Algoritmo para otimizar a utilização de tinta em obras cubistas

Gabriel Weich[*], João Paulo Medeiros Cecilio[†]
Faculdade de Informática — PUCRS

13 de outubro de 2018

Resumo

Este artigo descreve alternativas de solução para o primeiro problema proposto na disciplina de Algoritmos e Estruturas de Dados II no terceiro semestre, que trata do cálculo da área restante de cada cor em um plano cartesiano após sucessivas adições de retângulos com medidas e cores variadas. São apresentados alguns caminhos para a solução do problema e um deles, que é a nossa solução final, é analisado mais detalhadamente. Em seguida são mostrados os resultados obtidos para diferentes casos de teste.

Introdução

O problema descrito na proposição deste trabalho é de, dadas as coordenadas (x,y) de retângulos identificados por diferentes cores e após sussessivas sobreposições desses em um plano cartesiano, calcular qual seria a área final somada de cada cor. Para tal foram fornecidas listas de testes com as coordenadas (x,y) dos cantos inferior esquerdo e superior direito de cada retângulo com sua cor indicada.

Com base nos conceitos aprendidos em aula sobre algoritimos, serão testadas algumas soluções possíveis para o problema, analisadas as suas complexidades e tempo de execução, para ao final apontar e detalhar de forma mais aprofundada a melhor solução encontrada, assim como as conclusões obtidas a partir dessas tentativas.

1 Primeiras soluções

1.1 Uma abordagem simples

Inicialmente, nos propusemos a testar o mais óbvio que seria efetivamente preencher a informação de cada retângulo em alguma estrutura de dados para a partir daí somar a área de cada cor. Assim poderíamos verificar problemas que já eram esperados ou até problemas inesperados, sobre os quais soluções futuras mais complexas teriam também de passar.

Para tal, já procuramos uma solução inicial que tivesse uma performance mais adequada. Utilizamos o pacote *numpy* da linguagem de programação Python. Esse pacote é voltado para operações eficientes em grandes matrizes, além de possuir fácil portabilidade para C++.

O algoritimo inicial consistiria em:

[*]gabriel.weich@acad.pucrs.br
[†]joao.cecilio@acad.pucrs.br

1. Inverter a lista de retângulos, do último (prioritário) para o primeiro.

2. Representar uma matriz em numpy quer seria a área de desenho (64000x64000).

3. Sucessivamente verificar se a posição está livre e marcar na estrutura a cor do ponto.

O primeiro resultado que obtivemos falhou logo no início pois ao tentar criar uma estrutura de dados única com essa dimensão, o numpy não possibilitou nem a alocação de memória para tal estrutura. Tentamos alterar o tipo de dado do array, mas a menor unidade que conseguimos foram booleanos e mesmo assim não foi possível.

1.2 Insistindo no simples

Pesquisando descobrimos uma biblioteca Python chamada bitarray que possibilitava a criação de arrays de bits. Essa biblioteca é implementada em C++ e possui os seguintes limites de tamanho para cada array: 2^{34} para sistemas 32 bits e 2^{63} para sistemas 64 bits, o que atendia nossa proposição.

A proposição seria realizar a mesma operação de desenhar ponto a ponto, verificando a área já desenhada, mas com instruções lógicas bit a bit. Com isso, esperava-se que mesmo sendo uma solução bastante simples, seria muito performática.

Abaixo, a versão mais otimizada para a principal operação do algoritimo, que era a de gravar na área de desenho e computar as áreas (foram omitidas demais operações de carregamento dos casos de teste e etc):

```
1  import bitarray as ba
2  area_desenho = [ba.bitarray(64000) for i in range (0 ,64000)]
3  [i.setall(False) for i in area_desenho]
4
5  for i in range(0 ,len(retangulos)):
6      ret2 = [(retangulos[i][0] ,retangulos[i][1]) ,(retangulos[i][2] ,retangulos[i][3])]
7      array = ba.bitarray(64000)
8      array.setall(False)
9      array[ret2[0][0]:ret2[1][0]+1] = True
10     for j in range(ret2[0][1] ,ret2[1][1]+1):
11         b = array^area_desenho[j]&array
12         area_desenho[j] = area_desenho[j] | b
13         cores[retangulos[i][4]] = cores[retangulos[i][4]] + b.count()
```

1.3 Conclusões e resultados preliminares da primeira solução

O algoritmo acima, resolve o problema proposto, mas com uma complexidade O(n^2), e leva em média **30s** para computar o primeiro caso de teste, de 100 amostras. Algumas otimizações compilando para diretamente o programa para C++ e até mesmo reescrevendo diretamente nessa segunda linguagem, melhorou a performance para 9s.

Uma das expectativas era de que a operação bitwise fosse muito rápida. No entanto, investigando como o python realizava essas operações e como estava implementanda a biblioteca bitarray, descobrimos que es operações são feitas em pacotes de 8 bits por vez, uma das razões da demora.

Apesar de conseguir resolver o problema, o algoritimo não possui a performance adequada para solucionar o maior caso de testes de cem mil retângulos em um tempo viável. Esse tempo de execução varia em n^2, sendo n o número de retângulos, e pode levar ainda mais tempo dependendo de outra variável: o tamanho dos retângulos.

1.4 Reconsiderando o problema

Depois de reconsiderar o problema, buscamos encontrar uma solução para trabalhar somente com as coordenadas. Nessa solução o programa recebia as coordenadas do retângulo a ser inserido e fazia uma verificação com todos os outros retângulos já inseridos no plano a fim de encontrar os retângulos em que ocorre sobreposição e então calcular a quantidade de área sobreposta e descontar do resultado final.

Ao considerar essa solução, deveríamos encontrar as diversas maneiras que um retângulo pode se sobrepor a outro, pois o cálculo da área sobreposta é diferente para cada caso. Para isso desenvolvemos dezenas de verificações que faziam esse trabalho.

O algoritmo, porém, funcionava apenas para os casos em que a sobreposição era sobre retângulos que ainda não haviam sido sobrepostos, visto que para os outros casos o algoritmo teria que trabalhar com sobreposição de sobreposição, e para esses casos não encontramos solução viável.

Além disso, tal algoritmo também não teria boa performance, pois continuava sendo $O(n^2)$. Isso porque a cada novo retângulo inserido deveria ser feita uma verificação com todos os outros já inseridos no plano.

2 A solução final

Decidimos então buscar outra solução, mudando a abordagem sobre o problema. Continuamos com a ideia de inserir os retângulos em órdem inversa, ou seja, de trás para frente, pois os últimos retângulos da lista, os primeiros a serem inseridos, estariam sempre se sobrepondo aos subsequentes.

Dessa forma, a cada novo retângulo inserido bastaria calcular a área que ficou disponível para esse retângulo, ou seja, a área que ainda não foi ocupada por algum retângulo inserido anteriormente, e então adicionar essa área ao resultado.

No entanto, dessa vez, procuramos otimizar a forma de cálculo da área ocupada por cada retângulo.

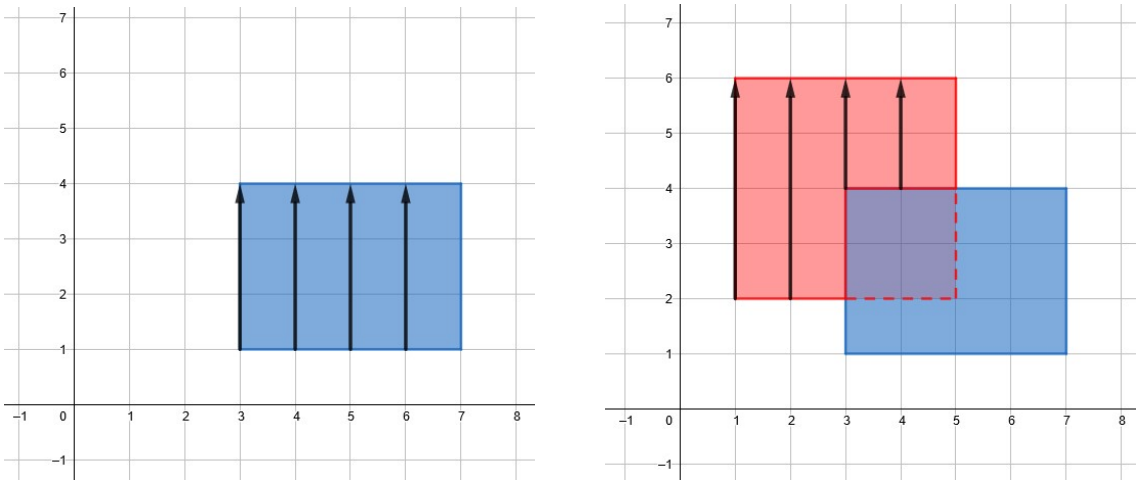
A cada inserção, precisaríamos demarcar a área ocupada por cada retângulo, para isso, a cada retângulo inserido é feito uma iteração sobre toda a sua largura, ponto a ponto, e adicionado o intervalo correspondente a sua altura em uma das posições de um vetor de cinquenta mil posições, chamado aqui de vetor principal, o qual representa todos os pontos do eixo das abscissas.

Ao adicionarmos um intervalo em determinado ponto X do vetor principal, verificamos os intervalos já existentes nesse vetor, e calculamos então quanto do novo intervalo está disponível nesse ponto (quanto não pertence a outros intervalos), adicionamos esse resultado à lista de cores e incluímos o novo intervalo no ponto.

2.1 Ilustração do algoritmo

A figura acima ilustra a inserção de dois retângulos. Para inserir o primeiro retângulo (último da lista) iremos iterar sobre o eixo das abscissas com um contador indo de 3 até 6 e inserindo nessas mesmas posições do vetor principal um segundo vetor de valor [1, 4] que representa a altura inserida do retângulo (y1 e y2). Como todos os pontos estavam vazios o algoritmo vai adicionar + 3 na cor azul em cada uma das 4 iteração, resultando num total de 12 unidades de área acrescentadas à cor azul.

Para inserir o retângulo vermelho o algoritmo também irá iterar sobre o eixo X, indo de 1 até 4. Na primeira e na segunda iteração todas as posições estão disponíveis, portanto ele irá adicionar o intervalo [2, 6] às posições 1 e 2 do vetor principal e incrementar o vermelho em + 4 a cada iteração. Já na terceira e quarta iteração o intervalo não está mais vazio, o algoritmo vai então ajustar o intervalo das posições 3 e 4 para [1, 6] e adicionar + 2 no vermelho a cada iteração.



(a) Primeiro retângulo (b) Segundo retângulo

Figura 1: Dois retângulos sendo inseridos no plano

2.2 Código e detalhamento do algoritmo

O algoritmo irá inserir os intervalo em um determinado ponto X numa estrutura de linked list, sendo cada intervalo um nodo e cada nodo guardando uma referência para o próximo nodo.

Segue abaixo o método que adiciona um novo intervalo à lista, esse método é chamado a cada iteração pelo eixo X de um retângulo e retorna a área inserida.

```
1
2  procedimento adicionaIntervalo(int y0 , int y1)
3
4      // Se a lista está vazia insere o nodo no início
5      se inicio = null então
6          inicio = Nodo(y0 , y1)
7          retorna y1 −y0
8      fim
9
10     Nodo nodo ← inicio
11     Nodo aux ← inicio
12     int area ← 0
13
14     // Iteramos sobre a lista até encontrar o nodo com intervalo final imediatamente acima do y0
15     // ou até chegar ao final da lista
16     enquanto nodo.proximo ≠ null faça
17         se nodo.final > y0 então
18             break
19         fim
20         aux ← nodo
21         nodo ← nodo.proximo
22     fim
23
24     // Verifica se o novo intervalo está inteiramente dentro de outro intervalo já existente
25     se nodo.inicio ⩽ y0 and nodo.final ⩾ y1 então
26         retorna 0 ;
27     fim
28
```

```
29
30      // Verifica se o novo intervalo está antes do começo do próximo intervalo já existente
31      se nodo.inicio > y1 então
32
33          // Insere antes do primeiro intervalo
34          se nodo = inicio então
35              inicio ← Nodo(y0 , y1)
36              inicio.proximo ← aux
37
38          // Insere entre dois nodos
39          senão faça
40              aux.proximo ← novo Nodo(y0 ,y1)
41              aux.proximo.proximo = nodo
42          fim
43
44          retorna y1−y0
45      fim
46
47      // Insere no final da lista
48      se nodo.final < y0 então
49          nodo.proximo = novo Nodo(y0 ,y1)
50          retorna y1−y0
51      fim
52
53      // Se o novo intervalo começa antes do próximo nodo e segue adiante , então estende
54      // o início do próximo nodo até o início do novo intervalo (y0)
55      se nodo.inicio ⩾ y0 então
56          area ← area + (nodo.inicio −y0)
57          nodo.inicio ← y0
58      fim
59
60      Nodo intervalo = nodo
61
62      // Continuamos a iterar sobre a lista até encontrarmos a posição final do novo
63      // intervalo (y1) ou até o final da lista
64      enquanto nodo.proximo ≠ null faça
65
66          // Interrompe o iteração ao encontrar o nodo com final imediatamente acima do y1
67          se nodo.final > y1 então
68              break
69          fim
70
71          se intervalo ≠ nodo então
72              area ← area + (nodo.inicio −aux.final)
73          fim
74
75          aux ← nodo
76          nodo ← nodo.proximo
77      fim
78
```

```
79      // Se o próximo nodo começa depois do término do novo intervalo
80      se nodo.incio > y1 então
81          area ← area + (y1 −aux.final)
82          intervalo.final ← y1
83          invervalo.proximo ← nodo
84          retorna area
85      fim
86
87      // Se o final do novo intervalo está em um nodo diferente do início
88      se intervalo ≠ nodo então
89          area ← area + (nodo.inicio −aux.final)
90      fim
91
92      intervalo.proximo = nodo.proximo
93
94      // Se o final do novo intervalo está depois do término do último nodo
95      se nodo.final ⩽ y1 então
96          area ← area + (y1 −nodo.final)
97          intervalo.final = y1
98
99      // Estende o intervalo até o final do nodo em que se encontra o y1 do novo intervalo
100     senão faça
101         intervalo.final = nodo.final
102     fim
103
104     retorna area
105 fim
```

Resultados

Depois de implementar o algoritmo acima na linguagem Java e executar diferentes casos de teste, obtivemos os seguintes resultados:

Tabela 1: Resultados

	100	1.000	10.000	50.000	100.000
verde-claro	83.494.896	294.628.818	26.527.940	197.218.193	69.823.182
vermelho	177.104.759	41.308.626	430.834.907	19.697.714	233.144.402
azul-claro	239.927.075	72.960.681	17.175.578	161.008.111	19.719.680
amarelo	55.867.740	43.734.300	19.881.385	168.262.252	194.578.814
verde-escuro	118.533.604	125.751.102	348.353.443	127.164.023	119.277.132
marrom	101.156.394	751.955.626	190.891.102	179.134.256	192.074.167
azul-escuro	424.094.008	23.134.350	883.701.524	53.280.155	25.242.169
cinza	49.203.160	456.533.372	248.460.304	1.177.336.248	111.023.519
dourado	102.449.481	83.800.170	192.982.112	247.561.886	465.688.136
violeta	126.990	381.298.990	86.347.762	52.688.362	727.899.501
preto	294.224.403	135.850.986	18.682.727	8.251.078	122.098.760
laranja	646.273.801	71.553.660	30.620.362	107.777.871	219.125.373
tempo(ms)	26	253	2.755	13.657	26.250

Conclusões

Através das soluções desenvolvidas, conseguimos evouir na abordagem do problema a fim de encontrar a mais adequada.

Todas elas, de alguma forma, contribuiram para que encontrássemos a solução final, pois com elas tivemos uma notável evolução na compreensão do problema.

As soluções apresentadas possuem, em geral, uma ideia relativamente simples por trás, no entanto a real implementação das soluções mostrou-se bastante dificultosa, principalmente porque foi preciso pensar em todos os casos de interseção e como lidar com eles.

Inicialmente, procuramos uma solução mais simples, com o objetivo de esbarrar nos principais problemas que um algoritimo poderia ter ao tentar resolver esse problema.

Percebemos, que por mais que tivéssemos estratégias de performance para tornar rápida a operação de comparação de todos os retângulos ponto a ponto, não seriam soluções viáveis, por isso partimos para uma abordagem diferente.

A última solução mostrou-se bastante satisfatória, pois implementa uma ideia simples com um algoritmo claro e com performance regular.

A complexidade do algoritmo da solução final não se baseia apenas na quantidade de retângulos mas também na largura média deles, visto que a cada retângulo inserido temos que iterar sobre toda a sua largura. Sendo assim, a complexidade final do algoritmo é O(n.m) em que "n"é a quantidade de retângulos e "m"a média da largura dos retângulos. Com base nos diferentes casos de teste é possível ainda verificar um crescimento linear no tempo de execução do algoritmo.

Mesmo assim, concluimos que nesse formato conseguimos ter um tempo muito mais satisfatório que a operação O(n^2), levando ao entendimento final de que nem sempre estratégias para tornar mais rápidas operações complexas são eficientes, e que é preciso repensar o problema e procurar analisá-lo por todos ângulos em busca da melhor abordagem.

Referências

[1] Cormen, T. H.; Leiserson, E. C.; Rivest, R. L.: **"Introduction to Algorithms"**. Mc-Graw Hill Book Co., The MIT Electrical Engineering and Computer Science Series, Cambridge, 1990.

[2] NumPy <http://www.numpy.org/> Acesso em 14 de abril de 2018.

[3] Python Software Foundation - efficient arrays of booleans – C extension <https://pypi.org/project/bitarray/> Acesso em 14 de abril de 2018.

[4] An Extensive Examination of Data Structures Using C 2.0 - <https://msdn.microsoft.com/en-us/library/ms379575(v=vs.80).aspx> Acesso em 14 de abril de 2018.