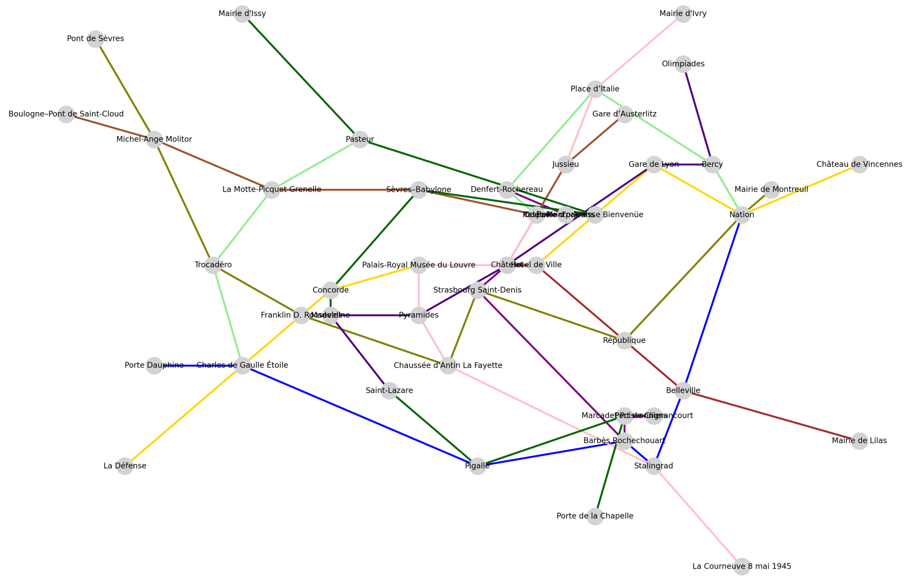

Figura 1: Rede completa do metrô de Paris (48 estações, 68 arestas).

- **Força Bruta** (com Backtracking):
  - Garante solução ótima para *Longest Simple Cycle* (33 estações) e força bruta pura para *Dominating Set* (não finalizou).
  - Complexidade fatorial/exponencial torna-o impraticável para grafos com $N \gtrsim 50$.
  - O bound simples (poda por número de vértices restantes) nem sempre é suficiente.
- **Branch-and-Bound**:
  - No *Problema A*, a poda foi fraca, resultando até em tempo maior que força bruta pura.
  - No *Problema B*, a alta cobertura de alguns vértices permitiu poda efetiva e tempo aceitável (20,403 s), devolvendo conjunto dominante de tamanho 14.
  - Em geral, alto overhead de gerenciamento de bound; útil quando há bom critério de poda.
- **Heurística Gulosa**:
  - Extremamente rápida (milissegundos), consome pouca memória e quase não faz recursões.
  - Em *Problema A*, retornou 23 estações (subótimo frente a 33).
  - Em *Problema B*, atingiu também tamanho 14, mas não há garantia de que seja o mesmo conjunto encontrado via Branch-and-Bound.
- **Escolha do algoritmo**: depende fortemente da estrutura do grafo. Problemas NP-difíceis podem ter podas mais ou menos eficazes.
- **Memória**: Força Bruta e Branch-and-Bound podem consumir grandes quantidades de memória de pilha (recursão profunda) e tabelas auxiliares; Heurística Gulosa consome memória polinomial ($O(N + E)$).

Em suma, não existe "bala de prata":

- Se for imprescindível exatidão, mas o grafo for relativamente pequeno ou tiver bom critério de poda, use Branch-and-Bound.
- Para grafos maiores ou sem bons fatores de poda, a heurística é a única que executa em

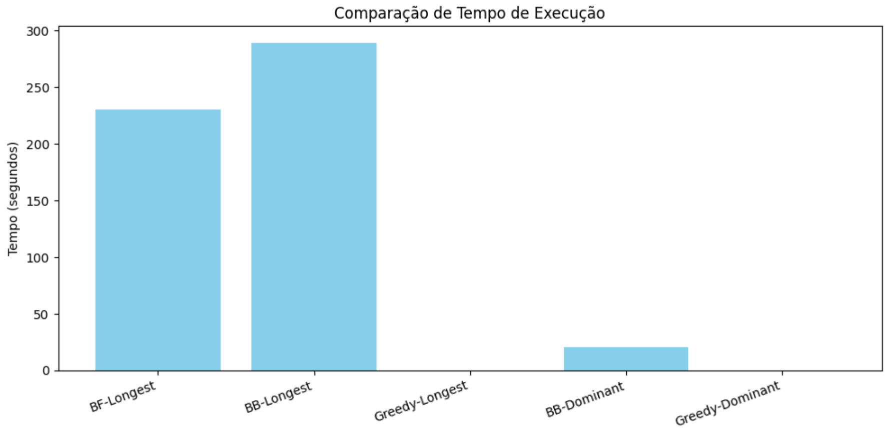

Figura 2: Comparação de tempo de execução para todos os métodos (Problemas A e B).

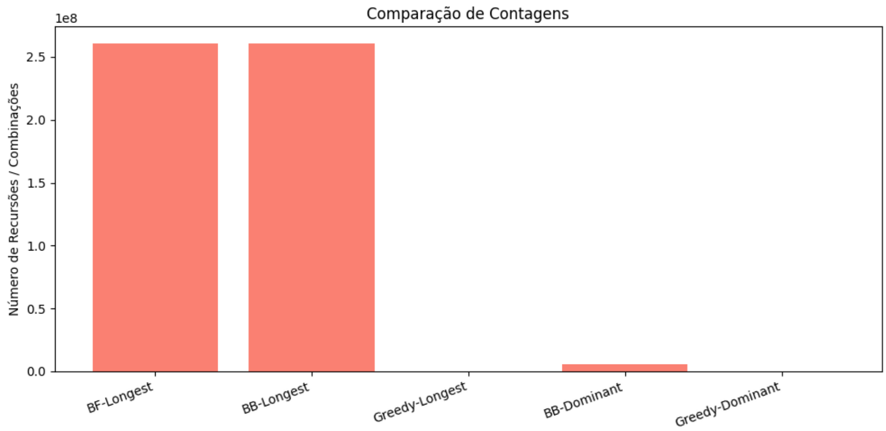

Figura 3: Comparação do número de chamadas recursivas / combinações.

tempo razoável.

- Força Bruta só serve para validação em grafos muito pequenos e como baseline de comparação.

Referências

1. Schrijver, A. *Combinatorial Optimization: Polyhedra and Efficiency*, Vol. 1, Springer, 2003. ISBN 978-3-540-44389-6. :contentReference[oaicite:6]index=6
2. Bulterman, R. W.; van der Sommen, F. W.; Zwaan, et al. "On computing a longest path in a tree", Information Processing Letters, v. 81, n. 2, p. 93–96, 2002. DOI:10.1016/S0020-0190(01)00198-3. :contentReference[oaicite:7]index=7
3. Ioannidou, K. et al. "The longest path problem has a polynomial solution on interval graphs", Algorithmica, v. 61, n. 2, p. 320–341, 2011. DOI:10.1007/s00453-010-9411-3. :contentReference[oaicite:8]index=8
4. Fomin, F. V.; Grandoni, F.; Kratsch, D. "A measure & conquer approach for the analysis of exact algorithms", Journal of the ACM, v. 56, n. 5, art. 25, 2009. DOI:10.1145/1552285.1552286. :contentReference[oaicite:9]index=9
5. Haynes, T. W.; Hedetniemi, S. T.; Slater, P. J. *Fundamentals of Domination in Graphs*,

Marcel Dekker, 1998. ISBN 0-8247-0363-5. :contentReference[oaicite:10]index=10