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

Índice Geral I

1 Introdução **3**

2 Teoria **5**

3 Parte Experimental **7**
3.1 Realização Experimental 7
3.2 Sistema Experimental 7
3.3 Resultados Experimentais 10

4 Conclusão **15**

5 Códigos **17**

6 Bibliografia **25**

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).

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

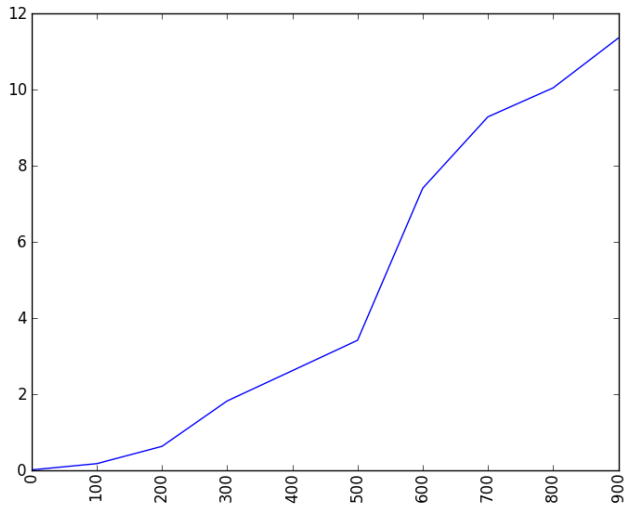

Figura 3.1: Inserção de dados.

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.

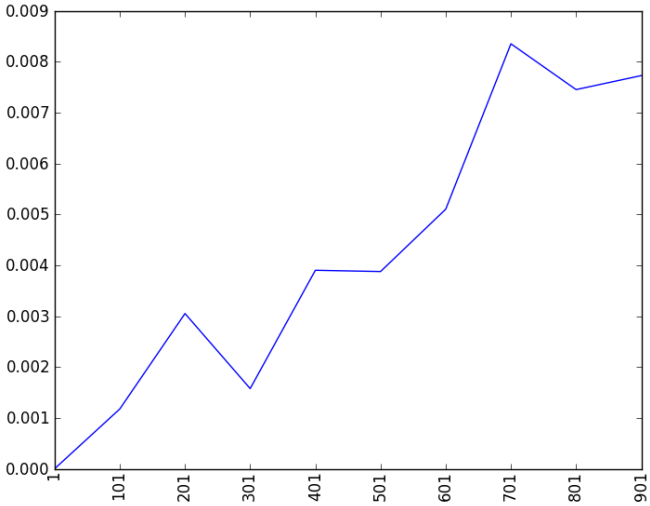

Figura 3.2: Pesquisa do elemento mais próximo.

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.

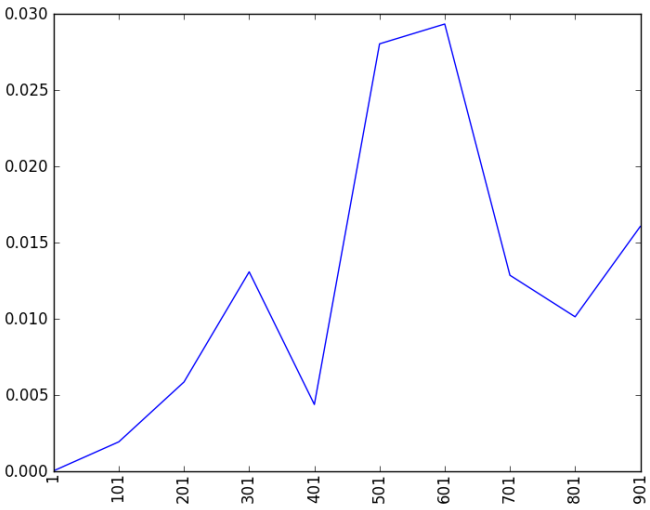

Figura 3.3: Remocao de um elemento.

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. 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

```
#!/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)
        s += ', RC = ' + str(self.RC)
        s += ', dim = ' + str(self.dim)
        return s
```

No.py

```
#!/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

    def __partition__(self, A, p, r):
        x = A[r].key[self.dim:]
        i = p - 1
        for j in range(p, r):
            if A[j].key[self.dim:] <= x:
                i += 1
                A[i], A[j] = A[j], A[i]
                pass
            pass
        A[i+1], A[r] = A[r], A[i+1]
        return i+1
    # FIM DO PARTITION
    ################################################


    ################################################
    # livro pag 177
    # RANDOMIZED QUICKSORT

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

Quicksort.py

```
#!/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
```

```
18
19  class KDTree(object):
20    '''
21    Estrutura de dados kdtree
22    '''
23    def __init__(self, size, dimention):
24      '''
25      Construtor
26      @param dimention dimensao da chave para ordenacao dos dados
27      @param size tamanho da arvore
28      '''
29      #sys.setrecursionlimit(10000)
30      self.root = self.nil = No(None, None)
31      self.dimention = dimention
32      self.stack = []
33      self.lista = []
34      self.free = [i for i in range(size)]
35      self.pointers = [None for i in range(size)]
36
37    def malloc(self, no):
38      '''
39      Tentar a alocacao de um No na arvore
40      @param no Objecto a ser inserido na arvore
41      @return -1 em caso de erro ou o no ja inserido na arvore
42      '''
43      if len(self.free) > 0:
44        x = self.free.pop()
45        self.pointers[x] = no.key
46        no.pointer = x
47        self.insert(self.root, no)
48        return no
49      else:
50        print "out of space"
51        return -1
52
53    def freeNo(self, x):
54      '''
55      Eliminar no da arvore
56      @param x no a Eliminar
57      @return -1 caso o no nao exista ou o se a operacao correr com sucesso
58      '''
59      if self.pointers[x.pointer] != None:
60        self.pointers[x.pointer] == None
61        self.free.append(x.pointer)
62        self.delete(x)
63        return 0
64      else:
65        return -1
66
67    def insert(self, a, z, balance = True):
68      '''
69      Inserir no na arvore
```

```
70      @param a local a inserir o no
        @param z no a inserir
72      @param balance informacao para balancear ou nao a arvore (true por
            defeito)
        '''
74      z.parent = self.nil
        z.left = self.nil
76      z.right = self.nil
        z.LC = z.RC = 0
78
        y = self.nil
80      x = a
        dim = -1
82
        while x != self.nil:
84        dim = (dim + 1) % self.dimention
          if x.key == z.key:
86          x.valor = z.valor
            break
88
          z.dim = dim
90        y = x
          if z.key[dim:] < x.key[dim:]:
92          x.LC += 1
            x = x.left
94        else:
            x.RC += 1
96          x = x.right
        z.parent = y
98      if y == self.nil:
          self.root = z
100     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

          else:
            z.parent.RC -= i
          z = z.parent

        if x.parent.left == x:
          x.parent.left = self.nil
        else:
          x.parent.right = self.nil

    def inorderWalk(self, x, lista):
      '''
      Percorrer a arvore devolvendo uma lista ordenada com os nos
      @param x no a partir do qual se constroi a lista
      @param lista para guardar os dados
      '''
      if x != self.nil:
        self.inorderWalk(x.left, lista)
        lista.append( x )
        self.inorderWalk(x.right, lista)


    def __reInsert(self):
      '''
      Metodo para voltar a inserir na arvore nos
      que tenham sido retirados por motidos de balanceamento
      '''
      if len(self.stack) > 0:
        a = self.stack.pop(0)
        if len(a) == 1:
          self.insert(self.root, a.pop(0), False)
        elif len(a) == 0:
          pass
        else:
          k = int(math.floor(len(a)/2))
          x = a.pop(k)
          x.parent = x.left = x.right = None
          self.insert(self.root, x, False)
          b = a[:k]
          if len(b) > 0:
            self.stack.append(b)
          b = a[k:]
          if len(b) > 0:
            self.stack.append(b)
        if len(self.stack) > 0:
          self.__reInsert()
      else:
        self.checkBalance()

    def nearestNeighbour(self, no):
      '''
```

```
    Percorre todos os elementos da arvore e calcula a distancia eucladiana
    devolve o no com menor distancia
    @param no No que pretendemos fazer a pesquisa
    @return No mais proximo do fornecido
    '''
    lista = []
    self.inorderWalk(self.root, lista)
    distancia = sys.maxint
    x = self.nil
    for i in lista:
      if no != i:
        temp = 0
        for k in range(self.dimention):
          temp += (i.key[k] - no.key[k])**2
        temp = math.sqrt(temp)
        if distancia > temp:
          distancia = temp
          x = i
    return x

  def isBalanced(self, x):
    '''
    verificar se o numero de sucessores para cada lado
    do no em causa corresponde a dois ramos com a mesma altura
    @param x no a analisar
    '''
    a = x.LC
    b = x.RC
    if (a+b) < 2 : return True
    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


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

      if self.isBalanced(x): continue
      parent = x.parent
      dim = x.dim
```

```
      self.__clear(x)
      lista = []
      self.inorderWalk(x, lista)
      Quicksort(lista, (dim + 1) % self.dimention)
      self.stack.append(lista)
      self.__reInsert()
    pass

  def delete(self, z):
    '''
    Eliminar um no e colocar todos os seus sucessores
    na lista para serem reinseridos na arvore
    @param z no a ser eliminado
    '''
    if z.parent != self.nil:
      if z.parent.left == z.key:
        z.parent.left = self.nil
        z.parent.LC = 0
      else:
        z.parent.right = self.nil
        z.parent.RC = 0
    lista = []
    self.inorderWalk(z.left, lista)
    self.inorderWalk(z.right, lista)
    self.stack.append(lista)

    z.parent = z.left = z.right = self.nil
    z.LC = z.RC = 0
    self.__reInsert()
    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