Developer Manual for TreeScaper

Zhifeng Deng

June 4, 2020

Contents

1 Introduction

2 Basic data structures

Most of the basic data structures are constructed by Wen Huang. They includes basic array, matrix, string, mapping and file stream. They are, basically the c++ built-in structure warped up with convenient functions and operators. For example, the matrix class integrates singular value decomposition from CLAPACK. These data structures' header file and implementation files are prefixed with "w".

There are other more complicated data structures for specific algorithm and mathematical objects such as trees and community. They will be addressed in the next section.

2.1 Array

Members of array of type `T` are consisted of a pointer of `T`* `vec` and a static integer `length` that indicates the length. The member functions and operators are given below.

1. `friend std::istream &operator>>`

   This operator does nothing and will not assign value to the array from the `istream`. According to particular needs of reading data, this may later be implemented with actual reading functionality.

2. `friend std::ostream &operator<<`

   This operator will output the length and its components separated by ":" and ",".

Example: an array of `char[]` from "a" to "e" is outputted in the format of

`{ 5 :  a, b, c, d, e}`

3. `const Array& Array::operator=(const Array &right)`

   The assignment operator will free the pointer `vec` on the left and allocate a new `vec`. Then it assigns values from right hand side to left hand side component-wise.

   The operator returns a pointer of the array on the left.

4. `const Array& Array::operator+=(const Array &right)`

   This operator creates a new array with the right hand side attached to left hand side. It allocates `Array` of the correct length and then assigns values accordingly. Then call the assignment operator `=` to overwrite the current array.

   Warning: calling `=` costs repeated and unnecessary copy-pasting.

5. `const Array& Array::operator-=(const Array &right)`

   This operator creates a new array from the left hand side, with every component presented in the `right` removed and calls assignment operator to overwrite the current `Array`.

   Warning: calling `=` costs repeated and unnecessary copy-pasting.

   Warning: the new array is built incrementally and calls `resize` everytime, will brings the complexity to $O(length^2 \times right.length)$ other than $O(length \times right.length)$.

6. `bool Array::operator==`

   Compares two array component-wise after comparing the length.

7. `bool Array::operator<`

   The logic of the compare operator is to set the array with component-wise greater component to be the greater one. If the lengths are different, only compare the first k components where k is the smaller length. If the first k components happens to be the same (component-wise), the longer `Array` is greater than the shorter one.

8. `Array Array::operator(const int index, const int end)`

   This operator extract sub-array from the current array. It takes two indices as parameters and return a new array that has the values from `index` to `end`.

   Warning: this implementation is different than the implementation of `String::operator()`, which also takes two integers as parameters but the first one is the starting index and the second one is the length of the sub-string, instead of the ending index.

2.2 Matrix

Members of a matrix of type `T` are consisted of a pointer of pointers `T`** implemented inrow-major and two static integers `row` and `col` which indicate the dimensions of the matrix. This class also calls classic linear algebra algorithms from CLAPACK.

Overloaded operators are given below.

1. friend istream &operator».

   Warning: This operator does nothing, i.e., it does not assign values from the input stream.

2. friend ostream &operator«.

   This operator output the matrix in the format of

   ```
   { ( 3 , 2 )
   a, b, c
   d, e, f
   }
   ```

3. friend Matrix operator+(Matrix<T> left, Matrix<T> right).

   This operator resize the left and right matrices to the lager dimension by calling member function resize and then create a new matrix and assign values from entry-wise addition.

   Warning: the resize of left and right is silent here, which will permanently change the matrix being summed.

4. friend Matrix operator+(Matrix<T> left, S right).

   This operator accepting a number right of type S on the right will add right entry-wise to each element in Matrix left, i.e., shift the matrix by right.

5. friend Matrix operator+(S left, Matrix<T> right)

   Shift the matrix by left entry-wise.

6. friend Matrix operator-

   See friend Matrix operator+.

7. Matrix operator*(const Matrix<T> &left, const Matrix<T> &right)

   Implement matrix multiplication.

8. friend Matrix operator*(S value, Matrix<T> mat) or (Matrix<T> mat, S value).

   This operator return a new matrix entry-wise scaled by value. Note that this operator does not change the original matrix.

   Warning: the matrix getting rescaled should be passed by reference in order to avoid construction/destruction computation.

9. friend Matrix operator/(Matrix<T> mat, S value)

   This operator return a new matrix entry-wise divided by value. Note that this operator does not change the original matrix. Also note that this is not the syntax used in some advanced language where A/B means $B^{-1}A$.

   Warning: the matrix getting rescaled should be passed by reference in order to avoid construction/destruction computation.

10. `T &operator()(const int r, const int c = 0)`

    This operator returns the entry at r-row and c-column.

Important member functions are given below.

1. `Matrix<double> compute_scalar_product_matrix()`

   This function return a `double` type matrix $S \in \mathbb{R}^{n\times n}$ from the current matrix $D^{(2)} \in \mathbb{R}^{n\times n}$. S is the centering-scaled $D^{(2)}$, which is assumed to be a squared distance matrix of n points $\{p_i\}_{i=1,\cdots,n}$, $D^{(2)}_{ij} = D^{(2)}_{ji} = d^2(p_i, p_j)$.

   Note that the classical multidimensional scaling method, MDS assumes these n points lie on some Euclidean space $\mathbb{R}^k$ equipped with classic 2-norm distance. And S is a squared distance matrix of transformed n points such that the arithmetic mean of new points is $0^k \in \mathbb{R}^k$. Also note that the arithmetic mean in Euclidean space is also the Karcher mean defined by

   $$\arg\min_{m\in\mathbb{R}^k} \sum_{i=1}^{n} d^2(p_i, m).$$

   The formula of computing S is given by

   $$S = -\frac{1}{2} J D^{(2)} J$$

   where $J = I - \frac{1}{n}\mathbf{1}\mathbf{1}^T$ and $\mathbf{1}\mathbf{1}^T$ is all-1 matrix. Also note that the transformation $D^{(2)} \to S$ preserves the solution of MDS, i.e.,

   $$\arg\min_{B\in\mathbb{R}^{n\times k}} \left\| BB^T - D^{(2)} \right\|_F^2 = \arg\min_{B\in\mathbb{R}^{n\times k}} \left\| -\frac{1}{2} JBB^T J + \frac{1}{2} JD^{(2)} J \right\|_F^2$$

   where B is the Euclidean coordinate matrix of n points in $\mathbb{R}^k$ which generates (approximately) the squared distance matrix $D^{(2)}$.

   Error: The invariance of transformation seems to be only true for Euclidean space. For more abstract $D^{(2)}$ generated from more general metric on Riemannian manifold, the scaling $-\frac{1}{2}JD^{(2)}J$ does not preserves positive definiteness of the squared distance matrix, which further causes negative eigenvalues in the following PCA, the eigendecomposition, process.

   Warning: The squared distance matrix $D^{(2)}$ used in here is inconsistent with the distance matrix D computed in `compute_Distance_Matrix`, in difference of entry-wise squared or not.

2. `Matrix<double> compute_Distance_Matrix()`

   This function return a `double` type matrix $D \in \mathbb{R}^{n\times n}$ computed from the current matrix $M \in \mathbb{R}^{n\times k}$. M is consider as coordinate matrix of n points in k-dimensional Euclidean space, i-th row represents the k-tuple Euclidean coordinates of a point p_i. And D is the distance matrix where $D_{ij} = D_{ji} = d(p_i, p_j) = \|p_i - p_2 j\|_2$ is the 2-norm distance between points p_i, p_j.

   Warning: the resulting distance matrix D is dense, the symmetric structure is not exploited here.

3 Algorithms

3.1 Nonlinear dimensional reduction(NLDR)

This part collects algorithms and their important subroutines implemented in TreeScaper. The main goal in these algorithms is that given a squared distance matrix $D^{(2)}$ or distance matrix D of n points $\{p_i\}_{i=1,\cdots,n}$ on some metric space, find n points $\{p'_i\}_{i=1,\cdots,n} \subset \mathbb{R}^k$, such that the distance matrix D' for $\mathbf{p}'$ approximates D the best, under the cost functions defined in different algorithm. Note that these n new points $\mathbf{p}'$ can be represents by the coordinate matrix $B \in \mathbb{R}^{n\times k}$ which is often used as the output of these NLDR algorithms.

3.1.1 Classical Multidimensional scaling(MDS)

.

Classical Multidimensional scaling assumes $D^{(2)}$ is generated from Euclidean space with typical vector 2-norm as distance. The implemented algorithm `NLDR::CLASSIC_MDS` contains 4 parts:

1. Compute centered matrix S from the given tree distance matrix D by calling

   `Matrix::compute_Scalar_Matrix`.

2. Perform singular value decompositions(SVD) to S by calling `Matrix::SVD_LIB` to obtain

   $$S = U\Sigma V^T.$$

   Note that since S is symmetric, there exist eigen-decomposition $S = Q\Lambda Q^T$, i.e., there exists a signature matrix E, which has only 1 or -1 in diagonal and 0 elsewhere, such that $U\Sigma V^T = Q\Lambda EEQ^T = Q(\Lambda E)(QE)^T$ and $U = Q$, $\Sigma = \Lambda E$ and $V = QE$. This implies eigen-decomposition from CLAPACK is more efficient.

   Also note that if $D^{(2)}$ uses vector 2-norm in Euclidean space, S is positive definite and SVD coincides with eigen-decomposition.

   Warning: when there exist files named consistently that indicates SVD has been done and U, V, Σ has been stored, the routine will not do it again but simply read them from files. This is silent and could cause problem if those files are not actually inconsistent.

3. In case of performing MDS for $D^{(2)}$ from other metric space, which cause the presence of negative eigenvalues, it selects the k eigenvectors Q_{i_j}, $j = 1, \cdots k$, where λ_{i_j} are the k most largest positive eigenvalues.

   Error: memory leakage happens in this process whenever a negative eigenvalues encountered in the k most largest in magnitude eigenvalues. This problem is temporarily fixed but the theoretical explanation and necessity of this process is still needed. the classical MDS may not be suitable at all for Tree subjects.

4. Produce the coordinates matrix B for n points $\mathbf{p}' \subset \mathbb{R}^k$ by

   $$B = \begin{bmatrix} \sqrt{\lambda_{i_1}} Q_{i_1} & \cdots & \sqrt{\lambda_{i_k}} Q_{i_k} \end{bmatrix} \in \mathbb{R}^{n\times k}.$$

5. Compute the stress that estimate how good BB^T approximate $D^{(2)}$ by calling NLDR::CLASSIC_MDS_stres

   Note that classical MDS do not need to compute the stress since B already minimized the stress function. However, since Tree space is not a Euclidean space with appropriate distance, the output B does not minimize the stress function.

For more information of MDS, see here.

Implementations of some routines

Data structure

1. `Matrix`

Description	Row-major 2-dimensional array.	
Member	`row`	Number of rows.
	`col`	Number of columns.
	`**matrix`	Pointers to each row.
Member function	`resize`	Change the dimensions.

2. `Ptree`

Description	Index base array-type unweighted tree with adjacency matrix.	
Member	`leaf_number`	
	`*parent`	Array of indices of the parent.
	`*lchild`	Array of indices of the right child.
	`*rchild`	Array of indices of the left child.
	`**edge`	Adjacency matrix.
Member function	none	

3. `NEWICKNODE`

Description	Linked node pointed to its children and parent.	
Member	`Nchildren`	Number of children.
	`label`	
	`weight`	
	`*child`	List of children.
	`hv1`	Hash value for unknown use.
	`hv2`	Hash value that identifies the bipartition.
	`bitstr`	Bit string that represents the leaves contained in the (sub-)tree.
	`parent`	
Member function	none	

4. `NEWICKTREE`

Description	A `NEWICKNODE` that represents the root.	
Member	`root`	A `NEWICKNODE`.
Member function	none	

5. `TreeOPE`

Description	Operation associated to one NEWICKTREE. Note that most of the method are implemented in recursive pre-order.	
Member		
Member function	loadnewicktree	Read NEWICKTREE.
	loadnewicktree2	Read NEWICKTREE.
	floadnewicktree	Read NEWICKTREE.
	loadnode	Read NEWICKTREE.
	loadleaf	Read NEWICKTREE.
	parsetree	Read NEWICKTREE.
	parsenode	Read NEWICKTREE.
	parseleaf	Read NEWICKTREE.
	addchild	Link child to the parent.
	dfs_compute_hash	Assigned hash values to all (sub-)tree which identifies the structure and therefore the bipartition.
	bipart	Store hash values in one big array for computing RF distance.
	findleaf	Find a leaf by the NEWICKNODE::label.
	normalizedTree	Lift a unrooted tree to a rooted tree.
	newick2lcbb	Convert NEWICKTREE to Ptree for computing matching distance.
	newick2ptree	Implementation of newick2lcbb.
	sumofdegree	
	bipartcount	Count the occurrence of particular bipartition.
	Addbipart	Insert nodes to the current tree so that there exist a (sub-)tree that contains only a given set of leaves.

Description	Multiple NEWICKTREEs with member function that computes different distances.	
Member		
Member function	`initialTrees`	Read trees from file.
	`ReadTrees`	Read trees from file.
	`compute_numofbipart`	
	`Compute_Hash`	Generate hash table for computing hash values in a tree.
	`Compute_Bipart_Matrix`	Generate a sparse matrix that stores the weight of bipartition, its frequency of occurrence.
	`Compute_Bipart_Covariance`	Generate the covariance matrix according to the formula.
	`Compute_RF_dist_by_hash`	Generate the RF-distance matrix according to the formula.
	`pttree`	Construct the adjacency matrix of a `Ptree`.
	`compute_matrix`	Generate matrix for computing matching distance by accumulating common edges from two `Ptree`s.
	`Compute_Matching_dist`	Compute the matching distance between two trees by the XOR table created from all possible bipartitions.
	`Compute_Affinity_dist`	Compute the affinity distance from the given distance matrix.

`TreeOPE` related routines.

1. `TreeOPE::loadnewicktree`.

Argument	(char *fname, int *error)	
Description	Read tree from formatted string that stores bipartition. The implementation is given in floadnewicktree. Same level of the node is paired by "()" and separated by ",".	
Complexity **Memory space**		
Associated routine	floadnewicktree	Implementation by recursive processing the string in preorder.
Comments	This routine is better implemented by stack structure. It can only process unweighted tree. Also this routine takes the file name as input while the duplication version loadnewicktree2 takes FILE type, customized fstream type. This routine seems to be insecure and redundant.	
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.

2. TreeOPE::loadnewicktree2.

Argument	(FILE *fp, int *error)	
Description	Duplication version of loadnewicktree but with customized fstream. Actual implementation is not given in here, but in floadnewicktree	
Complexity **Memory space**		
Associated routines	floadnewicktree	Implementation by recursive processing the string in preorder.
Comments	This routine also seems to be redundant since the main thread of TreeScaper never called it. There is another input routine parsetree, which can handle both weighted and unweighted tree, is used in TreeScaper.	
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.

3. TreeOPE::floadnewicktree.

Argument	`(FILE *fp, int *error)`
Description	A pair of nodes are created by `loadnode` when "(" is encountered.
Complexity **Memory space**	
Associated routine	`loadnode`
Comments	This routine also seems to be redundant since the main thread of TreeScaper never called it. There is another input routine `parsetree`, which can handle both weighted and unweighted tree, is used in TreeScaper.
Error code	-1 Out of memory. -2 Parse error, the parentheses in string does not match.

4. `TreeOPE`::`loadnode`.

Argument	`(FILE *fp, int *error)`
Description	Create internal nodes. When this function is called, a "(" has been read, if `fp` continue to read "(", next pair of nodes should be generated, i.e., `loadnode` is called again, otherwise a leaf is encountered and `loadleaf` will be called. When ")" is encountered, it is at the end of the current pair of nodes and should exit the routine to returned to previous level of node.
Complexity **Memory space**	
Associated routine	`loadleaf` Add a leaf and return to previous level. `addchild` Add the new pair of nodes to their parent. `readlabelandweight` Read additional information from string.
Comments	This is better implemented by stack structure. Also note that this method read leaves in preorder traversal.
Error code	-1 Out of memory. -2 Parse error, the parentheses in string does not match.

5. `TreeOPE`::`parsetree`.

Argument	`(char *str, int *error, NEWICKTREE *testtree)`
Description	Duplicate version of `floadnewicktree`.
Complexity **Memory space**	
Associated routine	`parsenode`
Comments	This is the routine used in TreeScaper.
Error code	-1 Out of memory. -2 Parse error, the parentheses in string does not match.

6. TreeOPE::`parsenode`.

Argument	`(FILE *fp, int *error)`	
Description	Duplicated version `loadnode`.	
Complexity		
Memory space		
Associated routine	`parseleaf`	Add a leaf and return to previous level.
	`addchild`	Add the new pair of nodes to their parent.
	`parselabelandweight`	Read additional information from string.
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.

7. TreeOPE::`dfs_compute_hash`.

Argument	(NEWICKNODE* startNode, LabelMap &lm, HashRFMap &vec_hashrf, unsigned treeIdx, unsigned &numBitstr, unsigned long long m1, unsigned long long m2, bool WEIGHTED, unsigned int NUM_Taxa, map<unsigned long long, Array<char> *> &hash2bitstr, int numofbipartions)	
Description	It assigned hash value to all leaves set, for internal node, the hash values are computed by the sum of its children's hash values (and mod m1 or m2). For each internal node, it determines a sub-tree rooted by itself from the current tree. Such subtree is uniquely represented by the hash value of its root. The leaves contained in the subtree are also represented by the bit string. For example, 01001100 represents that the subtree contains leaf 2, 5 and 6. The mapping from hash values to the leaves it contain is stored in hash2bitstr.	
Complexity **Memory space**		
Associated routine	Array::SetBitArray	Set the some positions, the index of leaves, of a bit array to 1.
	Array::OrbitOPE	OR operation of bit array, it realizes the functionality of making the bit string of the root having 1 in every leaf's index that the subtree has.
	add_of	Bit-wise addition for hash values.
Comments	Note that hash value to subtree is bijection and subtree to leaves it contains is subjection. Therefore, the mapping hash2bitstr is subjection. Also note that the operations, addition and modulus, on hash values are done in bit-wise manner.	
Error code	none	Terminate with specific error message (overflow in hash value additions).

8. TreeOPE::bipart.

Argument	`(NEWICKNODE *const startnode, unsigned int &treeIdx, unsigned long long *matrix_hv, unsigned int *matrix_treeIdx, double *matrix_weight, int &idx, int depth, bool isrooted)`	
Description	Store hash values, TreeIdx and weights in the given arrays.	
Complexity **Memory space**		
Associated routine		
Comments	Note that the "TreeIdx" is an identical array. Each tree will generate one set of such arrays and these arrays from different trees are pasted together and sorted by the hash values. By comparing hash values, identical bipartitions among different trees can be easily found.	
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.

9. TreeOPE::`findleaf`.

Argument	`(std::string leafname, NEWICKNODE *currentnode, NEWICKNODE *parent, int *icpt)`
Description	Find leaf `leafname` and return it. icpt also record which subtree under root the leaf lies in.
Complexity **Memory space**	
Associated routine	none

10. TreeOPE::`normalizedTree`.

Argument	`(NEWICKNODE *lrpt, NEWICKTREE *newickTree, int indexchild)`	
Description	Lift a unrooted tree to a rooted tree.	
Complexity **Memory space**		
Associated routine	`normalizedNode`	It's implementation.

11. TreeOPE::`newick2lcbb`.

Argument	(const NEWICKTREE *nwtree, int num_leaves, struct Ptree *tree)	
Description	Convert NEWICKTREE to Ptree, which is used to compute matching distance.	
Complexity		
Memory space		
Associated routine	newick2ptree	Implementation of newick2lcbb.
Comments	Note that Ptree does not stored hash values and weights, i.e., the bipartition and weight information are lost. Also note that the edges matrix of Ptree is not computed here.	

12. TreeOPE::sumofdegree.

Argument	(NEWICKNODE *node, bool isrooted, int depth)	
Description	Return the sum of degrees of all nodes.	
Complexity		
Memory space		
Associated routine		
Comments		
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.

13. TreeOPE::bipartcount.

Argument	(NEWICKNODE *node, bool isrooted, map<unsigned long long, unsigned long long> &bipcount, int depth)
Description	Count the occurrence of particular subtree, bipartition, by its hash value and store the result in the external mapping bipcount
Complexity	
Memory space	
Associated routine	
Comments	

14. TreeOPE::Addbipart.

Argument	(NEWICKNODE* startNode, double freq, unsigned long long hash, Array<char> &bitstr, int NumTaxa, bool &iscontained)
Description	Given bitstr that represents a set of leaves. Insert internal nodes from leaf-set to root that collects those leaves lie in bitstr so that there is a subtree containing exactly the same set of leaves in the resulting new tree.
Complexity	
Memory space	
Associated routine	none
Comments	There is a better way to implement this functionality.

Trees related routines.

1. Trees::initialTrees.

Argument	(string fname)	
Description	Initialize a set of NEWICKEDTREEs by calling loadnewickedtree2. For Nexus trees, it only create a leaveslabelsmaps that stores the labels of leaf set.	
Complexity		
Memory space		
Associated routine	loadnewicktree2	Create each tree.
Comments	Complicated string operations are done here, which is unnecessary.	
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.
	-3	Failure of opening file.

2. Trees::ReadTrees.

Argument	none	
Description	A duplicated version of initialTrees except it calls parsetree for both Newicked and NEXUS type of tree. Also lifted the tree if it is unrooted.	
Complexity		
Memory space		
Associated routine	parsetree	Create each tree.
	normalizedTree	Lift a unrooted tree.
Comments	Very complicated string operations are done here, which is really unnecessary.	
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.
	-3	Failure of opening file.

3. Trees::compute_numofbipart.

Argument	none
Description	It computes the numbers of bipartition for all trees and stores them in the array numberofbipartition. The formula is given by $$s/2 - n$$ where s is the sum of degrees and n is the number of leaf.
Complexity	
Memory space	
Associated routine	sumofdegree

4. Trees::Compute_Hash.

Argument	`none`
Description	Generate the hash table for computing the hash values in a tree.
Complexity **Memory space**	
Associated routine	`dfs_compute_hash`

5. Trees::`Compute_Bipart_Matrix`.

Argument	`none`	
Description	The arrays of indivial tree's hashvalue, tree index and weight created from `bipart` were combined and sorted. Since the hash value represents the unique subtree structure, i.e.. a bipartition, the number of unique bipartion can be counted via checking the hash value. As a result, a sparse bipartition matrix that stores weight of unique bipartition versus trees is created.	
Complexity **Memory space**		
Associated routine	`bipart`	Create arrays of hash values, weights with tree index of one tree.
	`Sort`	Sort the 3 arrays attached from all trees by the hash values, so that we can easily count the occurrence for each hash value, i.e., bipartition.
	`sort`	Seems to be built-in sort for array that sort a temperate hash value array for certain later operation.
Comments	The `sort` which is different then `Sort` is confusing here. Is it the default sort in c++?	

6. Trees::`Vec_multiply`.

Argument	`(const double* Vec1, const double* Vec2, int Unique_idx)`
Description	It return a rank-1 matrix $$M = v_1 v_2^T.$$
Complexity **Memory space**	
Associated routine	none
Comments	It is confusing with the `SparseMatrix`::`Multiply_vec` and should be integrated in `Vector` class.

7. Trees::`Compute_Bipart_Covariance`.

Argument	`(bool ISWEIGHTED)`
Description	Compute the bipartition covariance matrix from the matrix, `C`, created by Compute_Bipart_Matrix, `M`. Let $M_1 = MM^T$, $v_1 = mean(M)$, $v_2 = sum(M)$, $M_2 = v2v1^T$ and $M_3 = v1v1^T$, then $$C = (M_1 - M_2 - M_2^T + n * M_3)/(n-1).$$
Complexity **Memory space**	
Associated routine	SparseMatrix::transpose SparseMatrix::Multiply — Matrix-Matrix multiplication. SparseMatrix::Mean — Matrix mean. SparseMatrix::Multiply_vec — Matrix-vector multiplication. Trees::Vec_Multiply — Rank-1 matrix.
Comments	Note that it is implemented via sparse matrix-vector multiplication.

8. Trees::`Compute_RF_dist_by_hash`.

Argument	`(bool ISWEIGHTED)`
Description	Compute the unweighted/weighted RF distance. For the unweighted distance, accumulate the number of each unique bipartition's occurrencein each tree, f_{ij}, and the number of bipartitions, n_i, then $$d_{ij} = \frac{n_i + n_j - 2f_{ij}}{2}.$$ For weighted case, it is more complicated. The result is stored in the matrix `dist_URF` or `dist_RF`.
Complexity **Memory space**	
Associated routine	none
Comments	none

9. Trees::`pttree`.

Argument	`(struct Ptree *treeA, int node)`
Description	It constructs the edge matrix of `treeA` which should be implemented in Ptree.
Complexity **Memory space**	
Associated routine	none

10. Trees::`compute_matrix`.

Argument	`(int *r, int range, struct Ptree *tree1, struct Ptree *tree2)`
Description	It accumulates the number common edges from two trees and store in a vectorized matrix, `r`.
Complexity **Memory space**	
Associated routine	none
Comments	For n trees, there are $\binom{n}{2} = n(n-1)$ comparisons and this function will be called $n(n-1)$ times.

11. Trees::`tree_mmdis`.

Argument	`none`
Description	This distance is given by the solution of Hungarian algorithm of the cost matrix, `r`, given by `compute_matrix`.
Complexity **Memory space**	
Associated routine	`array_to_matrix` Recover `r` to a matrix.
Comments	`r` is an $(k-3) \times (k-3)$ matrix where k is the number of leaves. The main complexity goes into generating distance matrix and running Hungarian algorithm.

12. Trees::`Compute_Matching_dist`.

Argument	`none`
Description	The matching distance is given by the solution to Hungarian algorithm on the table with entries of number of XOR element in `bitstrofatree`, which are all possible bipartitions of one tree.
Complexity **Memory space**	
Associated routine	`Get_bipartitionofonetree`
Comments	Line 1415 may have a bug.

13. Trees::`Compute_Affinity_dist`.

Argument	`(String str_matrix, int type)`
Description	This routine compute the affinity distance, d_a, from the given distance ,d. The formula is either $$d_a = \frac{1}{\varepsilon_{rel} + d}$$ or $$d_a = e^{-d},$$ depending on the flag `type`. It accepts unweighted/weighted RF-distance, Matching-distance, SPR-distance or distance given in file.
Complexity **Memory space**	
Associated routine	none

14. Trees::`temp`.

Argument	`none`	
Description		
Complexity		
Memory space		
Associated routine		
Comments		
Error code	-1	Out of memory.
	-2	Parse error, the parentheses in string does not match.