Trabalho Prático III - Algoritmos I

Lucas Braz Rossetti
2020041590

Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

lucasbraz@ufmg.br

Modelagem Computacional

O problema apresentado deixa claro que a plantação é alinhada, isto é, cada fileira de árvores possui o mesmo número de macieiras. Essa propriedade torna possível a representação do campo em uma matriz cujas entradas armazenam a quantidade de maçãs produzidas pela respectiva macieira. Também sabemos que a colheita é feita da primeira fileira até a última, de modo que os movimentos possíveis são: para baixo e pelas diagonais inferiores esquerda e direita.

Assim, a modelagem imediata do problema é achar a sequência de elementos da matriz em linhas subsequentes cuja soma é máxima, considerando as limitações de movimento. Logo, se a matriz é dada por $A = a_{ij}$, então o resultado será uma sequência de índices $j_1, j_2, \dots, j_k, \dots, j_n$ tais que o índice j_k se refere ao elemento a_{kj_k} e, além disso , a soma máxima será dada por

$$\sum_{k=1}^{n} a_{kj_k}$$

Descrição da Implementação

Como o problema apresentado pode ser enquadrado como uma questão de otimização, o paradigma de programação escolhido foi a programação dinâmica, uma vez que sua solução seria a mais natural dentre os outros paradigmas conhecidos. Além disso, também foi adotado o sentido bottom-up da formulação do problema.

Então, considerando a modelagem computacional apresentada anteriormente, seja F e W o número de linhas e colunas da matriz representativa $A[F][W]$, respectivamente. Queremos encontrar os índices das colunas de cada linha da matriz cuja soma dos elementos correspondentes equivale a soma máxima, para isso, devemos computar os trajetos de soma máxima para cada elemento inicial e escolher aquele que leva à maior soma. Os subproblemas implícitos se resumem a determinar a soma máxima para o trajeto que inicia de um elemento qualquer e termina no final da matriz, ou seja, podemos definir uma equação de Bellman como

> $OPT(i, j)$: soma máxima para um trajeto que inicia em $A[i][j]$ e termina na linha F-1

Note que, ao usarmos essa definição, podemos reconstruir o trajeto a partir da soma trivialmente, uma vez que temos apenas três casos possíveis de movimentação:

Se o elemento não está nas extremidades da matriz, isto é, $j \neq 0$ e $j \neq W - 1$, a soma máxima para o trajeto que inicia em $A[i][j]$ será definida pelas somas máximas dos trajetos imediatos à sua posição, ou seja, pelas diagonais esquerda, direita ou pela posição inferior. Lembre-se também que, como estamos usando o sentido bottom-up, consideramos a soma dos elementos de baixo:

$$OPT(i, j) = A[i, j] + max(OPT(i + 1, j - 1) , OPT(i + 1, j) , OPT(i + 1, j + 1))$$

Porém, se o elemento está na extremidade esquerda da matriz (j = 0) , temos

$$OPT(i, j) = A[i, j] + max(OPT(i + 1, j) , OPT(i + 1, j + 1))$$

De forma similar, se o elemento está na extremidade direita da matriz, teremos

$$OPT(i,j) \;=\; A[i,j] \;+\; max(OPT(i\,+\,1, j\,-\,1)\,,\,OPT(i\,+\,1, j)\,)$$

Ao final da construção da matriz de somas parciais, a soma máxima estará na sua primeira fileira, armazenada no índice que inicia o respectivo trajeto. Assumindo que A e M são as matrizes de dados e soma máxima parcial, respectivamente, o seguinte algoritmo descreve a determinação da soma máxima:

$FOR\ j \;=\; 0 \quad TO \quad W\,-\,1$:
$\qquad M[F\,-\,1][j] \;=\; A[F\,-\,1][j]$
$FOR\ i \;=\; F\,-\,2 \quad TO \quad 0$:
$\qquad FOR\ j \;=\; 0 \quad TO \quad W\,-\,1$:
$\qquad\qquad IF\ j \neq 0\ AND\ j \neq W\,-\,1$:
$\qquad\qquad\qquad M[i,j] \;=\; A[i,j] \;+$
$\qquad\qquad\qquad\qquad +\; max(M[i\,+\,1, j\,-\,1]\,,\,M[i\,+\,1, j]\,,\,M[i\,+\,1, j\,+\,1])$
$\qquad\qquad ELSE\ IF\ j \;=\; 0$:
$\qquad\qquad\qquad M[i,j] \;=\; A[i,j] \;+ max(M[i\,+\,1, j]\,,\,M[i\,+\,1, j\,+\,1])$
$\qquad\qquad ELSE\ IF\ j \;=\; W\,-\,1$:
$\qquad\qquad\qquad M[i,j] \;=\; A[i,j] \;+ max(M[i\,+\,1, j]\,,\,M[i\,+\,1, j\,-\,1])$

$RETURN\ max(M[0,j])\ FOR\ j \;=\; 0\ TO\ W\,-\,1$

Como já foi comentado, para determinar o trajeto final, basta começar pelo elemento da primeira linha com a maior soma e seguir a mesma lógica de movimentação, escolhendo sempre o elemento da linha inferior com valor máximo dentre as três (ou duas) opções.

O código foi implementado em três arquivos principais: “main.cpp” , “harvest.h” e “harvest.cpp” , além de um makefile. Os dois últimos se referem à classe Harvester, que gerencia todo o fluxo de funcionamento, desde a coleta e organização dos dados acerca da quantidade de maçãs até o processamento da soma máxima.

Esta classe possui três métodos relevantes, além de um construtor nulo: **readInput()** , que lê o arquivo de entrada de acordo com a formatação indicada e constrói a matriz de dados **field**, **maxApples()**, que faz o processamento sobre a matriz e retorna um vetor com o caminho de árvores cuja soma de maçãs é máxima, e, por último, **printPath()** , que executa os métodos anteriores e imprime a solução , dado um arquivo de entrada.

Em especial, o método **maxApples()** foi dividido em duas etapas: a primeira trata de aplicar a Equação de Bellman modelada anteriormente para encontrar a soma máxima de maçãs, enquanto a segunda etapa determina um caminho de índices a partir dos resultados da programação dinâmica.

Usando o sentido bottom-up, a função constrói a matriz de somas parciais **sumMirror** da mesma forma descrita no algoritmo apresentado. Um vetor **path** é criado com o objetivo de armazenar em path[i] o índice da coluna do elemento $field[i][j]$ que maximiza a soma , além disso, a soma máxima também é contida em seu último elemento. O primeiro índice é dado pelo elemento de maior valor da primeira linha de sumMirror.

Em seguida, para popular o vetor path com os índices corretos, a cada iteração considera-se apenas os movimentos possíveis a partir do último índice percorrido. Assim, a matriz sumMirror é consultada para escolher aquele que irá maximizar a soma final e atualiza-se um contador de soma **partSum** (ao final do loop, deve ser verdade que $partSum \ = \ sumMirror[0][path[0]]$).

A função retorna o vetor path ao final, que possui todas as informações a serem impressas pelo método printPath() em uma configuração formatada.

<u>**Análise de Complexidade**</u>

Considere F o número de fileiras de macieiras e W o número de macieiras por fileira no arquivo de entrada. Em “main.cpp”, constrói-se um objeto do tipo Harvester, que tem custo constante , e é executado o método **printPath()**. Assim, a complexidade total será dada por

$$T(W, F) \ = T(readInput()) \ + \ T(maxApples()) \ + \ T(impress\text{ão}\ do\ caminho)$$

Em $readInput()$, um loop itera pelas fileiras da matriz, enquanto outro loop interno itera por cada coluna, logo a complexidade da função será $O(F\ *\ W)$.

Já em $maxApples()$, a matriz com somas parciais sumMirror é atualizada de baixo pra cima, gerando um custo proporcional às suas dimensões, isto é , $O(F\ *\ W)$. Em seguida, o cálculo de path[0] é dado pelo maior elemento de sua primeira linha, que possui um custo linear em W , além disso, para que um trajeto seja construído, a matriz é percorrida em profundidade, passando por apenas um elemento de cada linha, gerando um custo linear em F. Portanto, podemos concluir que a complexidade total do método será

$$T(maxApples()) \ = \ O(F\ *\ W) + O(W) + O(F) \ = \ O(F\ *\ W)$$

Por último,a impressão do trajeto possui complexidade $O(F)$, uma vez que o vetor armazena um índice para cada linha da matriz. Assim, a complexidade de tempo total do programa é

$$T(W, F) \ = \ O(F\ *\ W) \ + \ O(F\ *\ W) \ + \ O(F) \ = \ O(F\ *\ W)$$