Trabalho Prático 0

Quadtree

Aluno: Enzo Roiz

Matrícula: 2013062839

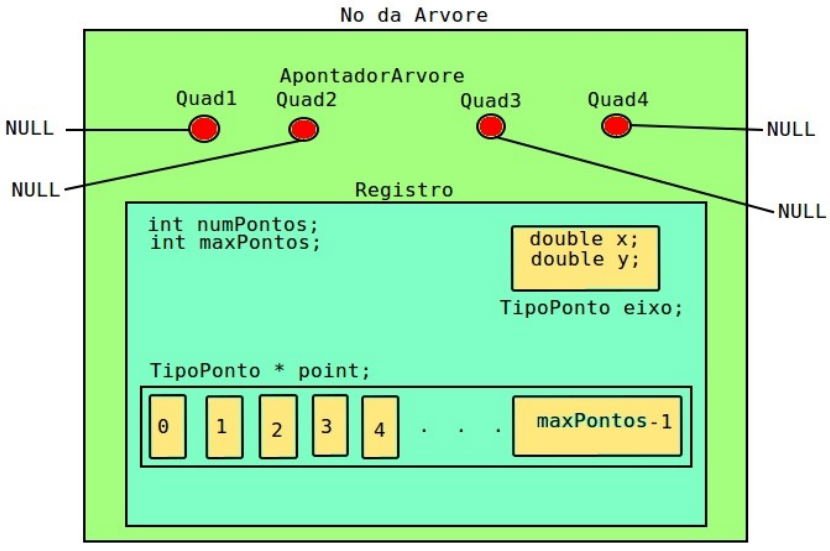
Índice

Descrição do problema

A Quadtree é uma estrutura de dados usada para representar pontos em uma região bidimensional e tem por base a divisão de um quadrante em quatro outros sub-quadrantes toda vez que o quadrante inicial exceder o máximo de pontos permitidos, realocando os pontos deste para os sub-quadrantes criados. Assim a área onde os pontos estão inseridos será sempre subdividida em outros quadrantes de modo que em cada um deles haja no máximo o número de pontos permitidos. Neste trabalho, representaremos pontos em um espaço utilizando Quadtree. O programa deverá ler um arquivo contendo informações necessárias para a montagem da Quadtree e também áreas a serem consultadas. Como saída, o programa terá o número de quadrantes em que a região bidimensional foi dividida, além dos pontos encontrados nas áreas determinadas pelas consultas.

Implementação

Para este trabalho, a estrutura de dados utilizada foi a chamada Quadtree. É como uma árvore binária, porém em vez de dois ponteiros para os filhos, tem quatro ponteiros, sendo um para cada filho do quadrante. Para a montagem da estrutura, temos um tipo *Nó*, com ponteiros para os quadrantes filhos *Quad1, Quad2, Quad3, Quad4* e uma variável do tipo *Registro*. No tipo *Registro*, temos as características de cada quadrante, sendo elas o número de pontos já inseridos no quadrante *numPontos*, o número máximo de pontos que podem ser inseridos *maxPontos*, o eixo do quadrante que é o ponto central do quadrante *TipoPonto eixo*, e um ponteiro para *TipoPonto * point*, que, através de malloc(), reservará uma área de memória com *maxPontos* váriáveis do *TipoPonto,* que contém as coordenadas “x” e “y” de um ponto. Podemos ver um esquema da estrutura de dados na figura abaixo.



Funções

Para a implementação do trabalho foram usadas as seguintes funções:

void PegaArgumentos(int , char **, char **, char **):

Recebe como parâmetros argc: número de argumentos com os quais a função main foi chamada na linha de comando. **argv da matriz de strings onde estão armazenados o nome do programa, os argumentos passados pela linha de comando, além de ** entrada e **saida, strings onde serão armazenados os nomes dos arquivos de entrada e saída passados pela linha de comando.

void Inicializa(ApontadorArvore *);

Recebe um ponteiro para árvore e o inicializa como *NULL* para que este seja a raiz da árvore.

void InsereArvore(Registro, ApontadorArvore *);

Recebe um registro e um ponteiro para árvore. Cria um quadrante e insere nele o registro com suas características, como a mais importante delas os quatro ponteiros para os sub-quadrantes.

void InserePonto(TipoPonto, ApontadorArvore *, int, int, int);

Recebe um ponto e um ponteiro pra árvore, a altura da árvore e os limites “x” e “y” da área onde serão inseridos os pontos. Nesta função o ponto será inserido no seu quadrante correspondente. O ponto principal do programa é a montagem da árvore. O esquema para inserção de cada ponto é o seguinte:

Função de inserção():

O quadrante apontado é uma folha ?

Se sim:

A lista de pontos está cheia ?

Se sim:

Cria os quatro quadrantes filhos com suas respectivas características.

Redireciona os pontos da lista para seus filhos.

Chama a Função de Inserção para o ponto lido no respectivo quadrante filho

Se não: //Lista Cheia.

Insere o ponto na lista de pontos do quadrante.

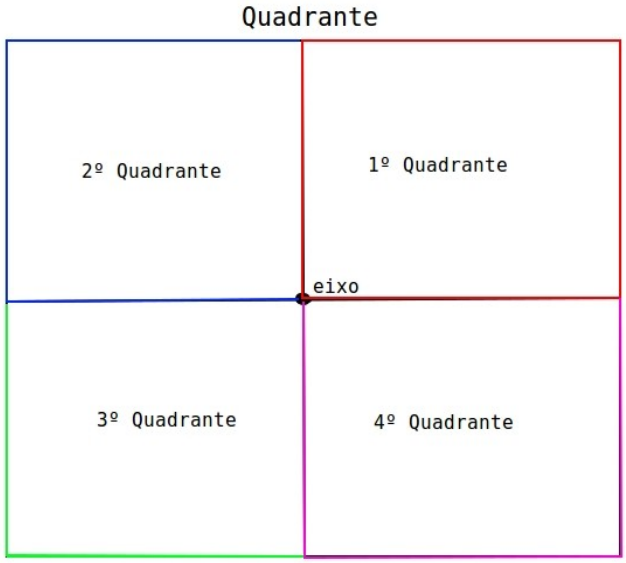
Se não: //Quadrante não é folha.

Chama a Função de Inserção para o ponto e seu respectivo quadrante até que o quadrante seja uma folha da Quadtree.

Para o ponto “andar” na árvore e encontrar o quadrante certo para ser inserido foi adotado um sistema de “eixos”, que são pontos centrais dos quadrantes. A altura da árvore anteriormente citada é utilizada para determinar estes eixos. Considerando a altura “h” da Quadtree, e que cada quadrante é subdivido ao meio no eixo “x” e no eixo “y”, a menor distância do eixo às bordas do quadrante podem ser obtidas pela fórmula *limiteDaÁreaObservada/2^h*, onde o limite da área observada são os limites “x” e “y” especificados no arquivo de entrada. Então os eixos dos quadrantes filhos, são definidos sob a forma:

eixoQuadFilho = eixoQuadPai + ou – limiteDaÁreaObservada/2^(h+1);

O “+ ou -” da fórmula acima se dá devido a cada quadrante ter uma avaliação diferente.



Por exemplo:

Sendo *limiteX*=64 e *limiteY*=64; os eixos anteriores = 0;

Para h=0; O quadrante então terá eixo x=0 +64/2^1 e y =0 + 64/2^1; logo (32,32)

Para h=1; Os quadrantes então terão eixos:

1º quadrante: x=32 + 64/2^2 e y=32 + 64/2^2; logo (48,48)

2º quadrante: x=32 - 64/2^2 e y=64/2^2; logo (16,48)

3º quadrante: x=32 - 64/2^2 e y=32 - 64/2^2; logo (16,16)

4º quadrante: x=32 + 64/2^2 e y= 32 - 64/2^2; logo (48,16)

Assim, quando vamos inserir um ponto na Quadtree, caso o quadrante analisado não seja uma folha da Quadtree, então fazemos:

x ponto >= x eixo ?

Se sim:

y ponto >= y eixo ?

Se sim:

ponto vai para o 1º Quadrante

Se não:

ponto vai para o 4º Quadrante

Se não // x ponto < x eixo

y ponto >= y eixo ?

Se sim:

ponto vai para o 2º Quadrante

Se não:

ponto vai para o 3º Quadrante

void numQuadrantes(ApontadorArvore *, int *);

Recebe um ponteiro para árvore e um ponteiro para inteiro. Nesta função é feita a contagem de quadrantes de uma árvore, além de ser feita a desalocação de memória, alocada dinamicamente antes, da lista de pontos de cada registro e da árvore. O ponteiro pra inteiro permite que não se perca o número de quadrantes contados após o retorno da função.

void ImprimeConsultaRetangulo(TipoPonto *, TipoPonto *, int, FILE *);

Recebe um vetor de pontos contendo todos os pontos do espaço lidos, outra matriz contendo os pontos (x1,y1) e (x2,y2) do quadrante a ser verificado pela consulta, o número de pontos, um ponteiro para o arquivo onde serão impressos os resultados da consulta feita. Verifica os pontos que estão dentro do retângulo de consulta e os imprime no arquivo.

void Particao(int, int, int *, int *, TipoPonto *);

void Ordena(int, int, TipoPonto*);

void QuickSort(TipoPonto *, int *);

O algoritmo QuickSort foi escolhido para a implementação do programa por ser o algoritmo de ordenação interna mais rápido que se conhece. Todas as funções acima citadas fazem parte dele. Em todas estas funções são passados

como parâmetro, um vetor de pontos. O algoritmo do *QuickSort()* é recursivo, chamando as próprias funções para a ordenação. A função implementada olha a coordenada “x” do ponto, como critério para a ordenação. Em caso de empate, pontos com coordenada “x” igual, será olhada então a coordenada “y” do ponto para a ordenação. Ficando os pontos ordenados em “x” e os que pontos que tiverem “x” igual, tem “y” de forma ordenada.

void LeArquivo(char *, char *);

Recebe como parâmetros os nomes dos arquivos de entrada e saída do programa. É a função que coordena o programa. Nela são lidas as informações necessárias para a montagem da quadtree. É inicializada a árvore, são colocadas as características do quadrante raiz, é criado um vetor de pontos para ser passado para a função de consulta, além da chamada das funções *InsereArvore()*, definindo a raiz da árvore, *QuickSort()*, para a ordenação dos pontos, *InserePontos()*, que insere na árvore cada ponto lido do arquivo e *ImprimeConsultaRetangulo()*. Imprime, antes da chamada da consulta, o número de quadrantes no arquivo de saída e desaloca a memória alocada para guardar a lista de pontos.

int main(int argc, char ** argv)

Na main é chamada a função *PegaArgumentos()*, que recebe os argumentos passados por linha de comando, e a função *LeArquivo()* com os parâmetros lidos da linha de comando.

Antes do término da execução, é dado *free()* em todas as estruturas de dados que foram alocadas dinamicamente pelo programa.

Análise de complexidade

Espacial

As funções *PegaArgumentos(), Inicializa(), InsereArvore(),* têm sua complexidade **O(1)**, visto que em nenhuma delas existe loop ou chamada recursiva.

O algoritmo de ordenação *QuickSort()* tem complexidade no pior caso $O(n^2)$, porém são casos muito raros e para números aleatórios, a complexidade é, experimentalmente comprovada, próxima de **O(n*(log(n)))**, onde n é o número de pontos que são lidos pelo programa.

A função *ImprimeConsultaRetangulo()* é **O(n)** uma vez que no seu pior caso, o retângulo da consulta é o mesmo retângulo da Quadtree, ou seja, toda a lista de pontos será percorrida e todos os pontos da Quadtree serão consultados e retornados.

A função *numQuadrantes()* no seu pior caso terá os pontos inseridos sequencialmente no mesmo quadrante, percorrendo assim a quadtree atrás quadrantes que não tenham filhos. Então é de ordem **O(d+h),** onde d é o número de quadrantes e h é a altura da Quadtree.

A função *InserePonto(),* que é responsável pela construção da Quadtree, tem complexidade média da ordem $O(\log_4 n)$, que corresponde a uma Quadtree balanceada. Porém considerando o pior caso, a cada quadrante cheio é necessário que os pontos existentes sejam redirecionados para os sub-quadrantes filhos, chamando a função *InserePonto()* recursivamente para cada ponto do quadrante. Mas no pior caso, cada quadrante poderá ter apenas um único ponto. Se considerarmos então pontos suficientemente próximos uns aos outros, teremos de particionar um mesmo quadrante inúmeras vezes com sub-quadrantes vazios, para que as partições, e consequentemente os quadrantes, sejam cada vez menores e consigam então fazer com que estes pontos próximos uns aos outros sejam realocados em quadrantes diferentes. Podemos então deduzir que, se cada partição em quatro quadrantes, corresponde a mais um nível na altura da Quadtree, e para cada ponto inserido teremos de particionar um quadrante mais uma vez, a complexidade no pior caso é da ordem de **O(n*(h))**.

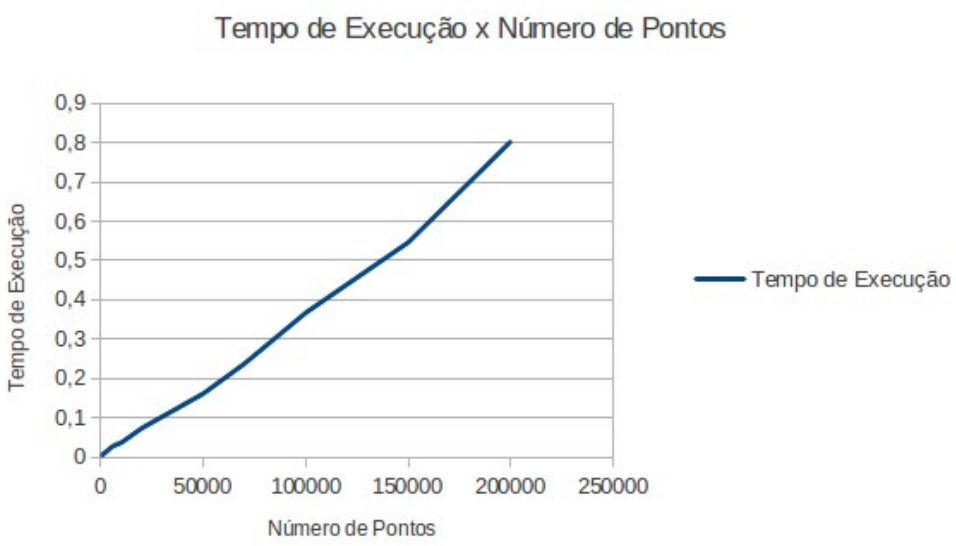
A função *LeArquivo()* tem ordem de complexidade igual à função de maior custo, uma vez que ela faz a chamada das demais funções.

Portanto podemos concluir que a complexidade do programa será então max(O(n*(h)), (n*(log(n)))), mas podemos ver então que h será sempre maior que log(n), então podemos concluir para esse caso que a complexidade do programa é dada por: **O(n*(h)).**

Temporal

Para a complexidade temporal, os experimentos foram feitos para apenas uma única instância, numa área de 100000 x 100000, com 10 consultas e variando-se apenas o número de pontos a serem colocados na Quadtree. Os pontos para análise foram gerados aleatoriamente utilizando a função *GeradorDePontos()*, que segue comentada no código do programa. A tabela do número de pontos / tempo de execução e o gráfico seguem abaixo.

Pontos	Tempo
100	0,003
500	0,004
1000	0,007
2000	0,01
5000	0,024
7000	0,03
10000	0,036
20000	0,073
50000	0,161
70000	0,237
100000	0,367
150000	0,546
200000	0,803



Podemos ver através do gráfico, Tempo de Execução(s) x Número de Pontos, que o tempo de execução aumenta à medida que o número de pontos aumenta. Este aumento é praticamente linear, como podemos ver pelo gráfico e também pela tabela, onde para dez mil pontos, o tempo de execução é 0,36s e quando dobramos o número de pontos, o tempo de execução é praticamente dobrando, sendo 0,73s. Então a função que melhor descreveria o tempo de execução do programa é **T(n) = O(n)**;

Máquina utilizada

A máquina utilizada no desenvolvimento e testes dos algoritmos foi um notebook Dell Vostro 3500, com 4Gb de memória RAM DDR3 e processador Intel Core i5, 2.4GHz. Além disso, o desenvolvimento e testes foram todos feitos em ambientes Linux, utilizando a IDE Code::Blocks 12.11.

Conclusão

Foi um trabalho difícil, porém interessante de ser feito e demandou muitas horas, não só de programação, mas de tentativas de abstração do problema para o ambiente computacional. A utilização de árvores para implementação do programa foi algo novo pra mim que nunca o tinha feito e aprimorou muito os meus conhecimentos nessa parte, assim como também a utilização do Makefile para rodar o programa por linha de comando e a programação em ambiente Linux. Implementei primeiramente o algoritmo de forma estática, para entender como funcionaria a recursividade e após isto implementei o TP utilizando árvores. Uma das maiores dificuldades foi a ausência de testes para conferir se a resposta que o programa dava era certa, mas conferi com testes de outros alunos e grande parte deles deu resultados iguais.