Pontifícia Universidade Católica do Rio Grande do Sul
Escola Politécnica
Ciência da Computação

Desenho, Análise e Implementação de Algoritmos de Programação Dinâmica e Branch-and-bound

Gabriel W. Piazenski, Gustavo W. M. da Silva e Lorenzo D. More

Trabalho II da disciplina "Projeto e Otimização de Algoritmos"

Professor: Rafael Scopel

Porto Alegre
Julho de 2024

1 O Problema 1 (Algoritmo Dinâmica)

O objetivo do primeiro problema é analisar e implementar um algoritmo do tipo *Backtracking* para encontrar a melhor a forma de gerir uma squad de desenvolvedores (DEVs) que precisam escolher uma tarefa para realizar a cada semana, em um esquema de sprints semanais. As tarefas possíveis são divididas em duas categorias: a de baixa dificuldade (por exemplo, criar uma API) e a de alta dificuldade (por exemplo, criar um detector de textos elaborados pelo ChatGPT). A objetivo, a cada semana, é escolher entre uma tarefa de baixa ou alta dificuldade.

Para cada semana i, existe uma recompensa, em dinheiro, para as tarefas: l_i para as de baixa dificuldade, e h_i para as de alta dificuldade. Porém, há uma restrição na escolha de uma tarefa de alta dificuldade, que é ter que ficar, na semana anterior a i, ou seja, i - *1*, sem fazer nada. Os detalhes estão resumidos a seguir:

1. **Tarefas de Baixa Dificuldade:**

   Podem ser realizadas em qualquer semana, independentemente da atividade da semana anterior;

   Recompensa: l_i.

2. **Tarefas de Alta Dificuldade:**

   Requerem uma semana de preparação anterior, durante a qual nenhuma outra tarefa é realizada;

   Podem ser selecionadas na primeira semana sem restrições;

   Recompensa: h_i.

3. **Não fazer nada:**

   Necessário para realizar uma tarefa de alta dificuldade;

   Recompensa: 0.

Os valores são associados às respectivas tarefas, ou seja, dependem da tarefa escolhida para a semana. Logo, o objetivo desse algoritmo é escolher entre tarefas de baixa ou alta dificuldade, ou não fazer nada para maximizar o valor da recompensa obtida pela equipe em uma sequência de n semanas. Em outras palavras, dados os conjuntos de valores $l_1, l_2, ..., l_n$ e $h_1, h_2, ..., h_n$, solicita-se encontrar um plano de valor máximo.

1.1 Resposta do item 1 - Problema 1

O item 1 fornece um algoritmo (que foi implementado pelo código de nome "exemplo.java") incorreto para a resolução do problema. Foi solicitado mostrar o porquê disso, além de fornecer uma instância na qual ele não retorna a resposta correta.

O problema no algoritmo ocorre se for escolhido uma tarefa de alta dificuldade em alguma semana diferente da última semana. Isso ocorre devido ao fato de que quando se escolhe uma atividade de alta dificuldade, na semana seguinte a esta, será obrigatoriamente escolhido uma tarefa de baixa dificuldade, pois o algoritmo na parte que analisa se é melhor escolher a tarefa de alta dificuldade pula a semana seguinte da semana que já foi escolhido fazer uma tarefa de alta dificulade.

Qualquer instância que apresente uma escolha por tarefa de alta dificuldade logo após já ter feito uma escolha por uma atividade de alta dificuldade gera esse problema. Em outras palavras, se na semana i for escolhido fazer uma tarefa de alta dificuldade (desde que não seja a última semana), na semana i + *1* o algoritmo não vai analisar se é melhor fazer nada na semana i e fazer a tarefa de alta dificuldade na semana i + *1*. Se ele decidiu que na semana i vai ser uma tarefa de alta de

dificuldade, ele vai pular a análise de se vale a pena, ou não, fazer uma tarefa de alta dificuldade na semana *i + 1*, decidindo colocar uma tarefa de baixa dificuldade na semana *i + 1*.

Isso foi constatado no código *exemplo.java* quando se colocou os seguintes valores para l (tarefa de baixa dificuldade) e h (tarefa de alta dificuldade):

- $l \leftarrow [10, 1, 11, 10]$
- $h \leftarrow [5, 50, 1000, 1]$

A resposta que o algoritmo retorna é 71. Entretanto, é de fácil análise visual que esse não é o melhor resultado (1020 seria resposta correta). Esse erro é devido à escolha, na semana 2, da tarefa de alta dificuldade, de valor 50; o que faz o algoritmo não analisar se é melhor realizar a tarefa de alta dificuldade, de valor 1000, na semana 3. Assim, acaba escolhendo a tarefa de baixa dificuldade, de valor 11, na semana 3.

Diante do exposto, esse algoritmo parece estar entre um algortimo *greedy* e um algoritmo *backtracking*, pois ele possui uma limitada capacidade de voltar para tentar achar um processo mais otimizado que o que se encontraria com um algoritmo puramente *greedy*.

1.2 Resposta do item 2 - Problema 1

Para o item 2, foi solicitado um algoritmo eficiente (tempo polinomial baixo) que assuma valores para o conjunto de entrada ($l_1, l_2, ..., l_n$ e $h_1, h_2, ..., h_n$), retorna o valor de um plano ideal (valor maximizado) e lista quais tarefas foram executadas em cada semana. O algoritmo pode ser encontrado na sequência. Após, ocorrerá sua análise e será mostrado o tempo de execução.

1.3 O Algoritmo

Algorithm 1 solveP1(int l[], int h[])

```
n ← l.length                                   ▷ Size of sequence of weeks
S ← int[n + 1]                                 ▷ Set of tasks done in each week
week ← string[n + 1]                           ▷ Set of the type of assignment
S[1] ← max{l[0], h[0]}
for i = 2; i ≤ n; i++ do
    if S[i − 1] + l[i − 1] > S[i − 2] + h[i − 1] then   ▷ Best option is low difficult assignment
        S[i] ← S[i − 1] + l[i − 1]
    else                                       ▷ Best option is high difficult assignment
        S[i] ← S[i − 2] + h[i − 1]
    end if
end for
for eachweek do
     printweek[i]                              ▷ Prints the type of assignment for each week
end for
     return S[n]                               ▷ Return the value for an ideal plan
```

1.4 Análise do Algoritmo

O algoritmo usa uma abordagem de programação dinâmica para calcular o valor máximo acumulado, baseando-se em subproblemas resolvidos anteriormente. Haja vista que a programação dinâmica é uma técnica de otimização usada para resolver problemas complexos, quebrando-os em subproblemas menores e solucionando cada um desses subproblemas apenas uma vez, armazenando seus resultados para evitar cálculos redundantes.

Tendo em vista que o código acima descrito é uma versão simplificada do código, será feita uma análise com base no código que foi enviado em anexo a esse relátório.

Assim, o algoritmo começa inicializando arrays que armazenarão os valores máximos acumulados (maxAcumulado[]) e as escolhas que serão feitas semanalmente (semana[] e semana2[]).

Diante da peculiaridade de a primeira semana poder ser a única semana que começa com tarefa de alta dificuldade sem prévia semana fazendo nada, é feita uma simples escolha de qual apresenta melhor custo benefício entre as tarefas de baixa e alta dificuldade, sendo armazenadas/acumuladas no array maxAcumulado.

A partir da segunda semana o algoritmo resolve o subproblema de maximizar o valor acumulado até aquela semana, considerando duas possibilidades:

- *Tarefa de Baixa Dificuldade: Adiciona o valor da tarefa de baixa dificuldade da semana atual ao valor acumulado da semana anterior.*
- *Tarefa de Alta Dificuldade: Adiciona o valor da tarefa de alta dificuldade da semana atual ao valor acumulado de duas semanas atrás (considerando que a semana anterior foi "nada").*

Para tanto, utiliza-se os resultados das semanas anteriores (*maxAcumulado[i - 1]* e *maxAcumulado[i - 2]*) para a semana i, e desta forma é construída a solução da semana i e armazena-se os valores de acordo.

Com o array *maxAcumulado* preenchido, o algoritmo imprime as escolhas feitas em cada semana e o valor armazenado na última posição, por ser o máximo acumulado, o valor da solução ótima, que foi encontrando a partir dos subproblemas resolvidos.

1.5 Implementação e Tempo de Execução

A complexidade total do método *solveP1* é $O(n)$, sendo n o número de semanas (ou o tamanho dos arrays l e h). Portanto o tempo de execução do código fornecido é linear, pois precisa percorrer todo o tamanho do array.

O código-fonte comentado dos dois itens pode ser encontrado nos arquivos *.java* enviados junto desse relatório. A análise do problema 2 inicia na próxima página.

2 O Problema 2 (Algoritmo de Branch and Bound)

O problema 2 se trata de resolver o problema da mochila (em inglês *knapsack*) abaixo utilizando Branch-and-Bound.

Este problema é de optimização combinatória e é um problema NP-completo. O nome dá-se devido ao modelo de uma situação em que é necessário preencher uma mochila com objetos de diferentes pesos e valores. O objetivo é que se preencha a mochila com o maior valor possível, não ultrapassando o peso máximo.

Este problema é a base do primeiro algoritmo de chave pública (chaves assimétricas).

2.1 O Algoritmo

Algorithm 2 solveP2(int n, int wi[], int vi[], int W)

```
Sort items by value per weight (vi[i] / wi[i]) in descending order
maxProfit ← 0                                   ▷ Highest profit for the problem
S ← Items[]                                     ▷ List of items added to the knapsack
queue ← PriorityQueue
queue ← rootNode                                ▷ (level = -1)
while queue is not empty do
    for each: Node ∈ queue
    Compute Cost Function for two children
    Compute Upper Bound for two children
    if node.children.upperBound > limit then
        Exclude Node
    else
        queue ← Node
    end if
end while

for eachitem do
    printitem[i]                                ▷ Prints the list of items
end for
return maxProfit
```

2.2 Análise do Algoritmo

O algoritmo usa a técnica de *Branch and Bound* para encontrar o subconjunto mais valioso dos itens que cabem na mochila.

Antes de tudo o algoritmo inicializa e prepara as estruturas as quais irá utilizar para o seu bom funcionamento.

Assim o código inicia criando uma fila ordenada pelo valor de cada item divido pelo seu respectivo peso, em ordem decrescente, com intuito de comçear pelo que possui maior valor por peso.

Com base nessa fila de prioridades ser fará uma escolha se cada item dessa lista será adicionado à mochila ou não. No código, optou-se por criar uma lista dizendo *true* para o caso de o item ser adicionado à mochila e *false* para o contrário. Essas duas listas se relacionam com base no index de cada lista, assim o item na posição i (da lista de itens) se relaciona com a lista de se foi ou não

posto na mochila que apresenta o mesmo index i, ou seja, seria o mesmo que dizer listaDeItens[i] listaDeSeColoca[i] são relacionados.

Após criado o nodo raiz, que não possui item posto na mochila, começa-se a percorrer a lista, indo no primeiro item da lista e criando um nodo que possui esse item e outro nodo que não possui esse item, calcula-se o limite superior de todos os nodos desse nível da árvore e o nodo que apresentar o maior limite superior é escolhido como o caminho que será continuado. Nesse nodo escolhido se fará o mesmo procedimento, criando dois nodos um com e um sem a adição do segundo item da lista e assim por diante até o fim da lista.

O algoritmo segue percorrendo a lista e só poda um ramo da árvore se esse ramo desrespeitar a viabilidade, o valor limite do nó não é melhor que o valor da melhor solução até então encontrada e o subconjunto de soluções viáveis apresente apenas um nó com estimativa inferior a de outro ramo, algo que permite ter uma percepção da solução final pela estimativa.

Por solução viável se entende que é um ponto no espaço de busca no problema que satisfaça todas as restrições do problema, nesse caso o peso limite da mochila não pode ser ultrapassado.

Cada nó da árvore representa um subconjunto dos itens dados e esse dado é utilizado para atualizar as informações sobre qual é o melhor conjunto visto até o momento após gerar cada novo nó na árvore.

Portanto o que algoritmo do *Branch and Bound* faz no problema da mochila, de forma resumida, é pegar uma lista de itens que podem ser postos na mochila e ordena por ordem de quem tem maior valor por peso pra quem tem menor valor por peso. Começando com nenhum item na mochila, passa a criar 2 caminhos, um com e outro sem esse primeiro item (ou o próximo da lista) e calcula qual dos caminhos apresenta uma estimativa possuir mais valor por item a ser posto na mochila, descartando caminhos que apresentem que não são o de maior estimativa e os caminhos que excedam o peso permitido da mochila.

Retornando uma lista de *true* e *false* informando qual item foi ou não posto na mochila, assim como o valor final do itens postos.

Essa abordagem é eficiente porque utiliza a poda de nós para reduzir o espaço de busca, explorando apenas as soluções promissoras.

Embora a complexidade no pior caso seja exponencial, na prática, a utilização de bons limites superiores permite resolver problemas de tamanho moderado de forma eficiente.

Isso se deve ao problerma da mochila ser um problema combinatório onde se deseja maximizar o valor dos itens colocados na mochila, respeitando uma capacidade máxima de peso e o algoritmo *Branch and Bound* explorar uma árvore de decisões em que cada nó representa uma solução parcial do problema, sendo necessário fatores limitantes para podar as partes da árvore que não aparentem conter uma solução ótima.

2.3 Implementação e Tempo de Execução

O tempo de execução do algoritmo *Branch and Bound* em seu pior caso é exponencial, $O(2^n)$ isso é devido a no pior caso ser quando todos os subconjuntos dos itens precisam ser explorados. Não obstante, devido à limitação do valor nos nós *Bound*, o algoritmo tende a ser muito mais eficiente do que a busca exaustiva.

A ordenação dos itens por valor/peso possui complexidade de O(log n).

A complexidade de exporação dos nós possui O(n) no seu pior caso, tendo em vista quantos nós são realmente explorados e quantos são limitados pelo algoritmo. Cada nó requer a atualização do valor acumulado e peso acumulado, bem como o cálculo do bound.

Por fim, as filas de prioridade apresentam uma complexidade de $O(\log k)$ tal que k é o número de nós na fila. No pior caso, k pode ser exponencial em n.

O código-fonte comentado referente ao problema 2 pode ser encontrado nos arquivos *.java* enviados junto desse relatório.