alist format

David MacKay, Matthew Davey, and John Lafferty have all written low density parity check matrices in a format called an alist. GHG.p is one program that writes alists.

Here is the C structure for an alist:

```
typedef struct {
  int N , M ;      /* size of the matrix */
  int **mlist;     /* list of integer coordinates in the m direction where the non-zero entries are */
  int **nlist;     /* list of integer coordinates in the n direction where the non-zero entries are */
  int *num_mlist;  /* weight of each row, m */
  int *num_nlist;  /* weight of each column n */
  int *l_up_to ;
  int *u_up_to ;
  int *norder ;
  int biggest_num_m ;       /* actual biggest sizes */
  int biggest_num_n ;
  int biggest_num_m_alloc ; /* sizes used for memory allocation */
  int biggest_num_n_alloc ;
  int tot ;
  int same_length ;  /* whether all vectors in mlist and nlist have same length */
} alist_matrix ;
```

When written to file, this is the format:

```
void write_alist ( FILE *fp , alist_matrix *a ) {
  /* this assumes that mlist and nlist have the form of a rectangular
     matrix in the file; if lists have unequal lengths, then the
     entries should be present (eg zero values) but are ignored
     */
  int N = a->N , M = a->M ;

  fprintf ( fp , "%d %d\n" , N , M ) ;
  fprintf ( fp , "%d %d\n" , a->biggest_num_n , a->biggest_num_m ) ;
  write_ivector ( fp , a->num_nlist , 1 , N ) ;
  write_ivector ( fp , a->num_mlist , 1 , M ) ;
  write_imatrix ( fp , a->nlist , 1 , N , 1 , a->biggest_num_n ) ;
  write_imatrix ( fp , a->mlist , 1 , M , 1 , a->biggest_num_m ) ;
}
```

Here is an example of an list in a file called 12.4.3.111 (actually this is a bad example, since normally our parity check matrices are wider than they are high; this one is transposed, I don't know why):

```
12 16
4 3
4 4 4 4 4 4 4 4 4 4 4 4
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
3 8 10 13
4 7 9 13
2 5 7 10
4 6 11 14
3 9 15 16
1 6 9 10
4 8 12 15
2 6 12 16
1 7 14 16
3 5 12 14
2 11 13 15
1 5 8 11
6 9 12
3 8 11
1 5 10
2 4 7
3 10 12
4 6 8
2 3 9
1 7 12
2 5 6
1 3 6
4 11 12
7 8 10
1 2 11
4 9 10
5 7 11
5 8 9
```

and here is a ps files showing the matrix in ordinary 1/0 format: A.ps. Here it is verbatim.

```
 0 0 1 0 0 0 0 1 0 1 0 0 1 0 0 0
 0 0 0 1 0 0 1 0 1 0 0 0 1 0 0 0
 0 1 0 0 1 0 1 0 0 1 0 0 0 0 0 0
 0 0 0 1 0 1 0 0 0 0 1 0 0 1 0 0
```

```
 0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 1
 1 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0
 0 0 0 1 0 0 0 1 0 0 0 1 0 0 1 0
 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 1
 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1
 0 0 1 0 1 0 0 0 0 0 0 1 0 1 0 0
 0 1 0 0 0 0 0 0 0 0 1 0 1 0 1 0
 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0
```

The alist row '3 8 10 13' says the indices of the 1s in the top row. The alist row '3 8 11' says the indices of the 1s in the 2nd column.

By convention, the righthand M*M matrix is an invertible matrix, if this can be arranged. (Many of my programs check for invertibility of this matrix.)

NB: If the rows or columns are irregular, you must pad the low-weight rows/columns with zeroes so as to make the two sets of lists regular.

Here is an example of using an alist to do a matrix multiplication:

```
void alist_times_cvector_sparse_mod2
( alist_matrix *a , unsigned char *x , unsigned char *y ) {
  int n , m , i ;
  int *nlist ;

  for ( m = 1 ; m <= a->M ; m++ ) {
    y[m] = 0 ;
  }
  for ( n = 1 ; n <= a->N ; n++ ) {
    if ( x[n] ) {
      nlist = a->nlist[n] ;
      for ( i = a->num_nlist[n] ; i >= 1 ; i -- ) {
        y[ nlist[i] ] ^= 1 ;
      }
    }
  }
}
```

Conventions in my software

I use a command 'cvector' to allocate memory for a character vector whose pointer is

```
  unsigned char *y
```

Similarly I use a command 'ivector'/'dvector' to allocate memory for a double vector whose pointer is

```
 int *y
 double *y
```

These allocation commands are in nrutil.c / nrutil.h.

G format

For some codes, the generator matrix is also provided. It's in a dense format that is read by Radford Neal's software in the MNC package.

"which bits are the message bits?"

This is an issue I don't bother to resolve. My viewpoint is that the code is a set of constraints, rather than viewing the code as a mapping from message bits to transmitted bits. Whenever I measure bit error rate, I find the average error rate of all of the bits. The user is free to select any convenient linearly independent set of bits to be the "message" bits. For a regular code I expect that any choice of this subset will give the same "bit error rate" as the average bit error rate. In a few of the codes I have used I have arranged the alist in such a way that the first K bits are indeed linearly independent, so the first K of N could be viewed as message bits.

David MacKay *<mackay@mrao.cam.ac.uk>*
Last modified: Wed Mar 29 14:13:14 2006