RaqApproach Reference Manual

0.1

Generated by Doxygen 1.4.7

Wed Jul 26 20:22:54 2006

Contents

Chapter 1

RaqApproach Namespace Index

1.1 RaqApproach Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 2

RaqApproach Class Index

2.1 RaqApproach Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

RaqApproach Namespace Documentation

3.1 std Namespace Reference

The RAq Approach: main.cpp.

3.1.1 Detailed Description

The RAq Approach: main.cpp.

Generates the distance matrix necessary for the creation of tree. Currently, "uncorrected distances" and the "score-based distances" options are the only ones that work. As a caveat, be warned that these two distance methods will produce different trees.

Chapter 4

RaqApproach Class Documentation

4.1 element Class Reference

this class stores the distance and corresponding taxa

`#include <elements.h>`

Public Member Functions

- **element (string t, double d)**
  copy constructor
- **element ()**
  default constructor
- **string get_taxa ()**
  get_taxa function
- **double get_dist ()**
  get distance function

Private Attributes

- double **dist**
  distance between two taxa
- **string taxa**
  actual two taxa

4.1.1 Detailed Description

this class stores the distance and corresponding taxa

this class assists with the clustering process by storing a set of taxa and their associated distance

4.1.2 Constructor & Destructor Documentation

4.1.2.1 element::element (string *t*, double *d*)

copy constructor

takes in a string and a double and creates an element object out of them

Parameters:

t holds taxa to be copied into element object

d holds distance to be copied into element object

See also:

element() (p. 8)

< copy constructor

4.1.2.2 element::element ()

default constructor

creates an element object with null values

See also:

element(string t, double d) (p. 8)

< default constructor

4.1.3 Member Function Documentation

4.1.3.1 double element::get_dist () [inline]

get distance function

Returns:

distance associated with an element

4.1.3.2 string element::get_taxa () [inline]

get_taxa function

Returns:

the taxa associated with the element

The documentation for this class was generated from the following file:

- elements.h

4.2 helper Class Reference

helper class that assists with hashing procedure

`#include <elements.h>`

Public Member Functions

- **helper (vector< short > i, vector< int > t)**
  copy constructor
- **helper ()**
  default constructor
- **vector< short > get_id ()**
  get id function
- **vector< int > get_tree ()**
  get tree function
- **void display ()**
  display function

Private Attributes

- vector< short > **id**
  holds id of taxa or set of taxa
- **vector< int > taxa_list**
  holds taxa or set of taxa

4.2.1 Detailed Description

helper class that assists with hashing procedure

this class works closely with the merge section of code that merges common taxa together

4.2.2 Constructor & Destructor Documentation

4.2.2.1 helper::helper (vector< short > *i*, vector< int > *t*)

copy constructor

takes in a vector<short> and a vector<int> to create a helper object

Parameters:

i holds id of taxa or set of taxa
t holds taxa or set of taxa

< copy constructor

4.2.3 Member Function Documentation

4.2.3.1 void helper::display ()

display function

displays both the id and taxa list associated with that id

< displays id and associated taxa

4.2.3.2 vector<int> helper::get_tree () [inline]

get tree function

Returns:

list of taxa associated with helper object

The documentation for this class was generated from the following file:

- elements.h

4.3 Node Struct Reference

`#include <tree.h>`

Public Attributes

- elements **data**
- **Node** ∗ **left**
- **Node** ∗ **right**
- **Node** ∗ **parent**
- **nid id**

4.3.1 Detailed Description

Node (p. 11) is the data structure primarily used for the decomposition step

Parameters:

∗***left*** pointer to the left child

∗***right*** pointer to the right child

∗***parent*** pointer to the parent of current node

id identifier vector of the node

The documentation for this struct was generated from the following file:

- tree.h

4.4 SuperNode Struct Reference

`#include <tree.h>`

Public Attributes

- int **taxa**
  is 0 if there is no taxa: only occurs when it is an internal node
- **SuperNode** * **left**
  would be NULL if is a leaf node
- **SuperNode** * **right**
  would be NULL if is a leaf node
- **SuperNode** * **parent**
  is never NULL... except when it is the root
- **bool internal**
  indicates whether or not is an internal node
- **nid id**
  identification of the node

4.4.1 Detailed Description

SuperNode (p. 12) is the data structure primarily used for the super tree construction step

Parameters:

taxa holds taxa. is 0 when node is internal
*****left*** pointer to the left child
*****right*** pointer to the right child
*****parent*** pointer to the parent
internal indicates whether or not is internal node
id identification of the node (-1 when it is not the root node)

The documentation for this struct was generated from the following file:

- tree.h

4.5 SuperTree Class Reference

`#include <tree.h>`

Public Member Functions

- **SuperTree ()**
  create empty tree
- **SuperTree (SuperNode ∗, int)**
  create new tree with passed node as the new main root.
- **∼SuperTree ()**
  destructor
- **void insert (int, int, nid, bool)**
  insert new node as child of current.
- **void insert (SuperTree &, SuperTree &)**
  overloaded insert fuction
- **void remove (SuperNode ∗)**
  delete a node and its subtree
- **int value () const**
  value function
- **nid Id () const**
  Id function.
- **void left ()**
  navigate the tree go to left child
- **void right ()**
  go to right child
- **void parent ()**
  go to parent
- **void reset ()**
- **void SetCurrent (SuperNode ∗)**
  set the location of the current pointer
- **SuperTree (const SuperTree &)**
  copy constructor
- **SuperNode ∗ pointer_left () const**
  return subtree (node) pointers returns pointer to left child

- SuperNode * pointer_right () const
  returns pointer to right child
- SuperNode * pointer_parent () const
  returns pointer to parent
- SuperNode * pointer_current () const
  returns current pointer
- SuperNode * root () const
  returns pointer to main_root
- int peek_left () const
  return values of children and parent without leaving current node return taxa of left child
- int peek_right () const
  return taxa of right child
- int peek_parent () const
  return taxa of parent
- void DisplayInorder (SuperNode *) const
  print the tree or a subtree. do an "in-order" traversal
- void DisplayPreorder (SuperNode *) const
  do a "pre-order" traversal
- void DisplayPostorder (SuperNode *) const
- void Newick (list< string > &, SuperNode *)
  displays newick format of tree
- void clear ()
  delete all nodes in the tree
- bool IsEmpty () const
  checks to see if a tree is empty
- bool IsFull () const
  checks to see if the tree is full
- bool IsInternal () const
  checks to see the current node is an internal node

Private Member Functions

- SuperNode * CopyTree (SuperNode *, SuperNode *) const
  create a new copy of a subtree if passed to the constructor

- **SuperNode * CopyTree (SuperNode *, SuperNode *, int) const**
  CopyTree created for personal devices.

Private Attributes

- **SuperNode * current**
  pointer to current node
- **SuperNode * main_root**
  pointer to root node
- **bool subtree**
  does it reference a part of a larger object?

4.5.1 Detailed Description

Basic data structure that creates the subtrees and final supertree in the final steps of the algorithm

4.5.2 Constructor & Destructor Documentation

4.5.2.1 SuperTree::SuperTree ()

create empty tree

creates empty tree with default root node which has no value. set current to main root node.

4.5.2.2 SuperTree::SuperTree (SuperNode *, int)

create new tree with passed node as the new main root.

set current to main root.

Parameters:

SuperNode* node to set as root of tree

int indicates where object should point to (0: node of original tree 1: new copy of the subtree

See also:

SuperTree(SuperNode*, int) (p. 15), **SuperTree(const SuperTree &)** (p. 16), **~SuperTree()** (p. 15)

4.5.2.3 SuperTree::~SuperTree ()

destructor

See also:

SuperTree() (p. 15), **SuperTree(SuperNode*, int)** (p. 15), **SuperTree(const SuperTree &)** (p. 16)

< delete all nodes

4.5.2.4 SuperTree::SuperTree (const SuperTree &)

copy constructor

allows **SuperTree** (p. 13) to do a "deep" copy

Parameters:

SuperTree& **SuperTree** (p. 13) object to be copied

4.5.3 Member Function Documentation

4.5.3.1 void SuperTree::clear ()

delete all nodes in the tree

< use the remove function on the main root

< since there are no more items, set main_root to NULL

4.5.3.2 SuperNode * SuperTree::CopyTree (SuperNode * *root*, SuperNode * *parent*, int *dummy*) const [private]

CopyTree created for personal devices.

Does same thing as the one above, accept is created for own devices sets the ids of all the internal nodes to -1

Parameters:

root pointer to root of **SuperTree** (p. 13)

parent pointer to parent of **SuperNode** (p. 12)

dummy just a dummy value to indicate a new function (boo! bad coding practice!)

Returns:

pointer location of new root node

< base case - if the node doesn't exist, return NULL.

< make a new location in memory

< make a copy of the node's data

< set the new node's parent

< copy the left subtree of the current node. pass the current node as the subtree's parent

< do the same with the right subtree

< what makes this function different from the other one!

< return a pointer to the newly created node.

4.5.3.3 SuperNode * SuperTree::CopyTree (SuperNode * *root*, SuperNode * *parent*) const [private]

create a new copy of a subtree if passed to the constructor

The second parameter is a pointer to the parent of the subtree being passed. Since parent of the main root is always NULL, we pass NULL as the second parameter in the class constructor

Parameters:

root pointer to root of **SuperTree** (p. 13)

parent pointer to parent of **SuperNode** (p. 12)

Returns:

pointer location of new root node

< base case - if the node doesn't exist, return NULL.

< make a new location in memory

< make a copy of the node's data

< set the new node's parent

< copy the left subtree of the current node. pass the current node as the subtree's parent

< do the same with the right subtree

< return a pointer to the newly created node.

4.5.3.4 void SuperTree::DisplayPostorder (SuperNode *) const

do a "post-order" traversal

4.5.3.5 nid SuperTree::Id () const

Id function.

Returns:

id at current location (vector<short>)

4.5.3.6 void SuperTree::insert (SuperTree &, SuperTree &)

overloaded insert fuction

inserts a right and left supertree as subtrees

Parameters:

SuperTree (p. ***13***) & left tree to be inserted

SuperTree (p. ***13***) & right tree to be inserted

See also:

insert(int, int, nid, bool) (p. 18)

< if the tree has no nodes, make a root node, disregard pos.

< node created, exit the function

Parameters:

right insert left and right subtrees

4.5.3.7 void SuperTree::insert (int, int, nid, bool)

insert new node as child of current.

insert new taxa into supertree:

Parameters:

int taxa (-1 or 0 if it is an internal node)

int location to insert (0=left 1=right 2 = parent)

nid id of **Node** (p. 11) to be inserted (-1 if is an internal node)

bool indicates whether or not the node is internal

See also:

insert(SuperTree &, SuperTree &) (p. 17)

if the tree has no nodes, make a root node, disregard pos.

< node created, exit the function

< if new node is a left child of current

< if child already exists, replace value

< if it is a left child, copy the id of the parent

< else if new node is a right child of current

< if child already exists, replace value

< if it is a right child, copy the id of the parent

4.5.3.8 bool SuperTree::IsEmpty () const

checks to see if a tree is empty

< If there aren't any nodes in the tree, main_root points to NULL

4.5.3.9 void SuperTree::Newick (list< string > &, SuperNode *)

displays newick format of tree

recrusively outputs the tree in Newick format

Parameters:

list<string>& contains tree so far in Newick format

SuperNode* pointer to root of tree we want to output in Newick format

4.5.3.10 int SuperTree::peek_left () const

return values of children and parent without leaving current node return taxa of left child

advantage: we don't have to leave the node! (self-explanatory)

4.5.3.11 void SuperTree::remove (SuperNode *)

delete a node and its subtree

recursively deletes the node pointed to by **SuperNode** (p. 12) and all the nodes in its subtree

Parameters:

SuperNode* indicates root of tree to be deleted

See also:

clear() (p. 16)

< base case - if the root doesn't exist, do nothing

< perform the remove operation on the nodes left subtree first

< perform the remove operation on the nodes right subtree first

< if the main root is being deleted, main_root must be set to NULL

< make sure the parent of the subtree's root points to NULL, since the node no longer exists

< set current to the parent of the subtree removed.

4.5.3.12 void SuperTree::reset ()

go to main_root

4.5.3.13 void SuperTree::SetCurrent (SuperNode *)

set the location of the current pointer

Parameters:

SuperNode* location that current pointer should be set to

4.5.3.14 int SuperTree::value () const

value function

Returns:

taxa at current location

The documentation for this class was generated from the following file:

- tree.h

4.6 Tree Class Reference

`#include <tree.h>`

Public Member Functions

- **Tree ()**
  create empty tree
- **Tree (Node *, int)**
  create new tree with passed node as the new main root.
- **~Tree ()**
  destructor
- **void insert (const elements &, int)**
  inserts a new node as child of current
- **void remove (Node *)**
  deletes a node and its subtree
- **Tree (const Tree &)**
  copy constructor
- **elements value () const**
  value function
- **nid Id () const**
  Id function.
- **void left ()**
  navigate the tree go left
- **void right ()**
  go right
- **void parent ()**
  go the the parent
- **void reset ()**
- **void SetCurrent (Node *)**
  set placement of the current pointer
- **Node * pointer_left () const**
  return subtree (node) pointers returns pointer to left child
- **Node * pointer_right () const**
  returns pointer to right child

- **Node * pointer_parent () const**
  returns pointer to parent
- **Node * pointer_current () const**
  returns current pointer
- **Node * root () const**
  returns pointer to root of Tree (p. 20)
- **elements peek_left () const**
  return values of children and parent without leaving current node returns elements of left child
- **elements peek_right () const**
  returns elements of right child
- **elements peek_parent () const**
  returns elements of parent
- **void DisplayInorder (Node *) const**
  print the tree or a subtree. print an "in-order" traversal
- **void DisplayPreorder (Node *) const**
  print a "pre-order" traversal
- **void DisplayPostorder (Node *) const**
  print a "post-order" traversal
- **void clear ()**
  delete all nodes in the tree
- **bool IsEmpty () const**
  check to see if the tree is empty or full
- **bool IsFull () const**
  checks to see if tree is full

Private Member Functions

- **Node * CopyTree (Node *, Node *) const**
  create a new copy of a subtree if passed to the constructor

Private Attributes

- **Node * current**
  pointer to current Node (p. 11)

- **Node * main_root**

  pointer to the root of the Tree (p. 20)

- **bool subtree**

  does it reference a part of a larger object?

4.6.1 Detailed Description

Basic data structure that creates the decomposition tree and guide tree in the decomposition step

4.6.2 Constructor & Destructor Documentation

4.6.2.1 Tree::Tree ()

create empty tree

creates tree with default root node which has no value. set current to main root node.

4.6.2.2 Tree::Tree (Node *, int)

create new tree with passed node as the new main root.

set current to main root. if the second parameter is 0, the new object simply points to the node of the original tree. If the second parameter is 1, a new copy of the subtree is created, which the object points to.

Parameters:

Node* indicates root location

int indicates where object points to (0: node of original tree, 1: new subtree)

4.6.2.3 Tree::~Tree ()

destructor

calls the clear function to recursively remove subtrees

< delete all nodes

4.6.2.4 Tree::Tree (const Tree &)

copy constructor

performs a "deep copy" of tree object

See also:

Tree() (p. 22), **Tree(Node*, int)** (p. 22)

4.6.3 Member Function Documentation

4.6.3.1 void Tree::clear ()

delete all nodes in the tree

< use the remove function on the main root

< since there are no more items, set main_root to NULL

4.6.3.2 Node * Tree::CopyTree (Node * *root*, Node * *parent*) const [private]

create a new copy of a subtree if passed to the constructor

The second parameter is a pointer to the parent of the subtree being passed. Since parent of the main root is always NULL, we pass NULL as the second parameter in the class constructor

Parameters:

root pointer to root of tree

parent pointer to parent of **Node** (p. 11)

Returns:

pointer location of new root node

< base case - if the node doesn't exist, return NULL.

< make a new location in memory

< make a copy of the node's data

< set the new node's parent

< copy the left subtree of the current node. pass the current node as the subtree's parent

< do the same with the right subtree

< return a pointer to the newly created node.

4.6.3.3 nid Tree::Id () const

Id function.

Returns:

corresponding id of current **Node** (p. 11) (vector<short>)

See also:

value() (p. 25)

4.6.3.4 void Tree::insert (const elements &, int)

inserts a new node as child of current

inserts a new element into tree at desired position.

Parameters:

elements& list<element> that should be inserted into tree

int indicates position that node should be inserted (0: left 1:right)

if the tree has no nodes, make a root node, disregard pos.

< node created, exit the function

< if new node is a left child of current

< if child already exists, replace value

< if is left child, copy the id of the parent

< push_back the value one (we're creating a new level)

< else, new node is a right child of current

< if child already exists, replace value

< if it is a right child, copy the id of the parent

< increment the last element by one (we're on the same level)

Parameters:

pos insert as child of current 0=left 1=right. if item already exists, replace it

4.6.3.5 bool Tree::IsEmpty () const

check to see if the tree is empty or full

< If there aren't any nodes in the tree, main_root points to NULL

4.6.3.6 elements Tree::peek_left () const

return values of children and parent without leaving current node returns elements of left child

advantage: we don't have to leave the node! (self-explanatory)

4.6.3.7 void Tree::remove (Node *)

deletes a node and its subtree

recursively removes the node pointed to by Node* and all of its subtress

Parameters:

Node* points to root of tree that is to be deleted

< base case - if the root doesn't exist, do nothing

< perform the remove operation on the nodes left subtree first

< perform the remove operation on the nodes right subtree first

< if the main root is being deleted, main_root must be set to NULL

< make sure the parent of the subtree's root points to NULL, since the node no longer exists

< set current to the parent of the subtree removed.

4.6.3.8 void Tree::reset ()

go to main_root

4.6.3.9 void Tree::SetCurrent (Node *)

set placement of the current pointer

Parameters:

Node* points to location that current should be set to

4.6.3.10 elements Tree::value () const

value function

Returns:

list<element> of current **Node** (p. 11)

See also:

Id() (p. 23)

The documentation for this class was generated from the following file:

- tree.h

Index