# Problemas de Otimização
# Planejamento de Localização de Lojas

**Carlos Dias Maia**[1], **Ranier Pereira Nunes de Melo**[1]

[1]Pontifícia Universidade Católica de Minas Gerais
Instituto de Ciências Exatas e Informática ICEI
Curso de Graduação em Ciência de Dados - Campus Praça da Liberdade
Belo Horizonte – MG – Brazil

***Abstract.*** *This article presents the solution to an optimization problem: the distribution of store installations according to a criterion of minimum cost, using two project techniques: Brute Force and Branch-and-Bound. A brief description of the techniques is provided, along with an analysis of their complexity order, the implementation of the programs, and tests with simulated data. The average execution times of each technique are compared across the presented scenarios. Finally, it is demonstrated that the Brute Force technique achieves the problem's solution but has a longer execution time compared to the Branch-and-Bound technique in the vast majority of cases.*

***Resumo.*** *Este artigo apresenta a resolução de um problema de otimização: a distribuição de instalação de lojas de acordo com um critério de menor custo, utilizando duas técnicas de projeto: Força Bruta e Branch-and-Bound. São apresentadas uma breve descrição das técnicas, juntamente com a análise da ordem de complexidade, a implementação dos programas, e testes com dados simulados. Os tempos médios de execução de cada técnica são comparados diante dos cenários apresentados. Por fim, demonstra-se que a técnica de Força Bruta atinge a solução do problema, mas tem um tempo de execução maior em comparação à técnica de Branch-and-Bound na grande maioria dos casos.*

## 1. Introdução

Problemas de otimização envolvem a seleção do melhor elemento a partir de um conjunto de alternativas disponíveis, de acordo com um critério específico [Boyd and Vandenberghe 2004]. No contexto da Ciência da Computação, esses problemas são frequentemente classificados como NP-Difíceis, o que significa que não possuem uma solução polinomial determinística conhecida [Hochba 1997]. O objetivo deste trabalho é apresentar duas técnicas de projeto para lidar com esses problemas intratáveis: Força Bruta e Enumeração Implícita (*Branch-and-Bound*).

O problema de otimização que será objeto de análise para apresentar a aplicação dessas duas técnicas de projeto é a otimização da localização de lojas franqueadas em uma determinada região. O objetivo é identificar um arranjo de localizações que evite a competição entre lojas da mesma franquia e que apresente o menor custo de instalação dessas lojas na região de interesse.

Para cada uma das $n$ franquias, existe uma lista de locais candidatos a serem escolhidos. Para cada ponto candidato, são conhecidos o custo de instalação e suas coordenadas $x$, $y$. Duas franquias quaisquer não podem estar a uma distância menor que $D$

quilômetros. Todas as coordenadas $x$, $y$ dos pontos candidatos estão no intervalo de 0 a 500 quilômetros. Caso não seja possível encontrar uma solução que inclua as $n$ franquias devido à restrição de distância, a solução ótima será aquela que maximize o número de franquias instaladas, respeitando a restrição de distância.

Nas próximas sessões, serão apresentadas as soluções para o problema, considerando suas ordens de complexidade em termos de tempo e memória no pior caso. Também será apresentada a implementação dos programas, que foram divididos em módulos. Por fim, será incluído um relatório de testes dos programas, apresentando alguns possíveis arranjos de localização de lojas e seus respectivos custos de instalação, a análise de complexidade dos programas desenvolvidos, além da comparação dos tempos de execução das soluções e da eficiência entre a técnica de Força Bruta e a técnica *Branch-and-Bound*[Lawler and Wood 1966].

## 2. Solução proposta

### 2.1. Descrição das técnicas

Para solucionar o problema foram propostas duas técnicas amplamente utilizadas dentro do contexto de problemas NP-Dificeis dentro da computação.

A primeira técnica de projeto a ser utilizada será a técnica Força Bruta, que consiste em gerar todas as alternativas possíveis dentro da disponibilidade do problema e escolher aquela alternativa que atenda ao critério de otimização [Cormen et al. 2009]. Ou seja, produz-se todos os arranjos possíveis de lojas, as distâncias entre os estabelecimentos, os custos de instalação referentes a esses arranjos e escolhe-se aquele arranjo que tenha o menor custo. Apesar de produzir o resultado que atenda o critério de otimização, esse tipo de técnica atinge esse objetivo ao custo de tempo de processamento e memória, já que produz alternativas que não são relevantes para a otimização. Em problemas de grande escala, a técnica de força bruta pode se tornar impraticável devido à explosão combinatória, tornando essencial a consideração de heurísticas ou outras abordagens mais eficientes.

A segunda técnica utilizada é o *Branch-and-Bound*. Este procedimento enumera implicitamente todas as possíveis soluções para o problema em consideração, os arranjos de possíveis distâncias entre as lojas, armazenando soluções parciais chamadas subproblemas em uma estrutura de árvore. Nós não explorados na árvore geram nós-filhos, particionando o espaço de soluções em regiões menores que podem ser resolvidas recursivamente (*Branch*), e regras são usadas para ”podar” regiões do espaço de busca que são comprovadamente subótimas (*Bound*), no caso, os arranjos que tenham o menor custo até aquele ponto de exploração da árvore. Uma vez que toda a árvore tenha sido explorada, a melhor solução encontrada na busca é retornada [Morrison et al. ]. A técnica *Branch-and-Bound* é particularmente eficaz para reduzir o espaço de busca, especialmente em problemas onde o espaço de solução é vasto. Ela se aproveita de limites superiores e inferiores para evitar a exploração de soluções inviáveis ou subótimas, garantindo uma solução mais eficiente em termos de tempo de processamento em comparação com a força bruta.

### 2.2. Análise de Complexidade

No pior caso, ambas as técnicas apresentam a mesma ordem de complexidade. Podemos obter a ordem de complexidade gerando-se a equação de recorrência:

Para o caso base da equação de recorrência, quando não há mais lojas para processar, a função verifica a combinação atual e retorna, resultando em uma complexidade constante de $T(0) = O(1)$.

No caso recursivo, para cada loja, a função realiza uma chamada recursiva para cada uma das $k$ opções disponíveis e, potencialmente, uma chamada adicional ao excluir a loja. Assim, para a loja atual, a complexidade é $k \times T(n-1)$, mais uma chamada adicional $T(n-1)$ no pior caso.

Portanto, a complexidade total da função pode ser expressa como $k \times T(n-1) + T(n-1)$ no pior caso.

$$T(n) = \begin{cases} O(1) & \text{se } n = 0 \\ (k+1) \times T(n-1) & \text{se } n > 0 \end{cases}$$

Pode-se expandir a equação de recorrência da seguinte forma:

$$T(n) = (k+1) \times T(n-1)$$

$$T(n) = (k+1) \times (k+1) \times T(n-2)$$

$$T(n) = (k+1)^2 \times T(n-2)$$

$$\vdots$$

$$T(n) = (k+1)^n \times T(0)$$

Como $T(0) = O(1)$, temos:

$$T(n) = O((k+1)^n)$$

Além disso, a função `eh_combinacao_valida` também contribui para a complexidade total. Esta função verifica se a combinação de pontos respeita a distância mínima, usando dois loops aninhados para calcular a distância entre todos os pares de pontos na combinação. A complexidade desta função é $O(m^2)$, onde $m$ é o número de pontos na combinação.

Portanto, a complexidade total da solução, levando em consideração a verificação de validade, é:

$$T(n) = O((k+1)^n \cdot m^2)$$

Sendo assim, no pior caso, a solução tem complexidade $T(n) = O((k+1)^n \cdot m^2)$.

Porém, o algoritmo *Branch-and-Bound*, raramente irá ter o pior caso, uma vez que, difirente do Força Bruta, não gera todas as soluções possíveis. Seu caso médio pode ser expressado por:

$$T_{\text{media}} \approx O(E(n) \cdot T_{\text{nó}})$$

Onde:

- $E(n)$ é o número médio de nós explorados.
- $T_{\text{nó}}$ é o tempo médio para processar cada nó.

## 3. Implementação

O programa implementado foi dividido em módulos. A escolha dessa estrutura foi feita para que fossem apresentadas as duas técnicas, Força Bruta e *Branch-and-Bound* de forma individualizada, ao mesmo tempo que torna-se mais organizado a estruturação das outras funcionalidades do programa, como os *benchmarks* de cada método, a geração de arquivos para testes e a representação gráfica de resultados, dentre outros. Cada módulo será apresentado individualmente com maior destaque para as técnicas de projeto, já que são o objetivo principal do presente trabalho.

### 3.1. ForçaBruta.py

Módulo do programa em que aplica-se a técnica Força Bruta para obtenção da solução ótima. Responsável pela produção e análise de todas as possíveis combinações de localidade (coordenadas *x* e *y*) de loja e seus respectivos custos que são possíveis dentro do espaço definido pelo problema, e por fim apresentando a solução ótima.

A função *encontrar_combinacao_otima*, exposta no **Algorithm 1** é responsável por gerar e verificar todas as combinações possíveis de distância dentro de um espaço possível, respeitando-se a distância mínima imposta pelo problema (*distancia_minima*) dentro de um conjunto de lojas e iterando sobre as opções disponíveis para cada loja. Essa iteração ocorre dentro de um *loop* que percorre todas as opções de lojas para a loja atual. Dentro desse *loop*, a função faz chamadas recursivas para continuar explorando as combinações das lojas restantes. Essencialmente, a função usa tanto a iteração quanto a recursão para encontrar a combinação ótima de lojas que respeitam a distância mínima especificada.

Se nenhuma combinação válida é encontrada para a loja atual, a função tenta excluir a loja atual e prosseguir com a próxima. Este é um exemplo de recursão descendente, onde a função se chama a si mesma com parâmetros modificados para resolver subproblemas.

**Algorithm 1** Força Bruta

**Require:** $lojas$, $distancia\_minima$
**Ensure:** $(custo\_otimo, combinacao\_otima)$

**function** ENCONTRAR_COMBINACAO_OTIMA($lojas, distancia\_minima, loja\_atual \leftarrow 1, combinacao\_atual \leftarrow \emptyset, custo\_atual \leftarrow 0$)
  **if** $combinacao\_atual = \emptyset$ **then**
    $combinacao\_atual \leftarrow []$
  **end if**
  **if** $loja\_atual >$ len$(lojas)$ **then**
    **if** EH_COMBINACAO_VALIDA($combinacao\_atual, distancia\_minima$) **then**
      **return** $(custo\_atual, combinacao\_atual)$
    **else**
      **return** $(\infty,$ None$)$
    **end if**
  **end if**
  $custo\_otimo \leftarrow \infty$
  $combinacao\_otima \leftarrow$ None
  **for all** $opcao \in lojas[loja\_atual]$ **do**
    $novo\_custo \leftarrow custo\_atual + opcao[2]$
    $nova\_combinacao \leftarrow combinacao\_atual + [(opcao[0], opcao[1], opcao[2], loja\_atual)]$
    $(custo, combinacao) \leftarrow$ ENCONTRAR_COMBINACAO_OTIMA($lojas, distancia\_minima, loja\_atual + 1, nova\_combinacao, novo\_custo$)
    **if** $custo < custo\_otimo$ **then**
      $custo\_otimo \leftarrow custo$
      $combinacao\_otima \leftarrow combinacao$
    **end if**
  **end for**
  **return** $(custo\_otimo, combinacao\_otima)$
**end function**

### 3.2. BranchAndBound.py

Módulo do programa em que aplica-se a técnica de *Branch-and-Bound* para obtenção da solução ótima. Responsável pela produção e análise eficiente de combinações determinadas por um limite inferior (*limite_inferior*), onde cria-se uma combinação de lojas e suas distâncias, avalia-se, de forma recursiva, se a solução atual está com valor superior do que outra solução já encontrada. Caso seja encontrada, aquele nó é podado e aquela alternativa de solução não prossegue. Ou seja, as possibilidades que já estão direcionando-se a uma solução não ótima dentro da árvore decisória, já são previamente analisadas e podadas antes mesmo de sua finalização, diminuindo o tempo necessário de processamento para produção da solução ótima.

Cabe ressaltar que o programa *BranchAndBound.py* é estruturado em cima do que já foi apresentado em relação ao módulo *ForçaBruta.py*, adicionando-se a condição

de limite inferior e de poda, conforme pode ser visualizado em **Algorithm 2** na função *encontrar_combinacao_otima*, onde se compara o custo da solução atual com o menor custo encontrado, armazenado na variável *melhor_custo_ate_agora*.

### 3.3. geraTestes.py

Módulo do programa responsável pela geração de arquivos com valores aleatórios. Cada cenário é produzido pela função *gera_cenarios_aleatorios* produzindo valores aleatórios para as variáveis: coordenadas *x* e *y* em uma faixa de 0 a 500, o custo de instalação (*custo*) no valor de 500 a 2000, retornando-se *lojas*, variável que armazena a lista que será preenchida pelos valores gerados aleatoriamente. A variável *n* na função representa o número de franquias. São gerados dez cenários aleatórios para cada *n* franquias. O número de franquias varia de 4 até 20, gerando-se 170 simulações ao todo. Os arquivos de simulação são armazenados na estrutura em arquivos txt, com a seguinte nomenclatura: scenario_*n*_*simulation*.txt.

### 3.4. benchmark - Branch-and-Bound.py

Módulo do programa responsável pela execução do programa *BranchAndBound.py*. O programa é alimentado pelos pontos simulados presentes nos arquivos de formato txt(*scenario_n*_sim*.txt*). Ao finalizar a execução do programa, o resultado das simulações são armazenados no arquivo *Tempos de execução - output - Branch and Bound.txt*, sendo estruturados linha a linha na seguinte forma: nome do arquivo referente a simulação, tempo de execução do programa *BranchAndBound*, a combinação ótima de localização de lojas nas coordenadas *x* e *y* e o custo ótimo.

### 3.5. benchmarks - Força bruta.py

Módulo do programa responsável pela execução do programa *ForçaBruta.py*. O programa é alimentado pelos pontos simulados presentes nos arquivos de formato txt(*scenario_n*_sim*.txt*). Ao finalizar a execução do programa, o resultado das simulações são armazenados no arquivo *Tempos de execução - outut - Força Bruta*, sendo estruturados linha a linha na seguinte forma: nome do arquivo referente a simulação, tempo de execução do programa *ForçaBruta*, a combinação ótima de localização de lojas nas coordenadas *x* e *y* e o custo ótimo.

### 3.6. carregarDados.py

Módulo do programa responsável por carregar os arquivos de formato txt(*scenario_n*_sim*.txt*). Contém a função *carregar_dados_txt_para_dicionario*. Função responsável pela leitura dos dados dos arquivos de texto alimentados e transformá-los em formato de dicionário, onde o índice é a franquia que armazenada todos as coordenadas *x* e *y* e os referentes custos de instalação das opções de lojas que correspondem a ela.

### 3.7. distancia.py

Módulo do programa responsável pelo cálculo de distância entre dois pontos, executado pela função *distancia* que calcula a distância entre 2 pontos $(x_1, y_1)$ e $(x_2, y_2)$ utilizando a fórmula de distância euclidiana, dada pela fórmula abaixo:

$$d\left(x, y\right) = \sqrt{\sum_{i=1}^{n} \left(x_i - y_i\right)^2}$$

**Algorithm 2** Branch-and-Bound

**Require:** $lojas$, $distancia\_minima$
**Ensure:** Combinação ótima de lojas

**function** `encontrar_combinacao_otima`($lojas$, $distancia\_minima$, $loja\_atual \leftarrow 1$, $combinacao\_atual \leftarrow$ None, $custo\_atual \leftarrow 0$, $melhor\_custo\_ate\_agora \leftarrow \infty$)
  **if** $combinacao\_atual$ is None **then**
    $combinacao\_atual \leftarrow []$
  **end if**
  **if** $loja\_atual >$ len($lojas$) **then**
    **if** `is_valid_combination`($combinacao\_atual$, $distancia\_minima$) **then**
      **return** $(custo\_atual, combinacao\_atual)$
    **else**
      **return** $(\infty,$ None$)$
    **end if**
  **end if**
  $limite\_inferior \leftarrow custo\_atual$
  **for** $loja \leftarrow loja\_atual$ **to** len($lojas$) **do**
    $custo_minimo \leftarrow \min($opcao[2] **for** opcao in$lojas[loja])$
    $limite\_inferior \leftarrow limite\_inferior + custo\_minimo$
  **end for**
  **if** $limite\_inferior > melhor\_custo\_ate\_agora$ **then**
    **return** $(\infty,$ None$)$
  **end if**
  $custo\_otimo \leftarrow \infty$
  $combinacao\_otima \leftarrow$ None
  **for each** $opcao$ **in** $lojas[loja\_atual]$ **do**
    $novo\_custo \leftarrow custo\_atual + opcao[2]$
    $nova\_combinacao \leftarrow combinacao\_atual + [(opcao[0], opcao[1], opcao[2], loja\_atual)]$
    $(custo, combinacao) \leftarrow$ `encontrar_combinacao_otima`($lojas, distancia\_minima, loja\_atual + 1, nova\_combinacao, novo\_custo, melhor\_custo\_ate\_agora$)
    **if** $custo < melhor\_custo\_ate\_agora$ **then**
      $melhor\_custo\_ate\_agora \leftarrow custo$
    **end if**
    **if** $custo < custo\_otimo$ **then**
      $custo\_otimo \leftarrow custo$
      $combinacao\_otima \leftarrow combinacao$
    **end if**
  **end for**
  **return** $(custo\_otimo, combinacao\_otima)$

### 3.8. plotSolucao.py

Módulo do programa responsável por executar a representação gráfica das soluções ótimas para cada simulação. Utiliza-se a biblioteca *matplotlib* utilizada para criar

visualizações gráficas. A função *plotar_solucao* extrai os dados obtidos pela simulação scenario_*n*_*simulation* e a representa em uma imagem de formato *PNG*, apresentando a localização de cada loja em um plano cartesiano, onde os limites de *x* e *y* são delimitados entre -100 e 500 (os valores negativos foram adicionados para melhor visualização gráfica) e por fim apresenta-se o custo de instalação das lojas. Ou seja, apresenta-se a solução ótima em forma visual.

## 4. Relatório de testes

Os testes foram executados nos respectivos módulos *benchmark* de cada técnica de projeto conforme apresentado na sessão de Implementação. Os arquivos utilizados para a execução dos testes, são aqueles que foram produzidos no módulo *geraTestes.py*, com valores aleatórios referentes às coordenadas *x* e *y* e o custo de instalação das lojas, mantendo-se constante os valores de quantidade de lojas por franquia *m*=2, ou seja para cada valor N de franquias criaram-se 2 pontos candidatos, e distância mínima de um quilômetro *D*=1.

Os 170 arquivos formam 17 cenários definidos pela quantidade N de franquias com 10 simulações cada. Os tempos médios de execução em cada cenário foram compilados para cada técnica de projeto a título de comparação de performance de ambos os algoritmos de acordo com a definição de N. Todos os dados estão registrados na Tabela 1.

Além do registro tabular, apresenta-se os gráficos individualizados de cada técnica representando o tempo médio de execução em função de N franquias: Força Bruta (Figura 1) e *Branch-and-Bound* (Figura 2). Também é apresentado a comparação do tempo médio de execução das duas técnicas (Figura 3). Verifica-se imediatamente que quanto maior o valor de N, menor é o tempo médio de execução do *Branch-and-Bound* em relação a Força Bruta.

Por fim, também apresenta-se alguns resultados das simulações de distribuição de loja, por meio do módulo *plotSolucao.py*, os arquivos utilizados no módulo **benchmark - Branch-and-Bound.py** são os seguintes:

1. 5 Franquias - Simulação 6 - Tempo de execução: 0s (colocar figura)
2. 10 Franquias - Simulação 10 - Tempo de execução: 0s (colocar figura)
3. 15 Franquias - Simulação 3 - Tempo de execução: 0,001s (colocar figura)

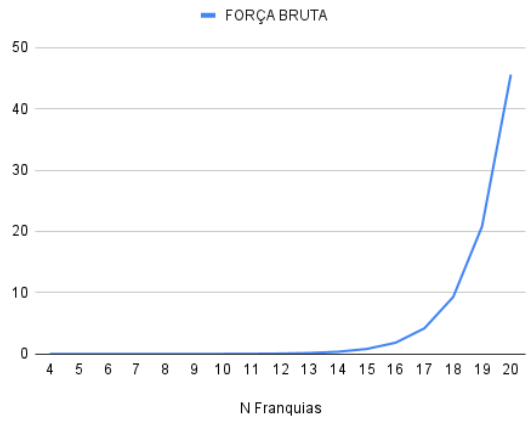


**Figure 1. Tempo médio em segundos em função de N - Força Bruta**

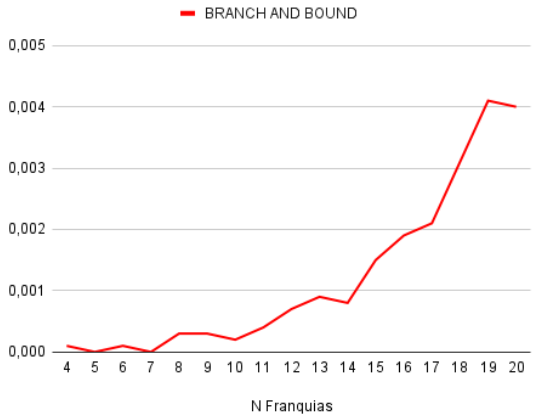


**Figure 2. Tempo médio em segundos em função de N - *Branch-and-Bound***

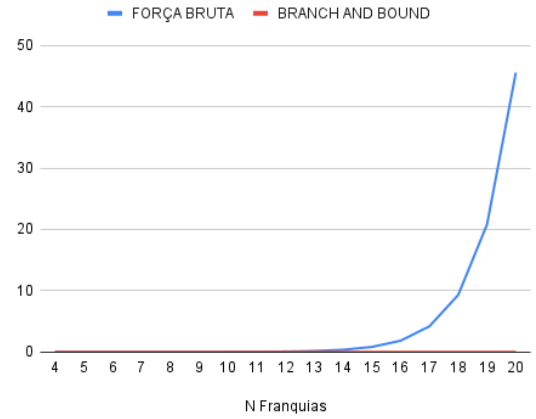


**Figure 3. Tempo médio em segundos em função de N**

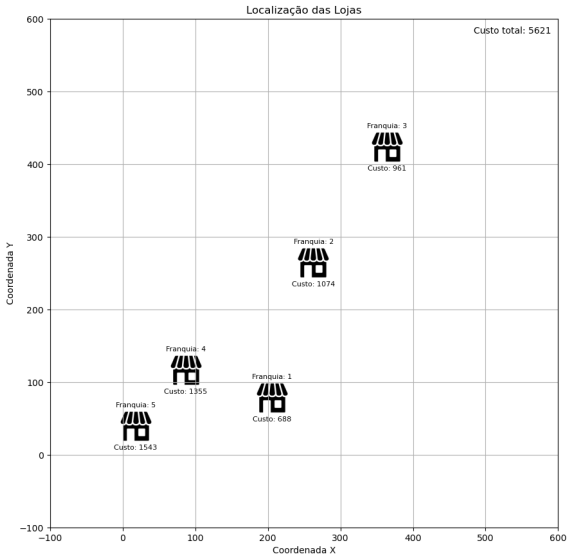


**Figure 4. 5 Franquias - Simulação 6 - Tempo de execução: 0s**

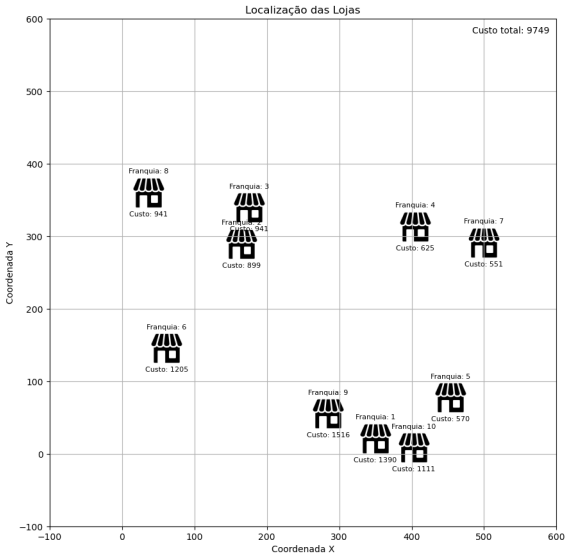


**Figure 5. 10 Franquias - Simulação 10 - Tempo de execução: 0s**

**Table 1. Tempos de execução médio das técnicas**

| N | Força Bruta (s) | *Branch-and-Bound* (s) |
|---|---|---|
| 4 | 0,0003 | 0,0002 |
| 5 | 0,00025 | 0,00082 |
| 6 | 0,00128 | 0,00063 |
| 7 | 0,00276 | 0,00048 |
| 8 | 0,00812 | 0,00083 |
| 9 | 0,02748 | 0,00112 |
| 10 | 0,06015 | 0,00408 |
| 11 | 0,13529 | 0,00588 |
| 12 | 0,30071 | 0,0094 |
| 13 | 0,53877 | 0,00677 |
| 14 | 1,26136 | 0,00723 |
| 15 | 2,83418 | 0,03508 |
| 16 | 4,73605 | 0,06247 |
| 17 | 10,51198 | 0,05566 |
| 18 | 22,74599 | 0,05201 |
| 19 | 45,40233 | 0,14307 |
| 20 | 88,38337 | 0,34613 |

## 5. Conclusão

A análise comparativa entre os algoritmos de Força Bruta e *Branch-and-Bound* demonstra diferenças significativas em termos de eficiência computacional e complexidade de tempo. O algoritmo de Força Bruta, apesar de sua simplicidade conceitual e fácil implementação, revela-se inadequado para problemas de grande escala devido ao seu crescimento exponencial na exploração de todas as possíveis combinações de soluções. A complexidade de tempo $O(m^n)$ deste método torna-o inviável para aplicações práticas onde $n$, a entrada de dados, é grande.

Por outro lado, o algoritmo *Branch-and-Bound* apresenta uma abordagem mais sofisticada, utilizando a poda de nós para eliminar subproblemas que não podem melhorar a solução conhecida. Essa técnica reduz significativamente o número de nós explorados, resultando em uma complexidade de tempo média representada por $T_{\text{média}} \approx O(E(n) \cdot T_{\text{nó}})$. A eficácia da poda depende da estrutura do problema e das heurísticas utilizadas, mas, em geral, o *Branch-and-Bound* mostra-se substancialmente mais eficiente que a força bruta.

Em resumo, enquanto a Força Bruta pode ser considerada como uma abordagem teórica útil para pequenos problemas ou como um ponto de partida para compreensão básica do problema, o *Branch-and-Bound* se destaca como uma técnica poderosa e prática para resolver problemas de otimização combinatória de maneira mais eficiente. A utilização de um limite inferior e poda de subproblemas não promissores permitiu que o Branch and Bound reduzi-se drasticamente o tempo de processamento, tornando-o a escolha preferível no problema trabalhado nesse artigo e em cenários reais onde a eficiência computacional é crucial.

Esta conclusão reafirma a importância da escolha do algoritmo adequado para problemas de otimização, destacando o Branch and Bound como uma técnica eficaz para encontrar soluções ótimas em um tempo computacionalmente viável.

## 6. Bibliografia

## References


Boyd, S. P. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Hochba, D. S. (1997). Approximation algorithms for np-hard problems. *ACM Sigact News*, 28(2):40–52.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations research*, 14(4):699–719.

Morrison, D. R., Jacobson, S. H., Sauppe, J. J., and Sewell, E. C. Branch-and-bound algorithms: Recent advances in searching, branching, and pruning.