Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 02
Algoritmos I

João Pedro Moreira Smolinski

2024023996
jp.smolinski05@gmail.com

Conteúdo

1 Introdução

Este documento detalha a implementação do algoritmo computacional que consiste em analisar a infraestrutura da cidade de Triangulândia, desenvolvido como parte do Trabalho Prático 02 da disciplina de Algoritmos I (DCC206-2025/2). O projeto visa criar dois algoritmos capazes de encontrar a melhor solução para os dois projetos do prefeito, o novo muro da cidade e uma área de preservação ambiental.

Para tal, o prefeito da cidade, Roberto Ângulo, deseja encontrar a altura máxima do muro em formato de triângulo que vai ser construído na cidade. Além disso, deseja encontrar as três árvores na área verde da cidade que formem o triângulo de menor perímetro para a reserva.

A solução proposta, detalhada nas próximas seções, será um sistema em C++ que processa essas duas requisições em um tempo bastante eficiente. Este relatório abordará a modelagem, solução e complexidade da solução encontrada, além das considerações finais e referências bibliográficas utilizadas.

2 Modelagem

O enunciado exige complexidade não quadrática e uma solução gulosa para o primeiro problema e de divisão e conquista para o segundo.

Além disso, os dois problemas têm naturezas bastante distintas. Por isso, a explicação individual de cada um será bastante detalhada.

- **Problema 1 -** O problema 1, de encontrar o maior muro triangular, precisa de uma solução gulosa. É importante ressaltar que ordenar as pilhas para a solução não é possível, suas posições são fixas. Além disso, mesmo com falta de blocos, não é permitido, ao retirar um bloco de uma pilha, inserir em outra. Dessa forma a solução proposta foi um tratamento inicial das pilhas para inserir a solução gulosa.

  Sabe-se que se a maior pilha, P_i tem altura H, as pilhas P_{i-1} e P_{i+1} têm altura $H-1$. As alturas vão decaindo até as pilhas $P_{i-(H-1)}$ e $P_{i+(H-1)}$ que têm altura 1.

  Ou seja, a altura máxima do triângulo centrado em P_i depende do tamanho de P_i e das alturas máximas dos triângulos retângulos (escadas) à esquerda e à direita de P_i. Se P_{i-1} a P_{i-j} pode ser formatado como uma escada crescente com j de tamanho e P_{i+1} a P_{i+k} pode ser formatado como uma escada decrescente com k de tamanho, então a altura máxima do triângulo centrado em P_i vai ser o menor número entre $altura(P_i)$, j e k.

  Dessa forma, o processamento inicial vai, para cada pilha P_i, limitar sua altura máxima de acordo com a pilha P_{i-1} iterativamente (Sabendo que a pilha P_1, a mais a esquerda, tem altura máxima 1). Após isso, iniciando da pilha mais a direita, enquanto pudermos andar para a esquerda, pegaremos a altura máxima possível, limitando agora também pela direita. A pilha P_n também tem altura máxima 1, pela mesma limitação da P_1.

  A solução do triângulo máximo vai ser obtido pela escolha gulosa da pilha cujas limitações laterais fizeram ela ser a maior do conjunto.

- **Problema 2 -** O problema 2 exige uma solução de divisão e conquista. Afinal, comparar todos os conjuntos de três árvores possíveis teria custo cúbico.

  Para a modelagem desse problema, as árvores vão ser tratadas como pontos em um plano de forma que o triângulo de menor perímetro formado por três delas, vai ser a solução do problema. A modelagem é baseada em dividir as árvores pelas suas posições nos eixos X e Y do plano. Para o grande conjunto das árvores, a primeira metade com menor valor no eixo X é separada da segunda metade de maior valor no eixo X. Em cada subconjunto de pontos, calcula-se o menor perímetro dentro do subconjunto (Se necessário, acontece outra subdivisão, até um caso base). Então o menor perimetro entre os dois subconjuntos é pego.

  Após isso, resta verificar os triângulos que "cruzam" os conjuntos, ou seja, que tem pontos com X menor e maior do que o X mediano. Para evitar o custo alto de calcular cada triângulo que cruza a mediana do eixo, é necessário definir uma faixa de pontos aptos a compor um triângulo pequeno.

  Como o menor perímetro entre os dois subconjuntos já foi calculado, p_{min}, pontos que estiverem em uma distância em X de mais da metade do menor perímetro ($p_{min}/2$) já serão descartados. Como duas arestas do triângulo vão cruzar o eixo, o comprimento de "ir e voltar" desse ponto já vai ultrapassar p_{min}.

  Com os pontos selecionados dentro desta faixa, é possível ordená-los em relação a Y. Após essa ordenação, se a diferença em Y de um ponto a outro for maior do que $p_{min}/2$, a interação entre eles será descartada. Dessa forma, para cada ponto na faixa teremos uma faixa em Y de comprimento $p_{min}/2$ de pontos que podem formar o menor perímetro. Basta testar esses pontos e verificar se são menores, assim encontrando a solução para o problema.

3 Solução

Para a solução de cada um dos problemas optou-se pelo uso dos algorítmos que combinassem a melhor relação entre eficiência e simplicidade. Como os problemas são independentes, cada parte será explicada inteiramente.

3.1 Solução para o Muro

A lógica para a solução do problema é de dois laços. Como explicado anteriormente na modelagem, o primeiro laço faz a limitação do triângulo retângulo da esquerda da pilha. Ou seja, escolhemos o menor entre o tamanho da pilha e o tamanho da escada crescente à esquerda mais 1. Após isso, iterativamente a partir da pilha mais a direita, que vai receber altura máxima como 1, fazemos o processo espelhado. Andando para a esquerda, comparamos a altura máxima atual (que já era o mínimo entre o tamanho da pilha e da escada esquerda) com a escada da direita. Ao mesmo tempo, iniciamos uma variável de máximo com valor 1 e atualizamos cada vez que encontramos um valor maior.

O pseudocódigo de resolução do problema segue abaixo:

Algorithm 1 Encontrar o muro de maior tamanho - MaiorMuro($alturas$)

Entrada: Vetor $alturas$ com n elementos
Saída : A maior altura possível do muro
$alturas[0] \leftarrow 1$;
$alturas[n-1] \leftarrow 1$;
for $i \leftarrow 1$ **to** $n-1$ **do**
  $alturas[i] \leftarrow \min(alturas[i-1]+1, alturas[i])$;
$maiorAltura \leftarrow 1$;
for $i \leftarrow n-2$ **to** 1 **do**
  $alturas[i] \leftarrow \min(alturas[i+1]+1, alturas[i])$;
  $maiorAltura \leftarrow \max(alturas[i], maiorAltura)$;
return $maiorAltura$;

A decisão gulosa do algoritmo está em ajustar cada máximo localmente ao menor valor possível que respeite as restrições do vizinho à esquerda primeiramente e depois à direita, sem olhar o futuro global. A cada passo, escolhe a menor altura possível entre o limite atual e o imposto pelos vizinhos diretamente adjascentes, o que garante que a diferença entre posições consecutivas não exceda 1.

3.2 Solução para o Perímetro

Como comentado na modelagem, vamos alocar os pontos em um vetor e ordenar em relação a coordenada X. Depois disso vamos chamar de forma recursiva dividindo o vetor na metade até um caso base de 5 ou menos pontos (em que todas as combinações podem ser testadas pois o custo não vai subir tanto), unindo os menores perímetros de cada um dos subvetores. Além disso, calcularemos a faixa de pontos que podem ter triângulos cruzando o X mediano, ordenaremos em relação a Y e calcularemos os possíveis triângulos.

A lógica do laço de encontrar os possíveis triângulos dentro da faixa é simples. Como estão ordenados em relação a Y, para cada ponto P_i, percorre os outros cujo valor em Y seja maior ou igual, se a diferença entre o valor da coordenada Y de P_i e da coordenada Y de outro ponto for maior do que metade do menor perímetro calculado até o momento, P_i não vai mais encontrar um perímetro menor. Isso acontece pois a distância relativa a Y só vai aumentar para P_i. Da mesma forma, caso os pontos P_i e P_j já tenham sido escolhidos e o terceiro ponto P_k esteja sendo testado, se a distância em relação a Y de P_i e P_k for maior do que metade do menor perímetro calculado, o "link" P_i e P_j não vai retornar um perímetro menor.

No cálculo do perímetro, importante citar, é verificado se os pontos não são colineares. Como o problema trata de uma *rea* de preservação, os 3 pontos serem colineares resultaria em área 0, e não seria um triângulo e sim uma linha. A verificação é feita conferindo se a distância entre os pontos mais distantes é igual a soma das distâncias dos pontos mais distantes com o terceiro ponto, se for igual, são colineares.

O pseudocódigo de resolução do problema segue abaixo, a chamada $Perimetro(a,\ b,\ c)$ calcula o perímetro entre os 3 pontos (retorna ∞ caso seja colinear, para que não seja escolhido). Além disso o método $menor.atualiza(a,\ b,\ c)$ atualiza automaticamente o resultado ótimo, fazendo o desempate por ordem lexicográfica se necessário:

Algorithm 2 Encontrar o menor perímetro - MenorPerim(*pontos*)

Entrada: Vetor *pontos* com n elementos, ordenado em X
Saída : O menor perímetro possível entre 3 pontos

```
menor ← ∞;
if n ≤ 5 then
    for i ← 0 to n − 1 do
        for j ← i + 1 to n − 1 do
            for k ← j + 1 to n − 1 do
                p ← Perimetro(pontos[i], pontos[j], pontos[k]);
                if p ≤ menor.p then
                    menor.atualiza(pontos[i], pontos[j], pontos[k]);
resultadoEsquerda ← MenorPerim(pontos[0 : n/2]);
resultadoDireita ← MenorPerim(pontos[n/2 : n − 1]);
if resultadoDireita.p ≤ menor.p then
    menor.atualiza(resultadoDireita);
if resultadoEsquerda.p ≤ menor.p then
    menor.atualiza(resultadoEsquerda.);
faixa ← ∅;
for i ← 0 to n − 1 do
    if (| pontos[i].x − pontos[n/2].x |) < menor.p / 2 then
        faixa ← faixa + {pontos[i]};
faixa.ordenaY();
m ← faixa.tamanho();
for i ← 0 to m − 1 do
    for j ← i + 1 to m − 1 do
        if faixa[j].y − faixa[i].y ≥ menor.p / 2 then
            break;
        for k ← j + 1 to m − 1 do
            if faixa[k].y − faixa[i].y ≥ menor.p / 2 then
                break;
            p ← Perimetro(faixa[i], faixa[j], faixa[k]);
            if p ≤ menor.p then
                menor.atualiza(faixa[i], faixa[j], faixa[k]);
return menor;
```

Como é possível notar, a solução tem um claro caráter de Divisão e Conquista. Divide-se o vetor no meio, acha o melhor resultado em cada um dos subvetores e o melhor entre eles (Divisão). Após isso, verifica as possíveis soluções que cruzam os dois subvetores (Conquista).

4 Análise de Complexidade

Para fins de sintetização e evitar repetições necessárias de termos, a variável N será usada para se referir ao tamanho do vetor de pilhas da primeira parte e ao vetor de pontos da segunda parte. Além disso, a complexidade total do programa, tanto espacial quanto temporal, vai ser dada pela soma das complexidades dos dois problemas, já que são independentes e tratam de grandezas diferentes.

4.1 Problema 1 - Maior muro

Como ficou claro no pseudocódigo da seção anterior, o problema contém 2 laços independentes. Dentro de cada um desses laços, ambos de tamanho $O(N)$, são executadas operações de complexidade $O(1)$, como atribuições e *min* ou *max*. Dessa forma, a complexidade assintótica temporal dessa parte é $O(N + N) \in O(N)$.

Em caráter de memória, além da variável de máximo $O(1)$, é usado um vetor para armazenar os valores das pilhas $O(N)$. Ou seja, a complexidade assintótica espacial é $O(N)$.

4.2 Problema 2 - Menor Perímetro

Inicialmente, ordena todas os pontos (árvores) em relação a coordenada X. A ordenação é feita usando a função `std::sort` da biblioteca `algorithm`, que era permitida se usada com parsimônia (Não viola as restrições). Tal ordenação tem complexidade temporal $O(N \times \log(N))$ e espacial $O(\log(N))$.

Após a chamada recursiva, a função entra em uma relação de recorrência do tipo $T(N) = aT(N/b) + f(N)$. Nesse caso, a função chama recursivamente duas vezes, para metade do vetor, ou seja, $aT(N/b) = 2T(N/2)$. $f(N)$ representa o custo da função fora da chamada recursiva, que no caso, é dominado pela identificação e testes com a faixa de pontos. Construir a faixa é iterar sobre cada ponto, ou seja, tem complexidade $O(N)$ espacial e temporal. Além disso, é necessário ordenar a faixa pela variável Y, para evitar que os testes escalem a $O(N^3)$. A ordenação tem custo temporal $O(N \times \log(N))$ e espacial $O(\log(N))$. Este custo em particular depende do tamanho da faixa, mas podemos considerar que é na ordem de N.

Com a faixa selecionada e ordenada, o programa entra em um laço triplo. Isso aparenta ter complexidade $O(N^3)$, mas na realidade é muito menor. Graças a ordenação por Y os laços internos não executam mais do que um número pequeno de vezes C, tendo custo $O(N + C + C)$. Isso acontece, pois até o momento encontramos o menor perímetro entre os dois subvetores (p_{min}) e construímos a faixa a partir disso. Então para cada ponto dentro da faixa, há uma janela de interesse que contém os possíveis pontos que vai interagir. Essa janela tem dimensões $p_{min} \times (p_{min}\ /\ 2)$, e é impossível que uma grande parte de todos os N pontos estejam nessa janela (O que faria a complexidade $O(N^3)$).

Supondo que existissem n pontos dentro da janela, pelo menos $n/2$ pontos estariam em algum dos lados (divididos pelo X mediano). Porém em uma janela de tamanho $p_{min} \times (p_{min}\ /\ 2)$ haveriam ao menos $n/2$ pontos acumulados em uma janela $(p_{min}\ /\ 2) \times (p_{min}\ /\ 2)$. É inevitável, que para um n grande, a distância deles seja pequena. E para um n grande, haveria dentro dessa janela, algum triângulo de perímetro menor do que p_{min}. Isso seria uma contradição, já que se houvesse um triângulo menor em um dos lados, o perímetro deste é que seria p_{min}, o que comprova que na verdade, o laço triplo não é na ordem de $O(N^3)$. Como o limite geométrico imposto pelo p_{min} atua nos dois laços internos, esses vão rodar uma pequena quantidade de vezes, sendo uma constante muito pequena C. Por isso a complexidade do laço triplo fica $O(N \times C \times C) \in O(N)$. É um caso muito parecido com o problema do par de pontos mais próximo.

Dessa forma, $f(N)$ tem custo temporal $O(N + N \times \log(N) + N) \in O(N \times \log(N))$. A relação de recorrência fica $T(N) = 2T(N/2) + O(N \times \log(N))$. Podemos interpretar como uma árvore de recorrência que se expande para:

- **Nível 0:** Tamanho N, custo $O(N \times \log(N))$.
- **Nível 1:** Tamanhos $N/2$, custo $2 \times f(N/2) \in O(N \times \log(N/2)$.
- **...**
- **Profundidade:** O problema é dividido no meio em cada nível, então a árvore tem $O(\log(N))$ níveis.

A complexidade temporal então, pode ser calculada como $(Trabalho\ por\ nivel) \times (Niveis) \in O(N \times \log(N) \times \log(N)) \in O(N \times \log^2(N))$.

Para a complexidade espacial, a pilha de recusão vai ter tamanho $O(\log(N))$ e a soma de todos os vetores simultâneos é $O(N)$. As ordenações também tem custo $O(\log(N))$, totalizando $O(N)$ espacial.

5 Considerações Finais

Destarte, a implementação do trabalho foi muito proveitosa e serviu para colocar em prática as diferentes abordagens de resolução de problemas vistos em sala. A parte mais complexa foi a salto de pensar na faixa de pontos para o problema 2 e além disso ordenar pela coordenada Y, não foi um pensamento intuitivo a princípio. Apesar disso, o processo de pensar em um algoritmo guloso e um de divisão e conquista foi muito divertido e de extrema importância para sintetizar os aprendizados.

6 Referências

Referências

[1] Jon Kleinberg, e Eva Tardos. *Algorithm design.* Pearson Education.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, e Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009. Versão traduzida: *Algoritmos – Teoria e Prática*, 3ª Edição, Elsevier, 2012.

[3] Vinicius F. dos Santos, Jussara M. de Almeida, e Renato Vimieiro. *Algoritmos I.* Enunciado do Trabalho Prático 02, Departamento de Ciência da Computação, ICEx/UFMG.

[4] Wikipedia *Closest Pair of Points Problem* `https://en.wikipedia.org/wiki/Closest_pair_of_points_problem`