DOCUMENTAÇÃO TP2 – AEDS II

ALUNO: Bruno Maletta Monteiro
MATRÍCULA: 2017015150
DATA: 03/12/2017

1. Introdução

O projeto constituiu-se de um plano cartesiano, delimitado por um quadrado, que contém estrelas, cada uma localizada em um ponto e possuindo um peso associado. A operação que pode ser feita ao plano é dividi-lo em quatro quadrantes, e dividir os quadrantes em quatro novos e menores quadrantes, e assim sucessivamente.

O objetivo dessas divisões é deixar cada quadrante com no máximo uma estrela. Após as divisões dos quadrantes, representadas por uma árvore quádrupla, dever-se-ia printar os centros (e pesos) dos quadrantes em ordem *postorder*, isso é, os filhos antes do nó pai, de forma que o último quadrante a ser exibido é sempre o quadrante raíz.

O peso de cada quadrante é a soma dos pesos das estrelas nele contidas. Além disso, todo o projeto foi trabalhado com divisão inteira, de forma que o menor quadrante possível tem tamanho 1x1 e estrelas localizadas no mesmo ponto são consideradas como uma só, e seu peso é a soma dos pesos das estrelas lá localizadas.

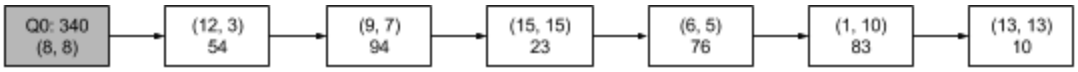
2. Desenvolvimento

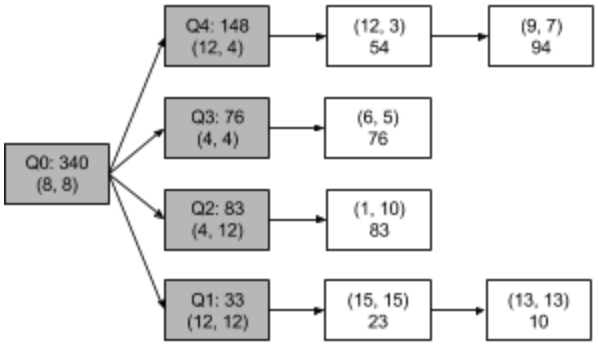
Para resolver o problema, além de utilizar uma árvore quádrupla para representar os quadrantes, implementei uma lista encadeada para cada nó da árvore, que representa as estrelas contidas no quadrante. Assim, para dividir as estrelas para os quadrantes filhos, só é preciso iterar pela lista de estrelas, e colocar cada estrela no quadrante filho correspondente, o que pode ser feito sem alocação extra de memória, só redirecionando os ponteiros, de forma que a lista do quadrante pai termina vazia após a divisão.

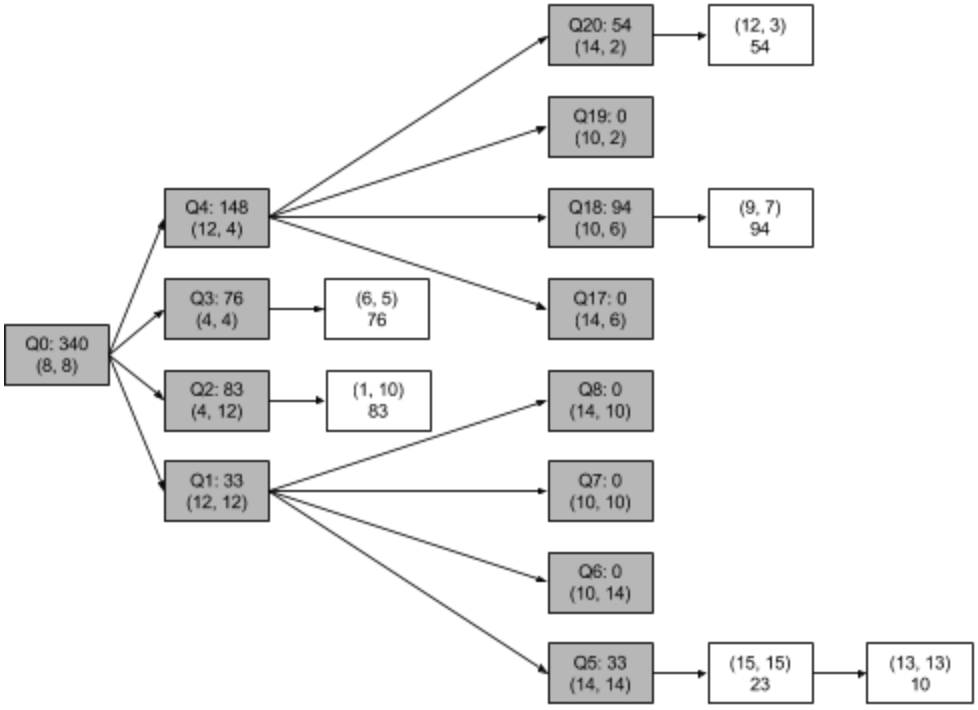
Os quadrantes são divididos recursivamente até que o quadrante atual possua no máximo uma estrela, ou tenha tamanho mínimo, 1x1. Por exemplo, para o seguinte caso de teste:

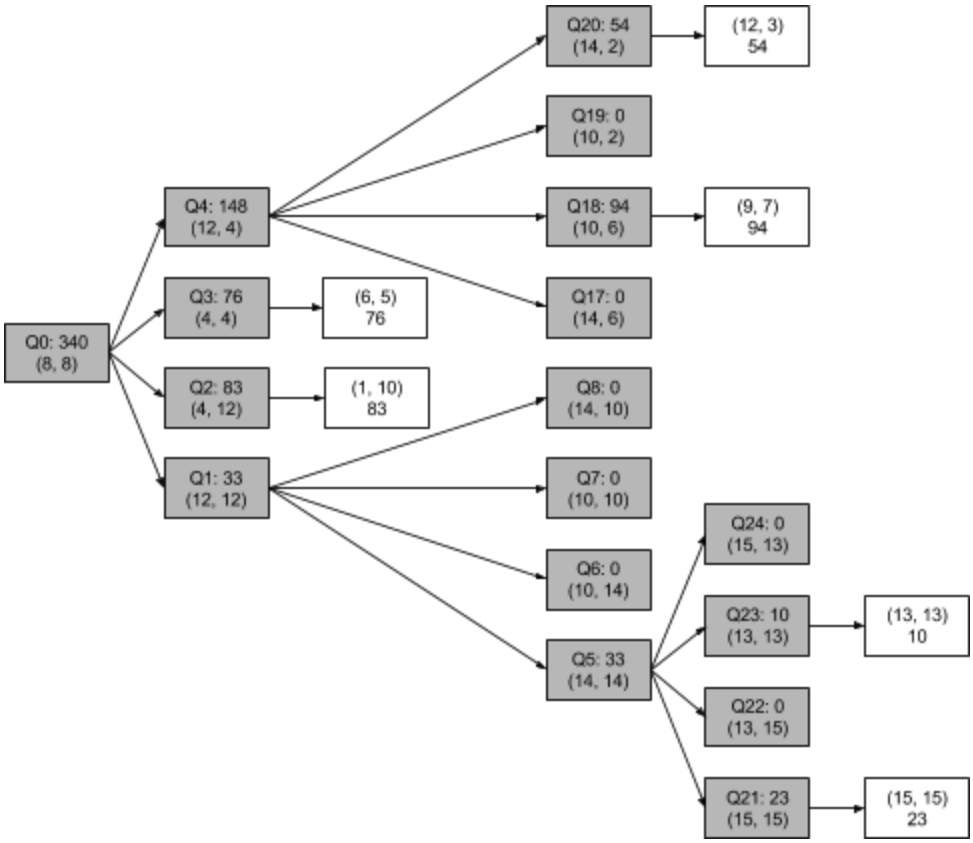
16 6
12 3 54
9 7 94
15 15 23
6 5 76
1 10 83
13 13 10

O programa funcionará da seguinte forma, em cada nível de profundidade:









3. Exemplo

Como exemplo, gerei o seguinte caso de teste:

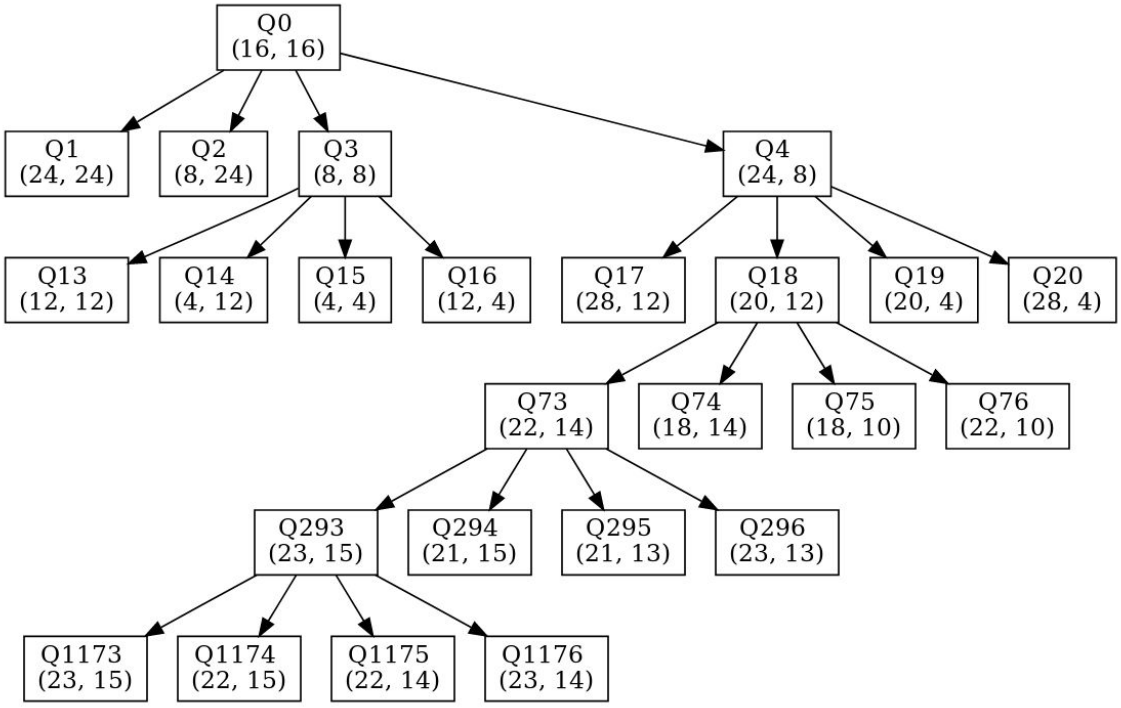
```
32 10
2 2 43
4 15 1
23 14 37
22 14 45
19 6 87
18 9 83
27 0 73
15 19 88
16 14 98
23 31 95
```

Ao executar o meu programa, a saída correspondente foi:

```
(24, 24) 95
(8, 24) 88
(12, 12) 0
```

(4, 12) 1
(4, 4) 43
(12, 4) 0
(8, 8) 44
(28, 12) 0
(23, 15) 0
(22, 15) 0
(22, 14) 45
(23, 14) 37
(23, 15) 82
(21, 15) 0
(21, 13) 0
(23, 13) 0
(22, 14) 82
(18, 14) 98
(18, 10) 83
(22, 10) 0
(20, 12) 263
(20, 4) 87
(28, 4) 73
(24, 8) 423
(16, 16) 650

Que representa a seguinte árvore:



4. Análise de Complexidade

O meu algoritmo faz quatro coisas principais: lê a entrada, particiona os quadrantes, printa a árvore, e libera a memória. Vamos analisar a complexidade de cada uma dessas operações separadamente.

a) **Leitura dos dados**: Para ler a entrada, o programa recebe primeiro os valores de N e L, e depois lê cada uma das N estrelas. Assim, sua complexidade é: $O(N)$.

b) **Particionamento de quadrantes**: Para entender a complexidade do particionamento dos quadrantes, precisamos saber primeiro o número máximo de nós da árvore. Como os critérios para as chamadas recursivas são o número de estrelas do quadrante, e, finalmente, o tamanho do mesmo, no pior caso, cada estrela é "dupla", ou seja, há duas estrelas em cada posição, o que acarretará o particionamento máximo, até que o quadrante fique de tamanho 1x1.Nesse cenário de pior caso, cada par de estrelas causará a criação de um nó de profundidade máxima, que é $log_4(L^2) = log_2 L$. Portanto, no pior caso cada estrela gera $O(\log(L))$ nós na árvore, de forma que o número de nós da árvore é $O(N \log(L))$, pois há $O(N)$ pares de estrelas. Mas, além disso, há a função *divide_estrelas(Quadrante*)*, que é chamada em cada quadrante. Para analisar sua complexidade, devemos notar que ela itera linearmente pelo número de estrelas no quadrante, que, em cada altura da árvore, soma-se, no máximo, em N. Portanto, ela é $O(N)$ para cada altura da árvore, e a altura total da árvore é $O(\log(L))$. Assim, a complexidade de dividir as estrelas é, no total, $O(N \log(L))$. Finalmente, a complexidade total de particionar os quadrantes é $O(N \log(L)) + O(N \log(L)) = O(N \log(L))$.

c) **Exibição dos dados**: Como há $O(N \log(L))$ nós na árvore, printar a árvore possui a complexidade $O(N \log(L))$.

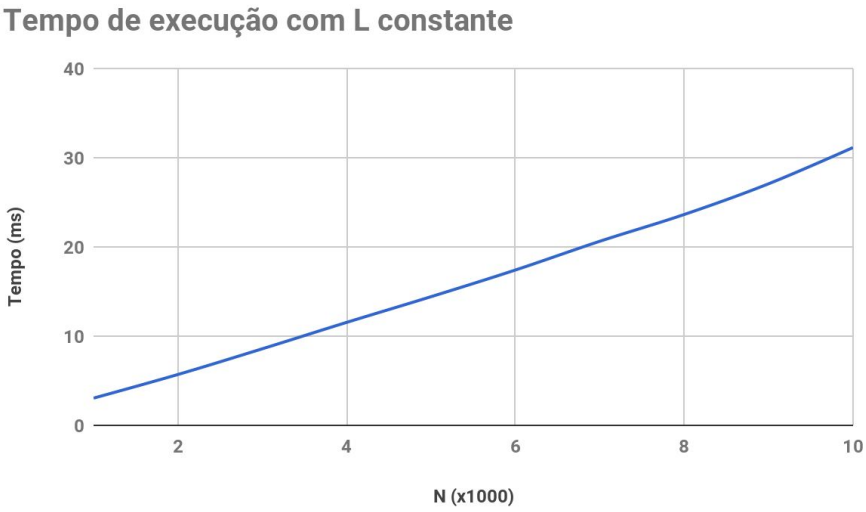
d) **Liberação de memória**: Libera os nós das árvores, em $O(N \log(L))$, e as listas de estrelas, que somam em N células. Assim, liberar a memória é $O(N \log(L)) + O(N) = O(N \log(L))$.

Com isso, vemos que a complexidade total do algoritmo é $O(N \log(L))$.

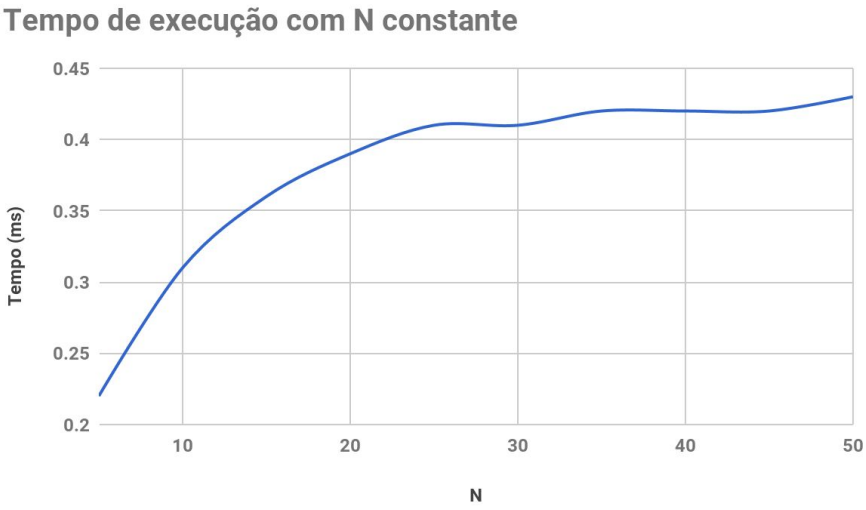
5. Análise Experimental

Para fazer a análise experimental, testei separadamente a influência de *N* e *L*. Para isso, iniciei deixando *L* constante e variando apenas *N*. Testei, para cada tamanho de entrada, 100 entradas diferentes, e fiz uma média aritmética dos tempos de execução, e fiz o mesmo para o *L*. Obtive os seguintes valores:

N	Tempo(ms)
1000	3.05
2000	5.70
3000	8.58
4000	11.54
5000	14.42
6000	17.41
7000	20.63
8000	23.63
9000	27.07
10000	31.13



L	Tempo(ms)
5	0.22
10	0.31
15	0.36
20	0.39
25	0.41
30	0.41
35	0.42
40	0.42
45	0.42
50	0.43



6. Conclusão

Vemos que a estrutura de dados escolhida é muito importante para uma boa complexidade do algoritmo. A utilização de um vetor, no lugar de uma lista encadeada, por exemplo, necessitaria de custo extra de alocação de memória para os quadrantes filhos. Ter um vetor (ou lista) de estrelas totais, e iterar por ele para cada quadrante teria complexidade $O(N\,L^2)$, o que é muito pior. Por esses motivos escolhi implementar com lista encadeada, o que faz o custo de memória para as partições ser menor, e garante complexidade $O(N \log(L))$.

O projeto foi de simples implementação, e seus maiores desafios foi a escolha de uma estrutura de dados adequada e sua análise de complexidade.

Além disso, observa-se que a análise experimental mostrou os resultados esperados: a complexidade do algoritmo varia linearmente com o número de estrelas, e de forma logarítmica em relação ao tamanho do quadrante inicial.