Pontifícia Universidade Católica de Minas Gerais

Fundamentos de Projeto e Análise de Algoritmos

Relatório do Trabalho Prático em Grupo

Gabriel Estevão Nogueira Sobrinho, Luana Fleury e Vinicius Gonzaga Guilherme

2023
Belo Horizonte

Informações

Critério de avaliação escolhido: **Individual**

Linguagem de programação: **Java**

Algoritmos implementados: **Divisão e Conquista, Backtracking e Programação Dinâmica**

Link do repositório: https://github.com/luanaf4/FPAAtrabalhopratico

Divisão de tarefas

Integrante	Tarefas executadas
Vinicius	• Implementação do método de Programação Dinâmica • Elaboração da seção Programação Dinâmica • Participação na seção de Comparação
Luana	• Implementação do método de Backtracking • Elaboração das seções Backtracking, Informações, Comparação entre Métodos e Considerações Finais • Participação na seção de Comparação
Gabriel	• Implementação do método de Divisão e Conquista • Elaboração da seção Divisão e Conquista • Participação na seção de Comparação

Sumário

Backtracking

Luana Fleury

Primeira implementação (Corresponde a Backtracking no repositório)

Estratégia Geral:

O código começa definindo várias variáveis, incluindo o número de caminhões, as rotas disponíveis e a quilometragem ideal que cada caminhão deve percorrer (calculada como a soma total da quilometragem das rotas dividida pelo número de caminhões). Em seguida, ele chama a função *distribuirRotas*, que realiza a distribuição das rotas.

Critério de Poda:

O critério de poda é que a quilometragem total de um caminhão após a adição de uma rota não deve exceder a quilometragem ideal e a diferença entre a maior e a menor quilometragem total dos caminhões deve ser menor do que a menor diferença encontrada até agora.

Passo a passo:

1. Inicializa as variáveis.
2. Começa a distribuição das rotas chamando a função `distribuirRotas`.
3. Verifica se todas as rotas foram atribuídas:
    - Se sim, verifica se a distribuição atual é melhor do que a melhor encontrada até agora. Se for, atualiza a melhor distribuição e as melhores rotas dos caminhões.
    - Se não, tenta adicionar a rota atual a cada caminhão, desde que a quilometragem total do caminhão não exceda a quilometragem ideal e a diferença entre a maior e a menor quilometragem dos caminhões seja menor do que a menor diferença encontrada até agora.
4. Se a rota puder ser adicionada a um caminhão, adiciona e chama a função recursivamente para a próxima rota.
5. Remove a rota do caminhão e continua o processo para o próximo caminhão.
6. Repita os passos 3-5 até que todas as rotas tenham sido atribuídas e todas as possibilidades tenham sido exploradas.
7. No final, a melhor distribuição de rotas e as melhores rotas dos caminhões serão armazenadas nas variáveis correspondentes.

Testes:

Teste 1

Conjunto de rotas 1 (fornecido na mensagem do Canvas):
40;36;38;29;32;28;31;35;31;30;32;30;29;39;35;38;39;35;32;38;32;33;29;33;29;39;28

Número de Caminhões:3

Resultado:

```
Melhor distribuição: [300, 300, 300]
Caminhão 1: 300 km
Rotas: 40 36 38 29 32 28 31 35 31
Caminhão 2: 300 km
Rotas: 30 32 30 29 39 35 38 39 28
Caminhão 3: 300 km
Rotas: 35 32 38 32 33 29 33 29 39
Tempo de execução: 1 ms / 1575000 ns
```

A distribuição de quilometragem entre os caminhões atingiu a quilometragem ideal. Devemos levar em consideração que a soma total da quilometragem é uma divisão exata por 3 (número de caminhões). Será discutido no tópico "Discussão" a razão de uma divisão exata ser mais "fácil" para a distribuição realizada pelo algoritmo.

Teste 2

Conjunto de rotas 2 (fornecido na mensagem do Canvas):
32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27

Número de Caminhões:3

Resultado: Como explicado na apresentação, o segundo Conjunto de Rotas não rodou nessa versão do algoritmo. Isso também é discutido no tópico "Discussão".

Teste 3

Gerador de Problemas: O teste começa com 6 rotas que aumentam de 1 em 1, 10 conjuntos de cada tamanho e três caminhões. O teste para ao exceder o limite de tempo de 30 segundos.

O número de caminhões e conjuntos permanece o mesmo, apenas as rotas aumentam.

Utilizei a dispersão de 0.5

Resultado:

```
Número de rotas: 21, Tempo médio de execução: 7138 ms / 7139140800 ns
Rotas: [18, 19, 13, 15, 15, 17, 16, 15, 13, 13, 18, 13, 15, 18, 17, 16, 19, 17, 19, 15, 19, 14]
Rotas: [16, 17, 13, 14, 17, 17, 13, 19, 15, 18, 14, 19, 13, 14, 16, 14, 15, 17, 17, 18, 17, 15]
Rotas: [18, 13, 16, 13, 19, 19, 13, 19, 19, 17, 15, 19, 15, 13, 15, 17, 13, 13, 16, 19, 14, 15]
Rotas: [15, 17, 19, 13, 19, 17, 15, 14, 16, 15, 18, 17, 14, 15, 14, 14, 16, 14, 13, 17, 13, 14]
Rotas: [15, 15, 13, 17, 14, 14, 14, 13, 19, 19, 17, 19, 15, 18, 16, 16, 14, 15, 18, 17, 14, 17]
Rotas: [14, 18, 16, 16, 19, 15, 14, 14, 15, 19, 13, 18, 13, 14, 13, 14, 15, 13, 18, 16, 18, 14]
Rotas: [15, 17, 19, 13, 19, 16, 16, 15, 18, 19, 16, 16, 17, 14, 13, 15, 18, 16, 14, 18, 13, 18]
Rotas: [14, 14, 13, 18, 14, 17, 17, 15, 18, 16, 15, 16, 16, 13, 15, 19, 16, 14, 14, 16, 18, 14]
Rotas: [17, 19, 17, 13, 13, 15, 16, 13, 19, 19, 13, 13, 17, 17, 19, 15, 16, 18, 16, 19, 15, 18]
Rotas: [16, 16, 13, 15, 18, 18, 19, 19, 16, 14, 13, 17, 13, 15, 14, 13, 14, 18, 17, 18, 15, 14]
Número de rotas: 22, Tempo médio de execução: 8856 ms / 8856840500 ns
Rotas: [13, 14, 15, 16, 18, 19, 19, 17, 14, 15, 15, 13, 18, 17, 13, 17, 15, 19, 18, 19, 15, 17, 18]
Rotas: [13, 13, 14, 18, 18, 17, 17, 17, 14, 14, 15, 16, 18, 18, 14, 18, 17, 16, 15, 15, 16, 17, 19]
Rotas: [17, 14, 13, 18, 19, 13, 16, 17, 18, 19, 15, 13, 17, 18, 16, 14, 15, 18, 15, 18, 16, 18, 16]
Rotas: [16, 15, 17, 17, 15, 15, 18, 19, 17, 13, 16, 18, 19, 14, 16, 18, 18, 16, 16, 17, 13, 14, 14]
Rotas: [14, 14, 19, 19, 15, 13, 18, 17, 19, 17, 19, 18, 17, 15, 14, 15, 17, 13, 19, 14, 15, 17, 18]
Rotas: [17, 19, 13, 13, 16, 13, 19, 13, 18, 16, 18, 19, 19, 15, 14, 18, 17, 15, 18, 16, 15, 13, 16]
Rotas: [16, 19, 18, 18, 16, 17, 16, 16, 17, 13, 13, 14, 18, 16, 17, 13, 14, 16, 13, 14, 14, 15, 13]
Rotas: [19, 17, 16, 15, 18, 13, 18, 19, 16, 15, 17, 15, 13, 18, 13, 17, 17, 18, 18, 16, 14, 15, 13]
Rotas: [16, 17, 14, 19, 16, 16, 13, 16, 17, 19, 19, 15, 15, 18, 19, 14, 15, 16, 16, 13, 13, 13, 17]
Rotas: [14, 14, 16, 18, 16, 17, 17, 18, 18, 15, 18, 19, 16, 13, 17, 13, 17, 18, 18, 15, 14, 14, 18]
Número de rotas: 23, Tempo médio de execução: 47887 ms / 47888120200 ns
```

O algoritmo rodou 10 conjuntos de 22 rotas em um tempo médio de execução de 8856 ms (8856840500 ns), último número de rotas que não excedeu o tempo limite de 30 segundos.

10 conjuntos de 23 rotas levaram 47887 ms (47888120200 ns) para rodar, ultrapassando o limite de tempo.

Segunda Implementação (Corresponde a Backtracking2 no repositório)

Estratégia Geral:

A estratégia geral segue a mesma linha de raciocínio da primeira implementação. A grande diferença entre os dois algoritmos é o critério de poda.

Critério de Poda:

if (caminhoes[i] <= *quilometragemIdeal + 1:* Verifica se a quilometragem total do caminhão atual, após adicionar a rota atual, não excede a quilometragem ideal por mais de 1.

1 foi o número que utilizei para ser considerado como "margem de tolerância", buscando facilitar a distribuição de rotas em que a quilometragem total não fosse uma divisão exata pelo número de caminhões.

A quilometragem ideal é a soma total das rotas dividida pelo número de caminhões.

(Arrays.stream(caminhoes).max().getAsInt() - Arrays.stream(caminhoes).min().getAsInt() < *menorDiferenca):* Calcula a diferença atual entre a maior e a menor quilometragem dos caminhões e verifica se ela é menor que a menor diferença encontrada até agora. Se a

diferença atual for maior que a menor diferença, então não há necessidade de continuar explorando essa distribuição, tendo em vista que ela não vai levar a uma melhor distribuição.

Passo a passo:

1. Inicializa as variáveis
2. Começa a distribuição das rotas chamando a função *distribuirRotas(0).*
3. Verifica se todas as rotas foram atribuídas:
    - Se sim, calcula a diferença entre a maior e a menor quilometragem dos caminhões. Se essa diferença for menor do que a menor diferença encontrada até agora, atualiza a menor diferença e a melhor distribuição.
4. Se nem todas as rotas foram distribuídas, tenta adicionar a rota atual a cada caminhão, um por vez:
    - Para cada caminhão, verifica se a adição da rota não fará com que a quilometragem total do caminhão exceda a quilometragem ideal + 1 e se a diferença atual entre a maior e a menor quilometragem dos caminhões é menor do que a menor diferença encontrada até agora.
    - Se ambas as condições forem atendidas, adiciona a rota ao caminhão, marca como melhor rota até o momento e chama a função *distribuirRotas* recursivamente para a próxima rota.
5. Após tentar adicionar a rota ao caminhão, remove a rota do caminhão e desmarca-a.
6. Repete os passos 3-5 até que todas as rotas tenham sido distribuídas e todas as possíveis distribuições tenham sido exploradas.
7. No final, a melhor distribuição e a menor diferença serão armazenadas nas variáveis correspondentes.

Testes

Teste 1

Conjunto de rotas 1 (fornecido na mensagem do Canvas):
40;36;38;29;32;28;31;35;31;30;32;30;29;39;35;38;39;35;32;38;32;33;29;33;29;39;28

Número de Caminhões:3

Resultado:

```
Melhor distribuição: [300, 301, 299]
Caminhão 1: 300 km
Rotas: 40 36 38 29 32 28 31 35 31
Caminhão 2: 301 km
Rotas: 30 32 30 29 39 35 38 39 29
Caminhão 3: 299 km
Rotas: 35 32 38 32 33 33 29 39 28
Tempo médio de execução: 0ms / 657000ns
```

A distribuição da quilometragem entre os caminhões não atingiu a quilometragem ideal, que, nesse caso, seria 300 km para cada caminhão. Apenas um dos caminhões atingiu

essa quilometragem, enquanto os outros dois receberam 301 km (caminhão 2) e 299 km (caminhão 3). A diferença entre os resultados do primeiro algoritmo e do segundo algoritmo serão discutidas mais à frente.

Teste 2

Conjunto de rotas 2 (fornecido na mensagem do Canvas):
32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27

Número de Caminhões:3

Resultado:

```
Melhor distribuição: [326, 327, 327]
Caminhão 1: 326 km
Rotas: 32 51 32 43 42 30 42 29 25
Caminhão 2: 327 km
Rotas: 51 43 51 27 32 29 43 24 27
Caminhão 3: 327 km
Rotas: 55 29 32 44 55 29 53 30
Tempo médio de execução: 30ms / 30791000ns
```

O segundo algoritmo conseguiu rodar o segundo conjunto de testes. A diferença entre a maior quilometragem total e a menor quilometragem total foi de apenas 1.

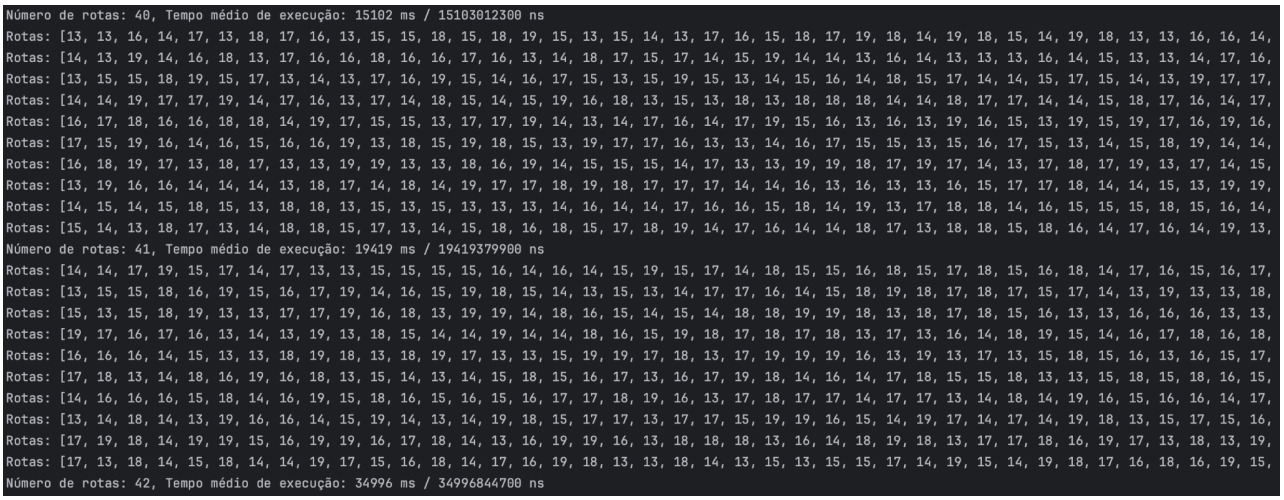
Teste 3

Gerador de Problemas: O teste começa com 6 rotas que aumentam de 1 em 1, 10 conjuntos de cada tamanho e três caminhões. O teste para ao exceder o limite de tempo de 30 segundos.

O número de caminhões e conjuntos permanece o mesmo, apenas as rotas aumentam.

Utilizei a dispersão de 0.5

Resultado:



O segundo algoritmo conseguiu chegar a 41 rotas dentro do período limite de 30 segundos, sendo seu tempo médio de execução 19419 ms (19419379900 ns). 42 rotas ultrapassou o limite por 4 segundos.

Quadro de Comparações (Backtracking e Backtracking2)

Teste	Algoritmo	Tempo de Execução (ms)/(ns)	Resultado
1	**1**	**1 ms** **1575000 ns**	**300,300,300** **Diferença entre maior e menor quilometragem: 0**
1	**2**	**0 ms** **657000 ns**	**300,301,299** **Diferença entre maior e menor quilometragem: 1**
2	**1**	**X**	**Não rodou**
2	**2**	**30 ms** **30791000 ns**	**326,327,327** **Diferença entre maior e menor quilometragem: 1**
3	**1**	**8856 ms** **88568440500 ns**	**22 rotas no tempo limite de 30 segundos**
3	**2**	**19419 ms**	**41 rotas no tempo limite de 30 segundos**

		19419379900 ns	

Análise Crítica dos resultados

Teste 1: O resultado do teste no algoritmo 1 alcançou um melhor resultado, chegando a quilometragem ideal de 300 km para cada caminhão, porém levou mais tempo que o algoritmo dois. O algoritmo dois, por sua vez, não alcançou a quilometragem ideal, mas se mostrou mais eficiente no tempo.

Teste 2: O algoritmo 1 não foi capaz de rodar o conjunto de teste 2. Já o algoritmo 2, rodou o conjunto em 30 ms / 30791000 ns e obteve uma diferença de 1.

A mudança do resultado se deve à alteração do critério de poda. Enquanto o algoritmo 1 apresenta um critério de poda mais rigoroso, o algoritmo 2 apresenta um mecanismo de "margem de tolerância", deixando a poda mais "tolerante". Esse mecanismo foi acrescentado para tornar o algoritmo mais eficiente em distribuições em que a quilometragem ideal não for um valor exato.

A ideia foi que, para casos de divisões não exatas, o algoritmo não ficasse tentando encontrar o valor exato da quilometragem ideal, tendo em vista que seria impossível, pois não poderíamos dividir uma mesma rota entre dois caminhões.

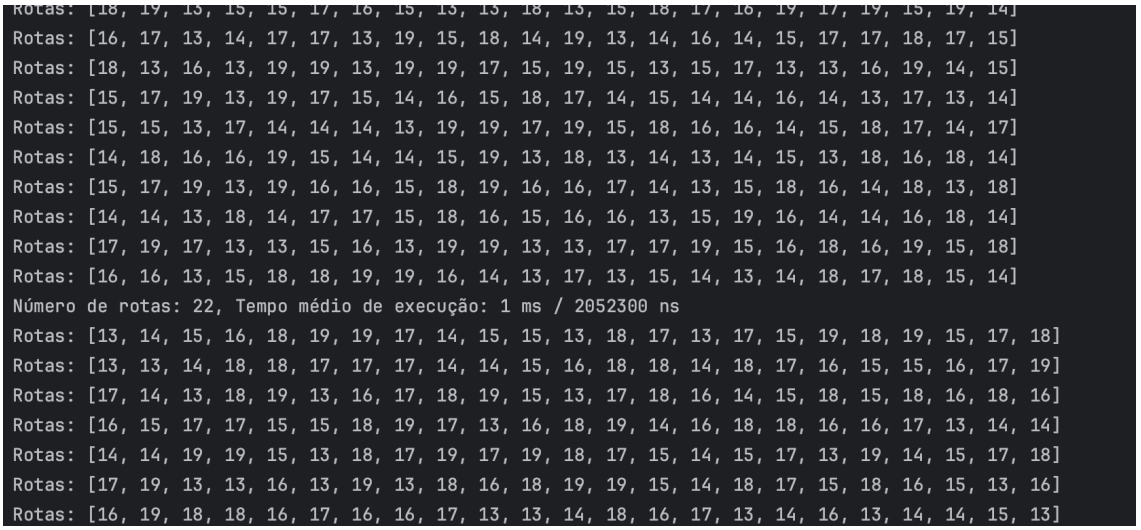
Acrescentei uma tolerância de quilometragemIdeal + 1 ("margem de erro") para que o algoritmo não ficasse buscando um valor que seria impossível de obter quando a divisão não fosse exata.

Essa mudança se mostrou eficaz, tendo em vista que no algoritmo 2 foi possível rodar os dois conjuntos de testes e em menor tempo.

A queda no tempo de execução também se deve à mudança no critério de poda, pois, por existir uma "margem de tolerância" no algoritmo 2 ele passa menos tempo tentando encontrar uma divisão que seja o valor exato da *quilometragemIdeal,* passando por todas as possibilidades e executando a poda mais rapidamente.

Teste 3 (Gerador de problemas): O algoritmo 2 se mostrou bem mais rápido que o algoritmo 1, chegando a 41 rotas dentro do tempo limite de 30 segundos, enquanto o algoritmo 1 obteve 22 rotas.

Para uma melhor análise, segue o tempo de execução do algoritmo 2 com 22 rotas:



O algoritmo 1 chegou ao resultado de 22 rotas no tempo de 8856 ms

/ 88568440500 ns. Enquanto isso, o algoritmo 2 levou 1 ms / 2052300 ns.

Novamente, o motivo da diferença no tempo de execução entre os dois algoritmos foi a mudança no critério de poda.

Considerações Finais:

No dia da apresentação, expliquei a lógica dos dois códigos e informei sobre a diferença de resultado entre os dois e sobre o problema que estava tendo para lidar com o tempo de execução do algoritmo 2.

Depois da apresentação, fiz mudanças no código 2, sendo a principal delas a mudança no critério de poda. Antes da mudança, o critério de poda estava assim:

```
if (caminhoes[i] + rotas[rotaAtual] <= quilometragemIdeal &&
                        caminhoes[i] <= quilometragemIdeal &&
                       (Arrays.stream(caminhoes).max().getAsInt()
- Arrays.stream(caminhoes).min().getAsInt() < menorDiferenca))
```

Com esse critério, estava demorando por volta de 30 a 40 minutos para rodar um teste. Decidi mudar, pois considerei o critério ineficiente devido ao tempo que o algoritmo estava levando para distribuir as rotas.

O critério atual é esse:

```
if (caminhoes[i] <= quilometragemIdeal + 1 &&
```

```
(Arrays.stream(caminhoes).max().getAsInt() -
Arrays.stream(caminhoes).min().getAsInt() < menorDiferenca))
```

Na primeira versão, o critério é que a quilometragem total de um caminhão após a adição de uma rota e sua quilometragem atual não podem ultrapassar a quilometragem ideal **(total de quilometragem / número de caminhões).** Além disso, a a diferença entre a maior e a menor quilometragem total dos caminhões deve ser menor do que a menor diferença encontrada até agora.

Já na segunda versão (versão atual), a quilometragem total de um caminhão pode exceder a quilometragem ideal em 1. Isso permite uma maior flexibilidade na distribuição das rotas, especialmente quando a soma total da quilometragem não é uma divisão exata pelo número de caminhões. A segunda parte do critério de poda permanece a mesma do primeiro algoritmo: a diferença entre a maior e a menor quilometragem total dos caminhões deve ser menor do que a menor diferença encontrada até agora.

Por fim, conclui que a diferença no critério resultou em diferentes comportamentos para os algoritmos e, por consequência, diferentes resultados.

O código 1 permaneceu o mesmo, fora mudanças de estrutura (para adicionar a impressão do tempo de execução em **ns**) e adição do Javadoc.

Conclusão:

Como já citado, o primeiro algoritmo é mais rigoroso e só aceita soluções que atendam ao valor exato da quilometragem atual, sendo impossível considerar casos que não possuem uma divisão exata entre quilometragem total e número de caminhões. O segundo algoritmo é mais flexível e aceita soluções que estão próximas da quilometragem ideal, sendo possível considerar casos de divisões não exatas.

Portanto, considero o algoritmo 2 mais eficiente. Por mais que não consiga chegar ao valor exato da quilometragem ideal, consegue se manter entre a "margem de tolerância" de 1 e apresenta uma velocidade maior. O algoritmo faz uma distribuição que atende aos requisitos: *"A empresa deseja fazer a distribuição de maneira que cada caminhão cumpra a mesma quilometragem, evitando assim que ao final do período existam caminhões ociosos enquanto outros ainda estão executando várias rotas. Se não for possível cumprir a mesma quilometragem, que a diferença entre a quilometragem dos caminhões seja a menor possível, diminuindo o problema."*

A diferença entre a quilometragem dos caminhões se manteve na menor possível nos dois testes de conjuntos de rotas realizados, sendo essa diferença 1.

Divisão e conquista

Gabriel Estevão

Primeira implementação (Corresponde a DivisaoConquista.java no repositório)

Estratégia Geral:

Anteriormente, havia sido criado um algoritmo que recebia as rotas a serem distribuidas, e depois de ordenadas eram divididas recursivamente pela função dividir, com base na média de quilometragem. A divisão continuava até que a soma das distâncias ultrapasse a média, momento em que a rota é atribuída ao próximo caminhão. No entanto, o algoritmo não seguia completamente a técnica de Divisão e Conquista, pois negligenciou a segunda fase do algoritmo, que é responsável por combinar as menores soluções previamente divididas para produzir uma solução para a instância original.

Testes:

Teste 1:

Conjunto de testes:

40;36;38;29;32;28;31;35;31;30;32;30;29;39;35;38;39;35;32;38;32;33;29;33;29;39;28

Quantidade de caminhões: 3

Resultado:

```
Quantidade de rotas geradas: 27
Caminhão 1: 294Km
Caminhão 2: 299Km
Caminhão 3: 267Km
Tempo de execução: 9050600 nanossegundos.
```

Teste 2:

Conjunto de testes:

Resultados:
32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27

O algoritmo realizou o teste de maneira ágil. Embora tenha havido uma diferença considerável na distribuição entre os caminhões 2 e 3, foi notável o equilíbrio na distribuição das rotas entre os caminhões 1 e 2.

```
Quantidade de rotas geradas: 26
Caminhão 1: 311Km
Caminhão 2: 309Km
Caminhão 3: 305Km
Tempo de execução: 9272100 nanossegundos.
```

É observável que a distribuição foi excepcionalmente equilibrada, apresentando uma mínima diferença geral de apenas 6 quilômetros.

Segunda implementação (Corresponde a DivisaoEConquista.java no repositório):

Estratégia geral:

O algoritmo começa calculando a soma total das rotas. Em seguida, o algoritmo itera sobre o conjunto de rotas e, usando o operador módulo, atribui cada rota ao próximo caminhão na sequência, garantindo uma distribuição inicial parecida das rotas entre os caminhões.

Depois disso, o algoritmo divide o conjunto de rotas atribuídas a cada caminhão em dois subconjuntos e processa cada um separadamente de forma recursiva, onde cada subconjunto é continuamente dividido até que haja apenas uma rota em cada subconjunto(o caso base).

E finalmente, o algoritmo cria um novo array com tamanho combinado dos dois arrays de entrada e copia os elementos de cada array de entrada para este novo array. O resultado é a combinação das soluções.

Passo a passo:

1. O método `main` um array chamado de inteiros com um tamanho igual ao número de caminhões.
    a. Cada rota é atribuída a um caminhão, distribuindo as rotas igualmente entre os caminhões.
2. O metódo `divisaoConquista` é chamado recebe um array de inteiros, e o divide em dois subarrays (`rotas1` e `rotas2`), e chama a si mesma para resolver cada subarray e em seguida, combina as soluções utilizando o método combinar.
3. O método `combinar` recebe os dois subconjuntos, ele cria um novo array (`rotas`) com o tamanho combinado dos dois subconjuntos e copia os elementos para esse novo conjunto que em seguida, é retornado.

Teste 1:

Conjunto de testes:

40;36;38;29;32;28;31;35;31;30;32;30;29;39;35;38;39;35;32;38;32;33;29;33;29;39;28

Quantidade de caminhões: 3

Resultado:

```
Caminhão 1: 291 km
Caminhão 2: 319 km
Caminhão 3: 290 km
Tempo de execução: 27500 nanossegundos.
```

O algoritmo apresentou resultados notáveis. Apesar de haver uma diferença geral de 29 quilômetros na quilometragem distribuída entre os caminhões 2 e 3, é notável o equilíbrio na distribuição das rotas entre os caminhões 1 e 3.

Teste 2:

Conjunto de testes:

Resultados:
32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27

```
Caminhão 1: 335 km
Caminhão 2: 372 km
Caminhão 3: 273 km
Tempo de execução: 24400 nanossegundos.
```

Diferente do primeiro teste, o algoritmo encontrou dificuldade para distribuir as rotas de maneira equitativa entre os caminhões, obtendo uma diferença geral de 99 quilômetros.

Teste utilizando o gerador de problemas:

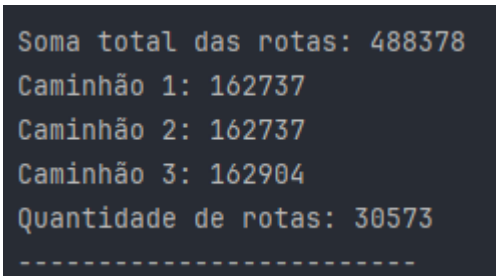
Tamanho de rotas inicial: 22 (tamanho máximo limite do backtracking)

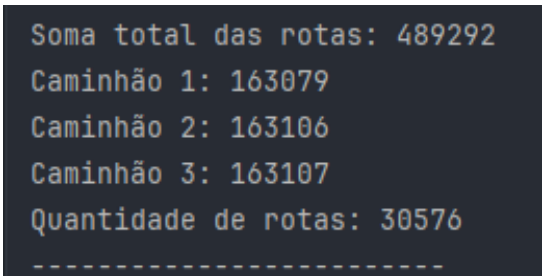
Resultados:

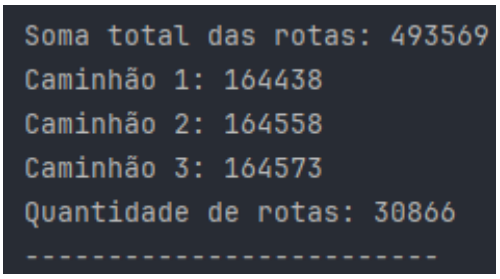
A medição do tempo de execução foi realizado utilizando o método `System.nanoTime()`. Assim que o tempo limite de 30 segundos é atingido, a execução é finalizada.

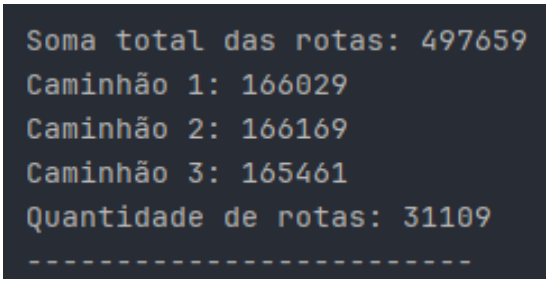
O algoritmo foi capaz de distribuir, em até 30 segundos, conjuntos de testes de até 31327, o que chegou a ser uma surpresa, considerando um limite máximo extremamente elevado para apenas 30 segundos.

Note que em algumas situações, obtiveram resultados gratificantes:

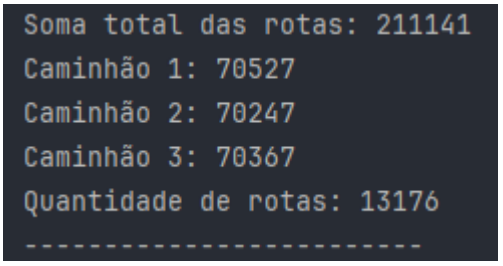


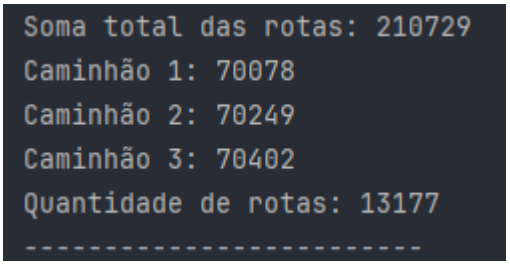






Em contraste, tiveram resultados minimamente razoáveis:





Discussão:

É importante destacar os resultados dos dois primeiros testes de ambos os algoritmos. A primeira implementação do algoritmo apresentou desempenho excelente. Embora não tenha alcançado a distribuição ideal, sua eficiência superou a da segunda implementação, que obteve resultados inferiores nos mesmos conjuntos de teste. Um fator possível para essa superioridade pode ser a simplicidade com que a primeira implementação atribui as rotas aos caminhões, já que a segunda implementação utiliza meios de combinação de subproblemas para uma solução global sem uso de operações ou cálculos adicionais além da concatenação de subconjuntos.

Conclusão:

Ambas as implementações do algoritmo apresentaram resultados notáveis, embora com diferenças significativas. A primeira implementação, apesar de não seguir estritamente a técnica de Divisão e Conquista, demonstrou excelente desempenho e eficiência superior à segunda implementação nos mesmos conjuntos de teste. Isso pode ser atribuído à sua abordagem simplificada de atribuição de rotas aos caminhões.

Por outro lado, a segunda implementação, que emprega uma estratégia de Divisão e Conquista mais rigorosa, incluindo a combinação de subproblemas para uma solução global, mostrou-se menos eficiente nos testes realizados. No entanto, conseguiu distribuir um número surpreendentemente alto de rotas dentro do limite de tempo de 30 segundos, destacando seu potencial para lidar com grandes volumes de dados.

Esses resultados ressaltam a importância de escolher a estratégia de algoritmo apropriada com base nas características específicas do problema e dos dados em questão.

Programação Dinâmica

Vinícius Gonzaga Guilherme

Estratégia Geral:

O código começa definindo os dados que serão utilizados para a divisão de subconjuntos de rotas para a quantidade de caminhões existentes.
Após issó será calculado a quilometragem ideal para cada caminhão a partir da soma dos valores das rotas em um conjunto dividido pela quantidade de caminhões no problema.
Após isso para cada conjunto de rotas havia uma iteração contínua sobre a quantidade de caminhões, onde para cada caminhão se executava o método `execute()` da classe `ProgramacaoDinamica`. Esse método que é responsável por gerar a tabela de valores para ser utilizada no decorrer do algoritmo.
Quanto gerada, é chamado o método `findRouteSet()` responsável por encontrar os valores dentro do conjunto de rodas que somados seja igual ao valor ideial descoberto no início do algoritmo.

Características Do Algoritmo:

Para resolver o problema com uma solução através da programação dinâmica eu fiz a criação e utilização de uma tabela para memorização das de soluções que já foram feitas anteriormente, onde as linhas são as rotas de um conjunto inteiro e as colunas da tabela são os somatórios.

Passo a passo:

1. Definição dos valores das rotas, conjuntos e caminhões.
2. Para cada conjunto de rotas faça:
    a. Somatório das rotas dividido pela quantidade de caminhões para encontrar o valor ideal para cada caminhão.
    b. para cada caminhão faça:
        i. Criação da tabela de memória utilizando o valor ideal para um caminhão e a quantidade de rotas.
        ii. Preenchimento da tabela com valores inválidos para representá-la vazia.
        iii. Busca pelas rotas no conjunto de rotas para atingir o valor mais próximo possível do valor ideal para cada caminhão.
        iv. Remoção das rotas que foram adicionadas ao caminhão do conjunto geral de rotas.
    c. Finalização do cronômetro e impressão dos resultados.

Testes:

Conjunto de testes:

40;36;38;29;32;28;31;35;31;30;32;30;29;39;35;38;39;35;32;38;32;33;29;33;29;39;28

Quantidade de caminhões: 3

Resultado:

```
Tempo de execução: 23ms
Tempo de execução: 23000000ns
--------------------------------------------------
Conjunto resultado do Caminhão 1: [31, 35, 31, 28, 32, 29, 38, 36, 40]
Soma total das rotas: 300

Conjunto resultado do Caminhão 2: [32, 35, 38, 35, 39, 29, 30, 32, 30]
Soma total das rotas: 300

Conjunto resultado do Caminhão 3: [28, 39, 29, 33, 29, 33, 32, 38, 39]
Soma total das rotas: 300
```

Para esse conjunto de teste o algoritmo foi capaz de encontrar uma solução perfeita, encontrando um conjunto de rotas que possui a soma igual a melhor opção para um caminhão para todos os caminhões. Dessa forma encontrando uma diferença igual a zero para os subconjuntos selecionados.

Conjunto de testes:

32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27

Quantidade de caminhões: 3

Resultado:

```
Tempo de execução: 41ms
Tempo de execução: 41000000ns
--------------------------------------------------
Conjunto resultado do Caminhão 1: [25, 43, 51, 42, 30, 42, 43, 51]
Soma total das rotas: 327

Conjunto resultado do Caminhão 2: [29, 43, 55, 29, 27, 29, 51, 32, 32]
Soma total das rotas: 327

Conjunto resultado do Caminhão 3: [27, 24, 30, 53, 29, 55, 44, 32, 32]
Soma total das rotas: 326
```

Para essa segunda leva de rotas, não foi possível encontrar um subconjunto de rotas perfeito para todos os caminhões. O motivo disso é porque a soma de todas as rotas não dará um número exatamente perfeito para se dividir com a quantidade de caminhões, dessa forma havendo um resto na divisão.

Conjunto de testes:

Utilizando os dados adquiridos no algoritmo de Backtracking: “O algoritmo rodou 10 conjuntos de 22 rotas em um tempo médio de execução de 8856 ms (8856840500 ns), último número de rotas que não excedeu o tempo limite de 30 segundos.” será feito uma onda de 10 testes consecutivos para conjuntos de rotas gerados aleatoriamente. Para cada conjunto de rotas, o algoritmo será executado 10 vezes no intuito de se calcular uma média do tempo de execução para encontrar a solução do problema. Além disso a cada iteração, a quantidade

de rotas será multiplicado pelo número da iteração, fazendo com que a quantidade de rotas T vá aumentando de T em até chegar à um limite de 10T.

Quantidade de caminhões: 3

Resultado:

```
Tempo médio de execução com 1T: 1000000ns
Tempo médio de execução com 2T: 3000000ns
Tempo médio de execução com 3T: 8000000ns
Tempo médio de execução com 4T: 10000000ns
Tempo médio de execução com 5T: 21000000ns
Tempo médio de execução com 6T: 32000000ns
Tempo médio de execução com 7T: 31000000ns
Tempo médio de execução com 8T: 44000000ns
Tempo médio de execução com 9T: 60000000ns
Tempo médio de execução com 10T: 68000000ns
```

Pode-se perceber que o algoritmo de programação dinâmica é de extrema eficiência já que o teste inicial nessas condições demorou cerca de 8 minutos em uma solução utilizando algoritmo de Backtracking. A medida que a quantidade de rotas foi aumentando, o algoritmo acabou levando mais tempo para a execução, mas a diferença de tempo de execução em relação ao tamaho da entrada de dados que foi adicionada, pode-se dizer que o custo-benefício é muito grande para sua utilização.

Analisando os Valores dos Resultados:

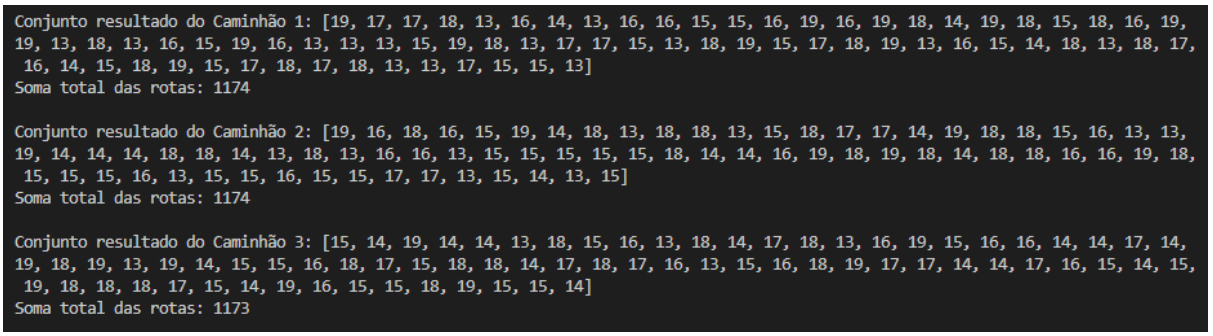
Ao analisar os valores encontrados em diferentes resultados com diferentes entradas pode-se perceber que o algoritmo mantêm uma enorme eficiência para encontrar os melhores resultados possíveis.

```
Conjunto resultado do Caminhão 1: [18, 18, 17, 13, 17, 13, 19]
Soma total das rotas: 115

Conjunto resultado do Caminhão 2: [13, 15, 14, 14, 15, 15, 13, 16]
Soma total das rotas: 115

Conjunto resultado do Caminhão 3: [16, 17, 18, 18, 14, 16, 16]
Soma total das rotas: 115
```

Nesse conjunto com poucos números o algortimo encontrou as soluções perfeitas para cada caminhão.



Já nessa iteração, o algoritmo não encontrou a solução perfeita para um dos caminhões, havendo uma diferença de 1 U.M. Porém o motivo dessa imperfeição se dá por conta da soma das rotas do conjunto inteiro, dividido pela quantidade de caminhões não dá um número exato, fazendo com que dessa forma uma das entidades fique com um valor diferente.

Conclusão:

Em resumo, o algoritmo de programação dinâmica demonstrou uma ótima eficiência nos testes. Ele conseguiu lidar com diferentes conjuntos de rotas e números de conjuntos de rotas, encontrando soluções ótimas soluções ou soluções perfeitas na maioria dos casos. Comparado com o algoritmo de Backtracking, o algoritmo de Programação Dinâmica mostrou ser um incomparável em quesito agilidade, sendo muito mais rápido, principalmente conforme o tamanho dos dados que aumentou durante a execução do terceiro teste..

Nos testes em que não conseguimos uma solução perfeita para todos os caminhões, entendemos que são casos em que não existe uma solução matemática de fato.

Comparações entre métodos

Valores de referência:

- Conjunto de rotas: 40, 36, 38, 29, 32, 28, 31, 35, 31, 30, 32, 30, 29, 39, 35, 38, 39, 35, 32, 38, 32, 33, 29, 33, 29, 39, 28
- Quantidade de caminhões: 3

Estratégia	Tempo de execução	Qualidade de resultados	Distribuição (km /caminhão)
Backtracking	**1 ms** **1575000 ns**	A distribuição de quilometragem entre os caminhões atingiu a quilometragem ideal. Devemos levar em consideração que a soma total da quilometragem é uma divisão exata por 3 (número de caminhões) e que o critério de poda do algoritmo 1 só é eficiente em divisões exatas.	Caminhão 1: 300 Caminhão 2: 300 Caminhão 3: 300

Backtracking2	**0 ms** **657000 ns**	A distribuição da quilometragem entre os caminhões não atingiu a quilometragem ideal, que, nesse caso, seria 300 km para cada caminhão. Apenas um dos caminhões atingiu essa quilometragem, enquanto os outros dois receberam 301 km (caminhão 2) e 299 km (caminhão 3). A diferença entre os resultados é causada pela alteração no critério de poda (Tópico discutido mais a fundo na seção de Backtracking)	Caminhão 1: 300 Caminhão 2: 301 Caminhão 3: 299
Divisão e Conquista	**27500 ns**	O algoritmo apresentou resultados notáveis. Apesar de haver uma diferença geral de 29 quilômetros na quilometragem distribuída entre os caminhões 2 e 3, é notável o equilíbrio na distribuição das rotas entre os caminhões 1 e 3.	Caminhão 1: 291 Caminhão 2: 319 Caminhão 3: 290
Programação Dinâmica	**23 ms** **23000000ns**	Para esse conjunto de teste, o algoritmo foi capaz de encontrar uma solução perfeita, encontrando um conjunto de rotas que possui a soma igual a melhor opção para um caminhão para todos os caminhões. Dessa forma encontrando uma diferença igual a zero para os subconjuntos selecionados.	Caminhão 1: 300 Caminhão 2:300 Caminhão 3: 300

Valores de referência:

- Conjunto de rotas:
  32;51;32;43;42;30;42;51;43;51;29;25;27;32;29;55;43;29;32;44;55;29;53;30;24;27
- Quantidade de caminhões: 3

Estratégia	Tempo de execução	Qualidade de resultados	Distribuição (km /caminhão)
Backtracking	**X**	**Não rodou (Motivo e análise na seção Backtracking)**	Caminhão 1: 0 Caminhão 2: 0 Caminhão 3: 0

Backtracking2	**30 ms** **30791000 ns**	O algoritmo 2, rodou o conjunto em 30 ms / 30791000 ns e obteve uma diferença de 1. Esse conjunto de rotas não possui uma divisão exata para três caminhões, ou seja, não seria possível dividir as rotas igualmente entre eles. Tendo isso em vista, a diferença de 1 é a mínima que poderia ser obtida.	Caminhão 1: 326 Caminhão 2: 327 Caminhão 3: 327
Divisão e Conquista	**24400 ns**	Diferente do primeiro teste, o algoritmo encontrou dificuldade para distribuir as rotas de maneira equitativa entre os caminhões, obtendo uma diferença geral de 99 quilômetros.	Caminhão 1: 335 Caminhão 2: 372 Caminhão 3: 273
Programação Dinâmica	**41 ms** **41000000 ns**	Para essa segunda leva de rotas, não foi possível encontrar um subconjunto de rotas perfeito para todos os caminhões. O motivo disso é porque a soma de todas as rotas não dará um número exatamente perfeito para se dividir com a quantidade de caminhões, dessa forma havendo um resto na divisão	Caminhão 1: 327 Caminhão 2:327 Caminhão 3: 326

Considerações Finais

Analisando os resultados obtidos nos dois conjuntos de testes, podemos perceber que cada algoritmo apresenta suas vantagens e desvantagens dependendo do conjunto de dados e do objetivo.

Para o primeiro conjunto de rotas, em que a divisão é exata, o algoritmo de Backtracking apresentou um excelente resultado, com uma quilometragem ideal para cada caminhão. O mesmo acontece com o algoritmo de Programação Dinâmica, que também conseguiu encontrar uma solução perfeita. Já o algoritmo Backtracking2, se mostrou levemente menos eficiente, apresentando uma diferença de 1 km. No entanto, o algoritmo de Divisão e Conquista não foi tão eficiente, apresentando uma diferença de 29 km na distribuição.

O segundo conjunto de rotas, em que a divisão não é exata, o algoritmo de Backtracking não conseguiu rodar, enquanto o Backtracking 2 obteve uma diferença mínima, que, nesse caso, era o melhor resultado possível. A Programação Dinâmica também obteve um bom desempenho, igualando sua diferença à diferença do backtracking, mas levando mais tempo para concluir a distribuição. O algoritmo de Divisão e Conquista, por outro lado, apresentou uma grande diferença na distribuição das rotas, não sendo eficiente para essa situação.

Em termos de tempo de execução, o algoritmo de Divisão e Conquista foi o mais rápido em ambos os conjuntos de rotas, mas, em contrapartida, apresentou a maior diferença de quilometragem em ambos os casos. Por mais que a Programação Dinâmica e o Backtracking/Bakctracking2 levem mais tempo para distribuir as rotas, eles entregam resultados melhores, que chegam mais perto do objetivo de dividir a quilometragem igualmente entre os caminhões.

Portanto, a escolha do algoritmo deve ser feita considerando o objetivo da tarefa. Se a prioridade for encontrar a distribuição mais adequada possível (que é o caso deste trabalho), os algoritmos de Programação Dinâmica podem ser a melhor escolha. Porém, se alcançar o menor tempo de execução for o objetivo, o algoritmo de Divisão e Conquista pode ser mais adequado.

Observações importantes: Os dados das duas tabelas foram retirados de suas respectivas seções no relatório. (Autoria de cada seção se encontra na página de informações desse relatório)
As considerações finais foram feitas com base nos dados da tabela