gemoda Reference Manual

Generated by Doxygen 1.4.4

Fri Mar 3 09:05:14 2006

Contents

Chapter 1

gemoda Directory Documentation

1.1 FastaSeqIO/ Directory Reference

Files

- file fastaSeqIO.c
- file fastaSeqIO.h

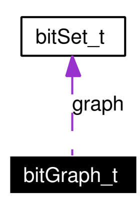
Chapter 2

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().

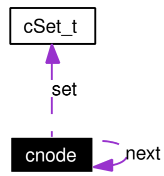
The documentation for this struct was generated from the following file:

- 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(), getLargestSupport(), initheadCll(), main(), makeAlternateCentroid(), mergeIntersect(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), pruneCll(), 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(), findCliqueCentroid(), main(), makeAlternateCentroid(), mergeIntersect(), mllToCSet(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), printCSet(), pruneCll(), pushConvClique(), removeSupers(), swapNodecSet(), uniqClique(), 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(), findCliqueCentroid(), getLargestSupport(), main(), mllToCSet(), outputRealPats(), outputRealPatsWCentroid(), printCll(), printCllPattern(), printCSet(), pruneCll(), removeSupers(), 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(), outputRealPats(), 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.

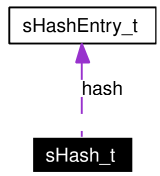
Referenced by initRdh(), initRdhIndex(), and main().

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().

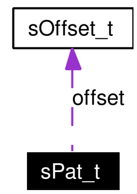
The documentation for this struct was generated from the following file:

- 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

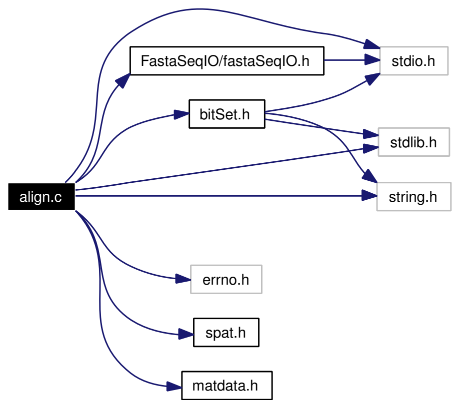
Chapter 3

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().

```
{
  int i;
  int points = 0;
  int x, y;

    // Go over each character in the L-length window
    for (i = 0; i < L; i++)
    {

    // The integer corresponding to the character in
    // the first string, so that we can look it up
    // in one of our scoring matricies.
    x = aaOrder[(int) s1[i]];

    // And for the second character
    y = aaOrder[(int) s2[i]];

    // If the characters aren't going to be in the scoring
    // matrix, they get a -1 value...which we'll give zero
    // points to here.
    if (x != -1 && y != -1)
    {

        // Otherwise, they get a score that is looked up
        // in the scoring matrix
        points += mat[x][y];
    }
    }
```

```
  return points;
}
```

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().

```
{
  bitGraph_t * sg = NULL;
  int score;
  int i, j;

    // Assign a new bitGraph_t object, with (wc x wc) possible
    // true/false values
    sg = newBitGraph (wc);
  for (i = 0; i < wc; i++)
    {
       for (j = i; j < wc; j++)
     {

         // Get the score for the alignment of word i and word j
         score =
         alignMat (words[i].string, words[j].string, words[i].length, mat);

         // If that score is greater than threshold, set
         // a bit to 'true' in our bitGraph_t object
         if (score >= threshold)
         {

         // We use 'bitGraphSetTrueSym' because, if i=j,
         // then j=i for most applications.  However, this
         // can be relaxed for masochists.
         bitGraphSetTrueSym (sg, i, j);
         }
     }
     }

     // Return a pointer to this new bitGraph_t object
```

```
    return sg;
}
```

3.1.4 Variable Documentation

3.1.4.1 const int aaOrder[ ]

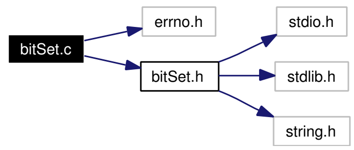
Definition at line 32 of file matrices.h.

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 bitcount32 (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.

```
{
  /*
     works for 32-bit numbers only
   */
  /*
     fix last line for 64-bit numbers
   */

  register unsigned int tmp;

  tmp = n - ((n >> 1) & 033333333333) - ((n >> 2) & 011111111111);
  return ((tmp + (tmp >> 3)) & 030707070707) % 63;
}
```

3.2.2.2 int bitcount32_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().

```
{
  // works only for 32-bit ints

  return bits_in_char[n & 0xffu]
    + bits_in_char[(n >> 8) & 0xffu]
    + bits_in_char[(n >> 16) & 0xffu] + bits_in_char[(n >> 24) & 0xffu];
}
```

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.

```
{
  n = PCCOUNT (n, 0);
  n = PCCOUNT (n, 1);
  n = PCCOUNT (n, 2);
  n = PCCOUNT (n, 3);
  n = PCCOUNT (n, 4);
  n = PCCOUNT (n, 5);       // for 64-bit integers
  return n;
}
```

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().

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

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 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 598 of file bitSet.c.

References bitSetIntersection(), and bitGraph_t::graph.

Referenced by getStatMat(), and oldGetStatMat().

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

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.

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

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().

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

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().

```
{
  int i;
  for (i = 0; i < bg->size; i++)
    {
      setFalse (bg->graph[i], i);
    }
  return 0;
}
```

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().

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

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().

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

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().

```
{
  int i;
  for (i = 0; i < bg->size; i++)
    {
      setTrue (bg->graph[i], i);
    }
  return 0;
}
```

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().

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

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.

```
{
  int i;
  if ((s1->slots != s2->slots) || (s1->slots != s3->slots)
      || (s1->slots != s4->slots))
    {
      fprintf (stderr, "Sets aren't same size!\n");
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < s1->slots; i++)
    {
      s4->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
      s4->tf[i] = BSINTERSECTION (s3->tf[i], s4->tf[i]);
    }
  return 0;
}
```

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.

```
{
  int i;
  if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
    {
      fprintf (stderr, "Sets aren't same size!\n");
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < s1->slots; i++)
    {
```

```
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 {
301   int i;
302   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
303     {
304       fprintf (stderr, "Sets aren't same size!\n");
305       fprintf (stderr, "set 1 slots = %d\n", s1->slots);
306       fprintf (stderr, "set 2 slots = %d\n", s2->slots);
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");
```

```
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < s1->slots; i++)
    {
      s3->tf[i] = (s1->tf[i] + s2->tf[i]);
    }
  return 0;
}
```

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().

```
{
  int i;
  if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
    {
      fprintf (stderr, "Sets aren't same size!\n");
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < s1->slots; i++)
    {
      s3->tf[i] = BSUNION (s1->tf[i], s2->tf[i]);
    }
  return 0;
}
```

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().

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

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.

```
{
  int i;
  if (bg1->size != bg2->size)
    {
      fprintf (stderr, "Graphs are not the same size!");
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < bg1->size; i++)
    {
      copySet (bg1->graph[i], bg2->graph[i]);
    }
  return 0;
}
```

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().

```
{
  int i;
  if (s1->slots != s2->slots)
    {
      fprintf (stderr, "Sets are not the same size!");
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < s1->slots; i++)
    {
      s2->tf[i] = s1->tf[i];
    }
  return 0;
}
```

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.

```
{
  int i;
  int sum = 0;
  // Iterate over all bitSets in the bitGraph
  for (i = 0; i < bg->size; i++)
    {
      sum += countSet (bg->graph[i]);
    }
  return sum;
}
```

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().

```
{
  int i;
  int sum = 0;
  int (*bitCounter) () = &bitcount32_precomp;
  // Currently there is no support for 64-bit architectures.

  if (sizeof (bit_t) * 8 != 32)
    {
      fprintf (stderr,
           "\nSorry, no support for 64-bit architectures just yet! - countSet\n");
      fflush (stderr);
      exit (0);
    }

  // Just count the number of true bits in each char, and do this for
  // (num of chars per int) chars.
  for (i = 0; i < s1->slots; i++)
    {
      sum += bitCounter (s1->tf[i]);
    }
```

```
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;
856   if (bg != NULL)
857     {
858       if (bg->graph != NULL)
859     {
860       for (i = 0; i < bg->size; i++)
861         {
862           deleteBitSet (bg->graph[i]);
863         }
864       free (bg->graph);
865       bg->graph = NULL;
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().

```
160 {
161   if (s1->tf != NULL)
162     {
163       free (s1->tf);
164       s1->tf = NULL;
165     }
```

```
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 < bg1->size; i++)
795     {
796       emptySet (bg1->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 < bg1->size; i++)
779     {
780       fillSet (bg1->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.

```
{
  int i;
  for (i = 0; i < s1->slots; i++)
    {
      s1->tf[i] = ~s1->tf[i];
    }
  return 0;
}
```

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.

```
{
  int i;
  for (i = 0; i < bg1->size; i++)
    {
      if (checkBit (bs, i))
    {
      bitSetIntersection (bg1->graph[i], bs, bg1->graph[i]);
    }
      else
    {
      emptySet (bg1->graph[i]);
    }
    }
  return 0;
}
```

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().

```
{
  bit_t *b = (bit_t *) malloc (bytes);
  if (b == NULL)
    {
      fprintf (stderr, "\nMemory error --- couldn't allocate bitArray!"
           " - newBitArray\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // Set them all false
  memset (b, 0, bytes);
  return b;
}
```

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().

```
{
  bitGraph_t *bg = NULL;
  int i;
  bg = (bitGraph_t *) malloc (sizeof (bitGraph_t));
  if (bg == NULL)
    {
      fprintf (stderr, "Memory error - Cannot allocate bitGraph - "
           "newBitGraph\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  bg->size = size;
  bg->graph = (bitSet_t **) malloc (size * sizeof (bitSet_t *));
  if (bg->graph == NULL)
    {
      fprintf (stderr, "Memory error - Cannot allocate bitGraphGraph - "
           "newBitGraph\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < size; i++)
    {
      bg->graph[i] = newBitSet (size);
    }
  return bg;
}
```

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().

```
{
  bitSet_t *s1 = (bitSet_t *) malloc (sizeof (bitSet_t));
  if (s1 == NULL)
    {
      fprintf (stderr, "\nMemory error --- couldn't allocate biSet!"
           " - newBitSet\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // Fill in details about the bitSet, allocate bitSet
  s1->max = size;
  s1->slots = BSNUMSLOTS (size);
  s1->bytes = s1->slots * sizeof (bit_t);
  s1->tf = newBitArray (s1->bytes);
  return s1;
}
```

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().

```
{
  // slot is our starting slot, the
  // slot containing bit 'start'
  int slot = BITSLOT (start);
  int i;
  // 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
  int stop;
  bit_t bitFalse;
```

```
  memset (&bitFalse, 0, sizeof (bit_t));


  // s1->max is the number of bits in s1
  // test to see if we're looking too high
  if (start >= s1->max)
    {
      return -1;
    }
  // s1->slots is the number of available slots
  // skip over empty slots
  while (slot < s1->slots)
    {
      /*
         printf("w");
       */
      if (s1->tf[slot] != bitFalse)
    {
      // this slot is not empty

      // if each slot is, say 32 bits and
      // we asked for nextBitBitSet(s1, 5),
      // then slot 0 will be non-zero.  but,
      // instead of starting at 0, start at 5!
      if (BSBITSIZE * slot > start)
        {
          // set start to index of first
          // bit in this slot
          start = BSBITSIZE * slot;
        }
      // set the stop, with a a check against the 'max'
      // element of the bitSet_t object
      if (BSBITSIZE * (slot + 1) > s1->max)
        {
          stop = s1->max;
        }
      else
        {
          stop = BSBITSIZE * (slot + 1);
        }
      for (i = start; i < stop; i++)
        {
          if (checkBit (s1, i))
        {
          return i;
        }
        }
    }
      slot++;
    }
  return -1;
}
```

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().

```
{
  int i;
  for (i = 0; i < s1->max; i++)
    {
      printf ("%d", (BSTEST (s1->tf, i) ? 1 : 0));
    }
  return 0;
}
```

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().

```
{
  int i;
  for (i = 0; i < bg->size; i++)
    {
      printBinaryBitSet (bg->graph[i]);
      printf ("\n");
    }
  return 0;
}
```

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().

```
{
  int i;
  printf ("bitSet (addr = %d; %d members)\n", (int) s1, countSet (s1));
  printf ("\tmax = %d\n", s1->max);
  printf ("\tslots = %d\n", s1->slots);
  printf ("\tbytes = %d\n", s1->bytes);
  printf ("\tmembers =");


  for (i = 0; i < s1->max; i++)
    {
      if (BSTEST (s1->tf, i))
    {
      printf (" %d", i);
    }
    }
  printf ("\n");
  return 0;
}
```

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().

```
{
  /*
     if (BSNUMSLOTS(x) > s1->slots) { Conditional changed, 5/25, by MPS: check x against s1->max,
     should be safer
   */
  if (x >= s1->max)
    {
      fprintf (stderr, "Set isn't large enough! - setFalse\n");
      fflush (stderr);
      exit (0);
    }
  BSCLEAR (s1->tf, x);
  return 0;
}
```

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.

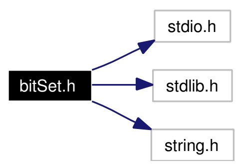
Referenced by bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), filterIter(), findCliques(), and setStackTrue().

```
{
  if (x >= s1->max)
    {
      fprintf (stderr, "Set isn’t large enough! - setTrue\n");
      fflush (stderr);
      exit (0);
    }
  BSSET (s1->tf, x);
  return 0;
}
```

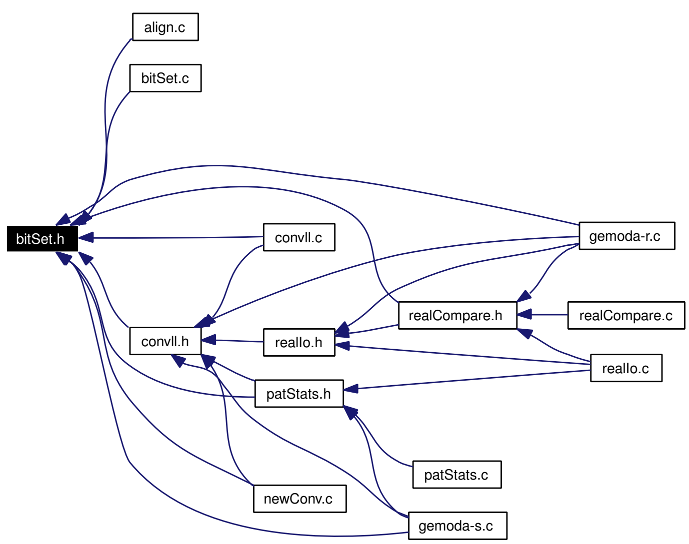
3.3 bitSet.h File Reference

```
#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(), deleteBitGraph(), filterGraph(), findCliques(), getStatMat(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

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 bitSet_t::tf.

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

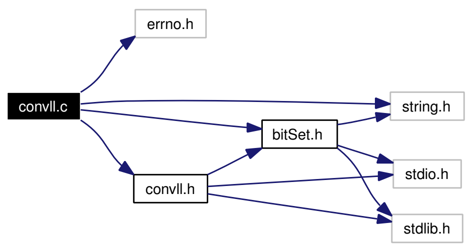
References BSSET, bitSet_t::max, and bitSet_t::tf.

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 numOffsets, 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 {
484   int i = 0;
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     {
503       fprintf (stderr, "\nNULL set for clique! - addToStacks\n");
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);
215   int i = 0, start = 0;
216   cSet_t *holder = (cSet_t *) malloc (sizeof (cSet_t));
217
218   // Memory error checking
219   if (holder == NULL)
220     {
221       fprintf (stderr, "\nMemory Error - bitSetToCSet - [1]\n%s\n",
222            strerror (errno));
223       fflush (stderr);
```

```
      exit (0);
    }
  // More memory checking
  holder->members = (int *) malloc (cliqueSize * sizeof (int));
  if (holder->members == NULL)
    {
      fprintf (stderr, "\nMemory Error - bitSetToCSet - [2]\n%s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }

  // For each member of the clique in the bitSet,
  for (i = 0; i < cliqueSize; i++)
    {
      // Find the next one, add its location to the members array
      holder->members[i] = nextBitBitSet (clique, start);
      // (But check for errors... if we get to the end of the
      // bitSet, then something is wrong)
      if (holder->members[i] == -1)
    {
      fprintf (stderr, "\nClique error - not enough members\n");
      fflush (stderr);
      exit (0);
    }
      // Increment to move on in the nextBitBitSet search
      start = holder->members[i] + 1;
    }

  holder->size = cliqueSize;
  return holder;
}
```

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().

```
{
  int *seqNums = NULL;
  int thisSeq = 0, i = 0, j = 0;
  seqNums = (int *) malloc (p * sizeof (int));

  if (seqNums == NULL)
    {
      fprintf (stderr, "Memory error - checkCliquecSet\n%s\n",
           strerror (errno));
```

```
      fflush (stderr);
      exit (0);
    }
  // Initialize an array of integers of size p to sentinel values of -1
  for (i = 0; i < p; i++)
    {
      seqNums[i] = -1;
    }
  j = 0;

  if (cliquecSet->size < 1)
    {
      fprintf (stderr, "\nClique of zero size! - checkCliquecSet\n");
      fflush (stderr);
      exit (0);
    }
  // Find the first sequence number.
  seqNums[0] = indexToSeq[cliquecSet->members[0]];
  // Iterate over the remaining size of the clique
  for (i = 1; i < cliquecSet->size; i++)
    {
      // Find the next sequence number.
      thisSeq = indexToSeq[cliquecSet->members[i]];
      // The member list is in monotonic order, so we only need
      // to compare the current member to the previous member to
      // find out if it comes from the same sequence.
      // If it’s not from the same sequence, increment the unique
      // sequence counter (j), store the next sequence number
      // in the array.
      // Also check to see if we’ve already reached the p threshold,
      // and if so, then bail out.
      if (thisSeq != seqNums[j])
    {
      j++;
      seqNums[j] = thisSeq;
      if (j == p - 1)
        {
          break;
        }
    }
    }

  // Now just see what the value of the last number in the array is;
  // 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
  // the -p criterion.
  if (seqNums[p - 1] == -1)
    {
      free (seqNums);
      return (0);
    }
  else
    {
      free (seqNums);
      return (1);
    }
}
```

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 {
1420   int i = 0;
1421   mll_t **memList = NULL;
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     }
1432   // The number of offsets will never change, so this can be defined
1433   // now, though we will have to change what is in these arrays later.
1434   for (i = 0; i < numOffsets; i++)
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
1442   // the initial set of cliques must be non-null.  Those are then
1443   // convolved and the linked list for the next round is set to head,
1444   // so this continues until the linked list for the "next round" at
1445   // the end of some round is null.
1446   while (*head != NULL)
1447     {
1448       // First we get the inverse information for this round: find
1449       // out which cliques each offset is a member of.
1450       memList = fillMemberStacks (*head, memList);
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.
```

```
      memList = emptyMemberStacks (memList, numOffsets);
      popAllCll (*head);
      // Enforce the -p flag for subsequent rounds.
      if (p > 1)
    {
      nextPhase = pruneCll (nextPhase, indexToSeq, p);
    }
      // And move on to the next round of convolution.
      *head = nextPhase;
      length++;
    }

  free (memList);

  return allCliques;
}
```

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().

```
{
  int i = 0;

  for (i = 0; i < size; i++)
    {
      memberStacks[i] = popWholeMemStack (memberStacks[i]);
    }

  return memberStacks;
}
```

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().

```
{
  cll_t *curr = head;
  // Just go down the linked list calling addToStacks
  while (curr != NULL)
    {
      memberStacks = addToStacks (curr, memberStacks);
      curr = curr->next;
    }

  return memberStacks;
}
```

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().

```
{
  // Check to make sure that the head is not already initialized.
  if (head->set != NULL)
    {
      printf ("Stack head already initialized!");
      exit (0);
    }
  // Make the head’s set pointer point to the new set.
  head->set = newset;
  return head;
}
```

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().

```
{

  int i = 0, j = 0, status = 0;

  // Make sure we are still in-bounds, otherwise we bail out
  // We'll refer to the offset currently being analyzed from the
  // first clique as the 'first offset' and the offset currently
  // being analyzed from the second clique as the 'second offset'
  while ((i < first->set->size) && (j < second->set->size))
    {
      // If the second offset is earlier than the first offset plus
      // one, then we move on to the next possible second offset
      if ((first->set->members[i] + 1) > second->set->members[j])
    {
      j++;
    }
      // If the second offset is later than the first offset plus
      // one, then we move on the next possible first offset
      else if ((first->set->members[i] + 1) < second->set->members[j])
    {
      i++;
    }
      // Otherwise, the second offset is equal to the first offset
      // plus one, so we have an extendable node.  Push that on
      // to the intersection stack, move both the first and second
      // offsets to their respective next possible offsets, and
      // increment the support counter for the new clique (status)
      else
    {
      intersection = pushMemStack (intersection, first->set->members[i]);
      i++;
      j++;
      status++;
    }
    }

  // Send the value of the clique's new support out of this function
  *newSupport = status;
  return intersection;
}
```

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().

```
{
  int sizecount = 0, i = 0;
  cSet_t *cliqueCset = malloc (sizeof (cSet_t));
  mll_t *head = clique;
  if (cliqueCset == NULL)
    {
      fprintf (stderr, "Memory error - mllToCSet cSet\n%s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // First count up how many members there are in the member linked list
  while (head != NULL)
    {
      sizecount++;
      head = head->next;
    }

  head = clique;
  cliqueCset->size = sizecount;
  cliqueCset->members = (int *) malloc (sizecount * sizeof (int));

  if (cliqueCset->members == NULL)
    {
      fprintf (stderr, "Memory error - mllTlCSet cliquemembers\n%s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // In order to stay in the same format as with bitSet translation to
  // cSet, we ensure that the ids of the members are ascending with
  // ascending index number in the cSet.  This is accomplished by noting
  // that since the intersection members are pushed onto the stack,
  // a LIFO operation, that the first intersected nodes off the stack
  // will have the highest ids, so we will put them at the end of
  // the members array with the higher index values.
  for (i = sizecount - 1; i >= 0; i--)
    {
      cliqueCset->members[i] = head->cliqueMembership;
      head = head->next;
    }

  return cliqueCset;
}
```

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   // 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     }
```

```
  // Both the members and set have been freed, so now can free the cll_t
  // without leaking anything.

  free (head);
  head = NULL;
  return newHead;
}
```

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().

```
{
  // by default the new head is NULL...is important later
  mll_t *newHead = NULL;
  if (head == NULL)
    {
      fprintf (stderr, "\nCan’t pop a null linked list - popMemStack\n");
      fflush (stderr);
      exit (0);
    }
  if (head->next != NULL)
    {
      newHead = head->next;
    }
  free (head);
  head = NULL;
  return newHead;
}
```

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().

```
{
  while (head != NULL)
    {
```

```
      head = popMemStack (head);
    }
  return head;
}
```

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.

```
{
  int i = 0;
  cll_t *curr = head;
  while (curr != NULL)
    {
      printf ("id = %d\n", curr->id);
      // Make sure the clique is nonzero in size before attempting
      // to print it
      if ((curr->set != NULL) && (curr->set->size > 0))
    {
      if (curr->length >= 0)
        {
          printf ("Length = %d\n", curr->length);
        }
      printf ("Size = %d\n", curr->set->size);
      printf ("Members = \n");
      for (i = 0; i < curr->set->size; i++)
        {
          printf ("\t%d\n", curr->set->members[i]);
        }
      printf ("***********************************************\n");
    }
      else
    {
      fprintf (stderr, "\nClique has no members! -- printCll\n");
      fflush (stderr);
      exit (0);
    }
      curr = curr->next;
    }
  return EXIT_SUCCESS;
}
```

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.

```
{
  int i = 0;

  printf ("\nSupport = %d\t(id = %d)\n", node->set->size, node->id);
  printf ("Members = \n");
  for (i = 0; i < node->set->size; i++)
    {
      printf ("\t%d\n", node->set->members[i]);
    }
  return 1;
}
```

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.

```
{
  int i = 0;
  if (node->size == 0)
    {
      fprintf (stderr, "cSet has no members! - printCSet\n");
      fflush (stderr);
      exit (0);
    }
  else
    {
      printf ("\nSupport = %d\n", node->size);
      printf ("Members = \n");
      for (i = 0; i < node->size; i++)
    {
      printf ("\t%d\n", node->members[i]);
    }
      return 1;
    }
}
```

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.

```
{
  int i = 0;
  mll_t *curr = NULL;

  for (i = 0; i < size; i++)
    {
      curr = memberStacks[i];
      printf ("Offset %d: ", i);
      while (curr != NULL)
    {
      printf ("%d,", curr->cliqueMembership);
      curr = curr->next;
    }
      printf ("\n");
    }
}
```

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().

```
{
  int i = 0, j = 0, thisSeq = 0;
  int *seqNums = NULL;
  cll_t * curr = head;
  cll_t * prev = NULL;
  cll_t * storage = NULL;

    // We'll do this similar to the pruneBitGraph function... we will
    // keep track of which source sequence each motif occurrence was in.
    // Again, since the occurrences are listed monotonically, we only
    // need to compare the last non-sentinel index to the current
    // sequence number.
```

```
    seqNums = (int *) malloc (p * sizeof (int));
  if (seqNums == NULL)
    {
      fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  while (curr != NULL)
    {

    // First make sure the set size is at least p.
    // This is redundant, but extremely simple and not expensive,
    // so we'll leave it in just as a check.
    if (curr->set->size < p)
    {
      if (prev != NULL)
        {
          prev->next = curr->next;
        }
      else
        {
          head = curr->next;
        }
      storage = curr->next;
      free (curr->set->members);
      free (curr->set);
      free (curr);
      curr = storage;
      continue;
    }
      for (i = 0; i < p; i++)
    {
      seqNums[i] = -1;
    }
      j = 0;
      seqNums[0] = indexToSeq[curr->set->members[0]];

    // Note, we've checked to make sure size > p, and we know
    // p must be 2 or greater, so we can start at 1 without
    // worrying about segfaulting
    for (i = 1; i < curr->set->size; i++)
    {
      thisSeq = indexToSeq[curr->set->members[i]];
      if (thisSeq != seqNums[j])
        {
          j++;
          seqNums[j] = thisSeq;
          if (j == p - 1)
        {
          break;
        }
        }
    }

    // Same story as before... if the last number is -1,
    // then we didn't have enough to fill up the <p> different
    // 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 {
347   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
352   // and only proceed if that criterion is met.  Otherwise, free the
353   // memory that we had allocated up to this point.
354   if (p > 1)
355     {
356       if (checkCliquecSet (cliquecSet, indexToSeq, p))
357     {
358       cliquecounter++;
359       /*
360          printf("%d\n",cliquecounter);
```

```
       */
      /*
         fflush(stdout);
       */
      head = pushcSet (head, cliquecSet);
    }
      else
    {
      free (cliquecSet->members);
      free (cliquecSet);
    }
      // If the -p flag wasn't set, then just push the cSet onto the linked
      // list.
    }
  else
    {
      cliquecounter++;
      /*
         printf("%d\n",cliquecounter);
       */
      /*
         fflush(stdout);
       */
      head = pushcSet (head, cliquecSet);
    }
  return head;
}
```

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().

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

```
    {
      a->id = 0;
      a->next = NULL;
    }
  else
    {
      a->next = head;
      a->id = head->id + 1;
    }
  a->set = NULL;
  a->length = -1;
  a->stat = -1;
  return a;
}
```

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().

```
{
  int status = 0;
  cSet_t *cliquecSet = NULL;

  // First change the clique to something we can used more easily
  cliquecSet = mllToCSet (clique);
  // Then check to make sure it's unique by finding out its status
  status = uniqClique (cliquecSet, head);

  // printf("Candidate:\n");
  // printCSet(cliquecSet);

  // If we get -2, then this clique is a subset, so just free
  // the cSet we just made and move on.
  if (status == -2)
    {
      free (cliquecSet->members);
      free (cliquecSet);
      cliquecSet = NULL;
    }
  // If we get -1, then this is a unique clique, so push it on.
  else if (status == -1)
    {
```

```
      head = pushcSet (head, cliquecSet);
    }
  // Otherwise, this clique is a superset, so we'll first remove
  // all of the other cliques of which this is a superset.  Then
  // we'll swap out the first clique of which this is a superset
  // with this current clique.  The clique being removed is free'd
  // within the swapNode function.
  else
    {
      head = removeSupers (head, status, cliquecSet);
      head = swapNodecSet (head, status, cliquecSet);
    }
  return head;
}
```

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().

```
{
  head = pushCll (head);
  head = initheadCll (head, newset);
  return head;
}
```

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().

```
{
  mll_t *a = NULL;
  a = (mll_t *) malloc (sizeof (mll_t));
  // Memory error checking
  if (a == NULL)
    {
      fprintf (stderr, "\nMemory Error - pushMemStack: %s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }
  if (head == NULL)
    {
      a->next = NULL;
    }
  else
    {
      a->next = head;
    }
  // Store the number of the clique of which the node is a member.
  // Note that we assume no duplication, which is guaranteed
  // by our method of filling the member stacks, which is quite simple:
  // go through all members of a clique (which have no duplicates
  // because they are constructed from merge-intersections or from
  // bitSet_t's) and add that clique to each node's membership list.
  a->cliqueMembership = cliqueNum;
  return a;
}
```

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 uniqclique 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().

```
{
  int foundStatus = 0;
  cll_t *curr = head;
  cll_t *prev = NULL;
  int i = 0, j = 0, breakFlag = 0;
```

```

  while (curr != NULL)
    {
      if (curr->id == node)
    {
      foundStatus = 1;
      break;
    }
      curr = curr->next;
    }

  if (foundStatus == 0)
    {
      fprintf (stderr, "\nFirst clique not found! (removeSupers)\n");
      fflush (stderr);
      exit (0);
    }
  // Now this is trickier, to remove nodes from the middle of a linked
  // list; this means that we need to remember which node we were just
  // at so that we can connect it to the node after the one we are
  // about to delete.
  prev = curr;
  curr = curr->next;

  // This code is similar to that in uniqClique.
  // Descend through all members of the next level's linked list.
  while (curr != NULL)
    {
      i = 0;
      j = 0;
      breakFlag = 0;
      // The proposed convolved clique will be referred to as the
      // 'first' clique, and the current clique being analyzed
      // in the next level is the 'second' clique.
      // Continue if we have more members in both cliques.  We will
      // have already broken out if it is not possible for this
      // second clique to be a subset of the first.
      while ((i < newClique->size) && (j < curr->set->size))
    {
      // If the current member of the first clique is
      // less than the current member of the second clique
      // then it is still possible that the first is a
      // superset of the second, so move on to the next
      // member.
      if (newClique->members[i] < curr->set->members[j])
        {
          i++;
        }
      // If the current member of the first clique is greater
      // than the current member of the second clique, then
      // the proposed second clique cannot be a subset since
      // its members are all in ascending order.  We also
      // know that since the first clique already has
      // a subset in this linked list, the current node
      // cannot possibly be a superset of the proposed
      // clique, so we can just disregard that.  Thus,
      // we make a flag signifying this and break out.
```

```
      else if (newClique->members[i] > curr->set->members[j])
        {
          breakFlag = 1;
          break;
        }
      else
        {
          i++;
          j++;
        }
    }
      // If the breakflag is 1, then we know
      // that there is a member of the second clique not in the
      // first, and so the second is not a subset.  If the breakflag
      // is 0 but j is less than the second clique's size, then
      // we must have broken because we ran out of members in the
      // first clique... thus, there is a member of the second
      // clique not in the first.  Thus, only if the breakflag is
      // 0 and j is equal to the size of the second clique do we
      // know that every member of the second clique is in the first
      // and that the second clique can thus be removed.
      if ((breakFlag == 0) && (j == curr->set->size))
    {
      // Make the previous clique point to the next one
      // instead of the current one.
      prev->next = curr->next;
      // Free all of the memory used by the current clique.
      free (curr->set->members);
      free (curr->set);
      free (curr);
      curr = prev->next;
    }
      else
    {
      // Otherwise, the current second clique is not a
      // subset of the first, and we advance the prev and
      // curr pointers.
      prev = curr;
      curr = curr->next;
    }
    }
  return head;
}
```

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 emptySet(), and setStackTrue().

Referenced by wholeCliqueConv().

```
{
  int i = 0;
  emptySet (queue);

  // Go through each offset in the list
  for (i = 0; i < listsize; i++)
    {
      // Check to make sure that's a valid offset number, and if so
      // then set its stack true in the queue.
      if (list[i] + 1 < numOffsets)
    {
      queue = setStackTrue (memList, list[i] + 1, queue);
    }
      else
    {
      fprintf (stderr, "\nInvalid offset number! - searchMemsWithList\n");
      fprintf (stderr, "\nlist[i]+1 (%d) >= numOffsets (%d)\n",
           list[i] + 1, numOffsets);
      fflush (stderr);
      exit (0);
    }
    }

  return queue;
}
```

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().

```
{
  mll_t *curr = memList[i];

  // Traverse down the member linked list
  while (curr != NULL)
    {
      // Set the bit in queue corresponding to the current clique
      // membership true
      setTrue (queue, curr->cliqueMembership);
```

```
      curr = curr->next;
    }

  return queue;
}
```

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(), pushConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

```
{
  cll_t *first = NULL, *second = NULL;
  mll_t *survivingMems = NULL;
  // int flag = 0;
  int newSupport = 0;
  // cll_t *checker = head;

  // Check to make sure we're looking for legitimate cliques.
  if ((firstClique > head->id) || (secondClique > head->id))
    {
      fprintf (stderr, "\nNonexistent clique! - singleCliqueConv\n");
      fflush (stderr);
      exit (0);
    }
  // Our guesses depend on monotonic traversal.  If we don't find
  // the first clique, then bail out.
  while ((*firstGuess)->id != firstClique)
    {
      if ((*firstGuess)->next != NULL)
    {
      *firstGuess = (*firstGuess)->next;
    }
```

```
      else
    {
      fprintf (stderr, "\nFirst clique not found! - singleCliqueConv\n");
      fflush (stderr);
      exit (0);
    }
    }
  first = *firstGuess;

  // Our guesses depend on monotonic traversal.  If we don’t find
  // the second clique, then bail out.
  while ((*secondGuess)->id != secondClique)
    {
      if ((*secondGuess)->next != NULL)
    {
      (*secondGuess) = (*secondGuess)->next;
    }
      else
    {
      fprintf (stderr, "\nSecond clique not found! - singleCliqueConv\n");
      fflush (stderr);
      exit (0);
    }
    }
  second = *secondGuess;
  // Find out what the surviving members are when the first clique
  // is convolved with the second clique
  survivingMems =
    mergeIntersect (first, second, survivingMems, printStatus, &newSupport);

  // If the first clique is subsumed by the second, then it is not
  // maximal, so don’t print it.
  // printStatus true means print it!
  if (newSupport == first->set->size)
    {
      setFalse (printStatus, first->id);
    }
  // If the second clique is subsumed by the first, then it is not
  // maximal, so don’t print it.
  if (newSupport == second->set->size)
    {
      setFalse (printStatus, second->id);
    }

  // If the support of the clique just formed by convolution meets the
  // support criterion, then push it on to the linked list for
  // the next phase of convolution.
  if (newSupport >= support)
    {
      // printf("Push %d and %d\n",first->id,second->id);
      nextPhase = pushConvClique (survivingMems, nextPhase);
      // printf("---------\n");
      // printCll(nextPhase);
      // printf("---------\n");
    }
  // Pop the surviving members; they are no longer needed, as they
  // either didn’t meet the support criterion or have been pushed on
```

```
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
909   // 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     }
927   // Then we free the useless clique's members and its set data structure
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 {
823   int i = 0, j = 0;
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
835       // in the next level is the "second" clique.
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) &&
840          ((asubbflag == 1) || (bsubaflag == 1)))
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
846       // traversed in ascending order.
847       if (cliquecSet->members[i] < head->set->members[j])
848         {
849           i++;
850           asubbflag = 0;
851         }
852       // Similarly, if the current member of the second
853       // clique is less than the current member of the
854       // second clique, the second can't be a subset
```

```
   // of the first.
   else if (cliquecSet->members[i] > head->set->members[j])
    {
     j++;
     bsubaflag = 0;
    }
   // Otherwise, they matched this time, so move them
   // both on.
   else
    {
     i++;
     j++;
    }
  }

   // If the proposed clique is a subset of some other clique
   // in the next level, then return -2, and it won't be added.
   // (Note, this also is how exact duplicates are handled.)
   if ((asubbflag == 1) && (i == cliquecSet->size))
  {
   return (-2);
  }
   // If the proposed clique is a superset of some other clique(s)
   // in the next level, then return the id of the first clique
   // of which it is a superset.
   if ((bsubaflag == 1) && (j == head->set->size))
  {
   return (head->id);
  }
   // If the proposed clique has not been found to be a superset
   // or a subset yet, then move on to the next clique in
   // the next level.
   head = head->next;
  }
 // If we've gotten here, we've checked all cliques in the previous
 // level and haven't found the proposed clique to be a superset or
 // a subset... if so, then we're all good, so return a -1.
 return (-1);
}
```

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().

```
{
  bitSet_t *queue = NULL;
  cSet_t *cliquesToSearch = NULL;
  int i = 0;
  cll_t **secondGuess = NULL;

  // This bitSet will be used to create a "queue" of the different
  // cliques that must be convolved against the current primary clique.
  // A bitset is used to make it easy to deal with duplicates, where
  // multiple clique members' next offsets
  // are all members of some other specific clique.
  queue = newBitSet (head->id + 1);
  queue =
    searchMemsWithList (node->set->members, node->set->size, memList,
            numOffsets, queue);
  // We'll use this "secondGuess" to store where the previous clique
  // being convolved was... since we will progressing monotonically
  // in descending order, this will save us some time in traversing the
  // linked list looking for the clique that we want.
  secondGuess = (cll_t **) malloc (sizeof (cll_t *));
  if (secondGuess == NULL)
    {
      fprintf (stderr, "Memory error - wholeCliqueConv\n%s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // If the offsets that we are looking for are in no other cliques,
  // we can just bail out now.
  if (countSet (queue) == 0)
    {
      deleteBitSet (queue);
      return nextPhase;
    }
  // Otherwise, we start our secondGuess at the head and get going.
  *secondGuess = head;

  // We change the bitSet to something more useful.
  cliquesToSearch = bitSetToCSet (queue);

  // Note that we start from the end of the cSet member list so that
  // we can convolve the highest-id cliques first, which are at the
  // beginning of our stack of cliques.
  for (i = cliquesToSearch->size - 1; i >= 0; i--)
    {
      nextPhase = singleCliqueConv (head, node->id, firstGuess,
                    cliquesToSearch->members[i], secondGuess,
```

```
                     nextPhase, printStatus, support);
    }

  // And then we free everything that we created
  deleteBitSet (queue);
  free (cliquesToSearch->members);
  free (cliquesToSearch);
  free (secondGuess);
  return nextPhase;
}
```

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 wholeCliqueConv. 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().

```
{
  bitSet_t *printStatus = NULL;
  cll_t *curr = *head;
  cll_t *prev = NULL;
  cll_t *nextPhase = NULL;
  cll_t **firstGuess = NULL;

  // Create a bitset to keep track of print status for this level.
  // It starts off all true, and gets changed to false if the patterns
  // are not maximal.
  printStatus = newBitSet ((*head)->id + 1);
  fillSet (printStatus);
  firstGuess = (cll_t **) malloc (sizeof (cll_t *));
  if (firstGuess == NULL)
    {
      fprintf (stderr, "Memory error - wholeRoundConv\n%s\n",
           strerror (errno));
      fflush (stderr);
      exit (0);
    }
  // Start off at the head.
  *firstGuess = *head;
  // Convolve a whole clique at a time, traversing the linked list.
  // Note that firstGuess gets altered within the function.
  while (curr != NULL)
    {
```

```
      nextPhase =
    wholeCliqueConv (*head, curr, firstGuess, memList, numOffsets,
             nextPhase, printStatus, support);
      curr = curr->next;
    }

  // Now go back to the head for printing output
  curr = *head;

  // printf("\n****************************************************\n");
  // printf("Length = %d", length);
  // printf("\n****************************************************\n");

  // For each clique that is still 'true' in printStatus and is thus
  // maximal, perform some sort of output.  Yankcll will pull out the
  // clique and save it for printing at a later time.
  while (curr != NULL)
    {
      if (checkBit (printStatus, curr->id))
    {
      // This is the line that makes the allCliques output.
      // Can either printcll, or add to allCliques.
      // printCllPattern(curr, length);
      yankCll (head, prev, &curr, allCliques, length);
    }
      else
    {
      prev = curr;
      curr = curr->next;
    }
    }

  // And clean up.
  deleteBitSet (printStatus);
  free (firstGuess);
  return nextPhase;
}
```

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().

```
{
  if (*curr == NULL)
    {
      fprintf (stderr, "\nCan’t yank from end of cll!\n");
      fflush (stderr);
      exit (0);
    }
  // If we’re not on the head, change the previous node’s "next".
  // If we are on the head, make the new head be our current node’s "next".
  if (prev != NULL)
    {
      prev->next = (*curr)->next;
    }
  else
    {
      *head = (*curr)->next;
    }

  // Change next in curr, then change id and length information in curr
  (*curr)->next = *allCliques;

  if (*allCliques != NULL)
    {
      (*curr)->id = (*allCliques)->id + 1;
    }
  else
    {
      (*curr)->id = 0;
    }

  (*curr)->length = length;

  *allCliques = *curr;

  if (prev != NULL)
    {
      *curr = prev->next;
    }
  else
    {
      *curr = *head;
    }
  return (1);
}
```

3.4.3 Variable Documentation

3.4.3.1 int cliquecounter = 0

Definition at line 335 of file convll.c.

Referenced by pushClique().

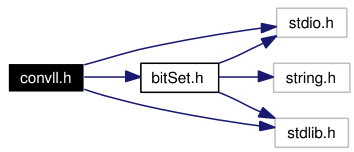
3.5 convll.h File Reference

`#include <stdio.h>`

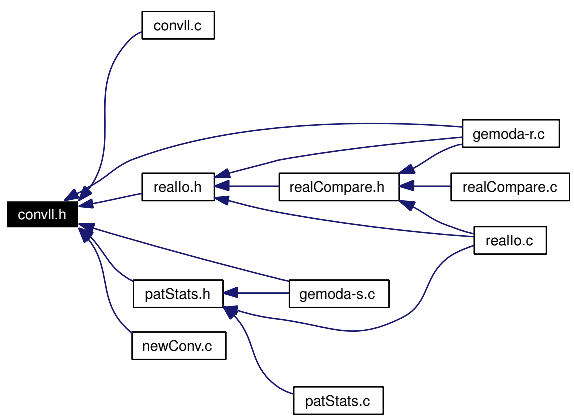
`#include <stdlib.h>`

`#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 numOffsets, 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 uniqclique 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(), pushConvClique(), 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 wholeCliqueConv. 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.

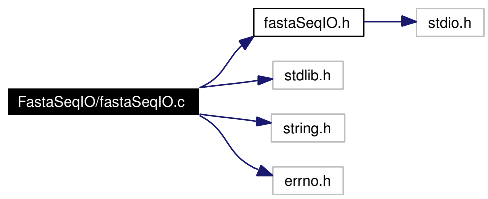
References cnode::id, and cnode::next.

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.

```
{
    long start;
    long count = 0;
    int myChar;
    int newLine = 1;
    start = ftell(INPUT);
    myChar = fgetc(INPUT);
    while (myChar != EOF) {
        if (newLine == 1 && myChar == '>') {
            count++;
        }
        if (myChar == '\n') {
            newLine = 1;
        } else {
            newLine = 0;
        }
        myChar = fgetc(INPUT);
    }
    fseek(INPUT, start, SEEK_SET);
    return count;
}
```

3.6.2.2 long countLines (FILE * *INPUT*)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

```
{
    long start;
    long count = 1;
    int myChar;
```

```
    int status = 0;
    start = ftell(INPUT);
    myChar = fgetc(INPUT);
    while (myChar != EOF) {
        if (myChar == '\n') {
            count++;
            status = 1;
        } else {
            status = 0;
        }
        myChar = fgetc(INPUT);
    }
    if (status == 1) {
        count--;
    }
    fseek(INPUT, start, SEEK_SET);
    return count;
}
```

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().

```
{
    int i;
    for (i = 0; i < numberOfSequences; i++) {
        if (arrayOfSequences[i].label != NULL) {
            free(arrayOfSequences[i].label);
        }
        arrayOfSequences[i].label = NULL;

        if (arrayOfSequences[i].seq != NULL) {
            free(arrayOfSequences[i].seq);
        }
        arrayOfSequences[i].seq = NULL;
    }
    if (arrayOfSequences != NULL) {
        free(arrayOfSequences);
    }
    arrayOfSequences = NULL;
    return EXIT_SUCCESS;
}
```

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 }
```

3.6.2.5 long measureLine (FILE ∗ *INPUT*)

Definition at line 25 of file fastaSeqIO.c.

Referenced by ReadFile().

```
26 {
27     long start;
28     long count = 0;
29     int myChar;
30     start = ftell(INPUT);
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.

```
14                                                   {
15     int i;
16     for(i=start; i<stop; i++){
17         putchar(seq->seq[i]);
18     }
19     return 0;
20 }
```

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().

```
{
    char **buf = NULL;
    long nl;
    long tls = 0;
    int i=0;

    nl = countLines(INPUT);
    if( nl == 0){
        fprintf(stderr, "\nNo sequences! Error!\n\n");
        fflush(stderr);
        return NULL;
    }
    buf = (char **) malloc ( (int)(nl+1) * sizeof(char *));
    if ( buf == NULL){
        fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
        fflush(stderr);
        exit(0);
    }

    //  measure the first line
    tls = measureLine(INPUT) + 1;
    if(tls != 0){
        buf[i] = (char *) malloc ( tls * sizeof(char));
        if ( buf[i] == NULL){
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
            fflush(stderr);
            exit(0);
        }
    }
    fgets(buf[i], tls, INPUT);
    do{
        if(buf[i][ strlen(buf[i])-1 ] == ’\n’){
            buf[i][ strlen(buf[i])-1 ] = ’\0’;
        }
        tls = measureLine(INPUT) + 1;
        if(tls != 0){
            i++;
            buf[i] = (char *) malloc ( tls * sizeof(char) );
            if ( buf[i] == NULL){
                fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
                fflush(stderr);
                exit(0);
            }
        }
    }while( fgets(buf[i], tls, INPUT) != NULL );
    free(buf[i]);
    buf = (char **) realloc ( buf, i * sizeof(char *) );
    if ( buf == NULL){
        fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
```

```
155         fflush(stderr);
156         return NULL;
157     }
158     //  I think that 'i' might actually be the # of lines
159     //  plus one here?  somehow line 131 isn't being freed,
160     //  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, sSize_t::start, and sSize_t::stop.

Referenced by main().

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

```
            k++;
        }

        //  record how large each sequence is
        i = -1;
        for( j=k ; j<nl ; j++){
            if(buf[j][0] == '>'){
                i++;
                ll[i].start = j;
                ll[i].stop = j;
                ll[i].size = strlen( buf[j] );;
                ss[i].start = j+1;
                ss[i].size = 0;
            }else{
                ss[i].stop = j;
                ss[i].size += strlen( buf[j] );;
            }
        }

        aos = (fSeq_t *) malloc ( ns * sizeof(fSeq_t));
        if( aos == NULL){
            fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
            fflush(stderr);
            exit(0);
        }
        initAofFSeqs(aos, ns);

        for ( i=0 ; i<ns ; i++ ){
            if( ll[i].size > 0 ){
                aos[i].label = (char *) malloc ( (ll[i].size+1) * sizeof(char) );
                if( aos[i].label == NULL){
                    fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
                    fflush(stderr);
                    exit(0);
                }
                aos[i].label[0] = '\0';
                for ( j=ll[i].start ; j<=ll[i].stop ; j++ ){

                    //  both instances of strcat here are using
                    //  .label/.seq's that are NULL and that is
                    //  throwing a memory error in valgrind
                    aos[i].label = strcat ( aos[i].label, buf[j] );
                }
            }
            if( ss[i].size > 0 ){
                aos[i].seq = (char *) malloc ( (ss[i].size+1) * sizeof(char) );
                if( aos[i].seq == NULL){
                    fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
                    fflush(stderr);
                    exit(0);
                }
                aos[i].seq[0] = '\0';
                for ( j=ss[i].start ; j<=ss[i].stop ; j++ ){
                    aos[i].seq = strcat ( aos[i].seq, buf[j] );
                }
            }
        }
```

```
291     free(ll);
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;
175     char **buf = NULL;
176     fSeq_t *aos;
177
178     buf = ReadFile(INPUT, &nl);
179     if(buf == NULL){
180         return NULL;
181     }
182     aos = (fSeq_t *) malloc ( nl * sizeof(fSeq_t));
183     if( aos == NULL){
184         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
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;
```

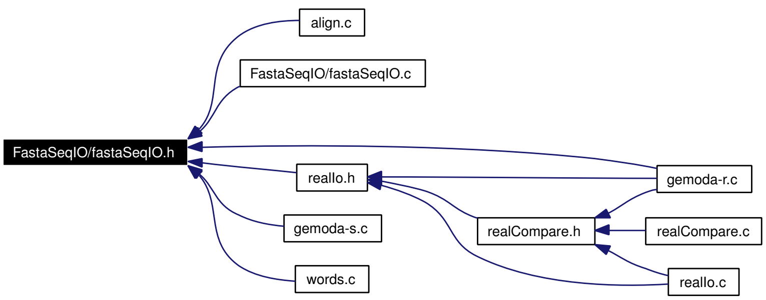
```
    for (i = start; i <= stop; i++) {
        fprintf(MY_FILE, "%s\n", arrayOfSequences[i].label);
        fprintf(MY_FILE, "%s\n", arrayOfSequences[i].seq);
    }
    return EXIT_SUCCESS;
}
```

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, sSize_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.

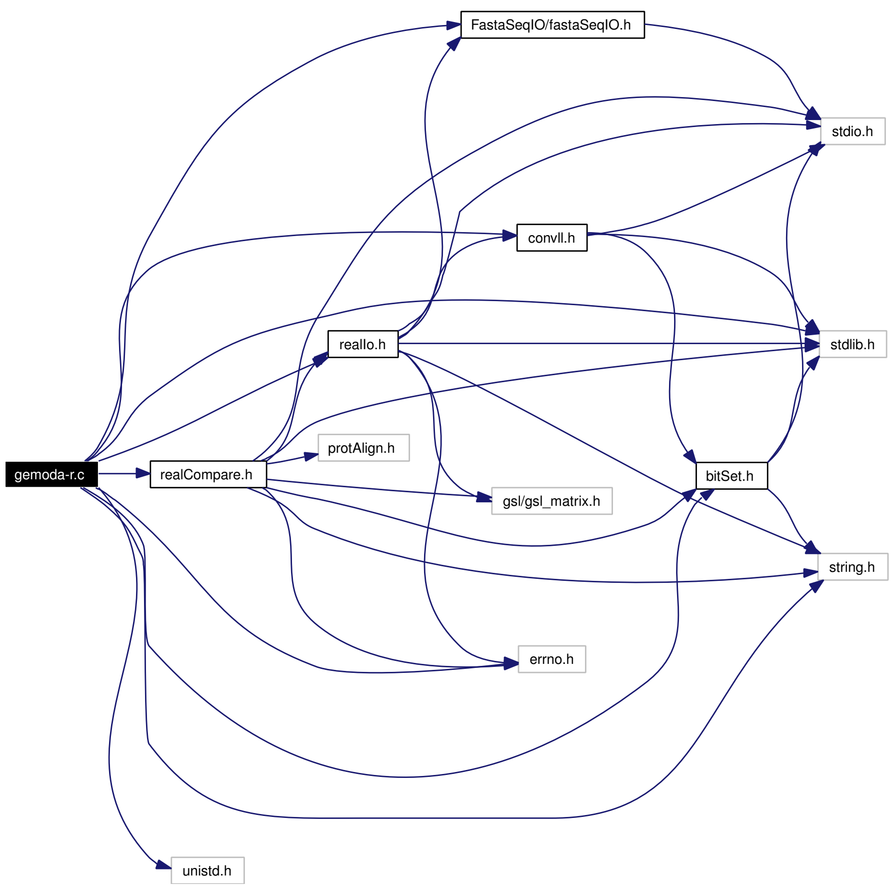
3.7.1.9 int WriteFSeqA (FILE * *MY_FILE*, fSeq_t * *arrayOfSequences*, int *start*, int *stop*)

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 **offsetToIndex, 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().

```
{
  bitSet_t * cand = NULL;
  bitSet_t * mask = NULL;
  bitSet_t * Q = NULL;
  int size = bg->size;
  cll_t * elemPats = NULL;
```

```
cll_t * allCliques = NULL;
cll_t * curr = NULL;

  // contains indices (rows) containing the threshold value.
  cand = newBitSet (size);
mask = newBitSet (size);
Q = newBitSet (size);
fillSet (cand);
fillSet (mask);

  // Note that we prune based on p before setting the diagonal false.
  if (p > 1)
  {
    bg =
  pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
  }

  // Now we set the main diagonal false for clustering and filtering.
  bitGraphSetFalseDiagonal (bg);
filterGraph (bg, support, R);
fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
fflush (NULL);
if (clusterMethod == 0)
  {
    findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
  }
else
  {
    singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
            p);
  }
fprintf (OUTPUT_FILE,
      "Clusters found!  Now filtering clusters (if option set)...\n");
fflush (NULL);
if (p > 1)
  {
    elemPats = pruneCll (elemPats, indexToSeq, p);
  }
deleteBitSet (cand);
deleteBitSet (mask);
deleteBitSet (Q);

  // Now let's convolve what we made.
  if (noConvolve == 0)
  {
    fprintf (OUTPUT_FILE, "Now convolving...\n");
    fflush (NULL);
    allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
  }

else
  {
    curr = elemPats;
    while (curr != NULL)
  {
    yankCll (&elemPats, NULL, &curr, &allCliques, 0);
  }
```

```
    }
  return allCliques;
}
```

3.8.2.2 int countExtraParams (char * *s*)

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

Referenced by main().

```
{
  int i = 0;
  int numParams = 1;
  for (i = 0; i < strlen (s); i++)
    {
      if (s[i] == ',')
    {
      numParams++;
    }
    }
  return numParams;
}
```

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(), deleteBitGraph(), freeD(), freeRdh(), getStatMat(), rdh_t::indexToSeq, rdh_t::offsetToIndex, outputRealPats(), outputRealPatsWCentroid(), parseExtraParams(), popAllCll(), readRealData(), realComparison(), bitGraph_t::size, rdh_t::size, sortByStats(), and usage().

```
{
  int inputOption = 0;
  char *sequenceFile = NULL;
  FILE *SEQUENCE_FILE = NULL;
  char *outputFile = NULL;
  FILE *OUTPUT_FILE = NULL;
  int L = 0;
  int status = 0;
  double g = 0;
```

```
int sup = 2;
int R = 1;
int P = 0;
int compFunc = 0;
double *extraParams = NULL;
int numExtraParams = 0;
int i = 0, j = 0;
/*
   int j, k, i, l;
 */
int noConvolve = 0;
int samp = 1;
int supportDim = 0, lengthDim = 0;
bitGraph_t *oam = NULL;
unsigned int **d = NULL;
int oamSize = 0;

cll_t *allCliques = NULL;
/*
   cll_t *curCliq = NULL;
 */
/*
   int curSeq;
 */
/*
   int curPos;
 */
int clusterMethod = 0;
int joelOutput = 0;

// gemoda-r new stuff
rdh_t *data = NULL;

/*
   Get command-line options
 */
while ((inputOption = getopt (argc, argv, "p:m:e:i:o:l:g:k:c:njs:")) != EOF)
  {
    switch (inputOption)
  {
    // Comparison metric
  case 'm':
    compFunc = atoi (optarg);
    break;
    // Input file
  case 'i':
    sequenceFile = optarg;
    break;
    // Output file
  case 'o':
    outputFile =
      (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
    if (outputFile == NULL)
      {
        fprintf (stderr, "Error allocating memory for options.\n");
        exit (EXIT_FAILURE);
      }
```

```
      else
        {
          strcpy (outputFile, optarg);
        }
      break;
      // Minimum motif length
    case 'l':
      L = atoi (optarg);
      break;
      // Minimum motif similarity score
    case 'g':
      g = atof (optarg);
      status++;
      break;
      // Minimum support (number of motif occurrences)
    case 'k':
      sup = atoi (optarg);
      break;

/****************************************************************
 * Recursive initial pruning: an option for clique finding.
 *   It takes all nodes with less than the minimum
 *   number of support and removes all of their nodes, and does this
 *   recursively so that nodes that are connected to many sparsely connected
 *   nodes will be removed and not left in the
 * This option is deprecated as it is at worst no-gain and at best useful.
 *   It will be on by default for clique-finding, but can be turned
 *   back off with some
 *   minor tweaking.  For almost all cases in which it does not speed
 *   up computations, it will have a trivial time to perform.  Thus, if
 *   clique-finding is turned on, then R is set to 1 by default.
        case 'r':
            R = 1;
            break;
*************************************************************************/
      // Optional pruning parameter to require at motif occurrences
      // in at least P distinct input sequences

    case 'p':
      P = atoi (optarg);
      break;

      // Clustering method.
    case 'c':
      clusterMethod = atoi (optarg);
      break;
      // Extra parameters for comparison function
    case 'e':
      numExtraParams = countExtraParams (optarg);
      extraParams = parseExtraParams (optarg, numExtraParams);
      break;
    case 'n':
      noConvolve = 1;
      break;
    case 'j':
      joelOutput = 1;
      break;
```

```
    case ’s’:
      samp = atoi (optarg);
      break;
      // Catch-all.
    case ’?’:
      fprintf (stderr, "Unknown option ‘-%c’.\n", optopt);
      usage (argv);
      return EXIT_SUCCESS;
    default:
      usage (argv);
      return EXIT_SUCCESS;
    }
    }
  // Require an input file, a nonzero length, and a similarity threshold
  // to be set.
  if (sequenceFile == NULL || L == 0 || status < 1)
    {
      usage (argv);
      return EXIT_SUCCESS;
    }
  // Open the sequence file
  if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
    {
      fprintf (stderr, "Couldn’t open file %s; %s\n", sequenceFile,
           strerror (errno));
      exit (EXIT_FAILURE);
    }
  // Open the output file
  if (outputFile != NULL)
    {
      if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
    {
      fprintf (stderr, "Couldn’t open file %s; %s\n", outputFile,
           strerror (errno));
      exit (EXIT_FAILURE);
    }
    }
  else
    {
      OUTPUT_FILE = stdout;
    }



  // Verbosity in output helps to distinguish output files.
  fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
  fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %f\n", L, sup, g);
  if (P > 1)
    {
      fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
    }
  if (R > 0)
    {
      fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
    }

  data = readRealData (SEQUENCE_FILE);
```

```
  fclose (SEQUENCE_FILE);
  // printf("size = %d,indexSize = %d\n",data->size,data->indexSize);
  // printf("size1 = %d,size2 = %d\n",data->seq[0]->size1,data->seq[0]->size2);
  // for(i = 0; i < 2; i++) {
  // for(j = 0; j < 3; j++) {
  // printf("%lf,%lf,%lf\n",gsl_matrix_get(data->seq[i],j,0),
  // gsl_matrix_get(data->seq[i],j,1),
  // gsl_matrix_get(data->seq[i],j,2));}}
  oam = realComparison (data, L, g, compFunc, extraParams);
  // printf("oam->size = %d\n", oam->size);
  if ((samp > 0) && (clusterMethod == 0))
    {
      // We are currently using one gap per sequence, as done in
      // realCompare.c's call to initRdhIndex in realComparison.
      // Note that this is data->size, NOT oam->size.
      d =
    getStatMat (oam, sup, L, &supportDim, &lengthDim, data->size, samp,
            OUTPUT_FILE);
    }
  else
    {
      d = NULL;
      supportDim = 0;
    }

  allCliques =
    convolve (oam, sup, R, data->indexToSeq, P, clusterMethod,
          data->offsetToIndex, data->size, noConvolve, OUTPUT_FILE);

  oamSize = oam->size;
  // Do some early memory cleanup since this is so big.
  deleteBitGraph (oam);

  if ((samp > 0) && (clusterMethod == 0))
    {
      cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize, data->size);
      calcStatAllCliqs (d, allCliques, oamSize - data->size);
      allCliques = sortByStats (allCliques);
    }

  if (joelOutput == 0)
    {
      outputRealPats (data, allCliques, L, OUTPUT_FILE, d);
    }
  else
    {
      outputRealPatsWCentroid (data, allCliques, L, OUTPUT_FILE, extraParams,
                   compFunc);
    }

  freeD (d, supportDim);
  freeRdh (data);
  allCliques = popAllCll (allCliques);
  fclose (OUTPUT_FILE);

  return 0;
}
```

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().

```
{
  int i = 0, j = 0, k = 0;
  int startLength = 0;
  double *extraParams = NULL;
  char *paramString = NULL;

  extraParams = (double *) malloc (numParams * sizeof (double));
  if (extraParams == NULL)
    {
      fprintf (stderr, "Can't allocate extra params!\n");
      exit (0);
    }
  j = 0;
  k = 0;
  startLength = strlen (s);
  for (i = 0; i < startLength; i++)
    {
      if (s[i] == ',')
    {
      // We've found an end.  So point the pointer to
      // the beginning of the previous string.
      paramString = &s[k];
      // Terminate the string where the comma used to be
      s[i] = '\0';
      // Update the location for the next string beginning
      k = i + 1;
      // Convert to a double and update the param number.
      extraParams[j] = atof (paramString);
      j++;
    }
    }
  // Don't forget to do the last one, which isn't comma-terminated.
  paramString = &s[k];
  extraParams[j] = atof (paramString);
  return (extraParams);
}
```

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;
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
410     // 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 = <p>.  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     // 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
439     // Find the first neighbor of this bit.
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];
```

```
    }

    // If it's the last sequence, then bail out so that we don't
    // segfault in the next step.
    if (seqNums[0] >= numOfSeqs - 1)
    {
      emptySet (bg->graph[i]);
      continue;
    }

    // Find the next neighbor of this bit, STARTING AT the first
    // bit in the next sequence.
    nextBit =
    nextBitBitSet (bg->graph[i],
               offsetToIndex[indexToSeq[nextBit] + 1][0]);

    // And iterate this until we run out of neighbors.
    while (nextBit >= 0)
    {
      j++;
      seqNums[j] = indexToSeq[nextBit];

        // Or until this new neighbor will fill up the array
        if (j == p - 1)
        {
          break;
        }

        // Or until this new neighbor is in the last sequence.
        if (seqNums[j] >= numOfSeqs - 1)
        {
          break;
        }

        // Get the next neighbor!
        nextBit =
        nextBitBitSet (bg->graph[i],
               offsetToIndex[indexToSeq[nextBit] + 1][0]);
    }

    // If we didn't have enough unique sequences, and either a) we
    // were in the nth-to-last sequence and there were no
    // neighbors after it, or b) we were in the last sequence,
    // then the last number will still be our sentinel, -1.  If
    // the last number is not a sentinel, then we have at least
    // p distinct sequence occurrences, so we're OK.
    if (seqNums[p - 1] == -1)
    {
      emptySet (bg->graph[i]);
    }
    }
  free (seqNums);
  return (bg);
}
```

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.

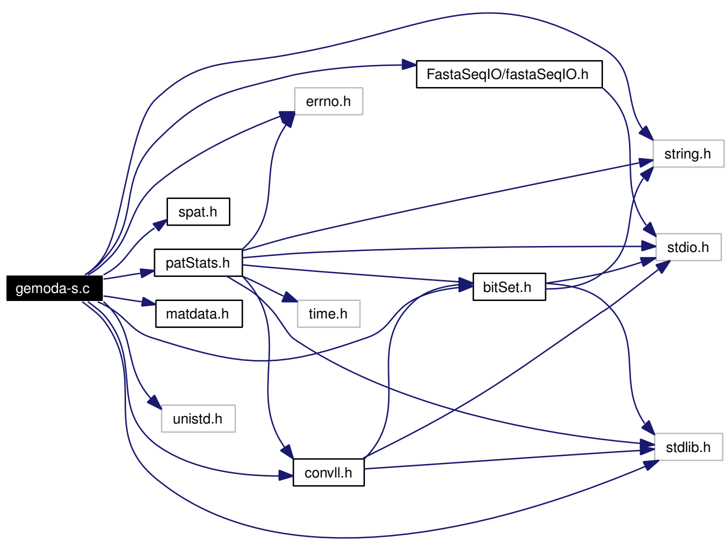
Referenced by main().

```
{
  fprintf (stdout,
       "Usage: %s -i <Fasta sequence file> "
       "-l <word size> \n\t-k <support> -g <threshold> -m <matrix name> [-z] \n\t[-c <cluster method [0
       "Required flags and input:\n\n"
       "-i <Fasta sequence file>:\n\t"
       "File containing all sequences to be searched, in Fasta format.\n\n"
       "-l <word size>:\n\t"
       "Minimum length of motifs; also the sliding window length\n\t"
       "over which all motifs must meet the similarity criterion\n\n"
       "-k <support>:\n\t"
       "Minimum number of motif occurrences.\n\n"
       "-g <threshold>:\n\t"
       "Similarity threshold.  Two windows, when scored with the\n\t"
       " similarity matrix defined by the -m flag, must have at least\n\t"
       " this score in order to be deemed 'connected'.  This criterion\n\t"
       " must be met over all sliding windows of length l.\n\n"
       "-c <cluster method [0|1]>:\n\t"
       "The clustering method to be used after evaluating the "
       "\n\tsimilarity of the unique words in the input.  Note that the "
       "\n\tclustering method will have a significant impact on both the "
       "\n\tresults that one obtains and the computation time.\n\n\t"
       "0: clique-finding\n\t\t"
       "Uses established methods to find all maximal cliques in the "
       "\n\t\tdata.  This will give the most thorough results (that are "
       "\n\t\tprovably exhaustive), but will also give less-significant "
       "\n\t\tresults in addition to the most interesting and most\n\t"
       "significant ones.  The results are deterministic but may take some "
       "\n\t\ttime on data sets with high similarity or if the similarity "
       "\n\t\tthreshold is set extremely low.\n\t"
       "1: single-linkage clustering\n\t\t"
       "Uses a single-linkage-type clustering where all nodes that "
       "\n\t\tare connected are put in the same cluster.  This method is "
       "\n\t\talso deterministic and will be faster than clique-finding, "
       "\n\t\tbut it loses guarantees of exhaustiveness in searching the "
       "\n\t\tdata set.\n\n",
       "-p <unique support>:\n\t"
       "A pruning parameter that requires the motif to occur in "
       "\n\tat least <unique support> different input sequences.  Note "
       "\n\tthat this parameter must be less than or equal to the total "
       "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
  fprintf (stdout, "\n");
}
```

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().

```
{
  bitGraph_t * sg = NULL;
  int score;
  int i, j;

```

```
    // Assign a new bitGraph_t object, with (wc x wc) possible
    // true/false values
    sg = newBitGraph (wc);
  for (i = 0; i < wc; i++)
    {
      for (j = i; j < wc; j++)
     {

        // Get the score for the alignment of word i and word j
        score =
        alignMat (words[i].string, words[j].string, words[i].length, mat);

        // If that score is greater than threshold, set
        // a bit to 'true' in our bitGraph_t object
        if (score >= threshold)
        {

        // We use 'bitGraphSetTrueSym' because, if i=j,
        // then j=i for most applications.  However, this
        // can be relaxed for masochists.
        bitGraphSetTrueSym (sg, i, j);
        }
     }
     }

     // Return a pointer to this new bitGraph_t object
     return sg;
}
```

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(), singleLinkage(), bitGraph_t::size, and yankCll().

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

```
bitSet_t * mask = NULL;
bitSet_t * Q = NULL;
int size = bg->size;
cll_t * elemPats = NULL;
cll_t * allCliques = NULL;
cll_t * curr = NULL;

  // contains indices (rows) containing the threshold value.
  cand = newBitSet (size);
mask = newBitSet (size);
Q = newBitSet (size);
fillSet (cand);
fillSet (mask);

  // Note that we prune based on p before setting the diagonal false.
  if (p > 1)
  {
    bg =
  pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
  }

  // Now we set the main diagonal false for clustering and filtering.
  bitGraphSetFalseDiagonal (bg);
filterGraph (bg, support, R);
fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
fflush (NULL);
if (clusterMethod == 0)
  {
    findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
  }
else
  {
    singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
            p);
  }
fprintf (OUTPUT_FILE,
      "Clusters found!  Now filtering clusters (if option set)...\n");
fflush (NULL);
if (p > 1)
  {
    elemPats = pruneCll (elemPats, indexToSeq, p);
  }
deleteBitSet (cand);
deleteBitSet (mask);
deleteBitSet (Q);

  // Now let’s convolve what we made.
  if (noConvolve == 0)
  {
    fprintf (OUTPUT_FILE, "Now convolving...\n");
    fflush (NULL);
    allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
  }

else
  {
    curr = elemPats;
```

```
      while (curr != NULL)
    {
      yankCll (&elemPats, NULL, &curr, &allCliques, 0);
    }
    }
  return allCliques;
}
```

3.9.2.3 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(), sHashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, sOffset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

Referenced by main().

```
{
  int i, j;
  int totalChars = 0;
  int hashSize;
  sHashEntry_t newEntry;
  sHashEntry_t *ep;
  sHash_t wordHash;
  sPat_t *words = NULL;
  int wc = 0;
  int prev = -1;
  int l;


  // Count the total number of characters.  This
  // is the upper limit on how many words we can have
  for (i = 0; i < numSeq; i++)
    {
      totalChars += strlen (seq[i].seq);
    }

  // Get a prime number for the size of the hash table
  hashSize = sieve3 ((long) (2 * totalChars));
  wordHash = initSHash (hashSize);

  // Chop up each sequence and hash out the words of size L
  for (i = 0; i < numSeq; i++)
    {
      prev = -1;

      // skip sequences that are too short to have
      // a pattern
      if (strlen (seq[i].seq) < L)
```

```
    {
      continue;
    }
      for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
    {

      // Make a hash table entry for this word
      newEntry.key = &(seq[i].seq[j]);
      newEntry.data = 1;
      newEntry.idx = wc;
      newEntry.L = L;

      // Check to see if it’s already in the hash table
      ep = searchSHash (&newEntry, &wordHash, 0);
      if (ep == NULL)
        {

          // If it’s not, create an entry for it
          ep = searchSHash (&newEntry, &wordHash, 1);

          // Increase the size of our word array
          words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
          if (words == NULL)
        {
          fprintf (stderr, "Error!\n");
          fflush (stderr);
        }
          // Add the new word
          words[wc].string = &(seq[i].seq[j]);
          words[wc].length = L;
          words[wc].support = 1;
          words[wc].offset =
        (sOffset_t *) malloc (1 * sizeof (sOffset_t));
          if (words[wc].offset == NULL)
        {
          fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
          fflush (stderr);
          exit (0);
        }
          words[wc].offset[0].seq = i;
          words[wc].offset[0].pos = j;
          words[wc].offset[0].prev = prev;
          words[wc].offset[0].next = -1;

          if (prev != -1)
        {
          words[prev].offset[words[prev].support - 1].next = wc;
        }
          prev = wc;
          wc++;

        }
      else
        {

          // If it is, increase the count for this word
          ep->data++;
```

```

          // add a new offset to the word array
          l = words[ep->idx].support;
          words[ep->idx].offset =
        (sOffset_t *) realloc (words[ep->idx].offset,
                     (l + 1) * sizeof (sOffset_t));
          words[ep->idx].offset[l].seq = i;
          words[ep->idx].offset[l].pos = j;
          words[ep->idx].offset[l].prev = prev;
          words[ep->idx].offset[l].next = -1;

          // Update the next/prev
          if (prev != -1)
        {
          words[prev].offset[words[prev].support - 1].next = ep->idx;
        }
          prev = ep->idx;

          // Have to put this down here for cases when we create
          // a word and it is immeadiately followed by itself!!
          words[ep->idx].support += 1;
        }
    }
    }


  destroySHash (&wordHash);
  *numWords = wc;
  return words;
}
```

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(), calcStatAllCliqs(), convolve(), countWords2(), cumDMatrix(), deleteBitGraph(), FreeFSeqs(), getMatrixByName(), getStatMat(), cnode::length, mat, MATRIX_SIZE, matrixlist(), cSet_t::members, newBitGraph(), cnode::next, sPat_t::offset, popAllCll(), sOffset_t::pos, ReadFSeqs(), sOffset_t::seq, cnode::set, cSet_t::size, bitGraph_t::size, sortByStats(), cnode::stat, and usage().

```
{
  int inputOption = 0;
  char *sequenceFile = NULL;
  char *outputFile = NULL;
  char *matName = NULL;
  FILE * SEQUENCE_FILE = NULL;
  FILE * OUTPUT_FILE = NULL;
  int L = 0;
  int numberOfSequences = 0;
  fSeq_t * mySequences = NULL;
  fSeq_t * (*seqReadFunct) () = &ReadFSeqs;
  sPat_t * words = NULL;
  int wc;
  int status = 0;
  int g = 0;
  int sup = 2;
  int R = 1;
  int P = 0;
  int (*mat)[MATRIX_SIZE] = NULL;
  int noConvolve = 0;
  int j, k, i, l;
  bitGraph_t * bg = NULL;
  bitGraph_t * oam = NULL;

    // new
  int **offsetToIndex = NULL;
  int *indexToSeq = NULL;
  int *indexToPos = NULL;
  int numberOfOffsets = 0;
  int pos1, pos2;

    // int *prevRowArray;
    sOffset_t * offset1, *offset2;
  cll_t * allCliques = NULL;
  cll_t * curCliq = NULL;
  int curSeq;
  int curPos;
  int clusterMethod = 0;

    // patStats
  int samp = 1;
  unsigned int **d = NULL;
  int supportDim = 0, lengthDim = 0;
  int oamSize = 0;

    /*
        Get command-line options
     */
    while ((inputOption = getopt (argc, argv, "i:o:l:g:k:m:p:zc:ns:")) != EOF)
```

```
    {
      switch (inputOption)
    {

        // Input file
    case 'i':
      sequenceFile = optarg;
      seqReadFunct = &ReadFSeqs;
      break;

        // Output file
    case 'o':
      outputFile =
        (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
      if (outputFile == NULL)
        {
          fprintf (stderr, "Error allocating memory for options.\n");
          exit (EXIT_FAILURE);
        }
      else
        {
          strcpy (outputFile, optarg);
        }
      break;

        // Minimum motif length
    case 'l':
      L = atoi (optarg);
      break;

        // Minimum motif similarity score
    case 'g':
      g = atoi (optarg);
      status++;
      break;

        // Minimum support (number of motif occurrences)
    case 'k':
      sup = atoi (optarg);
      break;

        // Similarity matrix used to find similarity score
    case 'm':
      getMatrixByName (optarg, &mat);
      matName = (char *) malloc (strlen (optarg) * sizeof (char));
      if (matName == NULL)
        {
          fprintf (stderr, "Error allocating memory for options.\n");
          exit (EXIT_FAILURE);
        }
      else
        {
          strcpy (matName, optarg);
        }
      break;

/****************************************************************
```

```
 * Recursive initial pruning: an option for clique finding.
 *   It takes all nodes with less than the minimum
 *   number of support and removes all of their nodes, and does this
 *   recursively so that nodes that are connected to many sparsely connected
 *   nodes will be removed and not left in the
 * This option is deprecated as it is at worst no-gain and at best useful.
 *   It will be on by default for clique-finding, but can be turned
 *   back off with some
 *   minor tweaking.  For almost all cases in which it does not speed
 *   up computations, it will have a trivial time to perform.  Thus, if
 *   clique-finding is turned on, then R is set to 1 by default.
        case 'r':
            R = 1;
            break;
*************************************************************************/
        // Optional pruning parameter to require at motif occurrences
        // in at least P distinct input sequences
    case 'p':
      P = atoi (optarg);
      break;

        // Clustering method.
    case 'c':
      clusterMethod = atoi (optarg);
      break;
    case 'n':
      noConvolve = 1;
      break;
    case 's':
      samp = atoi (optarg);
      break;

        // Catch-all.
    case '?':
      fprintf (stderr, "Unknown option `-%c'.\n", optopt);
      usage (argv);
      return EXIT_SUCCESS;
    case 'z':
      matrixlist ();
      return EXIT_SUCCESS;
    default:
      usage (argv);
      return EXIT_SUCCESS;
    }
    }

    // Require a similarity matrix
    if (mat == NULL)
    {
      usage (argv);
      return EXIT_SUCCESS;
    }

    // Require an input file, a nonzero length, and a similarity threshold
    // to be set.
    if (sequenceFile == NULL || L == 0 || status < 1)
    {
```

```
      usage (argv);
      return EXIT_SUCCESS;
    }

    // Open the sequence file
    if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
    {
      fprintf (stderr, "Couldn’t open file %s; %s\n", sequenceFile,
        strerror (errno));
      exit (EXIT_FAILURE);
    }

    // Open the output file
    if (outputFile != NULL)
    {
      if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
    {
      fprintf (stderr, "Couldn’t open file %s; %s\n", outputFile,
            strerror (errno));
      exit (EXIT_FAILURE);
    }
    }
  else
    {
      OUTPUT_FILE = stdout;
    }

    // Allocate some sequences
    mySequences = seqReadFunct (SEQUENCE_FILE, &numberOfSequences);
  if (mySequences == NULL)
    {
      fprintf (stderr, "\nError reading your sequences/text.");
      fprintf (stderr, "\nCheck the format/size of the file.");
      fprintf (stderr, "\nERROR:  %s\n", strerror (errno));
      return EXIT_FAILURE;
    }

    // Close the input files
    fclose (SEQUENCE_FILE);

    // Verbosity in output helps to distinguish output files.
    fprintf (OUTPUT_FILE, "\nMatrix used = %s\n", matName);
  fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
  fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %d\n", L, sup, g);
  if (P > 1)
    {
      fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
    }
  if (R > 0)
    {
      fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
    }

    // Find the unique words in the input.
    words = countWords2 (mySequences, numberOfSequences, L, &wc);

    /*
```

```
      fprintf(stderr, "Counted %d words\n", wc);
     */
    /*
      fflush(stderr);
     */

    // Align the words that we just found by applying the similarity
    // matrix to each pair of them.  Note that
    // bg is the adjacency matrix of words, but we
    // need an adjacency matrix of offsets instead.
    bg = alignWordsMat_bit (words, wc, mat, g);
  fprintf (OUTPUT_FILE, "\nAligned!  Creating offset matrix...\n");
  fflush (NULL);

    // Create an intermediate translation matrix
    // to store the offset number of each sequence number/position.
    //
    // Note that this matrix is better called "Index to offset", and
    // the other matrices are better called "offset to Seq" and
    // "offset to Pos"
    offsetToIndex = (int **) malloc (numberOfSequences * sizeof (int *));
  if (offsetToIndex == NULL)
    {
      fprintf (stderr,
        "Unable to allocate memory - offsetToIndex in gemoda.c\n%s\n",
        strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < numberOfSequences; i++)
    {

    // MPS 5/23/05: Added in "-L+2" to make there only be one
    // blank between sequences.
    offsetToIndex[i] =
    malloc ((strlen (mySequences[i].seq) - L + 2) * sizeof (int));
      if (offsetToIndex[i] == NULL)
    {
      fprintf (stderr,
            "Unable to allocate memory - offsetToIndex[%d] in gemoda.c\n%s\n",
            i, strerror (errno));
      fflush (stderr);
      exit (0);
    }

    // MPS 5/23/05: Added in "-L+2" to make there only be one
    // blank between sequences.
    for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
    {
      offsetToIndex[i][j] = numberOfOffsets;
      numberOfOffsets++;
    }
    }

    // Now create translation matrices such that we can get the sequence
    // or position number of a given offset.
    indexToSeq = (int *) malloc (numberOfOffsets * sizeof (int));
```

```
if (indexToSeq == NULL)
  {
    fprintf (stderr,
      "Unable to allocate memory - indexToSeq in gemoda.c\n%s\n",
      strerror (errno));
    fflush (stderr);
    exit (0);
  }
indexToPos = (int *) malloc (numberOfOffsets * sizeof (int));
if (indexToPos == NULL)
  {
    fprintf (stderr,
      "Unable to allocate memory - indexToPos in gemoda.c\n%s\n",
      strerror (errno));
    fflush (stderr);
    exit (0);
  }
k = 0;
for (i = 0; i < numberOfSequences; i++)
  {

  // MPS 5/23/05: Added in "-L+2" to make there only be one
  // blank between sequences.
  for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
  {
    indexToSeq[k] = i;
    indexToPos[k] = j;
    k++;
  }
  }

  // Now make an offset adjacency matrix!
  //
  oam = newBitGraph (numberOfOffsets);

  // Go through each unique word
  for (i = 0; i < wc; i++)
  {
    offset1 = words[i].offset;

  // Go through each occurrence
  for (k = 0; k < words[i].support; k++)
  {

      // Use the offsetToIndex translation to get the offset
      // of the first occurrence
      pos1 = offsetToIndex[offset1[k].seq][offset1[k].pos];

      // And go through each word in the first offset to
      // find words that meet the similarity threshold
      for (j = 0; j < wc; j++)
      {
        if (bitGraphCheckBit (bg, i, j))
      {
        offset2 = words[j].offset;

          // And find all of their occurrences,
```

```
            // using offsetToIndex to get the
            // offsets, and then setting those
            // locations in the offset adjacency
            // matrix true.
            for (l = 0; l < words[j].support; l++)
            {
              pos2 = offsetToIndex[offset2[l].seq][offset2[l].pos];
              bitGraphSetTrueSym (oam, pos1, pos2);
            }
        }
        }
    }
    }
  fprintf (OUTPUT_FILE, "Offset matrix created...");
  deleteBitGraph (bg);
  if ((samp > 0) && (clusterMethod == 0))
    {
      fprintf (OUTPUT_FILE, " taking preliminary statistics.\n");
      fflush (NULL);
      d =
    getStatMat (oam, sup, L, &supportDim, &lengthDim, numberOfSequences,
            samp, OUTPUT_FILE);
      fprintf (OUTPUT_FILE, "Now filtering...\n");
      fflush (NULL);
    }
  else
    {
      fprintf (OUTPUT_FILE, " now filtering.\n");
      fflush (NULL);
      d = NULL;
      supportDim = 0;
    }

    // Now we're convolving on offsets
    allCliques =
    convolve (oam, sup, R, indexToSeq, P, clusterMethod, offsetToIndex,
          numberOfSequences, noConvolve, OUTPUT_FILE);

    // Do some early memory cleanup to limit usage
    oamSize = oam->size;
  deleteBitGraph (oam);
  fprintf (OUTPUT_FILE, "Convolved!  Now making output...\n");
  fflush (NULL);
  if ((samp > 0) && (clusterMethod == 0))
    {
      cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize,
           numberOfSequences);
      calcStatAllCliqs (d, allCliques, numberOfOffsets - numberOfSequences);
      allCliques = sortByStats (allCliques);
    }

    // walk over the cliques and give some output in the format:
    // pattern <pattern id num>: len=<motif length> sup=<motif instances>
    // <sequence num> <position num> <motif instance>
    // ...
    curCliq = allCliques;

```

```
    i = 0;
  while (curCliq != NULL)
    {
      fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d", i,
        curCliq->length + L, curCliq->set->size);
      if (d != NULL)
    {
      fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
    }
      else
    {
      fprintf (OUTPUT_FILE, "\n");
    }

    for (j = 0; j < curCliq->set->size; j++)
    {
      pos1 = curCliq->set->members[j];
      curSeq = indexToSeq[pos1];
      curPos = indexToPos[pos1];
      fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);
      for (k = curPos; k < curPos + curCliq->length + L; k++)
        {
          fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]);
        }
      fprintf (OUTPUT_FILE, "\n");
    }
      fprintf (OUTPUT_FILE, "\n\n");
      curCliq = curCliq->next;
      i++;
    }

    // And do some memory cleanup
    // And cleanup of probability stuff...
    /*
        free(letterfreqs); delete_augmented_matrix(augmat);
     */
    allCliques = popAllCll (allCliques);
  free (indexToSeq);
  indexToSeq = NULL;
  free (indexToPos);
  indexToPos = NULL;
  for (i = 0; i < numberOfSequences; i++)
    {
      free (offsetToIndex[i]);
      offsetToIndex[i] = NULL;
    }

    // Free'ing added by MPS, 6/4
    for (i = 0; i < wc; i++)
    {
      free (words[i].offset);
    }
  free (words);

    // End free'ing added by MPS
    free (offsetToIndex);
  offsetToIndex = NULL;
```

```

    // -----------------------------------------

    // Free up fastaSequences
    FreeFSeqs (mySequences, numberOfSequences);
  fclose (OUTPUT_FILE);
  return 0;
}
```

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().

```
{
  fprintf (stdout, "\nThe following similarity matrices are installed "
        "with the default Gemoda installation.\n  Most of these "
        "were obtained from publically available BLAST distributions. \n\n"
         "dna_idmat:\n\t"
        "Identity matrix for DNA: returns 1 when A,C,G,T are "
        "compared to \n\tthemselves, 0 otherwise.\n\n"
        "identity_aa:\n\t"
        "Identity matrix for amino acids: returns 1 when any \n\t"
        "letter but J,O,U are compared to themselves, and 0 "
        "otherwise.\n\n"  "idmat:\n\t"
        "Similar to identity_aa, but it returns 10 in place "
        "of 1.\n\n"  "est_idmat:\n\t"
        "Similar to idmat, but it returns -10 in place of 0. "  "\n\n"
        "pam100:\n"  "pam110:\n"  "pam120:\n"  "pam130:\n"
        "pam140:\n"  "pam150:\n"  "pam160:\n"  "pam190:\n"
        "pam200:\n"  "pam210:\n"  "pam220:\n"  "pam230:\n"
        "pam240:\n"  "pam250:\n"  "pam260:\n"  "pam280:\n"
        "pam290:\n"  "pam300:\n"  "pam310:\n"  "pam320:\n"
        "pam330:\n"  "pam340:\n"  "pam360:\n"  "pam370:\n"
        "pam380:\n"  "pam390:\n"  "pam400:\n"  "pam430:\n"
        "pam440:\n"  "pam450:\n"  "pam460:\n"  "pam490:\n"
        "pam500:\n\t"
        "PAM matrices for various evolutionary distances.\n\n"
        "blosum30:\n"  "blosum35:\n"  "blosum40:\n"  "blosum45:\n"
        "blosum50:\n"  "blosum55:\n"  "blosum60:\n"  "blosum62:\n"
        "blosum65:\n"  "blosum70:\n"  "blosum75:\n"  "blosum80:\n"
        "blosum85:\n"  "blosum90:\n"  "blosum100:\n\t"
        "BLOSUM matrices for various evolutionary distances.\n\n"
        "blosumn:\n\t"  "BLOSUM matrix of unknown origin.\n\n"
        "dayhoff:\n\t"
        "'Vanilla-flavored' pam250, very similar to pam250.\n\n"
```

```
      "phat_t75_b73:\n"  "phat_t80_b78:\n"  "phat_t85_b82:\n\t"
      "BLOSUM-clustered scoring matrix with target frequency\n\t"
      "PHDhtm clustering = {75,80,85}percent and background frequency\n\t"
       "Persson-Argos clustering = {73,78,82}percent.\n\t"
      "From Ng, Henikoff, & Henikoff, Bioinformatics 16: 760.\n\n"
      "coil_mat:\n"  "alpha_mat:\n"  "beta_mat:\n\t"
      "Three structure-specific matrices described by Luthy,\n\t"
      "McLachlan, and Eisenberg in Proteins 10, 229-239, obtained from AAindex.\n\n");
  fprintf (stdout, "\n");
}
```

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().

```
{
  int i = 0, j = 0, nextBit = 0;
  int *seqNums = NULL;

    // Since we don't immediately know which node is in which source
    // sequence, we can't just count them up regularly.  Instead, we'll
    // need to keep track of which sequences they come from and
    // increment _something_.  What we chose to do here is just make
    // an array of integers of length = <p>.  Then, we try to put the
    // source sequence number of each neighbor (including itself, since
    // the main diagonal is still true at this time) into the next slot
    // Since we will monotonically search the bitSet, we can just
    // move on to the first bit in the next sequence using the
    // offsetToIndex structure so that we know the next sequence number
    // to be put in is always unique.
    seqNums = (int *) malloc (p * sizeof (int));
  if (seqNums == NULL)
    {
      fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
        strerror (errno));
      fflush (stderr);
      exit (0);
    }

    // So, for each row in the bitgraph...
```

```
for (i = 0; i < bg->size; i++)
{

// Make sure the whole array is -1 sentinels.
for (j = 0; j < p; j++)
{
  seqNums[j] = -1;
}
  j = 0;

// Find the first neighbor of this bit.
nextBit = nextBitBitSet (bg->graph[i], 0);
  if (nextBit == -1)
{
  continue;
}
  else
{

    // and put its sequence number in the array of ints.
    seqNums[0] = indexToSeq[nextBit];
}

// If it's the last sequence, then bail out so that we don't
// segfault in the next step.
if (seqNums[0] >= numOfSeqs - 1)
{
  emptySet (bg->graph[i]);
  continue;
}

// Find the next neighbor of this bit, STARTING AT the first
// bit in the next sequence.
nextBit =
nextBitBitSet (bg->graph[i],
           offsetToIndex[indexToSeq[nextBit] + 1][0]);

// And iterate this until we run out of neighbors.
while (nextBit >= 0)
{
  j++;
  seqNums[j] = indexToSeq[nextBit];

    // Or until this new neighbor will fill up the array
    if (j == p - 1)
    {
      break;
    }

    // Or until this new neighbor is in the last sequence.
    if (seqNums[j] >= numOfSeqs - 1)
    {
      break;
    }

    // Get the next neighbor!
    nextBit =
```

```
        nextBitBitSet (bg->graph[i],
               offsetToIndex[indexToSeq[nextBit] + 1][0]);
    }

    // If we didn't have enough unique sequences, and either a) we
    // were in the nth-to-last sequence and there were no
    // neighbors after it, or b) we were in the last sequence,
    // then the last number will still be our sentinel, -1.  If
    // the last number is not a sentinel, then we have at least
    // p distinct sequence occurrences, so we're OK.
    if (seqNums[p - 1] == -1)
    {
      emptySet (bg->graph[i]);
    }
    }
  free (seqNums);
  return (bg);
}
```

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.

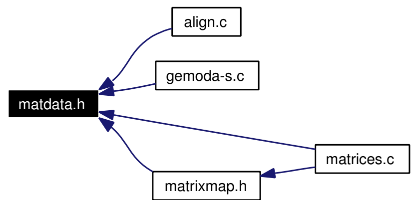
Definition at line 32 of file gemoda-s.c.

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

```
        "\n\tsimilarity of the unique words in the input.  Note that the "
        "\n\tclustering method will have a significant impact on both the "
         "\n\tresults that one obtains and the computation time.\n\n\t"
        "0: clique-finding\n\t\t"
        "Uses established methods to find all maximal cliques in the "
        "\n\t\tdata.  This will give the most thorough results (that are "
        "\n\t\tprovably exhaustive), but will also give less-significant "
         "\n\t\tresults in addition to the most interesting and most\n\t"
        "significant ones.  The results are deterministic but may take some "
        "\n\t\ttime on data sets with high similarity or if the similarity "
         "\n\t\tthreshold is set extremely low.\n\t"
        "1: single-linkage clustering\n\t\t"
        "Uses a single-linkage-type clustering where all nodes that "
        "\n\t\tare connected are put in the same cluster.  This method is "
        "\n\t\talso deterministic and will be faster than clique-finding, "
        "\n\t\tbut it loses guarantees of exhaustiveness in searching the "
         "\n\t\tdata set.\n\n"  "-p <unique support>:\n\t"
        "A pruning parameter that requires the motif to occur in "
        "\n\tat least <unique support> different input sequences.  Note "
        "\n\tthat this parameter must be less than or equal to the total "
         "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
  fprintf (stdout, "\n");
}
```

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

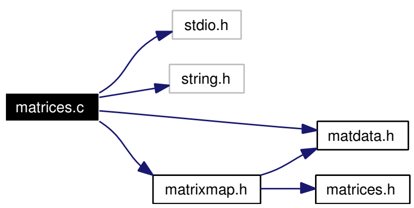
Definition at line 10 of file matdata.h.

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

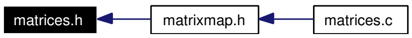
Definition at line 34 of file matrices.c.

References DEFAULT_MATRIX, and matrix_map.

```
{
  int i;
  for (i = 0; matrix_map[i].name != NULL; i++)
    {
      if (strcmp (name, matrix_map[i].name) == 0)
    {
      break;
    }
    }
  if (matrix_map[i].name != NULL)
    {
      *matp = (matrix_map[i].mat);
    }
  else
    {
      *matp = (DEFAULT_MATRIX);
    }
}
```

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]
- const int pam150 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam160 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam190 [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.

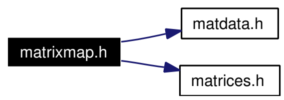
3.12.2.60 const int phat_t85_b82[MATRIX_SIZE][MATRIX_SIZE]

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().

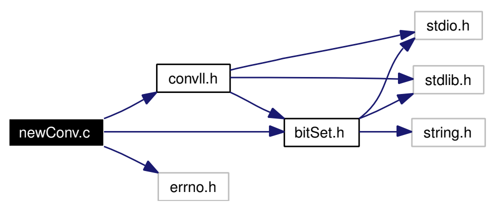
3.13.2.3 char* name

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 ∗∗offsetToIndex, 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(), singleLinkage(), bitGraph_t::size, and yankCll().

```
{
  bitSet_t * cand = NULL;
  bitSet_t * mask = NULL;
  bitSet_t * Q = NULL;
  int size = bg->size;
  cll_t * elemPats = NULL;
  cll_t * allCliques = NULL;
  cll_t * curr = NULL;

    // contains indices (rows) containing the threshold value.
    cand = newBitSet (size);
  mask = newBitSet (size);
  Q = newBitSet (size);
  fillSet (cand);
  fillSet (mask);

    // Note that we prune based on p before setting the diagonal false.
    if (p > 1)
    {
      bg =
    pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
    }

    // Now we set the main diagonal false for clustering and filtering.
    bitGraphSetFalseDiagonal (bg);
  filterGraph (bg, support, R);
  fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
  fflush (NULL);
  if (clusterMethod == 0)
    {
      findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
```

```
    }
  else
    {
      singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
              p);
    }
  fprintf (OUTPUT_FILE,
        "Clusters found!  Now filtering clusters (if option set)...\n");
  fflush (NULL);
  if (p > 1)
    {
      elemPats = pruneCll (elemPats, indexToSeq, p);
    }
  deleteBitSet (cand);
  deleteBitSet (mask);
  deleteBitSet (Q);

    // Now let's convolve what we made.
    if (noConvolve == 0)
    {
      fprintf (OUTPUT_FILE, "Now convolving...\n");
      fflush (NULL);
      allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
    }

  else
    {
      curr = elemPats;
      while (curr != NULL)
    {
      yankCll (&elemPats, NULL, &curr, &allCliques, 0);
    }
    }
  return allCliques;
}
```

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().

```
{
  bitSet_t * changed = newBitSet (graph->size);
  bitSet_t * work = newBitSet (graph->size);
  emptySet (changed);
  emptySet (work);

    // Iteratively call the filtering by copying the previous "work" into
    // "changed" after each iteration step.
    if (R == 1)
    {

      do
    {
      filterIter (graph, support, changed, work);
      copySet (work, changed);
    }
      while (countSet (changed) > 0);
    }
  else
    {

    // Otherwise, just do it once.
    filterIter (graph, support, changed, work);
    }
  deleteBitSet (changed);
  deleteBitSet (work);
  return 0;
}
```

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().

```
{
  int i = 0, j = 0;
  int lastBit = 0, nextBit = 0, lastRow = 0, nextRow = 0;
  int numNodes = 0;
  int changedSize = countSet (changed);
  emptySet (work);

    // Note the convention that the first time the function is called,
    // it is done with an empty "changed" bitSet as a sentinel.  It is
    // the responsibility of the master function calling the iterator
    // to catch future empty changed sets to know that convergence has
    // been achieved.
    //
    // So, if it's your first time through, go through each node and make
    // sure that each is connected to at least <support> - 1 others.
    if (changedSize == 0)
    {
      for (i = 0; i < graph->size; i++)
    {
      numNodes = countSet (graph->graph[i]);
      if (numNodes >= support - 1)
        {
          continue;
        }
      else
        {

        // Otherwise, zero it out, but going one by
        // one so that you can also zero out the
        // symmetric bit.
        lastBit = 0;
          for (j = 0; j < numNodes; j++)
        {
          nextBit = nextBitBitSet (graph->graph[i], lastBit);
          if (nextBit == -1)
            {
              fprintf (stderr,
                "\nEnd of bitSet reached! - initial\n");
              fflush (stderr);
              exit (0);
            }
          setFalse (graph->graph[i], nextBit);
          setFalse (graph->graph[nextBit], i);

            // And set that corresponding bit true
            // in the work bitSet so that we
            // know we changed it for the next
            // round.
            setTrue (work, nextBit);
          lastBit = nextBit + 1;
        }
        }
    }
```

```
    }
  else
    {

    // Otherwise, we've been here before, so just follow what
    // the changed bitSet says to do... only those bitSets that
    // were changed could possibly have gone under the minimum
    // support requirement.
    lastRow = 0;
      for (i = 0; i < changedSize; i++)
    {
      nextRow = nextBitBitSet (changed, lastRow);
      if (nextRow == -1)
        {
          fprintf (stderr, "\nEnd of bitSet reached! - iter,row\n");
          fflush (stderr);
          exit (0);
        }

        // So now we've found the row that needs to be checked.
        // We do the same thing we did above... either move
        // on if it has enough possible support, or zero
        // it out (with its symmetric locations) one by one.
        numNodes = countSet (graph->graph[nextRow]);
      if (numNodes >= support - 1)
        {
          lastRow = nextRow + 1;
          continue;
        }
      else
        {
          lastBit = 0;
          for (j = 0; j < numNodes; j++)
        {
          nextBit = nextBitBitSet (graph->graph[nextRow], lastBit);
          if (nextBit == -1)
            {
              fprintf (stderr,
                "\nEnd of BitSet reached! = iter,Bit\n");
              fflush (stderr);
              exit (0);
            }
          setFalse (graph->graph[nextRow], nextBit);
          setFalse (graph->graph[nextBit], nextRow);
          setTrue (work, nextBit);
          lastBit = nextBit + 1;
        }
          lastRow = nextRow + 1;
        }
    }
    }
  return 1;
}
```

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 bit-Graph_t::size.

Referenced by convolve().

```
{
  bitSet_t ** gammaOG = NULL;
  bitSet_t * candQ = newBitSet (oG->size);
  bitSet_t * newMask = newBitSet (oG->size);
  int i, q;
  int graphSize;
  int max = -1;
  int numBits;
  int u = 0;
  int newMaskCount;
  int candQCount;
  graphSize = oG->size;

    //
    // Find which vertex in subg maximizes |cand intersect gamma(u) |
    gammaOG = oG->graph;
  for (i = 0; i < graphSize; i++)
    {

    // Don't check this vertex if it's masked
    if (!(checkBit (mask, i)))
    {
      continue;
    }

    // cand is always a subset of mask, so intersecting
```

```
  // with mask is redundant
  bitSetIntersection (gammaOG[i], cand, candQ);
    numBits = countSet (candQ);
    if (numBits > max)
  {
    u = i;
    max = numBits;
  }
  }

  // Then do the extension of the q's
  qCount++;

  // This loop iterates over all possible values of cand - gamma() by
  // iterating over all possible values of cand but immediately
  // "continue"ing if the node is also in gamma(u)
  q = nextBitBitSet (cand, 0);
while (q != -1)
  {
    if (checkBit (gammaOG[u], q))
  {
    q = nextBitBitSet (cand, q + 1);
    continue;
  }

  // SUBGq = SUBG i Gamma
  bitSetIntersection (mask, gammaOG[q], newMask);
    newMaskCount = countSet (newMask);
    setTrue (Q, q);

  // Only recurse if there are more candidates to be included,
  // and they will allow us to reach the minimum support.
  if (newMaskCount > 0 && qCount + newMaskCount >= support)
  {

       // CANDq = CAND i Gamma
       bitSetIntersection (gammaOG[q], cand, candQ);
     candQCount = countSet (candQ);

       // only recurse if we can possibly get to a clique
       // of size with minimum support
       if (candQCount > 0 && qCount + candQCount >= support)
       {

       // recursion with
       // new candidates, new mask, and original graph
       findCliques (Q, candQ, newMask, oG, support, qCount, elemPats,
                indexToSeq, p);
       }
   }
     else if (qCount >= support)
   {

       // This should be done when:
       // 1. countSet(newMask) == 0 [connected subgraph is maximal]
       // 2. Qcount >= minCount [connected subgraph has enough nodes]
       *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);
133   return 0;
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 
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
411     // increment _something_.  What we chose to do here is just make
412     // an array of integers of length = <p>.  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     }
```

```

// So, for each row in the bitgraph...
for (i = 0; i < bg->size; i++)
{

// Make sure the whole array is -1 sentinels.
for (j = 0; j < p; j++)
{
  seqNums[j] = -1;
}
  j = 0;

// Find the first neighbor of this bit.
nextBit = nextBitBitSet (bg->graph[i], 0);
  if (nextBit == -1)
{
  continue;
}
  else
{

    // and put its sequence number in the array of ints.
    seqNums[0] = indexToSeq[nextBit];
}

// If it’s the last sequence, then bail out so that we don’t
// segfault in the next step.
if (seqNums[0] >= numOfSeqs - 1)
{
  emptySet (bg->graph[i]);
  continue;
}

// Find the next neighbor of this bit, STARTING AT the first
// bit in the next sequence.
nextBit =
nextBitBitSet (bg->graph[i],
           offsetToIndex[indexToSeq[nextBit] + 1][0]);

// And iterate this until we run out of neighbors.
while (nextBit >= 0)
{
  j++;
  seqNums[j] = indexToSeq[nextBit];

    // Or until this new neighbor will fill up the array
    if (j == p - 1)
    {
      break;
    }

    // Or until this new neighbor is in the last sequence.
    if (seqNums[j] >= numOfSeqs - 1)
    {
      break;
    }

```

```
      // Get the next neighbor!
      nextBit =
      nextBitBitSet (bg->graph[i],
             offsetToIndex[indexToSeq[nextBit] + 1][0]);
    }

    // If we didn't have enough unique sequences, and either a) we
    // were in the nth-to-last sequence and there were no
    // neighbors after it, or b) we were in the last sequence,
    // then the last number will still be our sentinel, -1.  If
    // the last number is not a sentinel, then we have at least
    // p distinct sequence occurrences, so we're OK.
    if (seqNums[p - 1] == -1)
    {
      emptySet (bg->graph[i]);
    }
    }
  free (seqNums);
  return (bg);
}
```

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().

```
{
  int i = 0, j = 0, thisSeq = 0;
  int *seqNums = NULL;
  cll_t * curr = head;
  cll_t * prev = NULL;
  cll_t * storage = NULL;

    // We'll do this similar to the pruneBitGraph function... we will
    // keep track of which source sequence each motif occurrence was in.
    // Again, since the occurrences are listed monotonically, we only
    // need to compare the last non-sentinel index to the current
    // sequence number.
    seqNums = (int *) malloc (p * sizeof (int));
  if (seqNums == NULL)
    {
      fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
```

```
    }
  while (curr != NULL)
    {

    // First make sure the set size is at least p.
    // This is redundant, but extremely simple and not expensive,
    // so we’ll leave it in just as a check.
    if (curr->set->size < p)
    {
      if (prev != NULL)
        {
          prev->next = curr->next;
        }
      else
        {
          head = curr->next;
        }
      storage = curr->next;
      free (curr->set->members);
      free (curr->set);
      free (curr);
      curr = storage;
      continue;
    }
      for (i = 0; i < p; i++)
    {
      seqNums[i] = -1;
    }
      j = 0;
      seqNums[0] = indexToSeq[curr->set->members[0]];

    // Note, we’ve checked to make sure size > p, and we know
    // p must be 2 or greater, so we can start at 1 without
    // worrying about segfaulting
    for (i = 1; i < curr->set->size; i++)
    {
      thisSeq = indexToSeq[curr->set->members[i]];
      if (thisSeq != seqNums[j])
        {
          j++;
          seqNums[j] = thisSeq;
          if (j == p - 1)
        {
          break;
        }
        }
    }

    // Same story as before... if the last number is -1,
    // then we didn’t have enough to fill up the <p> different
    // slots, so this doesn’t meet our criterion.
    if (seqNums[p - 1] == -1)
    {
      if (prev != NULL)
        {
          prev->next = curr->next;
        }
```

```
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 }
```

3.14.2.7 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, nextBitBitSet(), pushClique(), and setFalse().

Referenced by convolve().

```
157 {
158   int i = 0;
159   int j = 0;
160
161     // go to the first vertex that has not been clustered yet
162     i = nextBitBitSet (cand, 0);
163   if (i != -1)
164     {
```

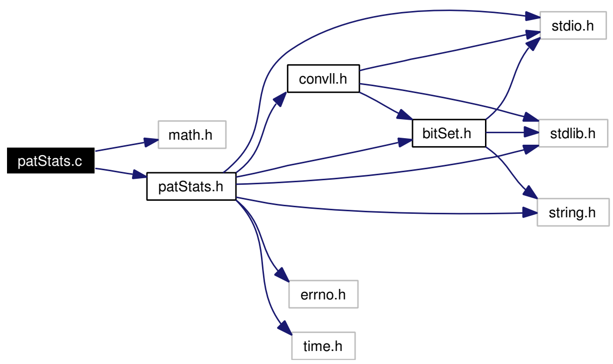
```

    // this vertex has been clustered
    setFalse (cand, i);

    // start a new cluster, Q
    copySet (oG->graph[i], Q);

    // go over each vertex in the cluster
    j = nextBitBitSet (Q, 0);
      while (j != -1)
    {

        // if this vertex has been clustered already, skip it and go
        // to the next one
        if (!checkBit (cand, j))
        {
          j = nextBitBitSet (Q, j + 1);
          continue;
        }

        // Add this vertex's neighbors to the current cluster
        bitSetUnion (Q, oG->graph[j], Q);

        // This vertex has now been clustered
        setFalse (cand, j);

        // go over each vertex in the cluster
        j = nextBitBitSet (Q, 0);
    }

    // Did we make a cluster that was large enough?
    if (countSet (Q) >= support)
    {
      *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
    }

    // recurse
    singleLinkage (Q, cand, mask, oG, support, 0, elemPats, indexToSeq,
                p);
    }
  else
    {
      return 0;
    }
  return 0;
}
```

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 currSupport, 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 ∗supportDim, 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().

```
{
  cll_t * curr = NULL;
  curr = allCliqs;
  while (curr != NULL)
    {
      curr->stat = calcStatCliq (d, curr, numWindows);
      curr = curr->next;
    }
  return (0);
}
```

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().

```
{
  double stat = 0;
  int i = 0;
  int supChooseTwo = 0;
  double interimP = 0;
  int support = cliq->set->size;
  int length = cliq->length;
  double numTrials = 0;
```

```
  if (support < 2)
    {
      fprintf (stderr, "Support for cluster less than 2... exiting.\n");
      fflush (stderr);
      exit (0);
    }

    // OK, so support is at least two.  So we make the connections all
    // on the first level, knowing that each node being connected has
    // at least zero in common.  There are [(size of cluster) - 1] of
    // these connections to be made.
    // And we know we can call for d[0][1] because if the second index
    // were out of bounds, then there would be no similarities, and
    // there would be no reason to call this function.
    interimP = ((double) d[0][1]) / ((double) d[0][0]);
  stat = pow (interimP, support - 1);
  stat *= ((double) numWindows * (numWindows - 1)) / ((double) 2);

    // Now we actually calculate the probability... the first connection
    // has to be made no matter what, and after that we multiply for
    // every connection after the first one.  So we descend iteratively
    // until we have made all connections, terminating after we've made
    // the single i = (n - 2) connection.  There is no i = (n - 1)
    // connection.
    for (i = 1; i < support - 1; i++)
    {
      interimP = ((double) d[i][1]) / ((double) d[i][0]);
      stat *= pow (interimP, support - i - 1);
      stat *= ((double) (numWindows - (i + 1))) / ((double) (i + 2));
    } supChooseTwo = (support * (support - 1)) / 2;

    // Remember that length = (numwindows - 1), or alternatively,
    // the number of extensions... normally we'd want to have the last
    // p be p[support][numwindows - 1], which corresponds to
    // alteredD[support][numwindows]/alteredD[support][numwindows-1],
    // so that means we want our last d to be d[support][numwindows].
    // Here, we note that the calculation of p's would be continuously
    // re-normalizing, so multiplying all p's is the same as dividing
    // the last d by the initial d.
    interimP = ((double) d[support][length + 1]) / ((double) d[support][1]);
  stat *= pow (interimP, supChooseTwo);
  return stat;
}
```

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().

Referenced by main().

```
{
```

```
  int maxSup = 0;
  int maxLen = 0;
  int i, j;
  int numWins = 0;

    maxSup = getLargestSupport (cliqs);
  maxLen = getLargestLength (cliqs);

/********* 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,
    //  the value of d decreases because only unique streaks are
    //  counted.
    // We also note that the number of disjoint node-pairs with a given
    //  number of other nodes in common is defined by d[support][0].
    // So, in order to properly account for all "unique" comparisons
    //  (which is equal to (# streaks + # disjoint node-pairs), we must
    //  add d[support][1] to d[support][0].

    for (i = 0; i < currSupport + 1; i++) {
        d[i][0] += d[i][1];
    }
    ********************/

    // We no longer need to do that, since now we sum across both
    // the support and the length dimensions.  Now, d[support][0] will
    // necessarily include d[support][1] being added to it.  We don't
    // want to add this anymore, otherwise we would be underestimating
    // the probability of making that first connection.  For instance,
    // if there were no nodes with 20 in common that weren't also
    // connected, and no nodes whatsoever with more than 20 in common,
    // we'd want the p[20][0] to be 1, which would be
    // d[20][1]/d[20][0].  When summing across length directions,
    // this happens naturally, whereas before we needed to do it
    // artificially as per above.  If we did above, we'd have the
    // probability of each node being 1/2 instead of 1.

    // Rather than storing doubles and doing lots of multiplications,
    // we're going to limit the number of operations done in the actual
    // probability calculation by only storing cumulative sums in d.
    // Now remember, what we're storing at each location is the
    // number of nodes with [i] or more nodes in common (including
    // each other and selves) that can be extended [j] times (with
    // their initial similarity counting as 1).
    //
    // We go up to the last possible index in the length direction, which
    // means going up to [maxLen].  We know that this is legitimate
    // because maxLen is less than or equal to the longest possible
    // diagonal, and the longest possible diagonal will be less
    // than or equal to currLength.  Since we have allotted
    // (currLength + 1) integers, we know we're OK to access [currLength].
    for (j = 0; j < currLength + 1; j++)
    {

    // We start at currSupport - 1, because currSupport will
    // clearly not be changed, and this makes it a much easier
    // 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     // 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.
602     // So it's (numWins - 1)*(numWins - 1 + 1)/2 to sum that up.
603     numWins = bgSize - numSeqs;
604   d[0][0] = numWins * (numWins - 1) / 2;
605 
606     /*
607         for (i = 0; i <= maxSup; i++) { printf("support = %d:\t",i); for (j = 0; j <=
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.

Referenced by main().

```
689 {
690   int i = 0;
691   if (d == 0)
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.
700     for (i = 0; i < supportDim + 1; i++)
701     {
702       free (d[i]);
703     }
704       free (d);
705       return 0;
```

```
    }
}
```

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().

```
{
  int len = 0;
  cll_t * curCliq = NULL;
  curCliq = cliqs;
  while (curCliq != NULL)
    {
      if (curCliq->length > len)
    {
      len = curCliq->length;
    }
      curCliq = curCliq->next;
    }

    // We return (len + 1) because the length of the shortest streak
    // is one, but is stored in the cluster data structure as being
    // zero (number of extensions that have been made).
    return (len + 1);
}
```

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().

```
{
  int size = 0;
  cll_t * curCliq = NULL;
  curCliq = cliqs;
  while (curCliq != NULL)
    {
      if (curCliq->set->size > size)
```

```
    {
      size = curCliq->set->size;
    }
      curCliq = curCliq->next;
    }
  return 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.

Referenced by main().

```
{
  int *Q = NULL;
  unsigned int **d = NULL;
  int i, j, k;
  int x, y;
  bitSet_t * X = NULL;
  int currSupport;
  int currLength;
  int multiplier = 50;
  int diagonal = 0;
  time_t probStart, probEnd;
  int timeNeeded = 0;
  int sampleCounter = 1;

    // int visitCounter = 0, uniqCounter = 0;
    currSupport = support * multiplier;
  currLength = length * multiplier;
  X = newBitSet (bg->size);

    // printf("Made bitSet of size %d\n", bg->size);
    Q = (int *) malloc (bg->size * sizeof (int));
  if (Q == NULL)
    {
      fprintf (stderr,
        "\nMemory error --- couldn't allocate array!"  "\n%s\n",
        strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < bg->size; i++)
    {
      Q[i] = 0;
    }
```

```
d =
  (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
if (d == NULL)
  {
    fprintf (stderr,
      "\nMemory error --- couldn’t allocate array!"  "\n%s\n",
      strerror (errno));
    fflush (stderr);
    exit (0);
  }
for (i = 0; i < currSupport + 1; i++)
  {
    d[i] =
  (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
    if (d[i] == NULL)
  {
    fprintf (stderr, "\nMemory error --- couldn’t allocate array!"
          "\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
  }
    for (j = 0; j < currLength + 1; j++)
  {
    d[i][j] = 0;
  }
  }

  // printf("size=%d\n",bg->size);
  time (&probStart);
for (i = 0; i < bg->size; i++)
  {
    if (i == 200)
  {
    time (&probEnd);
    timeNeeded = ((double) (probEnd - probStart)) /
      ((double) 60) * ((double) bg->size) / ((double) 200);
    if (timeNeeded > 2)
      {
        fprintf (OUTPUT_FILE,
          "Max total time to calculate probability:\n");
        fprintf (OUTPUT_FILE, "\t%d minutes\n", timeNeeded);
        fprintf (OUTPUT_FILE, "Actual time will be less than this, "
          "but at least half of it.\n");
        fprintf (OUTPUT_FILE,
          "To bypass excessive probability calculations,"
          " cancel and use a different value\n"
          " for the ’-s’ flag (samples every "
          "’s’ points).\n");
        fflush (NULL);
      }
  }
    j = nextBitBitSet (bg->graph[i], 0);
    while (j >= 0)
  {
    k = nextBitBitSet (bg->graph[i], j + 1);
    while (k >= 0)
      {
```

```
          if (checkBit (bg->graph[j], k) == 0)
        {
          if (sampleCounter == s)
            {
              bitGraphRowIntersection (bg, j, k, X);

            // visitCounter++;
            if (nextBitBitSet (X, 0) >= i)
            {

                // uniqCounter++;
                x = countSet (X);
              while (x > currSupport)
                {
                  d =
                increaseMem (d, 1, currSupport, currLength,
                         currSupport +
                         support * multiplier);
                  currSupport += support * multiplier;
                }
              d[x][0] += 1;
            }
              sampleCounter = 0;
            }
          sampleCounter++;
        }
          k = nextBitBitSet (bg->graph[i], k + 1);
        }
      if (j <= i)
        {
          j = nextBitBitSet (bg->graph[i], j + 1);
          continue;
        }
      bitGraphRowIntersection (bg, i, j, X);
      x = countSet (X);

        // Note, now we're using "diagonals" rather than
        // location in a horizontal array.  So you always
        // start from the main diagonal at 0 and move out.
        diagonal = j - i;

        // We change this to greater-than-one because
        // after Q[diagonal] is reduced to one, it isn't
        // visited again until we reach a new streak, (because
        // the next bit in the diagonal is a zero), and at
        // that point we want to start with a new diagonal
        // measure.
        if (Q[diagonal] > 1)
        {
          y = Q[diagonal] - 1;
          Q[diagonal]--;
        }
      else
        {
          y = measureDiagonal (bg, i, j);
          Q[diagonal] = y;
        }
```

```
      while (x > currSupport)
        {
          d = increaseMem (d, 1, currSupport, currLength,
                currSupport + support * multiplier);
          currSupport += support * multiplier;
        }
      while (y > currLength)
        {
          d =
        increaseMem (d, 2, currSupport, currLength,
                 currLength + length * multiplier);
          currLength += length * multiplier;
        }
      d[x][y]++;
      j = nextBitBitSet (bg->graph[i], j + 1);

        /*
            if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
         */
    }

    /*
        printf("done\n"); fflush(stdout);
     */
    }

    // We need to rescale by the sampling factor for all i>0 in d[i][0].
    //
    for (i = 1; i < currSupport; i++)
    {
      d[i][0] *= s;
    }

    // Now we only need to assign the correct value for d[0][0]...
    // but rather than figuring that out, we will just assign it in the
    // cumulative function, since there it is merely the number of unique
    // non-self comparisons and is easy to calculate.
    deleteBitSet (X);
  free (Q);
  *supportDim = currSupport;
  *lengthDim = currLength;
  return (d);
}
```

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().

```
{
  int i = 0, j = 0;
  if (dimToChange == 1)
    {
      d =
    (unsigned int **) realloc (d, (newVal + 1) * sizeof (unsigned int *));
      if (d == NULL)
    {
      fprintf (stderr, "\nMemory error --- couldn’t allocate array!"
            "\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
      for (i = currSupport + 1; i < newVal + 1; i++)
    {
      d[i] =
        (unsigned int *) malloc ((currLength + 1) *
                     sizeof (unsigned int));
      if (d[i] == NULL)
        {
          fprintf (stderr,
            "\nMemory error --- couldn’t allocate array!"
            "\n%s\n", strerror (errno));
          fflush (stderr);
          exit (0);
        }
      for (j = 0; j < currLength + 1; j++)
        {
          d[i][j] = 0;
        }
    }
      return d;
    }
  else if (dimToChange == 2)
    {
      for (i = 0; i < currSupport + 1; i++)
    {
      d[i] =
        (unsigned int *) realloc (d[i],
                      (newVal + 1) * sizeof (unsigned int));
      if (d[i] == NULL)
        {
          fprintf (stderr,
            "\nMemory error --- couldn’t allocate array!"
            "\n%s\n", strerror (errno));
          fflush (stderr);
          exit (0);
        }
      for (j = currLength + 1; j < newVal + 1; j++)
        {
          d[i][j] = 0;
        }
    }
      return d;
    }
```

```
  else
    {
      fprintf (stderr, "Invalid arguments to increaseMem!\n\n");
      fflush (stderr);
      exit (0);
    }
}
```

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().

```
{
  int len = 0;
  while (bitGraphCheckBit (bg, i + len, j + len) != 0)
    {
      len++;
    }
  return len;
}
```

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∼20 and K∼20, the memory used shoots down to ∼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.

```
{
  int *Q = NULL;
  unsigned int **d = NULL;
  int i, j;
  int x, y;
  bitSet_t * X = NULL;
  int currSupport;
  int currLength;
  int multiplier = 50;
  time_t probStart, probEnd;
  int timeNeeded = 0;
  currSupport = support * multiplier;
  currLength = length * multiplier;
  X = newBitSet (bg->size);

    // printf("Made bitSet of size %d\n", bg->size);
    Q = (int *) malloc (bg->size * sizeof (int));
  if (Q == NULL)
    {
      fprintf (stderr,
        "\nMemory error --- couldn't allocate array!"  "\n%s\n",
        strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < bg->size; i++)
    {
      Q[i] = 0;
    }
  d =
    (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
  if (d == NULL)
    {
      fprintf (stderr,
        "\nMemory error --- couldn't allocate array!"  "\n%s\n",
        strerror (errno));
      fflush (stderr);
      exit (0);
```

```
    }
  for (i = 0; i < currSupport + 1; i++)
    {
      d[i] =
    (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
      if (d[i] == NULL)
    {
      fprintf (stderr, "\nMemory error --- couldn’t allocate array!"
            "\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
      for (j = 0; j < currLength + 1; j++)
    {
      d[i][j] = 0;
    }
    }
  time (&probStart);
  for (i = 0; i < bg->size; i++)
    {
      if (i == 200)
    {
      time (&probEnd);
      timeNeeded = ((double) (probEnd - probStart)) /
        ((double) 60) * ((double) bg->size) / ((double) 200);
      if (timeNeeded > 2)
        {
          printf ("Max total time to calculate probability:\n");
          printf ("\t%d minutes\n", timeNeeded);
          printf ("Actual time will be less than this, but at",
               "least half of it.\n");
          printf ("To bypass excessive probability calculations,",
               "cancel and use the ’-d’ flag.\n");
          fflush (NULL);
        }
    }
      for (j = bg->size - 1; j > i; j--)
    {
      bitGraphRowIntersection (bg, i, j, X);
      x = countSet (X);
      if (Q[j - 1] != 0)
        {
          y = Q[j - 1] - 1;
          Q[j] = Q[j - 1] - 1;
        }
      else
        {
          y = measureDiagonal (bg, i, j);
          Q[j] = y;
        }
      while (x > currSupport)
        {
          d = increaseMem (d, 1, currSupport, currLength,
                currSupport + support * multiplier);
          currSupport += support * multiplier;
        }
      while (y > currLength)
```

```
          {
            d =
          increaseMem (d, 2, currSupport, currLength,
                   currLength + length * multiplier);
            currLength += length * multiplier;
          }
        d[x][y]++;

          /*
              if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
           */
      }

      /*
          printf("done\n"); fflush(stdout);
       */
      }

      // We know that the "blanks", inserted to delimit unique sequences
      // and prevent convolution through them, will skew our statistics,
      // so we subtract them.  We know that they will never be similar to
      // any others, so will only add to the d[0][0] number.  Furthermore,
      // we know how many they add.  Since d never hits the main diagonal
      // and only does the upper half of the matrix, the first one
      // contributes bgsize - 1 to d[0][0], the next bgsize - 2, etc.
      for (i = 0; i < numBlanks; i++)
      {
        d[0][0] -= bg->size - 1 - i;
      }
    deleteBitSet (X);
    free (Q);
    *supportDim = currSupport;
    *lengthDim = currLength;
    return (d);
  }
```

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().

Referenced by main().

```
  {
    cll_t * curCliq = NULL;
    cll_t ** arrayOfCliqs = NULL;
    int numOfCliqs = 0;
    int i = 0;
    curCliq = allCliqs;
    if (curCliq != NULL)
```

```
    {
      numOfCliqs = curCliq->id + 1;
    }
  else
    {
      return (NULL);
    }
  arrayOfCliqs = (cll_t **) malloc (numOfCliqs * sizeof (cll_t *));
  for (i = 0; i < numOfCliqs; i++)
    {
      arrayOfCliqs[i] = curCliq;
      curCliq = curCliq->next;
    }
  qsort (arrayOfCliqs, numOfCliqs, sizeof (cll_t *), statCompare);
  for (i = 0; i < numOfCliqs - 1; i++)
    {
      arrayOfCliqs[i]->next = arrayOfCliqs[i + 1];
    }
  arrayOfCliqs[numOfCliqs - 1]->next = NULL;
  return (arrayOfCliqs[0]);
}
```

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

Definition at line 709 of file patStats.c.

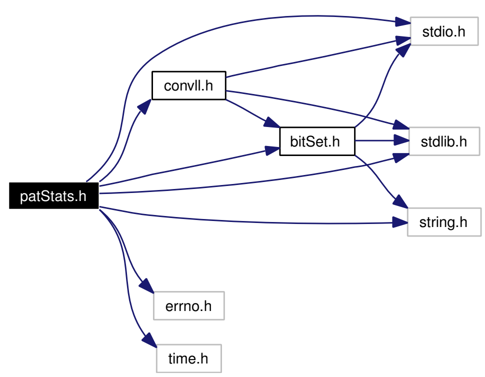
Referenced by sortByStats().

```
{
  double difference = (*first)->stat - (*second)->stat;
  if (difference < 0)
    {
      return (-1);
    }
  else if (difference > 0)
    {
      return (1);
    }
  else
    {
      return (0);
    }
}
```

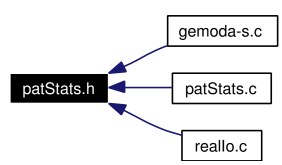
3.16 patStats.h File Reference

```
#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 ∗supportDim, 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.

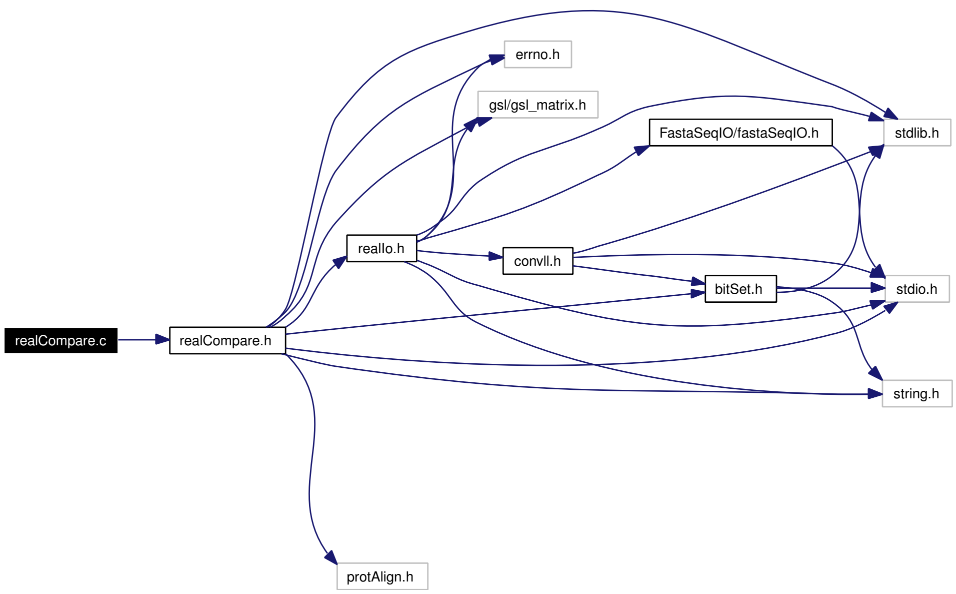
References cnode::id.

Referenced by main().

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 ∗extraParams)
- double generalMatchFactor (rdh_t ∗data, int win1, int win2, int L, double ∗extraParams)
- double massSpecCompareWElut (rdh_t ∗data, int win1, int win2, int L, double ∗extraParams)
- double(∗)(rdh_t ∗, int, 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 comparison 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     /*
118        double denominator=0.0;
119      */
120   double xsum;
121   double ysum;
122   double ldenom = 0.0;
123   double rdenom = 0.0;
124   int dim;
125   int seq1, pos1;
126   int seq2, pos2;
127   gsl_matrix_view view1;
128   gsl_matrix_view view2;
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
135     getRdhIndexSeqPos (data, win1, &seq1, &pos1);
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;
```

```
  mat2 = &view2.matrix;

    // Loop over each position
    for (i = 0; i < mat1->size1; i++)
    {
      xsum = 0.0;
      ysum = 0.0;

    // Loop over each dimension at each position
    for (j = 0; j < dim; j++)
    {
      xsum += gsl_matrix_get (mat1, i, j);
      ysum += gsl_matrix_get (mat2, i, j);
    }
      numerator += (i + 1) * sqrt (xsum * ysum);
      ldenom += (i + 1) * xsum;
      rdenom += (i + 1) * ysum;
    }
  return pow (numerator, 2.0) / (ldenom * rdenom);
}
```

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

Definition at line 264 of file realCompare.c.

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

```
{
  double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
  switch (compFunc)
    {
    case 0:
      comparisonFunc = &rmsdCompare;
      break;
    case 1:
      comparisonFunc = &generalMatchFactor;
      break;
    case 2:
      comparisonFunc = &massSpecCompareWElut;
      break;
    default:
      comparisonFunc = &rmsdCompare;
      break;
    }
  return (comparisonFunc);
}
```

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().

```
{
  int i, j;
  double numerator = 0.0;

    /*
       double denominator=0.0;
     */
  double xsum;
  double ysum;
  double cum;
  double ldenom = 0.0;
  double rdenom = 0.0;
  int dim;
  int seq1, pos1;
  int seq2, pos2;
  double weight = 2.0;
  gsl_matrix_view view1;
  gsl_matrix_view view2;
  gsl_matrix * mat1;
  gsl_matrix * mat2;
  double maxElut = -1;
  if (extraParams != NULL)
    {
      maxElut = extraParams[0];
    }
  dim = getRdhDim (data);

    // Find out which seq,pos pairs these two
    // windows correspond to
    getRdhIndexSeqPos (data, win1, &seq1, &pos1);
  getRdhIndexSeqPos (data, win2, &seq2, &pos2);

    // Get a reference to a submatrix.  That is,
    // '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);

    // Some error checking here would be nice!
    // Did we get the matrices we wanted?

    // This just makes it easier to handle the views
    mat1 = &view1.matrix;
  mat2 = &view2.matrix;
  cum = 1.0;

    // Loop over each position
    for (i = 0; i < mat1->size1; i++)
    {
```

```
      xsum = 0.0;
      ysum = 0.0;

    // First take the first dimension for elution time
    if (maxElut >= 0)
    {
      if (fabs
           (gsl_matrix_get (mat1, i, 0) - gsl_matrix_get (mat2, i, 0)) >
           maxElut)
        {
          cum = 0;
          break;
        }
    }

    // printf("\n");
    //
    // Loop over each subsequent dimension at each position
    for (j = 1; j < dim; j++)
    {

        // printf("mat1val=%lf,mat2val=%lf\n",gsl_matrix_get(mat1,i,j),
        // gsl_matrix_get(mat2,i,j));
        numerator += pow (j, weight) * sqrt (gsl_matrix_get (mat1, i, j)
                          *gsl_matrix_get (mat2, i,
                                   j));
      ldenom += pow (j, weight) * gsl_matrix_get (mat1, i, j);
      rdenom += pow (j, weight) * gsl_matrix_get (mat2, i, j);

        // printf("numer=%lf,ldenom=%lf,rdenom=%lf\n",numerator,
        // ldenom,rdenom);
    }
      cum *= pow (numerator, 2.0) / (ldenom * rdenom);
    }
  return pow (cum, 1.0 / L);
}
```

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().

Referenced by main().

```
{
  int i, j;
  int seq1, pos1;
  int seq2, pos2;
  bitGraph_t * bg = NULL;
  double score;
  double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
```

```

    // Initialize the rdh's index
    initRdhIndex (data, L, 1);

    // Allocate a new bit graph
    bg = newBitGraph (data->indexSize);

    // Choose the comparison function, pass a reference to it
    comparisonFunc = getCompFunc (compFunc);
  for (i = 0; i < data->indexSize; i++)
    {

    // Skip seperators
    getRdhIndexSeqPos (data, i, &seq1, &pos1);
      if (seq1 == -1 || pos1 == -1)
    {
      continue;
    }
      for (j = i; j < data->indexSize; j++)
    {
      getRdhIndexSeqPos (data, j, &seq2, &pos2);
      if (seq2 == -1 || pos2 == -1)
        {
          continue;
        }

        // This is the comparison function
        score = comparisonFunc (data, i, j, L, extraParams);

        // printf("score (%2d,%2d) vs. (%2d, %2d) =\t%lf\n",seq1, pos1, seq2, pos2,
        // score);
        if (compFunc == 0)
        {
          if (score <= g)
        {
          bitGraphSetTrueSym (bg, i, j);
        }
        }
      else if ((compFunc == 1) || (compFunc == 2))
        {
          if (score >= g)
        {
          bitGraphSetTrueSym (bg, i, j);
        }
        }
      else
        {
          fprintf (stderr, "Comparison function undefined in "
            "realComparison function,\n located in "
            "realCompare.c.  Exiting.\n\n");
          fflush (stderr);
          exit (0);
        }
    }
    }
  return bg;
}
```

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().

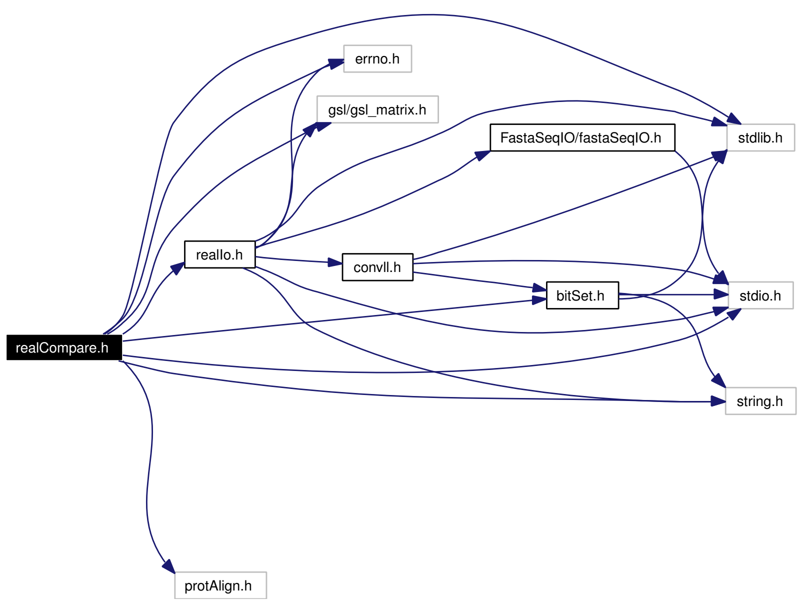
```
32 {
33   int trans = 1;
34   int rot = 1;
35   int dim;
36   double result = 0;
37   int seq1, pos1;
38   int seq2, pos2;
39   gsl_matrix_view view1;
40   gsl_matrix_view view2;
41   gsl_matrix * mat1;
42   gsl_matrix * mat2;
43   gsl_matrix * mat1copy;
44   gsl_matrix * mat2copy;
45
46     // The "rint" function is in math.h and rounds a number to the
47     // nearest integer.  It raises an "inexact exception" if the
48     // number initially wasn't an integer.
49     if (extraParams != NULL)
50     {
51       trans = rint (extraParams[0]);
52       rot = rint (extraParams[1]);
53     }
54   dim = getRdhDim (data);
55
56     // Find out which seq,pos pairs these two
57     // windows correspond to
58     getRdhIndexSeqPos (data, win1, &seq1, &pos1);
59   getRdhIndexSeqPos (data, win2, &seq2, &pos2);
60
61     // Get a reference to a submatrix.  That is,
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);

  // This just makes it easier to handle the views
  mat1 = &view1.matrix;
mat2 = &view2.matrix;

  // Create copies of the windows, because our comparison
  // will require altering the matrices
  mat1copy = gsl_matrix_alloc (mat1->size1, mat1->size2);
mat2copy = gsl_matrix_alloc (mat2->size1, mat2->size2);
gsl_matrix_memcpy (mat1copy, mat1);
gsl_matrix_memcpy (mat2copy, mat2);

  /*
      printf("matrix1:\n"); gsl_matrix_pretty_fprintf(stdout, mat1copy, "%f ");
     printf("\nmatrix2:\n"); gsl_matrix_pretty_fprintf(stdout, mat2copy, "%f ");
   */

  // Are we going to do a translation?
  if (trans == 1)
  {
    moveToCentroid (mat1copy);
    moveToCentroid (mat2copy);
  }

  // Are we going to do a rotation?
  if (rot == 1)
  {

  // Rotate mat2copy to have a minimal
  // rmsd with mat1copy
  rotateMats (mat1copy, mat2copy);
  }

  // Compute the rmsd between mat2copy and mat2copy
  result = gsl_matrix_rmsd (mat1copy, mat2copy);
gsl_matrix_free (mat1copy);
gsl_matrix_free (mat2copy);
return result;
}
```

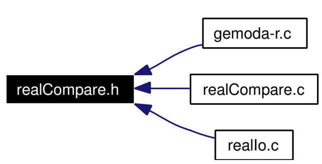
3.18 realCompare.h File Reference

```
#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 ∗extraParams)
- double generalMatchFactor (rdh_t ∗data, int win1, int win2, int L, double ∗extraParams)
- 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, 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, int, double∗) getCompFunc(int compFunc)

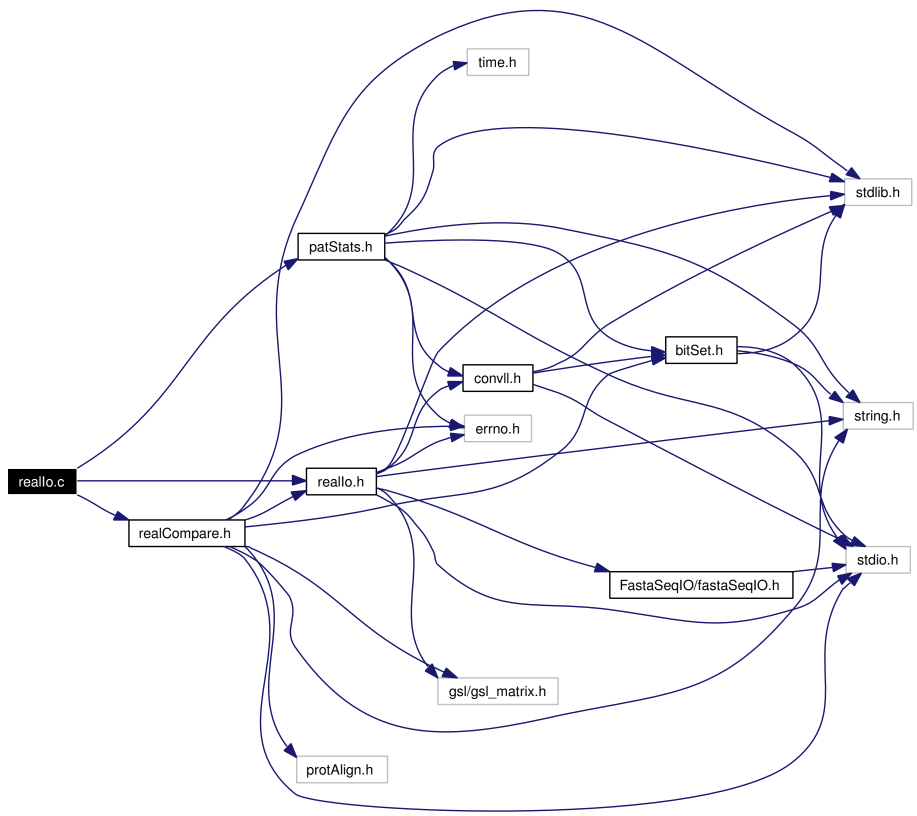
Definition at line 36 of file realCompare.h.

Referenced by findCliqueCentroid(), outputRealPatsWCentroid(), and realComparison().

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 realIo.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().

```
{
  int i;
  int thisDim = 0;
  int status = 1;
  int width;
  int fieldCount = 0;       // number of positions in a single sequence
  int numSeq = 0;       // number of sequences
  int dim = 0;          // The dimensionality of the sequences

  // NOTE this is not checking the dimensionality of the last sequence...
  // that's bad.  We can fix that though.
  // Check the dimensionality of each sequence
  for (i = 0; i < nl; i++)
    {
      if (buf[i][0] == '>')
    {

      // If this is only the second sequence we've seen,
      // record the dimensionality of the first sequence
      // as the dim to insist upon from here on out
      if (numSeq == 1)
        {
          dim = thisDim;

          // For other sequences, we need to check to make sure
          // that they've got the same dimensions as previous
          // sequences
        }
      else if (numSeq > 1)
        {

          // If the dimensions are wrong, quit with status=0
          if (thisDim != dim)
        {
          status = 0;
          break;
        }
        }
      numSeq++;
      width = 0;
      thisDim = 0;
    }
      else
    {

      // Field count can be different for each sequence but
      // must be the same for each dimension in a single sequence
      fieldCount = countFields (buf[i], sep);

      // If this is the first row of this sequence,
      // then store the number of fields
      if (thisDim == 0)
        {
```

```
          width = fieldCount;

          // If it's not the first row, make sure it has the
          // same number of fields as previous rows in this
          // sequence
        }
      else
        {
          if (fieldCount != width)
        {
          status = 0;
          break;
        }
        }
      thisDim++;
    }
    }

  // Pass back the numSeq and dim
  *numSeq_p = numSeq;
  *dim_p = thisDim;
  return status;
}
```

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;
94   int end = 0;
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
106   for (i = 0; i < strlen (s); i++)
107     {
108
```

```
      // The previous state was space
      if (status == 0)
    {

      // We hit a word
      if (s[i] != sep)
        {
          begin = i;
          status = 1;
        }
      else
        {           // We hit more space
          continue;
        }
    }
      else
    {           // The previous state was word
      if (s[i] != sep)
        {
          continue;
        }
      else
        {           // We hit a space
          end = i - 1;
          status = 0;

          // being and end now delimit a word,
          // turn that word into a double
          val = wordToDouble (s, begin, end);
          fieldCount++;
        }
    }
    }

  // At the end, if we were in a word, we have
  // one more field
  if (status == 1)
    {               // We're in a word
      val = wordToDouble (s, begin, strlen (s));
      fieldCount++;
    }
  return fieldCount;
}
```

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().

```
{
```

```
  int i = 0;
  int totalFields = 0;
  int seqNo = 0;
  while (i < nl)
    {

      // Hit a new sequence
      if (buf[i][0] == '>')
    {
      seqNo++;

      // Assume that the sequence has at least
      // one row (should have called checkRealDataFormat!
      // and that each row has the same number of fields
      totalFields += countFields (buf[i + 1], sep);
    }
      i++;
    }
  return totalFields;
}
```

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().

```
{
  double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
  int i = 0, j = 0, indmin = -1, counter = 0;
  double sim = 0, min = 0, flagmin = 0;
  double *cliqueAdjMat = NULL;
  cliqueAdjMat = (double *) malloc (curCliq->set->size * sizeof (double));
  if (cliqueAdjMat == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < curCliq->set->size; i++)
    {
      cliqueAdjMat[i] = 0;
    }

  // We'll accumulate our comparison function values... except here
  // we're really assuming that we're using a match factor, with
  // value less than one, so that we can subtract it from one to
  // get a distance, and then find the centroid by identifying the
  // node with the smallest cumulative Euclidean distance to all
```

```
// nodes.
// Note that we only need to compare each unique pair, and can apply
// the results from each comparison to each member of the pair,
// hence the somewhat odd indices of initiation for the for loops.
comparisonFunc = getCompFunc (compFunc);
for (i = 0; i < curCliq->set->size; i++)
  {
    for (j = i + 1; j < curCliq->set->size; j++)
  {
    sim =
      comparisonFunc (data, curCliq->set->members[i],
              curCliq->set->members[j], L, extraParams);

    // printf("i = %d, j = %d, L = %d, extra = %lf, sim =
    // %lf\n",i,j,L,extraParams[0],sim);
    cliqueAdjMat[i] += pow (1 - sim, 2);
    cliqueAdjMat[j] += pow (1 - sim, 2);
  }
  }

// Now we find the minimum Euclidean distance.
min = cliqueAdjMat[0];
indmin = 0;
for (i = 1; i < curCliq->set->size; i++)
  {

    // printf("index %d product = %lf\n",i,cliqueAdjMat[i]);
    if (cliqueAdjMat[i] < min)
  {
    indmin = i;
    min = cliqueAdjMat[i];
    flagmin = 0;
  }
    else if (cliqueAdjMat[i] == min)
  {
    flagmin = 1;
  }
  }

// If we had a duplicate on the minimum, we locate all duplicates.
if (flagmin == 1)
  {
    counter = 0;
    for (i = 0; i < curCliq->set->size; i++)
  {
    if (cliqueAdjMat[i] == min)
      {
        counter++;
        candidates[counter] = i;
      }
  }

    // Store the number of candidates at the array's beginning
    candidates[0] = counter;
    free (cliqueAdjMat);
    return (-1);
  }
```

```
  else
    {
      free (cliqueAdjMat);
      return (indmin);
    }
}
```

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.

Referenced by main().

```
{
  int i;
  if (data != NULL)
    {
      if (data->indexToPos != NULL)
    {
      free (data->indexToPos);
      data->indexToPos = NULL;
    }
      if (data->indexToSeq != NULL)
    {
      free (data->indexToSeq);
      data->indexToSeq = NULL;
    }
      if (data->offsetToIndex != NULL)
    {
      for (i = 0; i < data->size; i++)
        {
          free (data->offsetToIndex[i]);
          data->offsetToIndex[i] = NULL;
        }
      free (data->offsetToIndex);
      data->offsetToIndex = NULL;
    }
      for (i = 0; i < data->size; i++)
    {
      if (data->seq[i] != NULL)
        {
          gsl_matrix_free (data->seq[i]);
          data->seq[i] = NULL;
        }
      if (data->label[i] != NULL)
        {
          free (data->label[i]);
          data->label[i] = NULL;
```

```
          }
      }
        if (data->seq != NULL)
      {
        free (data->seq);
        data->seq = NULL;
      }
        if (data->label != NULL)
      {
        free (data->label);
        data->label = NULL;
      }
        free (data);
        data = NULL;
      }
    return data;
  }
```

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().

```
  {
    if (data == NULL || data->seq == NULL || data->seq[0] == NULL)
      {
        fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
        fflush (stderr);
        exit (0);
      }
    return data->seq[0]->size2;
  }
```

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(), massSpecCompareWElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsdCompare().

```
{
  if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
      || index > data->indexSize)
    {
      fprintf (stderr, "Passed bad data to getRdhIndexSeqPos -- error!");
      fflush (stderr);
      exit (0);
    }

  /*
     printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
   */
  /*
     fflush(stdout);
   */
  *seq = data->indexToSeq[index];
  *pos = data->indexToPos[index];
  return 0;
}
```

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().

```
{
  if (data == NULL || data->label == NULL || data->label[seqNo] == NULL)
    {
      fprintf (stderr, "Passed bad data to getRdhLabel -- error!");
      fflush (stderr);
      exit (0);
    }
  return data->label[seqNo];
}
```

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().

```
{
  if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
    {
      fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
      fflush (stderr);
      exit (0);
    }
  return data->seq[seqNo]->size1;
}
```

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().

```
{
  if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
      || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
    {
      fprintf (stderr, "Passed bad data to getRdhValue -- error!");
      fflush (stderr);
      exit (0);
    }
  return gsl_matrix_get (data->seq[seqNo], posNo, dimNo);
}
```

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().

```
{
  int i;
  rdh_t *data = NULL;

  // Allocate space for our structure
  data = (rdh_t *) malloc (sizeof (rdh_t));
  if (data == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  data->size = x;

  // Index has to be initialized later, once
  // we know the word size.
  data->indexSize = 0;
  data->indexToSeq = NULL;
  data->indexToPos = NULL;

  /*
     data->indexSize = y;
   */
  data->label = (char **) malloc (data->size * sizeof (char *));
  if (data->label == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  data->seq = (gsl_matrix **) malloc (data->size * sizeof (gsl_matrix *));
  if (data->seq == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < data->size; i++)
    {
      data->label[i] = NULL;
      data->seq[i] = NULL;
    }
  return data;
}
```

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().

```
{
  data->seq[seqNo] = gsl_matrix_alloc (x, y);
  if (data->seq[seqNo] == NULL)
    {
      return 0;
    }
  else
    {
      return 1;
    }
}
```

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 *seqGap*, 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::indexToSeq, rdh_t::offsetToIndex, and rdh_t::size.

Referenced by realComparison().

```
{
  int i, j, k;
  int numWindows = 0;
  int thisNumWindows;
  int numSeq;
  int seqLen = 0;

  // The number of sequences
  numSeq = data->size;
```

```
// Allocate offsetToIndex's outer structure
data->offsetToIndex = (int **) malloc (numSeq * sizeof (int *));
if (data->offsetToIndex == NULL)
  {
    fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
  }

// For each sequence
for (i = 0; i < numSeq; i++)
  {

    // How many windows are in this sequence
    seqLen = getRdhSeqLength (data, i);
    numWindows += seqLen - wordSize + 1;

    // And also use this to further allocate offsetToIndex
    data->offsetToIndex[i] =
  (int *) malloc ((seqLen - wordSize + 1) * sizeof (int));
    if (data->offsetToIndex[i] == NULL)
  {
    fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
  }
  }

// One index for each word plus seqGap between each sequence
// and a gap at the end
data->indexSize = numWindows + numSeq * seqGap;

// Allocate indexToSeq
// NOTE that it should be size of int, not int *... I think we got
// fortunate in the previous revision because they are the same
// size
data->indexToSeq = (int *) malloc (data->indexSize * sizeof (int));
if (data->indexToSeq == NULL)
  {
    fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
  }

// Allocate indexToPos
// See above for int vs. int* argument.
data->indexToPos = (int *) malloc (data->indexSize * sizeof (int));
if (data->indexToPos == NULL)
  {
    fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
  }

// Fill in the values
k = 0;
for (i = 0; i < numSeq; i++)
```

```
    {

      // How many windows are in this sequence?
      thisNumWindows = getRdhSeqLength (data, i) - wordSize + 1;

      // For each window, make an entry in the indexToSeq
      // and indexToPos and offsetToIndex
      for (j = 0; j < thisNumWindows; j++)
    {
      data->indexToSeq[k] = i;
      data->indexToPos[k] = j;
      data->offsetToIndex[i][j] = k;
      k++;
    }

      // Add gaps between sequences in the index.
      // Usually seqGap is just 1;
      for (j = 0; j < seqGap; j++)
    {

      // -1 means no sequence and no position
      data->indexToSeq[k] = -1;
      data->indexToPos[k] = -1;
      k++;
    }
    }
  return 0;
}
```

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
1213   // the maxSup, so we know we can reach candidates[maxSup] without
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.

Referenced by main().

```
1048 {
1049   int i, j, pos1;
1050   int curSeq, curPos;
1051   cll_t *curCliq = NULL;
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     {
1060       fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
```

```
    }
      else
    {
      fprintf (OUTPUT_FILE, "\n");
    }
      for (j = 0; j < curCliq->set->size; j++)
    {
      pos1 = curCliq->set->members[j];
      getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
      fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);
      fprintf (OUTPUT_FILE, "%lf\t",
           gsl_matrix_get (data->seq[curSeq], curPos, 0));

      /*
         for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
         mySequences[curSeq].seq[k]); }
       */
      fprintf (OUTPUT_FILE, "\n");
    }
      fprintf (OUTPUT_FILE, "\n\n");
      curCliq = curCliq->next;
      i++;
    }
  return 0;
}
```

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.

Referenced by main().

```
{
  int i, j, k, pos1, centroid;
  int curSeq, curPos;
  int maxSup = 0;
  cll_t *curCliq = NULL;
  double mfToCentroid = 0;
  double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
  int *candidates = NULL;
  curCliq = allPats;
  while (curCliq != NULL)
    {
      if (curCliq->set->size > maxSup)
    {
      maxSup = curCliq->set->size;
    }
```

```
      curCliq = curCliq->next;
    }
  candidates = (int *) malloc ((maxSup + 1) * sizeof (int));
  if (candidates == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i <= maxSup; i++)
    {
      candidates[i] = 0;
    }
  comparisonFunc = getCompFunc (compFunc);
  curCliq = allPats;
  i = 0;
  while (curCliq != NULL)
    {
      fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\n", i,
           curCliq->length + L, curCliq->set->size);
      centroid =
    findCliqueCentroid (data, curCliq, L, compFunc, extraParams,
                candidates);
      if (centroid < 0)
    {
      centroid = makeAlternateCentroid (data, curCliq, candidates);

      // fprintf(OUTPUT_FILE, "WARNING: No single node in"
      // " cluster has non-zero similarity to all other\n nodes"
      // " in cluster; centroid set to first node.\n");
      // centroid = 0;
    }
      for (j = 0; j < curCliq->set->size; j++)
    {
      pos1 = curCliq->set->members[j];
      getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
      fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);

      // fprintf(OUTPUT_FILE, "%lf\t",
      // gsl_matrix_get(data->seq[curSeq],curPos,0));
      mfToCentroid =
        comparisonFunc (data, curCliq->set->members[j],
                curCliq->set->members[centroid], L, extraParams);
      fprintf (OUTPUT_FILE, "%lf\t", mfToCentroid);

      /*
         for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
         mySequences[curSeq].seq[k]); }
       */
      fprintf (OUTPUT_FILE, "\n");
    }
      fprintf (OUTPUT_FILE, "\n\n");
      curCliq = curCliq->next;
      i++;
      for (k = 0; k <= maxSup; k++)
    {
      candidates[k] = 0;
```

```
    }
    }
  free (candidates);
  return 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().

```
{
  int i;
  int seqNo = -1;
  int totalNumFields;
  rdh_t *data = NULL;
  totalNumFields = countTotalFields (buf, nl, sep);

  /*
     data = initRdh(numSeq, totalNumFields + numSeq - 1);
   */
  data = initRdh (numSeq);

  // We're going to add an empty index between
  // windows that correspond to different
  // sequences

  // Fast forward to the first sequence
  i = 0;
  while (i < nl)
    {

      // Hit a new sequence
      if (buf[i][0] == '>')
    {
      seqNo++;      // Note that seqNo started at -1!
      pushOnRdhSeq (data, buf, i, dim, sep);
      i += dim + 1;
    }
      else
```

```
    {
      i++;
    }
    }

  /*
     printRdhSeq(data, 0, stdout);
   */
  return data;
}
```

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().

```
{
  int i, j;
  int len;
  int dim;
  len = getRdhSeqLength (data, seqNo);
  dim = getRdhDim (data);
  fprintf (FH, "%s\n", getRdhLabel (data, seqNo));
  for (i = 0; i < len; i++)
    {
      for (j = 0; j < dim; j++)
    {
      fprintf (FH, "%3.1f ", getRdhValue (data, seqNo, i, j));
    }
      fprintf (FH, "\n");
    }
  return 0;
}
```

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 setRdhLabel().

Referenced by parseRealData().

```
864 {
865   int i, j, k;
866   int numFields;
867
868   // 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
893   for (j = i + 1, k = 0; j < i + 1 + dim; j++, k++)
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, l, indexNo); indexNo++;
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().

Referenced by main().

```
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
995   // contents into 'buf'.  This function also
996   // alters the contents of the location pointed
997   // to by &nl.  Now nl is the number of lines
998   // in the file (or the size of the buff array.
999   buf = ReadFile (INPUT, &nl);
1000   if (buf == NULL)
1001     {
1002       return NULL;
1003     }
1004   status = checkRealDataFormat (buf, nl, sep, &numSeq, &dimensions);
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);
```

```

  // Free up our buffer
  for (i = 0; i < nl; i++)
    {
      if (buf[i] != NULL)
    {
      free (buf[i]);
    }
    }
  if (buf != NULL)
    {
      free (buf);
    }
  return data;
}
```

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().

```
{
  int i;
  int begin = 0;
  int end = 0;
  int status = 0;       // 0 = in sep, 1 = in word
  int fieldCount = 0;
  double val;

  // Make sure the string is not null and
  // the rdh_t gsl_matrix array is not null
  // and the selected gsl_matrix is not null
  if (s == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
    {
      fprintf (stderr, "Passed bad data to setRdhColFromString -- error!");
      fflush (stderr);
      exit (0);
    }

  // Loop over the length of the string
  for (i = 0; i < strlen (s); i++)
```

```
    {

      // The previous state was space
      if (status == 0)
    {

      // We hit a word
      if (s[i] != sep)
        {
          begin = i;
          status = 1;
        }
      else
        {           // We hit more space
          continue;
        }
    }
      else
    {           // The previous state was word
      if (s[i] != sep)
        {
          continue;
        }
      else
        {           // We hit a space
          end = i - 1;
          status = 0;
          val = wordToDouble (s, begin, end);

          // Go to the gsl_matrix object data->seq[seqNo]
          // and set the (fieldCount, colNo) = val;
          setRdhValue (data, seqNo, fieldCount, colNo, val);
          fieldCount++;
        }
    }
    }

  // At the end, if we were in a word, we have
  // one more field
  if (status == 1)
    {               // We're in a word
      val = wordToDouble (s, begin, strlen (s));

      // Added in, MPS 5/3/05 ---
      // And don't forget to set the RdhValue!
      setRdhValue (data, seqNo, fieldCount, colNo, val);
      fieldCount++;
    }
  return fieldCount;
}
```

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.

```
{
  if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
      || index > data->indexSize)
    {
      fprintf (stderr, "Passed bad data to getRdhValue -- error!");
      fflush (stderr);
      exit (0);
    }

  /*
     printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
   */
  /*
     fflush(stdout);
   */
  data->indexToSeq[index] = seqNo;
  data->indexToPos[index] = posNo;
  return 0;
}
```

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().

```
{
  if (data->seq == NULL || data->label == NULL)
    {
      fprintf (stderr, "Passed bad data to setRdhLabel -- error!");
      fflush (stderr);
      exit (0);
    }
  data->label[seqNo] = strdup (s);
  if (data->label[seqNo] == NULL)
    {
      fprintf (stderr, "\nMemory Error allocating label!\n%s\n",
           strerror (errno));
      fflush (stderr);
```

```
      exit (0);
    }
  return 0;
}
```

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 the data should be set.

Definition at line 575 of file realIo.c.

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

Referenced by setRdhColFromString().

```
{
  if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
      || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
    {
      fprintf (stderr, "Passed bad data to setRdhValue -- error!");
      fflush (stderr);
      exit (0);
    }
  gsl_matrix_set (data->seq[seqNo], posNo, dimNo, val);
  return 0;
}
```

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().

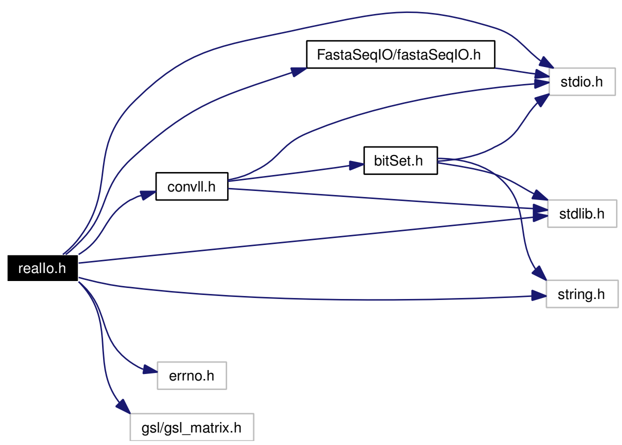
```
{
  char *str = NULL;
  char *endptr;
  double val;
  int size;
```

```
  int memsize;

  // Check for a sane substring
  if (end - begin <= 0)
    {
      fprintf (stderr, "\nInvalid argument to wordToDouble!\n");
      fflush (stderr);
      exit (0);
    }

  // Get the required string size
  memsize = end - begin + 2;    // An extra space in mem for null-termination
  size = end - begin + 1;

  // Get memory for a temporary string
  str = (char *) malloc (memsize * sizeof (char));
  if (str == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }

  // Make sure the string ends with a null char
  str[size] = ’\0’;

  // Copy the word into str
  str = strncpy (str, s + begin, size);

  // Set endptr to str as initial value
  endptr = str;
  val = strtod (str, &endptr);

  // endptr should point to the last char
  // used in the conversion if strtod worked
  if (val == 0 && endptr == str)
    {
      fprintf (stderr, "\nError making double from string: %s\n", str);
      fflush (stderr);
      exit (0);
    }
  free (str);
  return val;
}
```

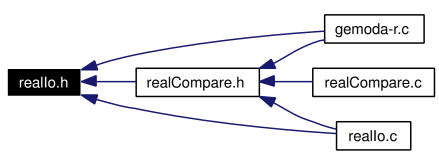
3.20 realIo.h File Reference

```
#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(), massSpecCompareWElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsdCompare().

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 *seqGap*, 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::indexToSeq, 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.

Referenced by main().

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*;

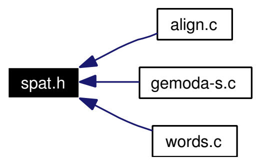
Definition at line 850 of file realIo.c.

References checkRealDataFormat(), parseRealData(), and ReadFile().

Referenced by main().

3.21 spat.h File Reference

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



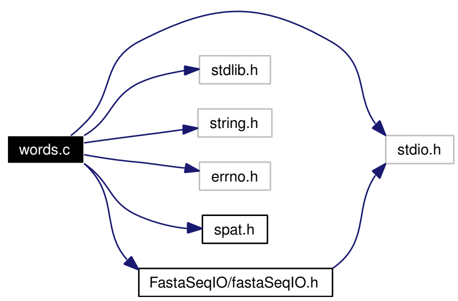
Data Structures

- 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(), sHashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, sOffset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

Referenced by main().

```
{
  int i, j;
  int totalChars = 0;
  int hashSize;
  sHashEntry_t newEntry;
  sHashEntry_t *ep;
  sHash_t wordHash;
  sPat_t *words = NULL;
```

```
int wc = 0;
int prev = -1;
int l;


// Count the total number of characters.  This
// is the upper limit on how many words we can have
for (i = 0; i < numSeq; i++)
  {
    totalChars += strlen (seq[i].seq);
  }

// Get a prime number for the size of the hash table
hashSize = sieve3 ((long) (2 * totalChars));
wordHash = initSHash (hashSize);

// Chop up each sequence and hash out the words of size L
for (i = 0; i < numSeq; i++)
  {
    prev = -1;

    // skip sequences that are too short to have
    // a pattern
    if (strlen (seq[i].seq) < L)
  {
    continue;
  }
    for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
  {

    // Make a hash table entry for this word
    newEntry.key = &(seq[i].seq[j]);
    newEntry.data = 1;
    newEntry.idx = wc;
    newEntry.L = L;

    // Check to see if it’s already in the hash table
    ep = searchSHash (&newEntry, &wordHash, 0);
    if (ep == NULL)
      {

        // If it’s not, create an entry for it
        ep = searchSHash (&newEntry, &wordHash, 1);

        // Increase the size of our word array
        words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
        if (words == NULL)
      {
        fprintf (stderr, "Error!\n");
        fflush (stderr);
      }
        // Add the new word
        words[wc].string = &(seq[i].seq[j]);
        words[wc].length = L;
        words[wc].support = 1;
        words[wc].offset =
      (sOffset_t *) malloc (1 * sizeof (sOffset_t));
```

```
        if (words[wc].offset == NULL)
      {
        fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
        fflush (stderr);
        exit (0);
      }
        words[wc].offset[0].seq = i;
        words[wc].offset[0].pos = j;
        words[wc].offset[0].prev = prev;
        words[wc].offset[0].next = -1;

        if (prev != -1)
      {
        words[prev].offset[words[prev].support - 1].next = wc;
      }
        prev = wc;
        wc++;

      }
    else
      {

        // If it is, increase the count for this word
        ep->data++;

        // add a new offset to the word array
        l = words[ep->idx].support;
        words[ep->idx].offset =
      (sOffset_t *) realloc (words[ep->idx].offset,
                     (l + 1) * sizeof (sOffset_t));
        words[ep->idx].offset[l].seq = i;
        words[ep->idx].offset[l].pos = j;
        words[ep->idx].offset[l].prev = prev;
        words[ep->idx].offset[l].next = -1;

        // Update the next/prev
        if (prev != -1)
      {
        words[prev].offset[words[prev].support - 1].next = ep->idx;
      }
        prev = ep->idx;

        // Have to put this down here for cases when we create
        // a word and it is immeadiately followed by itself!!
        words[ep->idx].support += 1;
      }
  }
  }


  destroySHash (&wordHash);
  *numWords = wc;
  return words;
}
```

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().

```
{
  int i;
  free (thisHash->iHashSize);
  free (thisHash->hashSize);
  for (i = 0; i < thisHash->totalSize; i++)
    {
      if (thisHash->hash[i] != NULL)
    {
      free (thisHash->hash[i]);
      thisHash->hash[i] = NULL;
    }
    }
  if (thisHash->hash != NULL)
    {
      free (thisHash->hash);
      thisHash->hash = NULL;
    }
  return 0;
}
```

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.

```
{
  int i;
  for (i = 0; i < wc; i++)
    {
      if (words[i].offset != NULL)
    {
      free (words[i].offset);
      words[i].offset = NULL;
    }
    }
  free (words);
  words = NULL;
  return 0;
}
```

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().

```
{
  unsigned long hash = 5381;
  int c;

  while ((c = *str++))
    hash = ((hash << 5) + hash) + c;    /* hash * 33 + c */

  return hash;
}
```

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.

```
{
  char *p;
  unsigned int h, g;

  h = 0;
  for (p = s; *p != '\0'; p++)
    {
      h = (h << 4) + *p;
      if ((g = h & 0xF0000000))
    {
      h ^= g >> 24;
      h ^= g;
    }
    }
  return h;
}
```

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().

```
{
  int i = 0;
  int step = 0;
  sHash_t this;

  this.totalSize = n;
  this.hashSize = (int *) malloc (n * sizeof (int));
  if (this.hashSize == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  this.iHashSize = (int *) malloc (n * sizeof (int));
  if (this.iHashSize == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  this.hash = (sHashEntry_t **) malloc (n * sizeof (sHashEntry_t *));
  if (this.hash == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  for (i = 0; i < n; i++)
    {
      this.hash[i] = NULL;
      this.hashSize[i] = 0;
      this.iHashSize[i] = step;
    }
  return this;
}
```

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.

```
{
  int i, j;
  char string[SHASH_MAX_KEY_SIZE];

  for (i = 0; i < thisHash->totalSize; i++)
    {
      for (j = 0; j < thisHash->hashSize[i]; j++)
    {

```

```
      strncpy (string, thisHash->hash[i][j].key, thisHash->hash[i][j].L);
      string[thisHash->hash[i][j].L] = '\0';
      fprintf (FH, "%s %d\n", string, thisHash->hash[i][j].data);

    }
    }
  return 0;
}
```

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.

```
{
  char *s = NULL;
  int i, j;
  int size = 0;
  for (i = 0; i < n; i++)
    {
      if (a[i].length > size)
    {
      s = (char *) realloc (s, a[i].length * sizeof (char));
    }
      strncpy (s, a[i].string, a[i].length);
      s[a[i].length] = '\0';
      printf ("%d:  %s\n", i, s);
      for (j = 0; j < a[i].support; j++)
    {
      printf ("\t%d %d -> (%d, %d)\n", a[i].offset[j].seq,
          a[i].offset[j].pos, a[i].offset[j].prev,
          a[i].offset[j].next);
    }
      printf ("\n");
    }
  free (s);
  return 0;
}
```

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, sHashEntry_t::key, sHashEntry_t::L, SHASH_MAX_KEY_SIZE, and sHash_t::totalSize.

Referenced by countWords2().

```
{
  char string[SHASH_MAX_KEY_SIZE];
  unsigned long (*hashFunction) () = &hash1;
  int i, thisIndex;
  int status = 0;

  // A string to store the key
  strncpy (string, newEntry->key, newEntry->L);
  string[newEntry->L] = '\0';

  // The index that this key hashes to
  thisIndex = hashFunction ((unsigned char *) string) % thisHash->totalSize;

  // For each member that has this index, check to see
  // if the key is the same
  for (i = 0; i < thisHash->hashSize[thisIndex]; i++)
    {
      if (strncmp (thisHash->hash[thisIndex][i].key, string, newEntry->L) ==
      0)
    {

      // We found a match
      /*
         printf("\t%s already in hash table!\n");
       */
      status = 1;
      return &(thisHash->hash[thisIndex][i]);
      break;

    }
    }

  // If we didn't find the key and we're told to create it,
  // then allocate new memory for the hashEntry and put it in
  if (status == 0 && create != 0)
    {

      // Allocate space for the new entry at this index
      if (thisHash->iHashSize[thisIndex] == 0)
    {
      thisHash->hash[thisIndex] =
        (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
    }
      else
    {
      thisHash->hash[thisIndex] =
        (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
                      (thisHash->iHashSize[thisIndex] +
                       1) * sizeof (sHashEntry_t));
    }
      if (thisHash->hash[thisIndex] == NULL)
    {
```

```
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
      // Increase our record of the size
      i = thisHash->hashSize[thisIndex];
      thisHash->hash[thisIndex][i] = *newEntry;
      thisHash->iHashSize[thisIndex]++;
      thisHash->hashSize[thisIndex]++;


      // Return a pointer to this entry
      return &(thisHash->hash[thisIndex][i]);
    }
  return NULL;
}
```

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().

```
{
  int i, p, j;
  int *a;
  a = (int *) malloc ((n + 1) * sizeof (int));
  if (a == NULL)
    {
      fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
      fflush (stderr);
      exit (0);
    }
  a[0] = 0;
  a[1] = 0;
  for (i = 2; i < n; i++)
    {
      a[i] = 1;
    }
  p = 2;
  do
    {
      j = 2 * p;
      do
    {
      a[j] = 0;
      j = j + p;
    }
```

```
      while (j <= n);
      p = p + 1;
    }
  while (p * p < 2 * n);
  for (i = n; i > 2; i--)
    {
      if (a[i])
    {
      free (a);
      return i;
    }
    }
  free (a);
  return 0;
}
```

Index