PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Trabalho prático FPAA - Leilão Energia

Integrantes:
Gabriel Alejandro Figueiro Galindo
Gabriel Vitor de Oliveira Morais
Lucas Hemétrio Teixeira
Pedro Henrique Moreira Caixeta Ferreira
Professor: João Caram Santos de Oliveira

Divisão de Tarefas: Teve-se uma colaboração de todo grupo ao implementar o main e cada membro implementou uma classe.

Backtracking: Pedro Henrique

Guloso: Gabriel Vitor

Divisão e Conquista: Gabriel Alejandro

Programação Dinâmica: Lucas Hemétrio

Relatório: Grupo

Conjunto de lances - https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Lance.java

Atributos:

energia: um inteiro que representa a quantidade de energia do lance em megawatts (MW).

valor: um inteiro que representa o valor monetário oferecido pelo lance.

Construtor Lance(int energia, int valor): Este é o construtor da classe. Ele é usado para criar novos objetos do tipo Lance.

Recebe como parâmetros a energia e o valor do lance e inicializa os atributos correspondentes do objeto.

Métodos Getters:

getEnergia(): retorna o valor do atributo energia do lance.

getValor(): retorna o valor do atributo valor do lance.

Método toString(): Este método sobrescreve o método toString() da classe Object em Java.

Ele retorna uma representação em forma de string do objeto Lance.

No caso, a string segue o formato: "Lance{energia=valorDaEnergia, valor=valorDoLance}".

Exemplo de Uso:

Lance lance1 = new Lance(50, 1000); // Cria um lance de 50 MW por 1000 unidades monetárias.

System.out.println(lance1.getEnergia()); // Imprime: 50

System.out.println(lance1.getValor()); // Imprime: 1000

System.out.println(lance1); // Imprime: Lance{energia=50, valor=1000}

A classe Lance fornece uma estrutura simples para representar lances em um sistema que lida com ofertas de energia. Ela armazena a energia e o valor de cada lance e oferece métodos para acessar essas informações, além de fornecer uma representação textual do objeto.

Conjunto gerador de dados - gerarConjuntoLances

Essa função tem como objetivo gerar uma lista de objetos do tipo Lance, que representam lances em um leilão de energia, provavelmente.

Funcionalidades:

Parâmetros:

tamanho: um inteiro que define quantos lances aleatórios devem ser gerados.

random: um objeto da classe Random usado para gerar números aleatórios.

Variáveis:

lances: uma lista vazia que armazenará os lances gerados.

Laço for: Itera tamanho vezes, criando um novo lance a cada iteração.

energia: um valor aleatório entre 1 e 1000, representando a energia do lance em MW.

valor: um valor aleatório entre 1 e 2000, representando o valor monetário do lance.

lances.add(...): um novo objeto Lance é criado com a energia e valor gerados e adicionado à lista lances.

Retorno: A função retorna a lista lances preenchida com os lances gerados aleatoriamente.

- A função gerarConjuntoLances simula a criação de um conjunto de lances de energia com valores aleatórios para energia e valor monetário.

Final do main:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Main.java

Segunda Função - duplicateListOfLists

Essa função é mais genérica e trabalha com listas de listas de qualquer tipo (T). Seu objetivo é criar uma cópia profunda de uma lista de listas.

Funcionalidades:

Parâmetros: originalList: a lista de listas original que será duplicada.

Variáveis: newList: uma nova lista vazia que armazenará a cópia da lista original.

Laço for each: Itera sobre cada sublista (sublist) dentro da originalList.

newSublist: cria uma nova lista que é uma cópia da sublist atual. É importante destacar que new ArrayList<>(sublist) cria uma nova lista com os mesmos elementos da sublist, mas não copia os objetos em si. Se os objetos dentro da sublista forem mutáveis e você modificá-los na cópia, alteração também será refletida na lista original.

newList.add(...): a nova sublista newSublist é adicionada à newList.

Retorno: A função retorna a newList, que contém uma cópia de todas as sublistas da originalList.

A função duplicateListOfLists cria uma nova lista contendo cópias das sublistas da lista original. É importante lembrar que essa cópia é superficial em relação aos objetos dentro das sublistas.

Observações:

O código assume que existe uma classe chamada Lance definida em outro lugar no projeto, que possui um construtor que aceita energia e valor como parâmetros.

A função duplicateListOfLists pode ser útil em situações em que você precisa modificar uma lista de listas sem alterar a lista original.

Final do main:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Main.java

Algoritmo Backtracking

Decisão de usar Backtracking: O problema apresentado é uma variação do problema da mochila 0/1, que é um problema clássico de otimização combinatória. A ideia é escolher um subconjunto de lances que maximize o valor total sem exceder a energia total disponível. O backtracking é uma abordagem adequada para esse tipo de problema porque ele explora todas as possíveis combinações de lances para encontrar a melhor solução.

Funcionamento do Backtracking: O algoritmo de backtracking funciona construindo progressivamente candidatos à solução e abandonando um candidato assim que determina que esse candidato não pode ser estendido para uma solução válida. No nosso caso, o algoritmo tenta incluir cada lance na solução e faz uma chamada recursiva para resolver o problema restante. Se a inclusão do lance leva a uma solução inválida (ou seja, a energia total dos lances excede a energia total disponível), o algoritmo volta atrás e tenta a próxima opção.

Ordenação dos Lances: Antes de iniciar o backtracking, os lances são ordenados em ordem crescente de energia. Isso é feito para otimizar o processo de backtracking, pois permite ao algoritmo considerar primeiro os lances que usam menos energia.

Estratégia de Poda: A estratégia de poda utilizada não se baseia em uma verificação explícita de se a energia do lance atual é maior que a energia restante. Em vez disso, a poda ocorre implicitamente devido à ordenação prévia dos lances por energia em ordem crescente.

Como os lances estão ordenados e a energia restante diminui ao longo da recursão, se em algum momento a energia restante for menor que a energia do lance atual, o algoritmo identifica que todos os lances subsequentes também terão energia insuficiente. Nesse ponto, a recursão retorna sem explorar as próximas ramificações, realizando a poda e evitando a análise de soluções inviáveis.

Cálculo da Energia Total Vendida e do Melhor Valor: Depois que o algoritmo de backtracking termina, ele calcula a energia total vendida e o melhor valor somando a energia e o valor dos lances na melhor combinação encontrada.

Medição do Tempo de Execução: O tempo de execução do algoritmo é medido usando a classe System do Java. O tempo antes e depois da execução do algoritmo é registrado e a diferença entre esses dois tempos é o tempo de execução do algoritmo.solução ótima.

Vantagens:

1. Solução completa: O backtracking garante que você encontrará a solução ótima se ela existir, porque explora todas as possíveis combinações de lances.
2. Simplicidade: O algoritmo de backtracking é relativamente simples de entender e implementar. Ele basicamente envolve uma busca em profundidade na árvore de decisões do problema.
3. Flexibilidade: O backtracking pode ser aplicado a uma ampla gama de problemas de otimização e pode ser facilmente adaptado para lidar com restrições e objetivos específicos.

Desvantagens:

1. Ineficiência: O backtracking pode ser ineficiente para problemas grandes, porque o tempo de execução pode crescer exponencialmente com o tamanho do problema. Isso pode ser mitigado até certo ponto através do uso de estratégias de poda, mas mesmo assim o backtracking pode ser muito lento para problemas muito grandes.
2. Espaço de memória: O backtracking requer uma quantidade significativa de memória para armazenar a pilha de chamadas recursivas, especialmente para problemas grandes. Isso pode ser um problema se a memória for limitada.
3. Dificuldade de paralelização: Devido à sua natureza recursiva e ao fato de que a decisão em cada passo depende das decisões anteriores, o backtracking é difícil de paralelizar. Isso significa que ele não pode tirar proveito completo dos sistemas de computação modernos com vários núcleos ou máquinas.

Explicação do código:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Backtracking.java

Atributos:

melhorCombinacao: Lista para armazenar a melhor combinação de lances encontrada.

melhorValorTotal: Variável para armazenar o valor total da melhor combinação.

Construtor Backtracking(): Inicializa melhorCombinacao como uma lista vazia e melhorValorTotal como 0.

Método resolver(lances, energiaTotal): Recebe a lista de lances e a energia total disponível como entrada.

Reinicia melhorValorTotal e melhorCombinacao.

Ordena os lances em ordem crescente de energia (para otimizar a poda).

Chama o método backtracking recursivo para explorar as combinações.

Retorna o melhorValorTotal encontrado.

Método backtracking(lances, energiaRestante, valorAtual, indiceAtual, combinacaoAtual): Implementa o algoritmo recursivo de backtracking.

Condições de Término: Se a energia restante for menor que 0 (inviável) ou todos os lances foram percorridos.

Se a solução atual for melhor que a melhor já encontrada, atualiza melhorCombinacao e melhorValorTotal.

Recursão: Passo 1: Não incluir o lance atual: Chama backtracking recursivamente, incrementando o índice para o próximo lance.

Passo 2: Incluir o lance atual (se possível): Verifica se a energia restante é suficiente para o lance atual.

Se sim, adiciona o lance à combinacaoAtual.

Chama backtracking recursivamente, atualizando a energia restante, o valor atual e o índice.

Remove o lance de combinacaoAtual (backtracking).

Métodos auxiliares: imprimirMelhorCombinacao(): Imprime os detalhes da melhor combinação encontrada.

getMelhorCombinacao(): Retorna a lista melhorCombinacao.

getEnergiaTotalMelhorCombinacao(): Calcula e retorna a energia total da melhor combinação.

getMelhorValor(): Retorna o valor de melhorValorTotal.

Funcionamento do algoritmo:

O algoritmo backtracking explora todas as combinações possíveis de lances de forma recursiva. Em cada passo, ele decide se inclui ou não o lance atual na combinação. A decisão de incluir um lance é baseada na energia restante. Se a energia for suficiente, o lance é adicionado e a recursão continua. Caso contrário, o lance é ignorado. A cada combinação válida encontrada, o algoritmo verifica se ela é melhor que a melhor combinação encontrada até o momento.

Observações:

A ordenação dos lances por energia no início do algoritmo permite uma otimização chamada "poda", onde a recursão é interrompida mais cedo para ramos que já são piores que a melhor solução encontrada.

O algoritmo garante encontrar a solução ótima (melhor combinação possível), mas sua complexidade de tempo é exponencial no pior caso.

A implementação utiliza estruturas de dados como List e Collections da biblioteca padrão do Java.

Algoritmo Guloso 1: Maior Valor Total Primeiro

Ordenação por Valor Total: Primeiramente, os lances são ordenados em ordem decrescente com base no valor total oferecido (V) por cada lote de energia. Isso é realizado através de uma função de comparação que classifica os lances com base no valor total.

Seleção dos Lances: Após a ordenação, o algoritmo percorre a lista de lances ordenada e seleciona os lances com os maiores valores totais. Ele acumula o valor e a energia dos lances selecionados até que a energia total disponível seja esgotada ou não seja possível adicionar mais lances sem exceder a energia disponível.

Implementação: O algoritmo mantém um rastreamento da energia total acumulada e do valor total acumulado. Para cada lance, verifica-se se a energia do lance pode ser adicionada sem exceder a energia total disponível. Se puder, o lance é adicionado à seleção.

Vantagens:

1. Simplicidade: É fácil de implementar e compreender, envolvendo apenas a ordenação dos lances por valor total e uma iteração simples pela lista ordenada.
2. Rapidez: A execução é rápida devido à eficiência das operações de ordenação e à única passagem pela lista de lances ordenados.
3. Utilização de Recursos: Prioriza lances com maiores valores totais, o que pode ser vantajoso em contextos onde maximizar o valor absoluto é mais importante do que a eficiência energética.

Desvantagens:

1. Eficiência Energética Limitada: Pode resultar em uma subutilização da energia disponível, pois não considera a relação entre valor e energia (V/K). Isso pode levar a uma situação onde lances com alta energia são selecionados mesmo se oferecerem um valor por unidade de energia inferior.
2. Não Garantia de Ótimo: Devido à abordagem gulosa, não garante a obtenção da solução ótima. Pode deixar de explorar combinações que poderiam maximizar melhor o valor total dentro do limite de energia disponível.

Comparação de valores:

Tempo de Ordenação: O tempo principal de execução está associado à ordenação inicial dos lances por valor total, que possui complexidade O(nlogn)O(n \log n)O(nlogn), onde nnn é o número de lances.

Tempo de Seleção: Após a ordenação, a seleção dos lances ocorre em O(n)O(n)O(n), resultando em uma execução total eficiente.

Explicação do código:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Guloso.java

Método: resolverEstrategia1(lances, energiaDisponivel):

Ordenação: Ordena a lista de lances em ordem decrescente de valor total (valor).

Inicialização: Define variáveis para controlar a energia vendida (energiaVendida), o valor total acumulado (valorTotal) e uma lista para armazenar os lances selecionados (lancesSelecionados).

Iteração: Percorre a lista de lances ordenada.

Seleção: Para cada lance, verifica se a energia do lance, somada à energia já vendida, não ultrapassa a energiaDisponivel. Se não ultrapassar, o lance é adicionado à lista lancesSelecionados, a energiaVendida é atualizada e o valorTotal é incrementado.

Impressão: Imprime a energia total vendida e o valor total obtido (atualmente comentado).

Método: resolverEstrategia2(lances, energiaDisponivel):

Segue a mesma lógica da estratégia 1, com a diferença na ordenação:

Ordenação: Ordena a lista de lances em ordem decrescente de valor por megawatt (valor / energia). Isso prioriza os lances com maior retorno por unidade de energia.

Método imprimirLancesSelecionados(estrategia, lancesSelecionados, num, energiaTotal):

Método auxiliar para imprimir os resultados de cada estratégia: Imprime o nome da estratégia.

Lista os lances selecionados, incluindo energia, valor e valor por megawatt.

Calcula e imprime a energia total vendida e a energia não vendida.

Ambas as estratégias são gulosas porque tomam a melhor decisão local em cada passo (selecionar o lance com maior valor ou valor por megawatt), sem considerar as consequências futuras. Essa abordagem não garante a solução ótima global, mas geralmente é eficiente e produz resultados satisfatórios.

Algoritmo Guloso 2: Maior Valor por Megawatt Primeiro

Ordenação por Valor por Megawatt: Os lances são ordenados em ordem decrescente com base no valor por megawatt (V/K) oferecido por cada lote de energia. Esta abordagem assegura que os lances com a melhor eficiência de valor por unidade de energia sejam considerados primeiro.

Seleção dos Lances: Após a ordenação, o algoritmo seleciona os lances com a melhor eficiência energética até que a energia disponível seja totalmente alocada.

Implementação: Similar ao primeiro algoritmo, percorre a lista ordenada após a ordenação e acumula valor e energia até que a energia disponível seja esgotada.

Vantagens:

1. Eficiência Energética: Prioriza lances com melhor valor por megawatt (V/K), o que geralmente resulta em uma utilização mais eficiente da energia disponível.
2. Melhor Utilização de Recursos: Tende a maximizar o valor total possível dentro do limite de energia, favorecendo lances que oferecem mais valor por unidade de energia consumida.
3. Complexidade Moderada: A implementação envolve uma ordenação adicional baseada na relação V/K, mas ainda mantém uma eficiência computacional adequada para conjuntos de dados razoavelmente grandes.

Desvantagens:

1. Complexidade Adicional: Requer uma ordenação adicional dos lances com base na eficiência V/K, o que pode aumentar ligeiramente o tempo de execução em comparação com o Algoritmo Guloso 1.
2. Heurística: Assim como o Algoritmo Guloso 1, não garante a solução ótima devido à natureza heurística. Pode não explorar todas as combinações possíveis que poderiam resultar na maximização absoluta do valor total.

Comparação de valores:

Tempo de Ordenação: Além da ordenação inicial por valor total, há uma ordenação adicional por valor por megawatt (V/K), que também possui complexidade O(nlogn)O(n \log n)O(nlogn).

Tempo de Seleção: A seleção subsequente dos lances é similar ao Algoritmo Guloso 1, ocorrendo em O(n)O(n)O(n).

Explicação do código:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/Guloso1.java

Método: resolverEstrategia2(lances, energiaDisponivel):

Ordenação:

A primeira etapa é ordenar a lista de lances recebida como parâmetro. A ordenação é feita em ordem decrescente de valor por megawatt, ou seja, a razão entre o valor do lance (lance.getValor()) e a energia do lance (lance.getEnergia()).

Isso significa que os lances com maior retorno financeiro por unidade de energia serão posicionados no início da lista.

Inicialização:

energiaVendida: Variável para controlar a quantidade de energia total já alocada para os lances selecionados, inicializada com zero.

valorTotal: Variável para acumular o valor total dos lances selecionados, inicializada com zero.

lancesSelecionados: Uma lista para armazenar os lances que foram escolhidos pelo algoritmo.

Iteração e Seleção: O código percorre a lista de lances lances (já ordenada) usando um laço for.

Para cada lance na lista: Verifica se a energia do lance atual (lance.getEnergia()), somada à energia já vendida (energiaVendida), ultrapassa a energia total disponível (energiaDisponivel).

Se a energia total não ultrapassar o limite: O lance é adicionado à lista de lancesSelecionados.

A energiaVendida é incrementada com a energia do lance atual.

O valorTotal é incrementado com o valor do lance atual.

Impressão e Retorno: Após percorrer todos os lances, o método chama imprimirLancesSelecionados para exibir os lances que foram escolhidos.

O método retorna o valorTotal obtido com a seleção dos lances.

Método imprimirLancesSelecionados(estrategia, lancesSelecionados):

Este método auxiliar recebe a estratégia utilizada e a lista de lances selecionados.

Ele imprime na tela: O nome da estratégia utilizada.

A lista de lances selecionados, mostrando para cada lance: A energia do lance.

O valor do lance.

O valor por megawatt do lance.

O algoritmo guloso implementado busca selecionar os lances com o maior valor por megawatt primeiro, até que a energia disponível seja totalmente utilizada ou não haja mais lances disponíveis. Essa estratégia visa maximizar o retorno financeiro, priorizando os lances mais eficientes em termos de energia.

Guloso 1 VS Guloso 2

Comparação das Estratégias:

Estratégia 1 (Maior Valor Total): Prioriza lances de alto valor, mesmo que consumam muita energia. Pode ser vantajosa se o objetivo é maximizar o lucro total, mesmo que sobre energia disponível.

Estratégia 2 (Maior Valor por Megawatt): Prioriza lances com melhor custo-benefício em termos de energia. Pode ser vantajosa se o objetivo é utilizar a energia disponível da forma mais eficiente possível.

A melhor estratégia depende do contexto específico do problema e dos objetivos a serem priorizados.

Algoritmo Divisão e Conquista

Decisão de usar Divisão e Conquista: O problema envolve decidir se incluímos ou não cada lance na venda de energia para maximizar o lucro total sem ultrapassar o valor de energia disponível. Isso é semelhante ao problema da mochila, onde se decide incluir ou não um item na mochila para maximizar o valor total sem exceder a sua capacidade de peso.

Funcionamento da Divisão e Conquista: A abordagem da divisão e conquista envolve dividir o problema em subproblemas menores, conquistar os subproblemas resolvendo-os recursivamente e combinar os resultados para obter a solução do problema original. No caso do leilão de energia, para cada lance, o código divide o problema em dois subproblemas:

- Um considerando a inclusão do lance.
- Outro considerando a exclusão do lance.

Depois disso, combina-se os resultados dos subproblemas ao escolher o máximo valor obtido entre incluir ou não incluir o lance.

Processo do Algoritmo:

- Caso Base: Se não há energia restante ou não há mais lances para considerar, o valor total obtido é zero.
- Divisão: O problema é dividido em dois subproblemas - Incluir o Lance Atual e Excluir o Lance Atual. No primeiro subproblema, se o lance atual pode ser acomodado, calcula-se o valor total incluindo este lance. Enquanto isso, no segundo subproblema, calcula-se o valor total sem incluir o lance.
- Conquista: O algoritmo recursivamente considera ambos os casos (incluir e excluir o lance atual) e retorna o valor máximo entre essas duas opções.

Vantagens:

1. Decisão Clara entre Incluir ou Excluir Lances: A abordagem de divisão e conquista adapta-se bem ao problema de decidir entre incluir ou excluir cada lance de energia. Isso simplifica a modelagem do problema, permitindo uma análise clara de cada escolha em termos de impacto no valor total obtido e na energia restante.
2. Simulação de Todas as Combinações Possíveis: A divisão e conquista avalia todas as possíveis combinações de lances, o que garante a obtenção da solução ótima. Isso é particularmente útil no caso do leilão, onde a solução ótima não é intuitiva e existem múltiplas combinações que podem levar a resultados próximos.

Desvantagens:

1. Escalabilidade Limitada: A abordagem de divisão e conquista pode enfrentar problemas de escalabilidade devido à sua complexidade temporal exponencial no pior caso. Isso ocorre porque o número de subproblemas cresce exponencialmente com o número de lances, tornando-o impraticável para grandes conjuntos de lances (alta quantidade de combinações a serem exploradas).
2. Recursão Profunda: A abordagem depende de chamadas recursivas profundas, o que pode resultar em alto consumo de memória, especialmente quando o número de lances é grande.

Explicação do código:https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/DivConquista.java

Atributos:

melhorCombinacao: Uma lista para armazenar a melhor combinação de lances encontrada.

melhorValorTotal: Um inteiro para armazenar o valor total da melhor combinação.

memo: Uma matriz de inteiros para armazenar resultados já calculados (memoização).

tomarLance: Uma matriz booleana para rastrear quais lances foram incluídos na solução ótima.

Construtor DivConquista():

Inicializa os atributos: melhorCombinacao como lista vazia, melhorValorTotal como 0, e memo e tomarLance como null (serão inicializados posteriormente).

Método resolver(lances, energiaTotal):

Recebe a lista de lances e a energia total disponível.

Inicializa as matrizes memo e tomarLance com as dimensões apropriadas.

Preenche a matriz memo com -1, indicando que nenhum resultado foi calculado ainda.

Chama divisaoEConquista para calcular o melhorValorTotal.

Chama reconstruirMelhorCombinacao para determinar quais lances compõem a solução ótima.

Imprime a melhor combinação encontrada usando imprimirMelhorCombinacao.

Retorna um array contendo o melhorValorTotal e a energia total vendida.

Método divisaoEConquista(lances, energiaRestante, indiceLance):

Caso Base: Se todos os lances foram considerados (indiceLance == lances.size()), retorna 0.

Verificação da Memoização: Se o resultado para os parâmetros atuais já estiver em memo, retorna o valor armazenado.

Divisão: Calcula o valor de incluir (incluirLance) ou excluir (excluirLance) o lance atual.

Combinação: Compara incluirLance e excluirLance, armazena o maior valor em memo e atualiza tomarLance de acordo.

Retorna o valor máximo encontrado.

Método reconstruirMelhorCombinacao(lances, energiaTotal): Reconstrói a melhor combinação de lances percorrendo a matriz tomarLance e adicionando os lances que foram marcados como true.

Método imprimirMelhorCombinacao(): Imprime a melhor combinação de lances encontrada, incluindo a energia, o valor de cada lance e o valor total.

A divisão e conquista explora as possibilidades de incluir ou excluir cada lance, e a memoização garante que cada subproblema seja resolvido apenas uma vez. A matriz tomarLance é crucial para rastrear quais lances compõem a solução ótima.

Algoritmo Programação Dinâmica

Decisão de usar Programação Dinâmica: A estratégia de programação dinâmica é ideal para solucionar o problema proposto, devido à sua eficiência em resolver problemas de otimização combinatória, como o problema da mochila, onde a escolha de incluir ou não um item afeta diretamente o resultado final.

Funcionamento da Programação Dinâmica: A abordagem de programação dinâmica resolve o problema em etapas, onde cada etapa constrói uma solução ótima baseada nas soluções ótimas das etapas anteriores. No caso do leilão de energia:

- Uma matriz dp é utilizada para armazenar os valores máximos de lucro alcançados para diferentes capacidades de energia e subconjuntos de lances.
- A matriz é preenchida iterativamente, considerando cada lance e decidindo se ele deve ser incluído na solução ótima.

Processo do Algoritmo:

1. Inicialização: A matriz dp é inicializada com zeros.
2. Preenchimento da matriz: Para cada lance disponível e para cada quantidade de energia até a capacidade total, decide-se se o lance atual será incluído na solução ótima, atualizando dp[i][w] com o valor máximo possível. A decisão é baseada na comparação entre incluir ou não incluir o lance atual.
3. Determinação da Solução Ótima: Após o preenchimento, determina-se a combinação de lances que resulta no valor máximo de lucro usando a matriz dp.

Vantagens:

Otimização garantida: a programação dinâmica garante encontrar a solução ótima global para o problema, pois explora todas as combinações possíveis de forma sistemática.

Evita recálculo de subproblemas: ao armazenar os resultados de subproblemas já resolvidos (memorização), evita-se o recálculo desnecessário, aumentando a eficiência do algoritmo.

Aplicável a vários problemas: a programação dinâmica é uma abordagem versátil que pode ser aplicada a uma ampla gama de problemas de otimização, como o problema da mochila, o problema da subsequência comum mais longa, e muitos outros.

Soluções parciais reutilizáveis: as soluções parciais construídas durante o processo podem ser reutilizadas para resolver problemas maiores, tornando a abordagem incremental e eficiente.

Desvantagens:

Uso de Memória: A abordagem de programação dinâmica desenvolvida requer o armazenamento de uma matriz dp, o que pode consumir mais memória, especialmente em casos onde a grandes conjuntos de dados de entrada.

Explicação do código:
https://github.com/DisciplinasProgramacao/projetodealgoritmos-leilaoenergia-fpaa_leilaoenergia_backtracking/blob/master/codigo/ProgDinamica.java

Atributos:

melhorCombinacao: Uma lista para armazenar a melhor combinação de lances encontrada.

melhorValorTotal: Um inteiro para armazenar o valor total da melhor combinação.

Construtor ProgDinamica(): Inicializa melhorCombinacao como uma lista vazia e melhorValorTotal como 0.

Método resolver(lances, energiaTotal):

Inicialização da Tabela dp: Cria uma tabela dp de tamanho (n + 1) x (energiaTotal + 1), onde n é o número de lances.

Cada célula dp[i][w] armazenará o valor máximo que pode ser obtido usando os primeiros i lances e com um limite de energia w.

Preenchimento da Tabela dp:

Utiliza dois loops aninhados para preencher a tabela dp:

O loop externo itera sobre os lances (i de 1 até n).

O loop interno itera sobre as energias possíveis (w de 0 até energiaTotal).

Para cada célula dp[i][w], existem duas opções:

Incluir o lance atual (lances.get(i - 1)): Isso só é possível se a energia do lance atual for menor ou igual à energia disponível (w). Nesse caso, o valor máximo seria o máximo entre:

O valor máximo sem incluir o lance atual (dp[i - 1][w]).

O valor do lance atual mais o valor máximo obtido usando os lances anteriores e a energia restante (dp[i - 1][w - lances.get(i - 1).getEnergia()] + lances.get(i - 1).getValor()).

Não incluir o lance atual: Nesse caso, o valor máximo seria o mesmo de usar apenas os lances anteriores (dp[i - 1][w]).

Recuperação da Solução:

Após preencher a tabela dp, o valor máximo possível estará na célula dp[n][energiaTotal].

Para recuperar a combinação de lances que resulta nesse valor máximo, o código percorre a tabela dp de trás para frente, verificando se cada lance foi incluído na solução ótima.

Se dp[i][w] for diferente de dp[i - 1][w], significa que o lance i foi incluído na solução ótima.

Retorno do Resultado: O método retorna um array contendo o valor máximo obtido (dp[n][energiaTotal]) e a energia total vendida na melhor combinação (getEnergiaTotalMelhorCombinacao()).

Método imprimirMelhorCombinacao(): Imprime a lista de lances que compõem a melhor combinação encontrada, juntamente com suas respectivas energias e valores.

Método getEnergiaTotalMelhorCombinacao(): Calcula e retorna a energia total vendida na melhor combinação de lances.

A programação dinâmica resolve o problema construindo uma tabela de valores máximos para todas as combinações possíveis de lances e energia disponível. Essa abordagem evita o recálculo de subproblemas, tornando o algoritmo mais eficiente do que a recursão pura (Divisão e Conquista sem memorização).

A solução com programação dinâmica garante encontrar a solução ótima global para o problema, ou seja, a combinação de lances que resulta no maior lucro possível, respeitando o limite de energia.

Resultados:

Foi escolhido energia total disponível de 11000 MW, cada lance de 1 a 1000 MW com valor de 1 a 2000, gerados aleatoriamente. O maior tamanho T dentro de 30 segundos de execução de Backtracking foi de 31. Os mesmos conjuntos foram usados para executar os demais algoritmos.

Algoritmo	Tempo Médio (s)	Energia Total (MW)	Valor Total

Backtracking	18.9697	10989, 10923, 10704	25717, 26705, 26283
Algoritmo Guloso 1	0.0002	10999, 10909, 10897	24962, 26317, 26629
Algoritmo Guloso 2	0.0002	10992, 10923, 10875	25982, 26705, 26652
Divisão e Conquista	0.005	10992, 10923, 10993	25982, 26705, 26698
Programação Dinâmica	0.0051	10992, 10985, 10994	25982, 46071, 46720

Para os teste com conjuntos de tamanho T até 10T executados pelos algoritmos Gulosos e Programação dinâmica obtivemos os seguintes resultados em segundos com aproximações de até 4 casas decimais:

Algoritmo	31	62	93	124	155	186	217	248	279	310
Guloso 1	0.0002	0.0	0.0	0.0001	0.0001	0.0002	0.0001	0.0002	0.0002	0.0003
Guloso 2	0.0002	0.0	0.0	0.0002	0.0001	0.0001	0.0002	0.0002	0.0002	0.0002
Programação Dinâmica	0.0061	0.0056	0.0054	0.0070	0.0107	0.009	0.009	0.0105	0.0118	0.0135

Comparação resultados:

Backtracking:

Este método garante encontrar a solução ótima, pois explora todas as possíveis combinações. No entanto, o tempo de execução é significativamente maior em comparação com os outros métodos, especialmente para conjuntos de dados maiores.

Portanto, embora possa fornecer a melhor solução, pode não ser prático para problemas grandes.

Algoritmo Guloso:

Este método é o mais rápido de todos, pois faz a escolha que parece ser a melhor em cada passo. No entanto, essa abordagem nem sempre leva à solução ótima global. Isso é evidente nos valores obtidos que são geralmente menores do que os dos outros métodos, como é mais claro olhando os resultados do Guloso 1.

Quando se trata de conjuntos maiores o algoritmo não apresenta aumento significativo no tempo de execução, mas também não gera resultados ótimos.

Portanto, o Algoritmo Guloso pode ser útil quando o tempo de execução é uma preocupação e uma solução aproximada é aceitável.

Divisão e Conquista:

Este método divide o problema em subproblemas menores, resolve cada um deles individualmente e combina suas soluções para resolver o problema original.

Ele apresenta um bom equilíbrio entre a qualidade da solução e o tempo de execução.

Portanto, a Divisão e Conquista é uma boa opção para problemas de otimização, pois fornece soluções de alta qualidade em um tempo de execução razoável.

Programação Dinâmica:

Semelhante à Divisão e Conquista, este método também divide o problema em subproblemas menores. No entanto, ele armazena os resultados dos subproblemas para evitar cálculos duplicados, tornando-o mais eficiente.

Quando se trata de uma maior quantidade de dados, o algoritmo tem um aumento constante no tempo de execução, mas continua dando os resultados ótimos.

A Programação Dinâmica também fornece um bom equilíbrio entre a qualidade da solução e o tempo de execução, tornando-a uma excelente opção para problemas de otimização.

Conclusão:

Os resultados apresentados acima mostram a comparação entre quatro diferentes algoritmos - Backtracking, Algoritmo Guloso, Divisão e Conquista e Programação Dinâmica - para resolver o mesmo problema de otimização.

Backtracking é um método que constrói todas as soluções possíveis para o problema. É uma abordagem exaustiva que garante encontrar a solução ótima, mas pode ser muito lento para problemas grandes, como evidenciado pelos tempos de execução relatados.

O Algoritmo Guloso, por outro lado, faz a escolha localmente ótima em cada etapa com a esperança de encontrar uma solução global ótima. No entanto, essa abordagem nem sempre leva à solução ótima. Os resultados mostram que o Algoritmo Guloso é significativamente mais rápido do que o Backtracking, mas os valores obtidos são geralmente menores.

A Divisão e Conquista é uma técnica que divide o problema em subproblemas menores, resolve cada um deles individualmente e combina suas soluções para resolver o problema original. Esta abordagem é eficiente e fornece a solução ótima, como mostrado nos resultados.

Finalmente, a Programação Dinâmica é uma técnica que resolve o problema dividindo-o em subproblemas menores e armazenando os resultados dos subproblemas para evitar cálculos duplicados. Este método é altamente eficiente e fornece a solução ótima, como evidenciado pelos resultados.

A conclusão da comparação entre os quatro algoritmos - Backtracking, Algoritmo Guloso, Divisão e Conquista e Programação Dinâmica - é que a escolha do algoritmo depende muito do tamanho do problema e das exigências de tempo e precisão.

O Backtracking, embora forneça a solução ótima, pode ser impraticável para problemas maiores devido ao seu alto tempo de execução. O Algoritmo Guloso é mais rápido, mas pode não fornecer a solução ótima, pois faz a escolha que parece ser a melhor em cada passo sem considerar as consequências futuras.

Por outro lado, a Divisão e Conquista e a Programação Dinâmica, que dividem o problema em subproblemas menores e constroem a solução a partir desses subproblemas, fornecem um bom equilíbrio entre tempo de execução e qualidade da solução. Eles são capazes de fornecer soluções de alta qualidade em um tempo de execução razoável, tornando-os preferíveis para problemas de otimização.

Portanto, a escolha do algoritmo deve ser baseada nas necessidades específicas do problema, levando em consideração fatores como o tamanho do conjunto de dados e a importância da precisão versus velocidade.