DCC206 – Algoritmos 1

Aula 14 – Dividir e Conquistar – Parte 3

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- Na aula de hoje, veremos outros exemplos de problemas que são resolvíveis com a estratégia de dividir-e-conquistar
- Os problemas que estudaremos tem diversas aplicações em várias áreas para além da computação
- Alguns estão relacionados a problemas de uma área da computação conhecida como geometria computacional
- Veremos que o maior desafio desses problemas, assim como ocorreu anteriormente, é projetar a fase de conquista dos algoritmos, isto é, como combinar as soluções dos subproblemas para obter a solução final

Encontrar um par de pontos mais próximo

- O primeiro problema que iremos investigar é relacionado a geometria computacional
- A área de geometria computacional lida com algoritmos eficientes para solucionar problemas geométricos
  - A implementação direta de soluções analíticas podem resultar em abordagens ineficientes e sujeitas a imprecisões numéricas
- Especificamente, dado um conjunto de n pontos no plano, queremos descobrir o par (ou melhor, um par) de pontos mais próximos entre si

Encontrar um par de pontos mais próximo

- Naturalmente, a primeira pergunta que fazemos é como medir a distância entre dois pontos
- Existem diferentes formas de se medir, no nosso caso, vamos considerar a distância euclidiana (distância em linha reta)
  - $d(a,b) = \sqrt{(b.x - a.x)^2 + (b.y - a.y)^2}$
- Assim, queremos descobrir o par de pontos (distintos) de um conjunto P tal que a distância dada pela função d seja a menor possível

Encontrar um par de pontos mais próximo

- A abordagem ingênua consiste em avaliar todos os pares e escolher aquele com a menor distância
- Essa abordagem, contudo, possui custo quadrático
- A pergunta que surge é: será que conseguimos um algoritmo com custo menor do que esse?
- Note que, como temos um número quadrático de pares para avaliar, qualquer algoritmo que se baseie no valor explícito das distâncias terminará com esse custo
- De fato, por muitos anos, esse problema foi considerado com custo $\Omega(n^2)$
- Somente na década de 1970 que Shamos e Hoey provaram a existência de um algoritmo com custo O(n log n)

Encontrar um par de pontos mais próximo

- Assim como no problema de contar inversões, devemos estruturar o problema de forma a não computar explicitamente a distância entre todos os pares de pontos do conjunto
- A forma como eles abordaram o problema foi através da estratégia dividir-e-conquistar
- Para entender a intuição, vamos considerar o problema em uma dimensão
- Nesse caso, queremos encontrar o par de pontos mais próximos numa reta
  - Ou seja, dado um vetor de pontos, queremos encontrar os dois elementos mais próximos entre si

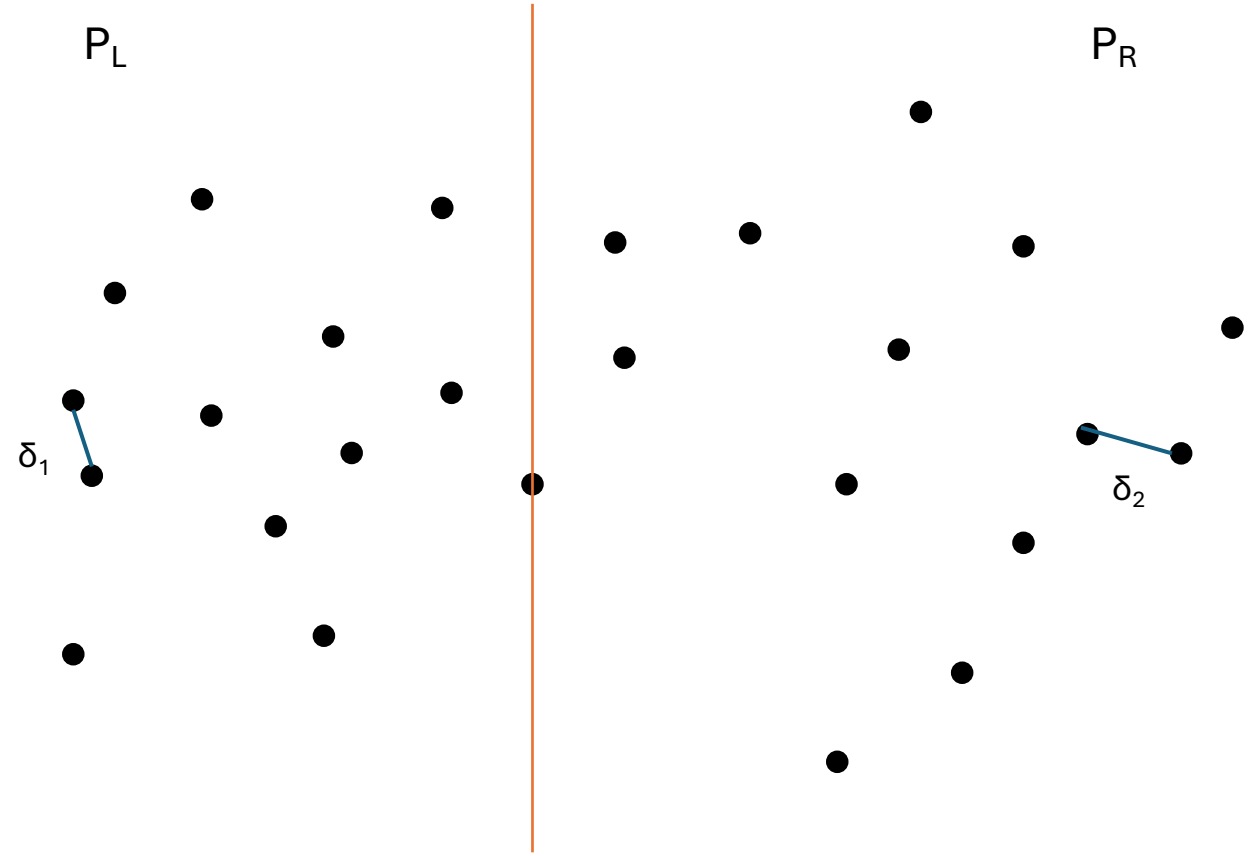
Encontrar um par de pontos mais próximo

- Como não temos qualquer informação sobre a disposição dos pontos no vetor (reta), a única forma de resolver o problema é efetivamente computando as distâncias entre um elemento e todos os demais
- No entanto, se ordenarmos os elementos, então o par de pontos mais próximos nesse conjunto obrigatoriamente terá que ser dois elementos consecutivos
- O desafio agora é como extrapolar essa lógica para duas dimensões

Encontrar um par de pontos mais próximo

- Seguindo a lógica de dividir-e-conquistar, podemos construir a solução dividindo o conjunto de pontos na metade, resolvendo o problema de cada metade, e combinando a solução
- Ou seja, podemos quebrar o conjunto P em duas metades, P_L e P_R, computar o par de pontos mais próximo em cada conjunto, e depois combinar o resultado
- O desafio nesse caso é como combinar os resultados uma vez que os dois subproblemas tenham sido resolvidos
  - A menor distância será uma das três: a menor de P_L, a menor de P_R, ou a menor entre os pontos de um lado e do outro

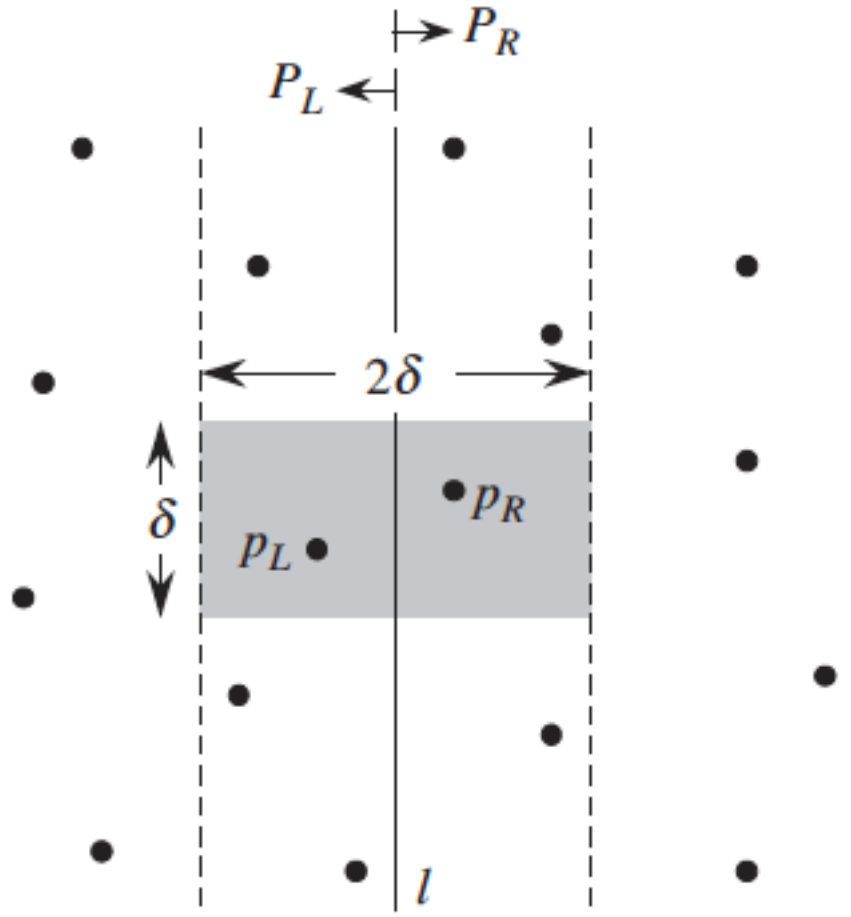
Encontra par de pontos mais próximo



Encontrar par de pontos mais próximo

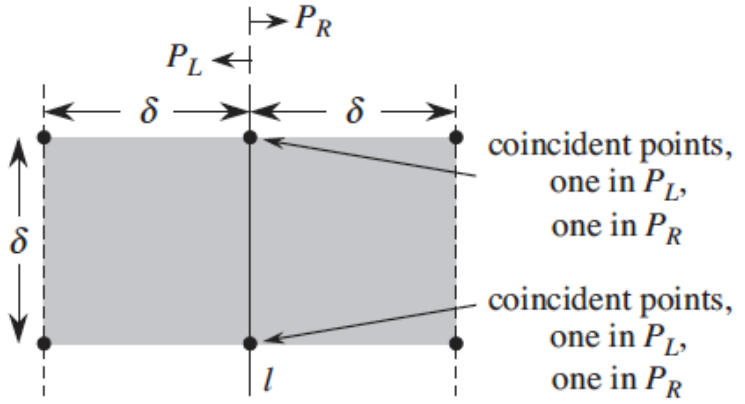
- O problema é que, em princípio, computar a menor distância entre um ponto de P_L e outro de P_R não é um problema mais fácil que computar a menor distância de pontos em P
- Porém, agora, temos uma informação adicional: só nos interessa avaliar os pontos que estão a uma distância menor que $\delta = \min(\delta_1, \delta_2)$ – as distâncias dos subproblemas da esquerda e direita
- Considerando que existe uma linha l que separa os pontos das duas metades, estamos restritos somente àqueles que estão a uma distância menor que δ dessa linha (mediana da coordenada x)
  - Ou seja, estamos considerando os pontos dentro de um retângulo δx2δ centrado em l

Encontrar par de pontos mais próximo



Encontrar par de pontos mais próximo

- Novamente, isso não ajuda muito, já que esse retângulo poderia conter todos os pontos
- No entanto, podemos mostrar que esse retângulo contém, no máximo, 8 pontos de P
- A distância entre os pontos de uma mesma partição é, pelo menos, δ
- Logo, podemos ter, no máximo, 4 pontos em um dos lados (metades δxδ) desse retângulo
- Sobre a linha, podem existir também, no máximo, 4 pontos, dois coincidentes de em cima e dois embaixo



Encontrar par de pontos mais próximo

- Dessa forma, cada ponto é comparado a, no máximo, 7 outros pontos
- Nosso problema, agora, é que não sabemos quais são esses pontos
- Também não sabemos como dividir rapidamente os pontos em duas metades
  - Na verdade, sabemos que devemos encontrar a mediana da coordenada x, mas calculá-la em cada subproblema pode aumentar a complexidade geral da solução
  - Se os pontos de P estivessem ordenados pela coordenada x, o custo de encontrar a mediana é $\Theta(1)$

Encontrar par de pontos mais próximo

- Podemos manter os pontos ordenados ao longo da execução do algoritmo
- Inicialmente, duplicamos o conjunto P (vetor de pontos), e ordenamos pelas coordenadas x e y, obtendo P_x e P_y
- Para dividir o problema em duas metades, encontramos o elemento da posição n/2 de P_x, e geramos P_{Lx} e P_{Rx}
- Para gerar P_{Ly} e P_{Ry}, percorremos P_y e comparamos a coordenada x de cada ponto avaliado com a mediana, se for menor ou igual, ele é inserido em P_{Ly}, caso contrário, em P_{Ry}
  - Note que os novos conjuntos também estarão ordenados
- O custo dessa operação é O(n)

Encontrar par de pontos mais próximo

- Para combinar as soluções, compute o valor de δ
- Em seguida, gere um conjunto Y contendo os pontos de Py que estejam dentro do retângulo δ x 2δ
  - Isso é feito em tempo linear
- Agora, para cada ponto em Y, calcule sua distância de cada um dos 7 pontos consecutivos
  - Caso a distância entre eles seja menor que δ, guarde o par e a distância
- Ao final, retorne o par com a menor distância entre os três valores computados

Encontrar par de pontos mais próximo

- O custo do nosso algoritmo é $T'(n) = T(n) + O(n \log n)$
  - Temos a chamada inicial para ordenar os pontos
  - Em seguida, executamos o algoritmo dividir-e-conquistar
- $T(n) = 2T(n/2) + cn$
- Assim, pelo teorema mestre, $T(n) = \Theta(n \log n)$
- Portanto, o custo do algoritmo é $O(n \log n)$

Problema do vetor de soma máxima

- Considere o seguinte problema: você se considera um especialista no mundo do futebol. Todos os dias entre no AlgBet para apostar nos resultados dos jogos. Naturalmente, você mais perde do que ganha, mas seus amigos desejam saber qual foi o maior lucro que você obteve jogando (mesmo que seja momentâneo). Como resolver esse problema?
- Se generalizarmos nosso problema, recebemos um vetor com n valores e queremos encontrar a subsequência com a maior soma de valores
- Exemplo:
  - A = [20, -30, 15, -10, 30, -20, -30, 30]
  - Possui soma máxima 35

Problema do vetor de soma máxima

- Obviamente, se o vetor tiver somente valores não-negativos, então a maior soma é a soma de todos os seus valores
- Como isso nem sempre acontece, precisamos desenvolver uma estratégia para computar esse valor
- Assim como no exemplo do conjunto de pontos, a solução ingênua tem custo quadrático
  - Investigamos todas as subsequências possíveis
- Vamos estudar se uma abordagem dividir-e-conquistar reduziria esse custo

Problema do vetor de soma máxima

- Dado um vetor A[esq..dir], a ideia da estratégia é dividi-lo em dois subproblemas menores
  - A[esq..meio] e A[meio+1..dir]
- Resolvemos cada um dos subproblemas e combinamos as soluções
- O desafio nesse problema é que a subsequência pode A[i..j] pode ocorrer em qualquer posição de A[esq..dir]
- Como dividimos A em duas partes, temos que:
  - A[i..j] pode estar totalmente na primeira parte, i.e. $esq \leq i \leq j \leq meio$
  - A[i,j] pode estar totalmente na segunda parte, i.e. $meio+1 \leq i \leq j \leq dir$
  - A[i,j] pode cruzar as duas metades, i.e. $esq \leq i < j \leq dir$

Problema do vetor de soma máxima

- Os dois primeiros casos são resolvidos exatamente pelas chamadas recursivas do algoritmo
- O terceiro caso é o equivalente à combinação dos resultados
- Nosso objetivo é combinar os resultados com o menor custo possível
- Vamos computá-la com tempo linear

Problema do vetor de soma máxima

- Novamente, computar o terceiro caso não é mais fácil que computar o problema como um todo
- Porém, agora, temos a restrição que a subsequência tem que cruzar as metades
- Dessa forma, o problema é redefinido como encontrar as subsequências A[i..meio] e A[meio+1..j] que resultem na soma máxima
- Logo, podemos encontrar a subsequência A[i..meio] de maior valor, depois encontrar A[meio+1..j] de maior valor, e combiná-las

Problema do vetor de soma máxima

- Isso pode ser facilmente computado iterando sobre os elementos de A[esq..meio] iniciando pela sequência A[i=meio..meio] e decrementando o valor de i para aumentar a sequência
- Sempre que uma sequência tiver valor da soma maior do que o atual, armazenamos esse valor/índices e continuamos a exploração
- O mesmo é feito de maneira análoga com a metade da direita

Problema do vetor de soma máxima

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Problema do vetor de soma máxima

- É notável que a função anterior computa a subsequência que cruza as metades com maior soma em tempo O(n)
- Logo, temos o custo total do algoritmo é $T(n) = 2T(n/2) + cn$
- Pelo teorema mestre, $T(n) = \Theta(n \log n)$

Leitura

- Seção 5.4 Kleinberg e Tardos
- Seção 4.1 e 33.3 CLRS