NPL project

General Description

The goal of the present project is to implement NPL (non-parametric linkage) analysis in the fastest possible way. This is the part in the Technion project of efficient implementing multipoint approach for gene linkage. Here is a general explanation about the whole project and the part of the present NPL project in it.

First step. Given a pedigree, information about the affected status of each individual in the pedigree, marker alleles information at specific loci for each individual, recombination values between adjacent loci and gene frequencies – the single point probabilities for each possible inheritance vector at each locus (inheritance distribution) are found. These probabilities depend only on the marker data for each locus itself. Each inheritance vector determines the inheritance pattern of specific locus in the pedigree. When inheritance distribution is found, the multipoint probabilities (for each inheritance vector at each locus) are computed using standard forward-backward HMM algorithm : assuming nodes in HMM chain are loci on the chromosome, hidden states are inheritance vectors, visible states are marker alleles at each specific locus (input data), emission probabilities are single point probabilities (of each inheritance vector at each locus) and transition probabilities are transitions between each pair of inheritance vectors at each pair of adjacent loci with specific recombination value between them. Such transition is computed by multiplying (number of correlated bits which are the same in 2 vectors)^(1 – recombination value) with (number of correlated bits which are different in 2 vectors)^(recombination value). So, using this method, we get 2 tables : left-to-right probabilities table (forward algorithm) and right-to-left probabilities table(backward algorithm).

Second step. Some scoring function should be defined to give the score for each inheritance vector depending on the observed phenotypes in the pedigree, either parametric or non-parametric (NPL score). The main difference between them is that parametric function (LOD-score) assumes linkage model in the computation while NPL doesn't. So when the linkage model is right, LOD-score should give more precise result, and in the case of misspecification of linkage model, NPL is supposed to give better result (it may be useful in the case of complex diseases).

As it was said above, the goal of this project is to implement the computation of NPL score. 2 definitions of scores were used: Spairs(v) and Sall(v) assuming v – some inheritance vector:

Spairs(v) – the number of pairs of alleles from distinct affected pedigree members which are IBD (identical by descent, that means got physically the same allele from the common ancestor);

Sall(v):

$$\mathrm{Sall}(v) = 2^{(-a)} * \sum_{h} \prod_{i=1}^{2f} \mathrm{bi}(h)!$$

where: a – the number of affected individuals in the pedigree
h – collection of alleles from affected individuals, when only 1 allele from each affected is picked (there are 2^a possible collections)
1..2f – (assuming f is the number of founders in the pedigree) 2f distinct founder alleles
bi(h) – the number of times founder allele i appears in h.

This score sharply increases if the number of affected individuals sharing some specific allele increases.
It is easily seen that the first score is computationally trivial while the second is complex.

Third step. Now we generalize the scoring function by taking its expected value over multipoint probability distribution of inheritance vectors at different loci:

$$S(x) = \sum_{w} S(w)*P[v(x) = w]$$

where: x – some locus (testpoint);
w – some inheritance vector (the sum is taken over all possible inheritance vectors);
S(w) – either Spairs(w) or Sall(w);
P[v(x) = w]=P – multipoint probability of vector w at x; if testpoint x is situated on some locus then P is computed by multiplication of left-to-right table[w,x] by right-to-left table[w,x]; else if x is situated somewhere between locus i and locus (i+1), one more step of forward algorithm is done using recombination value between locus i and x (for each vector w we store fromLeft(w)), one more step of backward algorithm is done using recombination value between locus (i+1) and x (storing in fromRight(w)), P=fromLeft(w)*fromRight(w).

All these calculations are performed using reduced inheritance vectors of length 2n-f (and so the number of vectors is 2^(2n-f)), when n is the number of nonfounders and f is the number of founders. We don't need to store all 2n bits of nonfounders in vector, because there is no information about the founders phase, so for each founder his first child's bit is set to 0 and doesn't appear in inheritance vector.

Algorithms and complexity

The main idea of the implementation is to go over all possible inheritance vectors in special order and for each vector to find Spairs and Sall scores. We use “gray code” to define the order of looping (over inheritance vectors):
Example of “gray code” for vector containing 3 bits: 000, 001, 011, 010, 110, 111, 101, 100. This order is very useful, because only 1 bit in vector is changed per iteration, and in the case when the inheritance bit is changed in the person which is not affected or has no affected children, NPL score (Spairs or Sall) doesn’t depend on this bit, so we can just copy the scores from the previous iteration.
This order is also used in the computation of Sall(v) for some vector v: we collect alleles from the affected individuals, one allele from each individual, get some collection and compute it’s score; when we want to get the next collection, we change 1 bit for some individual (in the order of “gray code”) and use the saved score of the previous collection to compute in O(1) score for current collection.
Here is more detailed description of the algorithm.

The main steps in the computation of Spairs and Sall scores:

Let’s assume:

F – number of founders in the pedigree
N – number of non-founders in the pedigree
A – number of affected individuals in the pedigree
SizeV = 2*N-F– the size of inheritance vector
PossibilitiesV = 2^SizeV – number of inheritance vectors

1) first of all, we need to initiate all the relevant data structures for the further computation:

- the array “ped” of the individuals in current pedigree, where indexes of parents are smaller than indexes of their children (parents appear to the left of their children in “ped” – topological sorting of pedigree graph), and for each person is stored the next information: indexes of parents in “ped” (mother and father), affected(true/false), allele1 and allele2 (out of 2*F founders alleles), useful (if the person has no affected children and is not affected himself => not useful, else useful). In this step, we initiate “ped” by using arrays PedigreeGraph, Founders and NonFounders (they already exist in the program), and each person in “ped” gets indexes of parents and affected status : time and memory complexity is O(F+N);
- we go over “ped” and create the linked list of affected persons which is then converted to array of pointers to affected persons in “ped”(Affected array): time complexity O(F+N), memory O(A);
- we go over all founders in “ped” and for each founder with index i we store alleles (2*i = allele1, 2*i+1 = allele2): time complexity O(F);

- we allocate array Falleles of size 2*F, which will be used for storing the number of times each founder allele appears in affected individuals during the computation of NPL scores, initiated to 0 at each cell: memory and time O(F);

- we also allocate array Aff of size A (for Sall computation), each cell with index i in Aff will reflect the status of collecting allele from appropriate affected individual (with index i in Affected): 1 if allele1 is collected, 2 if allele2 is collected (all cells are initiated to 1): memory O(A), time O(A);

- in order to initiate "useful" field of each person in "ped", we iteratively go over "ped" from the last index (F+N-1) to first and perform for each person: if he is affected => update for him and recursively for all his ancestors "useful"=true, else go to the next (previous in order of "ped") person; time complexity O(F+N);

- we allocate array Ivector (inheritance vector) of size SizeV, in each cell store: inheritance status (0 or 1; initiated to 0), index of person in "ped" for which this cell is responsible; parent status for this person (mother or father), for which this cell is responsible; last 2 fields we initiate by going through all nonfounders in "ped" and only for those nonfounders that are not first children of their parents, we store their bits in Ivector; all the alleles in nonfounders bits which are first children of their parents are stored once (in "ped", allele1 and allele2) and will be never changed (assuming inheritance pattern 0 at corresponding bits), for the rest of nonfounders – their alleles are stored using the inheritance pattern defined by Ivector (and will be changed at each iteration of the main loop, for each inheritance vector): time complexity O(N), memory O(SizeV);

Overall time and memory complexity of step 1: O(F+N).
Now we are ready to perform the computation of NPL scores, step 2.

2) As it was mentioned above, in the main loop of this step we go over all inheritance vectors in the "gray code" order, in each iteration we also remember the values of Spairs and Sall from the previous iteration. Before computing new scores we check if the current bit of Ivector to be changed corresponds to useful person or not: if not => we change this bit and copy the previous values of Sall and Spairs to the current scores, and go to the next vector. If the person is useful => we change the bit, rearrange alleles in "ped" according to new inheritance vector (this may take O(F+N) time) and compute Spairs(v) and Sall(v) separately (for the current inheritance vector v):

Computing Spairs(v): intiation: Spairs(v) = 0; we go through all affected persons using Affected array and for each allele (1 and 2) of person i we add to Spairs(v) Falleles[allele] and then increase Falleles[allele]++; this yields exactly the number of pairs of persons that share some founder alleles (IBD, because each founder allele is designated with unique number).
Time complexity: O(A).

Computing Sall(v): initiation: Sall(v) = 1; Sum =0;

For each affected person i from Affected we collect allele1 (by setting Aff[i] = 1 and increasing Falleles[i.allele1]++), and multiply Sall(v) <- Sall(v)*Falleles[i.person];

Time complexity of initiation: O(A);

This yelds initial Sall(v) score (= Pi (Falleles[i]!); Sum = Sall(v);

Now we iterate through all possible collections exactly the same way as we did with vectors in main loop using "gray code" order, and for each collection bit i to be changed we perform (assuming that the score from the previous iteration is Sall(v)):

```
If Aff[i] = 1 (then now we need to pick up allele2 of person i instead of allele1) :
    Aff[i] = 2;
    We divide Sall(v) by the number of times that allele1 of person i appears in Falleles (by Falleles[Affected[i].allele1]);
    Decrease Falleles[Affected[i].allele1]-- (because now we pick up allele2 instead of allele1);
    Increase Falleles[Affected[i].allele2]++;
    We multiply Sall(v) by the number of times that allele2 of person i appears in Falleles (Falleles[Affected[i].allele2]);
Else we do exactly the same but allele1 and allele2 are now changed in their places.
We add Sall(v) to Sum;
Each iteration takes O(1) time.
```

At the end of loop : Sall(v) = Sum/(2^A);

The overall time complexity of computing Sall(v) is O(A) + (number of iterations)*O(1) = O(A)+O(2^A) = O(2^A);

The overall time complexity of step 2:
(number of inheritance vectors)*(O(F+N) + O(A) + O(2^A)) = O(PossibilitiesV*(F+N+A+2^A)). Memory complexity of this step is equal to the memory complexity of step 1 = O(F+N) , we just use structures which were initialized in step 1.

So the memory complexity of the whole algorithm is O(F+N) and time complexity is O(PossibilitiesV*(F+N+A+2^A)) = O(2^(A+2*N-F) + (F+N+A)*2^(2*N-F)). By the definition of "O" it can be said that time complexity is O(2^(A+2*N-F)) only, but in practice O((F+N+A)*2^(2*N-F)) can be also meaningful.

External documentation

In order to implement the algorithms which were described in “Algorithms and complexity” part, the following data structures were used:
- childless founders were excluded using linked list “unlinkedFounders”:

```
typedef struct UnlinkedFounder
        {
                int founder; /* index of childless founder in the Pedigree*/
                struct UnlinkedFounder* next;
        }UnlinkedFounder;
UnlinkedFounder* unlinkedFounders;
```

This structure is defined in “gdata.h”, is initiated in “normalize.c” in function “prepareInput” by using array “tmpFounders”. “unlinkedFounders” is used in “NPLstudents.c” in function “getRealFounders” in order to exclude all the childless founders from the computation, and returns the array “RealFounders” which is the shortened version of array “Founders” from which were excluded all childless founders(simply, “RealFounders” = “Founders” – “unlinkedFounders”).

- all the relevant information (for the computation of NPL scores) for each person in the pedigree is stored in the array “ped” of pointers to “ped_node”s:

```
typedef struct ped_node
{
        struct ped_node* next_af;       /* only for affected individuals:
                                           pointer to the next affected individual
                                           in the pedigree                    */
        int mother;                     /* mother index (starting 0, in founder -1)*/
        int father;                     /* father index (starting 0, in founder -1)*/
        Bool af;                        /* false - not affected; true - affected   */
        int allele1;                    /* allele1 - from father; allele2 - from mother;
        */
        int allele2;                    /* two of 2*f alleles {0,1,...,2*f-1}          */
        Bool usefull;                   /* shows if the person is usefull in the computation
                                           of NPL
                                          (meaning he is affected or has affected children) */
}ped_node;
ped_node** ped;
```

This structure is defined in “NPLstudents.h” and is initiated in “NPLstudents.c” in the function “InitPed” using 3 arrays: “RealFounders”, “NonFounders” and “Ped” (=AllPedigreesList->PedigeeGraph). Usage of “RealFounders” and “NonFounders” is essential for topological sorting of all the persons in “ped”, so all the indexes of parents are smaller than indexes of their children in “ped”, it is critical in the following computations.

- all the affected individuals are pointed from the array “affected”, this is a fast way to access affecteds (is used in many functions):

```
ped_node** affected;
```

is defined in “NPLstudents.h” and initiated in “NPLstudents.c” in the function “InitAffected”; also all the affected individuals are linked together by the “next_af” pointers of “ped_node”s (of affected individuals), the last affected points to NULL. The size of “affected” array (and of the linked list of affected individuals):

```
int af_num;
```

- in the computation of Sall score we pick up collections of alleles from affected individuals, 1 allele from each affected, so we need to store the state of collection, t.m. for each affected which of 2 alleles is chosen, we use array:

```
int* aff;
```

of size “af_num” (aff[i] = 1 or aff[i] = 2 depending on which allele of affected “i” is chosen). “aff” is defined in “NPLstudents.h”, initiated in “NPLstudents.c” in the function “InitAffected”, and is used in functions “getSall” and “getSallTmp”.

- in the computation of Sall and Spairs scores we need to count the number of times each founder allele (out of 2*f founder alleles) appears in some collection (for Sall) or between pairs of distinct affected individuals (for Spairs), for this purpose we use:

```
int* f_alleles;
```

is defined in “NPLstudents.h” and initiated in “NPLstudents.c” in the function “InitFounderAlleles”, is used in “getSall”, “getSallTmp” and “getSpairs” functions.

- we also need to store each specific inheritance vector for which Spairs and Sall scores are computed; each cell in this vector is “responsible” for some allele (paternal or maternal) in some individual:

```
typedef struct vector_node
{
        int inheritance;                /*      grandpaternal(0) or grandmaternal(1) */
                                        /*      for the current allele               */
        Bool usefull;                   /*      like in ped_node                     */
        int person;                     /*      index of the person in "ped"         */
        int parent;                     /*      current allele inherited from father(0)
        */
                                        /*      or mother(1)                         */
}vector_node;
vector_node* Ivector;
```

This structure is defined in “NPLstudents.h” and initiated in “NPLstudents.c” in the function “InitVector” (inheritance in each cell is initiated to 0, the size of “Ivector” is 2*n-f), is used in the function “getNewScores”.

- 3 variables are used to store the result of computation for each inheritance vector:

```
double Spairs;
double Sall;
double SallTmp;
```

The meaning of 2 first variables is obvious. “SallTmp” stores the result of [multiplying factorials of number of times each founder allele appears in some collection], while “Sall” gets the sum of “SallTmp”s for all collections.
These variables are in “NPLstudents.h” and are used in “NPLstudents.c” in functions for computing NPL scores.

- the result of computation is stored in 2 arrays of size 2*n-f:

```
double* SpairsArray;
double* SallArray;
```

- the iteration through inheritance vectors is done in “gray code” order (and it is not the natural order of iteration), so for each new vector when the NPL score is computed, we need to compute also to which cell in “SpairsArray” or “SallArray” we should store the result, depending on vector that is treated. This can be done in O(1) if we store the index of previous cell (which was filled for the previous vector), the index of bit which has changed in vector, and array of size 2*n-f with powers of 2 stored (for example, if the size is 4, then the array is {1,2,4,8}). The index of the previous cell and the array:

```
int next;
int* powersOf2;
```

(is called “next”, because it gets also the index of the next cell to fill)

are defined in “NPLstudents.h”. “powersOf2” is initiated in “NPLstudents.c” in the function “computeScores”.
The index of bit to be changed in vector is computed in each iteration by the function “compute_algorithm” (defined in “sequence.h” and “sequence.c”), which passes this index to “getNewScores” (where the computation of a new “next” is performed, using this index, “next” and “powersOf2”).

Integration of the code into original

1) Add files NPLstudents.h, NPLstudents.c, sequence.h, sequence.c to the project
2) In file gh_main.c
   - add: *#include "NPLstudents.h"*
   - modify function "main()" by adding between existing lines "createTestPoints();" and "likehood();"
   the new line *computeNPL();*
3) In file gdata.h
   - add: *typedef enum {S_PAIRS, S_ALL} SCORE;*
   - add the definition of *unlinkedFounders;*
4) In the file normalize.c modify function "prepareInput()" to create linked list *unlinkedFounders;*
5) In the file likehood.h add definition: *double sumVector(VECTOR Vector);*
6) In the file likehood.c:
   - add: *#include "NPLstudents.h"*
   - add implementation of *double sumVector(VECTOR Vector);*
   - modify function likehood() to perform the multiplication of multipoint probabilities vectors with *SpairsArray* and *SallArray* , and free these 2 arrays in the end of the function.
7) In the file programSpec.h
   - add: *#include "NPLstudents.h"*
   - in the definition of the struct *TestPoint* add: *double NPL_Spairs;*
   double NPL_Sall;
8) In the file programSpec.c modify function "printTestPoints()" to print NPL scores Spairs and Sall.

Performance results and comparison between Superlink and Genehunter v2.1

Files	People num.	Loci num.	Result files	Run time (LOD + NPL Spairs, Sall), sec.	Run time Gene Hunter v2.1, sec.
datafile3.dat pedfile30.dat	7	6	results30.txt	0.0	0.0
datafile3.dat pedfile32.dat	7	6	results32.txt	0.0	0.0
datafile4.dat pedfile42.dat	7	6	results42.txt	0.0	0.0
datafile4.dat pedfile43.dat	7	6	results43.txt	0.0	0.0

datafile5.dat pedfile50.dat	15	9	results50.txt	0.0	0.2
datafile5.dat pedfile51.dat	15	9	results51.txt	1.0	1.8
datafile5.dat pedfile52.dat	15	9	results52.txt	0.1	0.5
datafile5.dat pedfile53.dat	15	9	results53.txt	0.1	0.5
datafile5.dat pedfile54.dat	15	9	results54.txt	11.3	10.9
datafile6.dat pedfile60.dat	15	99	results60.txt	0.8	2.6
datafile6.dat pedfile61.dat	15	99	results61.txt	11.3	24.0
datafile6.dat pedfile62.dat	15	99	results62.txt	0.0	0.3
datafile6.dat pedfile63.dat	15	99	results63.txt	0.7	2.5
datafileGB_27_0.dat pedfileGB_27_0_0.dat	15	1	results27_0.txt	1.3	1:00.8
datafileGB_27_1.dat pedfileGB_27_1_0.dat	15	3	results27_1.txt	2.6	1:05.9
datafileGB_27_2.dat pedfileGB_27_2_0.dat	15	5	results27_2.txt	3.9	1:08.3
datafileGB_27_3.dat pedfileGB_27_3_0.dat	15	7	results27_3.txt	4.8	1:11.9
datafileGB_67_0.dat pedfileGB_67_0_0.dat	15	1	results67_0.txt	35.4	59.8
datafileGB_67_1.dat pedfileGB_67_1_0.dat	15	3	results67_1.txt	37.1	1:04.4
datafileGB_67_2.dat pedfileGB_67_2_0.dat	15	5	results67_2.txt	38.0	1:06.9
datafileGB_67_3.dat pedfileGB_67_3_0.dat	15	7	results67_3.txt	29.5	1:08.9
datafileGB_90_0.dat pedfileGB_90_0_0.dat	15	1	results90_0.txt	6.5	1:00.2
datafileGB_90_1.dat pedfileGB_90_1_0.dat	15	3	results90_1.txt	7.7	1:01.8
datafileGB_90_2.dat pedfileGB_90_2_0.dat	15	5	results90_2.txt	8.5	1:03.6
datafileGB_90_3.dat pedfileGB_90_3_0.dat	15	7	results90_3.txt	9.4	1:05.8

All the input files (datafile?.dat and pedfile?.dat) and output files (results?.txt) are situated in archives “input.zip” and “output.zip” in “Downloads” section.

Ideas for improvement.

The general idea of the implementation of NPL analysis in this project is to iterate over inheritance vectors in the outermost loop and over alleles in the inner loop: for each new inheritance vector possibly there are O(F+N) alleles to rearrange, computation of Spairs takes O(A) and computation of Sall – O(2^A) [F – number of founders, N – number of nonfounders, A – number of affected]. Overall number of inheritance vectors is O(2^(2*N-F)), so the computation of Spairs for all vectors takes O((F+N+A)*2^(2*N-F)) and the computation of Sall – O((F+N+2^A)*2^(2*N-F)), as it was already mentioned in complexity analysis.

But this may be reduced to O(2^(2*N-F)) for Spairs and to O(2^(A+2*N-F)) for Sall, if we will use another principle: not to iterate over vectors in the outermost loop, but to “build” vector bit by bit and for each new bit of vector to compute new score using score (either Spairs or Sall) of the previous “shorter” vector (and without taking into consideration the rest of the bits that will be added to the vector later), when we start with vector of length 1 and compute initial score assuming only 1 bit. This principle avoids duplicated calculations for vectors that differ only for branches in the pedigree.

If the previous vector was v, we added some bit to v and got v’, then we can calculate new scores in the following manner (assuming that the previous scores were Spairs(v) and Sall(v)):

> <u>Spairs(v’)</u>: let’s assume we have array of 2*F alleles “f_alleles” in which for each founder allele j we store number of times allele j appears in pedigree in different affected individuals assuming vector v; if the added bit corresponds to allele i in some affected individual then: Spairs(v’) <– Spairs(v) + f_alleles[i] ; f_alleles[i]++;
> this simple calculation takes O(1).
> <u>Sall(v’)</u>: this case is much more complicated, because in order to use the previous score we need the scores for all collections (for each collection h we need to store ∏bi(h)! – multiplication of factorials of number of times each allele i appear in h), and not only that, for each collection h we need also array “f_alleles” (for each allele j storing the number of times j appears in h), that means memory complexity increases up to O(2*F*2^A); if we have all this and the added bit corresponds to allele i in some affected individual then we loop over all collections and for each collection h’ we use the score of corresponding collection h (from vector v) and array f_alleles of that collection and get: f_alleles[i]++; ∏bi(h’)! <- (∏bi(h)!)*f_alleles[i];
> we take sum over all new scores of the collections (dividing of sum by 2^A is performed only at the last step);
> for each collection O(1) operations are performed, so computing the new Sall score takes O(2^A) time.

The number of inheritance vectors is O(2^(2*N-F)), so we get the desired time complexity. It is not the exact implementation, its only the main idea and perhaps can be improved further (especially memory complexity in the calculation of new Sall score).