Trabalho de Implementação 02: Força Bruta e Branch and Bound

João Pedro L. de Pinho[1], **Lucas Carvalho da Luz**[1]

[1]ICEI – Pontifícia Universidade Católica de Minas Gerais (PUC)
Belo Horizonte – MG – Brazil

joao.pinho.1275438@sga.pucminas.br

lcluz@sga.pucminas.br

Abstract. *This paper presents the implementation of Java algorithms to solve the problem of product distribution among branches in a store network, considering the minimization of fuel consumption and the maximum capacity of the truck. Two approaches are compared: the Branch and Bound technique and the Brute Force approach. Tests were conducted on different problem instances, varying the number of stores and the truck's capacity. The results highlight the inefficiency of the Brute Force approach, while the Branch and Bound technique demonstrates satisfactory performance, providing optimal solutions in a timely manner. These findings emphasize the importance of using appropriate optimization techniques to solve specific types of problems.*

Resumo. *Este trabalho apresenta a implementação de algoritmos em Java para resolver o problema da distribuição de produtos entre as filiais de uma rede de lojas, considerando a minimização do consumo de combustível e a capacidade máxima de carga do caminhão. Duas abordagens são comparadas: Branch and Bound e Força Bruta. Foram realizados testes em diferentes instâncias do problema, variando o número de lojas e a capacidade do caminhão. Os testes destacam a ineficiência da abordagem de Força Bruta, enquanto o Branch and Bound demonstra um desempenho satisfatório, fornecendo soluções ótimas e em tempo hábil. Esses resultados mostram a importância de utilizar técnicas de otimização adequadas para a solução de determinados tipos de problemas.*

1. Introdução

A distribuição eficiente de produtos em redes de lojas é um desafio enfrentado por diversas empresas. A otimização desse processo pode resultar em economia de recursos, redução de custos, eficiência de tempo e maior satisfação dos clientes. Nesse contexto, esse trabalho tem como objetivo comparar duas abordagens para resolver o problema de distribuição de produtos em uma rede de lojas: o Branch and Bound e a Força Bruta.

O problema consiste em encontrar a rota ótima para um caminhão, que parte da matriz e visita todas as lojas apenas uma vez, entregando ou coletando produtos, e retorna à matriz vazio. O objetivo é minimizar o consumo de combustível, levando em consideração a distância percorrida e a quantidade de produtos transportados. O caminhão possui uma capacidade máxima de carga, e cada unicade de produuto afeta o consumo de combustível em uma determinada proporção.

A abordagem de Branch and Bound utiliza uma estratégia de ramificação e limitação para explorar subconjuntos promissores de soluções, reduzindo a busca e obtendo um resultado em tempo hábil. Por outro lado, a abordagem que utiliza Força Bruta gera todas as permutações possíveis e seleciona a solução válida com menor consumo de combustível. Embora ambos os métodos cheguem ao mesmo resultado ótimo em termos de rota e consumo de combustível, a diferença reside na eficiência e complexidade. O Branch and Bound é mais eficiente, enquanto a Força Bruta é ineficiente devido à sua complexidade fatorial.

Foram realizados testes comparativos em diferentes instâncias do problema, variando o número de lojas e a carga máxima do caminhão. O objetivo desses testes é demonstrar a superioridade do algoritmo Branch and Bound em termos de desempenho, com menor tempo de execução em comparação a abordagem de Força Bruta. Os resultados obtidos permitiram identificar a abordagem mais eficiente para resolver o problema de distribuição de produtos em redes de loja.

2. Solução Proposta

Como dito anteriormente, para a solução do problema foram utilizadas duas abordagens, Branch and Bound e Força Bruta. Ambas técnicas são utilizadas para o mesmo proposito, encontrar a solução ótima que minimize o consumo de combustível levando em consideração a capacidade máxima de carga do caminhão e a distância percorrida.

O algoritmo de Branch and Bound utiliza a estratégia de podar ramos que são considerados infrutíferos, dessa forma poupando a exploração de subconjuntos que violam as restrições do problema ou que são piores que a melhor solução encontrada até o momento.

Ele começa com uma solução inicial vazia e itera pelo conjunto disponível gerando as ramificações. Durante a busca, um limite inferior (lower bound) é aplicado para descartar os ramos que não levarão a solução ótima. A ideia é que, ao descartar soluções parciais que já possuem um consumo de combustível maior do que a melhor solução encontrada até o momento, é possível reduzir o espaço de busca. O desempenho de um programa Branch and Bound está fortemente relacionado à qualidade dos seus limitantes inferiores e superiores.

Quanto mais precisos forem esses limitantes, menos soluções parciais serão consideradas e mais rápido a solução ótima é encontrada. Portanto, a ordem de complexidade do problema da distribuição de produtos com a técnica de Branch and Bound depende do número de lojas, capacidade máxima do caminhão e a disposição dos produtos nas lojas, no geral, resulta-se em uma complexidade exponencial, limitada a ordem de $O(n^2)$.

Por outro lado, a abordagem de Força Bruta gera todas as permutações possíveis do conjunto de lojas e seleciona a solução com menor consumo de combustível. O algoritmo percorre todas as permutações, avaliando o consumo de combustível de cada uma delas e validando a solução, isto é, se não sobraram produtos a serem entregues. Essa abordagem é caracterizada por sua simplicidade conceitual, pois explora todas as possibilidades. No entanto, a sua complexidade é fatorial $O(n!)$, pois o número de permutações cresce exponencialmente com o número de lojas, tornando-se rapidamente inviável para problemas de grande escala, como será visto nos testes conduzidos.

3. Implementação

Antes de prosseguir com a explicação da implementação das soluções propostas, é importante contextualizar as classes desenvolvidas, seu papel no algoritmo de solução, bem como o ambiente de desenvolvimento e os testes realizados.

As implementações foram desenvolvidas no Visual Studio Code, utilizando projetos padrão para Java. Os testes foram realizados em um ambiente com as seguintes características:

1: **Sistema Operacional:** Windows 10.
2: **Processador:** Intel(R) Core(TM) i5-6200U CPU @ 2.30 GHz 2.40GHz.
3: **Memória RAM:** 8 GB.
4: **JDK:** Java 19.0.2.

Na implementação do algoritmo, foram criadas duas classes: a classe ”Loja” e a classe ”Caminhao”. Essas classes desempenham papéis fundamentais no controle das soluções propostas.

A classe ”Loja” representa uma loja e possui atributos como número de identificação, coordenadas no plano e lista de destinos disponíveis. O número de identificação serve como um identificador único para distinguir uma loja das demais. As coordenadas no plano (representadas pelos valores x e y) indicam a localização da loja no mapa e são utilizadas para calcular a distância euclidiana entre as lojas, auxiliando na determinação da rota ótima do caminhão.

A lista de destinos de uma loja representa os produtos disponíveis nessa loja. Cada produto é identificado por um número único. Essa lista é essencial para determinar quais produtos podem ser coletados e entregues pelo caminhão em cada visita à loja. Por exemplo, se a loja 1 tem o produto 2 em sua lista de destinos, o caminhão deve coletar o produto 2 na loja 1 e entregá-lo na loja 2.

A classe ”Loja” possui métodos que auxiliam nas soluções, como o método ”clone()” para clonar uma loja específica e o método ”clonarListaLoja()” para clonar uma lista de lojas. Esses métodos evitam o uso de referências originais durante os cálculos de rendimento, consumo e coleta de produtos. Além disso, a classe possui métodos de leitura de arquivo para criar a lista de lojas disponíveis, sendo essencial para carregar os dados das lojas a partir de um arquivo.

Há também métodos relacionados aos produtos presentes nas listas de destinos. O método ”getProductList()” extrai uma lista com todos os produtos presentes nas listas de destinos das lojas, sendo utilizado na implementação do algoritmo Branch and Bound. O método ”restouProdutos()” verifica se ainda há produtos restantes nas listas de destinos das lojas, auxiliando na validação de um caminho após o cálculo na implementação por força bruta.

A classe ”Caminhao” desempenha um papel central na solução proposta, representando o caminhão utilizado para realizar as entregas. Ela possui atributos que representam a capacidade máxima do caminhão, a carga atual e a lista de produtos presentes no caminhão.

O caminhão é caracterizado pela sua capacidade máxima, que define a quantidade máxima de produtos que ele pode transportar simultaneamente. A carga atual do cam-

inhão indica o número de produtos atualmente presentes nele, levando em consideração as entregas e coletas realizadas. A lista de produtos no caminhão indica os produtos que estão sendo transportados pelo caminhão em um determinado momento, essa lista é essencial para realizar as operações de entrega e coleta.

A classe "Caminhao" oferece métodos para manipular a carga do caminhão e realizar as operações necessárias para resolver o problema. O método "entregarProdutos()" é responsável por realizar a entrega de produtos em uma determinada loja. Ele remove os produtos entregues da lista do caminhão, atualiza a carga atual e retorna uma lista com os produtos entregues. O método "coletarProdutos()" realiza a coleta de produtos de uma loja, adicionando-os ao caminhão, atualizando a carga atual, removendo-os da lista de destinos da loja e retorna uma lista com os produtos coletados. Esses métodos garantem que a capacidade máxima do caminhão não seja excedida em nenhuma operação. O pseudo-código abaixo mostra a implementação desses métodos:

Algorithm 1 Entregar Produtos

```
1: function ENTREGARPRODUTOS(lojaDeEntrega, caminhao)
2:     produtosEntregues ← lista vazia
3:     if caminhao.contem(lojaDeEntrega) then
4:         Remover lojaDeEntrega da lista de produtos do caminhao
5:         Adicionar lojaDeEntrega à lista de produtosEntregues
6:         Atualizar a cargaAtual do caminhao
7:     return produtosEntregues
```

Algorithm 2 Coletar Produtos

```
1: function COLETARPRODUTOS(lojaDeColeta, caminhao)
2:     produtosColetados ← lista vazia
3:     if lojaDeColeta.destinosEstaDisponivel() then
4:         for cada produto na lista de destinos da lojaDeColeta do
5:             if caminhao.cargaAtual < CAPACIDADE_MAXIMA then
6:                 Adicionar produto à lista de produtos do caminhao
7:                 Adicionar produto à lista de produtosColetados
8:                 Remover produto da lista de destinos da lojaDeColeta
9:                 Decrementar o índice do loop em 1
10:                Incrementar a cargaAtual do caminhao
11:    return produtosColetados
```

3.1. Força Bruta

A abordagem de força bruta consiste em gerar todas as permutações possíveis das lojas, calcular o consumo de cada permutação e validar o caminho calculado. Em seguida, é selecionada a permutação com o menor consumo como a melhor solução. O algoritmo de força bruta pode ser dividido em três etapas: geração das permutações, cálculo do consumo e seleção da melhor solução. A implementação foi feita na classe "BruteForce".

3.1.1. Geração das Permutações

Para gerar todas as permutações possíveis das lojas, utilizou-se uma abordagem recursiva de troca de elementos. Inicialmente, temos uma lista com os índices das lojas, exceto a loja matriz (de onde o caminhão parte). A cada iteração, trocamos dois elementos da lista e chamamos recursivamente a função para o próximo índice. Esse processo é repetido até que todas as permutações sejam geradas. O pseudo código abaixo mostra como é feita a geração de permutações.

Algorithm 3 Gerar Permutações

```
1:  function GERARPERMUTACOES(lojas)
2:      permutacoes ← []
3:      index ← obterIndicesLojas(lojas)
4:      permutacoes(index, 0, permutacoes)
5:      return permutacoes
6:  function PERMUTACOES(permutacao, index, todasPermutacoes)
7:      if index = tamanho(permutacao) - 1 then
8:          adicionarCopia(permutacao, todasPermutacoes)
9:      else
10:         for i from index to tamanho(permutacao) - 1 do
11:             trocarElementos(permutacao, index, i)
12:             permutacoes(permutacao, index + 1, todasPermutacoes)
13:             trocarElementos(permutacao, index, i)
```

A função 'gerarPermutacoes' recebe a lista de lojas como parâmetro e retorna uma lista de inteiros, onde cada índice é uma permutação. Ela inicializa a lista 'permutacoes' vazia e obtém os índices das lojas utilizando a função 'obterIndicesLojas'. Em seguida, chama a função 'permutacoes' para gerar as permutações, passando a lista de índices, o índice inicial (0) e a lista 'permutacoes' como parâmetros. Por fim, retorna a lista de permutações.

A função 'permutacoes' é uma função recursiva que realiza a troca de elementos e a chamada recursiva. Ela verifica se o índice atual é igual ao tamanho da permutação menos um. Se for, significa que uma permutação completa foi gerada e é adicionada à lista 'todasPermutacoes'. Caso contrário, itera sobre os índices a partir do índice atual e realiza a troca de elementos. Em seguida, chama recursivamente a função para o próximo índice e desfaz a troca de elementos antes de continuar o loop.

3.1.2. Cálculo do Consumo

Após gerar todas as permutações, é necessário calcular o consumo de cada permutação. O método 'calcularConsumoPermutacao' é responsável por determinar o consumo de combustível de uma permutação específica de lojas e verificar se essa permutação é válida. O pseudo-código 4, mostra como é feito o cálculo e em seguida sua validação.

Algorithm 4 Calcular Consumo de uma Permutação

```
1:  function CALCULARCONSUMOPERMUTACAO(listaLoja, caminho)
2:      listaLojaCopy ← ClonarListaLoja(listaLoja)
3:      consumoDaPermutacao ← 0.0
4:      caminhao ← CriarCaminhao()
5:      rendimento ← 10.0
6:      AdicionarOrigemDestino(caminho)          ▷ Adiciona matriz na permutação.
7:      for cada índiceLoja de 0 até (tamanho do caminho - 1) do
8:          lojaAtual ← listaLojaCopy[caminho[índiceLoja]]
9:          proximaLoja ← listaLojaCopy[caminho[índiceLoja + 1]]
10:         EntregarProdutos(lojaAtual, caminhao)
11:         ColetarProdutos(lojaAtual, caminhao)
12:         rendimento ← ObterRendimento(caminhao)
13:         distancia ← CalcularDistancia(lojaAtual, proximaLoja)
14:         consumoDeTrajeto ← distancia / rendimento
15:         consumoDaPermutacao ← consumoDaPermutacao + consumoDeTrajeto
16:     if CargaCaminhao(caminhao) ≠ 0 ou RestouProdutos(listaLojaCopy) then
17:         consumoDaPermutacao ← ∞
18:     retorne consumoDaPermutacao
```

Primeiramente, a função realiza uma clonagem da lista original de lojas para evitar alterações indesejadas durante o cálculo do consumo. Isso é feito para garantir que cada permutação seja avaliada de forma isolada, sem interferir nas outras, já que como já visto, quando um produto é coletado é também excluído da lista de destinos da loja de coleta.

Em seguida, são inicializadas as variáveis locais para controle do método, como a variável que guarda o consumo da permutação, representando o valor acumulado do consumo de combustível de todas as trajetórias percorridas na permutação mediante ao rendimento do objeto caminhão criado.

A permutação é modificada para incluir a loja de origem (matriz) como o primeiro elemento e a loja de destino (matriz) como o último elemento. Essa modificação é necessária para garantir que o caminhão comece e termine sua rota na loja matriz, conforme a restrição do problema.

Em seguida, o método percorre a permutação de lojas. Para cada loja na permutação, são realizadas as operações de entrega e coleta de produtos no caminhão. Essas operações simulam o carregamento e descarregamento de produtos durante o percurso. O rendimento do caminhão é atualizado com base no seu estado atual, considerando a carga e descarga de produtos realizadas.

Após cada operação, o método calcula a distância entre a loja atual e a próxima loja na permutação. Essa distância é calculada levando em consideração as coordenadas geográficas das lojas. Com a distância calculada, o método determina o consumo de combustível do trajeto dividindo a distância pelo rendimento atual do caminhão.

Ao final da iteração sobre a permutação, o método realiza uma verificação da validade do caminho. Se o caminhão ainda tiver produtos em sua carga ou se houverem produtos não coletados nas lojas, a permutação é considerada inválida. Nesse caso, é

atribuído infinito ao valor retornado, indicando que essa permutação não é uma solução válida.

Caso a permutação seja considerada válida, a variável que acumulou o consumo da permutação representa o consumo de combustível total para percorrer essa rota específica. Essa informação será utilizada posteriormente na comparação com outras permutações para determinar a melhor solução possível.

3.1.3. Seleção da Melhor Solução

Após calcular o consumo de todas as permutações, é selecionada a permutação com o menor consumo como a melhor solução.

A implementação de Força Bruta recebe a lista de lojas como parâmetro. Inicializa as variáveis para guardar a melhor permutação e o menor consumo. Em seguida, gera todas as permutações chamando a função de gerar permutações descrita anteriormente.

O método itera sobre todas as permutações e calcula o consumo de cada permutação, como visto na seção anterior. Se o consumo da permutação atual for menor que o menor consumo registrado até o momento, o menor consumo e a melhor permutação são atualizados.

Ao final do algoritmo, a melhor permutação e o menor consumo são armazenados, se não houver solução possível, seja devido a carga máxima utilizada ou devido a configuração do arquivo de lojas, o valor da melhor permutação será nulo e do menor consumo infinito.

3.2. Branch and Bound

O algoritmo de *Branch and Bound* é uma técnica utilizada para resolver problemas de otimização. Ele explora de maneira inteligente o espaço de busca, evitando a avaliação de soluções que claramente não serão as melhores. No caso do problema de cálculo de consumo em rotas de entrega, o algoritmo busca encontrar o caminho de entrega com o menor consumo possível, avaliando as restrições para evitar entrar em ramos infrutíferos. Em linhas gerais o funcionamento resumido da solução pode ser observada abaixo:

Algorithm 5 Branch and Bound

```
1: procedure BRANCHNBOUND(listaLojas, permutacaoAtual, lojasDisponiveis, consumoAtual, index)
2:     if lojasDisponiveis está vazio then
3:         Atualizar melhorPermutacao se consumoAtual for menor que menorConsumo
4:     else
5:         for cada loja disponível do
6:             if a loja não possui produtos não coletados e carga do caminhão não é excedida then
7:                 Atualizar permutacaoAtual, lojasDisponiveis e consumoAtual
8:                 Chamar recursivamente branchNbound com os parâmetros atualizados
9:                 Desfazer as alterações realizadas (backtracking)
```

A função `branchNbound` é recursiva e realiza as seguintes etapas:

1. Verificação de condição de parada: Se não há mais lojas disponíveis, ou seja, um caminho foi encontrado, caso o consumo atual seja menor do que o menor consumo encontrado até o momento, atualiza-se o menor consumo e a melhor permutação.
2. Caso contrário, para cada loja disponível, verifica-se se a loja atual contém produtos que ainda não foram coletados. Se a restrição do problema for satisfeita, ou seja, a carga total da loja atual não excede a capacidade do caminhão e o consumo atual é menor do que o menor consumo encontrado até o momento, realiza-se as seguintes ações:
   (a) Adiciona os produtos entregues ao caminhão.
   (b) Atualiza a permutação atual adicionando a loja atual.
   (c) Remove a loja atual da lista de lojas disponíveis.
   (d) Calcula a distância entre a loja anterior e a loja atual, e atualiza o consumo atual de acordo com o consumo de trajeto.
   (e) Simula a entrega e coleta de produtos na loja atual e atualiza a lista de todos os produtos.
   (f) Chama recursivamente a função `branchNbound` para explorar as próximas ramificações.
3. Após o término da recursão, as ações realizadas no passo 2 são desfeitas para permitir a exploração de outras possibilidades.

Embora a abordagem que utiliza branch and bound não é caracterizada por sua simplicidade conceitual, em relação a força bruta, sua implementação é mais eficiente por que reduz significativamente o espaço de busca em linhas gerais.

3.3. Interface Gráfica

A interface gráfica do programa é implementada usando a biblioteca Swing do Java. A classe principal responsável pela construção da interface é chamada "TelaInicial". A interface é organizada em uma janela principal onde o usuário pode interagir com os diferentes componentes. A janela principal contém os seguintes componentes:

1. Um campo de texto para inserir o caminho do arquivo de entrada.
2. Um botão "Selecionar" que abre um diálogo para escolher o arquivo de entrada.
3. Um campo de texto para inserir a capacidade do caminhão.
4. Dois botões para selecionar a técnica de solução: "Força Bruta" e "Branch and Bound".

Quando o usuário clica no botão "Selecionar", um diálogo de seleção de arquivo é exibido para permitir a escolha do arquivo de entrada. O caminho selecionado é exibido no campo de texto correspondente.

Após inserir o caminho do arquivo de entrada e definir a capacidade do caminhão, o usuário pode escolher entre as duas técnicas de solução disponíveis. Ao clicar em um dos botões ("Força Bruta" ou "Branch and Bound"), o programa executa a técnica selecionada usando as informações fornecidas pelo usuário. A imagem abaixo mostra como é a interface descrita acima:

Após a execução da técnica de solução, uma nova janela é aberta para exibir a animação do melhor caminho encontrado. Essa janela mostra as lojas visitadas, a

Figure 1. Interface Gráfica do Programa.

distância percorrida, o consumo de combustível e a lista de produtos coletados até o momento.

A tela de ”Animação do Caminho” é uma janela separada que é aberta após a execução de uma técnica de solução (Força Bruta ou Branch and Bound) na tela principal. Também é feita usando a biblioteca Swing do Java.

Essa janela é responsável por exibir a animação do melhor caminho encontrado pelo algoritmo. Ela fornece uma representação visual do percurso realizado pelo caminhão, mostrando as lojas visitadas e a ordem em que foram percorridas. A tela de animação do caminho possui os seguintes elementos:

1. Um painel principal onde a animação é desenhada.
2. Uma legenda que exibe informações sobre o estado atual da animação, como produtos no caminhão, distância percorrida e consumo atual de combustível.

A animação é atualizada em intervalos regulares para mostrar o progresso do caminhão ao longo do caminho. Cada iteração da animação representa o movimento do caminhão de uma loja para outra.

As lojas são representadas como círculos no painel, e cada loja é identificada por um número. O caminho percorrido pelo caminhão é mostrado por meio de linhas que conectam as lojas na ordem em que são visitadas.

Durante a animação, a legenda é atualizada para exibir informações relevantes sobre o estado atual da animação. Isso inclui a lista de produtos presentes no caminhão, a distância percorrida, o consumo atual de combustível e a lista de produtos restantes nas lojas.

A animação continua até que o caminhão tenha visitado todas as lojas no caminho. Uma vez que a animação é concluída, a janela pode ser fechada e o usuário pode realizar uma nova execução ou encerrar o programa. A tela abaixo mostra um exemplo de uma animação finalizada para uma entrada de 9 lojas:

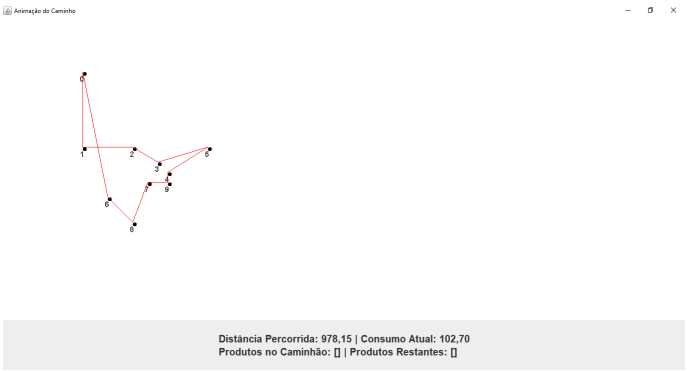


Figure 2. Animação do Caminho.

4. Relatório de testes

Antes de apresentar os testes é válido ressaltar que o tempo da animação da viagem e apresentação de interface gráfica não interfere no tempo do método analisado, ambos são analisados de forma separada.

Outro destaque importante é que nas permutações da Força Bruta, o tamanho das lojas é sempre desconsiderando a matriz, já que ela não faz parte da geração de permutações. O mesmo vale para o Branch and Bound.

Os testes são apresentados em tabelas e gráficos, que mostram o desempenho de cada método separadamente e em seguida é feito um comparativo das técnicas utilizadas. O tempo é medido em mili-segundos.

4.1. Força Bruta

Os resultados usando a abordagem por Força Bruta não foram satisfatórios. Como a complexidade é fatorial, rapidamente a viabilidade do método é colocada em cheque. As execuções foram limitadas a um tamanho de entrada de no máximo 10 a 11 lojas, qualquer valor acima o JDK acusava de falta de espaço na memória heap.

A tabela abaixo mostra uma média de um tempo de 10 execuções para diferentes instâncias do problema usando a abordagem por Força Bruta, para diferentes tamanhos de entrada e capacidade máxima do caminhão igual a 2. Em seguida é mostrado como está a disposição das lojas, melhor caminho encontrado e o consumo desse caminho.

Número de Lojas	Número de Produtos	Tempo em MS
2	1	2
4	3	2
6	4	14
8	5	234
9	7	886
10	8	8271
11	10	—-

Fonte: Elaborado pelo autor.

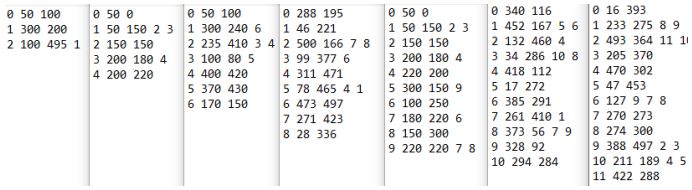


Figure 3. Disposição dos arquivos com listas de 2 a 11 lojas.

A lista abaixo mostra o melhor caminho e consumo de cada lista de lojas:

1. 2 Lojas: Rota encontrada: [0, 2, 1, 0] Consumo: 104.25
2. 4 Lojas: Rota encontrada: [0, 1, 2, 3, 4, 0] Consumo: 63.08
3. 6 Lojas: Rota encontrada: [0, 2, 3, 5, 4, 1, 6, 0] Consumo: 178.39
4. 8 Lojas: Rota encontrada: [0, 2, 7, 8, 5, 1, 3, 4, 6, 0] Consumo: 223.78
5. 9 Lojas: Rota encontrada: [0, 1, 2, 3, 5, 4, 9, 7, 8, 6, 0] Consumo: 102.70
6. 10 Lojas: Rota encontrada: [0, 3, 10, 8, 9, 7, 1, 6, 2, 5, 4, 0] Consumo: 287.87
7. 11 Lojas: Não executou com Força Bruta, apenas Branch and Bound.

A partir dos resultados é fácil notar os desafios da abordagem de força bruta para resolver o problema. Conforme o número de lojas aumenta, o tempo de execução cresce exponencialmente, tornando o algoritmo impraticável para instâncias maiores.

Outro fator que ficou evidenciado é que a carga máxima do caminhão e o número de produtos não têm diferença significativa no tempo de execução, apesar de implicarem em qual será a solução encontrada. O exemplo abaixo, considerando o arquivo com 8 lojas, mostra a rota encontrada para determinada entrada com capacidade máxima do caminhão igual a 2 e outra rota encontrada para a mesma entrada, dessa vez com capacidade máxima do caminhão igual a 3.

Rota com carga máxima 2: [0, 2, 7, 8, 5, 1, 3, 4, 6, 0], consumo: 223,78

Rota com carga máxima 3: [0, 2, 7, 5, 8, 1, 3, 4, 6, 0] consumo: 205,44

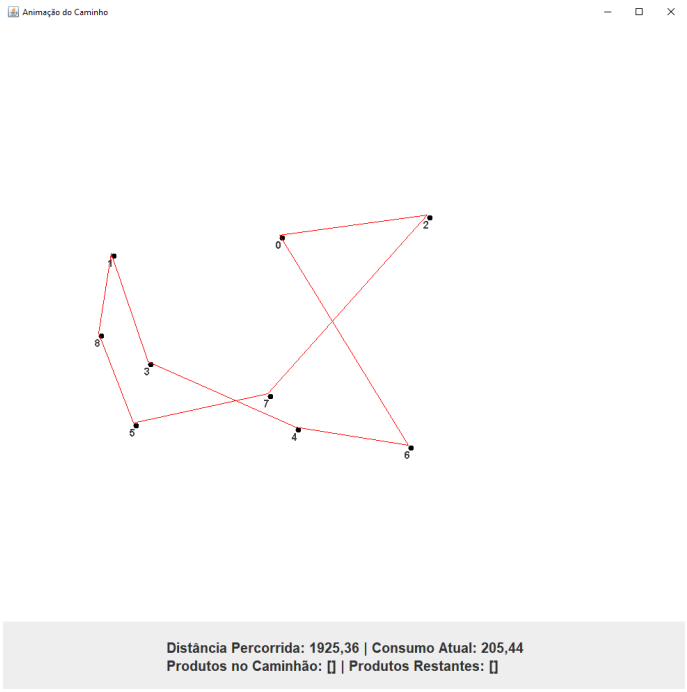
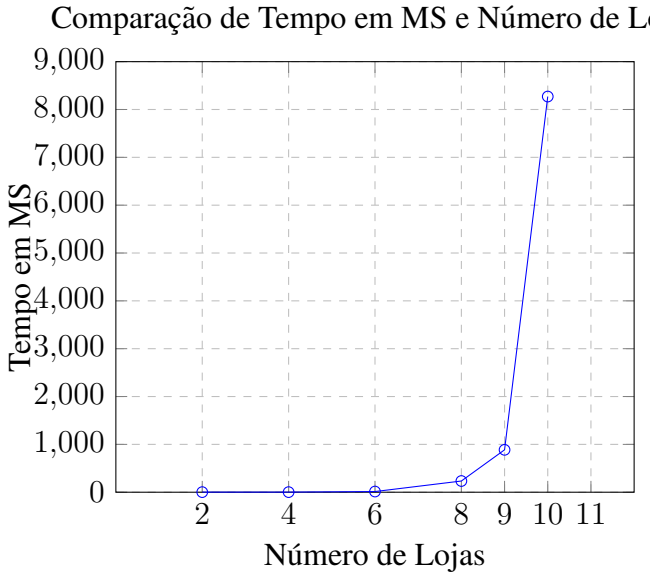


Figure 4. Ilustração do caminho percorrido com K = 3.

O gráfico abaixo mostra com clareza o caráter exponencial do tempo de execução em relação ao tamanho da entrada.



4.2. Branch and Bound

Os testes realizados com a abordagem de Branch and Bound mostraram que essa é uma técnica eficiente para resolver o problema proposto. A tabela de testes mostrada no método de Força Bruta, quando testada usando Branch and Bound a média de tempo para todos os tamanhos de entrada é de 1 mili-segundo, os caminhos encontrados bem como o consumo são iguais. O gráfico abaixo mostra um comparativo de tempo em mili-segundos entre as duas abordagens, deixando evidente a superioridade do método que usa a técnica de branch and bound.

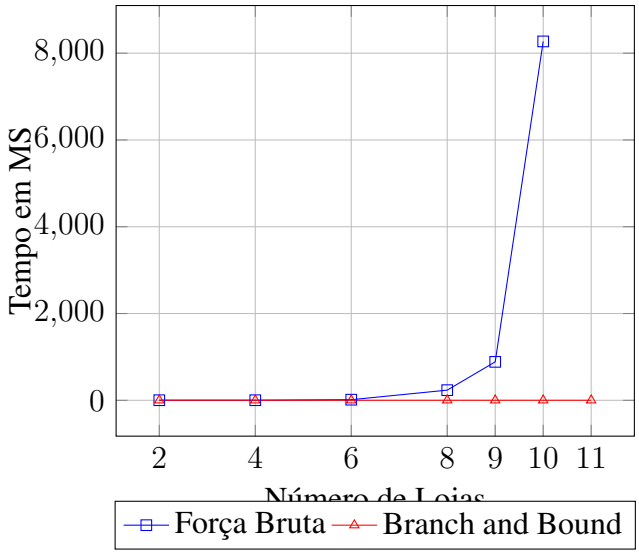


Figure 5. Comparação de Tempo de Execução entre Força Bruta e Branch and Bound

Como dito anteriormente, o desempenho de um método que usa essa abordagem está diretamente ligado a qualidade de seus limitantes inferiores ou superiores, como também as podas feitas durante o cálculo da solução.

O exemplo abaixo mostra que o fator que define o tempo de execução da abordagem por Branch and Bound não é só o tamanho da entrada em si, mas sim a qualidade das podas durante a execução.

```
0 400 400
1 320 120
2 189 283
3 176 341 1 2
4 234 472
5 134 164
6 451 361 5
7 153 174 4
8 421 341 9 10 11
9 125 153
10 163 131 12
11 114 261 13
12 123 423
13 165 165 14 15
14 412 235
15 500 324 21
16 321 413
17 313 414 16
18 500 153 17
19 312 413 20
20 316 313
21 234 164 22 23 24
22 125 120
23 421 261
24 316 134
```

```
0 400 400
1 320 120
2 189 283
3 176 341 1 2
4 234 472
5 134 164
6 451 361 5
7 153 174 4
8 421 341 9 10 11
9 125 153
10 163 131 12
11 114 261 13
12 123 423
13 165 165 14 15 16
14 412 235
15 500 324
16 499 330 17 18
17 231 161
18 163 151 19
19 123 152 20 21
20 155 412
21 312 441 22 23
22 413 145
23 321 354 24
24 315 317 25
25 361 397
```

Figure 6. Na esquerda a disposição do arquivo com 24 lojas e na direita com 25 lojas.

```
------------------------------
--Logs de Execução--
Melhor permutação encontrada: [0, 8, 10, 9, 7, 11, 12, 4, 3, 2, 13, 1, 14, 18, 15, 6, 17, 16, 19, 20, 5, 21, 22, 24, 23, 0]
Consumo da permutação: 341.81547653044373
Tempo em ms: 319537
------------------------------
--Logs de Execução--
Melhor permutação encontrada: [0, 8, 10, 9, 11, 12, 3, 2, 1, 13, 14, 15, 16, 6, 17, 18, 19, 5, 7, 20, 4, 21, 23, 24, 22, 25, 0]
Consumo da permutação: 355.34547409804594
Tempo em ms: 7724
------------------------------
```

Figure 7. Execução de um arquivo com 24 lojas e 25 lojas, respectivamente.

Como verificado, apesar de um número maior de lojas, o arquivo com 24 lojas obteve um tempo de execução consideravelmente maior que o de 25 lojas. Isso por que a qualidade de suas podas foram ruins, logo foi necessário o retorno e a reconfiguração do caminho diversas vezes.

5. Conclusão

A resolução eficiente de problemas complexos é um desafio recorrente na área da otimização. Ao lidar com o problema de distribuição de produtos em um cenário com múltiplas lojas e produtos, é crucial escolher a abordagem correta para obter soluções de alta qualidade dentro de um tempo computacional viável.

Neste trabalho, foram propostas duas soluções: a abordagem de força bruta e a abordagem de branch and bound. A escolha da técnica adequada é essencial para otimizar o tempo de execução e garantir a obtenção de soluções ótimas ou subótimas aceitáveis.

A força bruta é uma estratégia simples que consiste em enumerar todas as possíveis soluções do problema e selecionar a melhor dentre elas. Embora garanta a obtenção da solução ótima, seu principal inconveniente é a complexidade fatorial, que se torna rapidamente inviável à medida que o tamanho da instância aumenta. Portanto, a aplicação de força bruta em problemas de grande escala se torna impraticável devido ao tempo de execução excessivo.

Diante dessa limitação, o branch and bound surge como uma alternativa promissora. Essa abordagem baseia-se na exploração inteligente do espaço de soluções, utilizando técnicas de poda para eliminar ramos da árvore de busca que não levariam a soluções melhores do que a melhor solução encontrada até o momento.

A eficácia do branch and bound em relação à força bruta reside na sua capacidade de evitar a exploração desnecessária de soluções que são garantidamente piores ou idênticas às já encontradas. Essa estratégia de poda reduz significativamente o número de soluções a serem avaliadas, resultando em ganhos substanciais em termos de tempo de execução.

Portanto, considerando a complexidade do problema, a abordagem por branch and bound foi considerada superior à força bruta. Ao explorar de forma inteligente o espaço de soluções e utilizar técnicas de poda, o branch and bound demonstrou melhor eficiência em relação ao tempo de execução, permitindo a obtenção de soluções de qualidade em instâncias de tamanho considerável.

Em conclusão, este trabalho evidenciou a importância de selecionar a técnica correta para resolver o problema proposto. A abordagem por branch and bound foi identificada como superior à força bruta, devido à sua capacidade de reduzir o tempo de execução explorando de forma inteligente o espaço de soluções.