Análise de Algoritmos para Visitação Máxima e Cobertura de Guichês no Sistema Metroviário de Paris

Eduardo Araújo[1], **Henrique Lara**[1], **Luigi Louback**[1], **Thomas Neuenschwander**[1],

[1]Instituto de Ciências Exatas e Informática (ICEI)
Pontifícia Universidade Católica de Minas Gerais (PUC Minas) - Belo Horizonte, MG - Brasil

{eavsilva, henrique.lara, luigi.louback, tbaron}@sga.pucminas.br

Abstract. *This work addresses two challenges related to the Paris metro tourist pass: maximizing the stations visited in a cycle starting from a given origin and minimizing the number of stations with ticket booths while ensuring close access. We implemented exact methods such as brute-force and branch-and-bound, as well as fast heuristics, with a graphical interface for visualization. Tests assess the efficiency with various inputs, comparing both time and solution quality.*

Resumo. *Este projeto aborda dois problemas do passe turístico do metrô de Paris: encontrar o maior ciclo de estações a partir de uma origem e determinar o menor grupo de estações para guichês, assegurando proximidade. Desenvolvemos métodos exatos via força bruta e branch-and-bound, além de heurísticas polinomiais, exibidas em interface gráfica. Testes medem desempenho com diferentes dados, analisando tempo e resultados.*

1. Introdução

O metrô de Paris lançou um passe turístico que permite visitar várias estações, mas expira ao revisitar qualquer ponto. Este trabalho resolve dois desafios: (a) calcular o maior número de estações em um ciclo, iniciando e retornando ao mesmo local; e (b) identificar o menor grupo de estações para guichês, garantindo que nenhuma estação fique a mais de uma conexão de um guichê.

Métodos exatos e aproximados foram implementados para enfrentar esses problemas. A força bruta examina todas as possibilidades, enquanto o branch-and-bound reduz a busca com limites inferiores e superiores para o maior ciclo e o menor conjunto de guichês. Heurísticas, executadas em tempo polinomial, fornecem resultados rápidos, embora sem garantia de otimização. O código e a interface gráfica, desenvolvidos em Java, carregam arquivos de estações e linhas, desenham o grafo e destacam as respostas, auxiliando na análise.

Os testes utilizam dados baseados nas linhas sorteadas para o grupo ($N = 802717 \bmod 5$), incluindo apenas estações com duas ou mais linhas ou que sejam terminais, conforme o plano do metrô. O desempenho é medido por tempo de execução e qualidade das respostas, comparando abordagens exatas e aproximadas.

Este relatório técnico apresenta pseudocódigos, detalhes de implementação, resultados de testes e conclusões sobre a eficiência dos métodos aplicados. O objetivo é entregar um projeto funcional, bem documentado e com interface gráfica, enquanto se explora a complexidade de diferentes métodos algorítmicos.

2. Solução proposta

Para ambos os problemas, foram projetadas soluções baseadas nas três técnicas de projeto de algoritmos descritas anteriormente: força bruta, branch and bound e heurísticas.

Nesta seção, serão apresentados pseudocódigos que ilustram as principais ideias e passos das soluções implementadas. Eles foram escritos em inglês, seguindo o idioma utilizado no código-fonte para manter consistência.

As notações utilizadas nos pseudocódigos foram adaptadas para tornar a visualização dos elementos de cada problema mais clara e didática. Essas notações diferem dos nomes e estruturas usadas no código, pois o objetivo é priorizar a clareza conceitual.

2.1. Problema 1: Número máximo de estações a serem visitadas com um só passe

O problema consiste em determinar o número máximo de estações que um turista poderá visitar utilizando um único passe turístico, que permite que ele desembarque em uma estação e depois reembarque nela em direção a outra. No entanto, assim que retornar a uma estação já visitada, o passe perde a validade.

Para modelar essa situação, o sistema metroviário é representado como um grafo não direcionado, no qual os vértices correspondem às estações e as arestas representam conexões diretas entre elas. A tarefa se reduz a encontrar o ciclo de maior comprimento (em número de vértices) que começa e termina em uma estação raiz específica, sem repetir estações no caminho, exceto a de origem, como visto em (CACCETTA; VIJAYAN, 1991) e (AlgoMonster, 2025).

Nos pseudocódigos apresentados, são utilizadas as seguintes notações:

s: o vértice de origem para as buscas.

$u, v, s \in V(G)$: vértices pertencentes ao grafo

$\Gamma(v)$: vértices adjacentes ao vértice v

P: o caminho atual sendo explorado.

C: o maior ciclo encontrado.

max: $|C - s|$. Ou seja, a cardinalidade de C sem repetir s.

2.1.1. Solução utilizando Força Bruta

A técnica de força bruta para a solução desse problema consiste em explorar todas as possibilidade de caminhos partindo e terminando de um determinado vértice. Essa abordagem garante que, se uma solução para o problema existir, ela será encontrada, embora seu custo computacional possa ser proibitivo para grafos grandes e densos.

Essencialmente, o algoritmo constrói caminhos passo a passo, utilizando uma busca profunda. Se um caminho se fecha no vértice raiz e forma um ciclo válido e maior que o já registrado, ele é armazenado. Se em um caminho não é possível mais avançar (um vértice sem vizinhos para explorar), o algoritmo retrocede (backtracking), removendo o último vértice adicionado e tentando uma rota alternativa a partir do vértice anterior. Esse

processo continua até que todas as combinações possíveis de caminhos a partir da raíz tenham sido exaustivamente investigadas.

Algorithm 1 BruteForce(s)

```
max ← 0
C ← empty set
P ← {s}
BruteForceRecursive(P, s, s, max, C)
return max
```

Algorithm 2 BruteForceRecursive(P, s, v, max, C)

```
for each u in Γ(v) do
    if u = s and |P| ≥ 3 and |P| > max then
        max ← |P|
        C ← P ∪ {s}
    else if u ∉ P then
        P ← P ∪ {u}
        BruteForceRecursive(P, s, u, max, C)
        Remove u from P
    end if
end for
```

2.1.2. Solução utilizando Branch and Bound

A técnica de Branch and Bound aprimora a abordagem de força bruta ao incorporar estratégias de poda que reduzem significativamente o espaço de busca. Essencialmente, o algoritmo constrói caminhos passo a passo, utilizando uma busca em profundidade com backtracking. Quando um caminho se fecha no vértice raiz formando um ciclo válido maior que o previamente registrado, ele é armazenado como a melhor solução atual.

O primeiro mecanismo de poda utiliza um limite superior (upper bound) definido como $|V(G)|$, representando o número total de vértices no grafo. Esta escolha baseia-se na observação de que o maior ciclo possível seria um ciclo hamiltoniano. Embora este limite não seja particularmente restritivo, alternativas mais precisas exigiriam cálculos heurísticos ou aumentariam significativamente a complexidade computacional, neutralizando os benefícios da poda.

O segundo mecanismo, mais eficaz na prática, é a poda por grau do vértice. Vértices com grau menor que 2 não podem participar de ciclos, pois um ciclo requer que cada vértice tenha pelo menos duas arestas incidentes. Ao evitar a exploração de caminhos que incluem vértices de grau 1, o algoritmo elimina rapidamente porções substanciais do espaço de busca, mantendo a garantia de encontrar a solução ótima.

Algorithm 3 BranchAndBound(s)

```
max ← 0
C ← empty set
P ← {s}
BranchAndBoundRecursive(P, s, s, max, C)
return max
```

Algorithm 4 BranchAndBoundRecursive(P, s, v, max, C)

```
for each u in Γ(v) do
    if u = s and |P| ≥ 3 and |P| > max then
        max ← |P|
        C ← P ∪ {s}
    end if
    if u ∉ P then
        upperBound ← |V(G)|
        if upperBound > max and deg(u) > 1 then
            P ← P ∪ {u}
            BranchAndBoundRecursive(P, s, u, max, C)
            Remove u from P
        end if
    end if
end for
```

2.1.3. Solução utilizando Heurística

Inicialmente, para a solução do problema, foi proposta uma heurística gulosa, onde a cada passo, o vértice escolhido para expandir a busca era sempre o de maior grau dentre os vizinhos disponíveis. A idéia por trás dessa forma de pensar era de guiar a busca para as regiões do grafo que possuem maior numero de relaçoes, na expectativa de que essas regiões aumentassem as chances de formar ciclos com mais vértices. No entanto, durante os testes, verificou-se que essa estratégia apresentava resultados pouco satisfatórios, frequentemente encontrando ciclos curtos e distantes da solução ótima.

Diante dessas limitações, o algoritimo foi repensado, e chegou-se à conclusão que uma heurística estocástica que realizasse várias buscas aleatórias no grafo poderia ser uma estratégia que converteria em soluções mais aproximadas do que a proposta anterior.

A idéia por trás dessa dessa heurística aleatória é realizar k buscas imprevisíveis a partir do vétice raíz. Em cada busca, o algoritmo segue uma trajetória construindo um caminho até formar um ciclo, ou até não ser mais possível avançar. Para garantir que apenas vértices com potencial de contribuir para um ciclo sejam considerados, o algoritmo ignora vértices de grau 1. Ao final, o maior numero de vértice do ciclo encontrado entre todas as tentativas é retornado como a melhor solução aproximada para o problema.

Além disso, o número de buscas aleatórias K é definido de forma heurística como $k \leftarrow \lfloor \frac{|V|}{2} \rfloor$, com o objetivo de balancear o custo computacional e a qualidade das soluções geradas.

Algorithm 5 RandomizedHeuristicApproach(s)

```
max ← 0
C ← empty set
if |{u ∈ Γ(s) | deg(u) > 1}| < 2 then
    return C
end if
K ← ⌊|V|/2⌋
for i = 1 to K do
    P ← {s}
    v ← s
    foundCycle ← false
    while true do
        vOptions ← empty set
        for each u ∈ Γ(v) do
            if u = s and |P| ≥ 3 then
                if |P| > max then
                    max ← |P|
                    C ← P ∪ {s}
                end if
                foundCycle ← true
                break
            else if u ∉ P and deg(u) > 1 then
                vOptions ← vOptions ∪ {u}
            end if
        end for
        if foundCycle = true or vOptions = empty set then
            break
        end if
        u ← random vertex from vOptions
        P ← P ∪ {u}
        v ← u
    end while
end for
return C
```

Observa-se que se a verificacao inicial $|\{u \in \Gamma(s) \mid \deg(u) > 1\}| < 2$ é verdadeira, o vértice raiz s possui menos de dois vizinhos com grau maior que 1. Ou seja, ocorre quando não é possível realizar um caminho fechado (ciclo) a partir de s. No entanto, nota-se que essa condição não é suficiente para eliminar todos os casos inviáveis. Portanto, essa verificação inicial funciona apenas como uma filtragem preliminar e não substitui a necessidade de explorar caminhos adicionais para encontrar um ciclo.

2.2. Problema 2: Instalação de guichês para venda de passes

O segundo problema consiste em determinar as estações onde deverão ser instalados guichês de venda de passes, de forma que um turista nunca precise caminhar mais que

uma estação para encontrar um guichê. Isso assegura que o acesso aos guichês seja conveniente, minimizando o esforço de deslocamento.

A situação foi modelada como o problema do Conjunto Dominante Mínimo (Minimum Dominating Set - MDS) (GeeksforGeeks, 2024a) em um grafo não direcionado, onde os vértices representam as estações do metrô e as arestas indicam conexões diretas entre elas. O objetivo é encontrar o menor subconjunto de vértices (estações) tal que todos os vértices do grafo estejam contidos nesse conjunto ou sejam adjacentes a pelo menos um vértice do conjunto.

Para os pseudocódigos, considere as seguintes notações:

s: conjunto atual de vértices incluídos no conjunto dominante.

t: conjunto atual de vértices excluídos do conjunto dominante.

U: conjunto de vértices ainda indecisos, que podem ou não pertencer ao conjunto dominante.

$V(G)$: conjunto de todos os vértices do grafo.

$\Gamma(v)$: conjunto de vértices adjacentes ao vértice v.

MDS: o menor conjunto dominante encontrado até o momento.

$minSize$: o tamanho do menor conjunto dominante encontrado.

lb: limite inferior usado para poda no algoritmo branch and bound.

2.2.1. Solução utilizando Força Bruta

Inicialmente, para resolver o problema do conjunto dominante mínimo, adotamos um método de força bruta. Essa abordagem consiste em testar exaustivamente todos os subconjuntos possíveis dos vértices do grafo, buscando aquele de menor tamanho que seja um conjunto dominante válido.

Para isso, a implementação gera cada subconjunto possível dos vértices e verifica se ele é capaz de dominar o grafo. Na geração de todos os subconjuntos a abordagem vista em (GeeksforGeeks, 2024b) foi utilizada, visando gerar as combinações de forma mais rápida. A verificação é feita pela função `IsValidDominatingSet`, que garante que todo vértice do grafo esteja coberto, ou seja, que ele próprio ou algum de seus vizinhos faça parte do subconjunto, inspirado na abordagem do método naive em (GeeksforGeeks, 2024c).

Essa função é fundamental porque evita que subconjuntos inválidos sejam considerados soluções, permitindo filtrar rapidamente os candidatos durante a enumeração. O método de força bruta mantém então o subconjunto válido de menor tamanho encontrado até o momento.

Embora essa abordagem assegure a solução ótima, seu custo computacional cresce exponencialmente com o número de vértices, pois são gerados e testados todos os subconjuntos possíveis, o que a torna impraticável para grafos muito grandes.

Algorithm 6 BruteForce(V)

```
minSize ← ∞
MDS ← ∅
for each subset S ⊆ V do
    if |S| < minSize and IsValidDominatingSet(S) then
        minSize ← |S|
        MDS ← S
    end if
end for
return MDS
```

Algorithm 7 IsValidDominatingSet(S, G)

```
for each vertex v ∈ V(G) do
    if v ∉ S then
        hasNeighborInSet ← false
        for each neighbor u ∈ Γ(v) do
            if u ∈ S then
                hasNeighborInSet ← true
                break
            end if
        end for
        if hasNeighborInSet = false then
            return false
        end if
    end if
end for
return true
```

2.2.2. Solução utilizando Branch and Bound

A função recursiva `branchAndBound`, conforme detalhado no Algoritmo 8 gerencia três conjuntos de vértices: `currentInSet` (vértices atualmente incluídos no conjunto dominante parcial), `currentOutSet` (vértices explicitamente excluídos) e `undecidedVertices` (vértices ainda sem decisão). Possuindo como caso base se `currentInSet` forma um conjunto dominante completo, salvando a melhor solução.

A qualidade de um algoritmo de Branch and Bound está diretamente relacionada a qualidade de sua poda, calculada pela função `lowerBound` (detalhada no Algoritmo 9) para estimar o custo mínimo da solução que pode ser obtida a partir do estado atual. Se este limite inferior (lb) for maior ou igual ao `minSize` conhecido, o ramo atual da busca é descartado, pois não pode levar a uma solução melhor. Isso evita a exploração desnecessária de grande parte do espaço de soluções.

Se o ramo não for podado, o algoritmo seleciona o vértice com mais vizinhos dentre os `undecidedVertices` para a ramificação. Após a seleção, o algoritmo realiza duas chamadas recursivas: uma onde `nextVertex` é adicionado ao `currentInSet`

e outra onde é adicionado ao `currentOutSet`, explorando ambas as possibilidades. O menor resultado dessas duas ramificações é retornado.

A função `lowerBound` (Algoritmo 9) tem duas regras de poda. A primeira verifica se algum vértice em `notDominatedByCurrentSet` também está em `currentOutSet`; se tal vértice não puder ser coberto por nenhum vizinho em `undecidedVertices` (verificado usando *G.neighbors(v))*, a configuração atual é inviável. A segunda poda é realizada encontrando o potencial máximo de cobertura que um único vértice ainda não decidido tem.

A heurística para a escolha de vértices foi inspirada por (WANG et al., 2019), no qual o vértice de maior grau era selecionada para continuar a exploração, removendo-o do grafo em uma ramificação, e removendo seus vizinhos em outra. Embora o artigo trate outro problema, a heurística reduz o tempo médio de processamento para o problema do `Minimum Dominating Set`.

Algorithm 8 BranchAndBound(s, t, U)

```
if checkFullyDominated(s) then
    if |s| < minSize then
        minSize ← |s|
        MDS ← s
    end if
    return |s|
end if
if U = ∅ then
    return minSize
end if
lb ← lowerBound(s, t, U)
if lb ≥ minSize then
    return minSize
end if
v ← −1
maxDeg ← −1
for each u ∈ V(G) do
    deg ← |Γ[u] ∩ U|
    if deg < maxDeg then
        maxDeg ← deg
        v ← u
    end if
end for
U′ ← U \ {v}
s′ ← s ∪ {v}
r1 ← BranchAndBound(s′, t, U′)
t′ ← t ∪ {v}
r2 ← BranchAndBound(s, t′, U′)
return min(r1, r2)
```

Algorithm 9 LowerBound(s, t, U)

```
size ← |s|
ND ← ∅
for each v ∈ V(G) do
    if not isDominated(v, s) then
        ND ← ND ∪ {v}
    end if
end for
for each v ∈ ND do
    if v ∈ t then
        valid ← false
        for each u ∈ Γ(v) do
            if u ∈ U then
                valid ← true
                break
            end if
        end for
        if not valid then
            return ∞
        end if
    end if
end for
k ← 0
for each v ∈ U do
    dom ← Γ[v] ∩ ND
    if |dom| > k then
        k ← |dom|
    end if
end for
if ND = ∅ then
    return size
end if
if k = 0 then
    return ∞
end if
needed ← ⌈|ND|/k⌉
return size + needed
```

2.2.3. Solução utilizando Heurística

Optou-se pela heurística *Iterated Greedy* para resolver o problema do conjunto dominante mínimo, buscando um equilíbrio eficaz entre a qualidade das soluções e a eficiência computacional. A intenção foi encontrar um método capaz de escapar de mínimos locais e produzir soluções próximas do ótimo em tempo viável, evitando a complexidade e o custo computacional elevados de métodos exatos ou outras heurísticas que exigem muitos parâmetros ou são mais lentas.

A heurística *Iterated Greedy* inicia com uma solução construída de forma gulosa, selecionando vértices que cobrem o maior número possível de arestas ainda não dominadas. Como a solução inicial pode não ser ótima, o algoritmo realiza diversas iterações que alternam entre a remoção aleatória de vértices e a reconstrução da solução para restaurar a cobertura total das arestas.

Na fase de destruição, um conjunto fixo de vértices é removido aleatoriamente da solução atual. Essa remoção parcial gera uma solução temporariamente incompleta, forçando o algoritmo a sair da região do espaço de soluções onde estava preso, evitando que a busca fique ”presa” em mínimos locais — soluções subótimas que parecem ótimas localmente, mas não são globais.

Em seguida, a fase de reconstrução adiciona novos vértices seguindo um critério guloso: escolhe-se sempre o vértice que cobre o maior número de arestas ainda não dominadas pela solução parcial. Esse processo continua até que todas as arestas estejam cobertas novamente, garantindo a validade da solução.

A implementação segue essa lógica: parte-se de uma solução inicial gulosa, aplica-se a destruição aleatória e reconstrói-se a solução, atualizando a melhor resposta sempre que uma melhora é encontrada. O ciclo se repete até que o número máximo de iterações seja atingido.

Essa abordagem foi inspirada no trabalho de Casado et al. (2023) (CASADO et al., 2023), que aponta a complexidade NP-Difícil do problema e demonstra que a heurística consegue gerar soluções ótimas ou próximas do ótimo em tempo polinomial, mesmo em redes complexas. Os testes realizados comprovaram que a combinação de destruição aleatória e reconstrução gulosa ajuda a escapar de mínimos locais, produzindo resultados de alta qualidade de forma consistente.

Algorithm 10 IteratedGreedyApproach($V(G)$, $E(G)$, maxIterations, removalSize)

```
s ← ∅
uncoveredEdges ← E(G)
while uncoveredEdges ≠ ∅ do
    coverageCount(v) ← 0, ∀v ∈ V(G)
    for each edge (u, v) ∈ uncoveredEdges do
        coverageCount(u) ← coverageCount(u) + 1
        coverageCount(v) ← coverageCount(v) + 1
    end for
    bestVertex ← arg max_{v∈V(G)} coverageCount(v)
    s ← s ∪ {bestVertex}
    Remove edges covered by bestVertex from uncoveredEdges
end while
MDS ← s
for iteration ← 1 to maxIterations do
    partialS ← s
    for i ← 1 to removalSize do
        Remove a random vertex from partialS
    end for
    rebuiltS ← partialS
    uncovered ← ∅
    for each edge (u, v) ∈ E(G) do
        if u, v ∉ rebuiltS then
            uncovered ← uncovered ∪ {(u, v)}
        end if
    end for
    while uncovered ≠ ∅ do
        coverageCount(v) ← 0, ∀v ∈ V(G)
        for each edge (u, v) ∈ uncovered do
            coverageCount(u) ← coverageCount(u) + 1
            coverageCount(v) ← coverageCount(v) + 1
        end for
        bestVertex ← arg max_{v∈V(G)} coverageCount(v)
        rebuiltS ← rebuiltS ∪ {bestVertex}
        Remove edges covered by bestVertex from uncovered
    end while
    s ← rebuiltS
    if |s| < |MDS| then
        MDS ← s
    end if
end for
return MDS
```

3. Implementação

O projeto foi desenvolvido em Java 21 LTS. Para a representação do grafo, optou-se por uma lista de adjacência, implementada através da classe `AdjacencyList`. Visando

otimizar o desempenho computacional, os vértices, originalmente identificados pela classe `Station` (contendo informações em excesso como nome, cordenadas x e y) foram mapeados para inteiros. Dessa forma, a lista de `AdjacencyList` implementa um `Map<Integer, Set<Integer>>` para armazenar os vizionhos de cada vértice.

3.1. Representação do Grafo

A classe `AdjacencyList` contém métodos como:

- `neighbors(T vertex)`: retorna o conjunto de vértices adjacentes a um dado vértice
- `V()`: retorna o conjunto de todos os vértices do grafo
- `degree(T vertex)`: calcula o grau de um vértice especificado
- `addEdge(T v, T u)`: adiciona uma aresta entre dois vértices

A escolha de utilizar `Set` para armazenar os vizinhos de cada vértice foi estratégica para otimizar a complexidade computacional das operações críticas. Esta estrutura garante que as operações de verificação de adjacência (`contains`) e adição de vizinhos (`add`) sejam executadas em tempo constante $O(1)$ em média, em contraste com $O(N)$ que seria necessário ao usar listas. Além disso, o `Set` automaticamente elimina duplicações de arestas, mantendo a integridade do grafo não-direcionado sem necessidade de verificações adicionais.

Para o armazenamento do caminho atual durante as buscas, utilizamos `LinkedHashSet`, que mantém a ordem de inserção dos elementos enquanto, no caso médio, apresenta complexidade de tempo de busca de elemento igual a $O(1)$. Esta escolha foi fundamental para os algoritmos de busca de ciclos, onde é preciso tanto verificar rapidamente se um vértice já foi visitado quanto manter a sequência do caminho para reconstrução do ciclo.

3.2. Otimizações Algorítmicas

Diversas estratégias de otimização foram implementadas nos algoritmos para melhorar seu desempenho:

- **Podas:** Nos algoritmos de Branch and Bound, implementamos podas baseadas em limites superiores e inferiores calculados dinamicamente, além de podas por propriedades estruturais do grafo, como o grau dos vértices.
- **Estruturas de dados otimizadas:** Além dos conjuntos para verificação rápida de pertinência, utilizamos estruturas como `HashMap` para contagem de cobertura e `ArrayList` para manipulação eficiente de sequências.
- **Heurísticas adaptativas:** As abordagens heurísticas utilizam estratégias como seleção gulosa baseada em grau, mecanismos de backup para evitar becos sem saída, e técnicas estocásticas com número de iterações proporcional ao tamanho do grafo.
- **Verificações antecipadas:** Implementamos verificações preliminares de validade para evitar processamento desnecessário, como a função `isRootInvalid` que rapidamente identifica vértices raiz inadequados para iniciar buscas de ciclos.

Estas otimizações permitem que os algoritmos exatos (força bruta e branch and bound) encontrem soluções ótimas em tempo razoável para grafos de tamanho moderado, enquanto as heurísticas oferecem aproximações de qualidade mesmo para grafos maiores.

3.3. Interface Gráfica

Uma interface gráfica foi desenvolvida utilizando a biblioteca Swing para facilitar a interação com o sistema e visualizar os resultados. A interface permite:

- Carregar arquivos de estações e linhas do metrô
- Visualizar o grafo do sistema metroviário com as estações e conexões
- Selecionar e executar diferentes algoritmos para ambos os problemas
- Visualizar graficamente os resultados, com destaque para os ciclos encontrados ou os conjuntos dominantes

A classe `GUI` gerencia a interface, enquanto a classe interna `GraphPanel` é responsável pela renderização visual do grafo. Os resultados dos algoritmos são destacados no grafo, com vértices e arestas em vermelho indicando o ciclo máximo encontrado (para o Problema 1) ou os vértices selecionados para o conjunto dominante (para o Problema 2).

4. Relatório de Testes

Foram realizados testes para avaliar o desempenho dos algoritmos em diferentes grafos. Para cada problema, apresentamos as visualizações correspondentes às soluções obtidas pelos métodos: Força Bruta, Branch and Bound e Heurística.

Vale destacar que as soluções dos métodos exatos foram muito semelhantes visualmente em quase todos os casos, com poucos grafos apresentando diferenças mínimas. Por isso, optamos por não apresentar os mapas separados para ambos os métodos em todos os casos, evitando repetição. As imagens dos mapas estão incluídas junto aos resultados, mas foram posicionadas ao final do documento.

Foram propostos três cenários distintos de grafo para os testes: o grafo original (G3), com 49 vértices e 84 arestas, construído especificamente com base no sistema metroviário de Paris; e dois grafos artificiais menores, com 30 vértices e 45 arestas (G2), e com 20 vértices e 20 arestas (G1), utilizados para os testes em difentes escalas. Nos grafos utilizados para o Problema 1, a estação Nation foi definida como vértice raiz para as buscas, representando o ponto de partida e retorno nos ciclos.

Table 1. Tempos de execução (milissegundos) para os algoritmos testados

Grafo	Força Bruta	Branch and Bound	Heurística
Problema 1 (P1) - Maior ciclo			
G3	1220 ms	1067 ms	6 ms
G2	38 ms	5 ms	3 ms
G1	4 ms	3 ms	3 ms
Problema 2 (P2) - Conjunto dominante mínimo			
G3	Indefinido	3372 ms	15 ms
G2	163590 ms	21 ms	11 ms
G1	136 ms	8 ms	7 ms

Table 2. Tamanho dos ciclos retornados na solução (Problema 1)

Grafo	Força Bruta	Branch and Bound	Heurística
G3	32 vértices	32 vértices	23 vértices
G2	30 vértices	30 vértices	Indefinido
G1	16 vértices	16 vértices	15 vértices

Table 3. Número de vértices retornados na solução (Problema 2)

Grafo	Força Bruta	Branch and Bound	Heurística
G3	-	16 vértices	23 vértices
G2	9 vértices	9 vértices	17 vértices
G1	6 vértices	6 vértices	9 vértices

No Problema 2, o algoritmo de Força Bruta para o grafo original não finalizou em tempo viável, mas é certo que retornasse os mesmos 32 vértices do Branch and Bound. Já no Problema 1 com 30 vértices, a heurística estocástica não encontrou um ciclo maior, o que pode ocorrer devido à sua natureza não determinística, embora mantenha ótimo desempenho em tempo.

5. Conclusão

Este trabalho apresentou, implementou e comparou diferentes abordagens para a resolução dos problemas propostos no contexto do sistema metroviário de Paris: a maximização do número de estações visitadas em um ciclo (Problema 1) e a minimização do número de guichês necessários para cobertura adequada (Problema 2).

5.1. Análise de Complexidade

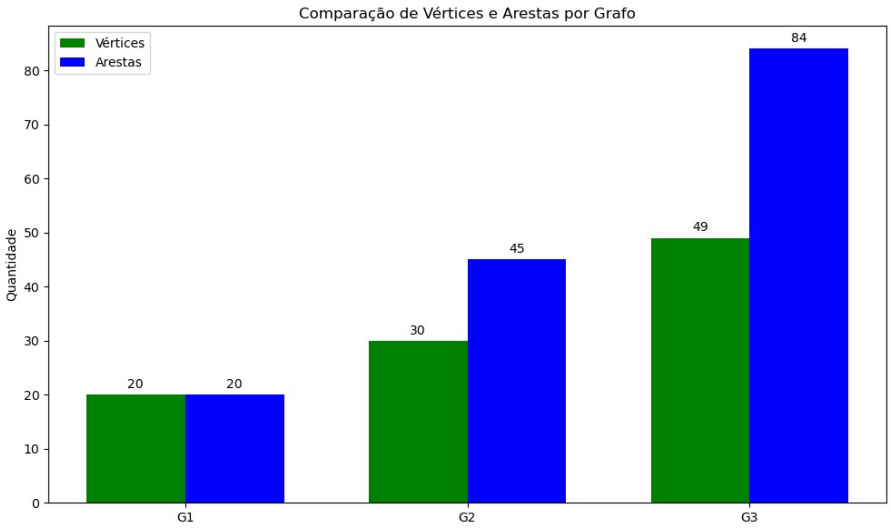
Para complementar os resultados obtidos pelas metodologias empregadas, segue abaixo a análise assintótica de tempo e memória de cada abordagem desenvolvida para os problemas em questão. Essa avaliação oferece uma perpectiva quantitativa sobre a ordem de grandeza de recusos computacionais gastos em variação do tamnho das instâncias fornecidas.

5.1.1. Problema 1: Força Bruta

A complexidade de tempo desse algoritmo é determinada pelo número de caminhos que ele explora. Para o primeiro vértice do caminho há no máximo $N - 1$ escolhas. Para o segundo, $N - 2$ escolhas (pois não se pode repetir os vértices já visitados), e assim por diante. Essa natureza exploratória leva à geração de uma quantidade de caminhos que cresce de forma fatorial com o número de vértices. Para cada caminho explorado, ao se atingir um vértice v, o algoritmo itera sobre seus vizinhos para tentar estender o caminho ou fechar um ciclo. No pior caso, um vértice pode ter até N - 1 vizinhos. Combinando o número de caminhos possíveis com o processamento em cada etapa, a complexidade de tempo do algoritmo é da ordem de $O(N!N)$ ou, simplificando para $O(N!)$.

A complexidade de memória refere-se ao espaço adicional utilizado pelo algoritmo, além da representação do grafo em si. Os principais componentes que consomem

Figure 1. Comparação de vértices e arestas dos três grafos testados



memória são: a lista C, que armazena o maior ciclo encontrado e pode conter até N vértices; o conjunto P, que armazena o caminho atual sendo explorado, também podendo conter até N vértices; e a pilha de recursão do sistema, cuja profundidade máxima pode atingir N (no caso de um caminho que percorra todos os vértices). Assim, a complexidade de memória auxiliar do algoritmo é $O(N)$.

5.1.2. Problema 1: Branch and Bound

A eficácia da poda por branch and bound é altamente dependente da estrutura do grafo e da qualidade das regras de poda. A condição $deg(u) > 1$ pode reduzir significativamente o espaço de busca em grafos com muitos vértices de grau 1. No entanto, em grafos mais densos, esta condição de poda raramente será verdadeira. Nestes cenários, o número de caminhos explorados pelo algoritmo de branch and bound ainda pode ser da ordem de N!. Portanto, embora frequentemente mais rápido na prática do que a força bruta, a complexidade de tempo no pior caso do algoritmo de branch and bound implementado permanece O(N!).

Quanto a complexidade de memória, o Branch and Bound possuí as mesmas condições que o de Força Bruta, permanecendo $O(N)$.

5.1.3. Problema 1: Heurístico

A abordagem heurística contrói k soluções candidatas independentes através de buscas saindo do vértice de origem. Em uma iteração, a cada vértice adicionado ao caminho seus vizinhos são examinados, podendo, no pior caso, gastar $O(N)$ de tempo para inspecionar-los. Para operações auxiliares como verificações de pertencimento ao caminho e adições a listas de candidatos são $O(1)$. Assim, a etapa de estender o caminho por um vértice consome $O(N)$ tempo, levando a um custo de $O(N^2)$ para construir um caminho completo em uma tentativa. Multiplicando pelo número de tentativas $k \leftarrow \lfloor \frac{|V|}{2} \rfloor$, a complexidade

de tempo total da heurística é de $O(N^3)$.

Assim, a etapa de estender o caminho por um vértice consome $O(N)$ tempo, levando a um custo de $O(N^2)$ para construir um caminho completo em uma tentativa. Multiplicando pelo número de tentativas $k \leftarrow \lfloor \frac{|V|}{2} \rfloor$, a complexidade de tempo total da heurística é de $O(N^3)$.

5.1.4. Problema 2: Força Bruta

A complexidade de tempo do algoritmo de força bruta para o problema do conjunto dominante mínimo é determinada pelo número de subconjuntos que precisam ser verificados. Para um grafo com N vértices, existem 2^N subconjuntos possíveis, e cada um deles precisa ser testado para verificar se é um conjunto dominante válido. A verificação de dominância para cada subconjunto requer examinar cada vértice do grafo e seus vizinhos, o que tem complexidade $O(N + E)$, onde E é o número de arestas. Portanto, a complexidade de tempo total é $O(2^N \cdot (N + E))$.

A complexidade de memória auxiliar é $O(N)$, necessária para armazenar o melhor conjunto dominante encontrado e o conjunto atual sendo testado. A estrutura coveredVertices também requer espaço $O(N)$ durante a verificação de dominância.

5.1.5. Problema 2: Branch and Bound

O algoritmo Branch and Bound para o conjunto dominante mínimo divide o espaço de busca em dois ramos para cada vértice: incluí-lo ou excluí-lo do conjunto dominante. A função lowerBound implementa duas estratégias de poda principais:

Verifica se algum vértice não dominado está no conjunto excluído e não pode ser coberto por vértices indecisos, caso em que o ramo é imediatamente podado. Calcula um limite inferior para o número de vértices adicionais necessários para cobrir os vértices ainda não dominados, baseado na máxima cobertura possível por um único vértice. A complexidade de tempo no pior caso permanece exponencial, $O(2^N \cdot (N + M))$, pois no cenário mais desfavorável, o algoritmo ainda precisaria explorar uma árvore de decisão binária completa. No entanto, as podas eficientes podem reduzir drasticamente o número de estados explorados na prática, especialmente em grafos com estrutura favorável.

A complexidade de memória é $O(N)$ para armazenar os conjuntos currentInSet, currentOutSet, undecidedVertices e a pilha de recursão, que pode atingir profundidade N no pior caso.

5.1.6. Problema 2: Heurístico

O algoritmo Iterated Greedy para o conjunto dominante mínimo opera em duas fases principais: construção inicial gulosa e refinamento iterativo através de destruição parcial e reconstrução.

A fase de construção inicial tem complexidade $O(E \cdot \log N)$, onde em cada iteração o algoritmo seleciona o vértice que cobre o maior número de arestas ainda não

cobertas, o que requer ordenar os vértices por sua cobertura.

A fase iterativa executa maxIterations ciclos de destruição e reconstrução. Em cada ciclo:

A fase de destruição remove aleatoriamente removalSize vértices, com complexidade $O(\text{removalSize})$. A fase de reconstrução identifica arestas descobertas ($O(E)$) e adiciona vértices para cobri-las ($O(E \cdot \log N)$). Portanto, a complexidade total é $O(E \cdot \log N + \text{maxIterations} \cdot (E \cdot \log N))$, que se simplifica para $O(\text{maxIterations} \cdot E \cdot \log N)$.

Como maxIterations e removalSize são constantes ou proporcionais a N, a complexidade de tempo é polinomial, especificamente $O(N \cdot E \cdot \log N)$ no pior caso, tornando esta abordagem significativamente mais eficiente que os métodos exatos para grafos grandes.

A complexidade de memória é $O(E)$ para armazenar as arestas e os conjuntos de vértices da solução, além das estruturas auxiliares como coverageCount.

5.2. Análise de tempo de execução

Os resultados experimentais obtidos através da implementação e execução dos algoritmos para os dois problemas propostos revelam padrões significativos quanto à eficiência computacional e qualidade das soluções.

Para o Problema 1 (ciclo máximo), observamos que o tempo de execução da Força Bruta (1220 ms no grafo G3) foi reduzido com o Branch and Bound (1067 ms), enquanto a abordagem heurística apresentou uma redução drástica para apenas 6 ms. Este padrão se manteve consistente nos grafos menores, com o Branch and Bound mostrando ganhos ainda mais expressivos no grafo G2 (de 38 ms para 5 ms). No grafo G1, que possui menor número de vértices e arestas, os tempos de execução foram similares entre Branch and Bound e a heurística (3 ms). Quanto à qualidade das soluções, como era esperado, os algoritmos exatos, encontraram ciclos idênticos em todos os grafos (32, 30 e 16 vértices), enquanto a heurística produziu resultados variáveis (23 vértices para G3, indefinido para G2, e 15 vértices para G1). É importante ressaltar que o algoritmo heurístico do Problema 1 é muito sensível à topologia do grafo, podendo apresentar variações significativas de desempenho dependendo da estrutura de conexões entre os vértices.

No Problema 2 (conjunto dominante mínimo), as diferenças de desempenho foram ainda mais pronunciadas. A Força Bruta mostrou-se completamente inviável para o grafo G3 e extremamente lenta para o grafo G2 (163590 ms). Em contraste, o Branch and Bound conseguiu resolver o problema no grafo G3 em 3372 ms e reduziu o tempo para o grafo G2 para apenas 21 ms. A abordagem heurística manteve tempos de execução consistentemente baixos (7-15 ms) para todas as instâncias. Quanto à qualidade das soluções, os algoritmos exatos encontraram conjuntos dominantes idênticos para os grafos menores (6 e 9 vértices), enquanto a heurística produziu soluções maiores (9 e 17 vértices). Para o grafo G3, o Branch and Bound encontrou uma solução com 16 vértices, enquanto a heurística retornou 23 vértices, sugerindo que para grafos maiores, a qualidade da solução heurística pode degradar significativamente.

Estes resultados confirmam as análises teóricas de complexidade e evidenciam importantes compromissos entre tempo de execução e qualidade da solução. As abordagens de Força Bruta, embora garantam otimalidade, tornam-se rapidamente impraticáveis

com o aumento do tamanho do problema. O Branch and Bound oferece um equilíbrio valioso, mantendo a garantia de otimalidade enquanto reduz significativamente o tempo de execução através de podas eficientes. As heurísticas destacam-se pela eficiência computacional superior, mas apresentam resultados variáveis em termos de qualidade da solução. Apesar de não garantirem otimalidade, demonstraram desempenho robusto em diversos cenários, sendo altamente recomendadas quando a agilidade é um fator crítico na tomada de decisão.

A escolha da técnica de projeto de algoritmo mais apropriada depende, portanto, dos requisitos específicos da aplicação, considerando o tamanho da instância do problema, as restrições de tempo e a necessidade de soluções ótimas versus aproximadas. Para instâncias pequenas a médias (como G1 e G2), os métodos exatos são viáveis e garantem otimalidade. Para instâncias maiores (como G3) ou aplicações com restrições de tempo rigorosas, as abordagens heurísticas oferecem um compromisso razoável, sacrificando potencialmente alguma qualidade da solução em favor da eficiência computacional.

References

AlgoMonster. *Longest Cycle in a Graph (Problem 2360)*. 2025. ⟨https://algo.monster/liteproblems/2360⟩. Acesso em: 29 mai. 2025.

CACCETTA, L.; VIJAYAN, K. Maximal cycles in graphs. *Discrete Mathematics*, North-Holland, v. 98, n. 1, p. 1–7, dez. 1991. Acesso em: 30 mai. 2025. Disponível em: ⟨https://www.sciencedirect.com/science/article/pii/0012365X9190407S⟩.

CASADO, A. et al. An iterated greedy algorithm for finding the minimum dominating set in graphs. *Mathematics and Computers in Simulation*, Elsevier, v. 207, p. 41–58, 2023. Acesso em: 28 mai. 2025.

GeeksforGeeks. *Dominant Set of a Graph*. 2024. ⟨https://www.geeksforgeeks.org/dominant-set-of-a-graph/⟩. Acesso em: 24 mai. 2024.

GeeksforGeeks. *Find all distinct subsets of a given set*. 2024. ⟨https://www.geeksforgeeks.org/find-distinct-subsets-given-set/⟩. Acesso em: 24 mai. 2025.

GeeksforGeeks. *Finding minimum vertex cover of a graph using Binary Search*. 2024. ⟨https://www.geeksforgeeks.org/finding-minimum-vertex-cover-graph-using-binary-search/⟩. Acesso em: 20 mai. 2025.

WANG, L. et al. An exact algorithm for minimum vertex cover problem. *Mathematics*, MDPI, v. 7, n. 7, p. 603, 2019. Acesso em: 25 mai. 2025. Disponível em: ⟨https://www.mdpi.com/2227-7390/7/7/603⟩.

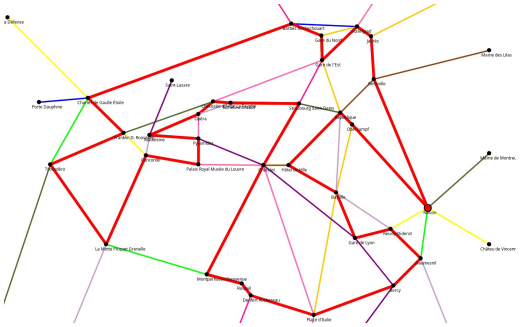


Figure 2. *
P1 Força Bruta / Branch and Bound
G3

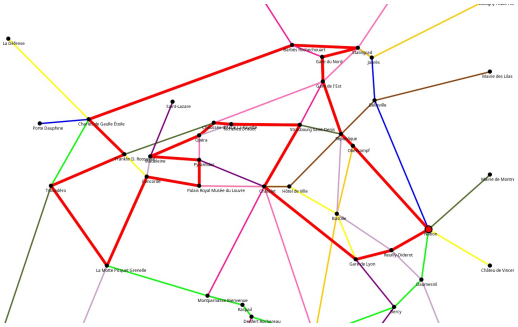


Figure 3. *
P1 Heurística Estocástica
G3

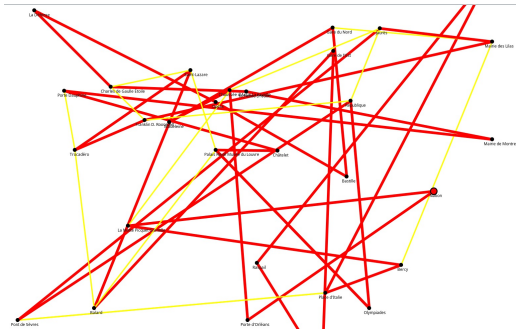


Figure 4. *
P1 Força Bruta / Branch and Bound
G2

Nenhum ciclo foi encontrado pela heurística.

P1 Heurística Estocástica
G2

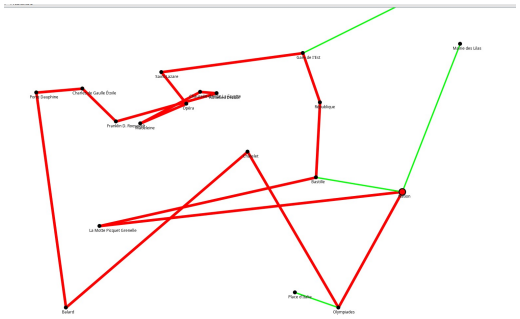


Figure 5. *
P1 Força Bruta / Branch and Bound
G1

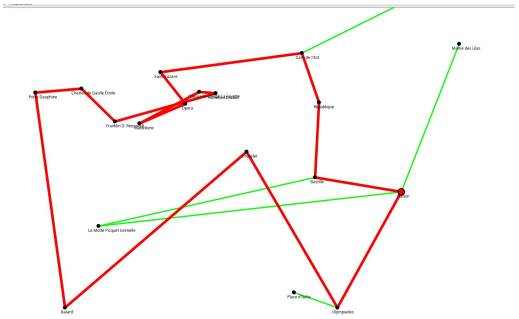


Figure 6. *
P1 Heurística Estocástica
G1

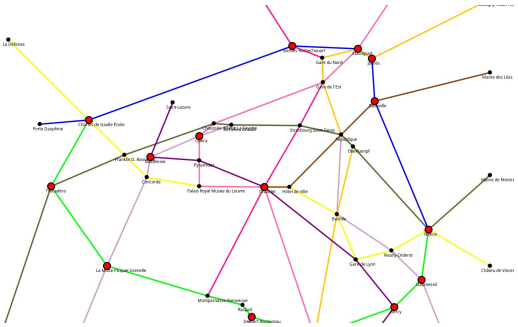


Figure 7. *
P2 Força Bruta / Branch and Bound
G3

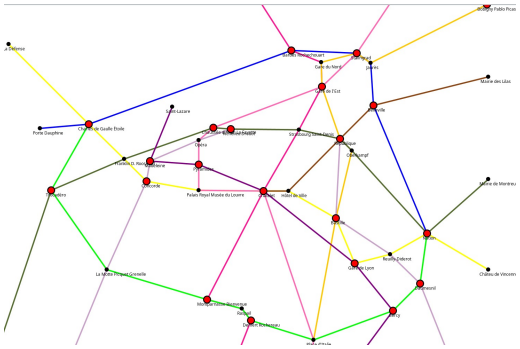


Figure 8. *
P2 Heurística Iterated Greedy
G3

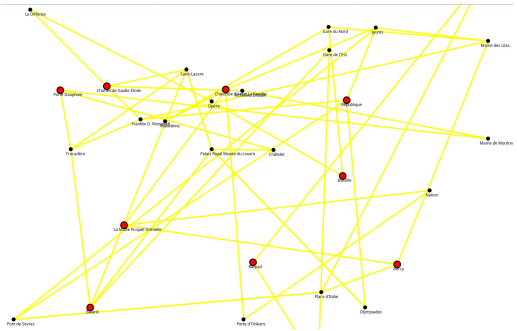


Figure 9. *
P2 Força Bruta / Branch and Bound
G2

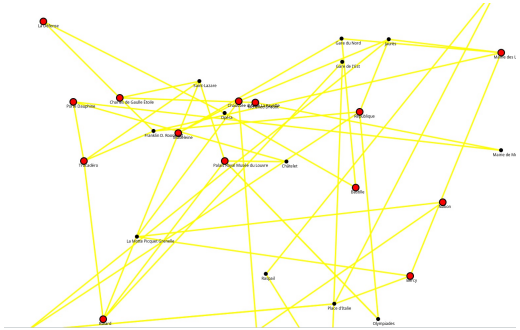


Figure 10. *
P2 Heurística Iterated Greedy
G2

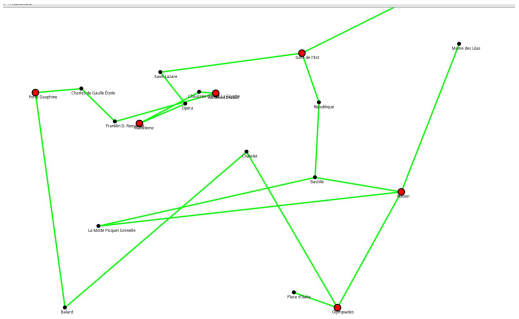


Figure 11. *
P2 Força Bruta / Branch and Bound
G1

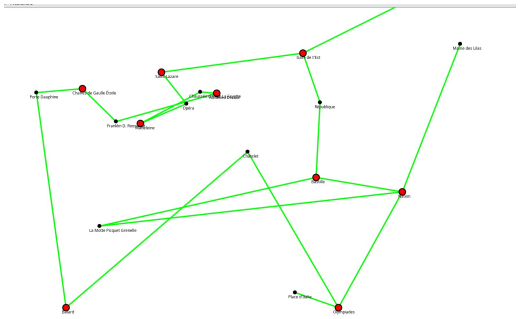


Figure 12. *
P2 Heurística Iterated Greedy
G1