PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Fundamentos de Projeto e Análise de Algoritmos

Bruno Rocha Corrêa Urbano
Henrique Dani Franco Nezio
Pedro de Barros Alves

Divisão de Tarefas:

Backtracking: Bruno Urbano
Divisão e Conquista: Pedro de Barros
Programação Dinâmica: Henrique Dani
Relatório: Grupo

Guloso: Como nosso grupo possui somente 3 integrantes, o professor informou que não precisaríamos realizar a implementação do algoritmo guloso.

Geração dos Dados:

Classe Geradora:

Criamos uma classe responsável por gerar conjuntos de dados de teste que são utilizados na execução dos 3 algoritmos implementados. A classe utiliza uma semente fixa para o gerador de números aleatórios (Random), garantindo que os mesmos conjuntos de dados sejam gerados em execuções diferentes. Isso permite comparações justas entre diferentes algoritmos, pois utilizam os mesmos conjuntos de dados. Além disso, na classe, é feita uma validação para que a energia produzida seja sempre menor que a soma de megawatts dos lances, fazendo assim com que todos os conjuntos não sejam inadequados.

Link github Classe Geradora

Representação dos Dados:
Implementamos duas classes principais para representar os dados do problema:

- Classe Lance: Representa um lance feito por uma empresa, contendo informações sobre a empresa interessada, a quantidade de megawatts ofertada e o valor do lance.
- Classe Conjunto: Contém a quantidade total de energia disponível para venda e uma lista de lances das empresas interessadas.

Link github Classe Conjunto e Lance

Algoritmo Backtracking

Link do Código do Algoritmo Backtracking

O algoritmo de backtracking tem a capacidade de explorar todas as combinações possíveis de lances para encontrar a solução ótima. Esse algoritmo é particularmente útil para resolver problemas de otimização como o proposto, onde queremos maximizar o valor total obtido pelas vendas de energia.

Poda de Soluções Não Promissoras:

Foi implementada uma estratégia de poda básica no algoritmo de backtracking. Se a solução atual tem um valor acumulado maior do que o valor máximo encontrado até o momento, atualizamos a melhor solução. Caso contrário, descartamos a solução atual, reduzindo o espaço de busca e melhorando a eficiência do algoritmo.

Funcionamento do Algoritmo Backtracking:

Inicialização: Recebe um objeto Conjunto, que contém a energia disponível e a lista de lances. Inicializa duas listas: bestSolution para armazenar a melhor solução encontrada e currentSolution para armazenar a solução atual durante o processo de backtracking. Chama a função recursiva backtrack para iniciar a busca pela combinação ótima de lances.

Recursão e Poda: A função backtrack é responsável por explorar todas as combinações possíveis de lances, utilizando a poda básica para evitar explorar soluções que não podem ser melhores do que a solução já encontrada.

Parâmetros:

- “lances”: Lista de lances disponíveis.
- “energiaDisponivel”: Energia ainda disponível para venda.
- “currentIndex”: Índice atual na lista de lances.
- “currentValue”: Valor acumulado da solução atual.
- “currentSolution”: Lista de lances na solução atual.
- “bestSolution”: Lista de lances na melhor solução encontrada até o momento.

- “maxValue”: Valor máximo encontrado até o momento.

Construção da Solução: A cada passo, a função decide incluir ou não um lance na solução atual. Se incluir, a função chama recursivamente backtrack com os parâmetros atualizados. Após a chamada recursiva, remove o lance da solução atual (backtracking) para explorar outras possibilidades.

Em mais detalhes:

- Verifica se a solução atual é melhor do que a melhor solução encontrada até o momento. Se for, atualiza bestSolution e maxValue.
- Para cada lance a partir do índice atual:
    - Verifica se o lance pode ser incluído na solução atual sem exceder a energia disponível.
    - Se puder, adiciona o lance à solução atual e chama recursivamente backtrack com a energia e o valor atualizados.

    - Após a chamada recursiva, remove o lance da solução atual (backtracking) para explorar outras combinações.

Melhor Solução: Após a verificação de todas as opções adequados, encontramos a melhor solução e a retornamos.

Conclusão

O algoritmo de backtracking é adequado para problemas como o descrito, onde é necessário explorar todas as combinações possíveis para encontrar a solução ótima. Apesar de ser ineficiente para grandes entradas devido ao crescimento exponencial do número de combinações, ele garante a solução ótima ao custo de tempo de execução. A poda básica ajuda a reduzir o espaço de busca, mas melhorias adicionais podem ser implementadas para torná-lo mais eficiente.

Programação Dinâmica

Link do Código do Algoritmo

Tomadas de decisões referente ao preenchimento da tabela:

Primeiramente, foram construídas duas tabelas, dp e keep. A tabela dp irá armazenar o valor máximo de dinheiro obtido para diferentes números de lances e capacidades de energia. Já a tabela keep, irá indicar se um lance foi incluído na solução ótima para diferentes números de lances e capacidades de energia. A implementação da tabela keep se baseia em uma estratégia adotada para facilitar na recuperação dos dados pertencentes à solução ótima encontrada. Diante disso, a tabela keep é uma adaptação que irá auxiliar, visando a melhora do algoritmo padrão da programação dinâmica, pois ela é usada para rastrear quais lances são incluídos na solução. Dessa forma, a tabela keep proporciona uma recuperação direta e simplificada das decisões. Portanto, as linhas, colunas e células das duas tabelas serão as seguintes:

- Linhas de ambas tabelas: As linhas da tabela representam os lances disponíveis. Diante disso, para um conjunto de N lances, a tabela irá conter N+1 linhas, em que a linha 0 representa a solução base sem considerar qualquer lance.

- Colunas de ambas tabelas: As colunas da tabela representam a capacidade disponível de energia em megawatts, variando de 0 até W (energia total disponibilizada para venda pela empresa produtora de energia).

- Células da tabela dp: A célula dp[i][w] armazena o valor máximo de dinheiro que pode ser obtido considerando os primeiros i lances e uma capacidade de w megawatts. A célula é preenchida baseando-se na escolha de incluir ou não o lance i no conjunto de soluções ótimas.

- Células da tabela keep: A célula keep[i][w] é um booleano que indica se o lance i foi incluído na solução ótima quando a capacidade disponível é w megawatts. A célula irá conter true se o lance i foi incluído na solução ótima, e false caso contrário.

A lógica para preenchimento das tabelas, será a seguinte:

- Percorre-se cada lance (i de 1 a N) e cada capacidade (w de 0 a W).
- Se o lance i pode ser incluído na capacidade atual w (megawatts_i <= w):
    - Calcula-se o valor de incluir este lance: incluiLance = dinheiro_i + dp[i-1][w-megawatts_i]
    - Compara-se incluiLance com a solução de não incluir o lance atual: dp[i - 1][w].
    - Se incluir o lance é melhor, atualiza-se dp[i][w] para incluiLance e marca-se keep[i][w] como true.

    - Caso contrário, apenas carrega-se o valor da solução anterior dp[i - 1][w] para dp[i][w].

- Se o lance não pode ser incluído, ou seja (megawatts_i > w), simplesmente copia-se o valor da solução anterior: dp[i][w] = dp[i - 1][w].

Funcionamento do algoritmo de programação dinâmica:

O algoritmo começa inicializando duas tabelas:

- dp: uma matriz bidimensional de inteiros que armazena o valor máximo de dinheiro obtido para diferentes números de lances e capacidades de energia.
- keep: uma matriz bidimensional de booleanos que indica se um lance foi incluído na solução ótima para diferentes números de lances e capacidades de energia.

Em seguida ele realiza o preenchimento das tabelas:

A tabela dp é preenchida de maneira iterativa. Para cada lance e cada capacidade de energia, o algoritmo decide se incluir ou não o lance na solução:

- Para cada lance i (de 1 a n), representando o i-ésimo lance no conjunto de lances.
- Para cada capacidade w (de 0 a W), representando a quantidade de megawatts disponíveis.

A decisão de incluir ou não o lance se baseia no seguinte:

- Se o lance atual pode ser incluído (lance.megawatts <= w):
    - Calcula-se o valor se o lance for incluído: includeLance = lance.dinheiro + dp[i - 1][w - lance.megawatts].
    - Compara-se este valor com a solução de não incluir o lance: dp[i - 1][w].
    - Atualiza-se dp[i][w] com o maior valor e ajusta-se keep[i][w] para true se o lance foi incluído.
- Se o lance não pode ser incluído (lance.megawatts > w), copia-se o valor da solução anterior.

Após preencher as tabelas dp e keep, a solução ótima é recuperada retrocedendo da última célula até a primeira:

- Iniciamos pela última linha (n) e a última coluna (W).
- Para cada lance incluído na solução ótima (keep[i][w] é true), adicionamos o lance à lista solution e reduzimos a capacidade disponível w pelo valor dos megawatts do lance incluído.
- Continua-se esse processo até atingir a primeira linha da matriz keep.

Conclusão

O algoritmo de programação dinâmica se mostrou bastante eficiente para problemas como o proposto para solução no trabalho. Ele se mostrou bastante eficiente com uma quantidade bastante considerável de interessadas, pois o tempo de execução foi bem baixo e garantiu soluções ótimas. Justamente por manter a memória dos resultados, um custo a ser analisado é a maior utilização de memória.

Algoritmo Divisão e Conquista

Link do Algoritmo Divisão e Conquista

O algoritmo de divisão e conquista divide um grande problema em várias partes menores, que por sua vez, podem ser divididas em partes menores ainda (caso seja necessário). Depois de dividir o problema em subproblemas, o algoritmo trata estes subproblemas, combinando os resultados, no final de sua execução. Este tipo de algoritmo é muito eficiente para grande conjunto de dados, possuindo uma complexidade de O(nlog(n)). Um exemplo de algoritmo de Divisão e Conquista famoso é o quicksort.

Funcionamento do Algoritmo Divisão e Conquista:

Inicialização:

Recebe um objeto Conjunto, que contém a energia disponível e a lista de lances. O algoritmo então inicializa um array de inteiros (resultado) com 2 posições. Este array é responsável por guardar o valor máximo de dinheiro obtido e o total de energia gasta de uma solução. A primeira posição do array resultado chama a função recursiva “maxValueHelper()”, responsável por dividir o problema em subproblemas e encontrar o valor máximo de dinheiro possível dentre estes subproblemas. A segunda posição do array resultado chama a função “energiaRestante()” que é responsável por calcular a energia utilizada por aquela solução.

Recursão e Divisão:

Como dito anteriormente, a função responsável pela divisão dos lances é a “maxValueHelper()”. A primeira operação que a função executa é uma checagem para verificar se o “ArrayList lances” está vazio. Além disso, verifica também se existe alguma quantidade de

energia disponível. Caso uma dessas duas condições for verdadeira, o algoritmo para. Logo, este é o ponto de parada da recursividade da função “maxValueHelper()”.

Parâmetros:

- List<lance> lances: lista contendo os lances;
- int energiaDisponivel: quantidade de energia disponível para os lances;
- int megawattsGastos: parâmetro responsável por armazenar a quantidade de energia utilizada pelos lances. É inicializado com 0

Construção da Solução:

A cada recursão, o algoritmo seleciona um lance da lista de lances e verifica se a quantidade de energia solicitada por ele é maior do que a quantidade de energia disponível. Caso essa condição seja verdadeira, o lanceAtual então é removido da lista de lances e a função se chama novamente, porém agora com uma sublista de lances não incluindo o lance anteriormente checado. Caso esta condição seja falsa, o algoritmo também retirará o lanceAtual da lista de lances, mas dessa vez calculará dois cenários: um cenário incluindo o lance atual na solução e um cenário não incluindo o lance atual na solução. Após este cálculo (que em ambos os cenários também chamam a função recursivamente), é feito um retorno contendo o valor máximo obtido entre os dois cenários.

Melhor Solução:

Como dito anteriormente, a melhor solução é encontrada escolhendo o valor máximo entre a solução incluindo o lanceAtual e a solução não o incluindo.

Conclusão:

O algoritmo de divisão e conquista é extremamente eficiente para estes problemas, mesmo com um número muito grande de dados. Por mais que o algoritmo não encontre soluções tão boas quanto o algoritmo de backtracking ou o de programação dinâmica, por se tratar de um algoritmo extremamente mais rápido do que os dois previamente citados, é uma troca que vale a pena. Além disso, mesmo os resultados não sendo os ótimos, ainda são satisfatórios.

Relatório Técnico

Resultados de Cada Algoritmo:

Link Resultados

- O algoritmo de backtracking encontrou soluções ótimas consistentemente, mas seu tempo de execução aumentou exponencialmente com o tamanho do conjunto de dados.
- A programação dinâmica também encontrou soluções ótimas, em uma execução bem rápida, mostrando ser eficiente mesmo para conjuntos maiores.
- O algoritmo de divisão e conquista teve uma execução muito rápida, mas frequentemente encontrou soluções subótimas.

Tabelas de Resultados:

CONJUNTO 10 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	0.0026	[3294, 4155], [592, 2043], [2778, 3809], [828, 1188], [...]
Programação Dinâmica	0.002	[3294, 4155], [592, 2043], [2778, 3809], [828, 1188], [...]
Divisão e Conquista	0.001	[3471, 3108], [678, 2043], [2973, 2937], [867, 605], [...]

CONJUNTO 20 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	0.0039	[[6216,9851], [9189, 11109], [2777, 6358], [2180, 4187],

Programação Dinâmica	0.0	[6216, 9851], [9189, 11109], [2777, 6358], [2180, 4187],[...]
Divisão e Conquista	0.0	[6618, 7460], [9202, 10243], [2916, 2339], [2287, 1981],[...]

CONJUNTO 30 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	2.8377	[5619, 11069], [12160, 13282], [5365, 11369], [9560, 17158], [...]
Programação Dinâmica	0.0	[5619, 11069], [12160, 13282], [5365, 11369], [9560, 17158], [...]
Divisão e Conquista	0.0	[5635, 7409], [12373, 11696], [5416, 6880], [9974, 15940],[...]

CONJUNTO 34 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	37.5057	[[10208,14836], [2492, 7579], [13910, 17327], [5635, 12568], [...]
Programação Dinâmica	0.0	[10208, 14836], [2492, 7579], [13910, 17327], [5635, 12568],[...]

Divisão e Conquista	0.0	[10276, 11793], [2502, 2754], [13913, 15438], [5744, 7629],[...]

CONJUNTO 102 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	N/A	N/A
Programação Dinâmica	0.0	[5741, 21020], [42455, 49777], [21749,38289], [328, 4859], [...]
Divisão e Conquista	0.0	[5747, 8034], [42486, 45767], [21753, 18429], [337, 1485], [...]

CONJUNTO 204 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	N/A	N/A
Programação Dinâmica	0.001	[35847, 70149], [77690, 103343], [20657,50592], [30889, 66912], [...]

Divisão e Conquista	0.0	[35854, 36907], [77694, 83271], [20667, 23079], [30906, 33834], [...]

CONJUNTO 340 ELEMENTOS		
Algoritmo	**Tempo de Execução (s)**	**Resultado (Megawatts Utilizado, Dinheiro Total)**
Backtracking	N/A	N/A
Programação Dinâmica	0.001	[72630, 116124], [41543, 92001], [36068, 89813], [42881, 98355], [...]
Divisão e Conquista	0.0	[72637, 67168], [41566, 43185], [36080, 36395], [42882, 44308],[...]

Comparação de Resultados

Algoritmo Backtracking:

O algoritmo de backtracking é baseado na exploração exaustiva de todas as possíveis combinações de alocação de energia para encontrar a solução que maximize o lucro total. Durante a execução, ele aplica estratégias de poda para eliminar caminhos não promissores, o que ajuda a reduzir o espaço de busca. No entanto, o tempo de execução cresce exponencialmente com o aumento do tamanho do conjunto de dados, tornando-o impraticável para conjuntos maiores. Diante disso, o algoritmo backtracking foi o mais demorado: demora em média mais de 30s com um conjunto de 34 elementos.

- **Benefícios Backtracking:**
    - Garante a obtenção de soluções ótimas.
    - Útil para conjuntos de dados pequenos a moderados.
- **Malefícios Backtracking:**
    - O tempo de execução cresce exponencialmente com o aumento do tamanho do conjunto.
    - Impraticável para grandes conjuntos de dados devido ao alto custo computacional.

Programação Dinâmica:

O algoritmo de programação dinâmica utiliza uma abordagem bottom-up para resolver subproblemas e armazenar seus resultados, evitando a recomputação. Essa técnica permite que o algoritmo encontre soluções ótimas de forma eficiente, mesmo para conjuntos de dados grandes.

No nosso estudo, o algoritmo foi executado até conjuntos de 340 elementos, mantendo tempos de execução muito baixos e encontrando soluções ótimas consistentemente. Este desempenho eficiente é particularmente notável em comparação ao backtracking, que teve seu tempo de execução aumentado drasticamente para conjuntos de tamanho semelhante. No entanto, o custo da programação dinâmica se reflete no uso de memória, pois a abordagem exige o armazenamento dos resultados intermediários de todos os subproblemas, o que pode se tornar um desafio em cenários com restrições de memória.

- **Benefícios Programação Dinâmica:**
    - Encontra soluções ótimas de forma eficiente.
    - Tempo de execução significativamente menor em comparação ao backtracking.
    - Adequado para grandes conjuntos de dados, como observado com conjuntos de até 340 elementos.
- **Malefícios Programação Dinâmica:**
    - Implementação pode ser mais complexa, exigindo uma compreensão aprofundada do problema.
    - Requer mais memória para armazenar os resultados intermediários.

Divisão e Conquista:

O algoritmo de divisão e conquista divide o problema em subproblemas menores, resolve cada um recursivamente e combina as soluções. Essa abordagem é rápida e simples, mas não garante a obtenção de soluções ótimas. No nosso estudo, o algoritmo foi executado até conjuntos de 340 elementos, mantendo tempos de execução extremamente baixos. No entanto, as soluções encontradas foram frequentemente subótimas, refletindo a natureza heurística dessa abordagem.

- **Benefícios Divisão e Conquista:**
    - Executa muito rapidamente.
    - Simples de implementar para certos tipos de problemas.
    - Mantém tempos de execução baixos mesmo para grandes conjuntos de dados, como os de 340 elementos testados.
- **Malefícios Divisão e Conquista:**
    - Não garante a obtenção de soluções ótimas.
    - Pode encontrar soluções subótimas frequentemente, comprometendo a qualidade das soluções.

Conclusão

A programação dinâmica destacou-se como a abordagem mais eficiente entre as testadas, combinando rapidez e precisão na obtenção de soluções ótimas. Este algoritmo mostrou-se capaz de resolver grandes conjuntos de dados, até 340 elementos, mantendo tempos de execução muito baixos. A capacidade de armazenar resultados intermediários e evitar a recomputação permite que a programação dinâmica opere de forma eficiente, mas essa vantagem vem ao custo de maior utilização de memória. Em cenários onde os recursos de memória são abundantes, a programação dinâmica é claramente a melhor escolha para problemas de otimização complexos.

Por outro lado, o algoritmo de backtracking, embora garantisse a obtenção da melhor solução possível, revelou-se impraticável para conjuntos maiores devido ao tempo de execução exponencialmente crescente. Enquanto a sua precisão é inquestionável, a escalabilidade do backtracking é severamente limitada, tornando-o inadequado para aplicações com grandes volumes de dados.

A abordagem de divisão e conquista apresentou a execução mais rápida entre os algoritmos testados, demonstrando eficiência notável em termos de tempo. No entanto, essa

velocidade foi alcançada ao custo da qualidade das soluções, que frequentemente eram subótimas. A simplicidade e a rapidez da divisão e conquista fazem dela uma boa escolha para obter soluções rápidas e razoáveis, especialmente em situações onde a precisão não é crítica e o tempo é um fator limitante.

Em resumo, a programação dinâmica é a abordagem mais equilibrada, oferecendo uma combinação ideal de eficiência e precisão para grandes conjuntos de dados. O backtracking pode ser utilizado para problemas menores onde a obtenção da solução ótima é crucial e o tempo de execução não é um fator restritivo. A divisão e conquista é adequada para cenários que demandam soluções rápidas e onde uma ligeira perda de qualidade na solução é aceitável.