Trabalho Prático 2 - Algoritmos I

Eduardo Santos Birchal

Matrícula: 2024023970

1 Introdução

Esse trabalho engloba a resolução de dois problemas da cidade fictícia de Triangulândia: a construção do maior muro triangular isósceles possível a partir da remoção de tijolos de uma parede; e a identificação do triângulo de menor perímetro formado pelas árvores de um parque.

Para tal, foi necessário o uso de um algoritmo guloso para o primeiro problema, e de um algoritmo de divisão e conquista para o segundo. Nesse documento, serão detalhados os algoritmos usados, e será feita uma análise de complexidade para cada um.

2 Modelagem

2.1 Problema I

O problema de construir o maior muro triangular isósceles é equivalente ao problema de construir o maior triângulo isósceles a partir de uma lista L de inteiros, onde cada inteiro representa o número L_i de tijolos na coluna i. Só é permitido remover tijolos, ou seja, apenas é permitido fazer subtrações aos valores (não se pode somar a L_i nem permutar colunas).

Para encontrar o maior triângulo em L, foi feito um algoritmo guloso que calcula o maior triângulo isósceles centrado em L_i, para cada i. Isso foi feito a partir de "cortes": cada coluna é reduzida primeiro ao ápice da maior diagonal superior que pode ser construída com ápice nela, e depois ao ápice da menor diagonal inferior que pode ser construída com ápice nela. Dessa forma, faz-se dois "cortes" na coluna: um diagonal superior, e um diagonal inferior, ambos com seu ápice em L_i; resta um triângulo isósceles, o maior possível na posição i.

Após todas as colunas serem reduzidas ao maior triângulo isósceles em i, torna-se aparente que o maior L_i é o maior triângulo isósceles em L.

2.2 Problema II

Modelando-se as árvores do parque como pontos (x, y) em um espaço bidimensional, pode-se resolver o problema da menor cerca usando um problema de triângulo de menor perímetro, por um método de divisão e conquista.

Primeiramente, o conjunto de n pontos é pré-processado, criando duas listas: uma ordenada por x (P_x) e outra por y (P_y). O algoritmo então divide recursivamente o conjunto de pontos ao meio por uma linha vertical, e encontra o triângulo de perímetro mínimo em cada metade. Seja p o menor perímetro encontrado nessas duas chamadas recursivas.

Um triângulo com perímetro menor que p pode existir com vértices em ambas as metades, mas todos os seus três vértices devem estar a uma distância de no máximo $p/2$ da linha de divisão vertical (senão, haveriam duas linhas maiores que $p/2$, cuja soma seria portanto maior que p). Isso cria uma faixa vertical de largura p no centro, contendo os pontos candidatos.

O algoritmo itera pelos pontos da faixa ordenados por y. Cada ponto só precisa ser comparado com um número constante de vizinhos na lista ordenada por y.

Finalmente, o problema exigia que, em caso de perímetros iguais, o triângulo com os índices originais lexicograficamente menores fosse escolhido. Isso foi implementado armazenando o índice original de cada ponto (de 1 a Z) junto com suas coordenadas. Em todas as etapas de comparação, se dois perímetros p_1 e p_2 fossem iguais, os conjuntos de índices dos triângulos correspondentes eram comparados lexicograficamente para realizar o desempate.

3 Solução

3.1 Problema I

A altura de um triângulo centrado na coluna i é limitada por três fatores: a altura original L_i, o tamanho da maior diagonal superior, e o tamanho da maior diagonal inferior.

O algoritmo implementa isso em duas passagens sobre a lista L de tamanho n, modificando-a no lugar:

1. A função `limit_to_max_upper` itera de $i = 1$ até $n - 1$. Para cada coluna, ela atualiza a altura L_i para ser o mínimo entre sua altura atual e a altura da coluna anterior mais um $(L_{i-1} + 1)$. Isso garante que L_i não é maior que a maior diagonal que pode ser formada a partir da esquerda. O caso base é $L_0 = \min(L_0, 1)$.

2. A função `limit_to_max_lower` faz o reverso. Ela itera de $i = n - 2$ até 0, atualizando L_i (que já foi limitada pela passagem anterior) para ser o mínimo entre seu valor atual e o valor da coluna à sua direita mais um $(L_{i+1} + 1)$. O caso base é $L_{n-1} = \min(L_{n-1}, 1)$.

Após as duas passagens, cada posição L_i na lista contém o valor da altura do maior triângulo isósceles que pode ser centrado em i. A solução final é simplesmente o maior valor encontrado na lista L após ela ser modificada.

Pode-se notar que o algoritmo é guloso: ele busca apenas maximizar os resultados dos cortes diagonais superior e inferior.

3.2 Problema II

A função principal dessa solução é `find_min_perimeter_triangle`, que recebe uma lista de tuplas no formato `((x, y), indice_original)`.

1. **Pré-processamento:** Antes de iniciar a recursão, o algoritmo lida com pontos duplicados. Como o critério de desempate é o menor índice lexicográfico, a lista é ordenada por (x, y, indice) e, em seguida, filtrada para manter apenas a primeira ocorrência de cada coordenada (x, y) única.

2. **Ordenação Inicial:** O algoritmo cria duas listas: P_x, com os pontos únicos ordenados por x, e P_y, com os pontos únicos ordenados por y.

3. **Função Recursiva:** A função `find_min_perimeter_recursive(Px, Py)` (que será chamada de rec) implementa a lógica principal:

   - **Caso Base:** Se o número de pontos n for pequeno (ex: $n \leq 5$), o problema é resolvido por força bruta, que verifica todos os triângulos possíveis.
   - **Divisão:** Encontra-se a mediana x_{med} e dividem-se as listas P_x e P_y em metades esquerda e direita (P_{xL}, P_{xR}, P_{yL}, P_{yR}).
   - **Conquista:** Duas chamadas recursivas são feitas: $(p_L, \text{idx}_L) = \text{rec}(P_{xL}, P_{yL})$ e $(p_R, \text{idx}_R) = \text{rec}(P_{xR}, P_{yR})$.
   - **Combinação (Metades):** O perímetro mínimo p_{min} é selecionado entre p_L e p_R. O critério de desempate lexicográfico $\text{idx}_L < \text{idx}_R$ é aplicado se $p_L = p_R$.
   - **Combinação (Faixa):** Uma "faixa" é criada contendo todos os pontos de P_y cuja distância x até x_{med} é menor que $p_{min}/2$.
   - **Busca na Faixa:** A função `find_subregion_min` é chamada. Ela itera por cada ponto p_i na faixa e o compara apenas com os pontos p_j, p_k seguintes que estejam a uma distância y menor que $p_{min}/2$. Isso otimiza a busca de $O(n^2)$ para $O(n)$, pois o número de vizinhos a verificar é constante. O desempate lexicográfico também é aplicado nesta busca.

O resultado final é a tupla de índices $(A1, A2, A3)$ retornada pela chamada inicial.
Disponível abaixo está o pseudocódigo para essa solução:

```
funcao find_min_perimeter_triangle(pontos_com_indices):
    pontos_unicos = remove_duplicatas_por_coordenada(pontos_com_indices)
    se len(pontos_unicos) < 3:
        retorne (infinito, Nulo)

    Px = ordena_por_x(pontos_unicos)
    Py = ordena_por_y(pontos_unicos)

    retorne find_min_perimeter_recursive(Px, Py)

funcao find_min_perimeter_recursive(Px, Py):
    n = len(Px)
    se n <= 5:
        retorne brute_force_min_perimeter(Px)

    (PxL, PxR, PyL, PyR, x_mediano) = divide_listas(Px, Py)

    (p_esq, idx_esq) = find_min_perimeter_recursive(PxL, PyL)
    (p_dir, idx_dir) = find_min_perimeter_recursive(PxR, PyR)

    (min_p, best_idx) = compara_e_escolhe_menor(p_esq, idx_esq, p_dir, idx_dir)

    faixa_central = []
    para ponto em Py:
        se abs(ponto.x - x_mediano) < min_p / 2:
            faixa_central.append(ponto)

    retorne find_subregion_min(faixa_central, min_p, best_idx)

funcao find_subregion_min(faixa, min_p_atual, best_idx_atual):
    n = len(faixa)
    para i de 0 até n-1:
        para j de i+1 até n-1:
            se faixa[j].y - faixa[i].y >= min_p_atual / 2:
                break
            para k de j+1 até n-1:
                se faixa[k].y - faixa[j].y >= min_p_atual / 2:
                    break

                p = perimetro(faixa[i], faixa[j], faixa[k])
                idx_atuais = sorted_indices(faixa[i], faixa[j], faixa[k])

                (min_p_atual, best_idx_atual) =
                    compara_e_escolhe_menor(p, idx_atuais, min_p_atual, best_idx_atual)

    retorne (min_p_atual, best_idx_atual)
```

4 Análise de Complexidade

4.1 Problema I

A solução é composta por três passagens sobre a lista.

1. A função `limit_to_max_upper` executa um único loop de $i = 1$ até $n - 1$. Cada iteração realiza operações de tempo $O(1)$. Portanto, a complexidade desta função é $O(n)$.

2. A função `limit_to_max_lower` executa um único loop de $i = n - 2$ até 0. Similarmente, cada iteração é $O(1)$, e a complexidade total da função é $O(n)$.

3. A função `max_triangle` chama as duas funções acima e, em seguida, executa a função `max(L)`. A busca pelo valor máximo em uma lista de tamanho n também tem complexidade $O(n)$.

A complexidade de tempo é a soma dessas operações sequenciais: $T(n) = O(n)+O(n)+O(n) = O(n)$.
A complexidade de espaço auxiliar é $O(1)$, pois o algoritmo modifica a lista original, sem alocar novas estruturas de dados proporcionais ao tamanho da entrada n.

4.2 Problema II

A solução é composta por um pré-processamento e uma função recursiva. Seja Z o número total de pontos e n o número de pontos com coordenadas únicas.

1. **Pré-processamento:**
   (a) Ordena-se a lista inicial de Z pontos para identificar duplicatas e manter o menor índice. Esta ordenação tem complexidade $O(Z \log Z)$. A filtragem subsequente para criar a lista de n pontos únicos é $O(Z)$.
   (b) As duas listas de n pontos únicos, P_x e P_y, são criadas. Isso requer duas ordenações, custando $O(n \log n)$.
2. **Recursão:**
   - A complexidade da função `find_min_perimeter_recursive` é descrita por $T(n) = 2T(n/2)+ O(n)$.
   - $2T(n/2)$ representa as duas chamadas recursivas para as metades esquerda e direita.
   - $O(n)$ representa o custo da etapa de combinação:
     (a) Dividir a lista P_y em P_{yL} e P_{yR}, o que exige uma varredura $O(n)$.
     (b) Criar a faixa central, o que exige outra varredura $O(n)$ de P_y.
     (c) A busca na faixa. Embora tenha três loops aninhados no caso base, na faixa otimizada, ela é $O(n)$. Isso ocorre pois o loop externo itera n vezes (sobre os pontos da faixa), mas os loops internos comparam cada ponto apenas com um número constante de vizinhos (devido à otimização da distância $p_{min}/2$).
   - Pelo Teorema Mestre, a recorrência $T(n) = 2T(n/2) + O(n)$ resolve para $T(n) = O(n \log n)$.

A complexidade de tempo é a soma do pré-processamento e da recursão: $O(Z \log Z) + O(n \log n) + O(n \log n)$. Como $n \leq Z$, o termo dominante é $O(Z \log Z)$.

A complexidade de espaço é $O(Z)$, visto que:

1. A lista inicial de Z árvores ocupa $O(Z)$.
2. As listas de pontos únicos P_x e P_y ocupam $O(n)$ de espaço.
3. A pilha de recursão atinge uma profundidade de $O(\log n)$. Em cada nível da recursão, são criadas listas auxiliares (como P_{yL}, P_{yR} e a faixa central) que somam $O(n)$ em tamanho. Como essas listas são liberadas após o retorno da chamada, o espaço auxiliar máximo usado pela pilha de recursão em qualquer momento é $O(n)$.

O espaço total é, portanto, $O(Z) + O(n) + O(n) = O(Z)$, pois $n \leq Z$.

5 Considerações Finais

Esse trabalho demonstrou que os paradigmas aprendidos em Algoritmos I são úteis para resolver uma grande classe de problemas. Aplicando o raciocínio guloso ou de divisão e conquista, pode-se solucionar problemas aplicáveis no mundo real de forma surpreendentemente eficiente. Resolver o Problema II demandou a generalização do famoso problema do par de pontos mais próximo, que foi um raciocínio não-trivial; ainda assim, sua realização mostra que problemas conhecidos são base para problemas novos, e que princípios como o da divisão e conquista se aplicam em inúmeros casos.

Referências

[1] Jon Kleinberg, Éva Tardos. *Algorithm Design*. Pearson Education, 1st edition, 2005.

[2] Shih-Yu Wu. *Algorithms StudyNote — 4: Divide and Conquer — Closest Pair*. https://medium.com/@shihyu-wu/algorithms-studynote-4-divide-and-conquer-closest-pair-49ba679ce3c7

[3] Wikipedia. *Closest pair of points problem*. https://en.wikipedia.org/wiki/Closest_pair_of_points_problem