gemoda Reference Manual

Generated by Doxygen 1.4.4

Fri Mar 3 09:05:14 2006

Contents

1	geme	oda Directory Documentation	1
	1.1	FastaSeqIO/ Directory Reference	1
2	geme	oda Data Structure Documentation	3
	2.1	bitGraph_t Struct Reference	3
	2.2	bitSet_t Struct Reference	5
	2.3	cnode Struct Reference	7
	2.4	cSet_t Struct Reference	9
	2.5	fSeq_t Struct Reference	11
	2.6	mnode Struct Reference	12
	2.7	rdh_t Struct Reference	13
	2.8	sHash_t Struct Reference	15
	2.9	sHashEntry_t Struct Reference	17
	2.10	sOffset_t Struct Reference	19
	2.11	sPat_t Struct Reference	21
	2.12	sSize_t Struct Reference	23
3	geme	oda File Documentation	25
	3.1	align.c File Reference	25
	3.2	bitSet.c File Reference	30
	3.3	bitSet.h File Reference	51
	3.4	convll.c File Reference	65
	2.5	aonyll h Eila Dafaranaa	06

ii CONTENTS

3.6	FastaSeqIO/fastaSeqIO.c File Reference	110
3.7	FastaSeqIO/fastaSeqIO.h File Reference	119
3.8	gemoda-r.c File Reference	122
3.9	gemoda-s.c File Reference	134
3.10	matdata.h File Reference	154
3.11	matrices.c File Reference	155
3.12	matrices.h File Reference	157
3.13	matrixmap.h File Reference	166
3.14	newConv.c File Reference	168
3.15	patStats.c File Reference	182
3.16	patStats.h File Reference	198
3.17	realCompare.c File Reference	201
3.18	realCompare.h File Reference	209
3.19	realIo.c File Reference	212
3.20	realIo.h File Reference	239
3.21	spat.h File Reference	243
3.22	words.c File Reference	244

Chapter 1

gemoda Directory Documentation

1.1 FastaSeqIO/ Directory Reference

FastaSeqIO

Files

- file fastaSeqIO.c
- file fastaSeqIO.h

2	gemoda Directory Documentation

Chapter 2

gemoda Data Structure Documentation

2.1 bitGraph_t Struct Reference

#include <bitSet.h>

Collaboration diagram for bitGraph_t:



Data Fields

- int size
- bitSet_t ** graph

2.1.1 Detailed Description

A bit graph is an array of bit sets. The graph must be of size size x size. This data structure is used to store adjacency matrices. In particular, a bit graph is used in the clustering step. It can easily be considered a set of sets.

Definition at line 48 of file bitSet.h.

2.1.2 Field Documentation

2.1.2.1 bitSet t** bitGraph t::graph

A pointer used to store an array of bitSet_t space objects.

Definition at line 56 of file bitSet.h.

Referenced by bitGraphCheckBit(), bitGraphRowIntersection(), bitGraphRowUnion(), bitGraphSetFalse(), bitGraphSetFalseDiagonal(), bitGraphSetFalseSym(), bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), copyBitGraph(), countBitGraphNonZero(), deleteBitGraph(), emptyBitGraph(), emptyBitGraphRow(), fillBitGraph(), filterIter(), findCliques(), getStatMat(), maskBitGraph(), newBitGraph(), printBitGraph(), pruneBitGraph(), and singleLinkage().

2.1.2.2 int bitGraph_t::size

The total size of a bit graph, which is assumed to be symmetric. There are *size* bit sets in a bit graph, each of size *size*.

Definition at line 53 of file bitSet.h.

Referenced by convolve(), copyBitGraph(), filterGraph(), findCliques(), getStatMat(), main(), newBitGraph(), and oldGetStatMat().

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

• bitSet.h

2.2 bitSet_t Struct Reference

#include <bitSet.h>

Data Fields

- int max
- int slots
- int bytes
- bit_t * tf

2.2.1 Detailed Description

A bit set is a data structure for storing set objects that allows for quick set operations such as intersections, unions, differences, and so forth. On a standard 32-bit architecture, 32 operations can be performed at the same time, greatly speeding the clique finding stage of the algorithm.

Definition at line 24 of file bitSet.h.

2.2.2 Field Documentation

2.2.2.1 int bitSet_t::bytes

This variable actually holds the total number of bits, rather than the number of bytes. However, we chose to keep this name rather than make a variety of changes.

Definition at line 37 of file bitSet.h.

Referenced by emptySet(), fillSet(), and newBitSet().

2.2.2.2 int bitSet_t::max

The maximum integer that can be set to true or false.

Definition at line 28 of file bitSet.h.

Referenced by newBitSet(), nextBitBitSet(), setFalse(), and setTrue().

2.2.2.3 int bitSet_t::slots

The total number of slots, where a slot holds a number of bits equal to the size of a bit_t space object.

Definition at line 32 of file bitSet.h.

Referenced by bitSet3WayIntersection(), bitSetDifference(), bitSetIntersection(), bitSetSum(), bitSetUnion(), copySet(), and newBitSet().

2.2.2.4 **bit_t* bitSet_t::tf**

A pointer to a bit_t, which is used to store an array of these objects.

Definition at line 40 of file bitSet.h.

Referenced by bitSet3WayIntersection(), bitSetDifference(), bitSetIntersection(), bitSetSum(), bitSetUnion(), checkBit(), copySet(), countSet(), deleteBitSet(), emptySet(), fillSet(), flipBits(), newBitSet(), nextBitBitSet(), printBinaryBitSet(), setFalse(), and setTrue().

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

• bitSet.h

2.3 cnode Struct Reference

#include <convll.h>

Collaboration diagram for cnode:



Data Fields

- $cSet_t * set$
- int id
- int length
- cnode * next
- double stat

2.3.1 Detailed Description

This data structure is a linked list for storing cliques. Each member of the linked list has a set, an ID number, a length (which gives the number of characters in the motif), a pointer to the next member of the linked list, and a floating-point number for storing statistical information.

Definition at line 35 of file convll.h.

2.3.2 Field Documentation

2.3.2.1 int cnode::id

Identification number for this member.

Definition at line 38 of file convll.h.

Referenced by addToStacks(), printCll(), printCllPattern(), pushCll(), removeSupers(), singleCliqueConv(), sortByStats(), swapNodecSet(), uniqClique(), wholeCliqueConv(), wholeRoundConv(), and yankCll().

2.3.2.2 int cnode::length

Length of this motif.

Definition at line 41 of file convll.h.

Referenced by calcStatCliq(), getLargestLength(), main(), outputRealPats(), outputRealPatsWCentroid(), printCll(), and pushCll().

2.3.2.3 struct cnode* cnode::next

A pointer to the next member, or the next motif.

Definition at line 42 of file convll.h.

Referenced by calcStatAllCliqs(), fillMemberStacks(), getLargestLength(), getLargestSupport(), main(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), pruneCll(), pushCll(), removeSupers(), singleCliqueConv(), sortByStats(), swapNodecSet(), uniqClique(), wholeRoundConv(), and yankCll().

2.3.2.4 cSet_t* cnode::set

The set for this member of the linked list.

Definition at line 37 of file convll.h.

Referenced by addToStacks(), calcStatCliq(), findCliqueCentroid(), getLargest-Support(), initheadCll(), main(), makeAlternateCentroid(), mergeIntersect(), output-RealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), prune-Cll(), pushCll(), removeSupers(), singleCliqueConv(), swapNodecSet(), uniqClique(), and wholeCliqueConv().

2.3.2.5 double cnode::stat

Used to store the statistical store of a motif.

Definition at line 43 of file convll.h.

 $Referenced\ by\ calcStatAllCliqs(),\ main(),\ outputRealPats(),\ and\ pushCll().$

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

• convll.h

2.4 cSet_t Struct Reference

#include <convll.h>

Data Fields

- int size
- int * members

2.4.1 Detailed Description

A cSet_t is used to hold a set of integers, in cases where the upper limit of integers size is unknown. Or, in cases where using a bit set would be impractical. This data structure is used throughout the convolution, where we have found heuristically that intersections of this data type are much faster than those for bitSet_t's, which would require a bit shift.

Definition at line 21 of file convll.h.

2.4.2 Field Documentation

2.4.2.1 int* cSet_t::members

Array of pointers to ints that holds the members of this set.

Definition at line 26 of file convll.h.

Referenced by addToStacks(), bitSetToCSet(), checkCliquecSet(), findClique-Centroid(), main(), makeAlternateCentroid(), mergeIntersect(), mllToCSet(), output-RealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), print-CSet(), pruneCll(), pushConvClique(), removeSupers(), swapNodecSet(), uniq-Clique(), and wholeCliqueConv().

2.4.2.2 int cSet_t::size

Number of members in this set.

Definition at line 24 of file convll.h.

Referenced by bitSetToCSet(), calcStatCliq(), checkCliquecSet(), findClique-Centroid(), getLargestSupport(), main(), mllToCSet(), outputRealPats(), output-RealPatsWCentroid(), printCll(), printCllPattern(), printCSet(), pruneCll(), remove-Supers(), singleCliqueConv(), uniqClique(), and wholeCliqueConv().

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

• convll.h

2.5 fSeq_t Struct Reference

#include <fastaSeqIO.h>

Data Fields

- char * seq
- char * label

2.5.1 Detailed Description

Definition at line 12 of file fastaSeqIO.h.

2.5.2 Field Documentation

2.5.2.1 char* fSeq_t::label

Definition at line 14 of file fastaSeqIO.h.

Referenced by FreeFSeqs(), initAofFSeqs(), and ReadFSeqs().

2.5.2.2 **char*** **fSeq_t::seq**

Definition at line 13 of file fastaSeqIO.h.

Referenced by FreeFSeqs(), initAofFSeqs(), printFSeqSubSeq(), ReadFSeqs(), and ReadTxtSeqs().

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

• FastaSeqIO/fastaSeqIO.h

2.6 mnode Struct Reference

#include <convll.h>

Collaboration diagram for mnode:



Data Fields

- int cliqueMembership
- mnode * next

2.6.1 Detailed Description

This data structure is just a link to list of integers used for bookkeeping during the convolution stage.

Definition at line 49 of file convll.h.

2.6.2 Field Documentation

2.6.2.1 int mnode::cliqueMembership

Clique to which this belongs.

Definition at line 52 of file convll.h.

Referenced by mllToCSet(), printMemberStacks(), pushMemStack(), and setStack-True().

2.6.2.2 struct mnode* mnode::next

A pointer to the next member in the linked list of mll_t space objects.

Definition at line 55 of file convll.h.

 $Referenced\ by\ mllToCSet(),\ popMemStack(),\ printMemberStacks(),\ pushMemStack(),\ and\ setStackTrue().$

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

• convll.h

2.7 rdh_t Struct Reference

#include <realIo.h>

Data Fields

- int size
- int indexSize
- char ** label
- gsl_matrix ** seq
- int * indexToSeq
- int * indexToPos
- int ** offsetToIndex

2.7.1 Detailed Description

This is a data structure, which is used to store real valued data. Basically, this is an array of gsl_matrix objects, where each matrix represents a single, multidimensional array that was read in from a FastA formatted file.

Definition at line 24 of file realIo.h.

2.7.2 Field Documentation

2.7.2.1 int rdh_t::indexSize

The size of the index, where the index is used to store pointers to the different sequences in this object.

Definition at line 30 of file realIo.h.

 $Referenced\ by\ getRdhIndexSeqPos(),\ initRdh(),\ initRdhIndex(),\ realComparison(),\ and\ setRdhIndex().$

2.7.2.2 int* rdh_t::indexToPos

The array of integers that tell us to which position in a sequence each index in the gsl_matrix array corresponds.

Definition at line 40 of file realIo.h.

Referenced by freeRdh(), getRdhIndexSeqPos(), initRdh(), initRdhIndex(), and set-RdhIndex().

2.7.2.3 int* rdh_t::indexToSeq

The array of integers that will tell us to which sequence each index and the gsl_matrix array corresponds.

Definition at line 37 of file realIo.h.

Referenced by freeRdh(), getRdhIndexSeqPos(), initRdh(), initRdhIndex(), main(), and setRdhIndex().

2.7.2.4 char** rdh_t::label

The array of labels that store the names of each sequence.

Definition at line 32 of file realIo.h.

Referenced by freeRdh(), getRdhLabel(), initRdh(), and setRdhLabel().

2.7.2.5 int** rdh_t::offsetToIndex

The array that points from a particular offset to its index.

Definition at line 42 of file realIo.h.

Referenced by freeRdh(), initRdhIndex(), and main().

2.7.2.6 gsl_matrix** rdh_t::seq

The array of matrices that store the data we read in.

Definition at line 34 of file realIo.h.

Referenced by freeRdh(), generalMatchFactor(), getRdhDim(), getRdhSeqLength(), getRdhValue(), initRdh(), initRdhGslMat(), massSpecCompareWElut(), outputReal-Pats(), rmsdCompare(), setRdhColFromString(), setRdhLabel(), and setRdhValue().

2.7.2.7 int rdh t::size

The number of sequences stored in this data structure.

Definition at line 27 of file realIo.h.

Referenced by initRdh(), initRdhIndex(), and main().

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

• realIo.h

2.8 sHash_t Struct Reference

Collaboration diagram for sHash_t:



Data Fields

- int * hashSize
- int * iHashSize
- int totalSize
- sHashEntry_t ** hash

2.8.1 Detailed Description

A data structure for a hash table. At its root, this structure is just an array of hash entry objects. As well, there are members used to track the size of the hash table.

Definition at line 132 of file words.c.

2.8.2 Field Documentation

2.8.2.1 sHashEntry_t** sHash_t::hash

An array sHashEntry_t space objects.

Definition at line 148 of file words.c.

Referenced by destroySHash(), printSHash(), and searchSHash().

2.8.2.2 int* sHash_t::hashSize

A pointer to an integer that is used to store an array of integers that keep track of the number of sHashEntry_t objects that are hashed to a particular integer.

Definition at line 138 of file words.c.

Referenced by destroySHash(), and searchSHash().

2.8.2.3 int* sHash_t::iHashSize

A pointer to an integer that is used to store an array of integers that keep track of the number of sHashEntry_t objects that are hashed to a particular integer.

Definition at line 143 of file words.c.

Referenced by destroySHash(), and searchSHash().

2.8.2.4 int sHash_t::totalSize

An integer that stores the total number of slots available in our hash.

Definition at line 146 of file words.c.

Referenced by initSHash(), and searchSHash().

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

• words.c

2.9 sHashEntry_t Struct Reference

Data Fields

- char * key
- int L
- int data
- int idx

2.9.1 Detailed Description

Type for a hash table entry. This datatype is used to populate a hash table. The most important members of this data structure are the string, or the key, and the index to which that key hashes.

Definition at line 114 of file words.c.

2.9.2 Field Documentation

2.9.2.1 int sHashEntry_t::data

A throw away variable, used to store any necessary data

Definition at line 121 of file words.c.

Referenced by countWords2(), and printSHash().

2.9.2.2 int sHashEntry_t::idx

The integer to which the key of length L hashes

Definition at line 123 of file words.c.

Referenced by countWords2().

2.9.2.3 char* sHashEntry_t::key

A pointer to a string

Definition at line 117 of file words.c.

Referenced by countWords2(), printSHash(), and searchSHash().

2.9.2.4 int sHashEntry_t::L

The length of the string that should be used to compute the hash

Definition at line 119 of file words.c.

Referenced by countWords2(), printSHash(), and searchSHash().

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

• words.c

2.10 sOffset_t Struct Reference

#include <spat.h>

Data Fields

- int seq
- int pos
- int next
- int prev

2.10.1 Detailed Description

This object is used to store the location of a particular word and a set of sequences. That is if we hash a word, we would like to know where it came from. This data structure provides that information.

Definition at line 13 of file spat.h.

2.10.2 Field Documentation

2.10.2.1 int sOffset_t::next

The index of the word that follows this word at *pos* plus 1.

Definition at line 23 of file spat.h.

Referenced by countWords2().

2.10.2.2 int **sOffset_t::pos**

The position in the sequence where the word is located.

Definition at line 20 of file spat.h.

Referenced by countWords2(), and main().

2.10.2.3 int sOffset_t::prev

The index of the word that precedes this word at *pos* minus 1.

Definition at line 26 of file spat.h.

Referenced by countWords2().

2.10.2.4 int sOffset_t::seq

The sequence from which the word came.

Definition at line 17 of file spat.h.

Referenced by countWords2(), and main().

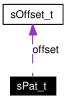
The documentation for this struct was generated from the following file:

• spat.h

2.11 sPat_t Struct Reference

#include <spat.h>

Collaboration diagram for sPat_t:



Data Fields

- char * string
- int length
- int support
- sOffset_t * offset

2.11.1 Detailed Description

This data structure is used to store the locations of all the instances of a particular word of length *length* in a set of sequences. This data structure is used principally by the string based version of Gemoda and is used to store words that are hashed before the comparison phase.

Definition at line 36 of file spat.h.

2.11.2 Field Documentation

2.11.2.1 int **sPat_t::length**

The length of this word.

Definition at line 43 of file spat.h.

Referenced by countWords2(), and printSPats().

2.11.2.2 sOffset_t* sPat_t::offset

An array of sOffset_t objects storing the loci, or offsets where this word occurs.

Definition at line 50 of file spat.h.

Referenced by countWords2(), destroySPatA(), and main().

2.11.2.3 char* sPat_t::string

The pointer to the string for this word.

Definition at line 40 of file spat.h.

Referenced by countWords2().

2.11.2.4 int sPat_t::support

The number of times this word occurs in the sequence set.

Definition at line 46 of file spat.h.

Referenced by countWords2().

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

• spat.h

2.12 sSize_t Struct Reference

Data Fields

- int start
- int stop
- int size

2.12.1 Detailed Description

Definition at line 165 of file fastaSeqIO.c.

2.12.2 Field Documentation

2.12.2.1 int **sSize_t::size**

Definition at line 168 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

2.12.2.2 int sSize_t::start

Definition at line 166 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

2.12.2.3 int sSize_t::stop

Definition at line 167 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

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

• FastaSeqIO/fastaSeqIO.c

gemoda Da	ta Structur	e Documentation

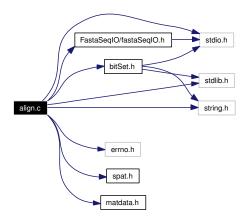
24

Chapter 3

gemoda File Documentation

3.1 align.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "FastaSeqIO/fastaSeqIO.h"
#include "spat.h"
#include "bitSet.h"
#include "matdata.h"
Include dependency graph for align.c:
```



Defines

• #define ALIGN_ALPHABET 256

Functions

- int alignMat (char *s1, char *s2, int L, int mat[][MATRIX_SIZE])
- bitGraph_t * alignWordsMat_bit (sPat_t *words, int wc, int mat[][MATRIX_-SIZE], int threshold)

Variables

• const int aaOrder []

3.1.1 Detailed Description

This file defines functions that are used to create a similarity graph, or adjacency matrix via the comparison of small windows within a set of sequences. This file is only used for string based sequences, and not real valued data. Usually, the adjacency matrix is created via a the alignment of the windows within the sequence set. Thus, the name of this file. However, other functions can certainly be defined for creating the adjacency matrix.

Definition in file align.c.

3.1.2 Define Documentation

3.1.2.1 #define ALIGN ALPHABET 256

Definition at line 24 of file align.c.

3.1.3 Function Documentation

3.1.3.1 int alignMat (char *s1, char *s2, int L, int mat[][MATRIX_SIZE])

This function takes as its arguments two pointers to strings, a length, and a scoring matrix. The function computes the score, or degree of similarity, between the two strings by comparing each character the in the strings from zero two L minus one. Each character receives a score that is looked up in the scoring matrix. This is most commonly used for amino acid sequences or DNA sequences; however, it is applicable to any series of characters. This function returns a single integer, which is the score between the two words.

Definition at line 44 of file align.c.

References aaOrder, and mat.

Referenced by alignWordsMat_bit().

```
45 {
46
     int i;
47
     int points = 0;
48
     int x, y;
49
       // Go over each character in the L-length window
50
51
       for (i = 0; i < L; i++)
52
53
54
       // The integer corresponding to the character in
55
       // the first string, so that we can look it up
56
       // in one of our scoring matricies.
57
       x = aaOrder[(int) s1[i]];
58
59
       // And for the second character
60
       y = aaOrder[(int) s2[i]];
61
62
       // If the characters aren't going to be in the scoring
63
       // matrix, they get a -1 value...which we'll give zero
       // points to here.
64
       if (x != -1 \&\& y != -1)
65
66
67
68
           // Otherwise, they get a score that is looked up
69
           // in the scoring matrix
70
           points += mat[x][y];
71
72
```

```
73 return points;
74 }
```

3.1.3.2 bitGraph_t* alignWordsMat_bit (sPat_t * words, int wc, int mat[][MATRIX_SIZE], int threshold)

This uses the function above. Here, we have an array of words (sPat_t objects) and we compare (align) them all. If their score is above 'threshold' then we will set a bit to 'true' in a bitGraph_t that we create. A bitGraph_t is essentially an adjacency matrix, where each member of the matrix contains only a single bit: are the words equal, true or false? The function traverses the words by doing and all by all comparison; however, we only do the upper diagonal. The function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

References alignMat(), bitGraphSetTrueSym(), mat, and newBitGraph().

Referenced by main().

```
90 {
     bitGraph_t * sg = NULL;
91
92
     int score;
93
     int i, j;
94
95
       // Assign a new bitGraph_t object, with (wc x wc) possible
96
       // true/false values
       sg = newBitGraph (wc);
98
     for (i = 0; i < wc; i++)
99
100
          for (j = i; j < wc; j++)
101
102
            // Get the score for the alignment of word {\tt i} and word {\tt j}
103
104
            score =
105
            alignMat (words[i].string, words[j].string, words[i].length, mat);
106
107
            // If that score is greater than threshold, set
            // a bit to 'true' in our bitGraph_t object
108
109
            if (score >= threshold)
110
111
112
            // We use 'bitGraphSetTrueSym' because, if i=j,
            // then j=i for most applications. However, this
113
            // can be relaxed for masochists.
114
115
            bitGraphSetTrueSym (sg, i, j);
116
117
118
119
120
        // Return a pointer to this new bitGraph_t object
```

```
121 return sg;
122 }
```

3.1.4 Variable Documentation

3.1.4.1 const int aaOrder[]

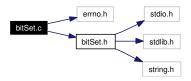
Definition at line 32 of file matrices.h.

Referenced by alignMat().

3.2 bitSet.c File Reference

```
#include "errno.h"
#include "bitSet.h"
```

Include dependency graph for bitSet.c:



Functions

- bit_t * newBitArray (int bytes)
- bitSet_t * newBitSet (int size)
- int setTrue (bitSet_t *s1, int x)
- int setFalse (bitSet_t *s1, int x)
- int flipBits (bitSet_t *s1)
- int fillSet (bitSet_t *s1)
- int emptySet (bitSet_t *s1)
- int checkBit (bitSet_t *s1, int x)
- int deleteBitSet (bitSet_t *s1)
- int bitSetUnion (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int copySet (bitSet_t *s1, bitSet_t *s2)
- int copyBitGraph (bitGraph_t *bg1, bitGraph_t *bg2)
- int bitSetDifference (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSetSum (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSetIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSet3WayIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3, bitSet_t *s4)
- int bitcount32 (unsigned int n)
- int bitcount32 precomp (unsigned int n)
- int bitcount64 (unsigned int n)
- int countSet (bitSet_t *s1)
- int nextBitBitSet (bitSet_t *s1, int start)
- int countBitGraphNonZero (bitGraph_t *bg)
- int printBitSet (bitSet_t *s1)
- int bitGraphRowUnion (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int bitGraphRowIntersection (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int printBinaryBitSet (bitSet_t *s1)
- int bitGraphCheckBit (bitGraph_t *bg, int x, int y)

```
• int bitGraphSetTrue (bitGraph_t *bg, int x, int y)
```

- int bitGraphSetFalse (bitGraph_t *bg, int x, int y)
- int bitGraphSetFalseSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueDiagonal (bitGraph_t *bg)
- int bitGraphSetFalseDiagonal (bitGraph_t *bg)
- int printBitGraph (bitGraph_t *bg)
- int maskBitGraph (bitGraph_t *bg1, bitSet_t *bs)
- int fillBitGraph (bitGraph_t *bg1)
- int emptyBitGraph (bitGraph_t *bg1)
- bitGraph_t * newBitGraph (int size)
- int emptyBitGraphRow (bitGraph_t *bg, int row)
- int deleteBitGraph (bitGraph_t *bg)

3.2.1 Detailed Description

This file defines functions for handling bit sets and bit graphs.

Definition in file bitSet.c.

3.2.2 Function Documentation

3.2.2.1 int bitcount 32 (unsigned int n)

Attempt at a fast way of counting how many true values are in a given bitSet_t. Currently deprecated, using precompiled version instead.

Definition at line 351 of file bitSet.c.

```
352 {
353
354
         works for 32-bit numbers only
355
356
357
        fix last line for 64-bit numbers
358
359
360
      register unsigned int tmp;
361
362
     tmp = n - ((n >> 1) & 03333333333) - ((n >> 2) & 0111111111111);
363
      return ((tmp + (tmp >> 3)) & 030707070707) % 63;
364 }
```

3.2.2.2 int bitcount 32_precomp (unsigned int n)

Uses bits_in_char data structure to determine the number of true bits in a 32-bit int in an efficient manner. Input: 32-bit int (equal to one slot in the bitSet). Output: number of true bits in the input integer.

Definition at line 396 of file bitSet.c.

Referenced by countSet().

3.2.2.3 int bitcount64 (unsigned int n)

Currently there is no support for 64-bit architectures.

Definition at line 420 of file bitSet.c.

3.2.2.4 int bitGraphCheckBit (bitGraph_t * bg, int x, int y)

Checks the value of a bit in a bitGraph_t object. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit to be checked, the index of the bit in that row that is to be checked. Output: the value of the bit in the bitGraph being checked.

Definition at line 628 of file bitSet.c.

References checkBit(), and bitGraph_t::graph.

Referenced by main(), and measureDiagonal().

```
629 {
630  return checkBit (bg->graph[x], y);
631 }
```

3.2.2.5 int bitGraphRowIntersection (bitGraph_t * bg, int row1, int row2, bitSet_t * s1)

Finds the intersection of two rows (bitSets) within a bitGraph_t object. Input: a bit-Graph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the intersection results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever both source bitSets had a true value).

Definition at line 598 of file bitSet.c.

References bitSetIntersection(), and bitGraph_t::graph.

Referenced by getStatMat(), and oldGetStatMat().

```
599 {
600 bitSetIntersection (bg->graph[row1], bg->graph[row2], s1);
601 return 0;
602 }
```

3.2.2.6 int bitGraphRowUnion (bitGraph_t * bg, int row1, int row2, bitSet_t * s1)

Finds the union of two rows (bitSets) within a bitGraph Input: a bitGraph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the union results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever one or both source bitSets had a true value).

Definition at line 584 of file bitSet.c.

References bitSetUnion(), and bitGraph t::graph.

```
585 {
586  bitSetUnion (bg->graph[row1], bg->graph[row2], s1);
587  return 0;
588 }
```

3.2.2.7 int bitGraphSetFalse (bitGraph_t * bg, int x, int y)

Sets a specific bit in a bitGraph false. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 654 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

```
655 {
656   setFalse (bg->graph[x], y);
657   return 0;
658 }
```

3.2.2.8 int bitGraphSetFalseDiagonal (bitGraph_t * bg)

Sets the main diagonal of a bitGraph false. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 714 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

Referenced by convolve().

```
715 {
716    int i;
717    for (i = 0; i < bg->size; i++)
718    {
719        setFalse (bg->graph[i], i);
720    }
721    return 0;
722 }
```

3.2.2.9 int bitGraphSetFalseSym (bitGraph_t * bg, int x, int y)

Sets a specific bit and its symmetric opposite in a bitGraph false. For instance, given that we wanted to set the 3rd bit in the 5th row false, this would also set the 5th bit in the 3rd row. Input: a bitGraph_t object, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 669 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

```
670 {
671    setFalse (bg->graph[x], y);
672    setFalse (bg->graph[y], x);
673    return 0;
674 }
```

3.2.2.10 int bitGraphSetTrue (bitGraph_t * bg, int x, int y)

Sets a specific bit in a bitGraph true. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 641 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

```
642 {
643    setTrue (bg->graph[x], y);
644    return 0;
645 }
```

3.2.2.11 int bitGraphSetTrueDiagonal (bitGraph_t * bg)

Sets the main diagonal of a bitGraph true. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 698 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

```
699 {
700    int i;
701    for (i = 0; i < bg->size; i++)
702    {
703         setTrue (bg->graph[i], i);
704    }
705    return 0;
706 }
```

3.2.2.12 int bitGraphSetTrueSym (bitGraph_t * bg, int x, int y)

Sets a specific bit and its symmetric opposite in a bitGraph true. For instance, given that we wanted to set the 3rd bit in the 5th row true, this would also set the 5th bit in the 3rd row. Input: a bitGraph, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 685 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

Referenced by alignWordsMat_bit(), main(), and realComparison().

```
686 {
687    setTrue (bg->graph[x], y);
688    setTrue (bg->graph[y], x);
689    return 0;
690 }
```

3.2.2.13 int bitSet3WayIntersection (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3, bitSet_t * s4)

Finds the intersection of 3 bitSets. Input: First bitSet to be intersected, second bitset to be intersected. third bitSet to be intersected, a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object with a true where all three source bitSets had a true.)

Definition at line 327 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

```
329 {
330
      int i;
      if ((s1->slots != s2->slots) || (s1->slots != s3->slots)
331
          || (s1->slots != s4->slots))
333
          fprintf (stderr, "Sets aren't same size!\n");
334
335
          fflush (stderr);
336
          exit (0);
337
        }
338
      for (i = 0; i < s1->slots; i++)
339
          s4->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
340
341
          s4->tf[i] = BSINTERSECTION (s3->tf[i], s4->tf[i]);
342
343
     return 0;
344 }
```

3.2.2.14 int bitSetDifference (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Locates all differences between two bitSets. The result bitSet contains a true at a given bit if the two source bitSets differ at that bit. Input: first bit set to be compared, second bit set to be compared, third bit set to store the results Output: integer success value of 0 (and an altered destination bitSet_t object with a true where the two source bit sets differed).

Definition at line 254 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

```
255 {
256
      int i;
257
      if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
258
          fprintf (stderr, "Sets aren't same size!\n");
259
260
          fflush (stderr);
261
          exit (0);
262
263
      for (i = 0; i < s1->slots; i++)
264
        {
```

```
265 s3->tf[i] = (s1->tf[i] & (~s2->tf[i]));
266 }
267 return 0;
268 }
```

3.2.2.15 int bitSetIntersection (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Finds the intersection of two bitsets. Input: First bitSet to be intersected, second bitSet to be intersected. a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object. with a true where both source bitSets had a true).

Definition at line 299 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowIntersection(), findCliques(), and maskBitGraph().

```
300 {
      int i;
301
302
     if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
303
          fprintf (stderr, "Sets aren't same size!\n");
304
305
          fprintf (stderr, "set 1 slots = %d\n", s1->slots);
         fprintf (stderr, "set 2 slots = %d\n", s2->slots);
306
307
          fprintf (stderr, "set 3 slots = %d\n", s3->slots);
308
          fflush (stderr);
309
          exit (0);
310
311
     for (i = 0; i < s1->slots; i++)
312
313
          s3->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
314
315
     return 0;
316 }
```

3.2.2.16 int bitSetSum (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Adds two bitSet_t objects together. Currently unknown functionality, not used in existing code.

Definition at line 275 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

```
276 {
277   int i;
278   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
279   {
280     fprintf (stderr, "Sets aren't same size!\n");
```

3.2.2.17 int bitSetUnion (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Finds the union of two bitSets Input: first bit set for the union, second bit set for the union. a bit set in which to store the results Output: an integer success value of 0 (and an altered third bitSet_t with the results of the union.

Definition at line 182 of file bitSet.c.

References BSUNION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowUnion(), and singleLinkage().

```
183 {
184
      int i;
185
      if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
186
187
          fprintf (stderr, "Sets aren't same size!\n");
          fflush (stderr);
188
189
          exit (0);
190
191
      for (i = 0; i < s1->slots; i++)
192
193
          s3->tf[i] = BSUNION (s1->tf[i], s2->tf[i]);
194
        }
195
      return 0;
196 }
```

3.2.2.18 int checkBit (bitSet_t * s1, int x)

Finds the value of a specific bit in a bitSet. Input: a bitSet, the number of the bit being queried. Output: the value of the bit being queried (1 or 0).

Definition at line 148 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

Referenced by bitGraphCheckBit(), findCliques(), getStatMat(), maskBitGraph(), nextBitBitSet(), singleLinkage(), and wholeRoundConv().

```
149 {
150    return BSTEST (s1->tf, x);
151 }
```

3.2.2.19 int copyBitGraph (bitGraph_t * bg1, bitGraph_t * bg2)

Copies the true/false contents of one bit graph into an existing bit graph. Both bit graphs must be the same size, and each corresponding bit set between the two bit graphs must be the same size. Input: source bit graph, destination bitGraph_t object. Output: integer success value of 0 (and an altered destination bit graph).

Definition at line 229 of file bitSet.c.

References copySet(), bitGraph_t::graph, and bitGraph_t::size.

```
230 {
231
      int i;
232
      if (bgl->size != bg2->size)
233
          fprintf (stderr, "Graphs are not the same size!");
235
          fflush (stderr);
236
          exit (0);
237
238
      for (i = 0; i < bgl->size; i++)
239
240
          copySet (bg1->graph[i], bg2->graph[i]);
241
242
      return 0;
243 }
```

3.2.2.20 int copySet (bitSet_t * s1, bitSet_t * s2)

Copies the true/false contents of one bit set into an existing bit set. Both bit sets must be the same size. Input: source bit set, destination bitSet_t object. Output: integer success value of 0 (and an altered destination bitset.

Definition at line 205 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

Referenced by copyBitGraph(), filterGraph(), and singleLinkage().

```
206 {
207
      int i;
208
      if (s1->slots != s2->slots)
209
210
          fprintf (stderr, "Sets are not the same size!");
211
          fflush (stderr);
212
          exit (0);
213
214
      for (i = 0; i < s1->slots; i++)
215
       {
216
          s2->tf[i] = s1->tf[i];
217
218
      return 0;
219 }
```

3.2.2.21 int countBitGraphNonZero (bitGraph_t *bg)

Counts the number of true (non-zero) values in a bitGraph_t object. Input: a bitGraph_t object. Output: the integer number of true (non-zero) values in the bitGraph_t object.

Definition at line 537 of file bitSet.c.

References countSet(), and bitGraph_t::graph.

```
538 {
539
     int i;
540
     int sum = 0;
541
     // Iterate over all bitSets in the bitGraph
542
     for (i = 0; i < bg->size; i++)
543
544
          sum += countSet (bg->graph[i]);
545
546
     return sum;
547 }
```

3.2.2.22 int countSet (bitSet_t * s1)

Counts the number of true values in a bitSet. Input: a bitSet_t object. Output: number of true values in that bitSet_t object.

Definition at line 437 of file bitSet.c.

References bitcount32_precomp(), and bitSet_t::tf.

Referenced by bitSetToCSet(), countBitGraphNonZero(), filterGraph(), filterIter(), findCliques(), getStatMat(), oldGetStatMat(), printBitSet(), singleLinkage(), and wholeCliqueConv().

```
438 {
439
     int i;
440
     int sum = 0;
441
     int (*bitCounter) () = &bitcount32_precomp;
442
      // Currently there is no support for 64-bit architectures.
443
444
      if (sizeof (bit_t) * 8 != 32)
445
       {
446
          fprintf (stderr,
447
               "\nSorry, no support for 64-bit architectures just yet! - countSet\n");
448
          fflush (stderr);
449
          exit (0);
450
451
      // Just count the number of true bits in each char, and do this for
452
      // (num of chars per int) chars.
453
454
      for (i = 0; i < s1->slots; i++)
455
456
          sum += bitCounter (s1->tf[i]);
457
```

```
458 return sum;
459 }
```

3.2.2.23 int deleteBitGraph (bitGraph_t * bg)

Deletes a bitGraph_t object from memory. Input: a bitGraph_t object to be deleted. Output: integer success value from 0 (and deletion of a bitGraph_t object).

Definition at line 853 of file bitSet.c.

References deleteBitSet(), and bitGraph_t::graph.

Referenced by main().

```
854 {
855
      int i;
      if (bg != NULL)
857
       {
858
          if (bg->graph != NULL)
859
          for (i = 0; i < bg->size; i++)
860
861
            {
862
              deleteBitSet (bg->graph[i]);
863
864
          free (bg->graph);
          bg->graph = NULL;
865
866
867
          free (bg);
868
          bg = NULL;
869
870
      return 0;
871 }
```

3.2.2.24 int deleteBitSet (bitSet_t * s1)

Performs memory management for the deletion of a bitSet_t structure. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 159 of file bitSet.c.

References bitSet t::tf.

Referenced by convolve(), deleteBitGraph(), filterGraph(), findCliques(), getStatMat(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

```
166 if (s1 != NULL)

167 {

168 free (s1);

169 s1 = NULL;

170 }

171 return 0;

172 }
```

3.2.2.25 int emptyBitGraph (bitGraph_t * bg1)

Sets all bits in the bitGraph_t object to false. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t with all false bits).

Definition at line 791 of file bitSet.c.

References emptySet(), and bitGraph_t::graph.

```
792 {
793    int i;
794    for (i = 0; i < bgl->size; i++)
795      {
796         emptySet (bgl->graph[i]);
797    }
798    return 0;
799 }
```

3.2.2.26 int emptyBitGraphRow (bitGraph_t * bg, int row)

Sets all bits in a bitGraph_t row (a bitSet_t object) false. Input: a bitGraph, a row in the bitGraph_t object to be emptied. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 841 of file bitSet.c.

References emptySet(), and bitGraph_t::graph.

```
842 {
843     emptySet (bg->graph[row]);
844     return 0;
845 }
```

3.2.2.27 int emptySet (bitSet_t * s1)

Sets all values in a bitSet to false. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 136 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

Referenced by emptyBitGraph(), emptyBitGraphRow(), filterGraph(), filterIter(), maskBitGraph(), pruneBitGraph(), and searchMemsWithList().

```
137 {
138   memset (s1->tf, 0, s1->bytes);
139   return 0;
140 }
```

3.2.2.28 int fillBitGraph (bitGraph_t *bg1)

Sets all bits in the bitGraph_t object to true. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t object with all true bits).

Definition at line 775 of file bitSet.c.

References fillSet(), and bitGraph t::graph.

```
776 {
777   int i;
778   for (i = 0; i < bgl->size; i++)
779      {
780       fillSet (bgl->graph[i]);
781     }
782   return 0;
783 }
```

3.2.2.29 int fillSet (bitSet_t * s1)

Sets all values in a bitSet to true. Input: a bitSet. Output: integer success value of 1.

Definition at line 124 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

Referenced by convolve(), fillBitGraph(), and wholeRoundConv().

```
125 {
126   memset (s1->tf, ~0, s1->bytes);
127   return 0;
128 }
```

3.2.2.30 int flipBits (bitSet_t * s1)

Inverts all values in a bitSet, making all trues false and all falses true. Input: a bitSet. Output: integer success value of 1.

Definition at line 108 of file bitSet.c.

References bitSet_t::tf.

3.2.2.31 int maskBitGraph (bitGraph_t * bg1, bitSet_t * bs)

Makes a bitGraph contain only true bits according to the bitmask given. Only locations with the row and column both true in the bitmask can be true if they were initially true. If they were false, they remain false. If the location does not have both the row and the column in the bitmask, it is made false. Note, this is not currently used in Gemoda. Input: a bitGraph, a mask in the form of a bitSet_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 752 of file bitSet.c.

References bitSetIntersection(), checkBit(), emptySet(), and bitGraph_t::graph.

```
753 {
754
      int i;
755
      for (i = 0; i < bg1->size; i++)
756
757
          if (checkBit (bs, i))
758
759
          bitSetIntersection (bgl->graph[i], bs, bgl->graph[i]);
760
        }
761
          else
762
        {
763
          emptySet (bg1->graph[i]);
764
765
766
      return 0;
767 }
```

3.2.2.32 bit_t* newBitArray (int bytes)

Creates a bit array for use in high-throughput intersections/unions. Input: desired size of bit array in byte. Output: a new bit array in bit_t forma. Note: this should not be called directly; see newBitSet.

Definition at line 20 of file bitSet.c.

Referenced by newBitSet().

```
21 {
22
    bit_t *b = (bit_t *) malloc (bytes);
23
     if (b == NULL)
24
25
         fprintf (stderr, "\nMemory error --- couldn't allocate bitArray!"
26
             " - newBitArray\n%s\n", strerror (errno));
27
         fflush (stderr);
28
         exit (0);
29
30
    // Set them all false
31
    memset (b, 0, bytes);
32
    return b;
33 }
```

3.2.2.33 bitGraph_t* newBitGraph (int size)

Creates a bitGraph_t data structure. Input: the size of the (square) bitGraph_t object. Output: a new bitGraph_t data structure.

Definition at line 807 of file bitSet.c.

References bitGraph_t::graph, newBitSet(), and bitGraph_t::size.

Referenced by alignWordsMat_bit(), main(), and realComparison().

```
808 {
809
     bitGraph_t *bg = NULL;
     int i;
810
     bg = (bitGraph_t *) malloc (sizeof (bitGraph_t));
     if (bg == NULL)
812
813
814
          fprintf (stderr, "Memory error - Cannot allocate bitGraph - "
815
               "newBitGraph\n%s\n", strerror (errno));
816
          fflush (stderr);
817
         exit (0);
818
       }
819
     bg->size = size;
820
     bg->graph = (bitSet_t **) malloc (size * sizeof (bitSet_t *));
821
     if (bg->graph == NULL)
822
823
          fprintf (stderr, "Memory error - Cannot allocate bitGraphGraph - "
               "newBitGraph\n%s\n", strerror (errno));
824
825
          fflush (stderr);
826
          exit (0);
827
828
      for (i = 0; i < size; i++)
829
         bg->graph[i] = newBitSet (size);
831
832
     return bg;
833 }
```

3.2.2.34 bitSet_t* newBitSet (int size)

Creates a bitSet data structure that contains a bit array and information about that bit array that is necessary for quick and efficient access of the array. Input: the desired length of the bit array. Output: a bitSet data structure.

Definition at line 43 of file bitSet.c.

References BSNUMSLOTS, bitSet_t::bytes, bitSet_t::max, newBitArray(), bitSet_t::slots, and bitSet_t::tf.

Referenced by convolve(), filterGraph(), findCliques(), getStatMat(), newBitGraph(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

```
44 {
45
     bitSet_t *s1 = (bitSet_t *) malloc (sizeof (bitSet_t));
46
     if (s1 == NULL)
47
48
         fprintf (stderr, "\nMemory error --- couldn't allocate biSet!"
49
              " - newBitSet\n%s\n", strerror (errno));
50
         fflush (stderr);
51
         exit (0);
52
53
    // Fill in details about the bitSet, allocate bitSet
54
     s1->max = size;
55
     s1->slots = BSNUMSLOTS (size);
     s1->bytes = s1->slots * sizeof (bit_t);
    s1->tf = newBitArray (s1->bytes);
57
    return s1;
59 }
```

3.2.2.35 int nextBitBitSet (bitSet_t * s1, int start)

Finds the index of the first non-zero bit at-or-after start. Input: a bitSet_t to be searched, the index of the start bit. Output: the index of the first non-zero bit at-or-after start.

Definition at line 468 of file bitSet.c.

References BITSLOT, BSBITSIZE, checkBit(), bitSet_t::max, and bitSet_t::tf.

Referenced by bitSetToCSet(), filterIter(), findCliques(), getStatMat(), pruneBitGraph(), and singleLinkage().

```
469 {
470
     // slot is our starting slot, the
     // slot containing bit 'start
472
     int slot = BITSLOT (start);
473
     int i;
474
     // stop is the bit to stop it --- it is equal to max, and it is
     // the index of a bit that does NOT belong to the bitset
475
476
     int stop;
477
    bit_t bitFalse;
```

```
478
      memset (&bitFalse, 0, sizeof (bit_t));
479
480
481
     // sl->max is the number of bits in sl
     // test to see if we're looking too high
482
483
     if (start >= s1->max)
484
      {
485
         return -1;
486
487
     // s1->slots is the number of available slots
     // skip over empty slots
488
489
      while (slot < s1->slots)
490
491
492
            printf("w");
493
494
         if (s1->tf[slot] != bitFalse)
495
          // this slot is not empty
496
497
         // if each slot is, say 32 bits and
498
499
         // we asked for nextBitBitSet(s1, 5),
500
         // then slot 0 will be non-zero. but,
501
         // instead of starting at 0, start at 5!
502
         if (BSBITSIZE * slot > start)
503
          {
504
              // set start to index of first
505
              // bit in this slot
             start = BSBITSIZE * slot;
506
507
508
         // set the stop, with a a check against the 'max'
509
          // element of the bitSet_t object
510
          if (BSBITSIZE * (slot + 1) > s1->max)
511
512
              stop = s1->max;
          }
513
514
         else
515
516
             stop = BSBITSIZE * (slot + 1);
517
518
          for (i = start; i < stop; i++)
519
520
             if (checkBit (s1, i))
521
              return i;
522
523
524
525
526
         slot++;
527
528
    return -1;
529 }
```

3.2.2.36 int printBinaryBitSet (bitSet_t * s1)

Prints a representation of a bitSet_t structure as a string of 1's and 0's. Input: a bitSet_t object to be printed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 611 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

Referenced by printBitGraph().

```
612 {
613   int i;
614   for (i = 0; i < s1->max; i++)
615     {
616      printf ("%d", (BSTEST (s1->tf, i) ? 1 : 0));
617     }
618   return 0;
619 }
```

3.2.2.37 int printBitGraph (bitGraph_t *bg)

Prints a representation of a bitGraph using printBinaryBitSet. Input: a bitGraph_t object. Output: integer success value of 0 (and stdout text as described above).

Definition at line 730 of file bitSet.c.

References bitGraph_t::graph, and printBinaryBitSet().

3.2.2.38 int printBitSet (bitSet_t * s1)

Prints a representation of a bitSet_t data structure. Input: a bitSet_t to be displayed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 555 of file bitSet.c.

References BSTEST, and countSet().

```
556 {
557
      int i;
    printf ("bitSet (addr = %d; %d members)\n", (int) s1, countSet (s1));
559
    printf ("tmax = %d\n", s1->max);
560
     printf ("\tslots = %d\n", s1->slots);
561
      printf ("\tbytes = %d\n", s1->bytes);
     printf ("\tmembers =");
562
563
564
565
      for (i = 0; i < s1->max; i++)
566
567
          if (BSTEST (s1->tf, i))
568
         printf (" %d", i);
569
570
571
     printf ("\n");
572
573
     return 0;
574 }
```

3.2.2.39 int setFalse (bitSet_t * s1, int x)

Sets a specific bit in a bitSet as false. Input: a bitSet, the number of the bit to be set as false. Output: integer success value of 1.

Definition at line 85 of file bitSet.c.

References BSCLEAR, bitSet_t::max, and bitSet_t::tf.

Referenced by bitGraphSetFalse(), bitGraphSetFalseDiagonal(), bitGraphSetFalseSym(), filterIter(), findCliques(), singleCliqueConv(), and singleLinkage().

```
86 {
87
88
        if (BSNUMSLOTS(x) > s1->slots) { Conditional changed, 5/25, by MPS: check x against s1->max,
89
       should be safer
90
    if (x >= s1->max)
91
92
93
         fprintf (stderr, "Set isn't large enough! - setFalse\n");
94
         fflush (stderr);
95
         exit (0);
96
97
    BSCLEAR (s1->tf, x);
98
    return 0;
99 }
```

3.2.2.40 int setTrue (bitSet_t * s1, int x)

Sets a specific bit in a bitSet as true. Input: a bitSet, the number of the bit to be set as true. Output: integer success value of 1.

Definition at line 67 of file bitSet.c.

References BSSET, bitSet_t::max, and bitSet_t::tf.

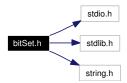
Referenced by bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), filterIter(), findCliques(), and setStackTrue().

```
68 {
69    if (x >= s1->max)
70     {
71         fprintf (stderr, "Set isn't large enough! - setTrue\n");
72         fflush (stderr);
73         exit (0);
74     }
75    BSSET (s1->tf, x);
76    return 0;
77 }
```

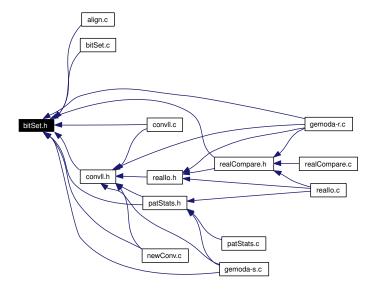
3.3 bitSet.h File Reference

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
```

Include dependency graph for bitSet.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct bitSet_t
- struct bitGraph_t

Defines

- #define BSBITSIZE (sizeof(bit_t) * 8)
 Get the size of a bit_t, which is an unsigned int.
- #define BSMASK(y) (((bit_t) 1) << y % BSBITSIZE)
- #define BITSLOT(y) (y / BSBITSIZE)
- #define BSSET(x, y) (x[BITSLOT(y)] = BSMASK(y))

Sets the y'th bit in x (a bitset_t) to be true using bitwise operators.

- #define BSCLEAR(x, y) (x[BITSLOT(y)] &= ~BSMASK(y))

 Sets the y'th bit in x (a bitset_t) to be false using bitwise operators.
- #define BSTEST(x, y) (x[BITSLOT(y)] & BSMASK(y))

 Tests whether the y'th bit in x (a bitset_t) is true.
- #define BSNUMSLOTS(n) ((n + BSBITSIZE 1) / BSBITSIZE)
- #define BSUNION(x, y) ((x)|(y))

Performs a union operation on two bit_t's with bitwise operators.

• #define BSINTERSECTION(x, y) ((x)&(y))

Performs an intersection operation on two bit_t's with bitwise operators.

Typedefs

• typedef unsigned int bit_t

Functions

- bit_t * newBitArray (int bytes)
- bitSet_t * newBitSet (int size)
- int setTrue (bitSet_t *s1, int x)
- int setFalse (bitSet_t *s1, int x)
- int bitSetDifference (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSet3WayDifference (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3, bitSet_t *s4)
- int bitSetSum (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int flipBits (bitSet_t *s1)
- int fillSet (bitSet_t *s1)
- int emptySet (bitSet_t *s1)
- int checkBit (bitSet_t *s1, int x)
- int copyBitGraph (bitGraph_t *bg1, bitGraph_t *bg2)

- int copySet (bitSet_t *s1, bitSet_t *s2)
- int deleteBitSet (bitSet_t *s1)
- int bitSetUnion (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSetIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSet3WayIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3, bitSet_t *s4)
- int countSet (bitSet t*s1)
- int fillBitGraph (bitGraph_t *bg1)
- int emptyBitGraph (bitGraph_t *bg1)
- int printBitSet (bitSet_t *s1)
- int nextBitBitSet (bitSet_t *s1, int start)
- int countBitGraphNonZero (bitGraph_t *bg)
- int printBinaryBitSet (bitSet t*s1)
- int bitGraphSetTrue (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueDiagonal (bitGraph_t *bg)
- int bitGraphSetFalseDiagonal (bitGraph_t *bg)
- int printBitGraph (bitGraph_t *bg)
- bitGraph_t * newBitGraph (int size)
- int deleteBitGraph (bitGraph_t *bg)
- int bitGraphRowUnion (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int bitGraphRowIntersection (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int bitGraphCheckBit (bitGraph_t *bg, int x, int y)
- int bitGraphSetFalseSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetFalse (bitGraph_t *bg, int x, int y)
- int emptyBitGraphRow (bitGraph_t *bg, int row)
- int maskBitGraph (bitGraph_t *bg1, bitSet_t *bs)

3.3.1 Detailed Description

This file provides declarations and definitions for a bit set object. The functions declared here are defined in bitSet.c.

Definition in file bitSet.h.

3.3.2 Define Documentation

3.3.2.1 #define BITSLOT(y) (y/BSBITSIZE)

Finds which bit_t, or "slot", in a bitset_t the y'th bit belongs to. Also used for testing single bits.

Definition at line 71 of file bitSet.h.

Referenced by nextBitBitSet().

3.3.2.2 #define BSBITSIZE (sizeof(bit_t) * 8)

Definition at line 63 of file bitSet.h.

Referenced by nextBitBitSet().

3.3.2.3 #define BSCLEAR(x, y) (x[BITSLOT(y)] &= \sim BSMASK(y))

Definition at line 77 of file bitSet.h.

Referenced by setFalse().

3.3.2.4 #define BSINTERSECTION(x, y) ((x)&(y))

Definition at line 93 of file bitSet.h.

Referenced by bitSet3WayIntersection(), and bitSetIntersection().

3.3.2.5 #define BSMASK(y) (((bit_t) 1) << y % BSBITSIZE)

Uses bit operations to make a mask the size of a bit_t with the y'th bit true and all other bits false. Used for testing a single bit.

Definition at line 67 of file bitSet.h.

3.3.2.6 #define BSNUMSLOTS(n) ((n + BSBITSIZE - 1) / BSBITSIZE)

Finds the total number of bit_t's ("slot"s) that are necessary for a bitset of length n. Uses integer division and supplements by BSBITSIZE - 1 to make sure that for slots that are less than full, a slot is still allocated, and for slots that are full, no extra slot is allocated.

Definition at line 87 of file bitSet.h.

Referenced by newBitSet().

3.3.2.7 #define BSSET(x, y) (x[BITSLOT(y)] = BSMASK(y))

Definition at line 74 of file bitSet.h.

Referenced by setTrue().

3.3.2.8 #define BSTEST(x, y) (x[BITSLOT(y)] & BSMASK(y))

Definition at line 80 of file bitSet.h.

Referenced by checkBit(), printBinaryBitSet(), and printBitSet().

3.3.2.9 #define BSUNION(x, y) ((x)|(y))

Definition at line 90 of file bitSet.h.

Referenced by bitSetUnion().

3.3.3 Typedef Documentation

3.3.3.1 typedef unsigned int bit_t

a bit_t is the size of an unsigned integer on the current architecture.

Definition at line 15 of file bitSet.h.

3.3.4 Function Documentation

3.3.4.1 int bitGraphCheckBit (bitGraph_t * bg, int x, int y)

Checks the value of a bit in a bitGraph_t object. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit to be checked, the index of the bit in that row that is to be checked. Output: the value of the bit in the bitGraph being checked.

Definition at line 599 of file bitSet.c.

References checkBit(), and bitGraph_t::graph.

Referenced by main(), and measureDiagonal().

3.3.4.2 int bitGraphRowIntersection (bitGraph_t * bg, int row1, int row2, bitSet_t * s1)

Finds the intersection of two rows (bitSets) within a bitGraph_t object. Input: a bitGraph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the intersection results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever both source bitSets had a true value).

Definition at line 572 of file bitSet.c.

References bitSetIntersection(), and bitGraph_t::graph.

Referenced by getStatMat(), and oldGetStatMat().

3.3.4.3 int bitGraphRowUnion (bitGraph_t * bg, int row1, int row2, bitSet_t * s1)

Finds the union of two rows (bitSets) within a bitGraph Input: a bitGraph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the union results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever one or both source bitSets had a true value).

Definition at line 560 of file bitSet.c.

References bitSetUnion(), and bitGraph_t::graph.

3.3.4.4 int bitGraphSetFalse (bitGraph_t * bg, int x, int y)

Sets a specific bit in a bitGraph false. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 623 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

3.3.4.5 int bitGraphSetFalseDiagonal (bitGraph_t * bg)

Sets the main diagonal of a bitGraph false. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 678 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

Referenced by convolve().

3.3.4.6 int bitGraphSetFalseSym (bitGraph_t * bg, int x, int y)

Sets a specific bit and its symmetric opposite in a bitGraph false. For instance, given that we wanted to set the 3rd bit in the 5th row false, this would also set the 5th bit in the 3rd row. Input: a bitGraph_t object, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 637 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

3.3.4.7 int bitGraphSetTrue (bitGraph_t * bg, int x, int y)

Sets a specific bit in a bitGraph true. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 611 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

3.3.4.8 int bitGraphSetTrueDiagonal (bitGraph_t * bg)

Sets the main diagonal of a bitGraph true. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 664 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

3.3.4.9 int bitGraphSetTrueSym (bitGraph_t * bg, int x, int y)

Sets a specific bit and its symmetric opposite in a bitGraph true. For instance, given that we wanted to set the 3rd bit in the 5th row true, this would also set the 5th bit in the 3rd row. Input: a bitGraph, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 652 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

Referenced by alignWordsMat_bit(), main(), and realComparison().

```
3.3.4.10 int bitSet3WayDifference (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3, bitSet_t * s4)
```

```
3.3.4.11 int bitSet3WayIntersection (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3, bitSet_t * s4)
```

Finds the intersection of 3 bitSets. Input: First bitSet to be intersected, second bitset to be intersected. third bitSet to be intersected, a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object with a true where all three source bitSets had a true.)

Definition at line 304 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

3.3.4.12 int bitSetDifference (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Locates all differences between two bitSets. The result bitSet contains a true at a given bit if the two source bitSets differ at that bit. Input: first bit set to be compared, second bit set to be compared, third bit set to store the results Output: integer success value of 0 (and an altered destination bitSet_t object with a true where the two source bit sets differed).

Definition at line 237 of file bitSet.c.

References bitSet t::slots, and bitSet t::tf.

3.3.4.13 int bitSetIntersection (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Finds the intersection of two bitsets. Input: First bitSet to be intersected, second bitSet to be intersected. a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object. with a true where both source bitSets had a true).

Definition at line 278 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowIntersection(), findCliques(), and maskBitGraph().

3.3.4.14 int bitSetSum (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Adds two bitSet_t objects together. Currently unknown functionality, not used in existing code.

Definition at line 256 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

3.3.4.15 int bitSetUnion (bitSet_t * s1, bitSet_t * s2, bitSet_t * s3)

Finds the union of two bitSets Input: first bit set for the union, second bit set for the union. a bit set in which to store the results Output: an integer success value of 0 (and an altered third bitSet_t with the results of the union.

Definition at line 175 of file bitSet.c.

References BSUNION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowUnion(), and singleLinkage().

3.3.4.16 int checkBit (bitSet_t * s1, int x)

Finds the value of a specific bit in a bitSet. Input: a bitSet, the number of the bit being queried. Output: the value of the bit being queried (1 or 0).

Definition at line 143 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

Referenced by bitGraphCheckBit(), findCliques(), getStatMat(), maskBitGraph(), nextBitBitSet(), singleLinkage(), and wholeRoundConv().

3.3.4.17 int copyBitGraph (bitGraph_t *bg1, bitGraph_t *bg2)

Copies the true/false contents of one bit graph into an existing bit graph. Both bit graphs must be the same size, and each corresponding bit set between the two bit graphs must be the same size. Input: source bit graph, destination bitGraph_t object. Output: integer success value of 0 (and an altered destination bit graph).

Definition at line 215 of file bitSet.c.

References copySet(), bitGraph_t::graph, and bitGraph_t::size.

3.3.4.18 int copySet (bitSet_t * s1, bitSet_t * s2)

Copies the true/false contents of one bit set into an existing bit set. Both bit sets must be the same size. Input: source bit set, destination bitSet_t object. Output: integer success value of 0 (and an altered destination bitset.

Definition at line 195 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

Referenced by copyBitGraph(), filterGraph(), and singleLinkage().

3.3.4.19 int countBitGraphNonZero (bitGraph_t * bg)

Counts the number of true (non-zero) values in a bitGraph_t object. Input: a bitGraph_t object. Output: the integer number of true (non-zero) values in the bitGraph_t object.

Definition at line 511 of file bitSet.c.

References countSet(), and bitGraph_t::graph.

3.3.4.20 int countSet (bitSet_t * s1)

Counts the number of true values in a bitSet. Input: a bitSet_t object. Output: number of true values in that bitSet_t object.

Definition at line 413 of file bitSet.c.

References bitcount32_precomp(), and bitSet_t::tf.

Referenced by bitSetToCSet(), countBitGraphNonZero(), filterGraph(), filterIter(), findCliques(), getStatMat(), oldGetStatMat(), printBitSet(), singleLinkage(), and wholeCliqueConv().

3.3.4.21 int deleteBitGraph (bitGraph_t * bg)

Deletes a bitGraph_t object from memory. Input: a bitGraph_t object to be deleted. Output: integer success value from 0 (and deletion of a bitGraph_t object).

Definition at line 799 of file bitSet.c.

References deleteBitSet(), and bitGraph_t::graph.

Referenced by main().

3.3.4.22 int deleteBitSet (bitSet_t * s1)

Performs memory management for the deletion of a bitSet_t structure. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 154 of file bitSet.c.

References bitSet_t::tf.

 $Referenced \ by \ convolve(), \ delete Bit Graph(), \ filter Graph(), \ find Cliques(), \ get Stat Mat(), \ old Get Stat Mat(), \ whole Clique Conv(), \ and \ whole Round Conv().$

3.3.4.23 int emptyBitGraph (bitGraph_t *bg1)

Sets all bits in the bitGraph_t object to false. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t with all false bits).

Definition at line 744 of file bitSet.c.

References emptySet(), and bitGraph t::graph.

3.3.4.24 int emptyBitGraphRow (bitGraph_t * bg, int row)

Sets all bits in a bitGraph_t row (a bitSet_t object) false. Input: a bitGraph, a row in the bitGraph_t object to be emptied. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 788 of file bitSet.c.

References emptySet(), and bitGraph_t::graph.

3.3.4.25 int emptySet (bitSet_t * s1)

Sets all values in a bitSet to false. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 131 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

 $Referenced\ by\ emptyBitGraph(),\ emptyBitGraphRow(),\ filterGraph(),\ filterIter(),\\ maskBitGraph(),\ pruneBitGraph(),\ and\ searchMemsWithList().$

3.3.4.26 int fillBitGraph (bitGraph_t *bg1)

Sets all bits in the bitGraph_t object to true. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t object with all true bits).

Definition at line 730 of file bitSet.c.

References fillSet(), and bitGraph_t::graph.

3.3.4.27 int fillSet (bitSet_t * s1)

Sets all values in a bitSet to true. Input: a bitSet. Output: integer success value of 1.

Definition at line 119 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

Referenced by convolve(), fillBitGraph(), and wholeRoundConv().

3.3.4.28 int flipBits (bitSet t * s1)

Inverts all values in a bitSet, making all trues false and all falses true. Input: a bitSet. Output: integer success value of 1.

Definition at line 105 of file bitSet.c.

References bitSet t::tf.

3.3.4.29 int maskBitGraph (bitGraph_t * bg1, bitSet_t * bs)

Makes a bitGraph contain only true bits according to the bitmask given. Only locations with the row and column both true in the bitmask can be true if they were initially true. If they were false, they remain false. If the location does not have both the row and the column in the bitmask, it is made false. Note, this is not currently used in Gemoda. Input: a bitGraph, a mask in the form of a bitSet_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 712 of file bitSet.c.

References bitSetIntersection(), checkBit(), emptySet(), and bitGraph_t::graph.

3.3.4.30 bit_t* newBitArray (int bytes)

Creates a bit array for use in high-throughput intersections/unions. Input: desired size of bit array in byte. Output: a new bit array in bit_t forma. Note: this should not be called directly; see newBitSet.

Definition at line 18 of file bitSet.c.

Referenced by newBitSet().

3.3.4.31 bitGraph_t* newBitGraph (int size)

Creates a bitGraph_t data structure. Input: the size of the (square) bitGraph_t object. Output: a new bitGraph_t data structure.

Definition at line 758 of file bitSet.c.

References bitGraph_t::graph, newBitSet(), and bitGraph_t::size.

Referenced by alignWordsMat_bit(), main(), and realComparison().

3.3.4.32 bitSet_t* newBitSet (int size)

Creates a bitSet data structure that contains a bit array and information about that bit array that is necessary for quick and efficient access of the array. Input: the desired length of the bit array. Output: a bitSet data structure.

Definition at line 40 of file bitSet.c.

References BSNUMSLOTS, bitSet_t::bytes, bitSet_t::max, newBitArray(), bitSet_t::slots, and bitSet_t::tf.

Referenced by convolve(), filterGraph(), findCliques(), getStatMat(), newBitGraph(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

3.3.4.33 int nextBitBitSet (bitSet_t * s1, int start)

Finds the index of the first non-zero bit at-or-after start. Input: a bitSet_t to be searched, the index of the start bit. Output: the index of the first non-zero bit at-or-after start.

Definition at line 452 of file bitSet.c.

References BITSLOT, BSBITSIZE, checkBit(), bitSet_t::max, bitSet_t::slots, and bit-Set_t::tf.

Referenced by bitSetToCSet(), filterIter(), findCliques(), getStatMat(), pruneBit-Graph(), and singleLinkage().

3.3.4.34 int printBinaryBitSet (bitSet_t * s1)

Prints a representation of a bitSet_t structure as a string of 1's and 0's. Input: a bitSet_t object to be printed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 584 of file bitSet.c.

References BSTEST, and bitSet t::tf.

Referenced by printBitGraph().

3.3.4.35 int printBitGraph (bitGraph_t *bg)

Prints a representation of a bitGraph using printBinaryBitSet. Input: a bitGraph_t object. Output: integer success value of 0 (and stdout text as described above).

Definition at line 692 of file bitSet.c.

References bitGraph_t::graph, and printBinaryBitSet().

3.3.4.36 int printBitSet (bitSet_t * s1)

Prints a representation of a bitSet_t data structure. Input: a bitSet_t to be displayed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 527 of file bitSet.c.

References BSTEST, and countSet().

3.3.4.37 int setFalse (bitSet_t * s1, int x)

Sets a specific bit in a bitSet as false. Input: a bitSet, the number of the bit to be set as false. Output: integer success value of 1.

Definition at line 84 of file bitSet.c.

References BSCLEAR, bitSet_t::max, and bitSet_t::tf.

Referenced by bitGraphSetFalse(), bitGraphSetFalseDiagonal(), bitGraphSetFalseSym(), filterIter(), findCliques(), singleCliqueConv(), and singleLinkage().

3.3.4.38 int setTrue (bitSet_t * s1, int x)

Sets a specific bit in a bitSet as true. Input: a bitSet, the number of the bit to be set as true. Output: integer success value of 1.

Definition at line 63 of file bitSet.c.

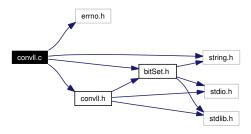
References BSSET, bitSet_t::max, and bitSet_t::tf.

Referenced by bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), filterIter(), findCliques(), and setStackTrue().

3.4 convll.c File Reference

```
#include <errno.h>
#include <string.h>
#include "convll.h"
#include "bitSet.h"
```

Include dependency graph for convll.c:



Functions

- cll_t * pruneCll (cll_t *head, int *indexToSeq, int p)
- cll_t * pushCll (cll_t *head)
- cll_t * popCll (cll_t *head)
- cll_t * popAllCll (cll_t *head)
- int printCll (cll_t *head)
- cll_t * initheadCll (cll_t *head, cSet_t *newset)
- cll_t * pushcSet (cll_t *head, cSet_t *newset)
- cSet_t * bitSetToCSet (bitSet_t *clique)
- int checkCliquecSet (cSet_t *cliquecSet, int *indexToSeq, int p)
- cll_t * pushClique (bitSet_t *clique, cll_t *head, int *indexToSeq, int p)
- mll_t * pushMemStack (mll_t *head, int cliqueNum)
- mll_t * popMemStack (mll_t *head)
- mll_t * popWholeMemStack (mll_t *head)
- mll_t ** addToStacks (cll_t *node, mll_t **memberStacks)
- mll_t ** fillMemberStacks (cll_t *head, mll_t **memberStacks)
- mll_t ** emptyMemberStacks (mll_t **memberStacks, int size)
- void printMemberStacks (mll_t **memberStacks, int size)
- bitSet_t * setStackTrue (mll_t **memList, int i, bitSet_t *queue)
- bitSet_t * searchMemsWithList (int *list, int listsize, mll_t **memList, int num-Offsets, bitSet_t *queue)

- cll_t * singleCliqueConv (cll_t *head, int firstClique, cll_t **firstGuess, int secondClique, cll_t **secondGuess, cll_t *nextPhase, bitSet_t *printStatus, int support)
- mll_t * mergeIntersect (cll_t *first, cll_t *second, mll_t *intersection, bitSet_t *printstatus, int *newSupport)
- int uniqClique (cSet_t *cliquecSet, cll_t *head)
- cll_t * swapNodecSet (cll_t *head, int node, cSet_t *newClique)
- cll_t * removeSupers (cll_t *head, int node, cSet_t *newClique)
- int printCSet (cSet_t *node)
- cll_t * pushConvClique (mll_t *clique, cll_t *head)
- cSet_t * mllToCSet (mll_t *clique)
- cll_t * wholeCliqueConv (cll_t *head, cll_t *node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t *nextPhase, bitSet_t *printStatus, int support)
- cll_t * wholeRoundConv (cll_t **head, mll_t **memList, int numOffsets, int support, int length, cll_t **allCliques)
- int yankCll (cll t **head, cll t *prev, cll t **curr, cll t **allCliques, int length)
- cll_t * completeConv (cll_t **head, int support, int numOffsets, int minLength, int *indexToSeq, int p)
- int printCllPattern (cll_t *node, int length)

Variables

• int cliquecounter = 0

3.4.1 Detailed Description

This file defines a number of functions for handling link lists of motifs, or cliques. The functions defined in this file are called extensively during the convolution stage of the Gemoda algorithm for both the sequence based and real value based software.

Definition in file convll.c.

3.4.2 Function Documentation

3.4.2.1 mll t** addToStacks (cll t * node, mll t ** memberStacks)

For one clique, it adds membership for that clique to all of its members' member stacks. Input: a specific clique in a clique linked list, an array of member stacks. Output: the array of updated member stacks.

Definition at line 482 of file convll.c.

References cnode::id, cSet_t::members, pushMemStack(), and cnode::set.

Referenced by fillMemberStacks().

```
483 {
      int i = 0;
484
485
      int cliqueNum = 0;
486
487
      // Make sure that we don't reference NULL values
488
      if (node->set != NULL)
489
490
          // Go through each member of the clique's set
491
          for (i = 0; i < node->set->size; i++)
492
493
          // Get the member's number
494
          cliqueNum = node->set->members[i];
495
          // Go to that member's linked list and push
496
          // on the number of the current clique
497
          memberStacks[cliqueNum] =
498
            pushMemStack (memberStacks[cliqueNum], node->id);
499
500
501
      else
502
          fprintf (stderr, "\nNULL set for clique! - addToStacks\n");
503
504
          fflush (stderr);
505
          exit (0);
506
507
      return memberStacks;
508 }
```

3.4.2.2 cSet_t* bitSetToCSet (bitSet_t * clique)

Converts a bitSet_t to a cSet_t for the purposes of pushing it onto a linked list of cliques. The bitSet_t data structure is used for massive comparisons during clique-finding but is unwieldy/inefficient when it is known that the structure is sparse. The cSet_t allows for efficient comparison of sparse bitSet_t's. Use this just before pushing a newly-discovered clique onto a clique linked list. Input: a new clique in the form of a bitSet_t. Output: the same clique in the form of a cSet_t.

Definition at line 212 of file convll.c.

References countSet(), cSet_t::members, nextBitBitSet(), and cSet_t::size.

Referenced by pushClique(), and wholeCliqueConv().

```
213 {
214
      int cliqueSize = countSet (clique);
      int i = 0, start = 0;
      cSet_t *holder = (cSet_t *) malloc (sizeof (cSet_t));
216
218
      // Memory error checking
219
      if (holder == NULL)
220
          fprintf (stderr, "\nMemory Error - bitSetToCSet - [1]\n%s\n",
221
222
               strerror (errno));
223
          fflush (stderr);
```

```
224
          exit (0);
225
        }
226
      // More memory checking
227
      holder->members = (int *) malloc (cliqueSize * sizeof (int));
228
      if (holder->members == NULL)
229
        {
          fprintf (stderr, "\nMemory Error - bitSetToCSet - [2]\n%s\n",
230
231
               strerror (errno));
232
          fflush (stderr);
233
          exit (0);
234
235
236
      // For each member of the clique in the bitSet,
237
      for (i = 0; i < cliqueSize; i++)</pre>
238
239
          // Find the next one, add its location to the members array
240
          holder->members[i] = nextBitBitSet (clique, start);
241
          // (But check for errors... if we get to the end of the
242
          // bitSet, then something is wrong)
243
          if (holder->members[i] == -1)
244
          fprintf (stderr, "\nClique error - not enough members\n");
245
246
          fflush (stderr);
247
          exit (0);
248
        }
          // Increment to move on in the nextBitBitSet search
249
250
          start = holder->members[i] + 1;
251
252
253
      holder->size = cliqueSize;
254
      return holder;
255 }
```

3.4.2.3 int checkCliquecSet (cSet_t * cliquecSet, int * indexToSeq, int p)

Checks to enforce the -p flag (minimum number of unique input sequences in which the motif occurs). Input: a clique in the form of a cSet_t, pointer to the index/sequence number data structure, the -p flag value. Output: An integer: 1 for success, 0 for failure.

Definition at line 266 of file convll.c.

References cSet_t::members, and cSet_t::size.

Referenced by pushClique().

```
276
          fflush (stderr);
277
          exit (0);
278
279
      // Initialize an array of integers of size p to sentinel values of -1
280
      for (i = 0; i < p; i++)
281
          seqNums[i] = -1;
282
283
284
      j = 0;
285
      if (cliquecSet->size < 1)</pre>
286
287
288
          fprintf (stderr, "\nClique of zero size! - checkCliquecSet\n");
289
          fflush (stderr);
290
          exit (0);
291
292
      // Find the first sequence number.
293
      seqNums[0] = indexToSeq[cliquecSet->members[0]];
      // Iterate over the remaining size of the clique
294
295
      for (i = 1; i < cliquecSet->size; i++)
296
297
          // Find the next sequence number.
298
          thisSeq = indexToSeq[cliquecSet->members[i]];
299
          // The member list is in monotonic order, so we only need
300
          \ensuremath{//} to compare the current member to the previous member to
          \ensuremath{//} find out if it comes from the same sequence.
301
302
          // If it's not from the same sequence, increment the unique
303
          // sequence counter (j), store the next sequence number
304
          // in the array.
305
          // Also check to see if we've already reached the p threshold,
306
          // and if so, then bail out.
307
          if (thisSeq != seqNums[j])
308
309
          j++;
310
          seqNums[j] = thisSeq;
311
          if (j == p - 1)
312
            {
313
              break;
314
315
316
317
      \ensuremath{//} Now just see what the value of the last number in the array is;
318
319
      // if it's the sentinel, then we didn't find instances in p
      // unique sequences. If it's not the sentinel, then we've met
320
321
      // the -p criterion.
322
      if (seqNums[p - 1] == -1)
323
       {
324
          free (seqNums);
325
          return (0);
326
327
      else
328
329
          free (seqNums);
330
          return (1);
331
332 }
```

3.4.2.4 cll_t* completeConv (cll_t ** head, int support, int numOffsets, int minLength, int * indexToSeq, int p)

Performs complete convolution given the starting list of cliques. Input: a pointer to the head of the initial clique linked list, the minimum support criterion value, the number of offsets in the sequence set, the minimum length of motifs (which is the length of motifs in the initial clique linked list), the index/Sequence data structure, and the value of the -p flag to prune based on unique sequence occurrences. Output: a linked list of all maximal cliques based on the initial clique linked list.

Definition at line 1417 of file convll.c.

References emptyMemberStacks(), fillMemberStacks(), popAllCll(), pruneCll(), and wholeRoundConv().

Referenced by convolve().

```
1419 {
       int i = 0;
1420
       mll_t **memList = NULL;
1421
1422
       cll_t *nextPhase = NULL;
1423
       cll_t *allCliques = NULL;
1424
       int length = minLength;
1425
       memList = (mll_t **) malloc (numOffsets * sizeof (mll_t *));
1426
       if (memList == NULL)
1427
1428
           fprintf (stderr, "Memory error - completeConv\n%s\n", strerror (errno));
1429
           fflush (stderr);
1430
           exit (0);
1431
       // The number of offsets will never change, so this can be defined
1432
1433
       // now, though we will have to change what is in these arrays later.
1434
       for (i = 0; i < numOffsets; i++)</pre>
1435
         {
1436
           memList[i] = NULL;
1437
1438
1439
       // NOTE: This assumes that the elemPats all meet the support criterion
1440
1441
       // So we'll do this as long as the head is non-null.. that means that
       // the initial set of cliques must be non-null. Those are then
1442
1443
       // convolved and the linked list for the next round is set to head,
       \ensuremath{//} so this continues until the linked list for the "next round" at
1444
1445
       // the end of some round is null.
       while (*head != NULL)
1446
1447
1448
           // First we get the inverse information for this round: find
1449
           // out which cliques each offset is a member of.
           memList = fillMemberStacks (*head, memList);
1450
1451
           // printf("numOffsets.bak = %d\n",numOffsets);
1452
           // // Then we convolve a whole round.
1453
           nextPhase =
1454
         wholeRoundConv (head, memList, numOffsets, support, length,
1455
                 &allCliques);
1456
           // Do some housekeeping.
```

```
1457
           memList = emptyMemberStacks (memList, numOffsets);
          popAllCll (*head);
1458
1459
          // Enforce the -p flag for subsequent rounds.
1460
          if (p > 1)
1461
         {
1462
           nextPhase = pruneCll (nextPhase, indexToSeq, p);
1463
1464
           // And move on to the next round of convolution.
1465
           *head = nextPhase;
1466
           length++;
1467
1468
1469
      free (memList);
1470
1471
      return allCliques;
1472 }
```

3.4.2.5 mll_t** emptyMemberStacks (mll_t ** memberStacks, int size)

After we have performed a round of convolution, this "empties" the member stacks by popping all nodes off each member linked list. Input: array of member linked lists, the size of that array (total number of offsets). Output: the array of now-empty member linked lists.

Definition at line 538 of file convll.c.

References popWholeMemStack().

Referenced by completeConv().

```
539 {
540    int i = 0;
541
542    for (i = 0; i < size; i++)
543      {
544         memberStacks[i] = popWholeMemStack (memberStacks[i]);
545    }
546
547    return memberStacks;
548 }</pre>
```

3.4.2.6 mll t** fillMemberStacks (cll t * head, mll t ** memberStacks)

Fills the entire memberStacks data structure by calling addToStacks for each clique in the clique linked list. Input: head of a clique linked list, array of member linked lists. Output: the array of updated member linked lists.

Definition at line 517 of file convll.c.

References addToStacks(), and cnode::next.

Referenced by completeConv().

```
518 {
      cll t *curr = head;
519
520
     // Just go down the linked list calling addToStacks
521
      while (curr != NULL)
522
523
          memberStacks = addToStacks (curr, memberStacks);
524
          curr = curr->next;
525
526
527
      return memberStacks;
528 }
```

3.4.2.7 cll_t* initheadCll (cll_t * head, cSet_t * newset)

Initializes the empty head of a linked list by adding a set to that head. Note: this is only called immediately after pushing onto a cll, because the push always creates a new empty head. This function should not be called by the user; see pushcSet. Input: head of a linked list, pointer to a cSet_t list of clique members. Output: head of a linked list.

Definition at line 172 of file convll.c.

References cnode::set.

Referenced by pushcSet().

```
173 {
174
      // Check to make sure that the head is not already initialized.
175
      if (head->set != NULL)
176
177
          printf ("Stack head already initialized!");
178
          exit (0);
179
180
      // Make the head's set pointer point to the new set.
181
      head->set = newset;
182
     return head;
183 }
```

3.4.2.8 mll_t* mergeIntersect (cll_t * first, cll_t * second, mll_t * intersection, bitSet_t * printstatus, int * newSupport)

Convolves two cliques in a non-commutative manner. It finds which members of the first clique are immediately followed by a member in the second clique. Input: pointer to the location in the linked list of the first clique to be convolved, pointer to the location in the linked list of the second clique to be convolved, a member linked list used to store the intersection of the two cliques, the printstatus bitSet, and a pointer to an integer with the support of the clique formed by convolution. Output: a member linked list with the intersection of the two cliques, plus the side effect of that intersection's cardinality being stored in the integer pointed to by newSupport.

Definition at line 759 of file convll.c.

References cSet_t::members, pushMemStack(), and cnode::set.

Referenced by singleCliqueConv().

```
761 {
762
763
      int i = 0, j = 0, status = 0;
764
      // Make sure we are still in-bounds, otherwise we bail out
765
766
      // We'll refer to the offset currently being analyzed from the
767
      // first clique as the 'first offset' and the offset currently
768
      // being analyzed from the second clique as the 'second offset'
769
      while ((i < first->set->size) && (j < second->set->size))
770
771
          // If the second offset is earlier than the first offset plus
772
          // one, then we move on to the next possible second offset
773
          if ((first->set->members[i] + 1) > second->set->members[j])
774
775
          j++;
776
777
          // If the second offset is later than the first offset plus
778
          // one, then we move on the next possible first offset
779
          else if ((first->set->members[i] + 1) < second->set->members[j])
780
        {
781
          i++;
782
783
          // Otherwise, the second offset is equal to the first offset
784
          // plus one, so we have an extendable node. Push that on
785
          // to the intersection stack, move both the first and second
786
          // offsets to their respective next possible offsets, and
787
          // increment the support counter for the new clique (status)
788
          else
789
          intersection = pushMemStack (intersection, first->set->members[i]);
790
791
          i++;
792
          j++;
793
          status++;
794
795
796
797
      // Send the value of the clique's new support out of this function
798
      *newSupport = status;
799
      return intersection;
800 }
```

3.4.2.9 $cSet_t* mllToCSet(mll_t* clique)$

Turns a member linked list used to store the intersection of two cliques into something more useful: a cSet_t structure. Input: a clique in mll_t form. Output: a clique in cSet_t form.

Definition at line 1145 of file convll.c.

References mnode::cliqueMembership, cSet_t::members, mnode::next, and cSet_t::size.

Referenced by pushConvClique().

```
1146 {
       int sizecount = 0, i = 0;
1147
       cSet_t *cliqueCset = malloc (sizeof (cSet_t));
1148
1149
       mll_t *head = clique;
1150
       if (cliqueCset == NULL)
1151
           fprintf (stderr, "Memory error - mllToCSet cSet\n%s\n",
1152
                strerror (errno));
1153
1154
           fflush (stderr);
1155
           exit (0);
1156
1157
       // First count up how many members there are in the member linked list
1158
       while (head != NULL)
1159
1160
           sizecount++;
1161
           head = head->next;
1162
         }
1163
1164
       head = clique;
1165
       cliqueCset->size = sizecount;
1166
       cliqueCset->members = (int *) malloc (sizecount * sizeof (int));
1167
1168
       if (cliqueCset->members == NULL)
1169
1170
           fprintf (stderr, "Memory error - mllTlCSet cliquemembers\n%s\n",
1171
                strerror (errno));
1172
           fflush (stderr);
1173
           exit (0);
1174
         }
1175
       // In order to stay in the same format as with bitSet translation to
1176
       // cSet, we ensure that the ids of the members are ascending with
1177
       // ascending index number in the cSet. This is accomplished by noting
1178
       // that since the intersection members are pushed onto the stack,
1179
       // a LIFO operation, that the first intersected nodes off the stack
1180
       // will have the highest ids, so we will put them at the end of
1181
       // the members array with the higher index values.
1182
       for (i = sizecount - 1; i >= 0; i--)
1183
1184
           cliqueCset->members[i] = head->cliqueMembership;
1185
           head = head->next;
1186
1187
1188
       return cliqueCset;
1189 }
```

3.4.2.10 $cll_t* popAllCll (cll_t* head)$

Shortcut function to pop all of the members of a linked list. Input: head of a linked list. Output: head of a now-empty linked list.

Definition at line 109 of file convll.c.

References popCll().

Referenced by completeConv(), and main().

```
110 {
111    while (head != NULL)
112    {
113         head = popCll (head);
114    }
115    return head;
116 }
```

3.4.2.11 $cll_t*popCll(cll_t*head)$

Removes the head of the clique linked list, returns the new head of the clique linked list, and frees the memory occupied by the old head. Input: head of a linked list. Output: head of a linked list.

Definition at line 66 of file convll.c.

References cSet_t::members, cnode::next, and cnode::set.

Referenced by popAllCll().

```
67 {
68
     \ensuremath{//} by default the new head is NULL...is important later
69
     cll_t *newHead = NULL;
70
     if (head == NULL)
71
72
         fprintf (stderr, "\nCan't pop a null linked list\n");
73
         fflush (stderr);
74
         exit (0);
75
76
     // unless this is the end of the linked list, set the new head
77
     // to the next member of the list. Otherwise, since by default the
78
     // new head is NULL, it will properly return an empty list
79
     if (head->next != NULL)
80
81
         newHead = head->next;
82
83
     // Check to see if there is a set. If there is, and there are members,
84
     // then first free the members. And if there is a set, then free it.
85
     if (head->set != NULL)
86
87
         if (head->set->members != NULL)
88
89
         free (head->set->members);
90
         head->set->members = NULL;
91
92
         free (head->set);
93
         head->set = NULL;
94
```

```
95  // Both the members and set have been freed, so now can free the cll_t
96  // without leaking anything.
97
98  free (head);
99  head = NULL;
100  return newHead;
101 }
```

3.4.2.12 mll_t* popMemStack (mll_t * head)

Pops the head off of a single member linked list. Input: head of a member linked list. Output: the new head of a member linked list after popping one item.

Definition at line 440 of file convll.c.

References mnode::next.

Referenced by popWholeMemStack().

```
441 {
      // by default the new head is NULL...is important later
442
443
     mll_t *newHead = NULL;
444
      if (head == NULL)
445
446
          fprintf (stderr, "\nCan't pop a null linked list - popMemStack\n");
447
          fflush (stderr);
448
          exit (0);
449
450
      if (head->next != NULL)
451
        {
452
          newHead = head->next;
453
        }
454
     free (head);
455
     head = NULL;
456
     return newHead;
457 }
```

3.4.2.13 mll_t* popWholeMemStack (mll_t * head)

Pops all items off of a member linked list. Input: head of a member linked list. Output: empty head of a member linked list.

Definition at line 465 of file convll.c.

References popMemStack().

Referenced by emptyMemberStacks(), and singleCliqueConv().

```
466 {
467 while (head != NULL)
468 {
```

3.4.2.14 int printCll ($cll_t * head$)

Prints the members (cliques) of a linked list in the format: id = unique id number of clique within linked list; Length = number of members of clique, if available; Size = length of each member of clique; Members = newline-separated list of members of the clique. Input: head of a linked list. Output: Gives text output, returns (meaningless) exit value.

Definition at line 128 of file convll.c.

References cnode::id, cnode::length, cSet_t::members, cnode::next, cnode::set, and c-Set t::size.

```
129 {
130
     int i = 0;
     cll_t *curr = head;
     while (curr != NULL)
132
133
         printf ("id = %d\n", curr->id);
134
135
         // Make sure the clique is nonzero in size before attempting
136
         // to print it
137
         if ((curr->set != NULL) && (curr->set->size > 0))
138
139
         if (curr->length >= 0)
140
           {
141
             printf ("Length = %d\n", curr->length);
142
         printf ("Size = %d\n", curr->set->size);
143
         printf ("Members = \n");
144
145
         for (i = 0; i < curr->set->size; i++)
146
147
             printf ("\t%d\n", curr->set->members[i]);
148
         149
150
       }
151
         else
152
153
         fprintf (stderr, "\nClique has no members! -- printCll\n");
154
         fflush (stderr);
155
         exit (0);
156
157
         curr = curr->next;
158
159
     return EXIT_SUCCESS;
160 }
```

3.4.2.15 int printCllPattern (cll_t * node, int length)

Prints out the contents of a clique linked list node in this format: support = number of motif occurrences (id = some id number); members = newline-separated list of offsets. Input: a specific node to be output, the length of the motif inside it. Output: text per above, and an integer success value.

Definition at line 1482 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

```
1483 {
1484
       int i = 0;
1485
1486
       printf ("\nSupport = %d\t(id = %d)\n", node->set->size, node->id);
1487
       printf ("Members = \n");
1488
       for (i = 0; i < node->set->size; i++)
1489
1490
           printf ("\t%d\n", node->set->members[i]);
1491
1492
       return 1;
1493 }
```

3.4.2.16 int printCSet ($cSet_t * node$)

Prints out the contents of a cSet_t in the following format: *support* = number of nodes in clique; *members* = newline-separated list of nodes in clique. Input: a clique in the form of a cSet_t object. Output: in text, the contents of the cSet_t object. An integer is returned as well, with 1 indicating success.

Definition at line 1068 of file convll.c.

References cSet_t::members, and cSet_t::size.

```
1069 {
1070
       int i = 0;
1071
       if (node->size == 0)
1072
1073
           fprintf (stderr, "cSet has no members! - printCSet\n");
1074
           fflush (stderr);
1075
           exit (0);
1076
1077
       else
1078
         {
1079
           printf ("\nSupport = %d\n", node->size);
           printf ("Members = \n");
1080
1081
           for (i = 0; i < node->size; i++)
1082
1083
           printf ("\t%d\n", node->members[i]);
1084
1085
           return 1;
1086
1087 }
```

3.4.2.17 void printMemberStacks (mll_t ** memberStacks, int size)

Prints the contents of the member stacks. Input: array of member linked lists, size of that array (total number of offsets). Output: only text output/no return value.

Definition at line 557 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

```
558 {
      int i = 0;
559
560
      mll_t *curr = NULL;
561
562
      for (i = 0; i < size; i++)
563
564
          curr = memberStacks[i];
565
          printf ("Offset %d: ", i);
566
          while (curr != NULL)
567
568
          printf ("%d,", curr->cliqueMembership);
569
          curr = curr->next;
570
571
          printf ("\n");
572
573 }
```

3.4.2.18 cll_t* pruneCll (cll_t * head, int * indexToSeq, int p)

Prunes a motif linked list of all motifs without support in at least

unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```
515 {
      int i = 0, j = 0, thisSeq = 0;
516
517
      int *seqNums = NULL;
      cll_t * curr = head;
      cll_t * prev = NULL;
519
      cll_t * storage = NULL;
520
521
        // We'll do this similar to the pruneBitGraph function... we will
522
523
        // keep track of which source sequence each motif occurrence was in.
524
        // Again, since the occurrences are listed monotonically, we only
525
        // need to compare the last non-sentinel index to the current
526
        // sequence number.
```

```
527
        seqNums = (int *) malloc (p * sizeof (int));
528
      if (seqNums == NULL)
529
530
          fprintf \ (stderr, \ "Memory error - pruneCll\n\$s\n", \ strerror \ (errno));
531
          fflush (stderr);
532
          exit (0);
533
534
      while (curr != NULL)
535
536
537
        // First make sure the set size is at least p.
538
        // This is redundant, but extremely simple and not expensive,
539
        // so we'll leave it in just as a check.
540
        if (curr->set->size < p)</pre>
541
542
          if (prev != NULL)
543
           {
544
             prev->next = curr->next;
545
546
          else
547
            {
548
              head = curr->next;
            }
549
550
          storage = curr->next;
551
          free (curr->set->members);
552
          free (curr->set);
553
          free (curr);
554
          curr = storage;
555
          continue;
556
557
          for (i = 0; i < p; i++)
558
        {
559
          seqNums[i] = -1;
560
        }
          j = 0;
561
562
          seqNums[0] = indexToSeq[curr->set->members[0]];
563
564
        // Note, we've checked to make sure size > p, and we know
565
        // p must be 2 or greater, so we can start at 1 without
566
        // worrying about segfaulting
567
        for (i = 1; i < curr->set->size; i++)
568
569
          thisSeq = indexToSeq[curr->set->members[i]];
570
          if (thisSeq != seqNums[j])
571
            {
572
              j++;
573
              seqNums[j] = thisSeq;
              if (j == p - 1)
574
575
576
              break;
577
578
579
        }
580
        // Same story as before... if the last number is -1,
581
        // then we didn't have enough to fill up the  different
583
        // slots, so this doesn't meet our criterion.
```

```
584
        if (seqNums[p - 1] == -1)
585
586
          if (prev != NULL)
587
            {
588
              prev->next = curr->next;
589
590
          else
591
592
              head = curr->next;
593
594
          storage = curr->next;
595
          free (curr->set->members);
596
          free (curr->set);
597
          free (curr);
598
          curr = storage;
599
600
          else
601
602
          prev = curr;
603
          curr = curr->next;
604
605
606
      free (seqNums);
607
      return (head);
608 }
```

3.4.2.19 cll_t* pushClique (bitSet_t * clique, cll_t * head, int * indexToSeq, int p)

Pushes a bitSet onto a clique linked list, performing all necessary manipulations in order to do so. Input: new clique in the form of a bitSet_t, head of a linked list, pointer to the index/sequence number data structure, integer value of the -p flag. Output: head of an updated clique linked list.

Definition at line 345 of file convll.c.

References bitSetToCSet(), checkCliquecSet(), cliquecounter, and pushcSet().

Referenced by findCliques(), and singleLinkage().

```
346 {
      cSet_t *cliquecSet = NULL;
348
349
      // Change the bitSet_t to a cSet_t
350
      cliquecSet = bitSetToCSet (clique);
351
      // If the -p flag has been assigned a value, then check the clique
      // and only proceed if that criterion is met. Otherwise, free the
353
      \ensuremath{//} memory that we had allocated up to this point.
354
      if (p > 1)
355
356
          if (checkCliquecSet (cliquecSet, indexToSeq, p))
357
358
          cliquecounter++;
359
             printf("%d\n",cliquecounter);
360
```

```
* /
361
362
363
             fflush(stdout);
           * /
364
365
          head = pushcSet (head, cliquecSet);
366
367
          else
368
369
          free (cliquecSet->members);
370
          free (cliquecSet);
371
372
          // If the -p flag wasn't set, then just push the cSet onto the linked
373
374
375
      else
376
377
          cliquecounter++;
378
379
             printf("%d\n",cliquecounter);
380
381
382
             fflush(stdout);
383
384
          head = pushcSet (head, cliquecSet);
385
386
      return head;
387 }
```

3.4.2.20 cll_t* pushCll (cll_t * head)

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to initheadCll, as the head pushed on here is empty and will be meaningless without any members. This function should NOT be used by the user; see pushcSet. Input: head of a linked list. Output: head of a linked list.

Definition at line 28 of file convll.c.

References cnode::id, cnode::length, cnode::next, cnode::set, and cnode::stat.

Referenced by pushcSet().

```
29 {
     // Make a pointer, verify memory
31
    cll_t *a = NULL;
    a = (cll_t *) malloc (sizeof (cll_t));
     if (a == NULL)
34
35
         fprintf (stderr, "\nMemory Error - pushCll\n%s\n", strerror (errno));
         fflush (stderr);
36
37
         exit (0);
38
39
     // Initialize id (sequential) and pointer to next item, but not
     // the cSet with the clique members
     if (head == NULL)
41
```

```
42
        {
43
          a \rightarrow id = 0;
44
          a->next = NULL;
       }
45
46
     else
47
        {
48
          a->next = head;
49
          a->id = head->id + 1;
50
       }
51
     a->set = NULL;
52
     a - > length = -1;
53
     a - stat = -1;
54
     return a;
55 }
```

3.4.2.21 cll_t* pushConvClique (mll_t * clique, cll_t * head)

Pushes a freshly-convolved clique, currently in mll_t form, onto the clique linked list for the next level. Also checks to make sure that the convolved clique is unique, and if it isn't, it takes appropriate action. Input: a convolved clique in mll_t form, the head of a clique linked list for the next level. Output: (potentially new) head of the clique linked list for the next level.

Definition at line 1099 of file convll.c.

References cSet_t::members, mllToCSet(), pushcSet(), removeSupers(), swapNodecSet(), and uniqClique().

Referenced by singleCliqueConv().

```
1100 {
1101
      int status = 0;
1102
      cSet_t *cliquecSet = NULL;
1103
1104
      // First change the clique to something we can used more easily
1105
      cliquecSet = mllToCSet (clique);
1106
      // Then check to make sure it's unique by finding out its status
1107
       status = uniqClique (cliquecSet, head);
1108
1109
      // printf("Candidate:\n");
1110
      // printCSet(cliquecSet);
1111
1112
       // If we get -2, then this clique is a subset, so just free
      // the cSet we just made and move on.
1113
1114
      if (status == -2)
1115
1116
           free (cliquecSet->members);
1117
           free (cliquecSet);
1118
           cliquecSet = NULL;
1119
1120
       // If we get -1, then this is a unique clique, so push it on.
1121
       else if (status == -1)
1122
         {
```

```
1123
           head = pushcSet (head, cliquecSet);
1124
1125
       // Otherwise, this clique is a superset, so we'll first remove
1126
       \ensuremath{//} all of the other cliques of which this is a superset. Then
1127
       // we'll swap out the first clique of which this is a superset
1128
       // with this current clique. The clique being removed is free'd
1129
       // within the swapNode function.
1130
       else
1131
1132
           head = removeSupers (head, status, cliquecSet);
1133
           head = swapNodecSet (head, status, cliquecSet);
1134
         }
1135
       return head;
1136 }
```

3.4.2.22 cll_t* pushcSet (cll_t * head, cSet_t * newset)

Function that pushes the contents of a cSet (set of members of a clique) onto a linked list of cliques. Input: head of a linked list, new clique in the form of a cSet_t. Output: head of a linked list.

Definition at line 192 of file convll.c.

References initheadCll(), and pushCll().

Referenced by pushClique(), and pushConvClique().

```
193 {
194    head = pushCll (head);
195    head = initheadCll (head, newset);
196    return head;
197 }
```

3.4.2.23 mll_t* pushMemStack (mll_t * head, int cliqueNum)

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 404 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

```
405 {
      mll_t *a = NULL;
406
      a = (mll_t *) malloc (sizeof (mll_t));
408
      // Memory error checking
409
      if (a == NULL)
410
          fprintf (stderr, "\nMemory Error - pushMemStack: %s\n",
411
412
               strerror (errno));
413
          fflush (stderr);
414
          exit (0);
415
416
      if (head == NULL)
417
418
          a->next = NULL;
419
420
      else
421
422
          a->next = head;
423
424
      // Store the number of the clique of which the node is a member.
      \ensuremath{//} Note that we assume no duplication, which is guaranteed
425
426
      // by our method of filling the member stacks, which is quite simple:
427
      // go through all members of a clique (which have no duplicates
428
      // because they are constructed from merge-intersections or from
429
      // bitSet_t's) and add that clique to each node's membership list.
430
      a->cliqueMembership = cliqueNum;
431
      return a;
432 }
```

3.4.2.24 cll_t* removeSupers (cll_t * head, int node, cSet_t * newClique)

This function finds all cliques in a linked list of which the proposed clique is a superset. It starts looking AFTER the first clique which has already been found to be a subset. In some senses, it is just a continuation of the unique function in order to take advantage of the fact that though a proposed clique can only be a subset of one existing next-level clique, it can be a superset of many existing next-level cliques. Input: head of a clique linked list, the id of the first node found to be a subset of the proposed clique, and the proposed clique (in cSet_t form). Output: the head of the clique linked list with all but the first subset (which was passed as an argument) removed. This function is now ready for swapNode to be called.

Definition at line 952 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

```
953 {
954    int foundStatus = 0;
955    cll_t *curr = head;
956    cll_t *prev = NULL;
957    int i = 0, j = 0, breakFlag = 0;
```

```
958
      while (curr != NULL)
959
960
961
          if (curr->id == node)
962
963
          foundStatus = 1;
964
          break;
965
966
          curr = curr->next;
967
968
969
      if (foundStatus == 0)
970
        {
971
          fprintf (stderr, "\nFirst clique not found! (removeSupers)\n");
972
          fflush (stderr);
973
          exit (0);
974
975
      // Now this is trickier, to remove nodes from the middle of a linked
976
      // list; this means that we need to remember which node we were just
977
      // at so that we can connect it to the node after the one we are
978
     // about to delete.
979
     prev = curr;
980
      curr = curr->next;
981
982
      // This code is similar to that in uniqClique.
      // Descend through all members of the next level's linked list.
983
      while (curr != NULL)
984
985
        {
          i = 0;
986
987
          j = 0;
988
          breakFlag = 0;
989
          \/\/ The proposed convolved clique will be referred to as the
990
          // 'first' clique, and the current clique being analyzed
991
          // in the next level is the 'second' clique.
992
          // Continue if we have more members in both cliques. We will
          // have already broken out if it is not possible for this
993
994
          // second clique to be a subset of the first.
995
          while ((i < newClique->size) && (j < curr->set->size))
996
997
          // If the current member of the first clique is
998
          // less than the current member of the second clique
999
          // then it is still possible that the first is a
1000
           // superset of the second, so move on to the next
1001
           // member.
1002
           if (newClique->members[i] < curr->set->members[j])
1003
             {
1004
               i++;
1005
1006
           // If the current member of the first clique is greater
1007
           // than the current member of the second clique, then
1008
           // the proposed second clique cannot be a subset since
1009
           // its members are all in ascending order. We also
1010
           // know that since the first clique already has
1011
           // a subset in this linked list, the current node
1012
           // cannot possibly be a superset of the proposed
           // clique, so we can just disregard that. Thus,
1013
1014
           \ensuremath{//} we make a flag signifying this and break out.
```

```
1015
           else if (newClique->members[i] > curr->set->members[j])
1016
1017
               breakFlag = 1;
1018
               break;
1019
1020
           else
1021
1022
               i++;
1023
               j++;
1024
             }
1025
1026
           // If the breakflag is 1, then we know
1027
           // that there is a member of the second clique not in the
1028
           // first, and so the second is not a subset. If the breakflag
1029
           // is 0 but j is less than the second clique's size, then
1030
           // we must have broken because we ran out of members in the
1031
           // first clique... thus, there is a member of the second
1032
           // clique not in the first. Thus, only if the breakflag is
1033
           \ensuremath{//} 0 and j is equal to the size of the second clique do we
1034
           // know that every member of the second clique is in the first
           \ensuremath{//} and that the second clique can thus be removed.
1035
1036
           if ((breakFlag == 0) && (j == curr->set->size))
1037
         {
1038
           // Make the previous clique point to the next one
1039
           // instead of the current one.
1040
           prev->next = curr->next;
1041
           // Free all of the memory used by the current clique.
1042
           free (curr->set->members);
1043
           free (curr->set);
1044
           free (curr);
1045
           curr = prev->next;
1046
         }
1047
           else
1048
         {
1049
           // Otherwise, the current second clique is not a
1050
           // subset of the first, and we advance the prev and
1051
           // curr pointers.
1052
           prev = curr;
1053
           curr = curr->next;
1054
1055
1056
      return head;
1057 }
```

3.4.2.25 bitSet_t* searchMemsWithList (int * list, int listsize, mll_t ** memList, int numOffsets, bitSet_t * queue)

Creates one large queue by calling "setStackTrue" for each member of a list of offsets. This then creates the union of clique membership for all offsets in the list being searched. Input: an array of offset numbers, the length of that array, an array of member linked lists, the length of that array (the total number of offsets), and a bitSet_t to store the union/queue. Output: the union/queue in a bitSet_t structure.

Definition at line 611 of file convll.c.

 $References\ empty Set(),\ and\ set Stack True().$

Referenced by wholeCliqueConv().

```
613 {
      int i = 0;
614
615
      emptySet (queue);
616
      // Go through each offset in the list
617
618
      for (i = 0; i < listsize; i++)
619
620
          // Check to make sure that's a valid offset number, and if so
621
          // then set its stack true in the queue.
622
          if (list[i] + 1 < numOffsets)</pre>
623
          queue = setStackTrue (memList, list[i] + 1, queue);
624
625
        }
626
          else
627
        {
          fprintf (stderr, "\nInvalid offset number! - searchMemsWithList\n");
628
          fprintf (stderr, "\nlist[i]+1 (%d) >= numOffsets (%d)\n",
629
               list[i] + 1, numOffsets);
630
631
          fflush (stderr);
632
          exit (0);
633
634
635
636
     return queue;
637 }
```

3.4.2.26 bitSet_t* setStackTrue (mll_t ** memList, int i, bitSet_t * queue)

Adds all of the members of a given stack to a "queue" in the form of a bitSet_t data structure. That is, for each clique in the member linked list, it sets the corresponding bit in the bitSet_t true. Input: array of member linked lists, an integer indicating a specific member linked list, and a bitSet_t of length >= the number of cliques in the current clique linked list. Ouput: the updated bitSet_t object.

Definition at line 585 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

3.4.2.27 cll_t* singleCliqueConv (cll_t * head, int firstClique, cll_t ** firstGuess, int secondClique, cll_t ** secondGuess, cll_t * nextPhase, bitSet_t * printStatus, int support)

Convolves one single clique against one other single clique. Note that this is non-commutative, so exchanging firstClique and secondClique will not give the same results. The "guess" pointers keep the location of the previous clique in the linked list so that we don't have to search the linked list from the beginning/end every time. We exploit our earlier tidiness in that we can reasonably guess that we will monotonically traverse down cliques. Input: head of the current clique linked list, the id number of the first clique, a pointer to a guess at the first clique, the id number of the second clique, a pointer to a guess at the second clique, the head of the clique linked list for the next round of convolution, a bitSet indicating which cliques should be output as maximal, and the minimum support flag. Output: the head of clique linked list for the next round of convolution (which may have changed if the two cliques could be convolved).

Definition at line 657 of file convll.c.

References cnode::id, mergeIntersect(), cnode::next, popWholeMemStack(), push-ConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

```
660 {
      cll_t *first = NULL, *second = NULL;
661
     mll_t *survivingMems = NULL;
662
     // int flag = 0;
664
     int newSupport = 0;
665
      // cll_t *checker = head;
666
667
      // Check to make sure we're looking for legitimate cliques.
668
      if ((firstClique > head->id) | (secondClique > head->id))
669
670
          fprintf (stderr, "\nNonexistent clique! - singleCliqueConv\n");
671
          fflush (stderr);
672
          exit (0);
673
674
      // Our guesses depend on monotonic traversal. If we don't find
675
      // the first clique, then bail out.
676
      while ((*firstGuess)->id != firstClique)
677
678
          if ((*firstGuess)->next != NULL)
679
680
          *firstGuess = (*firstGuess)->next;
681
```

```
682
          else
683
684
          fprintf (stderr, "\nFirst clique not found! - singleCliqueConv\n");
685
          fflush (stderr);
686
          exit (0);
687
688
689
      first = *firstGuess;
690
691
      // Our guesses depend on monotonic traversal. If we don't find
692
      // the second clique, then bail out.
693
     while ((*secondGuess)->id != secondClique)
694
695
          if ((*secondGuess)->next != NULL)
696
        {
697
          (*secondGuess) = (*secondGuess)->next;
698
        }
699
          else
700
701
          fprintf (stderr, "\nSecond clique not found! - singleCliqueConv\n");
702
          fflush (stderr);
703
          exit (0);
704
705
        }
706
     second = *secondGuess;
     // Find out what the surviving members are when the first clique
707
708
     // is convolved with the second clique
709
     survivingMems =
710
       mergeIntersect (first, second, survivingMems, printStatus, &newSupport);
711
712
     // If the first clique is subsumed by the second, then it is not
713
     // maximal, so don't print it.
714
      // printStatus true means print it!
715
      if (newSupport == first->set->size)
716
       {
717
          setFalse (printStatus, first->id);
718
719
      // If the second clique is subsumed by the first, then it is not
      // maximal, so don't print it.
721
      if (newSupport == second->set->size)
722
        {
723
          setFalse (printStatus, second->id);
724
        }
725
726
      // If the support of the clique just formed by convolution meets the
727
      // support criterion, then push it on to the linked list for
728
      // the next phase of convolution.
729
     if (newSupport >= support)
730
731
          // printf("Push %d and %d\n",first->id,second->id);
732
         nextPhase = pushConvClique (survivingMems, nextPhase);
          // printf("----\n");
733
734
         // printCll(nextPhase);
735
          // printf("----\n");
736
      // Pop the surviving members; they are no longer needed, as they
      // either didn't meet the support criterion or have been pushed on
738
```

```
739  // already
740  survivingMems = popWholeMemStack (survivingMems);
741
742  return nextPhase;
743 }
```

3.4.2.28 cll_t* swapNodecSet (cll_t * head, int node, cSet_t * newClique)

Swaps out a node in a linked list that has been found to be a subset of a node that is not yet in the list. Input: the head of a clique linked list, a specific node within that linked list that is to be removed, and the new clique that is the superset of the node to be removed (in cSet_t form). Output: the head of the altered clique linked list.

Definition at line 904 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, and cnode::set.

Referenced by pushConvClique().

```
905 {
906
      int foundflag = 0;
907
      cll_t *curr = head;
908
      // First we find the node that needs to be swapped out
910
      while (curr != NULL)
911
912
          if (curr->id == node)
913
        {
914
          foundflag = 1;
915
         break;
916
917
          curr = curr->next;
918
919
920
      // If we can't find it, then we get upset and exit.
921
      if (foundflag == 0)
922
923
          fprintf (stderr, "\nClique not found! (in swapNode)\n");
924
          fflush (stderr);
925
          exit (0);
926
      // Then we free the useless clique's members and its set data structure
927
928
     // before pointing its set to the new clique.
929
     free (curr->set->members);
930
     free (curr->set);
931
      curr->set = newClique;
932
      return head;
933
934 }
```

3.4.2.29 int uniqClique (cSet_t * cliquecSet, cll_t * head)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in cSet_t form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset. Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 821 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

```
822 {
      int i = 0, j = 0;
823
824
      int asubbflag = 1, bsubaflag = 1;
825
826
      // Descend through all members of the next level's linked list
827
      while (head != NULL)
828
829
          asubbflag = 1;
830
          bsubaflag = 1;
831
          i = 0;
832
          j = 0;
833
          // The proposed convolved clique will be referred to as the
834
          // "first" clique, and the current clique being analyzed
          // in the next level is the "second" clique.
835
836
          // Continue if we have more members in both cliques AND if it
837
          // is still possible for one clique to be a subset of
838
          // the other.
839
          while ((i < cliquecSet->size) && (j < head->set->size) &&
             ((asubbflag == 1) || (bsubaflag == 1)))
840
841
        {
842
          // If the current member of the first clique is less
843
          // than the current member of the second clique,
844
          // it is impossible for the first clique to be a
845
          // subset of the second (since the members are
          // traversed in ascending order.
846
          if (cliquecSet->members[i] < head->set->members[j])
847
848
            {
              i++;
849
              asubbflag = 0;
850
851
          // Similarly, if the current member of the second
852
          // clique is less than the current member of the
853
854
          // second clique, the second can't be a subset
```

```
855
          // of the first.
856
          else if (cliquecSet->members[i] > head->set->members[j])
857
            {
858
              j++;
859
              bsubaflag = 0;
860
          // Otherwise, they matched this time, so move them
861
862
863
          else
864
              i++;
865
866
              j++;
            }
867
868
869
870
          \ensuremath{//} If the proposed clique is a subset of some other clique
871
          // in the next level, then return -2, and it won't be added.
872
          // (Note, this also is how exact duplicates are handled.)
873
          if ((asubbflag == 1) && (i == cliquecSet->size))
874
875
          return (-2);
876
877
          // If the proposed clique is a superset of some other clique(s)
878
          // in the next level, then return the id of the first clique
879
          // of which it is a superset.
880
          if ((bsubaflag == 1) && (j == head->set->size))
881
          return (head->id);
882
883
884
          // If the proposed clique has not been found to be a superset
885
          // or a subset yet, then move on to the next clique in
886
          // the next level.
887
          head = head->next;
888
889
      // If we've gotten here, we've checked all cliques in the previous
890
      // level and haven't found the proposed clique to be a superset or
891
      // a subset... if so, then we're all good, so return a -1.
892
      return (-1);
893 }
```

3.4.2.30 cll_t* wholeCliqueConv (cll_t * head, cll_t * node, cll_t ** firstGuess, mll_t ** memList, int numOffsets, cll_t * nextPhase, bitSet_t * printStatus, int support)

Convolves one single clique against all possible cliques that could possibly be convolved. It does not attempt to convolve all other cliques, but prunes that set by first looking at the offsets that are in the clique, then collecting all of the cliques who have members that are one greater than the offsets in this clique, and then convolving those cliques in a sort of "queue" using the bitSet_t data structure. Input: the head of the clique linked list for the current level, the current node being convolved against in the linked list, the location of the previous node in the form of a pointer to a "guess", an array of member linked lists, the length of that array, the head of the clique linked list

for the next level, a bitSet_t for the printStatus of maximality, and the support criterion. Output: the head of the (possibly modified) clique linked list for the next level.

Definition at line 1208 of file convll.c.

References bitSetToCSet(), countSet(), deleteBitSet(), cnode::id, cSet_t::members, newBitSet(), searchMemsWithList(), cnode::set, singleCliqueConv(), and cSet_t::size.

Referenced by wholeRoundConv().

```
1211 {
1212
      bitSet_t *queue = NULL;
1213
       cSet_t *cliquesToSearch = NULL;
1214
       int i = 0;
1215
       cll_t **secondGuess = NULL;
1216
1217
       // This bitSet will be used to create a "queue" of the different
1218
       // cliques that must be convolved against the current primary clique.
1219
       // A bitset is used to make it easy to deal with duplicates, where
1220
       // multiple clique members' next offsets
1221
       // are all members of some other specific clique.
1222
       queue = newBitSet (head->id + 1);
1223
       queue =
1224
        searchMemsWithList (node->set->members, node->set->size, memList,
1225
                 numOffsets, queue);
1226
       // We'll use this "secondGuess" to store where the previous clique
1227
       // being convolved was... since we will progressing monotonically
1228
       // in descending order, this will save us some time in traversing the
1229
       // linked list looking for the clique that we want.
1230
       secondGuess = (cll_t **) malloc (sizeof (cll_t *));
1231
       if (secondGuess == NULL)
1232
1233
           fprintf (stderr, "Memory error - wholeCliqueConv\n%s\n",
1234
                strerror (errno));
           fflush (stderr);
1235
1236
           exit (0);
1237
       // If the offsets that we are looking for are in no other cliques,
1238
1239
       // we can just bail out now.
1240
       if (countSet (queue) == 0)
1241
         {
1242
           deleteBitSet (queue);
1243
           return nextPhase;
1244
1245
       // Otherwise, we start our secondGuess at the head and get going.
1246
       *secondGuess = head;
1247
1248
       // We change the bitSet to something more useful.
1249
       cliquesToSearch = bitSetToCSet (queue);
1250
1251
       // Note that we start from the end of the cSet member list so that
1252
       // we can convolve the highest-id cliques first, which are at the
1253
       // beginning of our stack of cliques.
1254
       for (i = cliquesToSearch->size - 1; i >= 0; i--)
1255
1256
           nextPhase = singleCliqueConv (head, node->id, firstGuess,
1257
                         cliquesToSearch->members[i], secondGuess,
```

3.4.2.31 cll_t* wholeRoundConv (cll_t ** head, mll_t ** memList, int numOffsets, int support, int length, cll_t ** allCliques)

Performs convolution on all cliques in a linked list by repeatedly calling wholeClique-Conv. Input: pointer to the head of a clique linked list for the current level, array of member linked lists, length of that array, minimum support threshold, the current length of motifs, and a pointer to a linked list containing all cliques that will be printed out. Output: the head of the clique linked list for the next level of convolution.

Definition at line 1279 of file convll.c.

References checkBit(), deleteBitSet(), fillSet(), cnode::id, newBitSet(), cnode::next, wholeCliqueConv(), and yankCll().

Referenced by completeConv().

```
1281 {
      bitSet_t *printStatus = NULL;
1282
      cll_t *curr = *head;
1283
      cll_t *prev = NULL;
1284
1285
      cll_t *nextPhase = NULL;
1286
      cll_t **firstGuess = NULL;
1287
      // Create a bitset to keep track of print status for this level.
1288
1289
      // It starts off all true, and gets changed to false if the patterns
1290
      // are not maximal.
1291
      printStatus = newBitSet ((*head)->id + 1);
1292
      fillSet (printStatus);
1293
      firstGuess = (cll_t **) malloc (sizeof (cll_t *));
1294
      if (firstGuess == NULL)
1295
        {
1296
           fprintf (stderr, "Memory error - wholeRoundConv\n%s\n",
1297
               strerror (errno));
1298
           fflush (stderr);
1299
           exit (0);
1300
1301
      // Start off at the head.
1302
      *firstGuess = *head;
1303
      // Convolve a whole clique at a time, traversing the linked list.
1304
      // Note that firstGuess gets altered within the function.
1305
      while (curr != NULL)
1306
         {
```

```
1307
         nextPhase =
1308
        wholeCliqueConv (*head, curr, firstGuess, memList, numOffsets,
1309
               nextPhase, printStatus, support);
1310
         curr = curr->next;
1311
1312
1313
      // Now go back to the head for printing output
1314
      curr = *head;
1315
      1316
      // printf("Length = %d", length);
1317
      1318
1319
1320
      // For each clique that is still 'true' in printStatus and is thus
1321
      // maximal, perform some sort of output. Yankcll will pull out the
1322
      // clique and save it for printing at a later time.
1323
      while (curr != NULL)
1324
1325
         if (checkBit (printStatus, curr->id))
1326
1327
         // This is the line that makes the allCliques output.
1328
         // Can either printcll, or add to allCliques.
1329
         // printCllPattern(curr, length);
1330
         yankCll (head, prev, &curr, allCliques, length);
1331
1332
         else
1333
1334
         prev = curr;
1335
         curr = curr->next;
1336
1337
1338
1339
      // And clean up.
1340
      deleteBitSet (printStatus);
1341
      free (firstGuess);
1342
      return nextPhase;
1343 }
```

3.4.2.32 int yankCll (cll_t ** head, cll_t * prev, cll_t ** curr, cll_t ** allCliques, int length)

Removes a clique from within a linked list in order to save it for later printing. This is done so that the cliques are not printed as they are convolved, but rather after all rounds of convolution are complete. Input: a pointer to the head of the current linked list, the clique prior to the one that is to be yanked (NULL if the clique to be yanked is the head), the clique that is to be yanked, a pointer to the head of the list with all cliques that are to be printed, and the length of the current motif. Output: Nothing is returned beyond a success integer, but it alters the current level cll_t, the value of curr, and the linked list of all cliques that are to be printed.

Definition at line 1359 of file convll.c.

References cnode::id, and cnode::next.

Referenced by convolve(), and wholeRoundConv().

```
1361 {
1362
       if (*curr == NULL)
1363
1364
           fprintf (stderr, "\nCan't yank from end of cll!\n");
           fflush (stderr);
1365
1366
           exit (0);
1367
1368
      // If we're not on the head, change the previous node's "next".
1369
       // If we are on the head, make the new head be our current node's "next".
1370
       if (prev != NULL)
1371
1372
          prev->next = (*curr)->next;
1373
1374
       else
1375
1376
           *head = (*curr)->next;
1377
1378
       // Change next in curr, then change id and length information in curr
1379
       (*curr)->next = *allCliques;
1380
1381
1382
       if (*allCliques != NULL)
1383
       {
1384
           (*curr)->id = (*allCliques)->id + 1;
1385
1386
       else
1387
1388
           (*curr)->id = 0;
1389
1390
1391
       (*curr)->length = length;
1392
1393
       *allCliques = *curr;
1394
1395
       if (prev != NULL)
1396
1397
           *curr = prev->next;
1398
1399
       else
1400
1401
           *curr = *head;
1402
1403
       return (1);
1404 }
```

3.4.3 Variable Documentation

3.4.3.1 int clique counter = 0

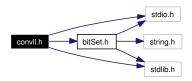
Definition at line 335 of file convll.c.

Referenced by pushClique().

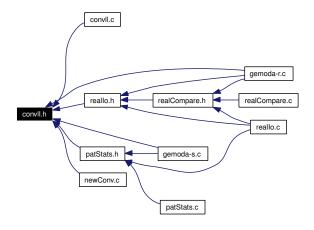
3.5 convll.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include "bitSet.h"
```

Include dependency graph for convll.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct cSet_t
- struct cnode
- struct mnode

Typedefs

- typedef cnode cll_t
- typedef mnode mll_t

Functions

```
• cll t * pushCll (cll t *head)
```

- cll_t * popCll (cll_t *head)
- cll_t * popAllCll (cll_t *head)
- int printCll (cll t *head)
- cll_t * initheadCll (cll_t *head, cSet_t *newset)
- cll_t * pushcSet (cll_t *head, cSet_t *newset)
- cll_t * pushClique (bitSet_t *clique, cll_t *head, int *indexToSeq, int p)
- mll_t * pushMemStack (mll_t *head, int cliqueNum)
- mll_t * popMemStack (mll_t *head)
- mll_t * popWholeMemStack (mll_t *head)
- mll_t ** addToStacks (cll_t *node, mll_t **memberStacks)
- mll_t ** fillMemberStacks (cll_t *head, mll_t **memberStacks)
- mll t ** emptyMemberStacks (mll t **memberStacks, int size)
- void printMemberStacks (mll_t **memberStacks, int size)
- bitSet_t * searchMemsWithList (int *list, int listsize, mll_t **memList, int num-Offsets, bitSet_t *queue)
- bitSet t * setStackTrue (mll t **memList, int i, bitSet t *queue)
- cll_t * singleCliqueConv (cll_t *head, int firstClique, cll_t **firstGuess, int secondClique, cll_t **secondGuess, cll_t *nextPhase, bitSet_t *printStatus, int support)
- mll_t * mergeIntersect (cll_t *first, cll_t *second, mll_t *intersection, bitSet_t *printStatus, int *newSupport)
- cll t * pushConvClique (mll t *clique, cll t *head)
- cSet_t * mllToCSet (mll_t *clique)
- cSet t * bitSetToCSet (bitSet t *clique)
- cll_t * wholeCliqueConv (cll_t *head, cll_t *node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t *nextPhase, bitSet_t *printStatus, int support)
- cll_t * wholeRoundConv (cll_t **head, mll_t **memList, int numOffsets, int support, int length, cll_t **allCliques)
- cll_t * completeConv (cll_t **head, int support, int numOffsets, int minLength, int *indexToSeq, int p)
- int printCllPattern (cll_t *node, int length)
- int uniqClique (cSet_t *clique, cll_t *head)
- cll_t * swapNodecSet (cll_t *head, int node, cSet_t *newClique)
- int yankCll (cll_t **head, cll_t *prev, cll_t **curr, cll_t **allCliques, int length)
- cll_t * removeSupers (cll_t *head, int node, cSet_t *newClique)

3.5.1 Detailed Description

This header file contains declarations and definitions for dealing with different kinds of sets that are used throughout the convolution stage of Gemoda.

Definition in file convll.h.

3.5.2 Typedef Documentation

3.5.2.1 typedef struct cnode cll_t

This data structure is a linked list for storing cliques. Each member of the linked list has a set, an ID number, a length (which gives the number of characters in the motif), a pointer to the next member of the linked list, and a floating-point number for storing statistical information.

3.5.2.2 typedef struct mnode mll_t

This data structure is just a link to list of integers used for bookkeeping during the convolution stage.

3.5.3 Function Documentation

3.5.3.1 mll_t** addToStacks (cll_t * node, mll_t ** memberStacks)

For one clique, it adds membership for that clique to all of its members' member stacks. Input: a specific clique in a clique linked list, an array of member stacks. Output: the array of updated member stacks.

Definition at line 425 of file convll.c.

References cnode::id, cSet_t::members, pushMemStack(), and cnode::set.

Referenced by fillMemberStacks().

3.5.3.2 cSet_t* bitSetToCSet (bitSet_t * clique)

Converts a bitSet_t to a cSet_t for the purposes of pushing it onto a linked list of cliques. The bitSet_t data structure is used for massive comparisons during clique-finding but is unwieldy/inefficient when it is known that the structure is sparse. The cSet_t allows for efficient comparison of sparse bitSet_t's. Use this just before pushing a newly-discovered clique onto a clique linked list. Input: a new clique in the form of a bitSet_t. Output: the same clique in the form of a cSet_t.

Definition at line 193 of file convll.c.

References countSet(), cSet_t::members, nextBitBitSet(), and cSet_t::size.

Referenced by pushClique(), and wholeCliqueConv().

3.5.3.3 cll_t* completeConv (cll_t ** head, int support, int numOffsets, int minLength, int * indexToSeq, int p)

Performs complete convolution given the starting list of cliques. Input: a pointer to the head of the initial clique linked list, the minimum support criterion value, the number of offsets in the sequence set, the minimum length of motifs (which is the length of motifs in the initial clique linked list), the index/Sequence data structure, and the value of the -p flag to prune based on unique sequence occurrences. Output: a linked list of all maximal cliques based on the initial clique linked list.

Definition at line 1267 of file convll.c.

References emptyMemberStacks(), fillMemberStacks(), popAllCll(), pruneCll(), and wholeRoundConv().

Referenced by convolve().

3.5.3.4 mll_t** emptyMemberStacks (mll_t ** memberStacks, int size)

After we have performed a round of convolution, this "empties" the member stacks by popping all nodes off each member linked list. Input: array of member linked lists, the size of that array (total number of offsets). Output: the array of now-empty member linked lists.

Definition at line 474 of file convll.c.

References popWholeMemStack().

Referenced by completeConv().

3.5.3.5 mll_t** fillMemberStacks (cll_t * head, mll_t ** memberStacks)

Fills the entire memberStacks data structure by calling addToStacks for each clique in the clique linked list. Input: head of a clique linked list, array of member linked lists. Output: the array of updated member linked lists.

Definition at line 455 of file convll.c.

References addToStacks(), and cnode::next.

Referenced by completeConv().

3.5.3.6 cll_t* initheadCll (cll_t * head, cSet_t * newset)

Initializes the empty head of a linked list by adding a set to that head. Note: this is only called immediately after pushing onto a cll, because the push always creates a new empty head. This function should not be called by the user; see pushcSet. Input: head of a linked list, pointer to a cSet_t list of clique members. Output: head of a linked list.

Definition at line 156 of file convll.c.

References cnode::set.

Referenced by pushcSet().

3.5.3.7 mll_t* mergeIntersect (cll_t * first, cll_t * second, mll_t * intersection, bitSet_t * printstatus, int * newSupport)

Convolves two cliques in a non-commutative manner. It finds which members of the first clique are immediately followed by a member in the second clique. Input: pointer to the location in the linked list of the first clique to be convolved, pointer to the location in the linked list of the second clique to be convolved, a member linked list used to store the intersection of the two cliques, the printstatus bitSet, and a pointer to an integer with the support of the clique formed by convolution. Output: a member linked list with the intersection of the two cliques, plus the side effect of that intersection's cardinality being stored in the integer pointed to by newSupport.

Definition at line 671 of file convll.c.

References cSet_t::members, pushMemStack(), cnode::set, and cSet_t::size.

Referenced by singleCliqueConv().

3.5.3.8 cSet_t* mllToCSet (mll_t * clique)

Turns a member linked list used to store the intersection of two cliques into something more useful: a cSet_t structure. Input: a clique in mll_t form. Output: a clique in cSet_t form.

Definition at line 1022 of file convll.c.

References mnode::cliqueMembership, cSet_t::members, mnode::next, and cSet_t::size.

Referenced by pushConvClique().

3.5.3.9 cll_t* popAllCll (cll_t * head)

Shortcut function to pop all of the members of a linked list. Input: head of a linked list. Output: head of a now-empty linked list.

Definition at line 101 of file convll.c.

References popCll().

Referenced by completeConv(), and main().

3.5.3.10 $cll_t*popCll(cll_t*head)$

Removes the head of the clique linked list, returns the new head of the clique linked list, and frees the memory occupied by the old head. Input: head of a linked list. Output: head of a linked list.

Definition at line 60 of file convll.c.

References cSet t::members, cnode::next, and cnode::set.

Referenced by popAllCll().

3.5.3.11 mll $t*popMemStack (mll \ t*head)$

Pops the head off of a single member linked list. Input: head of a member linked list. Output: the new head of a member linked list after popping one item.

Definition at line 388 of file convll.c.

References mnode::next.

Referenced by popWholeMemStack().

3.5.3.12 mll_t* popWholeMemStack (mll_t * head)

Pops all items off of a member linked list. Input: head of a member linked list. Output: empty head of a member linked list.

Definition at line 410 of file convll.c.

References popMemStack().

Referenced by emptyMemberStacks(), and singleCliqueConv().

3.5.3.13 int printCll ($cll_t * head$)

Prints the members (cliques) of a linked list in the format: id = unique id number of clique within linked list; Length = number of members of clique, if available; Size = length of each member of clique; Members = newline-separated list of members of the clique. Input: head of a linked list. Output: Gives text output, returns (meaningless) exit value.

Definition at line 118 of file convll.c.

References cnode::id, cnode::length, cSet_t::members, cnode::next, cnode::set, and c-Set_t::size.

3.5.3.14 int printCllPattern (cll_t * node, int length)

Prints out the contents of a clique linked list node in this format: support = number of motif occurrences (id = some id number); members = newline-separated list of offsets. Input: a specific node to be output, the length of the motif inside it. Output: text per above, and an integer success value.

Definition at line 1328 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

3.5.3.15 void printMemberStacks (mll_t ** memberStacks, int size)

Prints the contents of the member stacks. Input: array of member linked lists, size of that array (total number of offsets). Output: only text output/no return value.

Definition at line 491 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

3.5.3.16 cll_t* pushClique (bitSet_t * clique, cll_t * head, int * indexToSeq, int p)

Pushes a bitSet onto a clique linked list, performing all necessary manipulations in order to do so. Input: new clique in the form of a bitSet_t, head of a linked list, pointer to the index/sequence number data structure, integer value of the -p flag. Output: head of an updated clique linked list.

Definition at line 314 of file convll.c.

References bitSetToCSet(), checkCliquecSet(), cliquecounter, cSet_t::members, and pushcSet().

Referenced by findCliques(), and singleLinkage().

3.5.3.17 $cll_t* pushCll(cll_t* head)$

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to initheadCll, as the head pushed on here is empty and will be meaningless without any members. This function should NOT be used by the user; see pushcSet. Input: head of a linked list. Output: head of a linked list.

Definition at line 26 of file convll.c.

References cnode::id, cnode::length, cnode::next, cnode::set, and cnode::stat.

Referenced by pushcSet().

3.5.3.18 cll_t* pushConvClique (mll_t * clique, cll_t * head)

Pushes a freshly-convolved clique, currently in mll_t form, onto the clique linked list for the next level. Also checks to make sure that the convolved clique is unique, and if it isn't, it takes appropriate action. Input: a convolved clique in mll_t form, the head of a clique linked list for the next level. Output: (potentially new) head of the clique linked list for the next level.

Definition at line 980 of file convll.c.

References cSet_t::members, mllToCSet(), pushcSet(), removeSupers(), swapNodecSet(), and uniqClique().

Referenced by singleCliqueConv().

3.5.3.19 cll t* pushcSet (cll t* head, cSet t* newset)

Function that pushes the contents of a cSet (set of members of a clique) onto a linked list of cliques. Input: head of a linked list, new clique in the form of a cSet_t. Output: head of a linked list.

Definition at line 174 of file convll.c.

References initheadCll(), and pushCll().

Referenced by pushClique(), and pushConvClique().

3.5.3.20 mll t* pushMemStack (mll t * head, int cliqueNum)

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 358 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

3.5.3.21 cll_t* removeSupers (cll_t * head, int node, cSet_t * newClique)

This function finds all cliques in a linked list of which the proposed clique is a superset. It starts looking AFTER the first clique which has already been found to be a subset. In some senses, it is just a continuation of the uniquique function in order to take advantage of the fact that though a proposed clique can only be a subset of one existing next-level clique, it can be a superset of many existing next-level cliques. Input: head of a clique linked list, the id of the first node found to be a subset of the proposed clique, and the proposed clique (in cSet_t form). Output: the head of the clique linked list with all but the first subset (which was passed as an argument) removed. This function is now ready for swapNode to be called.

Definition at line 849 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

3.5.3.22 bitSet_t* searchMemsWithList (int * list, int listsize, mll_t ** memList, int numOffsets, bitSet_t * queue)

Creates one large queue by calling "setStackTrue" for each member of a list of offsets. This then creates the union of clique membership for all offsets in the list being searched. Input: an array of offset numbers, the length of that array, an array of member linked lists, the length of that array (the total number of offsets), and a bitSet_t to store the union/queue. Output: the union/queue in a bitSet_t structure.

Definition at line 540 of file convll.c.

References emptySet(), and setStackTrue().

Referenced by wholeCliqueConv().

3.5.3.23 bitSet_t* setStackTrue (mll_t ** memList, int i, bitSet_t * queue)

Adds all of the members of a given stack to a "queue" in the form of a bitSet_t data structure. That is, for each clique in the member linked list, it sets the corresponding bit in the bitSet_t true. Input: array of member linked lists, an integer indicating a specific member linked list, and a bitSet_t of length >= the number of cliques in the current clique linked list. Ouput: the updated bitSet_t object.

Definition at line 516 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

3.5.3.24 cll_t* singleCliqueConv (cll_t * head, int firstClique, cll_t ** firstGuess, int secondClique, cll_t ** secondGuess, cll_t * nextPhase, bitSet_t * printStatus, int support)

Convolves one single clique against one other single clique. Note that this is non-commutative, so exchanging firstClique and secondClique will not give the same results. The "guess" pointers keep the location of the previous clique in the linked list so that we don't have to search the linked list from the beginning/end every time. We exploit our earlier tidiness in that we can reasonably guess that we will monotonically traverse down cliques. Input: head of the current clique linked list, the id number of the first clique, a pointer to a guess at the first clique, the id number of the second clique, a pointer to a guess at the second clique, the head of the clique linked list for the next round of convolution, a bitSet indicating which cliques should be output as maximal, and the minimum support flag. Output: the head of clique linked list for the next round of convolution (which may have changed if the two cliques could be convolved).

Definition at line 580 of file convll.c.

References cnode::id, mergeIntersect(), cnode::next, popWholeMemStack(), push-ConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

3.5.3.25 cll_t* swapNodecSet (cll_t * head, int node, cSet_t * newClique)

Swaps out a node in a linked list that has been found to be a subset of a node that is not yet in the list. Input: the head of a clique linked list, a specific node within that linked list that is to be removed, and the new clique that is the superset of the node to be removed (in cSet_t form). Output: the head of the altered clique linked list.

Definition at line 804 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, and cnode::set.

Referenced by pushConvClique().

3.5.3.26 int uniqClique (cSet_t * cliquecSet, cll_t * head)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in cSet_t form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset. Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be

both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 729 of file convll.c.

References cnode::id, cSet t::members, cnode::next, cnode::set, and cSet t::size.

Referenced by pushConvClique().

```
3.5.3.27 cll_t* wholeCliqueConv (cll_t * head, cll_t * node, cll_t ** firstGuess, mll_t ** memList, int numOffsets, cll_t * nextPhase, bitSet_t * printStatus, int support)
```

Convolves one single clique against all possible cliques that could possibly be convolved. It does not attempt to convolve all other cliques, but prunes that set by first looking at the offsets that are in the clique, then collecting all of the cliques who have members that are one greater than the offsets in this clique, and then convolving those cliques in a sort of "queue" using the bitSet_t data structure. Input: the head of the clique linked list for the current level, the current node being convolved against in the linked list, the location of the previous node in the form of a pointer to a "guess", an array of member linked lists, the length of that array, the head of the clique linked list for the next level, a bitSet_t for the printStatus of maximality, and the support criterion. Output: the head of the (possibly modified) clique linked list for the next level.

Definition at line 1081 of file convll.c.

 $References\ bitSetToCSet(),\ countSet(),\ deleteBitSet(),\ cnode::id,\ cSet_t::members, newBitSet(),\ searchMemsWithList(),\ cnode::set,\ singleCliqueConv(),\ and\ cSet_t::size.$

Referenced by wholeRoundConv().

```
3.5.3.28 cll_t* wholeRoundConv (cll_t ** head, mll_t ** memList, int numOffsets, int support, int length, cll_t ** allCliques)
```

Performs convolution on all cliques in a linked list by repeatedly calling wholeClique-Conv. Input: pointer to the head of a clique linked list for the current level, array of member linked lists, length of that array, minimum support threshold, the current length of motifs, and a pointer to a linked list containing all cliques that will be printed out. Output: the head of the clique linked list for the next level of convolution.

Definition at line 1148 of file convll.c.

References checkBit(), deleteBitSet(), fillSet(), cnode::id, newBitSet(), cnode::next, wholeCliqueConv(), and yankCll().

Referenced by completeConv().

3.5.3.29 int yankCll (cll_t ** head, cll_t * prev, cll_t ** curr, cll_t ** allCliques, int length)

Removes a clique from within a linked list in order to save it for later printing. This is done so that the cliques are not printed as they are convolved, but rather after all rounds of convolution are complete. Input: a pointer to the head of the current linked list, the clique prior to the one that is to be yanked (NULL if the clique to be yanked is the head), the clique that is to be yanked, a pointer to the head of the list with all cliques that are to be printed, and the length of the current motif. Output: Nothing is returned beyond a success integer, but it alters the current level cll_t, the value of curr, and the linked list of all cliques that are to be printed.

Definition at line 1221 of file convll.c.

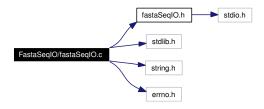
References cnode::id, and cnode::next.

Referenced by convolve(), and wholeRoundConv().

3.6 FastaSeqIO/fastaSeqIO.c File Reference

```
#include "fastaSeqIO.h"
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

Include dependency graph for fastaSeqIO.c:



Data Structures

• struct sSize t

Defines

- #define **BUFFER** 100000
- #define BIG_BUFFER 1000000

Functions

- int printFSeqSubSeq (fSeq_t *seq, int start, int stop)
- long measureLine (FILE *INPUT)
- long CountFSeqs (FILE *INPUT)
- long countLines (FILE *INPUT)
- int initAofFSeqs (fSeq_t *aos, int numSeq)
- char ** ReadFile (FILE *INPUT, int *n)
- fSeq_t * ReadTxtSeqs (FILE *INPUT, int *numberOfSequences)
- fSeq_t * ReadFSeqs (FILE *INPUT, int *numberOfSequences)
- int FreeFSeqs (fSeq_t *arrayOfSequences, int numberOfSequences)
- int WriteFSeqA (FILE *MY_FILE, fSeq_t *arrayOfSequences, int start, int stop)

3.6.1 Define Documentation

3.6.1.1 #define BIG_BUFFER 1000000

Definition at line 11 of file fastaSeqIO.c.

3.6.1.2 #define BUFFER 100000

Definition at line 10 of file fastaSeqIO.c.

3.6.2 Function Documentation

3.6.2.1 long CountFSeqs (FILE * INPUT)

Definition at line 44 of file fastaSeqIO.c.

```
45 {
46
       long start;
47
       long count = 0;
48
      int myChar;
49
      int newLine = 1;
50
      start = ftell(INPUT);
51
      myChar = fgetc(INPUT);
      while (myChar != EOF) {
52
53
           if (newLine == 1 && myChar == '>') {
54
               count++;
55
           if (myChar == '\n') {
56
57
              newLine = 1;
58
           } else {
59
               newLine = 0;
60
           myChar = fgetc(INPUT);
61
62
63
       fseek(INPUT, start, SEEK_SET);
64
       return count;
65 }
```

3.6.2.2 long countLines (FILE * *INPUT*)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

```
70 {
71    long start;
72    long count = 1;
73    int myChar;
```

```
74
       int status = 0;
75
       start = ftell(INPUT);
76
      myChar = fgetc(INPUT);
77
       while (myChar != EOF) {
78
           if (myChar == '\n') {
79
               count++;
80
               status = 1;
81
           } else {
82
               status = 0;
83
           }
           myChar = fgetc(INPUT);
84
85
86
       if (status == 1) {
87
           count--;
88
89
       fseek(INPUT, start, SEEK_SET);
90
       return count;
91 }
```

3.6.2.3 int FreeFSeqs (fSeq_t * arrayOfSequences, int numberOfSequences)

Definition at line 304 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by main().

```
305 {
306
        int i;
307
        for (i = 0; i < numberOfSequences; i++) {</pre>
            if (arrayOfSequences[i].label != NULL) {
308
309
                free(arrayOfSequences[i].label);
310
            arrayOfSequences[i].label = NULL;
311
312
            if (arrayOfSequences[i].seq != NULL) {
313
314
                free(arrayOfSequences[i].seq);
315
            arrayOfSequences[i].seq = NULL;
316
317
318
        if (arrayOfSequences != NULL) {
319
            free(arrayOfSequences);
320
321
        arrayOfSequences = NULL;
322
        return EXIT_SUCCESS;
323 }
```

3.6.2.4 int initAofFSeqs ($fSeq_t * aos$, int numSeq)

Definition at line 94 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by ReadFSeqs(), and ReadTxtSeqs().

```
95 {
96     int i;
97     for (i = 0; i < numSeq; i++) {
98         aos[i].seq = NULL;
99         aos[i].label = NULL;
100     }
101     return 1;
102 }</pre>
```

3.6.2.5 long measureLine (FILE * INPUT)

Definition at line 25 of file fastaSeqIO.c.

Referenced by ReadFile().

```
26 {
2.7
      long start;
28
      long count = 0;
29
      int myChar;
      start = ftell(INPUT);
30
31
      myChar = fgetc(INPUT);
32
      count++;
33
      while (myChar != '\n' && myChar != EOF) {
34
          count++;
35
          myChar = fgetc(INPUT);
36
37
      fseek(INPUT, start, SEEK_SET);
38
      return count;
39 }
```

3.6.2.6 int printFSeqSubSeq (fSeq_t * seq, int start, int stop)

Definition at line 14 of file fastaSeqIO.c.

References fSeq_t::seq.

3.6.2.7 char** ReadFile (FILE * INPUT, int * n)

Definition at line 105 of file fastaSeqIO.c.

References countLines(), and measureLine().

Referenced by ReadFSeqs(), readRealData(), and ReadTxtSeqs().

```
106 {
107
        char **buf = NULL;
108
        long nl;
109
        long tls = 0;
110
        int i=0;
111
112
        nl = countLines(INPUT);
113
        if(nl == 0){
114
            fprintf(stderr, "\nNo sequences! Error!\n\n");
115
            fflush(stderr);
116
            return NULL;
117
        buf = (char **) malloc ( (int)(nl+1) * sizeof(char *));
118
119
        if ( buf == NULL) {
120
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
121
            fflush(stderr);
122
            exit(0);
123
        }
124
        // measure the first line
125
126
        tls = measureLine(INPUT) + 1;
127
        if(tls != 0){
128
            buf[i] = (char *) malloc ( tls * sizeof(char));
129
            if ( buf[i] == NULL) {
                fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
130
131
                fflush(stderr);
132
                exit(0);
133
            }
134
        fgets(buf[i], tls, INPUT);
135
136
        do{
            if(buf[i][ strlen(buf[i])-1 ] == '\n'){
137
138
                buf[i][ strlen(buf[i])-1 ] = ' \0';
139
            tls = measureLine(INPUT) + 1;
140
141
            if(tls != 0){
142
                i++;
143
                buf[i] = (char *) malloc ( tls * sizeof(char) );
144
                if ( buf[i] == NULL){
145
                    fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
146
                     fflush(stderr);
147
                    exit(0);
148
                }
            }
149
        }while( fgets(buf[i], tls, INPUT) != NULL );
150
151
        free(buf[i]);
        buf = (char **) realloc ( buf, i * sizeof(char *) );
152
153
        if ( buf == NULL) {
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
154
```

```
155
             fflush(stderr);
156
             return NULL:
157
        // I think that 'i' might actually be the \mbox{\tt\#} of lines
158
        // plus one here? somehow line 131 isn't being freed,
159
160
        \ensuremath{//} or at least 2 bytes of it.
161
        *n = i;
162
        return buf;
163 }
```

3.6.2.8 **fSeq_t*** ReadFSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 199 of file fastaSeqIO.c.

References initAofFSeqs(), fSeq_t::label, ReadFile(), fSeq_t::seq, sSize_t::size, s-Size t::start, and sSize t::stop.

Referenced by main().

```
{
199
200
        int i,j,k;
201
        int nl, ns=0;
        char **buf = NULL;
202
203
        fSeq_t *aos;
        sSize_t *ss;
204
205
        sSize_t *11;
206
207
        buf = ReadFile(INPUT, &nl);
208
        if(buf == NULL){
            return NULL;
209
210
211
        // Count how many sequences we have
212
213
        for( j=0 ; j<n1 ; j++){
            if(buf[j][0] == '>'){
214
215
                ns++;
216
            }
217
        }
218
        ss = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
        if(ss == NULL){
219
220
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
221
            fflush(stderr);
222
            exit(0);
223
224
        11 = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
225
        if(11 == NULL){
226
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
227
            fflush(stderr);
228
            exit(0);
229
        }
230
231
        \ensuremath{//} find the first sequence
232
        k=0;
233
        while( buf[k][0] != '>'){}
```

```
234
            k++;
235
        }
236
237
        // record how large each sequence is
238
        i = -1;
        for( j=k ; j<nl ; j++){
239
240
            if(buf[j][0] == '>'){
241
                i++;
242
                ll[i].start = j;
243
                ll[i].stop = j;
                ll[i].size = strlen( buf[j] );;
244
245
                ss[i].start = j+1;
246
                ss[i].size = 0;
247
            }else{
248
                ss[i].stop = j;
                ss[i].size += strlen( buf[j] );;
249
250
            }
251
        }
252
253
        aos = (fSeq_t *) malloc ( ns * sizeof(fSeq_t));
254
        if( aos == NULL){
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
255
256
            fflush(stderr);
257
            exit(0);
258
        initAofFSeqs(aos, ns);
259
260
261
        for ( i=0 ; i<ns ; i++ ){
262
            if( ll[i].size > 0 ){
263
                aos[i].label = (char *) malloc ( (ll[i].size+1) * sizeof(char) );
264
                if( aos[i].label == NULL){
265
                     fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
266
                     fflush(stderr);
267
                     exit(0);
268
                aos[i].label[0] = ' \setminus 0';
269
270
                for ( j=ll[i].start ; j<=ll[i].stop ; j++ ){</pre>
271
272
                     // both instances of streat here are using
273
                         .label/.seq's that are NULL and that is
274
                     // throwing a memory error in valgrind
275
                     aos[i].label = strcat ( aos[i].label, buf[j] );
                }
276
277
            if( ss[i].size > 0 ){
278
                aos[i].seq = (char *) malloc ( (ss[i].size+1) * sizeof(char) );
279
280
                if( aos[i].seq == NULL){
281
                     fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
282
                     fflush(stderr);
283
                     exit(0);
284
285
                aos[i].seq[0] = '\0';
286
                for ( j=ss[i].start ; j<=ss[i].stop ; j++ ){</pre>
287
                     aos[i].seq = strcat ( aos[i].seq, buf[j] );
288
289
        }
290
```

```
291
        free(11);
292
        free(ss);
293
294
        for ( i=0 ; i<nl ; i++ ){
295
            free(buf[i]);
296
297
        free(buf);
298
299
        *numberOfSequences = ns;
300
        return aos;
301 }
```

3.6.2.9 **fSeq_t*** ReadTxtSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 172 of file fastaSeqIO.c.

References initAofFSeqs(), ReadFile(), and fSeq_t::seq.

```
172
173
        int i;
174
        int nl;
        char **buf = NULL;
175
176
        fSeq_t *aos;
177
        buf = ReadFile(INPUT, &nl);
178
179
        if(buf == NULL) {
180
           return NULL;
181
        aos = (fSeq_t *) malloc ( nl * sizeof(fSeq_t));
182
       if( aos == NULL) {
183
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
184
185
            fflush(stderr);
186
            exit(0);
187
188
       initAofFSeqs(aos, nl);
189
       for ( i=0 ; i<nl ; i++ ){
190
            aos[i].seq = buf[i];
191
192
       free(buf);
193
        *numberOfSequences = nl;
194
        return (aos);
195 }
```

3.6.2.10 int WriteFSeqA (FILE * MY_FILE, fSeq_t * arrayOfSequences, int start, int stop)

Definition at line 330 of file fastaSeqIO.c.

```
331 {
332 int i;
```

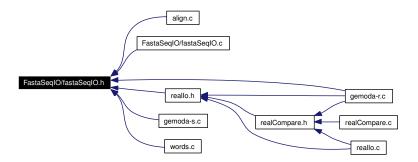
3.7 FastaSeqIO/fastaSeqIO.h File Reference

#include <stdio.h>

Include dependency graph for fastaSeqIO.h:



This graph shows which files directly or indirectly include this file:



Data Structures

• struct fSeq_t

Functions

- int printFSeqSubSeq (fSeq_t *seq, int start, int stop)
- long measureLine (FILE *INPUT)
- long countLines (FILE *INPUT)
- long CountFSeqs (FILE *INPUT)
- int initAofFSeqs (fSeq_t *aos, int numSeq)
- fSeq_t * ReadFSeqs (FILE *INPUT, int *numberOfSequences)
- int FreeFSeqs (fSeq_t *arrayOfSequences, int numberOfSequences)
- int WriteFSeqA (FILE *MY_FILE, fSeq_t *arrayOfSequences, int start, int stop)
- fSeq_t * ReadTxtSeqs (FILE *INPUT, int *numberOfSequences)

3.7.1 Function Documentation

3.7.1.1 long CountFSeqs (FILE * INPUT)

Definition at line 44 of file fastaSeqIO.c.

3.7.1.2 long countLines (FILE * INPUT)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

3.7.1.3 int FreeFSeqs (fSeq_t * arrayOfSequences, int numberOfSequences)

Definition at line 306 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by main().

3.7.1.4 int initAofFSeqs (fSeq_t * aos, int numSeq)

Definition at line 94 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by ReadFSeqs(), and ReadTxtSeqs().

3.7.1.5 long measureLine (FILE * *INPUT*)

Definition at line 25 of file fastaSeqIO.c.

Referenced by ReadFile().

3.7.1.6 int printFSeqSubSeq (fSeq_t * seq, int start, int stop)

Definition at line 14 of file fastaSeqIO.c.

References fSeq_t::seq.

3.7.1.7 **fSeq_t*** ReadFSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 199 of file fastaSeqIO.c.

References initAofFSeqs(), fSeq_t::label, ReadFile(), fSeq_t::seq, sSize_t::size, s-Size_t::start, and sSize_t::stop.

Referenced by main().

3.7.1.8 **fSeq_t*** ReadTxtSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 172 of file fastaSeqIO.c.

References initAofFSeqs(), ReadFile(), and fSeq_t::seq.

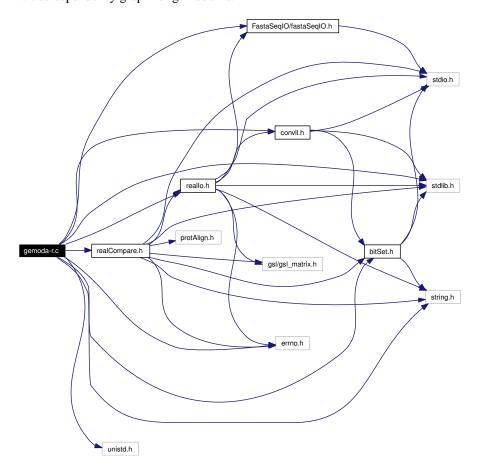
3.7.1.9 int WriteFSeqA (FILE * MY_FILE, fSeq_t * arrayOfSequences, int start, int stop)

Definition at line 332 of file fastaSeqIO.c.

3.8 gemoda-r.c File Reference

```
#include "bitSet.h"
#include "convll.h"
#include "FastaSeqIO/fastaSeqIO.h"
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include "realIo.h"
#include "realCompare.h"
```

Include dependency graph for gemoda-r.c:



Functions

- void usage (char **argv)
- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetTo-Index, int numOfSeqs, int p)
- int countExtraParams (char *s)
- double * parseExtraParams (char *s, int numParams)
- int main (int argc, char **argv)

3.8.1 Detailed Description

This file contains the main routine for the real valued version of Gemoda. There are also some accessory functions for printing information on how to use Gemoda and run it from the commandline.

Definition in file gemoda-r.c.

3.8.2 Function Documentation

3.8.2.1 cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberOfSequences, int noConvolve, FILE * OUTPUT_FILE)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

Referenced by main().

```
629 {
630    bitSet_t * cand = NULL;
631    bitSet_t * mask = NULL;
632    bitSet_t * Q = NULL;
633    int size = bg->size;
634    cll_t * elemPats = NULL;
```

```
635
      cll_t * allCliques = NULL;
     cll_t * curr = NULL;
636
637
638
        // contains indices (rows) containing the threshold value.
639
       cand = newBitSet (size);
640
     mask = newBitSet (size);
     Q = newBitSet (size);
641
642
     fillSet (cand);
     fillSet (mask);
643
        // Note that we prune based on p before setting the diagonal false.
645
646
        if (p > 1)
647
        {
         bg =
648
649
        pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650
651
652
        \ensuremath{//} 
 Now we set the main diagonal false for clustering and filtering.
        bitGraphSetFalseDiagonal (bg);
653
654
      filterGraph (bg, support, R);
      fprintf (OUTPUT_FILE, "Graph filtered! Now clustering...\n");
655
656
      fflush (NULL);
657
      if (clusterMethod == 0)
658
659
          findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660
661
      else
662
663
          singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664
                  p);
665
666
      fprintf (OUTPUT_FILE,
            "Clusters found! Now filtering clusters (if option set)...\n");
667
668
      fflush (NULL);
669
      if (p > 1)
670
       {
671
          elemPats = pruneCll (elemPats, indexToSeq, p);
672
673
      deleteBitSet (cand);
674
      deleteBitSet (mask);
675
      deleteBitSet (Q);
676
677
        // Now let's convolve what we made.
678
        if (noConvolve == 0)
679
680
          fprintf (OUTPUT_FILE, "Now convolving...\n");
681
          fflush (NULL);
682
          allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683
684
685
      else
686
       {
687
          curr = elemPats;
688
          while (curr != NULL)
689
690
          yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691
```

```
692  }
693  return allCliques;
694 }
```

3.8.2.2 int countExtraParams (char *s)

Definition at line 91 of file gemoda-r.c.

Referenced by main().

```
92 {
     int i = 0;
93
94
     int numParams = 1;
     for (i = 0; i < strlen(s); i++)
95
96
         if (s[i] == ',')
97
98
99
         numParams++;
100
101
102
      return numParams;
103 }
```

3.8.2.3 int main (int argc, char **argv)

This is the main routine of the real value Gemoda code. The code runs similarly to the sequence Gemoda code: there is a comparison phase, followed by a clustering phase, followed by a convolution phase. Only the comparison phase is unique to the real value Gemoda. Of course, since the data are formatted so differently, there are vastly different pieces of code in the front matter. In particular, there is no hashing of words obviously. As well, we use the GNU scientific library to store real value data as matrices that can be easily manipulated.

Definition at line 160 of file gemoda-r.c.

References calcStatAllCliqs(), convolve(), countExtraParams(), cumDMatrix(), delete-BitGraph(), freeD(), freeRdh(), getStatMat(), rdh_t::indexToSeq, rdh_t::offsetToIndex, outputRealPats(), outputRealPatsWCentroid(), parseExtraParams(), popAllCll(), read-RealData(), realComparison(), bitGraph_t::size, rdh_t::size, sortByStats(), and usage().

```
161 {
162
      int inputOption = 0;
163
      char *sequenceFile = NULL;
      FILE *SEQUENCE_FILE = NULL;
164
      char *outputFile = NULL;
165
166
      FILE *OUTPUT_FILE = NULL;
      int L = 0;
167
168
      int status = 0;
169
      double g = 0;
```

```
170
     int sup = 2;
171
     int R = 1;
172 int P = 0;
int compFunc = 0;
174
     double *extraParams = NULL;
175
     int numExtraParams = 0;
176
     int i = 0, j = 0;
177
178
       int j, k, i, l;
179
     int noConvolve = 0;
180
181
     int samp = 1;
182
     int supportDim = 0, lengthDim = 0;
     bitGraph_t *oam = NULL;
183
184
      unsigned int **d = NULL;
     int oamSize = 0;
185
186
      cll_t *allCliques = NULL;
187
188
189
      cll_t *curCliq = NULL;
190
191
     /*
192
      int curSeq;
193
      * /
194
195
       int curPos;
196
197
     int clusterMethod = 0;
198
     int joelOutput = 0;
199
200
     // gemoda-r new stuff
201
     rdh_t *data = NULL;
202
203
204
        Get command-line options
205
206
      while ((inputOption = getopt (argc, argv, "p:m:e:i:o:l:g:k:c:njs:")) != EOF)
207
       {
208
         switch (inputOption)
209
210
         // Comparison metric
211
        case 'm':
212
         compFunc = atoi (optarg);
213
         break;
         // Input file
214
215
        case 'i':
216
         sequenceFile = optarg;
217
         break;
218
         // Output file
        case 'o':
219
220
         outputFile =
221
            (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
222
          if (outputFile == NULL)
223
             fprintf (stderr, "Error allocating memory for options.\n");
224
225
              exit (EXIT_FAILURE);
226
```

```
227
         else
228
229
            strcpy (outputFile, optarg);
230
          }
231
        break;
232
        // Minimum motif length
233
       case '1':
234
        L = atoi (optarg);
235
        break;
236
        // Minimum motif similarity score
237
       case 'g':
238
        g = atof (optarg);
239
        status++;
240
        break;
241
        // Minimum support (number of motif occurrences)
       case 'k':
2.42
        sup = atoi (optarg);
243
244
        break;
245
246 /****************
248 * It takes all nodes with less than the minimum
249 *
       number of support and removes all of their nodes, and does this
250
       recursively so that nodes that are connected to many sparsely connected
251
       nodes will be removed and not left in the
\, 252 \, * This option is deprecated as it is at worst no-gain and at best useful.
253 *
       It will be on by default for clique-finding, but can be turned
254
       back off with some
       minor tweaking. For almost all cases in which it does not speed
255
256
       up computations, it will have a trivial time to perform. Thus, if
257
       clique-finding is turned on, then R is set to 1 by default.
258
          case 'r':
             R = 1;
259
260
              break;
262
        // Optional pruning parameter to require at motif occurrences
263
        // in at least P distinct input sequences
264
265
       case 'p':
        P = atoi (optarg);
266
267
        break;
268
        // Clustering method.
269
270
       case 'c':
271
        clusterMethod = atoi (optarg);
272
        break;
273
        // Extra parameters for comparison function
274
       case 'e':
275
        numExtraParams = countExtraParams (optarg);
276
        extraParams = parseExtraParams (optarg, numExtraParams);
277
        break;
278
       case 'n':
279
        noConvolve = 1;
280
        break;
       case 'j':
281
282
         joelOutput = 1;
283
        break;
```

```
284
       case 's':
285
         samp = atoi (optarg);
286
         break;
287
         // Catch-all.
288
        case '?':
289
         fprintf (stderr, "Unknown option '-%c'.\n", optopt);
290
         usage (argv);
291
         return EXIT_SUCCESS;
292
        default:
293
         usage (argv);
294
          return EXIT_SUCCESS;
295
296
297
     // Require an input file, a nonzero length, and a similarity threshold
298
     // to be set.
299
     if (sequenceFile == NULL | | L == 0 | | status < 1)
300
      {
301
          usage (argv);
302
         return EXIT_SUCCESS;
303
304
     // Open the sequence file
     if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
305
306
       {
307
          fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
308
              strerror (errno));
309
          exit (EXIT_FAILURE);
310
311
     // Open the output file
312
     if (outputFile != NULL)
313
       {
314
         if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
315
          fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
316
317
              strerror (errno));
318
          exit (EXIT_FAILURE);
319
320
321
      else
322
      {
         OUTPUT_FILE = stdout;
323
324
        }
325
326
327
328
     // Verbosity in output helps to distinguish output files.
     fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
329
     fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %f\n", L, sup, g);
330
331
     if (P > 1)
332
333
         fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
       }
334
     if (R > 0)
335
336
      {
337
          fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
338
339
     data = readRealData (SEQUENCE_FILE);
340
```

```
fclose (SEQUENCE_FILE);
      // printf("size = %d,indexSize = %d\n",data->size,data->indexSize);
342
      // printf("size1 = %d,size2 = %d\n",data->seq[0]->size1,data->seq[0]->size2);
344
      // for(i = 0; i < 2; i++) {
345
      // for(j = 0; j < 3; j++) {
346
      // printf("%lf,%lf,%lf\n",gsl_matrix_get(data->seq[i],j,0),
347
      // gsl_matrix_get(data->seq[i],j,1),
348
      // gsl_matrix_get(data->seq[i],j,2));}}
349
      oam = realComparison (data, L, g, compFunc, extraParams);
350
      // printf("oam->size = %d\n", oam->size);
351
      if ((samp > 0) && (clusterMethod == 0))
352
353
          // We are currently using one gap per sequence, as done in
354
          // realCompare.c's call to initRdhIndex in realComparison.
355
          // Note that this is data->size, NOT oam->size.
356
          d =
357
        getStatMat (oam, sup, L, &supportDim, &lengthDim, data->size, samp,
358
                OUTPUT_FILE);
359
        }
360
      else
361
        {
         d = NULL;
362
363
          supportDim = 0;
364
365
      allCliques =
366
367
        convolve (oam, sup, R, data->indexToSeq, P, clusterMethod,
368
              data->offsetToIndex, data->size, noConvolve, OUTPUT_FILE);
369
370
      oamSize = oam->size;
371
      // Do some early memory cleanup since this is so big.
372
      deleteBitGraph (oam);
373
374
      if ((samp > 0) && (clusterMethod == 0))
375
376
          cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize, data->size);
377
          calcStatAllCliqs (d, allCliques, oamSize - data->size);
378
          allCliques = sortByStats (allCliques);
379
380
381
      if (joelOutput == 0)
382
383
          outputRealPats (data, allCliques, L, OUTPUT_FILE, d);
384
385
      else
386
387
          outputRealPatsWCentroid (data, allCliques, L, OUTPUT_FILE, extraParams,
388
                       compFunc);
389
390
391
      freeD (d, supportDim);
392
      freeRdh (data);
393
      allCliques = popAllCll (allCliques);
394
      fclose (OUTPUT_FILE);
395
396
      return 0;
397 }
```

3.8.2.4 double* parseExtraParams (char * s, int numParams)

This was borrowed from the old gemoda-p code, there it used to parse filenames, here we are parsing comma-separated lists of doubles that are useful for SpecConnect.

Definition at line 110 of file gemoda-r.c.

Referenced by main().

```
111 {
112
      int i = 0, j = 0, k = 0;
113
      int startLength = 0;
114
      double *extraParams = NULL;
115
      char *paramString = NULL;
116
117
      extraParams = (double *) malloc (numParams * sizeof (double));
118
      if (extraParams == NULL)
119
        {
          fprintf (stderr, "Can't allocate extra params!\n");
120
121
          exit (0);
122
      j = 0;
123
      k = 0;
124
125
      startLength = strlen (s);
126
      for (i = 0; i < startLength; i++)</pre>
127
        {
128
          if (s[i] == ',')
129
        {
130
          // We've found an end. So point the pointer to
          // the beginning of the previous string.
131
132
          paramString = &s[k];
133
          // Terminate the string where the comma used to be
134
          s[i] = ' \setminus 0';
          // Update the location for the next string beginning
135
          k = i + 1;
          // Convert to a double and update the param number.
137
138
          extraParams[j] = atof (paramString);
139
140
141
142
      // Don't forget to do the last one, which isn't comma-terminated.
143
      paramString = &s[k];
144
      extraParams[j] = atof (paramString);
145
      return (extraParams);
146 }
```

3.8.2.5 bitGraph_t* pruneBitGraph (bitGraph_t * bg, int * indexToSeq, int ** offsetToIndex, int numOfSeqs, int p)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

Referenced by convolve().

```
404 {
405
      int i = 0, j = 0, nextBit = 0;
406
      int *seqNums = NULL;
408
        // Since we don't immediately know which node is in which source
409
        // sequence, we can't just count them up regularly. Instead, we'll
410
        // need to keep track of which sequences they come from and
        \ensuremath{//} increment \ensuremath{\texttt{\_something}}\xspace . What we chose to do here is just make
411
412
        // an array of integers of length = . Then, we try to put the
413
        // source sequence number of each neighbor (including itself, since
        // the main diagonal is still true at this time) into the next slot
415
        // Since we will monotonically search the bitSet, we can just
        // move on to the first bit in the next sequence using the
416
417
        // offsetToIndex structure so that we know the next sequence number
418
        // to be put in is always unique.
419
        seqNums = (int *) malloc (p * sizeof (int));
420
      if (seqNums == NULL)
421
422
          fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423
            strerror (errno));
424
          fflush (stderr);
425
          exit (0);
426
427
428
        // So, for each row in the bitgraph...
429
        for (i = 0; i < bg->size; i++)
430
431
432
        // Make sure the whole array is -1 sentinels.
433
        for (j = 0; j < p; j++)
434
435
          seqNums[j] = -1;
436
437
          j = 0;
438
        // Find the first neighbor of this bit.
439
440
        nextBit = nextBitBitSet (bg->graph[i], 0);
441
          if (nextBit == -1)
442
443
          continue;
444
445
          else
446
447
448
            // and put its sequence number in the array of ints.
449
            seqNums[0] = indexToSeq[nextBit];
```

```
450
451
452
        // If it's the last sequence, then bail out so that we don't
453
        // segfault in the next step.
454
        if (seqNums[0] >= numOfSeqs - 1)
455
456
          emptySet (bg->graph[i]);
457
          continue;
458
459
        // Find the next neighbor of this bit, STARTING AT the first
460
461
        // bit in the next sequence.
462
        nextBit =
        nextBitBitSet (bg->graph[i],
463
464
                   offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466
        // And iterate this until we run out of neighbors.
467
        while (nextBit >= 0)
468
        {
469
          j++;
470
          seqNums[j] = indexToSeq[nextBit];
471
            \ensuremath{//} Or until this new neighbor will fill up the array
472
473
            if (j == p - 1)
474
475
              break;
476
477
478
            // Or until this new neighbor is in the last sequence.
479
            if (seqNums[j] >= numOfSeqs - 1)
480
            {
481
              break;
            }
482
483
            // Get the next neighbor!
484
485
            nextBit =
486
            nextBitBitSet (bg->graph[i],
                   offsetToIndex[indexToSeq[nextBit] + 1][0]);
487
488
        }
489
490
        // If we didn't have enough unique sequences, and either a) we
491
        // were in the nth-to-last sequence and there were no
492
        // neighbors after it, or b) we were in the last sequence,
493
        // then the last number will still be our sentinel, -1. If
494
        // the last number is not a sentinel, then we have at least
495
        // p distinct sequence occurrences, so we're OK.
496
        if (seqNums[p - 1] == -1)
497
        {
498
          emptySet (bg->graph[i]);
499
500
501
      free (seqNums);
502
     return (bg);
503 }
```

3.8.2.6 void usage (char ** *argv*)

This function tells the user how to run Gemoda. The function displays all the available flags and gives an example of how to use the commandline to run the code.

Definition at line 35 of file gemoda-r.c.

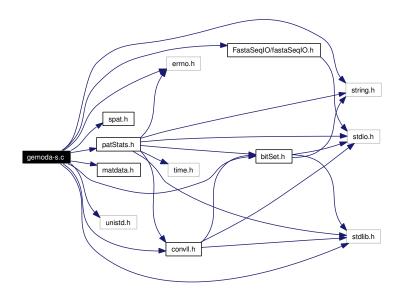
Referenced by main().

```
36 {
37
     fprintf (stdout,
          "Usage: %s -i <Fasta sequence file> "
38
39
          "-1 <word size> \n\t-k <support> -q <threshold> -m <matrix name> [-z] \n\t[-c <cluster method [(
40
          "Required flags and input:\n\n"
          "-i <Fasta sequence file>:\n\t"
41
42
          "File containing all sequences to be searched, in Fasta format.\n\n"
          "-l < word size > : \n\t"
43
44
          "Minimum length of motifs; also the sliding window length\n\t"
45
          "over which all motifs must meet the similarity criterion\n\"
46
          -k < support >: \n\t"
47
          "Minimum number of motif occurrences.\n\n"
          -g < threshold >: \n\t"
48
49
          "Similarity threshold. Two windows, when scored with the \n\t"
50
          " similarity matrix defined by the -m flag, must have at least\n\t"
51
          " this score in order to be deemed 'connected'. This criterion\n\t"
52
          " must be met over all sliding windows of length l.\n\"
          "-c <cluster method [0|1]>:\n\t"
53
54
          "The clustering method to be used after evaluating the "
55
          "\n\tsimilarity of the unique words in the input. Note that the "
56
          "\n\tclustering method will have a significant impact on both the "
57
          "\n\tresults that one obtains and the computation time.\n\t"
          "0: clique-finding\n\t\t"
58
59
          "Uses established methods to find all maximal cliques in the "
60
          \normalfont{\colored} "\n\t\tdata. This will give the most thorough results (that are "
61
          "\n\t\tprovably exhaustive), but will also give less-significant
          "\n\t\tresults in addition to the most interesting and most\n\t"
62
63
          "significant ones. The results are deterministic but may take some
64
          "\n\t\ttime on data sets with high similarity or if the similarity "
          \nt\tthreshold is set extremely low.\n\t"
65
66
          "1: single-linkage clustering\n\t\t"
          "Uses a single-linkage-type clustering where all nodes that "
67
68
          "\n\t\tare connected are put in the same cluster. This method is '
69
          "\n\t\talso deterministic and will be faster than clique-finding,
70
          "\n\t\tbut it loses guarantees of exhaustiveness in searching the
71
          \n "\n\t\tdata set.\n\n",
72
          "-p <unique support>:\n\t"
          "A pruning parameter that requires the motif to occur in "
73
74
          "\n\tat least <unique support> different input sequences. Note "
75
          "\n\tthat this parameter must be less than or equal to the total "
76
          "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
77
     fprintf (stdout, "\n");
78 }
```

3.9 gemoda-s.c File Reference

```
#include "bitSet.h"
#include "spat.h"
#include "convll.h"
#include "matdata.h"
#include "FastaSeqIO/fastaSeqIO.h"
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include "patStats.h"
```

Include dependency graph for gemoda-s.c:



Functions

- void usage (char **argv)
- void matrixlist (void)

- void getMatrixByName (char name[], int mat[][MATRIX_SIZE])
- bitGraph_t * alignWordsMat_bit (sPat_t *words, int wc, int mat[][MATRIX_-SIZE], int threshold)
- sPat t * countWords2 (fSeq t *seq, int numSeq, int L, int *numWords)
- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetTo-Index, int numOfSeqs, int p)
- int main (int argc, char **argv)

3.9.1 Detailed Description

This file houses the main routine for the sequence based Gemoda algorithm. In addition, there are a few helper functions which are used to inform the user how to run the software.

The Gemoda algorithm has three stages: comparison, clustering, and convolution. These three stages are called in serial from the main routine in this file.

Definition in file gemoda-s.c.

3.9.2 Function Documentation

```
3.9.2.1 bitGraph_t* alignWordsMat_bit (sPat_t * words, int wc, int mat[][MATRIX_SIZE], int threshold)
```

This uses the function above. Here, we have an array of words (sPat_t objects) and we compare (align) them all. If their score is above 'threshold' then we will set a bit to 'true' in a bitGraph_t that we create. A bitGraph_t is essentially an adjacency matrix, where each member of the matrix contains only a single bit: are the words equal, true or false? The function traverses the words by doing and all by all comparison; however, we only do the upper diagonal. The function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

References alignMat(), bitGraphSetTrueSym(), mat, and newBitGraph().

Referenced by main().

```
90 {
91    bitGraph_t * sg = NULL;
92    int score;
93    int i, j;
94
```

```
95
       // Assign a new bitGraph_t object, with (wc x wc) possible
96
       // true/false values
97
       sg = newBitGraph (wc);
98
     for (i = 0; i < wc; i++)
99
100
          for (j = i; j < wc; j++)
101
102
103
            // Get the score for the alignment of word i and word j
104
            score =
            alignMat (words[i].string, words[j].string, words[i].length, mat);
105
106
107
            // If that score is greater than threshold, set
108
            // a bit to 'true' in our bitGraph_t object
109
            if (score >= threshold)
110
111
            // We use 'bitGraphSetTrueSym' because, if i=j,
112
113
            // then j=i for most applications. However, this
114
            // can be relaxed for masochists.
115
            bitGraphSetTrueSym (sg, i, j);
116
            }
117
118
        }
119
120
        // Return a pointer to this new bitGraph_t object
121
        return sg;
122 }
```

3.9.2.2 cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberOfSequences, int noConvolve, FILE * OUTPUT_FILE)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References bitGraphSetFalseDiagonal(), completeConv(), deleteBitSet(), fillSet(), filterGraph(), findCliques(), newBitSet(), pruneBitGraph(), pruneCll(), single-Linkage(), bitGraph_t::size, and yankCll().

```
629 {
630 bitSet_t * cand = NULL;
```

```
631
      bitSet_t * mask = NULL;
     bitSet_t * Q = NULL;
632
     int size = bg->size;
634
      cll_t * elemPats = NULL;
635
      cll_t * allCliques = NULL;
636
      cll_t * curr = NULL;
637
638
       // contains indices (rows) containing the threshold value.
639
        cand = newBitSet (size);
640
      mask = newBitSet (size);
641
      Q = newBitSet (size);
642
     fillSet (cand);
643
      fillSet (mask);
644
645
        // Note that we prune based on p before setting the diagonal false.
646
        if (p > 1)
647
        {
         bg =
648
649
        pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650
651
652
        // Now we set the main diagonal false for clustering and filtering.
653
        bitGraphSetFalseDiagonal (bg);
654
      filterGraph (bg, support, R);
655
      fprintf (OUTPUT_FILE, "Graph filtered! Now clustering...\n");
      fflush (NULL);
656
657
      if (clusterMethod == 0)
658
659
         findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660
661
      else
662
       {
663
          singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664
                 p);
665
      fprintf (OUTPUT_FILE,
666
667
            "Clusters found! Now filtering clusters (if option set)...\n");
668
      fflush (NULL);
669
      if (p > 1)
670
        {
671
          elemPats = pruneCll (elemPats, indexToSeq, p);
672
      deleteBitSet (cand);
673
674
      deleteBitSet (mask);
675
      deleteBitSet (Q);
676
677
        // Now let's convolve what we made.
678
        if (noConvolve == 0)
679
680
          fprintf (OUTPUT_FILE, "Now convolving...\n");
681
          fflush (NULL);
682
          allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683
684
685
      else
686
          curr = elemPats;
687
```

```
688 while (curr != NULL)
689 {
690    yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691    }
692    }
693    return allCliques;
694 }
```

3.9.2.3 **sPat_t*** countWords2 ($\mathbf{fSeq_t} * seq$, int numSeq, int L, int * numWords)

Counts words of size L in the input FastA sequences, hashes all of the words, and returns an array of $sPat_t$ objects.

Definition at line 373 of file words.c.

References sHashEntry_t::data, destroySHash(), sHashEntry_t::data, initSHash(), s-HashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, s-Offset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

Referenced by main().

```
374 {
375
      int i, j;
376
     int totalChars = 0;
377
     int hashSize;
378
     sHashEntry_t newEntry;
379
     sHashEntry_t *ep;
     sHash_t wordHash;
380
     sPat_t *words = NULL;
381
     int wc = 0;
382
383
     int prev = -1;
384
     int 1;
385
386
387
      // Count the total number of characters. This
388
      // is the upper limit on how many words we can have
389
      for (i = 0; i < numSeq; i++)
390
391
          totalChars += strlen (seq[i].seq);
392
393
394
      // Get a prime number for the size of the hash table
395
      hashSize = sieve3 ((long) (2 * totalChars));
396
      wordHash = initSHash (hashSize);
397
398
      // Chop up each sequence and hash out the words of size {\tt L}
399
      for (i = 0; i < numSeq; i++)
400
401
          prev = -1;
402
403
          // skip sequences that are too short to have
404
          // a pattern
          if (strlen (seq[i].seq) < L)</pre>
405
```

```
406
        {
407
          continue;
408
409
          for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
410
411
412
          // Make a hash table entry for this word
413
         newEntry.key = &(seq[i].seq[j]);
414
         newEntry.data = 1;
415
          newEntry.idx = wc;
416
          newEntry.L = L;
417
418
          // Check to see if it's already in the hash table
419
          ep = searchSHash (&newEntry, &wordHash, 0);
420
          if (ep == NULL)
421
422
423
              // If it's not, create an entry for it
424
              ep = searchSHash (&newEntry, &wordHash, 1);
425
426
              // Increase the size of our word array
              words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
427
428
              if (words == NULL)
429
430
              fprintf (stderr, "Error!\n");
              fflush (stderr);
431
432
433
              // Add the new word
434
              words[wc].string = &(seq[i].seq[j]);
435
              words[wc].length = L;
436
              words[wc].support = 1;
437
              words[wc].offset =
            (sOffset_t *) malloc (1 * sizeof (sOffset_t));
438
439
              if (words[wc].offset == NULL)
440
441
              fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
442
              fflush (stderr);
443
              exit (0);
444
              words[wc].offset[0].seq = i;
445
446
              words[wc].offset[0].pos = j;
447
              words[wc].offset[0].prev = prev;
448
              words[wc].offset[0].next = -1;
449
              if (prev != -1)
450
451
452
              words[prev].offset[words[prev].support - 1].next = wc;
453
              prev = wc;
454
455
              WC++;
456
457
458
          else
459
            {
460
              // If it is, increase the count for this word
461
462
              ep->data++;
```

```
463
              // add a new offset to the word array
464
465
              1 = words[ep->idx].support;
              words[ep->idx].offset =
466
467
            (sOffset_t *) realloc (words[ep->idx].offset,
468
                            (l + 1) * sizeof (sOffset_t));
              words[ep->idx].offset[l].seq = i;
469
470
              words[ep->idx].offset[l].pos = j;
              words[ep->idx].offset[1].prev = prev;
471
472
              words[ep->idx].offset[l].next = -1;
473
474
              // Update the next/prev
475
              if (prev != -1)
476
477
              words[prev].offset[words[prev].support - 1].next = ep->idx;
478
479
              prev = ep->idx;
480
              // Have to put this down here for cases when we create
481
482
              // a word and it is immeadiately followed by itself!!
483
              words[ep->idx].support += 1;
            }
484
485
486
        }
487
488
489
     destroySHash (&wordHash);
490
      *numWords = wc;
491
     return words;
492 }
```

3.9.2.4 void getMatrixByName (char name[], int mat[][MATRIX_SIZE])

Referenced by main().

3.9.2.5 int main (int argc, char ** argv)

This is the main routine of the Gemoda source code. The routine performs basic operations such as parsing the input from the user and opening input files. Then, the function hashes words of length L. The unique words are aligned against each other to produce an adjacency matrix that says whether the unique word i is sufficiently similar, based on the user supplied threshold, to the unique word j. This adjacency matrix is then dereferenced into an adjacency matrix in which each index of the matrix represents a unique position in the input sequences, rather than a unique word. This dereferencing is required for the convolution stage. Finally, this adjacency matrix is convolved and the final motifs are returned as a linked list. The routine then closes all input and output files and frees up dynamically allocated memory.

Definition at line 187 of file gemoda-s.c.

References alignWordsMat_bit(), bitGraphCheckBit(), bitGraphSetTrueSym(), calc-StatAllCliqs(), convolve(), countWords2(), cumDMatrix(), deleteBitGraph(), Free-FSeqs(), getMatrixByName(), getStatMat(), cnode::length, mat, MATRIX_SIZE, matrixlist(), cSet_t::members, newBitGraph(), cnode::next, sPat_t::offset, popAllCll(), s-Offset_t::pos, ReadFSeqs(), sOffset_t::seq, cnode::set, cSet_t::size, bitGraph_t::size, sortByStats(), cnode::stat, and usage().

```
188 {
189
      int inputOption = 0;
      char *sequenceFile = NULL;
190
191
      char *outputFile = NULL;
192
      char *matName = NULL;
     FILE * SEQUENCE_FILE = NULL;
193
194
     FILE * OUTPUT_FILE = NULL;
195
     int L = 0;
196
      int numberOfSequences = 0;
197
      fSeq_t * mySequences = NULL;
     fSeq_t * (*seqReadFunct) () = &ReadFSeqs;
198
199
     sPat_t * words = NULL;
200
     int wc;
201
      int status = 0;
202
     int g = 0;
     int sup = 2;
203
204
     int R = 1;
205
     int P = 0;
206
      int (*mat)[MATRIX_SIZE] = NULL;
207
     int noConvolve = 0;
208
     int j, k, i, l;
209
     bitGraph_t * bg = NULL;
210
     bitGraph_t * oam = NULL;
211
       // new
212
     int **offsetToIndex = NULL;
213
     int *indexToSeq = NULL;
214
215
     int *indexToPos = NULL;
216
      int numberOfOffsets = 0;
217
     int pos1, pos2;
218
219
       // int *prevRowArray;
220
       sOffset_t * offset1, *offset2;
221
      cll_t * allCliques = NULL;
     cll_t * curCliq = NULL;
222
223
     int curSeq;
224
     int curPos;
225
      int clusterMethod = 0;
226
227
       // patStats
228
     int samp = 1;
      unsigned int **d = NULL;
229
230
      int supportDim = 0, lengthDim = 0;
231
      int oamSize = 0;
232
233
234
            Get command-line options
235
        while ((inputOption = getopt (argc, argv, "i:o:l:g:k:m:p:zc:ns:")) != EOF)
236
```

```
237
238
          switch (inputOption)
239
240
241
            // Input file
242
        case 'i':
243
         sequenceFile = optarg;
244
          seqReadFunct = &ReadFSeqs;
245
          break;
246
            // Output file
247
        case 'o':
248
249
          outputFile =
250
           (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
251
          if (outputFile == NULL)
252
              fprintf (stderr, "Error allocating memory for options.\n");
253
254
              exit (EXIT_FAILURE);
255
256
          else
257
           {
258
              strcpy (outputFile, optarg);
            }
259
260
          break;
261
           // Minimum motif length
262
263
        case 'l':
         L = atoi (optarg);
264
265
          break;
266
            // Minimum motif similarity score
267
268
        case 'g':
269
         g = atoi (optarg);
270
          status++;
271
          break;
272
273
            // Minimum support (number of motif occurrences)
274
        case 'k':
275
          sup = atoi (optarg);
276
          break;
277
278
            // Similarity matrix used to find similarity score
279
        case 'm':
280
          getMatrixByName (optarg, &mat);
          matName = (char *) malloc (strlen (optarg) * sizeof (char));
281
          if (matName == NULL)
282
283
              fprintf (stderr, "Error allocating memory for options.\n");
284
285
              exit (EXIT_FAILURE);
            }
286
287
          else
288
289
              strcpy (matName, optarg);
290
291
          break;
```

```
294 * Recursive initial pruning: an option for clique finding.
295 * It takes all nodes with less than the minimum
       number of support and removes all of their nodes, and does this
297 *
       recursively so that nodes that are connected to many sparsely connected
298 *
        nodes will be removed and not left in the
299 * This option is deprecated as it is at worst no-gain and at best useful.
300 *
       It will be on by default for clique-finding, but can be turned
301 *
       back off with some
       minor tweaking. For almost all cases in which it does not speed
302 *
303 *
        up computations, it will have a trivial time to perform. Thus, if
304 *
        clique-finding is turned on, then R is set to 1 by default.
305
           case 'r':
306
               R = 1;
307
              break;
// Optional pruning parameter to require at motif occurrences
309
310
           // in at least P distinct input sequences
311
       case 'p':
312
        P = atoi (optarg);
313
         break;
314
           // Clustering method.
315
316
       case 'c':
317
         clusterMethod = atoi (optarg);
318
         break;
       case 'n':
319
320
        noConvolve = 1;
321
         break;
322
       case 's':
323
         samp = atoi (optarg);
324
         break;
325
           // Catch-all.
326
327
       case '?':
328
         fprintf (stderr, "Unknown option '-%c'.\n", optopt);
329
         usage (argv);
330
        return EXIT_SUCCESS;
       case 'z':
331
         matrixlist ();
332
333
         return EXIT_SUCCESS;
334
       default:
335
         usage (argv);
336
         return EXIT_SUCCESS;
337
338
339
340
       // Require a similarity matrix
341
       if (mat == NULL)
342
343
         usage (argv);
344
         return EXIT_SUCCESS;
345
346
347
       // Require an input file, a nonzero length, and a similarity threshold
348
       // to be set.
349
       if (sequenceFile == NULL | L == 0 | status < 1)</pre>
350
```

```
351
          usage (argv);
         return EXIT_SUCCESS;
352
353
354
355
       // Open the sequence file
356
       if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
357
358
          fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
359
           strerror (errno));
360
          exit (EXIT_FAILURE);
361
362
363
        // Open the output file
364
       if (outputFile != NULL)
365
366
          if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
367
368
          fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
369
               strerror (errno));
370
          exit (EXIT_FAILURE);
371
372
       }
373
     else
374
       {
375
          OUTPUT_FILE = stdout;
376
377
378
       // Allocate some sequences
379
       mySequences = seqReadFunct (SEQUENCE_FILE, &numberOfSequences);
380
     if (mySequences == NULL)
381
382
          fprintf (stderr, "\nError reading your sequences/text.");
          fprintf (stderr, "\nCheck the format/size of the file.");
383
384
          fprintf (stderr, "\nERROR: %s\n", strerror (errno));
385
          return EXIT_FAILURE;
386
387
388
       // Close the input files
389
       fclose (SEQUENCE_FILE);
390
391
       // Verbosity in output helps to distinguish output files.
392
       fprintf (OUTPUT_FILE, "\nMatrix used = %s\n", matName);
     393
394
395
     if (P > 1)
396
       {
          fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
397
398
399
     if (R > 0)
400
       {
          fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
401
402
       }
403
404
       // Find the unique words in the input.
       words = countWords2 (mySequences, numberOfSequences, L, &wc);
405
406
        /*
407
```

```
408
           fprintf(stderr, "Counted %d words\n", wc);
409
410
        /*
411
           fflush(stderr);
412
413
        // Align the words that we just found by applying the similarity
414
415
        // matrix to each pair of them. Note that
416
        // bg is the adjacency matrix of words, but we
417
        // need an adjacency matrix of offsets instead.
418
        bg = alignWordsMat_bit (words, wc, mat, g);
419
      fprintf (OUTPUT_FILE, "\nAligned! Creating offset matrix...\n");
420
      fflush (NULL);
421
422
        // Create an intermediate translation matrix
        // to store the offset number of each sequence number/position.
423
424
        //
425
        // Note that this matrix is better called "Index to offset", and
426
        // the other matrices are better called "offset to Seq" and
427
        // "offset to Pos"
        offsetToIndex = (int **) malloc (numberOfSequences * sizeof (int *));
428
429
      if (offsetToIndex == NULL)
430
431
          fprintf (stderr,
432
            "Unable to allocate memory - offsetToIndex in gemoda.c\n%s\n",
            strerror (errno));
433
434
          fflush (stderr);
435
         exit (0);
436
437
      for (i = 0; i < numberOfSequences; i++)</pre>
438
439
        // MPS 5/23/05: Added in "-L+2" to make there only be one
440
441
        // blank between sequences.
442
        offsetToIndex[i] =
443
        malloc ((strlen (mySequences[i].seq) - L + 2) * sizeof (int));
444
          if (offsetToIndex[i] == NULL)
445
446
          fprintf (stderr,
                "Unable to allocate memory - offsetToIndex[%d] in gemoda.c\n%s\n",
447
448
                i, strerror (errno));
449
          fflush (stderr);
450
          exit (0);
451
452
453
        // MPS 5/23/05: Added in "-L+2" to make there only be one
454
        // blank between sequences.
455
        for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
456
457
          offsetToIndex[i][j] = numberOfOffsets;
458
          numberOfOffsets++;
459
460
461
462
        // Now create translation matrices such that we can get the sequence
463
        // or position number of a given offset.
        indexToSeq = (int *) malloc (numberOfOffsets * sizeof (int));
464
```

```
465
      if (indexToSeq == NULL)
466
467
          fprintf (stderr,
468
            "Unable to allocate memory - indexToSeq in gemoda.c\n%s\n",
469
            strerror (errno));
470
          fflush (stderr);
471
          exit (0);
472
473
      indexToPos = (int *) malloc (numberOfOffsets * sizeof (int));
474
      if (indexToPos == NULL)
475
476
          fprintf (stderr,
477
            "Unable to allocate memory - indexToPos in gemoda.c\n%s\n",
478
            strerror (errno));
479
          fflush (stderr);
          exit (0);
480
481
        }
482
     k = 0;
      for (i = 0; i < numberOfSequences; i++)</pre>
483
484
485
        // MPS 5/23/05: Added in "-L+2" to make there only be one
486
487
        // blank between sequences.
488
        for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
489
          indexToSeq[k] = i;
490
491
          indexToPos[k] = j;
492
         k++;
493
494
495
496
        // Now make an offset adjacency matrix!
497
        //
498
        oam = newBitGraph (numberOfOffsets);
499
500
        // Go through each unique word
501
        for (i = 0; i < wc; i++)
502
503
          offset1 = words[i].offset;
504
505
        // Go through each occurrence
506
        for (k = 0; k < words[i].support; k++)
507
        {
508
            // Use the offsetToIndex translation to get the offset
509
510
            // of the first occurrence
511
            pos1 = offsetToIndex[offset1[k].seq][offset1[k].pos];
512
513
            // And go through each word in the first offset to
514
            // find words that meet the similarity threshold
            for (j = 0; j < wc; j++)
515
516
517
              if (bitGraphCheckBit (bg, i, j))
518
            {
519
              offset2 = words[j].offset;
520
521
                // And find all of their occurrences,
```

```
522
                // using offsetToIndex to get the
523
                // offsets, and then setting those
                // locations in the offset adjacency
525
                // matrix true.
526
                for (1 = 0; 1 < words[j].support; 1++)</pre>
527
528
                  pos2 = offsetToIndex[offset2[1].seq][offset2[1].pos];
529
                  bitGraphSetTrueSym (oam, pos1, pos2);
530
531
532
533
534
      fprintf (OUTPUT_FILE, "Offset matrix created...");
535
      deleteBitGraph (bg);
      if ((samp > 0) \&\& (clusterMethod == 0))
537
538
539
          fprintf (OUTPUT_FILE, " taking preliminary statistics.\n");
540
          fflush (NULL);
541
542
        getStatMat (oam, sup, L, &supportDim, &lengthDim, numberOfSequences,
543
                samp, OUTPUT_FILE);
544
          fprintf (OUTPUT_FILE, "Now filtering...\n");
545
          fflush (NULL);
546
547
      else
548
          fprintf (OUTPUT_FILE, " now filtering.\n");
549
550
          fflush (NULL);
551
          d = NULL;
552
          supportDim = 0;
553
554
555
        // Now we're convolving on offsets
556
        allCliques =
557
        convolve (oam, sup, R, indexToSeq, P, clusterMethod, offsetToIndex,
558
              numberOfSequences, noConvolve, OUTPUT_FILE);
559
        // Do some early memory cleanup to limit usage
560
561
        oamSize = oam->size;
562
      deleteBitGraph (oam);
563
      fprintf (OUTPUT_FILE, "Convolved! Now making output...\n");
564
      fflush (NULL);
565
      if ((samp > 0) && (clusterMethod == 0))
566
567
          cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize,
568
               numberOfSequences);
          calcStatAllCliqs (d, allCliques, numberOfOffsets - numberOfSequences);
569
570
          allCliques = sortByStats (allCliques);
571
572
573
        \ensuremath{//} walk over the cliques and give some output in the format:
574
        // pattern <pattern id num>: len=<motif length> sup=<motif instances>
575
        // <sequence num> <position num> <motif instance>
576
577
        curCliq = allCliques;
578
```

```
579
        i = 0;
      while (curCliq != NULL)
580
581
582
          fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d", i,
583
            curCliq->length + L, curCliq->set->size);
584
          if (d != NULL)
585
586
          fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
587
        }
588
          else
589
        {
590
          fprintf (OUTPUT_FILE, "\n");
591
592
593
        for (j = 0; j < curCliq->set->size; j++)
594
595
          pos1 = curCliq->set->members[j];
596
          curSeq = indexToSeq[pos1];
          curPos = indexToPos[pos1];
597
          fprintf (OUTPUT_FILE, " %d\t%d\t", curSeq, curPos);
598
          for (k = curPos; k < curPos + curCliq->length + L; k++)
599
600
601
              fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]);
602
603
          fprintf (OUTPUT_FILE, "\n");
604
605
          fprintf (OUTPUT_FILE, "\n\n");
606
          curCliq = curCliq->next;
607
          i++;
608
609
610
        // And do some memory cleanup
611
        // And cleanup of probability stuff...
612
613
            free(letterfreqs); delete_augmented_matrix(augmat);
614
615
        allCliques = popAllCll (allCliques);
      free (indexToSeq);
616
617
      indexToSeq = NULL;
      free (indexToPos);
618
619
      indexToPos = NULL;
620
      for (i = 0; i < numberOfSequences; i++)</pre>
621
        {
622
          free (offsetToIndex[i]);
          offsetToIndex[i] = NULL;
623
624
625
626
        // Free'ing added by MPS, 6/4
627
        for (i = 0; i < wc; i++)
628
        {
629
          free (words[i].offset);
630
        }
631
     free (words);
632
        // End free'ing added by MPS
633
634
        free (offsetToIndex);
635
      offsetToIndex = NULL;
```

3.9.2.6 void matrixlist (void)

This function prints a list of the matrices that Gemoda can use to do the alignment of words. Most of these matrices are appropriate for amino acid sequences. In addition, there are matrices for DNA sequences and an identity matrix that is appropriate for other sequences, such as the analysis of English text. The matrix is selected using the -m flag.

Definition at line 99 of file gemoda-s.c.

Referenced by main().

```
100 {
101
     fprintf (stdout, "\nThe following similarity matrices are installed "
           "with the default Gemoda installation.\n Most of these "
102
            "were obtained from publically available BLAST distributions. \n\n"
103
104
            "dna_idmat:\n\t"
105
           "Identity matrix for DNA: returns 1 when A,C,G,T are "
106
            "compared to \n\tthemselves, 0 otherwise.\n\n"
107
           "identity_aa:\n\t"
108
           "Identity matrix for amino acids: returns 1 when any \n\t"
109
           "letter but J,O,U are compared to themselves, and 0 "
110
           "otherwise.\n\n" "idmat:\n\t"
           "Similar to identity_aa, but it returns 10 in place "
111
112
           "of 1.\n\" est_idmat:\n\"
113
           "Similar to idmat, but it returns -10 in place of 0. "
                                                                  "\n\n"
114
           "pam100:\n" "pam110:\n" "pam120:\n" "pam130:\n"
115
                        "pam190:\n"
           "pam140:\n"
                        "pam210:\n"
116
            "pam200:\n"
                                    "pam220:\n"
                                                 "pam230:\n"
                        "pam250:\n" "pam260:\n"
                                                 "pam280:\n"
            "pam240:\n"
117
           "pam290:\n"
118
                        "pam300:\n" "pam310:\n"
                                                 "pam320:\n"
                        119
           "pam330:\n"
                                                 "pam370:\n"
120
            "pam380:\n"
                        "pam390:\n"
                                    "pam400:\n"
                                                 "pam430:\n"
                        121
            "pam440:\n"
                                                 "pam490:\n"
           "pam500:\n\t"
122
           "PAM matrices for various evolutionary distances.\n\n"
123
           "blosum30:\n" \quad "blosum35:\n" \quad "blosum40:\n"
124
                                                      "blosum45:\n"
125
            "blosum50:\n"
                         "blosum55:\n"
                                         "blosum60:\n"
                                                       "blosum62:\n"
           "blosum65:\n" "blosum70:\n"
                                        "blosum75:\n" | blosum80:\n"
126
           "blosum85:\n"    "blosum90:\n"    "blosum100:\n\t"
127
128
           "BLOSUM matrices for various evolutionary distances.\n\n"
129
           "blosumn:\n\t"
                           "BLOSUM matrix of unknown origin.\n\n"
            "dayhoff:\n\t"
130
131
            "'Vanilla-flavored' pam250, very similar to pam250.\n\n"
```

```
132
           "phat_t85_b82:\n\t"
           "BLOSUM-clustered scoring matrix with target frequency\n\t"
133
           "PHDhtm clustering = {75,80,85}percent and background frequency\n\t"
134
135
            "Persson-Argos clustering = {73,78,82}percent.\n\t"
136
           "From Ng, Henikoff, & Henikoff, Bioinformatics 16: 760.\n\n"
137
           "coil_mat:\n" "alpha_mat:\n" "beta_mat:\n\t"
138
           "Three structure-specific matrices described by Luthy, \n\t"
139
           "McLachlan, and Eisenberg in Proteins 10, 229-239, obtained from AAindex.\n\n");
140
     fprintf (stdout, "\n");
141 }
```

3.9.2.7 **bitGraph_t*** pruneBitGraph (**bitGraph_t** * bg, int * indexToSeq, int ** offsetToIndex, int numOfSeqs, int p)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

References emptySet(), bitGraph_t::graph, and nextBitBitSet().

```
404 {
405
     int i = 0, j = 0, nextBit = 0;
406
     int *seqNums = NULL;
407
408
        // Since we don't immediately know which node is in which source
409
       // sequence, we can't just count them up regularly. Instead, we'll
        // need to keep track of which sequences they come from and
411
        // increment _something_. What we chose to do here is just make
412
        // an array of integers of length = . Then, we try to put the
413
        // source sequence number of each neighbor (including itself, since
414
        // the main diagonal is still true at this time) into the next slot
415
       // Since we will monotonically search the bitSet, we can just
416
        // move on to the first bit in the next sequence using the
417
        // offsetToIndex structure so that we know the next sequence number
418
        // to be put in is always unique.
419
       seqNums = (int *) malloc (p * sizeof (int));
420
     if (seqNums == NULL)
421
422
          fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423
            strerror (errno));
424
          fflush (stderr);
425
          exit (0);
426
427
428
        // So, for each row in the bitgraph...
```

```
429
        for (i = 0; i < bg->size; i++)
430
431
432
        \ensuremath{//} Make sure the whole array is -1 sentinels.
433
        for (j = 0; j < p; j++)
434
435
          seqNums[j] = -1;
436
          j = 0;
437
438
        \ensuremath{//} Find the first neighbor of this bit.
439
440
        nextBit = nextBitBitSet (bg->graph[i], 0);
441
          if (nextBit == -1)
442
443
          continue;
444
445
          else
446
447
448
            // and put its sequence number in the array of ints.
449
            seqNums[0] = indexToSeq[nextBit];
450
451
        // If it's the last sequence, then bail out so that we don't
452
453
        // segfault in the next step.
        if (seqNums[0] >= numOfSeqs - 1)
454
455
456
          emptySet (bg->graph[i]);
457
          continue;
458
459
460
        \ensuremath{//} Find the next neighbor of this bit, STARTING AT the first
        // bit in the next sequence.
461
462
        nextBit =
463
        nextBitBitSet (bg->graph[i],
464
                    offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466
        // And iterate this until we run out of neighbors.
467
        while (nextBit >= 0)
468
469
          j++;
470
          seqNums[j] = indexToSeq[nextBit];
471
472
            // Or until this new neighbor will fill up the array
            if (j == p - 1)
473
474
            {
475
              break;
476
            }
477
478
            // Or until this new neighbor is in the last sequence.
479
            if (seqNums[j] >= numOfSeqs - 1)
480
481
              break;
482
483
484
            // Get the next neighbor!
485
            nextBit =
```

```
486
            nextBitBitSet (bg->graph[i],
487
                   offsetToIndex[indexToSeq[nextBit] + 1][0]);
488
489
490
        // If we didn't have enough unique sequences, and either a) we
491
        // were in the nth-to-last sequence and there were no
        // neighbors after it, or b) we were in the last sequence,
492
493
        // then the last number will still be our sentinel, -1. If
494
        // the last number is not a sentinel, then we have at least
495
        // p distinct sequence occurrences, so we're OK.
        if (seqNums[p - 1] == -1)
496
497
        {
498
          emptySet (bg->graph[i]);
499
500
501
      free (seqNums);
502
      return (bq);
503 }
```

3.9.2.8 void usage (char ** argv)

This function describes the basic usage of Gemoda. It is invoked whenever the user submits poor input parameters or selects the help parameter. The function prints a list of possible parameters for Gemoda.

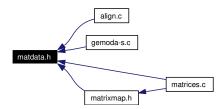
Definition at line 32 of file gemoda-s.c.

```
33 {
34
     fprintf (stdout, "Usage: %s -i <Fasta sequence file> "
35
           "-l <word size> \n\t-k <support> -g <threshold>"
           "-m <matrix name> [-z] \ln[-c < cluster method [0|1]>]"
36
37
           "[-p <unique support>] \n\n\n"
38
           "Required flags and input:\n\n"
39
           "-i <Fasta sequence file>:\n\t"
           "File containing all sequences to be searched, in Fasta format.\n\"
40
41
            "-l <word size>:\n\t"
42
           "Minimum length of motifs; also the sliding window length\n\t"
43
           "over which all motifs must meet the similarity criterion\n"
44
           "-k <support>:\n\t" "Minimum number of motif occurrences.\n\n"
           "-g <threshold>:\n\t"
45
46
           "Similarity threshold. Two windows, when scored with the \n\t"
           " similarity matrix defined by the -m flag, must have at least\n\t"
47
48
49
           " this score in order to be deemed 'connected'. This criterion\n\t"
50
            " must be met over all sliding windows of length 1.\n\"
51
           "-m <matrix name>:\n\t"
           "Name of the similarity matrix to be used to compare windows.\n\t"
52
53
            "Use -z to see a list of matrices installed by default.\n\
           "Optional flags and input:\n\n" -z:\n\t"
54
           "Lists all of the similarity matrices available with the \t^{"}
55
56
           "initial installation of Gemoda. Note that this overrides\n\t"
           "all other options and will only give this output.\n\"
57
           "-c <cluster method [0|1]>:\n\t'
58
           "The clustering method to be used after evaluating the " \,\,
59
```

```
60
           "\n\tsimilarity of the unique words in the input. Note that the "
61
62
           "\n\tclustering method will have a significant impact on both the "
63
            "\n\tresults that one obtains and the computation time.\n\n\t"
64
           "0: clique-finding\n\t\t"
65
           "Uses established methods to find all maximal cliques in the "
66
           "\n\t\tdata. This will give the most thorough results (that are "
67
68
           \n '\n\t\tprovably exhaustive), but will also give less-significant "
69
            "\n\t\tresults in addition to the most interesting and most\n\t"
70
71
           "significant ones. The results are deterministic but may take some "
72
73
           "\n\t\ttime on data sets with high similarity or if the similarity "
74
            \n '\n\t\tthreshold is set extremely low.\n\t"
75
           "1: single-linkage clustering\n\t\"
76
           "Uses a single-linkage-type clustering where all nodes that "
77
           "\n\t\tare connected are put in the same cluster. This method is "
78
79
           "\n\t\talso deterministic and will be faster than clique-finding, "
80
81
           "\n\ttbut it loses guarantees of exhaustiveness in searching the "
            \label{linear_n} $$ ''\in \mathbb{N}_{n} = -p \in \sup_{x\in \mathbb{N}_{n}} (x) 
82
83
           "A pruning parameter that requires the motif to occur in "
84
           "\n\tat least <unique support> different input sequences. Note "
85
86
           "\n\tthat this parameter must be less than or equal to the total "
87
            "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
88
    fprintf (stdout, "\n");
89 }
```

3.10 matdata.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

• #define MATRIX_SIZE 23

3.10.1 Detailed Description

This file defines the size of the scoring matrices so that we don't have to pound-include the whole matrices.h file due to worries about incompatibilities with earlier extern variable declarations.

Definition in file matdata.h.

3.10.2 Define Documentation

3.10.2.1 #define MATRIX_SIZE 23

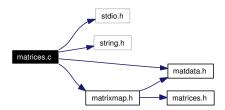
Definition at line 10 of file matdata.h.

Referenced by main().

3.11 matrices.c File Reference

```
#include <stdio.h>
#include <string.h>
#include "matdata.h"
#include "matrixmap.h"
```

Include dependency graph for matrices.c:



Defines

• #define DEFAULT_MATRIX blosum62

Functions

• void getMatrixByName (char name[], const int(**matp)[MATRIX_SIZE])

3.11.1 Detailed Description

This file contains functions for handling scoring matrices used for the sequence based Gemoda.

Definition in file matrices.c.

3.11.2 Define Documentation

3.11.2.1 #define DEFAULT_MATRIX blosum62

Definition at line 7 of file matrices.c.

Referenced by getMatrixByName().

3.11.3 Function Documentation

3.11.3.1 void getMatrixByName (char *name*[], const int ** *matp*[MATRIX_SIZE])

A simple function to take the matrix name argument given as input to gemoda and return the physical memory location of that matrix by using the matrix_map construct. Input: a string containing the matrix name a pointer to a two-dimensional array. Output: None, though the value of the pointer given as input is changed to reflect the location of the matrix

Definition at line 34 of file matrices.c.

References DEFAULT_MATRIX, and matrix_map.

```
35 {
36
     int i;
37
     for (i = 0; matrix_map[i].name != NULL; i++)
38
39
         if (strcmp (name, matrix_map[i].name) == 0)
40
41
         break;
42
43
     if (matrix_map[i].name != NULL)
44
45
46
         *matp = (matrix_map[i].mat);
47
48
     else
49
50
         *matp = (DEFAULT_MATRIX);
51
52 }
```

3.12 matrices.h File Reference

This graph shows which files directly or indirectly include this file:



Variables

- const int aaOrder []
- const int dna_idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int identity_aa [MATRIX_SIZE][MATRIX_SIZE]
- const int idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum100 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum30 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum35 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum40 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum45 [MATRIX SIZE][MATRIX SIZE]
- const int blosum50 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum55 [MATRIX SIZE][MATRIX SIZE]
- const int blosum60 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum62 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum65 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum70 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum75 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum80 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum85 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum90 [MATRIX SIZE][MATRIX SIZE]
- const int blosumn [MATRIX_SIZE][MATRIX_SIZE]
- const int dayhoff [MATRIX_SIZE][MATRIX_SIZE]
- const int pam100 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam110 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam120 [MATRIX SIZE][MATRIX SIZE]
- const int pam130 [MATRIX SIZE][MATRIX SIZE]
- const int pam140 [MATRIX_SIZE][MATRIX_SIZE]
- 150 DAARDIX GIZERAARDIX GIZE
- const int pam150 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam160 [MATRIX_SIZE][MATRIX_SIZE]
 const int pam190 [MATRIX_SIZE][MATRIX_SIZE]
- const int paint 90 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam200 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam210 [MATRIX_SIZE][MATRIX_SIZE]
 const int pam220 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam230 [MATRIX_SIZE][MATRIX_SIZE]

- const int pam240 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam250 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam260 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam280 [MATRIX SIZE][MATRIX SIZE]
- const int pam290 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam300 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam310 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam320 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam330 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam340 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam360 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam370 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam380 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam390 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam400 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam430 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam440 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam450 [MATRIX_SIZE][MATRIX_SIZE]
 const int pam460 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam490 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam500 [MATRIX_SIZE][MATRIX_SIZE]
- const int phat_t75_b73 [MATRIX_SIZE][MATRIX_SIZE]
- const int phat_t80_b78 [MATRIX_SIZE][MATRIX_SIZE]
 const int phat_t85_b82 [MATRIX_SIZE][MATRIX_SIZE]
- const int alpha_mat [MATRIX_SIZE][MATRIX_SIZE]
- const int beta_mat [MATRIX_SIZE][MATRIX_SIZE]
- const int coil_mat [MATRIX_SIZE][MATRIX_SIZE]

3.12.1 Detailed Description

This file contains a number of scoring matrices, most of which are intended for comparing amino acid sequences; however a few are for DNA. In general, if a user wants to add their own matrix for use with Gemoda, they should add it to this file and recompile Gemoda.

Note that users are not restricted to 23x23 matrices. By changing aaOrder, you can easily make matrices for comparing ANSII strings with up to 256 different characters.

All of the matrices below were obtained directly from BLAST/WU-BLAST; they are all also part of the public domain, so there is nothing intrinsic to BLAST with respect to the matrices. It was just the easiest way to get all of the matrices into our software.

The most popular matrix for amino acid sequences is blosum62.

A good location for getting new scoring matrices, such as those based on structural data, is the AAIndex. URLs tend to change, so rather than us listing it here, Google it! Definition in file matrices.h.

3.12.2 Variable Documentation

3.12.2.1 const int aaOrder[]

Definition at line 32 of file matrices.h.

Referenced by alignMat().

3.12.2.2 const int alpha_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1398 of file matrices.h.

3.12.2.3 const int beta_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1422 of file matrices.h.

3.12.2.4 const int blosum100[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 126 of file matrices.h.

3.12.2.5 const int blosum30[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 150 of file matrices.h.

3.12.2.6 const int blosum35[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 174 of file matrices.h.

3.12.2.7 const int blosum40[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 198 of file matrices.h.

3.12.2.8 const int blosum45[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 222 of file matrices.h.

3.12.2.9 const int blosum50[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 246 of file matrices.h.

3.12.2.10 const int blosum55[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 270 of file matrices.h.

3.12.2.11 const int blosum60[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 294 of file matrices.h.

3.12.2.12 const int blosum62[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 318 of file matrices.h.

3.12.2.13 const int blosum65[MATRIX SIZE][MATRIX SIZE]

Definition at line 342 of file matrices.h.

3.12.2.14 const int blosum70[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 366 of file matrices.h.

3.12.2.15 const int blosum75[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 390 of file matrices.h.

3.12.2.16 const int blosum80[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 414 of file matrices.h.

3.12.2.17 const int blosum85[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 438 of file matrices.h.

3.12.2.18 const int blosum90[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 462 of file matrices.h.

3.12.2.19 const int blosumn[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 486 of file matrices.h.

3.12.2.20 const int coil_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1446 of file matrices.h.

3.12.2.21 const int dayhoff[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 510 of file matrices.h.

3.12.2.22 const int dna_idmat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 50 of file matrices.h.

3.12.2.23 const int identity aa[MATRIX SIZE][MATRIX SIZE]

Definition at line 76 of file matrices.h.

3.12.2.24 const int idmat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 101 of file matrices.h.

3.12.2.25 const int pam100[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 534 of file matrices.h.

3.12.2.26 const int pam110[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 558 of file matrices.h.

3.12.2.27 const int pam120[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 582 of file matrices.h.

3.12.2.28 const int pam130[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 606 of file matrices.h.

3.12.2.29 const int pam140[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 630 of file matrices.h.

3.12.2.30 const int pam150[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 654 of file matrices.h.

3.12.2.31 const int pam160[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 678 of file matrices.h.

3.12.2.32 const int pam190[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 702 of file matrices.h.

3.12.2.33 const int pam200[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 726 of file matrices.h.

3.12.2.34 const int pam210[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 750 of file matrices.h.

3.12.2.35 const int pam220[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 774 of file matrices.h.

3.12.2.36 const int pam230[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 798 of file matrices.h.

3.12.2.37 const int pam240[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 822 of file matrices.h.

3.12.2.38 const int pam250[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 846 of file matrices.h.

3.12.2.39 const int pam260[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 870 of file matrices.h.

3.12.2.40 const int pam280[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 894 of file matrices.h.

3.12.2.41 const int pam290[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 918 of file matrices.h.

3.12.2.42 const int pam300[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 942 of file matrices.h.

3.12.2.43 const int pam310[MATRIX SIZE][MATRIX SIZE]

Definition at line 966 of file matrices.h.

3.12.2.44 const int pam320[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 990 of file matrices.h.

3.12.2.45 const int pam330[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1014 of file matrices.h.

3.12.2.46 const int pam340[MATRIX SIZE][MATRIX SIZE]

Definition at line 1038 of file matrices.h.

3.12.2.47 const int pam360[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1062 of file matrices.h.

3.12.2.48 const int pam370[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1086 of file matrices.h.

3.12.2.49 const int pam380[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1110 of file matrices.h.

3.12.2.50 const int pam390[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1134 of file matrices.h.

3.12.2.51 const int pam400[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1158 of file matrices.h.

3.12.2.52 const int pam430[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1182 of file matrices.h.

3.12.2.53 const int pam440[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1206 of file matrices.h.

3.12.2.54 const int pam450[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1230 of file matrices.h.

3.12.2.55 const int pam460[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1254 of file matrices.h.

3.12.2.56 const int pam490[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1278 of file matrices.h.

3.12.2.57 const int pam500[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1302 of file matrices.h.

3.12.2.58 const int phat_t75_b73[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1326 of file matrices.h.

3.12.2.59 const int phat_t80_b78[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1350 of file matrices.h.

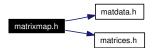
3.12.2.60 const int phat_t85_b82[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1374 of file matrices.h.

3.13 matrixmap.h File Reference

```
#include "matdata.h"
#include "matrices.h"
```

Include dependency graph for matrixmap.h:



This graph shows which files directly or indirectly include this file:



Variables

```
struct {
    char * name
    const int(* mat )[MATRIX_SIZE]
} matrix_map[]
```

3.13.1 Detailed Description

This file contains structures and functions for handling scoring matrices.

Definition in file matrixmap.h.

3.13.2 Variable Documentation

3.13.2.1 const int(* mat)[MATRIX_SIZE]

Definition at line 15 of file matrixmap.h.

Referenced by alignMat(), alignWordsMat_bit(), and main().

3.13.2.2 struct { ... } matrix_map[]

This data structure maps the names of common matrices to the names of their variables Referenced by getMatrixByName().

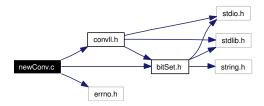
3.13.2.3 char* name

Definition at line 14 of file matrixmap.h.

3.14 newConv.c File Reference

```
#include "bitSet.h"
#include <errno.h>
#include "convll.h"
```

Include dependency graph for newConv.c:



Functions

- int findCliques (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int singleLinkage (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int filterIter (bitGraph_t *graph, int support, bitSet_t *changed, bitSet_t *work)
- int filterGraph (bitGraph_t *graph, int support, int R)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetTo-Index, int numOfSeqs, int p)
- cll_t * pruneCll (cll_t *head, int *indexToSeq, int p)
- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)

3.14.1 Detailed Description

This file contains the core functions that performed the convolution in the Gemoda algorithm. As well, there are two clustering functions defined in this file: one for single linkage clustering, and one for clique based clustering.

Definition in file newConv.c.

3.14.2 Function Documentation

3.14.2.1 cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberOfSequences, int noConvolve, FILE * OUTPUT FILE)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References bitGraphSetFalseDiagonal(), completeConv(), deleteBitSet(), fillSet(), filterGraph(), findCliques(), newBitSet(), pruneBitGraph(), pruneCll(), single-Linkage(), bitGraph_t::size, and yankCll().

```
629 {
     bitSet_t * cand = NULL;
630
      bitSet t * mask = NULL;
631
     bitSet_t * Q = NULL;
632
633
      int size = bg->size;
      cll_t * elemPats = NULL;
634
      cll_t * allCliques = NULL;
635
     cll_t * curr = NULL;
636
637
638
        // contains indices (rows) containing the threshold value.
639
        cand = newBitSet (size);
      mask = newBitSet (size);
640
641
      Q = newBitSet (size);
642
      fillSet (cand);
643
      fillSet (mask);
644
645
        // Note that we prune based on p before setting the diagonal false.
646
        if (p > 1)
647
648
          bg =
649
        pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650
651
        // Now we set the main diagonal false for clustering and filtering.
652
653
        bitGraphSetFalseDiagonal (bg);
654
      filterGraph (bg, support, R);
      fprintf (OUTPUT_FILE, "Graph filtered! Now clustering...\n");
655
656
      fflush (NULL);
      if (clusterMethod == 0)
657
658
659
          findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
```

```
660
      else
661
662
        {
663
          singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664
                  p);
665
      fprintf (OUTPUT_FILE,
666
667
            "Clusters found!
                               Now filtering clusters (if option set)...\n");
668
      fflush (NULL);
669
      if (p > 1)
670
        {
671
          elemPats = pruneCll (elemPats, indexToSeq, p);
672
673
      deleteBitSet (cand);
674
      deleteBitSet (mask);
675
      deleteBitSet (0);
676
677
        // Now let's convolve what we made.
        if (noConvolve == 0)
678
679
          fprintf (OUTPUT_FILE, "Now convolving...\n");
680
681
          fflush (NULL);
682
          allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683
684
685
      else
686
        {
          curr = elemPats;
687
688
          while (curr != NULL)
689
690
          yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691
692
693
      return allCliques;
694 }
```

3.14.2.2 int filterGraph (bitGraph_t * graph, int support, int R)

Function to "filter" the initial bitGraph that is being clustered. "Filtering" is the process of removing all nodes from the graph that cannot possibly be in motifs because they are not connected to enough other nodes. This can be done once (if R != 1), or it can be done recursively (if R == 1). When done recursively, it takes the just-filtered graph and checks all of the nodes that the recently removed node used to be connected to; since they have changed in connectivity, they may no longer be connected to enough nodes to be a member of a motif. This is iterated until convergence. Note that the default is to have recursive filtering on, as it ought to decrease the computational complexity of the clustering step and ought not have much of a computational footprint... in cases where it takes a while, it is probably having a good impact in the clustering step, whereas if it is not effective, it probably won't take that long anyway. Input: a bitGraph to be filtered, the minimum support that a motif must have, and the flag indicating recursive filtering or not. Output: Integer success value of 0 (and an altered bitGraph so that all nodes with connections have at least <min support>=""> connections).

Definition at line 359 of file newConv.c.

References copySet(), countSet(), deleteBitSet(), emptySet(), filterIter(), newBitSet(), and bitGraph_t::size.

Referenced by convolve().

```
360 {
361
      bitSet_t * changed = newBitSet (graph->size);
362
     bitSet_t * work = newBitSet (graph->size);
      emptySet (changed);
363
364
      emptySet (work);
365
366
        // Iteratively call the filtering by copying the previous "work" into
367
        // "changed" after each iteration step.
368
        if (R == 1)
369
370
371
          do
372
        {
373
          filterIter (graph, support, changed, work);
374
          copySet (work, changed);
375
376
          while (countSet (changed) > 0);
377
378
      else
379
        {
380
381
        // Otherwise, just do it once.
382
        filterIter (graph, support, changed, work);
383
384
      deleteBitSet (changed);
385
      deleteBitSet (work);
386
      return 0;
387 }
```

3.14.2.3 int filterIter (bitGraph_t * graph, int support, bitSet_t * changed, bitSet_t * work)

The iterator used to "filter" the graph. It takes information in the bitset telling which nodes' rows have changed and only checks them... this should make it pretty efficient time-wise at only a small memory cost. Note the convention that the first time this is called, the changed bitSet is empty... and that the master function is responsible for catching the signal that no changes were made in the last iteration. Input: the bitGraph to be filtered, the minimum support required for a motif to be returned, a bitSet with nodes changed from the previous iteration, and a bitSet to export the nodes changed in this iteration. Output: integer success value of 0 (and also a filtered bitGraph and a bitSet with the nodes changed in this iteration).

Definition at line 228 of file newConv.c.

References countSet(), emptySet(), bitGraph_t::graph, nextBitBitSet(), setFalse(), and

setTrue().

Referenced by filterGraph().

```
230 {
      int i = 0, j = 0;
231
232
      int lastBit = 0, nextBit = 0, lastRow = 0, nextRow = 0;
233
      int numNodes = 0;
234
     int changedSize = countSet (changed);
235
      emptySet (work);
236
237
        // Note the convention that the first time the function is called,
        // it is done with an empty "changed" bitSet as a sentinel. It is
238
239
        // the responsibility of the master function calling the iterator
240
        // to catch future empty changed sets to know that convergence has
        // been achieved.
241
242
243
        // So, if it's your first time through, go through each node and make
244
        // sure that each is connected to at least <support> - 1 others.
245
        if (changedSize == 0)
246
247
          for (i = 0; i < graph->size; i++)
248
249
          numNodes = countSet (graph->graph[i]);
250
          if (numNodes >= support - 1)
251
252
              continue;
253
254
          else
255
            {
256
257
            // Otherwise, zero it out, but going one by
258
            // one so that you can also zero out the
            // symmetric bit.
259
260
            lastBit = 0;
261
              for (j = 0; j < numNodes; j++)
262
              nextBit = nextBitBitSet (graph->graph[i], lastBit);
263
              if (nextBit == -1)
264
265
                {
                  fprintf (stderr,
266
267
                     "\nEnd of bitSet reached! - initial\n");
                  fflush (stderr);
268
269
                  exit (0);
270
271
              setFalse (graph->graph[i], nextBit);
272
              setFalse (graph->graph[nextBit], i);
273
274
                // And set that corresponding bit true
275
                // in the work bitSet so that we
276
                // know we changed it for the next
                // round.
277
278
                setTrue (work, nextBit);
279
              lastBit = nextBit + 1;
280
281
        }
282
```

```
283
      else
284
285
        {
286
287
        // Otherwise, we've been here before, so just follow what
288
        // the changed bitSet says to do... only those bitSets that
289
        // were changed could possibly have gone under the minimum
290
        // support requirement.
291
        lastRow = 0;
292
          for (i = 0; i < changedSize; i++)
293
294
          nextRow = nextBitBitSet (changed, lastRow);
295
          if (nextRow == -1)
296
297
              fprintf (stderr, "\nEnd of bitSet reached! - iter,row\n");
              fflush (stderr);
298
299
              exit (0);
300
301
302
            // So now we've found the row that needs to be checked.
303
            // We do the same thing we did above... either move
304
            // on if it has enough possible support, or zero
305
            // it out (with its symmetric locations) one by one.
306
            numNodes = countSet (graph->graph[nextRow]);
307
          if (numNodes >= support - 1)
308
309
              lastRow = nextRow + 1;
310
              continue;
311
312
          else
313
314
              lastBit = 0;
315
              for (j = 0; j < numNodes; j++)
316
317
              nextBit = nextBitBitSet (graph->graph[nextRow], lastBit);
318
              if (nextBit == -1)
319
                {
320
                  fprintf (stderr,
321
                    "\nEnd of BitSet reached! = iter,Bit\n");
322
                  fflush (stderr);
323
                  exit (0);
324
325
              setFalse (graph->graph[nextRow], nextBit);
326
              setFalse (graph->graph[nextBit], nextRow);
327
              setTrue (work, nextBit);
328
              lastBit = nextBit + 1;
329
330
              lastRow = nextRow + 1;
331
332
333
334
     return 1;
335 }
```

```
3.14.2.4 int findCliques (bitSet_t * Q, bitSet_t * cand, bitSet_t * mask, bitGraph_t * oG, int support, int qCount, cll_t ** elemPats, int * indexToSeq, int p)
```

Recursive algorithm to exhaustively enumerate all of the maximal cliques that exist in the data. This is one of the main workhorses of Gemoda when used in its exhaustive form. This algorithm was originally published by Etsuji Tomita, Akira Tanaka, and Haruhisa Takahasi as a Technical Report of IPSJ (Information Processing Society of Japan): Tomita, E, A Tanaka, & H Takahasi (1989). "An optimal algorithm for finding all of the cliques". SIG Algorithms 12, pp 91-98. Input: a bitset with the nodes currently in the clique, a bitset with the candidates for expanding the clique, a bitset inidcating the current subgraph being searched, the bitGraph to be searched for cliques, the minimum support parameter, a counter variable for keeping track of how many nodes are in the current clique, a linked list of cliques that have been discovered so far, and a pointer to the data structure that dereferences offset indexes into sequence numbers, and the minimum number of unique sequences that must contain the motif. Output: integer success value of 0 (but more importantly, the elemPats clique linked list is expanded to contain all elementary (minimum-length) motif cliques.

Definition at line 37 of file newConv.c.

References bitSetIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, newBitSet(), nextBitBitSet(), pushClique(), setFalse(), setTrue(), and bitGraph t::size.

Referenced by convolve().

```
40 {
41
     bitSet_t ** gammaOG = NULL;
     bitSet_t * candQ = newBitSet (oG->size);
     bitSet_t * newMask = newBitSet (oG->size);
43
     int i, q;
44
     int graphSize;
45
     int max = -1;
46
     int numBits;
48
     int u = 0;
49
     int newMaskCount;
50
     int candQCount;
51
     graphSize = oG->size;
52
53
54
       // Find which vertex in subg maximizes | cand intersect gamma(u) |
55
       gammaOG = oG->graph;
56
     for (i = 0; i < graphSize; i++)</pre>
57
58
       // Don't check this vertex if it's masked
59
60
       if (!(checkBit (mask, i)))
61
       {
62
         continue;
63
64
65
       // cand is always a subset of mask, so intersecting
```

```
66
       // with mask is redundant
       bitSetIntersection (gammaOG[i], cand, candQ);
67
         numBits = countSet (candQ);
69
         if (numBits > max)
70
71
         u = i;
72
         max = numBits;
73
74
75
76
       // Then do the extension of the q's
77
       aCount++;
78
79
       // This loop iterates over all possible values of cand - gamma() by
80
       // iterating over all possible values of cand but immediately
       // "continue"ing if the node is also in gamma(u)
81
82
       q = nextBitBitSet (cand, 0);
83
     while (q != -1)
84
       {
85
         if (checkBit (gammaOG[u], q))
86
87
         q = nextBitBitSet (cand, q + 1);
88
         continue;
89
       }
90
       // SUBGq = SUBG i Gamma
91
92
       bitSetIntersection (mask, gammaOG[q], newMask);
93
        newMaskCount = countSet (newMask);
         setTrue (Q, q);
94
95
96
       // Only recurse if there are more candidates to be included,
97
       \ensuremath{//} and they will allow us to reach the minimum support.
98
       if (newMaskCount > 0 && qCount + newMaskCount >= support)
99
100
101
            // CANDq = CAND i Gamma
102
            bitSetIntersection (gammaOG[q], cand, candQ);
          candQCount = countSet (candQ);
103
104
105
            // only recurse if we can possibly get to a clique
106
            // of size with minimum support
107
            if (candQCount > 0 && qCount + candQCount >= support)
108
            {
109
110
            // recursion with
111
            // new candidates, new mask, and original graph
112
            findCliques (Q, candQ, newMask, oG, support, qCount, elemPats,
113
                     indexToSeq, p);
114
115
116
          else if (qCount >= support)
117
118
119
            // This should be done when:
            // 1. countSet(newMask) == 0 [connected subgraph is maximal]
120
            // 2. Qcount >= minCount [connected subgraph has enough nodes]
121
122
            *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
```

```
123
124
125
        // Remove q from Q, and remove q from cand
126
        setFalse (Q, q);
127
          setFalse (cand, q);
128
          q = nextBitBitSet (cand, q + 1);
129
        }
130
      qCount--;
131
      deleteBitSet (candQ);
132
      deleteBitSet (newMask);
     return 0;
133
134 }
```

3.14.2.5 **bitGraph_t*** pruneBitGraph (**bitGraph_t** * bg, int * indexToSeq, int ** offsetToIndex, int numOfSeqs, int p)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

References emptySet(), bitGraph_t::graph, and nextBitBitSet().

```
404 {
405
     int i = 0, j = 0, nextBit = 0;
406
     int *seqNums = NULL;
407
        // Since we don't immediately know which node is in which source
409
        // sequence, we can't just count them up regularly. Instead, we'll
410
        // need to keep track of which sequences they come from and
411
        // increment _something_. What we chose to do here is just make
       // an array of integers of length = . Then, we try to put the
412
413
       // source sequence number of each neighbor (including itself, since
414
       // the main diagonal is still true at this time) into the next slot
415
        // Since we will monotonically search the bitSet, we can just
416
        // move on to the first bit in the next sequence using the
417
       // offsetToIndex structure so that we know the next sequence number
418
        // to be put in is always unique.
419
        seqNums = (int *) malloc (p * sizeof (int));
420
     if (seqNums == NULL)
421
        {
          fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
422
423
           strerror (errno));
424
          fflush (stderr);
425
          exit (0);
        }
426
```

```
427
        // So, for each row in the bitgraph...
428
429
        for (i = 0; i < bg->size; i++)
430
431
432
        // Make sure the whole array is -1 sentinels.
433
        for (j = 0; j < p; j++)
434
435
          seqNums[j] = -1;
436
          j = 0;
437
438
439
        // Find the first neighbor of this bit.
        nextBit = nextBitBitSet (bg->graph[i], 0);
440
441
          if (nextBit == -1)
442
443
          continue;
        }
444
445
          else
446
447
448
            // and put its sequence number in the array of ints.
449
            seqNums[0] = indexToSeq[nextBit];
450
451
        // If it's the last sequence, then bail out so that we don't
452
453
        // segfault in the next step.
454
        if (seqNums[0] >= numOfSeqs - 1)
455
456
          emptySet (bg->graph[i]);
457
          continue;
458
459
460
        // Find the next neighbor of this bit, STARTING AT the first
        // bit in the next sequence.
461
462
        nextBit =
463
        nextBitBitSet (bg->graph[i],
464
                   offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
        // And iterate this until we run out of neighbors.
466
467
        while (nextBit >= 0)
468
469
          j++;
470
          seqNums[j] = indexToSeq[nextBit];
471
472
            // Or until this new neighbor will fill up the array
473
            if (j == p - 1)
474
            {
475
              break;
476
477
478
            // Or until this new neighbor is in the last sequence.
            if (seqNums[j] >= numOfSeqs - 1)
479
480
            {
481
              break;
482
            }
483
```

```
484
            // Get the next neighbor!
            nextBit =
485
486
            nextBitBitSet (bg->graph[i],
487
                   offsetToIndex[indexToSeq[nextBit] + 1][0]);
488
        }
489
490
        // If we didn't have enough unique sequences, and either a) we
491
        // were in the nth-to-last sequence and there were no
492
        // neighbors after it, or b) we were in the last sequence,
493
        // then the last number will still be our sentinel, -1. If
494
        // the last number is not a sentinel, then we have at least
495
        // p distinct sequence occurrences, so we're OK.
496
        if (seqNums[p - 1] == -1)
497
498
          emptySet (bg->graph[i]);
499
500
501
      free (seqNums);
502
      return (bg);
503 }
```

3.14.2.6 cll_t* pruneCll (cll_t * head, int * indexToSeq, int p)

Prunes a motif linked list of all motifs without support in at least

unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```
515 {
516
     int i = 0, j = 0, thisSeq = 0;
517
      int *seqNums = NULL;
     cll_t * curr = head;
518
519
     cll_t * prev = NULL;
     cll_t * storage = NULL;
520
521
522
        // We'll do this similar to the pruneBitGraph function... we will
523
        // keep track of which source sequence each motif occurrence was in.
524
        // Again, since the occurrences are listed monotonically, we only
525
        // need to compare the last non-sentinel index to the current
526
        // sequence number.
527
        seqNums = (int *) malloc (p * sizeof (int));
528
      if (seqNums == NULL)
529
        {
530
          fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
531
          fflush (stderr);
532
          exit (0);
```

```
533
      while (curr != NULL)
534
535
536
537
        // First make sure the set size is at least p.
538
        // This is redundant, but extremely simple and not expensive,
        // so we'll leave it in just as a check.
539
540
        if (curr->set->size < p)</pre>
541
542
          if (prev != NULL)
543
544
              prev->next = curr->next;
545
546
          else
547
            {
548
              head = curr->next;
            }
549
550
          storage = curr->next;
551
          free (curr->set->members);
552
          free (curr->set);
553
          free (curr);
554
          curr = storage;
555
          continue;
556
557
          for (i = 0; i < p; i++)
558
559
          seqNums[i] = -1;
560
561
562
          seqNums[0] = indexToSeq[curr->set->members[0]];
563
564
        // Note, we've checked to make sure size > p, and we know
565
        \ensuremath{//} p must be 2 or greater, so we can start at 1 without
566
        // worrying about segfaulting
567
        for (i = 1; i < curr->set->size; i++)
568
569
          thisSeq = indexToSeq[curr->set->members[i]];
570
          if (thisSeq != seqNums[j])
571
            {
              j++;
572
573
              seqNums[j] = thisSeq;
574
              if (j == p - 1)
575
576
              break;
577
578
579
580
581
        // Same story as before... if the last number is -1,
582
        // then we didn't have enough to fill up the  different
        // slots, so this doesn't meet our criterion.
583
584
        if (seqNums[p - 1] == -1)
585
586
          if (prev != NULL)
587
            {
588
              prev->next = curr->next;
589
```

```
590
          else
591
592
              head = curr->next;
593
594
          storage = curr->next;
595
          free (curr->set->members);
596
          free (curr->set);
597
          free (curr);
598
          curr = storage;
599
600
          else
601
          prev = curr;
602
603
          curr = curr->next;
604
605
606
      free (seqNums);
607
      return (head);
608 }
```

3.14.2.7 int singleLinkage (bitSet_t * Q, bitSet_t * cand, bitSet_t * mask, bitGraph_t * oG, int support, int qCount, cll_t ** elemPats, int * indexToSeq, int p)

A recursive routine for single linkage clustering. This clustering is much faster than exhaustively enumerating all cliques, but it puts each node in only one cluster and is not guaranteed to give all possible motifs. Input: a bitSet containing the current motif, a bitSet containing candidates to be added to the current motif, a bitSet containing the current subgraph to be clustered, the original bitGraph to be clustered, the minimum support necessary for a motif to be returned, the current number of nodes in the motif, a linked list of elementary motifs (length is the same as the window size), pointer to a structure to derference index values to sequence numbers, and the minimum number of unique sequences that a motif must be in to be returned. Output: integer success value of 0 (but more importantly, the linked list elemPats is updated to contain all of the motifs of length = window size.

Definition at line 154 of file newConv.c.

References bitSetUnion(), checkBit(), copySet(), countSet(), bitGraph_t::graph, next-BitBitSet(), pushClique(), and setFalse().

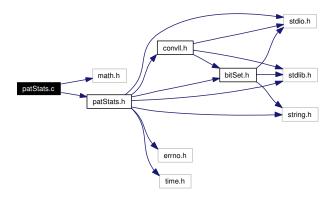
Referenced by convolve().

```
165
166
       // this vertex has been clustered
167
        setFalse (cand, i);
168
169
       // start a new cluster, Q
170
        copySet (oG->graph[i], Q);
171
172
        // go over each vertex in the cluster
173
        j = nextBitBitSet (Q, 0);
174
         while (j != -1)
175
176
177
           // if this vertex has been clustered already, skip it and go
178
           // to the next one
179
            if (!checkBit (cand, j))
180
181
              j = nextBitBitSet (Q, j + 1);
182
             continue;
183
184
            // Add this vertex's neighbors to the current cluster
185
186
           bitSetUnion (Q, oG->graph[j], Q);
187
188
           // This vertex has now been clustered
189
            setFalse (cand, j);
190
191
           // go over each vertex in the cluster
192
            j = nextBitBitSet (Q, 0);
193
194
195
       // Did we make a cluster that was large enough?
196
       if (countSet (Q) >= support)
197
198
          *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
199
200
201
       // recurse
202
        singleLinkage (Q, cand, mask, oG, support, 0, elemPats, indexToSeq,
203
                  p);
204
205
      else
206
207
         return 0;
208
     return 0;
209
210 }
```

3.15 patStats.c File Reference

```
#include <math.h>
#include "patStats.h"
```

Include dependency graph for patStats.c:



Functions

- int getLargestSupport (cll_t *cliqs)
- int getLargestLength (cll_t *cliqs)
- int measureDiagonal (const bitGraph_t *bg, const int i, const int j)
- unsigned int ** increaseMem (unsigned int **d, int dimToChange, int curr-Support, int currLength, int newVal)
- unsigned int ** oldGetStatMat (bitGraph_t *bg, int support, int length, int *supportDim, int *lengthDim, int numBlanks)
- unsigned int ** getStatMat (bitGraph_t *bg, int support, int length, int *support-Dim, int *lengthDim, int numBlanks, int s, FILE *OUTPUT_FILE)
- int cumDMatrix (unsigned int **d, cll_t *cliqs, int currSupport, int currLength, int bgSize, int numSeqs)
- double calcStatCliq (unsigned int **d, cll_t *cliq, int numWindows)
- int calcStatAllCliqs (unsigned int **d, cll_t *allCliqs, int numWindows)
- int freeD (unsigned int **d, int supportDim)
- int statCompare (const cll_t **first, const cll_t **second)
- cll_t * sortByStats (cll_t *allCliqs)

3.15.1 Detailed Description

This file defines functions that are used to compute the statistical significance of motifs for both the sequence based and real value based implementations of Gemoda. The basic approach we take, is to calculate the probability of establishing a single cluster, and to multiply this probability by the probability that the cluster can be extended an arbitrary number of locations. Essentially, this is the probability of getting and elementary motif during the clustering phase and having that motif convolved multiple times during the convolution phase.

Definition in file patStats.c.

3.15.2 Function Documentation

3.15.2.1 int calcStatAllCliqs (unsigned int ** d, cll t * allCliqs, int numWindows)

Definition at line 676 of file patStats.c.

References calcStatCliq(), cnode::next, and cnode::stat.

Referenced by main().

```
677 {
678
      cll_t * curr = NULL;
679
      curr = allCliqs;
680
      while (curr != NULL)
681
       {
682
          curr->stat = calcStatCliq (d, curr, numWindows);
683
          curr = curr->next;
684
685
     return (0);
686 }
```

3.15.2.2 double calcStatCliq (unsigned int **d, cll_t * cliq, int numWindows)

Definition at line 623 of file patStats.c.

References cnode::length, cnode::set, and cSet_t::size.

Referenced by calcStatAllCliqs().

```
624 {
625    double stat = 0;
626    int i = 0;
627    int supChooseTwo = 0;
628    double interimP = 0;
629    int support = cliq->set->size;
630    int length = cliq->length;
631    double numTrials = 0;
```

```
632
      if (support < 2)
633
          fprintf (stderr, "Support for cluster less than 2... exiting.\n");
634
635
          fflush (stderr);
636
          exit (0);
637
638
639
        // OK, so support is at least two. So we make the connections all
640
        // on the first level, knowing that each node being connected has
641
        // at least zero in common. There are [(size of cluster) - 1] of
642
        // these connections to be made.
643
        // And we know we can call for d[0][1] because if the second index
644
        // were out of bounds, then there would be no similarities, and
645
        // there would be no reason to call this function.
646
        interimP = ((double) d[0][1]) / ((double) d[0][0]);
647
      stat = pow (interimP, support - 1);
      stat *= ((double) numWindows * (numWindows - 1)) / ((double) 2);
648
649
650
        \ensuremath{//} 
 Now we actually calculate the probability... the first connection
651
        // has to be made no matter what, and after that we multiply for
        // every connection after the first one. So we descend iteratively
652
653
        // until we have made all connections, terminating after we've made
654
        // the single i = (n - 2) connection. There is no i = (n - 1)
655
        // connection.
656
        for (i = 1; i < support - 1; i++)
657
658
          interimP = ((double) d[i][1]) / ((double) d[i][0]);
659
          stat *= pow (interimP, support - i - 1);
660
          stat *= ((double) (numWindows - (i + 1))) / ((double) (i + 2));
661
        } supChooseTwo = (support * (support - 1)) / 2;
662
        // Remember that length = (numwindows - 1), or alternatively,
663
664
        \ensuremath{//} the number of extensions... normally we'd want to have the last
665
        // p be p[support][numwindows - 1], which corresponds to
666
        // alteredD[support][numwindows]/alteredD[support][numwindows-1],
        // so that means we want our last d to be d[support][numwindows].
667
668
        // Here, we note that the calculation of p's would be continuously
669
        // re-normalizing, so multiplying all p's is the same as dividing
670
        // the last d by the initial d.
671
        interimP = ((double) d[support][length + 1]) / ((double) d[support][1]);
672
      stat *= pow (interimP, supChooseTwo);
673
      return stat;
674 }
```

3.15.2.3 int cumDMatrix (unsigned int ** d, cll_t * cliqs, int currSupport, int currLength, int bgSize, int numSeqs)

Definition at line 522 of file patStats.c.

References getLargestLength(), and getLargestSupport().

```
524 {
```

```
525
      int maxSup = 0;
      int maxLen = 0;
526
527
      int i, j;
528
      int numWins = 0;
529
530
        maxSup = getLargestSupport (cliqs);
531
      maxLen = getLargestLength (cliqs);
532
533 /******* COMMENTED OUT
        // First we note that the number of unique streaks of a given
        // support is defined by d[support][1], where as 1 increases,
535
536
        // the value of d decreases because only unique streaks are
        // counted.
537
        // We also note that the number of disjoint node-pairs with a given
538
        // number of other nodes in common is defined by d[support][0].
539
540
        // So, in order to properly account for all "unique" comparisons
541
        // (which is equal to (# streaks + # disjoint node-pairs), we must
542
        // add d[support][1] to d[support][0].
543
544
        for (i = 0; i < currSupport + 1; i++) {
545
            d[i][0] += d[i][1];
546
        ,
********/
547
548
549
        // We no longer need to do that, since now we sum across both
        // the support and the length dimensions. Now, d[support][0] will
550
        // necessarily include d[support][1] being added to it. We don't
552
        // want to add this anymore, otherwise we would be underestimating
553
        // the probability of making that first connection. For instance,
554
        // if there were no nodes with 20 in common that weren't also
555
        // connected, and no nodes whatsoever with more than 20 in common,
556
        // we'd want the p[20][0] to be 1, which would be
557
        // d[20][1]/d[20][0]. When summing across length directions,
558
        // this happens naturally, whereas before we needed to do it
559
        // artificially as per above. If we did above, we'd have the
560
        // probability of each node being 1/2 instead of 1.
561
562
        // Rather than storing doubles and doing lots of multiplications,
563
        // we're going to limit the number of operations done in the actual
564
        // probability calculation by only storing cumulative sums in d.
565
        // Now remember, what we're storing at each location is the
566
        // number of nodes with [i] or more nodes in common (including
567
        // each other and selves) that can be extended [j] times (with
568
        // their initial similarity counting as 1).
569
        //
570
        // We go up to the last possible index in the length direction, which
571
        \ensuremath{//} means going up to [maxLen]. We know that this is legitimate
572
        // because maxLen is less than or equal to the longest possible
573
        // diagonal, and the longest possible diagonal will be less
574
        // than or equal to currLength. Since we have allotted
575
        // (currLength + 1) integers, we know we're OK to access [currLength].
576
        for (j = 0; j < currLength + 1; j++)
577
578
        // We start at currSupport - 1, because currSupport will
579
        // clearly not be changed, and this makes it a much easier
580
581
        // loop to read.
```

```
582
        for (i = currSupport - 1; i >= 0; i--)
583
584
          d[i][j] += d[i + 1][j];
585
586
587
      for (i = 0; i < currSupport + 1; i++)
588
589
          for (j = currLength - 1; j >= 0; j--)
590
591
          d[i][j] += d[i][j + 1];
592
593
        }
594
595
        // Now we need to forcibly set d[0][0] to its correct value... it's
596
        // just the total number of comparisons, not including comparisons
597
        \ensuremath{//} to delimiter 0's meant to separate sequences. The number of
598
        // windows is equal to the number of offsets minus the number
599
        // of sequences (assuming one delimiter per sequence). We don't count
600
        // the main diagonal, so the first row has one less, and we want to
601
        // sum over all the subsequent rows in the upper half of the matrix.
        // So it's (numWins - 1)*(numWins - 1 + 1)/2 to sum that up.
602
603
        numWins = bgSize - numSeqs;
     d[0][0] = numWins * (numWins - 1) / 2;
604
605
606
            for (i = 0; i <= maxSup; i++) { printf("support = d:\t",i); for (j = 0; j <=
607
608
           maxLen; j++) { printf("%d\t",d[i][j]); } printf("\n"); }
609
610
        return 1;
611 }
```

3.15.2.4 int freeD (unsigned int **d, int supportDim)

Definition at line 688 of file patStats.c.

```
689 {
690
      int i = 0;
      if (d == 0)
691
692
        {
693
          return 0;
694
        }
695
      else
696
        {
697
698
        // Still, it's supportDim + 1, because we have an extra
699
        // one for the "0" support.
        for (i = 0; i < supportDim + 1; i++)
700
701
702
          free (d[i]);
703
704
          free (d);
705
          return 0;
```

```
706 }
```

3.15.2.5 int getLargestLength (cll_t * cliqs)

Given a clique linked list, this function will return an integer which is equal to the length of the member of the linked list with the largest length.

Definition at line 44 of file patStats.c.

References cnode::length, and cnode::next.

Referenced by cumDMatrix().

```
45 {
46
     int len = 0;
    cll_t * curCliq = NULL;
47
48
     curCliq = cliqs;
49
     while (curCliq != NULL)
50
         if (curCliq->length > len)
51
52
53
         len = curCliq->length;
54
       }
55
         curCliq = curCliq->next;
56
57
58
       // We return (len + 1) because the length of the shortest streak
59
       // is one, but is stored in the cluster data structure as being
60
       // zero (number of extensions that have been made).
61
       return (len + 1);
62 }
```

3.15.2.6 int getLargestSupport (cll_t * cliqs)

Given a clique linked list, this function will return an integer which is equal to the support of the member of the linked list with the largest support.

Definition at line 22 of file patStats.c.

References cnode::next, cnode::set, and cSet_t::size.

Referenced by cumDMatrix().

```
23 {
24    int size = 0;
25    cll_t * curCliq = NULL;
26    curCliq = cliqs;
27    while (curCliq != NULL)
28    {
29      if (curCliq->set->size > size)
```

3.15.2.7 unsigned int** getStatMat (bitGraph_t * bg, int support, int length, int * supportDim, int * lengthDim, int numBlanks, int s, FILE * OUTPUT_FILE)

Definition at line 329 of file patStats.c.

References bitGraphRowIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, increaseMem(), measureDiagonal(), newBitSet(), nextBitBitSet(), and bitGraph_t::size.

```
331 {
     int *Q = NULL;
332
333
     unsigned int **d = NULL;
334
     int i, j, k;
335
     int x, y;
336 bitSet_t * X = NULL;
337
     int currSupport;
338
     int currLength;
     int multiplier = 50;
339
340 int diagonal = 0;
341 time_t probStart, probEnd;
     int timeNeeded = 0;
342
343
     int sampleCounter = 1;
344
345
       // int visitCounter = 0, uniqCounter = 0;
346
       currSupport = support * multiplier;
347
     currLength = length * multiplier;
348
     X = newBitSet (bg->size);
349
350
        // printf("Made bitSet of size %d\n", bg->size);
351
        Q = (int *) malloc (bg->size * sizeof (int));
      if (Q == NULL)
352
353
        {
354
          fprintf (stderr,
355
            "\nMemory error --- couldn't allocate array!" "\n%s\n",
356
            strerror (errno));
357
          fflush (stderr);
358
          exit (0);
359
360
      for (i = 0; i < bg->size; i++)
361
        {
362
          Q[i] = 0;
        }
363
```

```
364
       (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
365
366
      if (d == NULL)
367
368
          fprintf (stderr,
            "\nMemory error --- couldn't allocate array!" "\n%s\n",
369
370
            strerror (errno));
371
          fflush (stderr);
372
          exit (0);
373
374
      for (i = 0; i < currSupport + 1; i++)</pre>
375
376
         d[i] =
377
       (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
378
          if (d[i] == NULL)
379
380
          fprintf (stderr, "\nMemory error --- couldn't allocate array!"
381
                "\n%s\n", strerror (errno));
382
          fflush (stderr);
383
          exit (0);
384
385
          for (j = 0; j < currLength + 1; j++)
386
387
         d[i][j] = 0;
388
389
390
391
        // printf("size=%d\n",bg->size);
392
       time (&probStart);
393
      for (i = 0; i < bg->size; i++)
394
395
          if (i == 200)
396
397
          time (&probEnd);
          timeNeeded = ((double) (probEnd - probStart)) /
398
            ((double) 60) * ((double) bg->size) / ((double) 200);
399
400
          if (timeNeeded > 2)
401
402
              fprintf (OUTPUT_FILE,
403
                "Max total time to calculate probability:\n");
404
              fprintf (OUTPUT_FILE, "\t%d minutes\n", timeNeeded);
              fprintf (OUTPUT_FILE, "Actual time will be less than this, "
405
406
                "but at least half of it.\n");
407
              fprintf (OUTPUT_FILE,
408
                "To bypass excessive probability calculations,"
                " cancel and use a different value\n"
409
                " for the '-s' flag (samples every " \,
410
                "'s' points).\n");
411
412
              fflush (NULL);
413
            }
414
        }
415
          j = nextBitBitSet (bg->graph[i], 0);
416
          while (j >= 0)
417
          k = nextBitBitSet (bg->graph[i], j + 1);
418
419
          while (k >= 0)
420
            {
```

```
421
              if (checkBit (bg->graph[j], k) == 0)
422
423
              if (sampleCounter == s)
424
                {
425
                  bitGraphRowIntersection (bg, j, k, X);
426
427
                // visitCounter++;
428
                if (nextBitBitSet (X, 0) >= i)
429
430
                    // uniqCounter++;
431
432
                    x = countSet(X);
433
                  while (x > currSupport)
434
                    {
435
                      d =
436
                    increaseMem (d, 1, currSupport, currLength,
437
                              currSupport +
438
                              support * multiplier);
439
                      currSupport += support * multiplier;
440
441
                  d[x][0] += 1;
442
                  sampleCounter = 0;
443
444
445
              sampleCounter++;
446
447
              k = nextBitBitSet (bg->graph[i], k + 1);
448
449
          if (j <= i)
450
            {
451
              j = nextBitBitSet (bg->graph[i], j + 1);
452
              continue;
453
454
          bitGraphRowIntersection (bg, i, j, X);
455
          x = countSet(X);
456
457
            // Note, now we're using "diagonals" rather than
            // location in a horizontal array. So you always
458
459
            // start from the main diagonal at 0 and move out.
460
            diagonal = j - i;
461
462
            // We change this to greater-than-one because
            // after Q[diagonal] is reduced to one, it isn't
463
464
            // visited again until we reach a new streak, (because
            // the next bit in the diagonal is a zero), and at
465
466
            // that point we want to start with a new diagonal
467
            // measure.
468
            if (Q[diagonal] > 1)
469
470
              y = Q[diagonal] - 1;
471
              Q[diagonal]--;
472
            }
473
          else
474
475
              y = measureDiagonal (bg, i, j);
476
              Q[diagonal] = y;
477
```

```
478
          while (x > currSupport)
479
480
              d = increaseMem (d, 1, currSupport, currLength,
481
                    currSupport + support * multiplier);
482
              currSupport += support * multiplier;
483
484
          while (y > currLength)
485
            {
486
              d =
487
            increaseMem (d, 2, currSupport, currLength,
                     currLength + length * multiplier);
488
489
              currLength += length * multiplier;
490
491
          d[x][y]++;
492
          j = nextBitBitSet (bg->graph[i], j + 1);
493
494
                if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
495
496
497
498
499
500
            printf("done\n"); fflush(stdout);
501
        }
502
503
504
        // We need to rescale by the sampling factor for all i>0 in d[i][0].
505
506
        for (i = 1; i < currSupport; i++)</pre>
507
        {
508
          d[i][0] *= s;
509
        }
510
511
        // Now we only need to assign the correct value for d[0][0]...
512
        // but rather than figuring that out, we will just assign it in the
513
        // cumulative function, since there it is merely the number of unique
514
        // non-self comparisons and is easy to calculate.
515
        deleteBitSet (X);
516
      free (Q);
517
      *supportDim = currSupport;
518
      *lengthDim = currLength;
519
      return (d);
520 }
```

3.15.2.8 unsigned int** increaseMem (unsigned int ** d, int dimToChange, int currSupport, int currLength, int newVal)

This function is used to increase the size of an array of pointers to pointers to integers. dimToChange is 1 for the first dimension (support), 2 for the second dimension (length). newVal is the new value for the dimension to be changed, not including the "1" that should be added... so it should just be some integer times the initial support.

Definition at line 91 of file patStats.c.

 $Referenced\ by\ getStatMat(),\ and\ oldGetStatMat().$

```
93 {
94
     int i = 0, j = 0;
     if (dimToChange == 1)
95
96
97
98
       (unsigned int **) realloc (d, (newVal + 1) * sizeof (unsigned int *));
99
         if (d == NULL)
100
          fprintf (stderr, "\nMemory error --- couldn't allocate array!"
101
102
                 "\n%s\n", strerror (errno));
103
          fflush (stderr);
104
          exit (0);
105
106
          for (i = currSupport + 1; i < newVal + 1; i++)</pre>
107
          d[i] =
108
            (unsigned int *) malloc ((currLength + 1) *
109
110
                         sizeof (unsigned int));
111
          if (d[i] == NULL)
112
            {
113
              fprintf (stderr,
                 "\nMemory error --- couldn't allocate array!"
114
115
                 "\n%s\n", strerror (errno));
116
              fflush (stderr);
117
              exit (0);
118
119
          for (j = 0; j < currLength + 1; j++)
120
121
              d[i][j] = 0;
122
123
124
          return d;
125
      else if (dimToChange == 2)
126
127
        {
          for (i = 0; i < currSupport + 1; i++)</pre>
128
129
130
          d[i] =
            (unsigned int *) realloc (d[i],
131
132
                           (newVal + 1) * sizeof (unsigned int));
133
          if (d[i] == NULL)
134
            {
135
              fprintf (stderr,
                 "\nMemory error --- couldn't allocate array!"
136
137
                 "\n%s\n", strerror (errno));
138
              fflush (stderr);
139
              exit (0);
140
141
          for (j = currLength + 1; j < newVal + 1; j++)
142
143
              d[i][j] = 0;
144
145
146
          return d;
147
```

3.15.2.9 int measureDiagonal (const bitGraph_t * bg, const int i, const int j)

Given a bit graph, and two indices within that bit graph, this will return an integer which is equal to the number of values in the bit graph that are true along a diagonal that begins at the two indices. This routine is used to check for streaks in an adjacency matrix and is used during the convolution.

Definition at line 72 of file patStats.c.

References bitGraphCheckBit().

Referenced by getStatMat(), and oldGetStatMat().

```
73 {
74   int len = 0;
75   while (bitGraphCheckBit (bg, i + len, j + len) != 0)
76   {
77     len++;
78   }
79   return len;
80 }
```

3.15.2.10 unsigned int** oldGetStatMat (bitGraph_t * bg, int support, int length, int * supportDim, int * lengthDim, int numBlanks)

OK, here is something that is a little bit "hackish" but that we have to do. Since our initial matrix is being pruned and filtered before being clustered, but we need to calculate stats based on the original matrix, we need to get information from the matrix before pruning, so we're using this function. We could just make a copy of that matrix, but it's far too big, and that would cause an unneccessary constraint on memory, limiting the size of problems we can address. But we need to define just how big our d matrix is before we can use it. We could go through and compute the longest streak beforehand, and then redo everything, but we've already found the first step of finding all of the streaks to be fairly expensive (KLJ). So instead what we'll do is use the user's parameters as a benchmark and expand from there. We'll assume that most of the time, the biggest streak (number of extensions) will be less than 50 times the length given as input by the user, and the biggest support will be less than 50 times the minimum number of support given by the user. This seems perhaps overly conservative, but otherwise is reasonable. We then realize that even on a 64-bit computer, if the user gives

L=50 and K=50, we'll still use less than 48 MB of memory... and if L=50 and K=50, it is extremely likely that doubling the adjacency matrix would have been a much worse option. Scaling back to more common values of L \sim 20 and K \sim 20, the memory used shoots down to \sim 9MB, which is definitely acceptable. Now, if for some reason our initial allocation wasn't enough, then we'll have to go through and realloc all of our memory again. Somewhat time-consuming, but hopefully not done too often. Each time we find we try to put something in an index that doesn't exist, we'll reallocate our memory, adding twice as much in the dimension that was violated. It is important to us that we get back the final dimensions of this matrix, since in the support dimension we'll have to sum across all values, and in the length dimension we'll have to be sure we're not at the edge of a matrix during our d manipulations later on.

Definition at line 196 of file patStats.c.

References bitGraphRowIntersection(), countSet(), deleteBitSet(), increaseMem(), measureDiagonal(), newBitSet(), and bitGraph_t::size.

```
198 {
      int *Q = NULL;
199
      unsigned int **d = NULL;
200
201
      int i, j;
202
     int x, y;
203
     bitSet_t * X = NULL;
     int currSupport;
204
205
      int currLength;
206
     int multiplier = 50;
207
     time_t probStart, probEnd;
208
     int timeNeeded = 0;
209
     currSupport = support * multiplier;
210
      currLength = length * multiplier;
211
     X = newBitSet (bg->size);
212
        // printf("Made bitSet of size %d\n", bg->size);
213
214
        Q = (int *) malloc (bg->size * sizeof (int));
215
      if (Q == NULL)
216
        {
217
          fprintf (stderr,
218
            "\nMemory error --- couldn't allocate array!" "\n%s\n",
219
            strerror (errno));
220
          fflush (stderr);
221
          exit (0);
222
223
      for (i = 0; i < bg->size; i++)
224
        {
225
          Q[i] = 0;
226
        }
227
        (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
228
229
      if (d == NULL)
230
        {
231
          fprintf (stderr,
232
            "\nMemory error --- couldn't allocate array!" "\n%s\n",
233
            strerror (errno));
          fflush (stderr);
234
235
          exit (0);
```

```
236
      for (i = 0; i < currSupport + 1; i++)</pre>
237
238
239
          d[i] =
240
        (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
241
         if (d[i] == NULL)
242
243
          fprintf (stderr, "\nMemory error --- couldn't allocate array!"
244
                "\n%s\n", strerror (errno));
245
          fflush (stderr);
246
          exit (0);
247
248
          for (j = 0; j < currLength + 1; j++)
249
250
          d[i][j] = 0;
251
252
        }
253
      time (&probStart);
254
      for (i = 0; i < bg->size; i++)
255
256
          if (i == 200)
257
258
          time (&probEnd);
259
          timeNeeded = ((double) (probEnd - probStart)) /
260
            ((double) 60) * ((double) bg->size) / ((double) 200);
          if (timeNeeded > 2)
261
262
           {
263
              printf ("Max total time to calculate probability:\n");
264
              printf ("\t%d minutes\n", timeNeeded);
265
              printf ("Actual time will be less than this, but at",
266
                   "least half of it.\n");
267
              printf ("To bypass excessive probability calculations,",
268
                   "cancel and use the '-d' flag.n");
269
              fflush (NULL);
270
271
272
          for (j = bg->size - 1; j > i; j--)
273
274
          bitGraphRowIntersection (bg, i, j, X);
275
          x = countSet(X);
276
          if (Q[j - 1] != 0)
277
278
              y = Q[j - 1] - 1;
279
              Q[j] = Q[j - 1] - 1;
            }
280
281
          else
282
283
              y = measureDiagonal (bg, i, j);
284
              Q[j] = y;
285
286
          while (x > currSupport)
287
288
              d = increaseMem (d, 1, currSupport, currLength,
289
                    currSupport + support * multiplier);
              currSupport += support * multiplier;
290
291
292
          while (y > currLength)
```

```
293
294
295
            increaseMem (d, 2, currSupport, currLength,
296
                      currLength + length * multiplier);
297
              currLength += length * multiplier;
298
          d[x][y]++;
299
300
301
302
                if(x != 0) \{ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); \}
303
304
        }
305
306
307
            printf("done\n"); fflush(stdout);
308
309
        }
310
311
        \ensuremath{//} We know that the "blanks", inserted to delimit unique sequences
312
        // and prevent convolution through them, will skew our statistics,
        // so we subtract them. We know that they will never be similar to
313
        // any others, so will only add to the d[0][0] number. Furthermore,
314
315
        \ensuremath{//} we know how many they add. Since d never hits the main diagonal
316
        // and only does the upper half of the matrix, the first one
317
        // contributes bgsize - 1 to d[0][0], the next bgsize - 2, etc.
        for (i = 0; i < numBlanks; i++)</pre>
318
319
          d[0][0] -= bg->size - 1 - i;
320
321
322
      deleteBitSet (X);
323
      free (Q);
324
      *supportDim = currSupport;
325
      *lengthDim = currLength;
326
      return (d);
327 }
```

3.15.2.11 cll_t* sortByStats (cll_t * allCliqs)

This function is used to sort a link to list of cliques by the statistical significance of the motifs found in that linked list.

Definition at line 732 of file patStats.c.

References cnode::id, cnode::next, and statCompare().

```
733 {
734    cll_t * curCliq = NULL;
735    cll_t ** arrayOfCliqs = NULL;
736    int numOfCliqs = 0;
737    int i = 0;
738    curCliq = allCliqs;
739    if (curCliq != NULL)
```

```
740
741
         numOfCliqs = curCliq->id + 1;
742
743
      else
744
      {
745
         return (NULL);
746
      arrayOfCliqs = (cll_t **) malloc (numOfCliqs * sizeof (cll_t *));
747
748
      for (i = 0; i < numOfCliqs; i++)</pre>
749
750
         arrayOfCliqs[i] = curCliq;
         curCliq = curCliq->next;
751
752
753
      qsort (arrayOfCliqs, numOfCliqs, sizeof (cll_t *), statCompare);
754
      for (i = 0; i < numOfCliqs - 1; i++)
755
756
         arrayOfCliqs[i]->next = arrayOfCliqs[i + 1];
757
758
      arrayOfCliqs[numOfCliqs - 1]->next = NULL;
759
      return (arrayOfCliqs[0]);
760 }
```

3.15.2.12 int statCompare (const cll_t ** first, const cll_t ** second)

Definition at line 709 of file patStats.c.

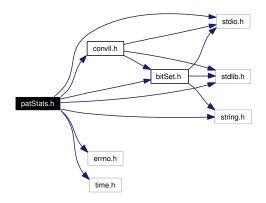
Referenced by sortByStats().

```
710 {
      double difference = (*first)->stat - (*second)->stat;
711
712
      if (difference < 0)
713
714
          return (-1);
715
716
      else if (difference > 0)
717
718
         return (1);
719
720
      else
721
722
          return (0);
723
724 }
```

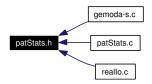
3.16 patStats.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "bitSet.h"
#include "convll.h"
#include <time.h>
```

Include dependency graph for patStats.h:



This graph shows which files directly or indirectly include this file:



Functions

- unsigned int ** getStatMat (bitGraph_t *bg, int support, int length, int *support-Dim, int *lengthDim, int numBlanks, int s, FILE *OUTPUT_FILE)
- int cumDMatrix (unsigned int **d, cll_t *cliqs, int currSupport, int currLength, int bgSize, int numSeqs)

- int calcStatAllCliqs (unsigned int **d, cll_t *allCliqs, int numWindows)
- cll_t * sortByStats (cll_t *allCliqs)
- int freeD (unsigned int **d, int supportDim)

3.16.1 Function Documentation

3.16.1.1 int calcStatAllCliqs (unsigned int ** d, cll_t * allCliqs, int numWindows)

Definition at line 623 of file patStats.c.

References calcStatCliq(), cnode::next, and cnode::stat.

Referenced by main().

3.16.1.2 int cumDMatrix (unsigned int ** d, cll_t * cliqs, int currSupport, int currLength, int bgSize, int numSeqs)

Definition at line 460 of file patStats.c.

References getLargestLength(), and getLargestSupport().

Referenced by main().

3.16.1.3 int freeD (unsigned int **d, int *supportDim*)

Definition at line 637 of file patStats.c.

Referenced by main().

3.16.1.4 unsigned int** getStatMat (bitGraph_t * bg, int support, int length, int * supportDim, int * lengthDim, int numBlanks, int s, FILE * OUTPUT_FILE)

Definition at line 289 of file patStats.c.

References bitGraphRowIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, increaseMem(), measureDiagonal(), newBitSet(), nextBitBitSet(), and bitGraph_t::size.

Referenced by main().

3.16.1.5 cll_t* sortByStats (cll_t * allCliqs)

This function is used to sort a link to list of cliques by the statistical significance of the motifs found in that linked list.

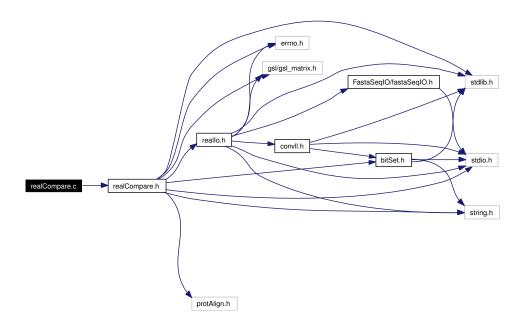
Definition at line 674 of file patStats.c.

References cnode::id.

3.17 realCompare.c File Reference

#include "realCompare.h"

Include dependency graph for realCompare.c:



Functions

- double rmsdCompare (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double generalMatchFactor (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double massSpecCompareWElut (rdh_t *data, int win1, int win2, int L, double *extraParams)
- double(*)(rdh_t *, int, int, double *) getCompFunc (int compFunc)
- bitGraph_t * realComparison (rdh_t *data, int L, double g, int compFunc, double *extraParams)

3.17.1 Detailed Description

This file defines a series of functions that are used during the comparison phase of the Gemoda algorithm in the real valued implementation. We define a handful of compar-

ison functions — some that are well suited to protein structure comparison and others that are more suited to the comparison of mass spectrometry spectra.

Definition in file realCompare.c.

3.17.2 Function Documentation

3.17.2.1 double generalMatchFactor (rdh_t * data, int win1, int win2, int L, double * extraParams)

This function is used to compute a generalized match factor, which is useful for computing the degree of similarity between mass spectrometry spectra.

Definition at line 111 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```
113 {
114
      int i, j;
115
      double numerator = 0.0;
116
117
           double denominator=0.0;
118
119
120 double xsum;
     double ysum;
121
122
     double ldenom = 0.0;
123
     double rdenom = 0.0;
124
     int dim;
125
     int seq1, pos1;
    int seq2, pos2;
126
127
     gsl_matrix_view view1;
     gsl_matrix_view view2;
128
129
     gsl_matrix * mat1;
130 gsl_matrix * mat2;
131
     dim = getRdhDim (data);
132
133
        // Find out which seq,pos pairs these two
134
       // windows correspond to
        getRdhIndexSeqPos (data, win1, &seq1, &pos1);
135
136
     getRdhIndexSeqPos (data, win2, &seq2, &pos2);
137
138
        // Get a reference to a submatrix. That is,
139
       // 'chop out' the window.
140
       view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
141
     view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
142
143
        // Some error checking here would be nice!
144
        // Did we get the matrices we wanted?
145
146
        // This just makes it easier to handle the views
147
        mat1 = &view1.matrix;
```

```
148
      mat2 = &view2.matrix;
149
150
        // Loop over each position
151
        for (i = 0; i < mat1->size1; i++)
152
153
          xsum = 0.0;
         ysum = 0.0;
154
155
156
       // Loop over each dimension at each position
157
       for (j = 0; j < dim; j++)
158
159
         xsum += gsl_matrix_get (mat1, i, j);
160
         ysum += gsl_matrix_get (mat2, i, j);
161
162
         numerator += (i + 1) * sqrt (xsum * ysum);
163
          ldenom += (i + 1) * xsum;
         rdenom += (i + 1) * ysum;
164
165
166
     return pow (numerator, 2.0) / (ldenom * rdenom);
167 }
```

3.17.2.2 double(*)(rdh_t *, int, int, double *) getCompFunc ()

Definition at line 264 of file realCompare.c.

References generalMatchFactor(), massSpecCompareWElut(), and rmsdCompare().

```
265 {
266
      double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
267
     switch (compFunc)
268
       {
269
       case 0:
         comparisonFunc = &rmsdCompare;
270
271
       case 1:
272
273
         comparisonFunc = &generalMatchFactor;
         break;
274
275
       case 2:
276
         comparisonFunc = &massSpecCompareWElut;
277
         break;
278
       default:
279
         comparisonFunc = &rmsdCompare;
280
         break;
281
282
     return (comparisonFunc);
283 }
```

3.17.2.3 double massSpecCompareWElut (rdh_t * data, int win1, int win2, int L, double * extraParams)

This function is used to compute the match factor between to mass spectrometry spectra in a similar manner to the previous function; however, this function imposes a penalty

for spectra that are separated by large distances in elution time. This function is commonly used by SpecConnect.

Definition at line 178 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```
180 {
181
      int i, j;
182
     double numerator = 0.0;
183
184
185
           double denominator=0.0;
         * /
186
187
     double xsum;
188
     double ysum;
189
     double cum;
190
     double ldenom = 0.0;
191
     double rdenom = 0.0;
192
     int dim;
193
     int seq1, pos1;
194
     int seq2, pos2;
195
     double weight = 2.0;
     gsl_matrix_view view1;
196
197
     gsl_matrix_view view2;
     gsl_matrix * mat1;
198
199
     qsl matrix * mat2;
200
     double maxElut = -1;
201
     if (extraParams != NULL)
202
          maxElut = extraParams[0];
203
204
     dim = getRdhDim (data);
205
206
207
        // Find out which seq,pos pairs these two
208
        // windows correspond to
209
        getRdhIndexSeqPos (data, win1, &seq1, &pos1);
210
     getRdhIndexSeqPos (data, win2, &seq2, &pos2);
211
212
        // Get a reference to a submatrix. That is,
        // 'chop out' the window.
213
214
        view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
     view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
215
216
217
        // Some error checking here would be nice!
218
        // Did we get the matrices we wanted?
219
220
        // This just makes it easier to handle the views
221
        mat1 = &view1.matrix;
     mat2 = &view2.matrix;
222
223
      cum = 1.0;
224
225
        // Loop over each position
226
        for (i = 0; i < mat1->size1; i++)
227
        {
```

```
228
          xsum = 0.0;
229
          ysum = 0.0;
230
231
        // First take the first dimension for elution time
232
        if (maxElut >= 0)
233
        {
234
          if (fabs
235
               (gsl_matrix_get (mat1, i, 0) - gsl_matrix_get (mat2, i, 0)) >
236
               maxElut)
237
              cum = 0;
238
239
              break;
240
241
        }
242
243
        // printf("\n");
244
        //
245
        // Loop over each subsequent dimension at each position
246
        for (j = 1; j < dim; j++)
247
248
            // printf("mat1val=%lf,mat2val=%lf\n",gsl_matrix_get(mat1,i,j),
249
250
            // gsl_matrix_get(mat2,i,j));
251
            numerator += pow (j, weight) * sqrt (gsl_matrix_get (mat1, i, j)
252
                             *gsl_matrix_get (mat2, i,
253
                                      j));
254
          ldenom += pow (j, weight) * gsl_matrix_get (mat1, i, j);
          rdenom += pow (j, weight) * gsl_matrix_get (mat2, i, j);
255
256
257
            // printf("numer=%lf,ldenom=%lf,rdenom=%lf\n",numerator,
258
            // ldenom.rdenom);
259
260
          cum *= pow (numerator, 2.0) / (ldenom * rdenom);
261
262
      return pow (cum, 1.0 / L);
263 }
```

3.17.2.4 bitGraph_t* realComparison (rdh_t * data, int L, double g, int compFunc, double * extraParams)

Definition at line 285 of file realCompare.c.

References bitGraphSetTrueSym(), getCompFunc, getRdhIndexSeqPos(), rdh_-t::indexSize, initRdhIndex(), newBitGraph(), and rmsdCompare().

```
287 {
288    int i, j;
289    int seq1, pos1;
290    int seq2, pos2;
291    bitGraph_t * bg = NULL;
292    double score;
293    double (*comparisonFunc) (rdh_t *, int, int, double *) = &rmsdCompare;
```

```
294
295
        // Initialize the rdh's index
296
        initRdhIndex (data, L, 1);
297
298
        // Allocate a new bit graph
299
        bg = newBitGraph (data->indexSize);
300
301
        // Choose the comparison function, pass a reference to it
302
        comparisonFunc = getCompFunc (compFunc);
303
      for (i = 0; i < data->indexSize; i++)
304
305
306
        // Skip seperators
307
        getRdhIndexSeqPos (data, i, &seq1, &pos1);
308
          if (seq1 == -1 || pos1 == -1)
309
310
          continue;
311
312
          for (j = i; j < data->indexSize; j++)
313
314
          getRdhIndexSeqPos (data, j, &seq2, &pos2);
315
          if (seq2 == -1 || pos2 == -1)
316
317
              continue;
318
319
320
            // This is the comparison function
321
            score = comparisonFunc (data, i, j, L, extraParams);
322
323
            // printf("score (%2d,%2d) vs. (%2d, %2d) =\t%1f\n",seq1, pos1, seq2, pos2,
324
            // score);
325
            if (compFunc == 0)
326
327
              if (score <= g)
328
329
              bitGraphSetTrueSym (bg, i, j);
330
331
332
          else if ((compFunc == 1) || (compFunc == 2))
333
334
              if (score >= g)
335
336
              bitGraphSetTrueSym (bg, i, j);
337
338
339
          else
340
341
              fprintf (stderr, "Comparison function undefined in "
342
                 "realComparison function, \n located in "
343
                "realCompare.c. Exiting.\n\n");
344
              fflush (stderr);
345
              exit (0);
            }
346
347
348
349
      return bg;
350 }
```

3.17.2.5 double rmsdCompare (rdh_t * data, int win1, int win2, int L, double * extraParams)

Calculate the rmsd between two windows, with optional translation and rotation. The input to this function is a real data handler object, two integers that point to the windows within the real data that are to be compared, an integer that specifies the length of the windows, and a pointer to a double precision floating point that can be used to store other parameters as needed. This last parameter is most useful for implementing other comparison functions, without having to make, too many changes to other parts of the code.

This function operates in three stages. First, we compute the centroid of each window and move the second window such that its centroid overlaps with that of the first window. Second, we use rigid body rotation to find the rotational matrix that minimizes the root mean squared deviation between the two windows. Finally, this function returns that minimized RMSD.

Definition at line 31 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc(), and realComparison().

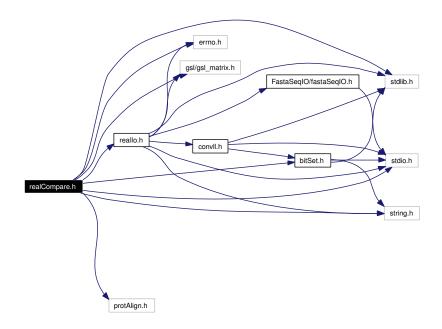
```
32 {
    int trans = 1;
33
34
    int rot = 1;
35
    int dim;
36
    double result = 0;
37
     int seq1, pos1;
38
    int seq2, pos2;
39
    gsl_matrix_view view1;
40
    gsl_matrix_view view2;
41
    gsl_matrix * mat1;
42
     gsl_matrix * mat2;
    gsl_matrix * matlcopy;
43
44
    gsl_matrix * mat2copy;
45
46
       // The "rint" function is in math.h and rounds a number to the
47
       // nearest integer. It raises an "inexact exception" if the
       // number initially wasn't an integer.
48
49
       if (extraParams != NULL)
50
51
         trans = rint (extraParams[0]);
52
         rot = rint (extraParams[1]);
53
       }
54
    dim = getRdhDim (data);
55
       // Find out which seq,pos pairs these two
56
57
       // windows correspond to
58
       getRdhIndexSeqPos (data, win1, &seq1, &pos1);
59
    getRdhIndexSeqPos (data, win2, &seq2, &pos2);
60
       // Get a reference to a submatrix. That is,
61
62
       // 'chop out' the window.
```

```
view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
    view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
64
66
       \ensuremath{//} This just makes it easier to handle the views
67
      mat1 = &view1.matrix;
68
    mat2 = &view2.matrix;
69
70
       // Create copies of the windows, because our comparison
71
      // will require altering the matrices
72
      matlcopy = gsl_matrix_alloc (mat1->size1, mat1->size2);
73
     mat2copy = gsl_matrix_alloc (mat2->size1, mat2->size2);
74
     gsl_matrix_memcpy (matlcopy, matl);
75
     gsl_matrix_memcpy (mat2copy, mat2);
76
77
78
           printf("matrix1:\n"); gsl_matrix_pretty_fprintf(stdout, matlcopy, "%f ");
79
          printf("\nmatrix2:\n"); gsl_matrix_pretty_fprintf(stdout, mat2copy, "%f ");
80
81
82
       // Are we going to do a translation?
83
       if (trans == 1)
84
        moveToCentroid (mat1copy);
85
86
        moveToCentroid (mat2copy);
87
88
89
       // Are we going to do a rotation?
       if (rot == 1)
90
91
92
       // Rotate mat2copy to have a minimal
93
94
       // rmsd with matlcopy
95
       rotateMats (mat1copy, mat2copy);
96
97
98
      // Compute the rmsd between mat2copy and mat2copy
99
      result = gsl_matrix_rmsd (mat1copy, mat2copy);
100
     gsl_matrix_free (matlcopy);
     gsl_matrix_free (mat2copy);
102
     return result;
103 }
```

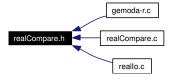
3.18 realCompare.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "realIo.h"
#include "bitSet.h"
#include "protAlign.h"
```

Include dependency graph for realCompare.h:



This graph shows which files directly or indirectly include this file:



Functions

- double rmsdCompare (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double generalMatchFactor (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double massSpecCompareWElut (rdh_t *data, int win1, int win2, int L, double *extraParams)
- bitGraph_t * realComparison (rdh_t *data, int l, double g, int compFunc, double *extraParams)

Variables

• double(*)(rdh_t *, int, int, double *) getCompFunc (int compFunc)

3.18.1 Detailed Description

This file contains declarations and definitions used for the comparison of real valued data during the comparison phase of Gemoda. The functions declared here are defined in realCompare.c.

Definition in file realCompare.h.

3.18.2 Function Documentation

3.18.2.1 double generalMatchFactor (rdh_t * data, int win1, int win2, int L, double * extraParams)

This function is used to compute a generalized match factor, which is useful for computing the degree of similarity between mass spectrometry spectra.

Definition at line 111 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

3.18.2.2 double massSpecCompareWElut (rdh_t * data, int win1, int win2, int L, double * extraParams)

This function is used to compute the match factor between to mass spectrometry spectra in a similar manner to the previous function; however, this function imposes a penalty for spectra that are separated by large distances in elution time. This function is commonly used by SpecConnect.

Definition at line 174 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

3.18.2.3 **bitGraph_t*** realComparison (rdh_t * data, int l, double g, int compFunc, double * extraParams)

Definition at line 272 of file realCompare.c.

References bitGraphSetTrueSym(), getCompFunc, getRdhIndexSeqPos(), rdh_-t::indexSize, initRdhIndex(), newBitGraph(), and rmsdCompare().

Referenced by main().

3.18.2.4 double rmsdCompare (rdh_t * data, int win1, int win2, int L, double * extraParams)

Calculate the rmsd between two windows, with optional translation and rotation. The input to this function is a real data handler object, two integers that point to the windows within the real data that are to be compared, an integer that specifies the length of the windows, and a pointer to a double precision floating point that can be used to store other parameters as needed. This last parameter is most useful for implementing other comparison functions, without having to make, too many changes to other parts of the code.

This function operates in three stages. First, we compute the centroid of each window and move the second window such that its centroid overlaps with that of the first window. Second, we use rigid body rotation to find the rotational matrix that minimizes the root mean squared deviation between the two windows. Finally, this function returns that minimized RMSD.

Definition at line 31 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc(), and realComparison().

3.18.3 Variable Documentation

3.18.3.1 double(*)(rdh_t*, int, int, double*) getCompFunc(int compFunc)

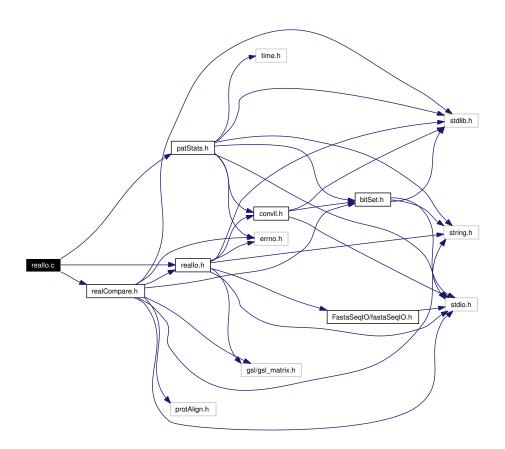
Definition at line 36 of file realCompare.h.

Referenced by findCliqueCentroid(), outputRealPatsWCentroid(), and real-Comparison().

3.19 realIo.c File Reference

```
#include "realIo.h"
#include "realCompare.h"
#include "patStats.h"
```

Include dependency graph for realIo.c:



Functions

- wordToDouble (char *s, int begin, int end)
- int countFields (char *s, char sep)
- int checkRealDataFormat (char **buf, int nl, char sep, int *numSeq_p, int *dim_p)
- int countTotalFields (char **buf, int nl, char sep)

- rdh_t * initRdh (int x)
- int getRdhSeqLength (rdh_t *data, int seqNo)
- int initRdhIndex (rdh_t *data, int wordSize, int seqGap)
- rdh_t * freeRdh (rdh_t *data)
- int getRdhDim (rdh t *data)
- int setRdhLabel (rdh_t *data, int seqNo, char *s)
- int setRdhValue (rdh_t *data, int seqNo, int posNo, int dimNo, double val)
- int setRdhIndex (rdh_t *data, int seqNo, int posNo, int index)
- int getRdhIndexSeqPos (rdh_t *data, int index, int *seq, int *pos)
- double getRdhValue (rdh t *data, int seqNo, int posNo, int dimNo)
- char * getRdhLabel (rdh_t *data, int seqNo)
- int printRdhSeq (rdh_t *data, int seqNo, FILE *FH)
- int setRdhColFromString (rdh_t *data, int seqNo, int colNo, char *s, char sep)
- int initRdhGslMat (rdh_t *data, int seqNo, int x, int y)
- int pushOnRdhSeq (rdh_t *data, char **buf, int startLine, int dim, char sep)
- rdh_t * parseRealData (char **buf, int nl, char sep, int numSeq, int dim)
- rdh_t * readRealData (FILE *INPUT)
- int outputRealPats (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, int **d)
- int findCliqueCentroid (rdh_t *data, cll_t *curCliq, int L, int compFunc, double *extraParams, int *candidates)
- int makeAlternateCentroid (rdh_t *data, cll_t *curCliq, int *candidates)
- int outputRealPatsWCentroid (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, double *extraParams, int compFunc)

3.19.1 Detailed Description

This file defines functions that are used for the parsing of user supplied data in the real valued implementation of Gemoda.

Definition in file reallo.c.

3.19.2 Function Documentation

3.19.2.1 int checkRealDataFormat (char ** buf, int nl, char sep, int * numSeq_p, int * dim_p)

Check that each sequence has the same dimensionality and that, within a sequence, each dimension has the same number of entries. Note: this routine alters *nunSeq_p and *dim_p! Also, you must call this routine before calling parseRealData. Otherwise, parseRealData is garunteed to die if the data turn out to be ill-formatted.

Definition at line 163 of file realIo.c.

References countFields().

Referenced by readRealData().

```
164 {
165
      int i;
166
      int thisDim = 0;
167
      int status = 1;
     int width;
168
169
     int fieldCount = 0;
                                // number of positions in a single sequence
     int numSeq = 0;
                             // number of sequences
170
171
      int dim = 0;
                             // The dimensionality of the sequences
172
173
      // NOTE this is not checking the dimensionality of the last sequence...
174
      // that's bad. We can fix that though.
175
      // Check the dimensionality of each sequence
176
      for (i = 0; i < nl; i++)
177
        {
178
          if (buf[i][0] == '>')
179
        {
180
181
          // If this is only the second sequence we've seen,
182
          // record the dimensionality of the first sequence
          // as the dim to insist upon from here on out
183
184
          if (numSeq == 1)
185
186
              dim = thisDim;
187
188
              // For other sequences, we need to check to make sure
189
              // that they've got the same dimensions as previous
190
              // sequences
191
          else if (numSeq > 1)
192
193
            {
194
195
              // If the dimensions are wrong, quit with status=0
196
              if (thisDim != dim)
197
198
              status = 0;
199
              break;
200
201
          numSeq++;
202
203
          width = 0;
204
          thisDim = 0;
205
206
          else
207
208
209
          // Field count can be different for each sequence but
210
          // must be the same for each dimension in a single sequence
          fieldCount = countFields (buf[i], sep);
211
212
213
          // If this is the first row of this sequence,
214
          // then store the number of fields
215
          if (thisDim == 0)
216
            {
```

```
217
              width = fieldCount;
218
219
              // If it's not the first row, make sure it has the
220
              // same number of fields as previous rows in this
221
              // sequence
222
223
         else
224
225
              if (fieldCount != width)
226
227
              status = 0;
228
             break;
229
230
231
          thisDim++;
232
233
234
235
     // Pass back the numSeq and dim
236
      *numSeq_p = numSeq;
     *dim_p = thisDim;
237
238
     return status;
239 }
```

3.19.2.2 int countFields (char *s, char sep)

Count the number of fields (delimited by 'sep') in a single string. I was going to use strsep in string.h for this; however, I don't like that it changes the input string, which makes free-ing the string later more tricky. Ignores consecutive seperators.

Definition at line 90 of file realIo.c.

References wordToDouble().

Referenced by checkRealDataFormat(), countTotalFields(), and pushOnRdhSeq().

```
91 {
92
    int i;
93
    int begin = 0;
    int end = 0;
94
95
    int status = 0;
                          // 0 = in sep, 1 = in word
96
    int fieldCount = 0;
97
    double val;
98
    if (s == NULL)
99
100
          fprintf (stderr, "Passed NULL string to countFields -- error!");
101
         fflush (stderr);
102
          exit (0);
103
104
105
      // Loop over the length of the string
      for (i = 0; i < strlen (s); i++)
106
107
108
```

```
109
          // The previous state was space
          if (status == 0)
110
        {
111
112
113
          // We hit a word
114
          if (s[i] != sep)
115
116
              begin = i;
              status = 1;
117
118
119
          else
120
            {
                         // We hit more space
121
              continue;
122
            }
123
        }
124
          else
125
                    // The previous state was word
          if (s[i] != sep)
126
127
            {
128
              continue;
            }
129
130
          else
                         // We hit a space
131
              end = i - 1;
132
133
              status = 0;
134
135
              // being and end now delimit a word,
              // turn that word into a double
136
137
              val = wordToDouble (s, begin, end);
138
              fieldCount++;
            }
139
140
        }
141
142
      // At the end, if we were in a word, we have
143
144
      // one more field
145
      if (status == 1)
146
                         // We're in a word
147
          val = wordToDouble (s, begin, strlen (s));
148
          fieldCount++;
149
        }
150
      return fieldCount;
151 }
```

3.19.2.3 int countTotalFields (char ** buf, int nl, char sep)

Count the number of fields in each sequence and return the sum of these.

Definition at line 246 of file realIo.c.

References countFields().

Referenced by parseRealData().

247 {

```
248
      int i = 0;
     int totalFields = 0;
249
250
      int seqNo = 0;
251
      while (i < nl)
252
       {
253
254
          // Hit a new sequence
255
          if (buf[i][0] == '>')
256
257
          seqNo++;
258
259
          // Assume that the sequence has at least
260
          // one row (should have called checkRealDataFormat!
261
          // and that each row has the same number of fields
262
          totalFields += countFields (buf[i + 1], sep);
263
264
          i++;
265
266
     return totalFields;
267 }
```

3.19.2.4 int findCliqueCentroid (rdh_t * data, cll_t * curCliq, int L, int compFunc, double * extraParams, int * candidates)

This function is used to find the centroid of a clique. That is, to find the center of mass.

Definition at line 1096 of file realIo.c.

References getCompFunc, cSet_t::members, cnode::set, and cSet_t::size.

Referenced by outputRealPatsWCentroid().

```
1098 {
1099
       double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
      int i = 0, j = 0, indmin = -1, counter = 0;
1100
1101
       double sim = 0, min = 0, flagmin = 0;
1102
      double *cliqueAdjMat = NULL;
1103
      cliqueAdjMat = (double *) malloc (curCliq->set->size * sizeof (double));
1104
       if (cliqueAdjMat == NULL)
1105
1106
           fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1107
           fflush (stderr);
1108
           exit (0);
1109
1110
      for (i = 0; i < curCliq->set->size; i++)
1111
        {
1112
           cliqueAdjMat[i] = 0;
1113
         }
1114
1115
       // We'll accumulate our comparison function values... except here
1116
      // we're really assuming that we're using a match factor, with
1117
       // value less than one, so that we can subtract it from one to
      // get a distance, and then find the centroid by identifying the
1118
1119
      // node with the smallest cumulative Euclidean distance to all
```

```
1120
       // nodes.
       // Note that we only need to compare each unique pair, and can apply
1121
1122
       // the results from each comparison to each member of the pair,
1123
       // hence the somewhat odd indices of initiation for the for loops.
1124
       comparisonFunc = getCompFunc (compFunc);
1125
       for (i = 0; i < curCliq->set->size; i++)
1126
1127
           for (j = i + 1; j < curCliq->set->size; j++)
1128
         {
1129
           sim =
1130
             comparisonFunc (data, curCliq->set->members[i],
1131
                     curCliq->set->members[j], L, extraParams);
1132
1133
           // printf("i = %d, j = %d, L = %d, extra = %lf, sim =
1134
           // %lf\n",i,j,L,extraParams[0],sim);
1135
           cliqueAdjMat[i] += pow (1 - sim, 2);
           cliqueAdjMat[j] += pow (1 - sim, 2);
1136
1137
1138
1139
       // Now we find the minimum Euclidean distance.
1140
1141
       min = cliqueAdjMat[0];
       indmin = 0;
1142
1143
       for (i = 1; i < curCliq->set->size; i++)
1144
1145
1146
           // printf("index %d product = %lf\n",i,cliqueAdjMat[i]);
1147
           if (cliqueAdjMat[i] < min)</pre>
1148
1149
           indmin = i;
1150
           min = cliqueAdjMat[i];
1151
           flagmin = 0;
1152
1153
           else if (cliqueAdjMat[i] == min)
1154
1155
           flagmin = 1;
1156
1157
1158
1159
       // If we had a duplicate on the minimum, we locate all duplicates.
1160
       if (flagmin == 1)
1161
1162
           counter = 0;
1163
           for (i = 0; i < curCliq->set->size; i++)
1164
1165
           if (cliqueAdjMat[i] == min)
1166
1167
               counter++;
1168
               candidates[counter] = i;
1169
1170
         }
1171
1172
           // Store the number of candidates at the array's beginning
1173
           candidates[0] = counter;
1174
           free (cliqueAdjMat);
1175
           return (-1);
1176
```

3.19.2.5 $rdh_t* freeRdh (rdh_t* data)$

This function returns a null pointer after freeing the memory associated with a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 462 of file realIo.c.

References rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::offsetToIndex, and rdh_t::seq.

```
463 {
464
      int i;
465
     if (data != NULL)
466
467
          if (data->indexToPos != NULL)
468
469
          free (data->indexToPos);
470
          data->indexToPos = NULL;
471
472
          if (data->indexToSeq != NULL)
473
        {
474
          free (data->indexToSeq);
475
          data->indexToSeq = NULL;
476
477
          if (data->offsetToIndex != NULL)
478
479
          for (i = 0; i < data->size; i++)
480
481
              free (data->offsetToIndex[i]);
482
              data->offsetToIndex[i] = NULL;
483
484
          free (data->offsetToIndex);
485
          data->offsetToIndex = NULL;
486
487
          for (i = 0; i < data->size; i++)
488
        {
489
          if (data->seq[i] != NULL)
490
            {
491
              gsl_matrix_free (data->seg[i]);
              data->seq[i] = NULL;
492
493
494
          if (data->label[i] != NULL)
495
496
              free (data->label[i]);
497
              data->label[i] = NULL;
```

```
498
499
500
          if (data->seq != NULL)
501
502
          free (data->seq);
503
          data->seq = NULL;
504
505
          if (data->label != NULL)
506
507
          free (data->label);
508
          data->label = NULL;
509
510
          free (data);
511
          data = NULL;
512
513
      return data;
514 }
```

3.19.2.6 int getRdhDim (rdh_t * data)

This function returns an integer equal to the dimensions of the data stored in a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 524 of file realIo.c.

References rdh_t::seq.

Referenced by generalMatchFactor(), getRdhValue(), massSpecCompareWElut(), printRdhSeq(), rmsdCompare(), and setRdhValue().

```
525 {
526    if (data == NULL || data->seq == NULL || data->seq[0] == NULL)
527    {
528         fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
529         fflush (stderr);
530         exit (0);
531    }
532    return data->seq[0]->size2;
533 }
```

3.19.2.7 int getRdhIndexSeqPos (rdh_t * data, int index, int * seq, int * pos)

This function is used to access and change the sequence and position values, given an index. The function takes four parameters: a pointer to the real data holder, *data*, an integer *index*, a pointer integer *seq*, and a pointer integer *pos*.

Definition at line 633 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

Referenced by generalMatchFactor(), makeAlternateCentroid(), massSpecCompare-WElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsd-Compare().

```
634 {
635
      if (data == NULL | data->indexToSeq == NULL | data->indexToPos == NULL
          || index > data->indexSize)
636
637
638
          fprintf (stderr, "Passed bad data to getRdhIndexSeqPos -- error!");
639
          fflush (stderr);
640
          exit (0);
641
642
643
644
        printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
645
646
647
        fflush(stdout);
648
      *seq = data->indexToSeq[index];
649
650
      *pos = data->indexToPos[index];
      return 0;
651
652 }
```

3.19.2.8 char* getRdhLabel (rdh_t * data, int seqNo)

This function is used to retrieve the label of a particular sequence in a real data holder object. The function takes two parameters: a pointer to the real data holder *data*; and an integer which is the sequence number to be accessed *seqNo*. The function returns a pointer to a string, which is the label for that sequence.

Definition at line 689 of file realIo.c.

References rdh t::label.

Referenced by printRdhSeq().

```
690 {
691    if (data == NULL || data->label == NULL || data->label[seqNo] == NULL)
692    {
693         fprintf (stderr, "Passed bad data to getRdhLabel -- error!");
694         fflush (stderr);
695         exit (0);
696    }
697    return data->label[seqNo];
698 }
```

3.19.2.9 int getRdhSeqLength (rdh_t * data, int seqNo)

This function returns an integer that is equal to the sequence length of a particular sequence within the real data holder object. The function takes two parameters: a

pointer to the real data holder, *data*, and the index of the sequence for which we need to know the length, *seqNo*.

Definition at line 331 of file realIo.c.

References rdh_t::seq.

Referenced by getRdhValue(), initRdhIndex(), printRdhSeq(), and setRdhValue().

3.19.2.10 double getRdhValue (rdh_t * data, int seqNo, int posNo, int dimNo)

This function is used to retrieve the value of a particular dimension, position, and sequence. The function takes four parameters: a pointer to the real data holder *data*; an integer which is the sequence number to be accessed *seqNo*; an integer that is the position number to be accessed *posNo*; and an integer that is the dimension to be accessed *dimNo*.

Definition at line 666 of file realIo.c.

References getRdhDim(), getRdhSeqLength(), and rdh_t::seq.

Referenced by printRdhSeq().

```
667 {
668
      if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
          || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
669
670
          fprintf (stderr, "Passed bad data to getRdhValue -- error!");
671
672
          fflush (stderr);
673
          exit (0);
674
      return gsl_matrix_get (data->seq[seqNo], posNo, dimNo);
675
676 }
```

3.19.2.11 $rdh_t* initRdh (int x)$

This function initializes a real data holder object. The function takes as its input a size *x* which is the number of sequences that will be stored in the object. The function returns a pointer to the object, which has been allocated the correct amount of memory.

Definition at line 277 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::seq, and rdh_t::size.

Referenced by parseRealData().

```
278 {
279
      int i;
      rdh_t *data = NULL;
280
282
      // Allocate space for our structure
      data = (rdh_t *) malloc (sizeof (rdh_t));
283
284
      if (data == NULL)
285
286
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
287
          fflush (stderr);
288
          exit (0);
289
290
     data -> size = x;
291
292
     // Index has to be initialized later, once
293
      // we know the word size.
294
     data->indexSize = 0;
295
      data->indexToSeq = NULL;
296
      data->indexToPos = NULL;
297
298
299
        data->indexSize = y;
300
      data->label = (char **) malloc (data->size * sizeof (char *));
301
302
      if (data->label == NULL)
303
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
304
305
          fflush (stderr);
306
          exit (0);
307
308
      data->seq = (gsl_matrix **) malloc (data->size * sizeof (gsl_matrix *));
309
      if (data->seq == NULL)
310
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
311
312
          fflush (stderr);
313
          exit (0);
314
315
      for (i = 0; i < data->size; i++)
316
317
          data->label[i] = NULL;
318
          data->seq[i] = NULL;
319
320
      return data;
321 }
```

3.19.2.12 int initRdhGslMat ($rdh_t * data$, int seqNo, int x, int y)

This function is used to initialize the memory for the matrix in which the real value to data are stored. To store these data, we use the GNU scientific library. The function takes four parameters: a pointer to the real data holder data; an integer, which is the sequence number to be set seqNo; an integer, which is the first dimension of the matrix size x; and an integer, which is the second dimension of the matrix size y;

Definition at line 829 of file realIo.c.

References rdh_t::seq.

Referenced by pushOnRdhSeq().

```
830 {
831
      data->seq[seqNo] = gsl_matrix_alloc (x, y);
832
      if (data->seq[seqNo] == NULL)
833
834
          return 0;
835
836
      else
837
        {
838
          return 1;
839
840 }
```

3.19.2.13 int initRdhIndex (rdh_t * data, int wordSize, int seqGap)

This function is used to initialize the two indices inside a real data holder. The function takes as its input three parameters a pointer to the real data holder, *data*, the size of the words to be compared during the comparison stage *wordSize*, and an integer *seq-Gap*, which is used to place empty data between unique sequences, such that we do not convolve from one sequence into another during the convolution stage.

Definition at line 358 of file realIo.c.

References getRdhSeqLength(), rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexTo-Seq, rdh_t::offsetToIndex, and rdh_t::size.

Referenced by realComparison().

```
359 {
360
      int i, j, k;
361
     int numWindows = 0;
362
      int thisNumWindows;
363
      int numSeq;
364
     int seqLen = 0;
365
366
      // The number of sequences
367
      numSeq = data->size;
368
```

```
369
      // Allocate offsetToIndex's outer structure
      data->offsetToIndex = (int **) malloc (numSeq * sizeof (int *));
370
371
      if (data->offsetToIndex == NULL)
372
373
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
374
          fflush (stderr);
375
          exit (0);
376
377
378
      // For each sequence
379
      for (i = 0; i < numSeq; i++)
380
        {
381
382
          \ensuremath{//} How many windows are in this sequence
383
          seqLen = getRdhSeqLength (data, i);
          numWindows += seqLen - wordSize + 1;
384
385
386
          // And also use this to further allocate offsetToIndex
          data->offsetToIndex[i] =
387
388
        (int *) malloc ((seqLen - wordSize + 1) * sizeof (int));
389
          if (data->offsetToIndex[i] == NULL)
390
391
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
392
          fflush (stderr);
393
          exit (0);
394
395
396
397
      // One index for each word plus seqGap between each sequence
398
      // and a gap at the end
399
      data->indexSize = numWindows + numSeq * seqGap;
400
401
      // Allocate indexToSeq
      // NOTE that it should be size of int, not int *\dots I think we got
402
403
      // fortunate in the previous revision because they are the same
404
     // size
405
      data->indexToSeq = (int *) malloc (data->indexSize * sizeof (int));
      if (data->indexToSeq == NULL)
406
407
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
408
409
          fflush (stderr);
410
          exit (0);
        }
411
412
413
      // Allocate indexToPos
414
      // See above for int vs. int* argument.
415
      data->indexToPos = (int *) malloc (data->indexSize * sizeof (int));
416
      if (data->indexToPos == NULL)
417
418
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
419
          fflush (stderr);
420
          exit (0);
421
422
      \ensuremath{//} Fill in the values
423
424
      k = 0;
425
      for (i = 0; i < numSeq; i++)
```

```
426
        {
427
428
          // How many windows are in this sequence?
429
          thisNumWindows = getRdhSeqLength (data, i) - wordSize + 1;
430
431
           // For each window, make an entry in the indexToSeq
432
          // and indexToPos and offsetToIndex
433
          for (j = 0; j < thisNumWindows; j++)</pre>
434
435
          data->indexToSeq[k] = i;
436
          data->indexToPos[k] = j;
437
          data->offsetToIndex[i][j] = k;
438
439
        }
440
          \ensuremath{//} Add gaps between sequences in the index.
441
442
          // Usually segGap is just 1;
443
          for (j = 0; j < seqGap; j++)
444
445
          // -1 means no sequence and no position
446
447
          data->indexToSeq[k] = -1;
448
          data->indexToPos[k] = -1;
449
          k++;
450
451
452
      return 0;
453 }
```

3.19.2.14 int makeAlternateCentroid (rdh_t * data, cll_t * curCliq, int * candidates)

This function is used to choose an alternate centroid for a given clique. In order to make the centroid decision slightly less dependent on input order, we decide to choose from the tied candidates the one whose relative position in the sequence is highest. There is no basis in theory for this, it is done so that a consistent choice is made. Only rarely will two spectra be tied for being a centroid and have the same sequence number. In that case, we pretty much have to default to the sequence number, which is what would be done without this function. Note that now though we are less sensitive to the order of input of the sequences, we are now more sensitive to the context surrounding a given spectrum. That is, if it is put in the beginning of the sequence, it is more likely to be chosen. This choice can only be justified insofar as if multiple choices are tied, then they are the same cumulative distance to the clique, and so *any* should be allowed to be chosen equally. There should be little difference in terms of tangible results. This just makes the semantics consistent.

Definition at line 1202 of file realIo.c.

References getRdhIndexSeqPos(), cSet_t::members, and cnode::set.

Referenced by outputRealPatsWCentroid().

```
1203 {
1204
      int indmin, min, i;
1205 int curSeq, curPos;
1206 int numCandidates = candidates[0];
1207
      indmin = candidates[1];
1208
      getRdhIndexSeqPos (data, curCliq->set->members[indmin], &curSeq, &curPos);
1209
      min = curPos;
1210
1211
      // We use less-than-or-equal here because we're starting at 1,
1212
      // so we want 1 to end. The length of candidates is one more than
      // the maxSup, so we know we can reach candidates[maxSup] without
1213
1214
      // a segfault.
1215
      for (i = 2; i <= numCandidates; i++)
1216
       {
1217
           getRdhIndexSeqPos (data, curCliq->set->members[candidates[i]], &curSeq,
1218
                 &curPos);
1219
           if (curPos < min)
1220
1221
           indmin = candidates[i];
1222
           min = curPos;
1223
1224
1225
      return (indmin);
1226 }
```

3.19.2.15 int outputRealPats (rdh_t * data, cll_t * allPats, int L, FILE * OUTPUT FILE, int ** d)

This function is used to print out motifs discovered by Gemoda in an attractive fashion. The function takes five parameters: a pointer to a real data holder object *data*; a pointer to a linked list of motifs *allPats*; an integer which is Gemoda's input parameter *L*; and a pointer to a file handle to which output is printed *OUTPUT_FILE*.

Definition at line 1046 of file realIo.c.

References getRdhIndexSeqPos(), cnode::length, cSet_t::members, cnode::next, rdh_-t::seq, cnode::set, cSet_t::size, and cnode::stat.

```
1048 {
1049
       int i, j, posl;
1050
      int curSeq, curPos;
      cll_t *curCliq = NULL;
1051
1052
      curCliq = allPats;
1053
      i = 0;
1054
       while (curCliq != NULL)
1055
         {
1056
           fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\t", i,
1057
                curCliq->length + L, curCliq->set->size);
1058
           if (d != NULL)
1059
         {
           fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
1060
```

```
1061
         }
           else
1062
1063
1064
           fprintf (OUTPUT_FILE, "\n");
1065
1066
           for (j = 0; j < curCliq->set->size; j++)
1067
1068
           pos1 = curCliq->set->members[j];
1069
           getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1070
           fprintf (OUTPUT_FILE, "
                                      %d\t%d\t", curSeq, curPos);
           fprintf (OUTPUT_FILE, "%lf\t",
1071
1072
                gsl_matrix_get (data->seq[curSeq], curPos, 0));
1073
1074
1075
              for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1076
              mySequences[curSeq].seq[k]); }
1077
           fprintf (OUTPUT_FILE, "\n");
1078
1079
1080
           fprintf (OUTPUT_FILE, "\n\n");
           curCliq = curCliq->next;
1081
1082
1083
         }
1084
       return 0;
1085 }
```

3.19.2.16 int outputRealPatsWCentroid (rdh_t * data, cll_t * allPats, int L, FILE * OUTPUT_FILE, double * extraParams, int compFunc)

This function is used to output real valued patterns in a format such that they are centered on a particular centroid.

Definition at line 1233 of file realIo.c.

References findCliqueCentroid(), getCompFunc, getRdhIndexSeqPos(), cnode::length, makeAlternateCentroid(), cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

```
1236 {
1237
       int i, j, k, posl, centroid;
1238
       int curSeq, curPos;
1239
       int maxSup = 0;
1240
       cll_t *curCliq = NULL;
1241
       double mfToCentroid = 0;
1242
       double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
1243
       int *candidates = NULL;
1244
       curCliq = allPats;
1245
       while (curCliq != NULL)
1246
1247
           if (curCliq->set->size > maxSup)
1248
1249
           maxSup = curCliq->set->size;
1250
```

```
1251
           curCliq = curCliq->next;
1252
1253
       candidates = (int *) malloc ((maxSup + 1) * sizeof (int));
1254
       if (candidates == NULL)
1255
         {
1256
           fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1257
           fflush (stderr);
1258
           exit (0);
1259
1260
       for (i = 0; i \le \max \sup; i++)
1261
        {
1262
           candidates[i] = 0;
        }
1263
      comparisonFunc = getCompFunc (compFunc);
1264
1265
       curCliq = allPats;
       i = 0;
1266
1267
      while (curCliq != NULL)
1268
        {
1269
           fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\n", i,
1270
                curCliq->length + L, curCliq->set->size);
1271
           centroid =
1272
         findCliqueCentroid (data, curCliq, L, compFunc, extraParams,
1273
                     candidates);
1274
           if (centroid < 0)
1275
1276
           centroid = makeAlternateCentroid (data, curCliq, candidates);
1277
1278
           // fprintf(OUTPUT_FILE, "WARNING: No single node in"
1279
           // " cluster has non-zero similarity to all other\n nodes"
           // " in cluster; centroid set to first node.\n");
1280
1281
           // centroid = 0;
1282
         }
1283
           for (j = 0; j < curCliq->set->size; j++)
1284
         {
1285
           pos1 = curCliq->set->members[j];
           getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1286
1287
           fprintf (OUTPUT_FILE, "
                                    %d\t%d\t", curSeq, curPos);
1288
1289
           // fprintf(OUTPUT_FILE, "%lf\t",
1290
           // gsl_matrix_get(data->seq[curSeq],curPos,0));
1291
           mfToCentroid =
1292
             comparisonFunc (data, curCliq->set->members[j],
1293
                     curCliq->set->members[centroid], L, extraParams);
1294
           fprintf (OUTPUT_FILE, "%lf\t", mfToCentroid);
1295
1296
1297
              for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1298
             mySequences[curSeq].seq[k]); }
1299
1300
           fprintf (OUTPUT_FILE, "\n");
1301
1302
           fprintf (OUTPUT_FILE, "\n\n");
1303
           curCliq = curCliq->next;
1304
1305
           for (k = 0; k \le maxSup; k++)
1306
1307
           candidates[k] = 0;
```

3.19.2.17 rdh_t* parseRealData (char ** buf, int nl, char sep, int numSeq, int dim)

This function is used to parse a single line of a fastA formatted input buffer containing real valued data. The function takes

parameters: a pointer to an array of pointers to characters, which stores the sequences that we will read from buf; an integer, which is the line in the buffer on which we should start nl; a single character, which is used to delimit the input data sep; an integer which is the number of the sequence that we are currently reading in numSeq; an integer that is the dimensionality of the input data dim;

Definition at line 933 of file realIo.c.

References countTotalFields(), initRdh(), and pushOnRdhSeq().

Referenced by readRealData().

```
934 {
935
      int i;
936
     int seqNo = -1;
      int totalNumFields;
937
     rdh_t *data = NULL;
938
939
      totalNumFields = countTotalFields (buf, nl, sep);
940
941
942
         data = initRdh(numSeq, totalNumFields + numSeq - 1);
943
944
      data = initRdh (numSeq);
945
      // We're going to add an empty index between
946
947
      // windows that correspond to different
948
      // sequences
949
950
      // Fast forward to the first sequence
951
      i = 0;
952
      while (i < nl)
953
        {
954
955
          // Hit a new sequence
956
          if (buf[i][0] == '>')
957
958
                         // Note that seqNo started at -1!
          seqNo++;
959
          pushOnRdhSeq (data, buf, i, dim, sep);
          i += dim + 1;
960
961
962
          else
```

```
963 {
964 i++;
965 }
966 }
967
968 /*
969 printRdhSeq(data, 0, stdout);
970 */
971 return data;
972 }
```

3.19.2.18 int printRdhSeq ($rdh_t * data$, int seqNo, FILE * FH)

This function is used to print out a real valued data sequence in a pretty manner. The function takes three parameters: a pointer to the real data holder *data*; an integer which is the sequence to be printed out *seqNo*; and a pointer to a file handle which is where the output will be printed *FH*.

Definition at line 710 of file realIo.c.

References getRdhDim(), getRdhLabel(), getRdhSeqLength(), and getRdhValue().

```
711 {
712
      int i, j;
713
     int len;
714
     int dim;
      len = getRdhSeqLength (data, seqNo);
715
716
      dim = getRdhDim (data);
717
      fprintf (FH, "%s\n", getRdhLabel (data, seqNo));
718
      for (i = 0; i < len; i++)
719
720
          for (j = 0; j < dim; j++)
721
722
          fprintf (FH, "%3.1f ", getRdhValue (data, seqNo, i, j));
723
724
          fprintf (FH, "\n");
725
726
      return 0;
727 }
```

3.19.2.19 int pushOnRdhSeq (rdh_t * data, char ** buf, int startLine, int dim, char sep)

This function is used to fill in a real data holder structure as we are reading in the sequences. Notably, this routine uses a few static variables, so it can only be called once and should not be used to alter the real data holder structure later. The function takes five parameters: a pointer to the real data holder *data*; a pointer to an array of pointers to characters, which stores the sequences that we will read from *buf*; an integer, which is the line in the buffer on which we should start *startLine*; an integer

that is the dimensionality of the input data *dim*; a single character, which is used to delimit the input data *sep*;

Definition at line 863 of file realIo.c.

References countFields(), initRdhGslMat(), setRdhColFromString(), and setRdh-Label().

Referenced by parseRealData().

```
864 {
865
      int i, j, k;
866
      int numFields;
867
     // NOTE THAT THESE ARE STATIC VARIABLES!!!!!
869
      // That is, they retain their last value on
870
      // each call to this function!
871
      static int seqNo = 0;
872
873
874
        static int indexNo=0;
875
876
      i = startLine;
877
878
      // Assume that the sequence has at least
879
      // one row (should have called checkRealDataFormat!
880
      numFields = countFields (buf[i + 1], sep);
881
882
      // Initialize the gsl_matrix object for this
883
      // sequence in 'data'
884
      //
885
      // NOTE THAT WE STORE THE TRANSPOSE OF WHAT'S IN
886
      // THE INPUT FILE -- x,y = position x, dimension y
887
      initRdhGslMat (data, seqNo, numFields, dim);
888
889
      // Set the sequence label
890
      setRdhLabel (data, seqNo, buf[i]);
891
892
      // Read in 'dim' rows
      for (j = i + 1, k = 0; j < i + 1 + dim; j++, k++)
893
894
        {
895
896
897
             printf("%d\n", countFields(buf[j], sep));
898
899
900
          // Set the k-th dimension of this sequence
901
          // STILL NOTE THE TRANSPOSE!
902
          setRdhColFromString (data, seqNo, k, buf[j], sep);
903
        }
904
905
906
         for ( l=0 ; l<numFields ; l++ ){ setRdhIndex(data, seqNo, 1, indexNo); indexNo++;</pre>
907
908
909
      seqNo++;
910
```

```
911 // Augment indexNo once more to have a -1 between each sequence!
912 /*
913 indexNo++;
914 */
915 return 0;
916 }
```

3.19.2.20 rdh_t* readRealData (FILE * INPUT)

This function is used to read in a fasta formatted file containing real value data and store the entire thing and a real data holder object. The function takes one parameter: a pointer to a file handle, which is where the data are read from *INPUT*;

Definition at line 983 of file realIo.c.

References checkRealDataFormat(), parseRealData(), and ReadFile().

```
984 {
985
     char **buf = NULL;
986
     int nl;
987
     int i;
988
      char sep = ' ';
989
     int numSeq = 0;
990
     int dimensions = 0;
991
     int status = 1;
992
     rdh_t *data = NULL;
993
994
     // Read the entire INPUT file and put it's
     // contents into 'buf'. This function also
     // alters the contents of the location pointed
996
     // to by &nl. Now nl is the number of lines
997
998
     // in the file (or the size of the buff array.
     buf = ReadFile (INPUT, &nl);
999
1000
      if (buf == NULL)
1001
         {
1002
          return NULL;
1003
      status = checkRealDataFormat (buf, nl, sep, &numSeq, &dimensions);
1004
1005
      if (numSeq <= 0 | dimensions <= 0 | status == 0)
1006
1007
           fprintf (stderr,
1008
                "Data file is poorly formatted or no sequences read!\n");
1009
           fprintf (stderr,
1010
                "Each sequence needs to be the same dimensionality! QUITTING!\n");
1011
           fprintf (stderr, "numSeq = %d, dimensions = %d, status = %d\n", numSeq,
1012
               dimensions, status);
1013
           exit (EXIT_FAILURE);
1014
         }
1015
1016
      // From here on, we assume that the sequence file is well-formatted
1017
      // to make the code more simple.
1018
      data = parseRealData (buf, nl, sep, numSeq, dimensions);
```

```
1019
       // Free up our buffer
1020
1021
       for (i = 0; i < nl; i++)
1022
1023
           if (buf[i] != NULL)
1024
1025
           free (buf[i]);
1026
1027
1028
       if (buf != NULL)
1029
1030
           free (buf);
1031
1032
       return data;
1033 }
```

3.19.2.21 int setRdhColFromString (rdh_t * data, int seqNo, int colNo, char * s, char sep)

This function is used to fill in the values of a sequence in a real data holder object by reading them straight from a string, which is assumed to be a series of floating-point values separated by some particular character. The function takes five parameters: a pointer to the real data holder *data*; an integer, which is the sequence number to be set *seqNo*; an integer representing the dimension of the sequence which is to be set *colNo*; a pointer to the string holding the floating-point values *s*; a character, which separates the floating-point values in the string *sep*;

Definition at line 744 of file realIo.c.

References rdh_t::seq, setRdhValue(), and wordToDouble().

Referenced by pushOnRdhSeq().

```
745 {
746
     int i;
747
     int begin = 0;
748
     int end = 0;
749
     int status = 0;
                            // 0 = in sep, 1 = in word
750
     int fieldCount = 0;
751
     double val;
752
753
      // Make sure the string is not null and
754
     // the rdh_t gsl_matrix array is not null
755
     // and the selected gsl_matrix is not null
756
     if (s == NULL | data->seq == NULL | data->seq[seqNo] == NULL)
757
758
          fprintf (stderr, "Passed bad data to setRdhColFromString -- error!");
759
          fflush (stderr);
760
          exit (0);
761
762
763
      // Loop over the length of the string
      for (i = 0; i < strlen(s); i++)
764
```

```
765
        {
766
767
          // The previous state was space
768
          if (status == 0)
769
770
771
          // We hit a word
772
          if (s[i] != sep)
773
774
              begin = i;
775
              status = 1;
776
            }
777
          else
778
                         // We hit more space
            {
779
              continue;
780
781
782
          else
783
                    \ensuremath{//} The previous state was word
784
          if (s[i] != sep)
785
786
              continue;
787
            }
788
          else
789
            {
                         // We hit a space
              end = i - 1;
790
791
              status = 0;
792
              val = wordToDouble (s, begin, end);
793
794
              // Go to the gsl_matrix object data->seq[seqNo]
795
              // and set the (fieldCount, colNo) = val;
796
              setRdhValue (data, seqNo, fieldCount, colNo, val);
797
              fieldCount++;
798
799
800
801
802
      // At the end, if we were in a word, we have
803
      // one more field
804
      if (status == 1)
805
                         // We're in a word
        {
806
          val = wordToDouble (s, begin, strlen (s));
807
808
          // Added in, MPS 5/3/05 ---
          // And don't forget to set the RdhValue!
809
810
          setRdhValue (data, seqNo, fieldCount, colNo, val);
811
          fieldCount++;
812
813
      return fieldCount;
814 }
```

3.19.2.22 int setRdhIndex (rdh_t * data, int seqNo, int posNo, int index)

This function is used to fill in entries in the indices of the real data holder. The function takes four parameters: a pointer to the real data holder, *data*, an integer specifying the

sequence number *seqNo*, an integer specifying the position number within the sequence *posNo*, and an integer specifying what the index for this sequence number and position number should be *index*.

Definition at line 600 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

```
601 {
602
      if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
603
          | | index > data->indexSize)
604
605
          fprintf (stderr, "Passed bad data to getRdhValue -- error!");
606
          fflush (stderr);
607
          exit (0);
608
609
610
611
         printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
612
613
614
         fflush(stdout);
615
616
      data->indexToSeq[index] = seqNo;
617
      data->indexToPos[index] = posNo;
618
      return 0;
619 }
```

3.19.2.23 int setRdhLabel ($rdh_t * data$, int seqNo, char *s)

This function will label a sequence within a real data holder object with a particular string. The function takes two parameters: a pointer to the real data holder, data, an integer seqNo, and a pointer to a string s.

Definition at line 543 of file realIo.c.

References rdh_t::label, and rdh_t::seq.

Referenced by pushOnRdhSeq().

```
544 {
545
      if (data->seq == NULL | data->label == NULL)
546
547
          fprintf (stderr, "Passed bad data to setRdhLabel -- error!");
548
          fflush (stderr);
549
          exit (0);
550
551
      data->label[seqNo] = strdup (s);
      if (data->label[seqNo] == NULL)
552
553
        {
          fprintf (stderr, "\nMemory Error allocating label!\n%s\n",
554
555
               strerror (errno));
556
          fflush (stderr);
```

```
557 exit (0);
558 }
559 return 0;
560 }
```

3.19.2.24 int setRdhValue (rdh_t * data, int seqNo, int posNo, int dimNo, double val)

This function will set a particular dimension at a particular position within a specified sequence to a user supplied value. The function takes five parameters: a pointer to the real data holder, *data*, an integer *seqNo* which is the sequence which needs its value set, two integers that specify the position number and the dimension number that needs to be set, and finally a double precision floating point number which is the value to which the data should be set.

Definition at line 575 of file realIo.c.

References getRdhDim(), getRdhSeqLength(), and rdh_t::seq.

Referenced by setRdhColFromString().

```
576 {
577
      if (data == NULL | data->seq == NULL | data->seq[seqNo] == NULL
          || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
578
579
580
          fprintf (stderr, "Passed bad data to setRdhValue -- error!");
581
          fflush (stderr);
          exit (0);
582
583
584
      gsl_matrix_set (data->seq[seqNo], posNo, dimNo, val);
585
     return 0;
586 }
```

3.19.2.25 wordToDouble (char * s, int begin, int end)

Turn the substring of s starting at char s[begin] and ending at s[end] int a double. INPUT: a string s, integer begin, and integer end. OUTPUT: a double. NOTE: Throws an error and dies if there's a problem making the double from the substring. No room for ill-formated data files. double

Definition at line 30 of file realIo.c.

Referenced by countFields(), and setRdhColFromString().

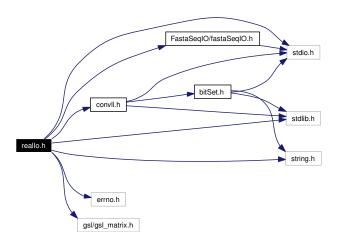
```
31 {
32   char *str = NULL;
33   char *endptr;
34   double val;
35   int size;
```

```
36
    int memsize;
37
    // Check for a sane substring
39
    if (end - begin <= 0)
40
41
        fprintf (stderr, "\nInvalid argument to wordToDouble!\n");
        fflush (stderr);
42
43
        exit (0);
44
    // Get the required string size
46
47
    48
    size = end - begin + 1;
49
50
    // Get memory for a temporary string
51
    str = (char *) malloc (memsize * sizeof (char));
    if (str == NULL)
52
53
54
        fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
55
        fflush (stderr);
56
        exit (0);
57
58
59
    // Make sure the string ends with a null char
60
    str[size] = ' \setminus 0';
61
62
    // Copy the word into str
63
    str = strncpy (str, s + begin, size);
64
65
    // Set endptr to str as initial value
    endptr = str;
66
67
    val = strtod (str, &endptr);
68
69
    // endptr should point to the last char
    // used in the conversion if strtod worked
70
71
    if (val == 0 && endptr == str)
72
     {
73
        fprintf (stderr, "\nError making double from string: %s\n", str);
74
        fflush (stderr);
75
        exit (0);
76
     }
77
    free (str);
78
    return val;
79 }
```

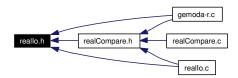
3.20 realIo.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "FastaSeqIO/fastaSeqIO.h"
#include "convll.h"
```

Include dependency graph for realIo.h:



This graph shows which files directly or indirectly include this file:



Data Structures

• struct rdh_t

Functions

- rdh_t * readRealData (FILE *INPUT)
- rdh_t * freeRdh (rdh_t *data)
- int initRdhIndex (rdh_t *data, int wordSize, int seqGap)
- int getRdhIndexSeqPos (rdh_t *data, int index, int *seq, int *pos)
- int getRdhDim (rdh_t *data)
- int outputRealPats (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, int **d)
- int outputRealPatsWCentroid (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, double *extraParams, int compFunc)

3.20.1 Function Documentation

3.20.1.1 $rdh_t* freeRdh (rdh_t* data)$

This function returns a null pointer after freeing the memory associated with a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 396 of file realIo.c.

References rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::offsetToIndex, rdh_t::seq, and rdh_t::size.

Referenced by main().

3.20.1.2 int getRdhDim (rdh_t * data)

This function returns an integer equal to the dimensions of the data stored in a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 447 of file realIo.c.

References rdh_t::seq.

Referenced by generalMatchFactor(), getRdhValue(), massSpecCompareWElut(), printRdhSeq(), rmsdCompare(), and setRdhValue().

3.20.1.3 int getRdhIndexSeqPos ($rdh_t * data$, int index, int * seq, int * pos)

This function is used to access and change the sequence and position values, given an index. The function takes four parameters: a pointer to the real data holder, *data*, an integer *index*, a pointer integer *seq*, and a pointer integer *pos*.

Definition at line 544 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

Referenced by generalMatchFactor(), makeAlternateCentroid(), massSpecCompare-WElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsd-Compare().

3.20.1.4 int initRdhIndex (rdh_t * data, int wordSize, int seqGap)

This function is used to initialize the two indices inside a real data holder. The function takes as its input three parameters a pointer to the real data holder, *data*, the size of the words to be compared during the comparison stage *wordSize*, and an integer *seq-Gap*, which is used to place empty data between unique sequences, such that we do not convolve from one sequence into another during the convolution stage.

Definition at line 307 of file realIo.c.

References getRdhSeqLength(), rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexTo-Seq, rdh t::offsetToIndex, and rdh t::size.

Referenced by realComparison().

```
3.20.1.5 int outputRealPats (rdh_t * data, cll_t * allPats, int L, FILE * OUTPUT_FILE, int ** d)
```

This function is used to print out motifs discovered by Gemoda in an attractive fashion. The function takes five parameters: a pointer to a real data holder object *data*; a pointer to a linked list of motifs *allPats*; an integer which is Gemoda's input parameter *L*; and a pointer to a file handle to which output is printed *OUTPUT FILE*.

Definition at line 904 of file realIo.c.

References getRdhIndexSeqPos(), cnode::length, cSet_t::members, cnode::next, rdh_-t::seq, cnode::set, cSet_t::size, and cnode::stat.

Referenced by main().

3.20.1.6 int outputRealPatsWCentroid (rdh_t * data, cll_t * allPats, int L, FILE * OUTPUT_FILE, double * extraParams, int compFunc)

This function is used to output real valued patterns in a format such that they are centered on a particular centroid.

Definition at line 1068 of file realIo.c.

References findCliqueCentroid(), getCompFunc, getRdhIndexSeqPos(), cnode::length, makeAlternateCentroid(), cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

3.20.1.7 rdh_t* readRealData (FILE * INPUT)

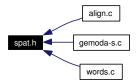
This function is used to read in a fasta formatted file containing real value data and store the entire thing and a real data holder object. The function takes one parameter: a pointer to a file handle, which is where the data are read from *INPUT*;

Definition at line 850 of file realIo.c.

 $References\ check Real Data Format (),\ parse Real Data (),\ and\ Read File ().$

3.21 spat.h File Reference

This graph shows which files directly or indirectly include this file:



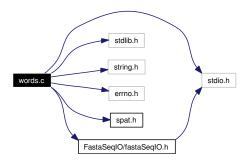
Data Structures

- struct sOffset_t
- struct sPat_t

3.22 words.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "spat.h"
#include "FastaSeqIO/fastaSeqIO.h"
```

Include dependency graph for words.c:



Data Structures

- struct sHashEntry_t
- struct sHash_t

Defines

• #define SHASH_MAX_KEY_SIZE 1000

Functions

- int sieve3 (long n)
- unsigned long hash1 (unsigned char *str)
- int hashpjw (char *s)
- sHash_t initSHash (int n)
- sHashEntry_t * searchSHash (sHashEntry_t *newEntry, sHash_t *thisHash, int create)
- int destroySHash (sHash_t *thisHash)

- int printSHash (sHash_t *thisHash, FILE *FH)
- int printSPats (sPat_t *a, int n)
- int destroySPatA (sPat_t *words, int wc)
- sPat_t * countWords2 (fSeq_t *seq, int numSeq, int L, int *numWords)

3.22.1 Detailed Description

This file defines functions that are used in the processing of string based sequences. There are a number of functions defined in this file better used for hashing strings so that the comparison phase can be sped up by only comparing unique words. Heuristically, we have noticed that for sequences in which there is a large degree of redundancy these hashing functions can significantly speed up the comparison phase.

Definition in file words.c.

3.22.2 Define Documentation

3.22.2.1 #define SHASH_MAX_KEY_SIZE 1000

Definition at line 192 of file words.c.

Referenced by printSHash(), and searchSHash().

3.22.3 Function Documentation

```
3.22.3.1 sPat_t* countWords2 (fSeq_t* seq, int numSeq, int L, int * numWords)
```

Counts words of size L in the input FastA sequences, hashes all of the words, and returns an array of sPat_t objects.

Definition at line 373 of file words.c.

References sHashEntry_t::data, destroySHash(), sHashEntry_t::idx, initSHash(), s-HashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, s-Offset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

```
374 {
375   int i, j;
376   int totalChars = 0;
377   int hashSize;
378   sHashEntry_t newEntry;
379   sHashEntry_t *ep;
380   sHash_t wordHash;
381   sPat_t *words = NULL;
```

```
382
      int wc = 0;
     int prev = -1;
383
384
     int l;
385
386
387
      // Count the total number of characters. This
388
      // is the upper limit on how many words we can have
389
      for (i = 0; i < numSeq; i++)
390
391
          totalChars += strlen (seq[i].seq);
392
393
394
     // Get a prime number for the size of the hash table
     hashSize = sieve3 ((long) (2 * totalChars));
395
396
      wordHash = initSHash (hashSize);
397
398
      // Chop up each sequence and hash out the words of size L
399
      for (i = 0; i < numSeq; i++)
400
       {
          prev = -1;
401
402
403
          // skip sequences that are too short to have
404
          // a pattern
405
          if (strlen (seq[i].seq) < L)</pre>
406
        {
407
          continue;
408
409
          for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
410
411
412
          // Make a hash table entry for this word
413
          newEntry.key = &(seq[i].seq[j]);
          newEntry.data = 1;
414
415
          newEntry.idx = wc;
416
          newEntry.L = L;
417
418
          // Check to see if it's already in the hash table
419
          ep = searchSHash (&newEntry, &wordHash, 0);
420
          if (ep == NULL)
421
422
423
              // If it's not, create an entry for it
              ep = searchSHash (&newEntry, &wordHash, 1);
424
425
426
              // Increase the size of our word array
              words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
427
428
              if (words == NULL)
429
430
              fprintf (stderr, "Error!\n");
431
              fflush (stderr);
432
433
              // Add the new word
434
              words[wc].string = &(seq[i].seq[j]);
435
              words[wc].length = L;
              words[wc].support = 1;
436
437
              words[wc].offset =
            (sOffset_t *) malloc (1 * sizeof (sOffset_t));
438
```

```
439
              if (words[wc].offset == NULL)
440
441
              fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
442
              fflush (stderr);
443
              exit (0);
444
445
              words[wc].offset[0].seq = i;
446
              words[wc].offset[0].pos = j;
447
              words[wc].offset[0].prev = prev;
448
              words[wc].offset[0].next = -1;
449
450
              if (prev != -1)
451
              words[prev].offset[words[prev].support - 1].next = wc;
452
453
454
              prev = wc;
455
              WC++;
456
457
458
          else
459
            {
460
              // If it is, increase the count for this word
461
462
              ep->data++;
463
464
              // add a new offset to the word array
465
              1 = words[ep->idx].support;
466
              words[ep->idx].offset =
467
            (sOffset_t *) realloc (words[ep->idx].offset,
468
                            (1 + 1) * sizeof (sOffset_t));
469
              words[ep->idx].offset[l].seq = i;
470
              words[ep->idx].offset[l].pos = j;
              words[ep->idx].offset[l].prev = prev;
471
              words[ep->idx].offset[l].next = -1;
472
473
474
              // Update the next/prev
475
              if (prev != -1)
476
477
              words[prev].offset[words[prev].support - 1].next = ep->idx;
478
479
              prev = ep->idx;
480
481
              // Have to put this down here for cases when we create
482
              // a word and it is immeadiately followed by itself!!
483
              words[ep->idx].support += 1;
484
485
486
487
488
489
      destroySHash (&wordHash);
     *numWords = wc;
490
491
      return words;
492 }
```

3.22.3.2 int destroySHash (sHash_t * thisHash)

Destroy a hash table, freeing the memory.

Definition at line 272 of file words.c.

References sHash_t::hash, sHash_t::hashSize, and sHash_t::iHashSize.

Referenced by countWords2().

```
273 {
274
      int i;
275
     free (thisHash->iHashSize);
276
     free (thisHash->hashSize);
277
      for (i = 0; i < thisHash->totalSize; i++)
278
279
          if (thisHash->hash[i] != NULL)
280
        {
281
          free (thisHash->hash[i]);
282
          thisHash->hash[i] = NULL;
283
284
285
      if (thisHash->hash != NULL)
286
287
          free (thisHash->hash);
288
          thisHash->hash = NULL;
289
290
     return 0;
291 }
```

3.22.3.3 int destroySPatA (sPat_t * words, int wc)

This function is used to free up the memory allocated in an array of sPat_t space objects. The function returns a null pointer.

Definition at line 352 of file words.c.

References sPat_t::offset.

```
353 {
354
      int i;
355
      for (i = 0; i < wc; i++)
356
        {
357
          if (words[i].offset != NULL)
358
          free (words[i].offset);
359
360
          words[i].offset = NULL;
361
362
363
     free (words);
364
      words = NULL;
365
      return 0;
366 }
```

3.22.3.4 unsigned long hash1 (unsigned char * str)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 73 of file words.c.

Referenced by searchSHash().

3.22.3.5 int hashpjw (char *s)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 89 of file words.c.

```
90 {
91
     char *p;
    unsigned int h, g;
92
93
    h = 0;
94
    for (p = s; *p != '\0'; p++)
95
96
97
        h = (h << 4) + *p;
98
         if ((g = h \& 0xF0000000))
99
100
         h ^= g >> 24;
         h ^= g;
101
102
103
104
    return h;
105 }
```

3.22.3.6 **sHash_t** initSHash (int n)

Allocates the memory for a sHash table and initializes some of the elements.

Definition at line 155 of file words.c.

References sHash_t::totalSize.

Referenced by countWords2().

```
156 {
157
      int i = 0;
158
     int step = 0;
159
      sHash_t this;
160
161
      this.totalSize = n;
      this.hashSize = (int *) malloc (n * sizeof (int));
162
163
      if (this.hashSize == NULL)
164
165
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
166
          fflush (stderr);
167
          exit (0);
168
169
      this.iHashSize = (int *) malloc (n * sizeof (int));
170
      if (this.iHashSize == NULL)
171
172
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
173
          fflush (stderr);
174
          exit (0);
175
      this.hash = (sHashEntry_t **) malloc (n * sizeof (sHashEntry_t *));
176
177
      if (this.hash == NULL)
178
179
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
180
          fflush (stderr);
          exit (0);
181
182
      for (i = 0; i < n; i++)
183
184
185
          this.hash[i] = NULL;
186
          this.hashSize[i] = 0;
187
          this.iHashSize[i] = step;
188
189
      return this;
190 }
```

3.22.3.7 int printSHash ($sHash_t * thisHash$, FILE * FH)

This function is used to print the hash out and is generally only used for error checking. Definition at line 298 of file words.c.

References sHashEntry_t::data, sHash_t::hash, sHashEntry_t::key, sHashEntry_t::L, and SHASH MAX KEY SIZE.

```
299 {
300    int i, j;
301    char string[SHASH_MAX_KEY_SIZE];
302
303    for (i = 0; i < thisHash->totalSize; i++)
304    {
305         for (j = 0; j < thisHash->hashSize[i]; j++)
306    {
307
```

3.22.3.8 int printSPats ($sPat_t * a$, int n)

This function is used to print out an array of sPat_t objects and is generally only used for error checking.

Definition at line 321 of file words.c.

References sPat_t::length.

```
322 {
      char *s = NULL;
323
324
      int i, j;
325
      int size = 0;
326
     for (i = 0; i < n; i++)
327
328
          if (a[i].length > size)
329
330
          s = (char *) realloc (s, a[i].length * sizeof (char));
331
332
          strncpy (s, a[i].string, a[i].length);
          s[a[i].length] = ' \0';
333
334
         printf ("%d: %s\n", i, s);
335
          for (j = 0; j < a[i].support; j++)
336
337
          printf ("\t%d %d -> (%d, %d)\n", a[i].offset[j].seq,
338
              a[i].offset[j].pos, a[i].offset[j].prev,
339
              a[i].offset[j].next);
340
         printf ("\n");
341
342
343
      free (s);
344
      return 0;
345 }
```

3.22.3.9 **sHashEntry_t*** searchSHash (**sHashEntry_t** * *newEntry*, **sHash_t** * *thisHash*, int *create*)

This function has two purposes. It searches for entries in the hash table and it puts new entries in.

Definition at line 198 of file words.c.

References sHash_t::hash, hash1(), sHash_t::hashSize, sHash_t::iHashSize, sHash_Entry_t::key, sHashEntry_t::L, SHASH_MAX_KEY_SIZE, and sHash_t::totalSize.

Referenced by countWords2().

```
199 {
200
      char string[SHASH_MAX_KEY_SIZE];
201
      unsigned long (*hashFunction) () = &hash1;
202
      int i, thisIndex;
203
     int status = 0;
204
205
      // A string to store the key
206
      strncpy (string, newEntry->key, newEntry->L);
207
      string[newEntry->L] = ' \setminus 0';
208
209
      // The index that this key hashes to
210
      thisIndex = hashFunction ((unsigned char *) string) % thisHash->totalSize;
211
212
      // For each member that has this index, check to see
213
      // if the key is the same
214
      for (i = 0; i < thisHash->hashSize[thisIndex]; i++)
215
216
          if (strncmp (thisHash->hash[thisIndex][i].key, string, newEntry->L) ==
217
          0)
218
        {
219
220
          // We found a match
221
222
             printf("\t%s already in hash table!\n");
223
          status = 1;
224
          return &(thisHash->hash[thisIndex][i]);
225
226
          break;
227
228
229
230
      // If we didn't find the key and we're told to create it,
231
232
      // then allocate new memory for the hashEntry and put it in
233
      if (status == 0 && create != 0)
234
235
          // Allocate space for the new entry at this index
236
237
          if (thisHash->iHashSize[thisIndex] == 0)
238
239
          thisHash->hash[thisIndex] =
240
            (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
241
        }
          else
242
243
          thisHash->hash[thisIndex] =
244
245
            (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
246
                           (thisHash->iHashSize[thisIndex] +
247
                            1) * sizeof (sHashEntry_t));
248
249
          if (thisHash->hash[thisIndex] == NULL)
250
```

```
251
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
252
          fflush (stderr);
253
          exit (0);
254
       }
255
          // Increase our record of the size
256
          i = thisHash->hashSize[thisIndex];
257
         thisHash->hash[thisIndex][i] = *newEntry;
258
          thisHash->iHashSize[thisIndex]++;
259
         thisHash->hashSize[thisIndex]++;
260
261
262
          // Return a pointer to this entry
263
         return &(thisHash->hash[thisIndex][i]);
264
265
     return NULL;
266 }
```

3.22.3.10 int sieve3 (long n)

Prime number generator: returns first prime number equal or less than

Parameters:

n.

Definition at line 27 of file words.c.

Referenced by countWords2().

```
28 {
29
     int i, p, j;
30
     int *a;
31
     a = (int *) malloc ((n + 1) * sizeof (int));
32
     if (a == NULL)
33
34
         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
35
         fflush (stderr);
36
         exit (0);
37
     a[0] = 0;
38
39
     a[1] = 0;
40
     for (i = 2; i < n; i++)
41
         a[i] = 1;
42
       }
43
44
     p = 2i
45
     do
46
         j = 2 * p;
47
48
         do
49
         a[j] = 0;
50
51
         j = j + p;
52
```

```
53
          while (j \le n);
54
         p = p + 1;
    while (p * p < 2 * n);
for (i = n; i > 2; i--)
56
57
58
59
          if (a[i])
60
61
          free (a);
62
          return i;
63
64
65 free (a);
66 return 0;
67 }
```

Index

aaOrder	bitGraphCheckBit
align.c, 29	bitSet.c, 32
matrices.h, 159	bitSet.h, 55
addToStacks	bitGraphRowIntersection
convll.c, 66	bitSet.c, 32
convll.h, 100	bitSet.h, 55
align.c, 25	bitGraphRowUnion
aaOrder, 29	bitSet.c, 33
ALIGN_ALPHABET, 27	bitSet.h, 55
alignMat, 27	bitGraphSetFalse
alignWordsMat_bit, 28	bitSet.c, 33
ALIGN_ALPHABET	bitSet.h, 56
align.c, 27	bitGraphSetFalseDiagonal
alignMat	bitSet.c, 34
align.c, 27	bitSet.h, 56
alignWordsMat_bit	bitGraphSetFalseSym
align.c, 28	bitSet.c, 34
gemoda-s.c, 135	bitSet.h, 56
alpha_mat	bitGraphSetTrue
matrices.h, 159	bitSet.c, 34
•	bitSet.h, 56
beta_mat	bitGraphSetTrueDiagonal
matrices.h, 159	bitSet.c, 35
BIG_BUFFER	bitSet.h, 57
fastaSeqIO.c, 111	bitGraphSetTrueSym
bit_t	bitSet.c, 35
bitSet.h, 55	bitSet.h, 57
bitcount32	bitSet.c, 30
bitSet.c, 31	bitSet.c
bitcount32_precomp	bitcount32, 31
bitSet.c, 31	bitcount32_precomp, 31
bitcount64	bitcount64, 32
bitSet.c, 32	bitGraphCheckBit, 32
bitGraph_t, 3	bitGraphRowIntersection, 32
bitGraph_t	bitGraphRowUnion, 33
graph, 4	bitGraphSetFalse, 33
size, 4	bitGraphSetFalseDiagonal, 34

11.6 16 71.6	11.0 5100
bitGraphSetFalseSym, 34	bitSetDifference, 57
bitGraphSetTrue, 34	bitSetIntersection, 58
bitGraphSetTrueDiagonal, 35	bitSetSum, 58
bitGraphSetTrueSym, 35	bitSetUnion, 58
bitSet3WayIntersection, 35	BITSLOT, 53
bitSetDifference, 36	BSBITSIZE, 53
bitSetIntersection, 37	BSCLEAR, 54
bitSetSum, 37	BSINTERSECTION, 54
bitSetUnion, 38	BSMASK, 54
checkBit, 38	BSNUMSLOTS, 54
copyBitGraph, 39	BSSET, 54
copySet, 39	BSTEST, 54
countBitGraphNonZero, 39	BSUNION, 55
countSet, 40	checkBit, 58
deleteBitGraph, 41	copyBitGraph, 59
deleteBitSet, 41	copySet, 59
emptyBitGraph, 42	countBitGraphNonZero, 59
emptyBitGraphRow, 42	countSet, 59
emptySet, 42	deleteBitGraph, 60
fillBitGraph, 43	deleteBitSet, 60
fillSet, 43	emptyBitGraph, 60
flipBits, 43	emptyBitGraphRow, 60
maskBitGraph, 44	emptySet, 60
newBitArray, 44	fillBitGraph, 61
newBitGraph, 45	fillSet, 61
newBitSet, 45	flipBits, 61
nextBitBitSet, 46	<u> </u>
	maskBitGraph, 61
printBinaryBitSet, 47	newBitArray, 62
printBitGraph, 48	newBitGraph, 62
printBitSet, 48	newBitSet, 62
setFalse, 49	nextBitBitSet, 62
setTrue, 49	printBinaryBitSet, 63
bitSet.h, 51	printBitGraph, 63
bitSet.h	printBitSet, 63
bit_t, 55	setFalse, 63
bitGraphCheckBit, 55	setTrue, 63
bitGraphRowIntersection, 55	bitSet3WayDifference
bitGraphRowUnion, 55	bitSet.h, 57
bitGraphSetFalse, 56	bitSet3WayIntersection
bitGraphSetFalseDiagonal, 56	bitSet.c, 35
bitGraphSetFalseSym, 56	bitSet.h, 57
bitGraphSetTrue, 56	bitSet_t, 5
bitGraphSetTrueDiagonal, 57	bitSet_t
bitGraphSetTrueSym, 57	bytes, 5
bitSet3WayDifference, 57	max, 5
bitSet3WayIntersection, 57	slots, 5
•	,

	11 00
tf, 6	blosum90
bitSetDifference	matrices.h, 160
bitSet.c, 36	blosumn
bitSet.h, 57	matrices.h, 160
bitSetIntersection	BSBITSIZE
bitSet.c, 37	bitSet.h, 53
bitSet.h, 58	BSCLEAR
bitSetSum	bitSet.h, 54
bitSet.c, 37	BSINTERSECTION
bitSet.h, 58	bitSet.h, 54
bitSetToCSet	BSMASK
convll.c, 67	bitSet.h, 54
convll.h, 100	BSNUMSLOTS
bitSetUnion	bitSet.h, 54
bitSet.c, 38	BSSET
bitSet.h, 58	bitSet.h, 54
BITSLOT	BSTEST
bitSet.h, 53	bitSet.h, 54
blosum100	BSUNION
matrices.h, 159	bitSet.h, 55
blosum30	BUFFER
matrices.h, 159	fastaSeqIO.c, 111
blosum35	bytes
matrices.h, 159	bitSet_t, 5
blosum40	bitset_t, 5
matrices.h, 159	calcStatAllCliqs
blosum45	patStats.c, 183
	patStats.h, 199
matrices.h, 159 blosum50	calcStatCliq
	-
matrices.h, 159	patStats.c, 183
blosum55	checkBit
matrices.h, 160	bitSet.c, 38
blosum60	bitSet.h, 58
matrices.h, 160	checkCliquecSet
blosum62	convll.c, 68
matrices.h, 160	checkRealDataFormat
blosum65	realIo.c, 213
matrices.h, 160	cliquecounter
blosum70	convll.c, 97
matrices.h, 160	cliqueMembership
blosum75	mnode, 12
matrices.h, 160	cll_t
blosum80	convll.h, 100
matrices.h, 160	cnode, 7
blosum85	id, 7
matrices.h, 160	length, 7

	1
next, 8	completeConv, 100
set, 8	emptyMemberStacks, 101
stat, 8	fillMemberStacks, 101
coil_mat	initheadCll, 101
matrices.h, 161	mergeIntersect, 102
completeConv	mll_t, 100
convll.c, 69	mllToCSet, 102
convll.h, 100	popAllCll, 102
convll.c, 65	popCll, 102
addToStacks, 66	popMemStack, 103
bitSetToCSet, 67	popWholeMemStack, 103
checkCliquecSet, 68	printCll, 103
cliquecounter, 97	printCllPattern, 103
completeConv, 69	printMemberStacks, 104
emptyMemberStacks, 71	pushClique, 104
fillMemberStacks, 71	pushCll, 104
initheadCll, 72	pushConvClique, 104
mergeIntersect, 72	pushcSet, 105
mllToCSet, 73	pushMemStack, 105
popAllCll, 74	removeSupers, 105
popCll, 75	searchMemsWithList, 106
popMemStack, 76	setStackTrue, 106
popWholeMemStack, 76	singleCliqueConv, 106
printCll, 77	swapNodecSet, 107
printCllPattern, 77	uniqClique, 107
printCSet, 78	wholeCliqueConv, 108
printMemberStacks, 79	wholeRoundConv, 108
pruneCll, 79	yankCll, 108
pushClique, 81	convolve
pushCll, 82	gemoda-r.c, 123
pushConvClique, 83	gemoda-s.c, 136
pushcSet, 84	newConv.c, 169
pushMemStack, 84	copyBitGraph
removeSupers, 85	bitSet.c, 39
searchMemsWithList, 87	bitSet.h, 59
setStackTrue, 88	copySet
singleCliqueConv, 89	bitSet.c, 39
swapNodecSet, 91	bitSet.h, 59
uniqClique, 91	countBitGraphNonZero
wholeCliqueConv, 93	bitSet.c, 39
wholeRoundConv, 95	bitSet.h, 59
yankCll, 96	countExtraParams
convll.h, 98	gemoda-r.c, 125
addToStacks, 100	countFields
bitSetToCSet, 100	realIo.c, 215
cll_t, 100	CountFSeqs
	<u> </u>

fastaSeqIO.c, 111	emptyMemberStacks
fastaSeqIO.h, 120	convll.c, 71
countLines	convll.h, 101
fastaSeqIO.c, 111	emptySet
fastaSeqIO.h, 120	bitSet.c, 42
countSet	bitSet.h, 60
bitSet.c, 40	
bitSet.h, 59	fastaSeqIO.c
countTotalFields	BIG_BUFFER, 111
realIo.c, 216	BUFFER, 111
countWords2	CountFSeqs, 111
gemoda-s.c, 138	countLines, 111
words.c, 245	FreeFSeqs, 112
cSet_t, 9	initAofFSeqs, 112
cSet_t	measureLine, 113
members, 9	printFSeqSubSeq, 113
size, 9	ReadFile, 113
cumDMatrix	ReadFSeqs, 115
patStats.c, 184	ReadTxtSeqs, 117
patStats.h, 199	WriteFSeqA, 117
	fastaSeqIO.h
data	CountFSeqs, 120
sHashEntry_t, 17	countLines, 120
dayhoff	FreeFSeqs, 120
matrices.h, 161	initAofFSeqs, 120
DEFAULT_MATRIX	measureLine, 120
matrices.c, 155	printFSeqSubSeq, 120
deleteBitGraph	ReadFSeqs, 120
bitSet.c, 41	ReadTxtSeqs, 121
bitSet.h, 60	WriteFSeqA, 121
deleteBitSet	FastaSeqIO/ Directory Reference, 1
bitSet.c, 41	FastaSeqIO/fastaSeqIO.c, 110
bitSet.h, 60	FastaSeqIO/fastaSeqIO.h, 119
destroySHash	fillBitGraph
words.c, 247	bitSet.c, 43
destroySPatA	bitSet.h, 61
words.c, 248	fillMemberStacks
dna_idmat	convll.c, 71
matrices.h, 161	convll.h, 101
	fillSet
emptyBitGraph	bitSet.c, 43
bitSet.c, 42	bitSet.h, 61
bitSet.h, 60	filterGraph
emptyBitGraphRow	newConv.c, 170
bitSet.c, 42	filterIter
bitSet.h, 60	newConv.c, 171

0.10110	405
findCliqueCentroid	patStats.c, 187
realIo.c, 217	getMatrixByName
findCliques	gemoda-s.c, 140
newConv.c, 173	matrices.c, 156
flipBits	getRdhDim
bitSet.c, 43	realIo.c, 220
bitSet.h, 61	realIo.h, 240
freeD	getRdhIndexSeqPos
patStats.c, 186	realIo.c, 220
patStats.h, 199	realIo.h, 240
FreeFSeqs	getRdhLabel
fastaSeqIO.c, 112	realIo.c, 221
fastaSeqIO.h, 120	getRdhSeqLength
freeRdh	realIo.c, 221
realIo.c, 219	getRdhValue
realIo.h, 240	realIo.c, 222
fSeq_t, 11	getStatMat
fSeq_t	patStats.c, 188
label, 11	patStats.h, 199
seq, 11	graph
d 122	bitGraph_t, 4
gemoda-r.c, 122	
convolve, 123	hash
countExtraParams, 125	sHash_t, 15
main, 125	hash1
parseExtraParams, 129	words.c, 248
pruneBitGraph, 130	hashpjw
usage, 132	words.c, 249
gemoda-s.c, 134	hashSize
alignWordsMat_bit, 135	sHash_t, 15
convolve, 136	
countWords2, 138	id
getMatrixByName, 140	cnode, 7
main, 140	identity_aa
matrixlist, 149	matrices.h, 161
pruneBitGraph, 150	idmat
usage, 152	matrices.h, 161
generalMatchFactor	idx
realCompare.c, 202	sHashEntry_t, 17
realCompare.h, 210	iHashSize
getCompFunc	sHash_t, 15
realCompare.c, 203	increaseMem
realCompare.h, 211	patStats.c, 191
getLargestLength	indexSize
patStats.c, 187	rdh_t, 13
getLargestSupport	indexToPos
Schargeswapport	maca for os

rdh_t, 13	matrices.c, 155
indexToSeq	DEFAULT_MATRIX, 155
rdh_t, 13	getMatrixByName, 156
initAofFSeqs	matrices.h, 157
fastaSeqIO.c, 112	aaOrder, 159
fastaSeqIO.h, 120	alpha_mat, 159
initheadCll	beta_mat, 159
convll.c, 72	blosum100, 159
convll.h, 101	blosum30, 159
initRdh	blosum35, 159
realIo.c, 222	blosum40, 159
initRdhGslMat	blosum45, 159
realIo.c, 223	blosum50, 159
initRdhIndex	blosum55, 160
realIo.c, 224	blosum60, 160
realIo.h, 241	blosum62, 160
initSHash	blosum65, 160
words.c, 249	blosum70, 160
	blosum75, 160
key	blosum80, 160
sHashEntry_t, 17	blosum85, 160
	blosum90, 160
L	blosumn, 160
sHashEntry_t, 17	coil_mat, 161
label	dayhoff, 161
fSeq_t, 11	dna_idmat, 161
rdh_t, 14	identity_aa, 161
length	idmat, 161
cnode, 7	pam100, 161
sPat_t, 21	pam110, 161
	pam120, 161
main	pam130, 161
gemoda-r.c, 125	pam140, 161
gemoda-s.c, 140	pam150, 162
makeAlternateCentroid	pam160, 162
realIo.c, 226	pam190, 162
maskBitGraph	pam200, 162
bitSet.c, 44	pam210, 162
bitSet.h, 61	pam220, 162
massSpecCompareWElut	pam230, 162
realCompare.c, 203	pam240, 162
realCompare.h, 210	pam250, 162
mat	pam260, 162
matrixmap.h, 166	pam280, 163
matdata.h, 154	pam290, 163
MATRIX_SIZE, 154	pam300, 163
	•

pam310, 163	cliqueMembership, 12
pam320, 163	next, 12
pam330, 163	
pam340, 163	name
pam360, 163	matrixmap.h, 166
pam370, 163	newBitArray
pam380, 163	bitSet.c, 44
pam390, 164	bitSet.h, 62
pam400, 164	newBitGraph
pam430, 164	bitSet.c, 45
pam440, 164	bitSet.h, 62
pam450, 164	newBitSet
pam460, 164	bitSet.c, 45
pam490, 164	bitSet.h, 62
pam500, 164	newConv.c, 168
phat_t75_b73, 164	newConv.c
phat_t80_b78, 164	convolve, 169
phat_t85_b82, 165	filterGraph, 170
matrix_map	filterIter, 171
matrixmap.h, 166	findCliques, 173
MATRIX SIZE	pruneBitGraph, 176
matdata.h, 154	pruneCll, 178
matrixlist	singleLinkage, 180
gemoda-s.c, 149	next
gemoda-s.c, 149 matrixmap.h, 166	
matrixmap.h, 166	cnode, 8
matrixmap.h, 166 mat, 166	cnode, 8 mnode, 12
matrixmap.h, 166 mat, 166 matrix_map, 166	cnode, 8 mnode, 12 sOffset_t, 19
matrixmap.h, 166 mat, 166	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102 mll_t	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241 outputRealPatsWCentroid
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102 mll_t convll.h, 100	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241 outputRealPatsWCentroid realIo.c, 228
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102 mll_t convll.h, 100 mllToCSet	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241 outputRealPatsWCentroid
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102 mll_t convll.h, 100 mllToCSet convll.c, 73	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241 outputRealPatsWCentroid realIo.c, 228 realIo.h, 241
matrixmap.h, 166 mat, 166 matrix_map, 166 name, 166 max bitSet_t, 5 measureDiagonal patStats.c, 193 measureLine fastaSeqIO.c, 113 fastaSeqIO.h, 120 members cSet_t, 9 mergeIntersect convll.c, 72 convll.h, 102 mll_t convll.h, 100 mllToCSet	cnode, 8 mnode, 12 sOffset_t, 19 nextBitBitSet bitSet.c, 46 bitSet.h, 62 offset sPat_t, 21 offsetToIndex rdh_t, 14 oldGetStatMat patStats.c, 193 outputRealPats realIo.c, 227 realIo.h, 241 outputRealPatsWCentroid realIo.c, 228

pam110	pam380
matrices.h, 161	matrices.h, 163
pam120	pam390
matrices.h, 161	matrices.h, 164
pam130	pam400
matrices.h, 161	matrices.h, 164
pam140	pam430
matrices.h, 161	matrices.h, 164
pam150	pam440
matrices.h, 162	matrices.h, 164
pam160	pam450
matrices.h, 162	matrices.h, 164
pam190	pam460
•	±
matrices.h, 162	matrices.h, 164
pam200	pam490
matrices.h, 162	matrices.h, 164
pam210	pam500
matrices.h, 162	matrices.h, 164
pam220	parseExtraParams
matrices.h, 162	gemoda-r.c, 129
pam230	parseRealData
matrices.h, 162	realIo.c, 230
pam240	patStats.c, 182
	- ~
matrices.h, 162	patStats.c
matrices.h, 162 pam250	±
pam250	calcStatAllCliqs, 183
pam250 matrices.h, 162	calcStatAllCliqs, 183 calcStatCliq, 183
pam250 matrices.h, 162 pam260	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184
pam250 matrices.h, 162 pam260 matrices.h, 162	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam320 matrices.h, 163 pam330	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam320 matrices.h, 163 pam330	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199 freeD, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163 pam340 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163 pam340	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199 freeD, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163 pam340 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199 freeD, 199 getStatMat, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163 pam340 matrices.h, 163 pam340 matrices.h, 163 pam360	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199 freeD, 199 getStatMat, 199 sortByStats, 199
pam250 matrices.h, 162 pam260 matrices.h, 162 pam280 matrices.h, 163 pam290 matrices.h, 163 pam300 matrices.h, 163 pam310 matrices.h, 163 pam320 matrices.h, 163 pam330 matrices.h, 163 pam330 matrices.h, 163 pam340 matrices.h, 163 pam360 matrices.h, 163	calcStatAllCliqs, 183 calcStatCliq, 183 cumDMatrix, 184 freeD, 186 getLargestLength, 187 getLargestSupport, 187 getStatMat, 188 increaseMem, 191 measureDiagonal, 193 oldGetStatMat, 193 sortByStats, 196 statCompare, 197 patStats.h, 198 patStats.h calcStatAllCliqs, 199 cumDMatrix, 199 freeD, 199 getStatMat, 199 sortByStats, 199 phat_t75_b73

1 164	: ICD I
matrices.h, 164	printSPats
phat_t85_b82	words.c, 251
matrices.h, 165	pruneBitGraph
popAllCll	gemoda-r.c, 130
convll.c, 74	gemoda-s.c, 150
convll.h, 102	newConv.c, 176
popCll	pruneCll
convll.c, 75	convll.c, 79
convll.h, 102	newConv.c, 178
popMemStack	pushClique
convll.c, 76	convll.c, 81
convll.h, 103	convll.h, 104
popWholeMemStack	pushCll
convll.c, 76	convll.c, 82
convll.h, 103	convll.h, 104
pos	pushConvClique
sOffset_t, 19	convll.c, 83
prev	convll.h, 104
sOffset_t, 19	pushcSet
printBinaryBitSet	convll.c, 84
bitSet.c, 47	convll.h, 105
bitSet.h, 63	pushMemStack
printBitGraph	convll.c, 84
bitSet.c, 48	convll.h, 105
bitSet.h, 63	pushOnRdhSeq
printBitSet	realIo.c, 231
bitSet.c, 48	Tearro.e, 231
bitSet.h, 63	rdh_t, 13
printCll	indexSize, 13
convll.c, 77	indexToPos, 13
convll.h, 103	indexToSeq, 13
printCllPattern	label, 14
convll.c, 77	offsetToIndex, 14
convli.h, 103	seq, 14
printCSet	size, 14
convll.c, 78	ReadFile
printFSeqSubSeq	fastaSeqIO.c, 113
fastaSeqIO.c, 113	ReadFSeqs
	-
fastaSeqIO.h, 120	fastaSeqIO.c, 115
printMemberStacks	fastaSeqIO.h, 120
convll.c, 79	readRealData
convll.h, 104	realIo.c, 233
printRdhSeq	realIo.h, 241
realIo.c, 231	ReadTxtSeqs
printSHash	fastaSeqIO.c, 117
words.c, 250	fastaSeqIO.h, 121

realCompare.c, 201	freeRdh, 240
realCompare.c	getRdhDim, 240
generalMatchFactor, 202	getRdhIndexSeqPos, 240
getCompFunc, 203	initRdhIndex, 241
massSpecCompareWElut, 203	outputRealPats, 241
realComparison, 205	outputRealPatsWCentroid, 241
rmsdCompare, 206	readRealData, 241
realCompare.h, 209	removeSupers
realCompare.h	convll.c, 85
generalMatchFactor, 210	convil.h, 105
getCompFunc, 211	rmsdCompare
massSpecCompareWElut, 210	realCompare.c, 206
realComparison, 211	realCompare.h, 211
rmsdCompare, 211	searchMemsWithList
realComparison	
realCompare.c, 205	convll.c, 87
realCompare.h, 211	convll.h, 106
realIo.c, 212	searchSHash
realIo.c	words.c, 251
checkRealDataFormat, 213	seq
countFields, 215	fSeq_t, 11
countTotalFields, 216	rdh_t, 14
findCliqueCentroid, 217	sOffset_t, 19
freeRdh, 219	set
getRdhDim, 220	cnode, 8
getRdhIndexSeqPos, 220	setFalse
getRdhLabel, 221	bitSet.c, 49
getRdhSeqLength, 221	bitSet.h, 63
getRdhValue, 222	setRdhColFromString
initRdh, 222	realIo.c, 234
initRdhGslMat, 223	setRdhIndex
initRdhIndex, 224	realIo.c, 235
makeAlternateCentroid, 226	setRdhLabel
outputRealPats, 227	realIo.c, 236
outputRealPatsWCentroid, 228	setRdhValue
parseRealData, 230	realIo.c, 237
•	setStackTrue
printRdhSeq, 231	
pushOnRdhSeq, 231	convll.c, 88
readRealData, 233	convll.h, 106
setRdhColFromString, 234	setTrue
setRdhIndex, 235	bitSet.c, 49
setRdhLabel, 236	bitSet.h, 63
setRdhValue, 237	SHASH_MAX_KEY_SIZE
wordToDouble, 237	words.c, 245
realIo.h, 239	sHash_t, 15
realIo.h	sHash_t

1 1 15	61
hash, 15	sSize_t, 23
hashSize, 15	stat
iHashSize, 15	cnode, 8
totalSize, 16	statCompare
sHashEntry_t, 17	patStats.c, 197
sHashEntry_t	stop
data, 17	sSize_t, 23
idx, 17	string
key, 17	sPat_t, 22
L, 17	support
sieve3	sPat_t, 22
words.c, 253	swapNodecSet
singleCliqueConv	convll.c, 91
convll.c, 89	convll.h, 107
convll.h, 106	
singleLinkage	tf
newConv.c, 180	bitSet_t, 6
size	totalSize
bitGraph_t, 4	sHash_t, 16
cSet_t, 9	
rdh_t, 14	uniqClique
sSize_t, 23	convll.c, 91
slots	convll.h, 107
bitSet_t, 5	usage
sOffset_t, 19	gemoda-r.c, 132
sOffset_t	gemoda-s.c, 152
next, 19	
pos, 19	wholeCliqueConv
prev, 19	convll.c, 93
seq, 19	convll.h, 108
sortByStats	wholeRoundConv
patStats.c, 196	convll.c, 95
patStats.h, 199	convll.h, 108
spat.h, 243	words.c, 244
sPat_t, 21	countWords2, 245
sPat_t	destroySHash, 247
length, 21	destroySPatA, 248
offset, 21	hash1, 248
string, 22	hashpjw, 249
support, 22	initSHash, 249
sSize_t, 23	printSHash, 250
sSize_t	printSPats, 251
size, 23	searchSHash, 251
start, 23	SHASH_MAX_KEY_SIZE, 245
stop, 23	sieve3, 253
start	wordToDouble

```
realIo.c, 237
WriteFSeqA
fastaSeqIO.c, 117
fastaSeqIO.h, 121
yankCll
convll.c, 96
convll.h, 108
```