INSTITUTO POLITÉCNICO DE BEJA

kd-trees
Trabalho de Época Normal
Estruturas de Dados e Algoritmos

Pedro Miguel Clemente Dias Moreira

n.º 10015

17 de março 2012

Conteúdo

1 Introdução

Contexto

Este trabalho é uma componente lectiva da disciplina Estruturas de Dados e Algoritmos que abrange o estudo da organização e armazenamento de informação de modo a que esta seja utilizada eficientemente.
Nele é aboradado um tipo de representação hierárquica de dados, nomeadamente uma árvore k-d que tem como principal característica a organização de dados num espaço k-dimensional.

Objectivos

Foi proposta a criação de uma biblioteca para operação com árvores k-d balanceadas que permita, pelo menos:

- a construção da árvore a partir de uma lista de dados
- a junção de elementos
- a remoção de elementos
- a pesquisa do elemento mais próximo

A árvore deve ser balanceada e ser construída sobre uma representação da memória por listas ligadas

Estrutura do relatório

Este relatório apresenta uma introdução sobre o tema abordado no trabalho, o código efectuado com as partes mais relevantes explicadas, os resultados experimentais, qual o protocolo utilizado para os obter e, finalmente, as conclusões retiradas dos mesmos.

2 Teoria

As árvores k-d são estruturas de dados de partição do espaço destinadas a organizar pontos num espaço k-dimensional. As mesmas são utilizadas em várias aplicações, nomeadamente pesquisas envovlendo uma chave multidimensional.
A biblioteca envolvida neste trabalho deve seguir o modelo de representação de memória através de listas ligadas, em que cada nó tem a sua representação na memória através de um ponteiro para uma lista de tamanho pré-definido.
A árvore deve estar balanceada o que, na prática, significa que não deve existir uma grande diferença na profundidade dos nós limítrofes da mesma (folhas).
A ordenação dos dados é feita pela comparação da chave dos nós, alterando o eixo de comparação a cada mudança de nível da árvore.
Quando o eixo a comparar é igual a mesma é feita no eixo seguinte. No caso de a chave ser igual a alguma existente na árvore apenas é actualizado o valor desse nó pelo novo.

3 Parte Experimental

3.1 Realização Experimental

A linguagem de programação escolhida para a elaboração do trabalho foi Python. O código foi desenvolvido com auxílio da ferramenta Sublime instalado num sistema operativo Windows 7 (64-bit). O computador utilizado para o desenvolvimento e aplicação dos testes foi:

- Marca: Insys
- Processador: AMD Athlon TF-20 1.60GHz
- Ram: 3 GB

3.2 Sistema Experimental

Class No

Classe que representa um Nó que será inserido na árvore

Atributos

- key: Chave nó
- valor: Valor que o nó terá
- LC: contador dos nós sucessores localizados à esquerda do mesmo
- RC: contador dos nós sucessores localizados à direita do mesmo
- dim: última dimensão pela qual o nó foi inserido (parte da chave utilizada na ordenação)
- parent: nó antecessor
- left: nó localizado à direita
- right: nó localizado à esquerda

Métodos

- str: Override do método str, para representação personalizada do nó

Class Quicksort

Classe utilizada para a ordenação dos dados, neste caso foi optada a utilização do randomized quicksort que tem sempre como tempo esperado O(n log n).

Atributos

- A: Lista a ser ordenada
- dim: dimensão da chave que será utilizada como base para a ordenação

Métodos

- init: Construtor que recebe uma lista, e a chave de ordenação que por defeito fica definida como o
- sort: método inicial para ordenar a lista, recebe como argumentos uma lista (A), o ponto inicial para se efectuar a ordenação (p), e o ponto final para a ordenação (q), este método dá início à ordenação chamando o método randomizedPartition
- randomizedPartition: para garantir que a lista não esta ordenada de forma decrescente e garantir o tempo de execução $O(nlogn)$ este método escolhe uma chave aleatória e troca-a com a final para que seja feita a ordenação, através do chamamento do método partition
- partition: método que ordena uma subpartição
- randomizedQuicksort: método recursivo que vai dividindo a lista em subpartições e ordenando-o com auxílio dos outros métodos até que fique totalmente ordenada.

Class KDTree

Classe principal onde é criada, armazenada e manipulada a árvore.

Atributos

- root: valor que vai representar a raiz da árvore
- nil: Nó vazio (sentinela), para onde vão apontar os nós que não tenha sucessores ou antecessores
- dimention: número de dimensões da árvore (K)

- stack: Pilha auxiliar utilizada para o rebalanceamento da árvore
- lista: lista auxiliar utilizada para o rebalanceamento da árvore
- free: lista de slots disponíveis para o armazenamento de nós
- pointers: lista de slots ocupadas por nós

Métodos

- malloc: Método utilizado para a alocação de um nó na árvore, começa por verificar a existência de slots livres e, caso exista, ocupa uma slot com o respectivo nó
- freeNo: Libertar uma slot removendo o apontador para o nó e marcando a slot como disponível para um novo objecto. Após esta operação da início ao processo da sua remoção da árvore posterior balanceamento.

  1item insert: Método para a inserção de um nó na árvore. Apesar o nó para onde será iniciada a inserção do mesmo, o que pouparia algum tempo de inserção $logn$ o mesmo não é utilizado neste trabalho. Após a conlusão da inserção o método dá início ao processo confirmação do balanceamento da árvore. Esta opção pode ser desactivada chamado o método com o atributo balance marcado a False Cada nó por onde o que vai ser inserido passar será actualizado o seu contador LC ou RC de acordo com a direcção por este tomada. Assim mantém-se actualizada a contagem dos nós sucessores de cada um.
- clear: Conta os nós sucessores somando os atributos LC e RC e vai retirando o valor aos nós antecessores do mesmo. No final coloca todos os nós sucessores na pilha auxiliar para posterior reinserção na árvore.
- inorderWalk: Percorre toda a árvore e coloca os nós de forma ordenada na lista fornecida
- reInsert: Método utilizado para a reinserção de nós colocados na pilha auxiliar, seja por motivos de balanceamento ou por alguma operação de remoção. A forma de operar é retirar a primeira lista da pilha, encontrar o elemento do meio, dividir a lista nesse local e voltar a inserir na pilha as duas partes resultantes de tal divisão, então insere o nó extraídos do meio de novo na árvore, sem conferir o balanceamento e volta a chamar-se enquanto houver elementos na pilha. Só então dá início à verificação do balanceamento.
- nearestNeighbour: Percorre todos os nós da árvore e calcula a distância euclidiana

$$\sqrt{\sum_{i=0}^{k}(a.key[k] - b.key[k])^2} \tag{3.1}$$

(raiz quadrada do somatório dos quadrados das diferenças entre os pontos das várias dimensões das chaves dos nós) para o nó fornecido, guardando sempre a distância mais curta, no final retorna o nó que teve o resultado mais baixo.

- isBalanced: Verificação se o número de nós sucessores para cada lado, está de acordo com os limites para o balanceamento. Assumindo que para estar balanceada as folhas não poderão ter mais do que um nível de profundidade entre si e sabendo que cada nível difere o número de elementos por potências de dois dignifica que o lado com menor número de elementos não poderá ter menos que o valor da potência do nível inferior ao do lado com maior número de elementos:

$$min(a,b) < 2 * *math.pow(int(math.floor(\log(max(a,b))))) - 1 \tag{3.2}$$

- checkBalance: Percorre todos os elementos da árvore em largura e verifica se os nós estão com valores válidos para uma árvore balanceada, com o auxílio do método isBalanced, caso não esteja balanceado, esse nó é retirado da árvore e os seus sucessores, a lista resultante é ordenada pela dimensão do nó com o problema e a lista é colocada na pilha auxiliar para serem reinseridos na árvore.
- delete: eliminar um nó colocando os seus sucessores na pilha auxiliar, chamando então o método para a reinserção dos nós e, por fim, confirmação do balanceamento

3.3 Resultados Experimentais

Protocolo Experimental

A forma utilizada para testar os dados consiste na criação de listas de nós em quantidades de 1 até 1000 com incrementos de 100. Para cada lista criada o método é testado 40 vezes e são somados os tempos das operações. No final o resultado é dividido pelo número de testes efectuados (40) e guardado o valor para posterior desenho do gráfico.

Apresentação e Discussão dos Resultados

Inserção de Elementos

Como é mostrado abaixo a inserção de elementos é aproximadamente $O(n * log(n^2))$ Este resultado acontece devido ao balanceamento da árvore em que pode ser retirada uma parte da árvore para ser reinserida, o que pode tornar a operação muti lenta, ao contrário de inserção sem balanceamento que levaria a $O(log(n))$.

Pesquisa do elemento mais próximo

A pesquisa do elemento mais próximo tem uma complexidade $O(n)$ pois são sempre percorridos todos os elementos da lista para se encontrar o elemento mais próximo

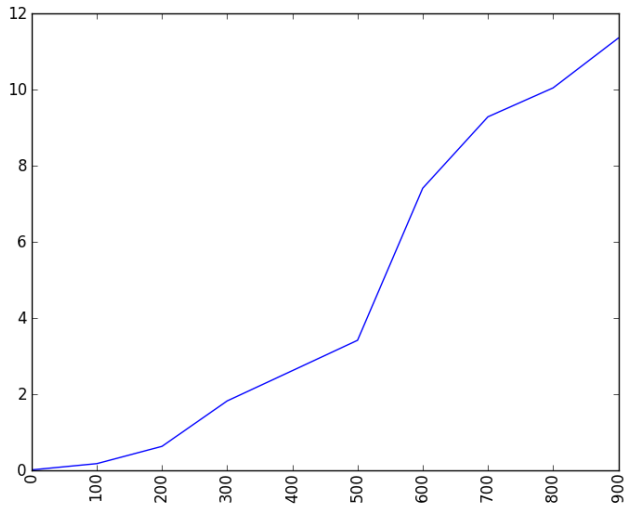

Figura 3.1: Inserção de dados.

N	tempo de execução (seg)
0	0.0
100	0.163800001144
200	0.61779999733
300	1.81072499752
400	2.60599999428
500	3.40314999819
600	7.39674999714
700	9.27110001445
800	10.030825007
900	11.3387999892

Tabela 3.1: inserção de dados.

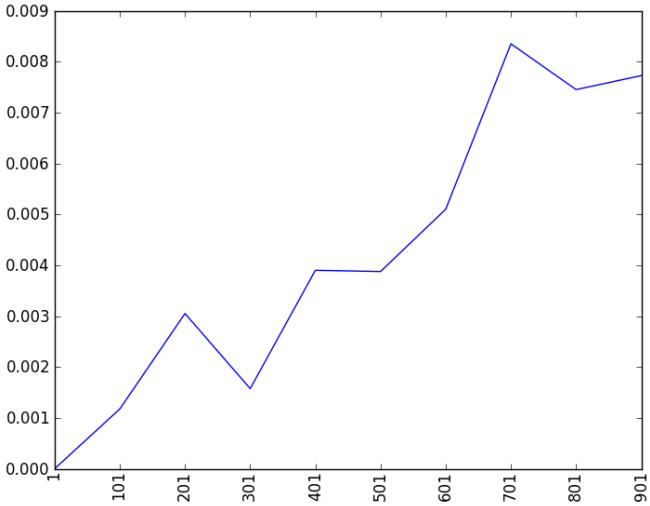

Figura 3.2: Pesquisa do elemento mais próximo.

N	tempo de execução (seg)
1	0.0
101	0.00117500424385
201	0.00304999947548
301	0.00157500505447
401	0.00389999151 23
501	0.00387500524521
601	0.00510001182556
701	0.00835000872612
801	0.00745000243187
901	0.00772499442101

Tabela 3.2: Pesquisa do Elemento mais próximo.

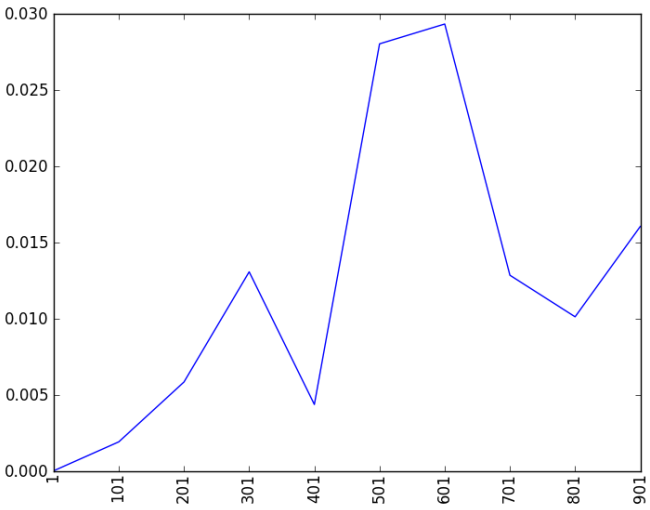

Figura 3.3: Remocao de um elemento.

N	tempo de execução (seg)
1	0.0
101	0.00190000534058
201	0.00582498311996
301	0.0130500078201
401	0.00434999465942
501	0.027999997139
601	0.0292999744415
701	0.0128249943256
801	0.0101000010967
901	0.0160250008106

Tabela 3.3: Remoção de um elemento.

Remoção de um elemento

A complexidade para a remoção de um elemento, como se pode verificar no gráfico (ignorando os valores mais distantes) é de $O(n * log(n))$.

4 Conclusão

O trabalho deveria ter sido elaborado com base no tipo Adaptive Kd-Tree. Pois julgo que seria mais simples efectuar as operações pretendidas.Fica essa importante alteração para trabalho futuro.

A parte da pesquisa do elemento mais próximo também pode ser bastante melhorada baseando-se na pesquisa em profundidade e em largura, tirando partido da alteração para uma Adaptive Kd-Tree.
Nesta parte assumi que a pesquisa é feita pela distância do ponto no espaço a partir das suas coordenadas e não a distância dentro da árvore.

Os gráficos também precisam ser melhorados, efectuando testes com valores mais elevados e obtendo a linha do gráfico uniforme, utilizando por exemplo o método dos mínimos quadrados.

Outra parte a melhorar neste trabalho é o relatório, que apesar de ter sido feito em latex, não aproveita grande parte das suas potencialidades, como por exemplo a bibliografia.

5 Códigos

Classe No

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
'''
autor: Pedro Moreira, 10015
data: 17 de Junho de 2012

'''

class No(object):
    '''
    Classe dos nos de cada elemento
    '''
    def __init__(self, key, valor):
        """
        Criar um no para utilizacao em RB Trees
        @param key, chave do no
        @param valor, valor que o no tem guardado
        """
        self.key = key
        self.valor = valor
        self.LC = 0
        self.RC = 0
        self.dim = 0

        # Ao ser criado o no fica com os apontadores para si proprio (NIL)
        self.parent = self.left = self.right = None
        pass

    def __str__(self):
        """
        Override do metodo str
        Impressao personalizada do no
        """
        s = str(self.key) + " : "#+ str(self.valor)
        s += '('
        s += str(self.parent.key) + ', '
        s += str(self.left.key) + ', '
        s += str(self.right.key)
        s += ', ' + str(self.size) + ')'
        s += ', LC = ' + str(self.LC)
```

```
41          s += ', RC = ' + str(self.RC)
            s += ', dim = ' + str(self.dim)
43          return s
```

No.py

Class QuickSort

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
'''
autor: Pedro Moreira, 10015
data: 31 de maio de 2012

algoritmo quicksort
'''
import random
class Quicksort():
    '''
    o que a classe faz...
    '''

    def __init__(self, A = [], dim = 0):
        '''
        construtor, pode inicializar
        uma lista de dados e ordena-la
        '''
        self.A = A
        self.dim = dim
        if len(A) > 1:
            self.sort(self.A, 0, len(self.A) - 1)

    ################################################
    # ORDENAR A LISTA

    def sort(self, A, p, q):
        '''
        ordenar lista
        '''
        #utilizando quicksort
        self.__randomizedQuicksort__(A, p, q)
        pass

    ################################################
    # livro pag 177
    # RANDOMIZED PARTITION

    def __randomizedPartition__(self, A, p, r):
        i = random.randint(p,r)
        A[r], A[i] = A[i], A[r]
        return self.__partition__(A, p, r)
    # FIM DO RANDOMIZED PARTITION
    ################################################


    ################################################
    # livro pag 169
    # PARTITION
```

```
51
53      def __partition__(self, A, p, r):
            x = A[r].key[self.dim:]
55          i = p - 1
            for j in range(p, r):
57              if A[j].key[self.dim:] <= x:
                    i += 1
59                  A[i], A[j] = A[j], A[i]
                    pass
61              pass
            A[i+1], A[r] = A[r], A[i+1]
63          return i+1
        # FIM DO PARTITION
        ################################################
65
67      ################################################
        # livro pag 177
69      # RANDOMIZED QUICKSORT

71      def __randomizedQuicksort__(self, A, p, r):
            if p < r:
73              q = self.__randomizedPartition__(A, p, r)
                #q = self.__partition__(A, p, r)
75              self.__randomizedQuicksort__(A, p, q - 1)
                self.__randomizedQuicksort__(A, q + 1, r)
77              pass
            pass
79      # FIM DO RANDOMIZED QUICKSORT
        ################################################
```

Quicksort.py

Class KDTree

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
'''
autor: Pedro Moreira, 10015
data: 16 de Junho de 2012

kd-tree
operacoes:
- criar arvore a partir de uma lista
- inserir
- apagar
- pesquisa do no mais proximo
'''
from Quicksort import *
from No import *
import math
import sys

class KDTree(object):
  '''
  Estrutura de dados kdtree
  '''
  def __init__(self, size, dimention):
    '''
    Construtor
    @param dimention dimensao da chave para ordenacao dos dados
    @param size tamanho da arvore
    '''
    #sys.setrecursionlimit(10000)
    self.root = self.nil = No(None, None)
    self.dimention = dimention
    self.stack = []
    self.lista = []
    self.free = [i for i in range(size)]
    self.pointers = [None for i in range(size)]

  def malloc(self, no):
    '''
    Tentar a alocacao de um No na arvore
    @param no Objecto a ser inserido na arvore
    @return -1 em caso de erro ou o no ja inserido na arvore
    '''
    if len(self.free) > 0:
      x = self.free.pop()
      self.pointers[x] = no.key
      no.pointer = x
      self.insert(self.root, no)
      return no
    else:
      print "out of space"
```

```
      return -1

  def freeNo(self, x):
    '''
    Eliminar no da arvore
    @param x no a Eliminar
    @return -1 caso o no nao exista ou o se a operacao correr com sucesso
    '''
    if self.pointers[x.pointer] != None:
      self.pointers[x.pointer] == None
      self.free.append(x.pointer)
      self.delete(x)
      return 0
    else:
      return -1

  def insert(self, a, z, balance = True):
    '''
    Inserir no na arvore
    @param a local a inserir o no
    @param z no a inserir
    @param balance informacao para balancear ou nao a arvore (true por
        defeito)
    '''
    z.parent = self.nil
    z.left = self.nil
    z.right = self.nil
    z.LC = z.RC = 0

    y = self.nil
    x = a
    dim = -1

    while x != self.nil:
      dim = (dim + 1) % self.dimention
      if x.key == z.key:
        x.valor = z.valor
        break

      z.dim = dim
      y = x
      if z.key[dim:] < x.key[dim:]:
        x.LC += 1
        x = x.left
      else:
        x.RC += 1
        x = x.right
    z.parent = y
    if y == self.nil:
      self.root = z
    elif z.key[dim:] < y.key[dim:]:
      y.left = z
```

```
102       else:
            y.right = z
104
          if balance:
106         self.checkBalance()
108     def __clear(self, x):
          '''
110       Contar os nos sucessores ao no pretendido
          e percorrer todos os seus antecessores retirando a respectiva contagem
112       @param x no a partir do qual sera para limpar as contagens
          '''
114       if x == self.root:
            self.root = self.nil
116       else:
            z = x
118         i = x.RC + x.LC + 1
            while(z != self.nil):
120           if z.parent.left == z:
                z.parent.LC -= i
122
              else:
124             z.parent.RC -= i
              z = z.parent
126
            if x.parent.left == x:
128           x.parent.left = self.nil
            else:
130           x.parent.right = self.nil
132     def inorderWalk(self, x, lista):
          '''
134       Percorrer a arvore devolvendo uma lista ordenada com os nos
          @param x no a partir do qual se constroi a lista
136       @param lista para guardar os dados
          '''
138       if x != self.nil:
            self.inorderWalk(x.left, lista)
140         lista.append( x )
            self.inorderWalk(x.right, lista)
142
144     def __reInsert(self):
          '''
146       Metodo para voltar a inserir na arvore nos
          que tenham sido retirados por motidos de balanceamento
148       '''
          if len(self.stack) > 0:
150         a = self.stack.pop(0)
            if len(a) == 1:
152           self.insert(self.root, a.pop(0), False)
            elif len(a) == 0:
```

```
154         pass
      else:
156         k = int(math.floor(len(a)/2))
            x = a.pop(k)
158         x.parent = x.left = x.right = None
            self.insert(self.root, x, False)
160         b = a[:k]
            if len(b) > 0:
162           self.stack.append(b)
            b = a[k:]
164         if len(b) > 0:
              self.stack.append(b)
166       if len(self.stack) > 0:
            self.__reInsert()
168     else:
        self.checkBalance()
170
  def nearestNeighbour(self, no):
172   '''
      Percorre todos os elementos da arvore e calcula a distancia eucladiana
174   devolve o no com menor distancia
      @param no No que pretendemos fazer a pesquisa
176   @return No mais proximo do fornecido
      '''
178   lista = []
      self.inorderWalk(self.root, lista)
180   distancia = sys.maxint
      x = self.nil
182   for i in lista:
        if no != i:
184       temp = 0
          for k in range(self.dimention):
186         temp += (i.key[k] - no.key[k])**2
          temp = math.sqrt(temp)
188       if distancia > temp:
            distancia = temp
190         x = i
      return x
192
  def isBalanced(self, x):
194   '''
      verificar se o numero de sucessores para cada lado
196   do no em causa corresponde a dois ramos com a mesma altura
      @param x no a analisar
198   '''
      a = x.LC
200   b = x.RC
      if (a+b) < 2 : return True
202   if (a+b) <= 4 and min(a,b) == 1: return True
      return min(a, b) > 2**(int(math.floor(math.log(max(a,b),2))))-1
204
```

```
206   def checkBalance(self):
        '''
208     percorrer todos os nos para
        verificar se a arvore esta balanceada
210     caso existam problemas, os nos sao retirados
        e ordenados para serem reinseridos na arvore
212     '''
        if self.root == self.nil:
214       return
        stack = []
216     stack.append(self.root)
        while len(stack) > 0:
218       x = stack.pop(0)
          if x.left   != self.nil : stack.append(x.left)
220       if x.right  != self.nil : stack.append(x.right)

222       if self.isBalanced(x): continue
          parent = x.parent
224       dim = x.dim
          self.__clear(x)
226       lista = []
          self.inorderWalk(x, lista)
228       Quicksort(lista, (dim + 1) % self.dimention)
          self.stack.append(lista)
230       self.__reInsert()
        pass
232
      def delete(self, z):
234     '''
        Eliminar um no e colocar todos os seus sucessores
236     na lista para serem reinseridos na arvore
        @param z no a ser eliminado
238     '''
        if z.parent != self.nil:
240       if z.parent.left == z.key:
            z.parent.left = self.nil
242         z.parent.LC = 0
          else:
244         z.parent.right = self.nil
            z.parent.RC = 0
246     lista = []
        self.inorderWalk(z.left, lista)
248     self.inorderWalk(z.right, lista)
        self.stack.append(lista)
250
        z.parent = z.left = z.right = self.nil
252     z.LC = z.RC = 0
        self.__reInsert()
254     self.checkBalance()
        pass
```

KDTree.py

6 Bibliografia

1 Cormen Thomas H., et all, Introduction to Algorithms, The MIT Press, 3 edition, 2009

2 k-d tree, wikipedia, available at http://en.wikipedia.org/wiki/K-d_tree, visited 17/06/2012

3 Leonardo Rodriguez Heredia, Cirano Iochpe e João Comba, Explorando a Multidimensionalidade da Kd-Tree para Suporte a Temporalidade em Dados Espaciais Vetoriais do Tipo Ponto, available at http://www.inf.ufrgs.br/~comba/papers/2003/geo-info.pdf, visited 17/06/2012