Towards A More Consistent and Reproducible Gene Expression Analysis

A thesis submitted by
Chai Haoqiang

in partial fulfilment for the
Degree of Bachelor of Science with Honours
in
Computational Biology

Supervisor: Professor Choi Kwok Pui
Co-Supervisor: Professor Limsoon Wong

Acknowledgement

I wish to express my gratitude to my supervisors, Prof Choi Kwok Pui and Prof Limsoon Wong, for their invaluable guidance and patience throughout the course of this project. I deeply appreciate the time they sacrificed for me.

Contents

Abstract

Contemporary methods of microarray analysis often have a tendency to produce different results from different datasets of the same disease. SNet is a technique that identifies specific connected portions of pathways that are significant for a certain disease, and the portions (termed as subnetworks) are shown to be consistent for different datasets. This project aims to further improve the consistency of SNet by adding two modifications, which are Various Alpha and Neighbour Merge. This project also increases the convenience of SNet by implementing a Graphic User Interface and visualization tool. It was shown that Neighbour Merge improved the consistency of SNet significantly, while Various Alpha did not.

Introduction

There are plenty of techniques for identifying significant differential gene expression. These techniques can be categorized into three approaches; namely, individual genes, gene pathways and gene classes approaches. Individual Gene techniques search for individual genes that are differentially expressed, such as t-test and Significance Analysis of Microarrays (SAM) (Tusher *et al.*, 2001). Gene Pathway techniques try to produce a list of gene networks solely from the analysis of the gene expression data, without using pre-existing biological information. Examples are Bayesian learning (Friedman *et al.*, 2000) and Boolean network learning (Lahdesmaki *et al.*, 2006). Gene Classes techniques test how gene classes behave as a whole. These techniques normally produce a list of pathways or gene groups based on both analysis of gene expression data and existing biological background knowledge. The commonly acknowledged challenge of these techniques is obtaining reproducible results; in other words, when applying these methods on different datasets for the same disease, the gene lists or gene group lists produced by same algorithm has little overlap for different datasets.

For this purpose, a new algorithm SNet was developed by Soh *et al.* to identify subnetworks which are expressed significantly within a phenotype of a microarray experiment (Soh *et al.*, ~~2012)~~. Carefully conducted experiments demonstrate that this technique shows greater consistency and hence reproducibility.

This paper presents the re-implementation of SNet and a continual investigation and development on SNet. Two modifications were applied on SNet algorithm, which are Various Alpha and Neighbour Merge, to achieve a more consistent and reproducible algorithm. On top of that, by implementing a Graphic User Interface (GUI) and

incorporating a visualization module, the SNet technique is made less complicated and more easily manageable. It was shown that Neighbour Merge increased the significance and reproducibility of SNet in the way of producing subnetworks with larger sizes and higher consistencies among different datasets, while Various Alpha did now show a significant improvement on SNet.

Methods and Materials

SNet Algorithm

This section reviews the SNet algorithm briefly.

Overview

The phenotype of interest is labelled as *d* and the remaining phenotypes are labelled as *d'*. First of all, a list of genes that are highly expressed within phenotype *d* is generated from the microarray experiment. This list of genes is then mapped onto all the pathways of human identified so far. A list of subnetworks *cc* (whose genes are highly expressed in phenotype *d*) is obtained by selecting the connected components. Next, a score (depending on the significance of its genes and its consistency among the patients) is calculated and assigned to each subnetwork. Finally the p-value is estimated for every single subnetwork within the list and only those with significant p-values are kept.

This process is elaborated in greater details in the following steps.

Step1: Subnetwork Extraction

For each patient within a phenotype, a ranked list of genes is selected according to the gene expression level in that patient. Then top α% of the genes from this list are

selected for each patient. This condensed gene list is referred to as GP_i for the i^{th} patient P_i. Next, the genes which appear in the GP_i for more than β% of all the patients with phenotype *d* is selected by iterating across gene list GP_i. This creates a list of genes GL which turns up highly expressed across most of the patients of phenotype *d*. Finally, using the programmatic interface of PathwayAPI (Soh *et al.*, 2010), gene list GL is segregated into the respective subnetworks. In this project, α is taken to be 10 and β is taken to be 50.

To segregate GL into the different subnetworks, firstly the genes in the gene list GL are mapped on all the pathways. (It's been highlighted that a gene is allowed to appear in more than one pathway). Next, by treating each gene as a vertex and each gene-gene relationship as an edge, the connected components (subnetworks) in each pathway formed by these edges (gene-gene relationships) and vertices (genes) are located in each pathway. This process is illustrated in the Figure 1 below.

As shown in the Figure 1, firstly a gene list was generated. Then it was mapped onto a pathway. After removing genes that do not belong to the gene list, two subnetworks one and two were generated.

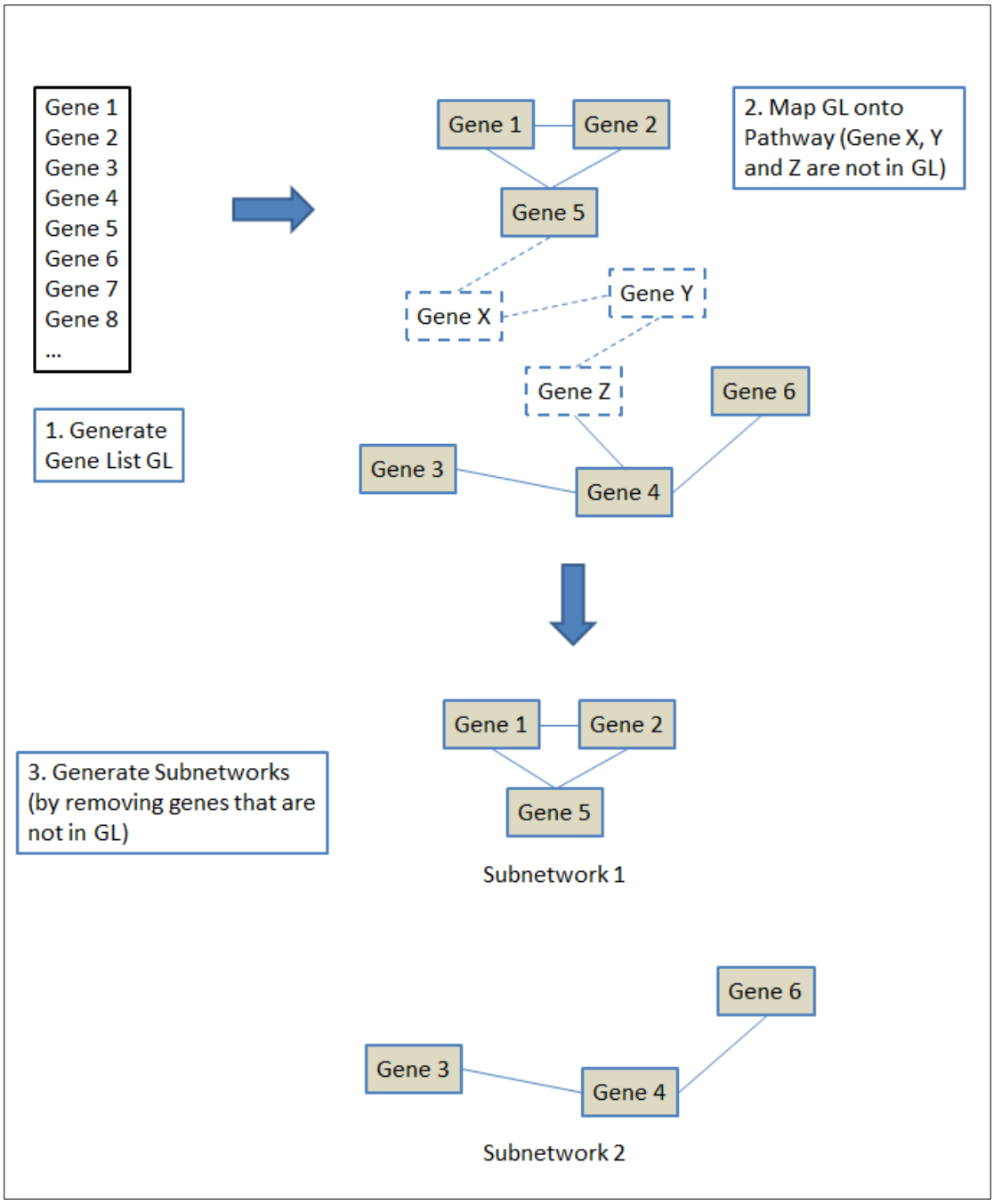

Figure 1 SNet Overview

Step 2: Subnetwork Scoring

For each subnetwork *sp* within *cc* and for each patient P_i (regardless of phenotype), the overall expression level of *sp* in P_i is computed by

$$\mathrm{SNet}_{\mathrm{sp,i}} = \sum_{\mathrm{g}\in \mathrm{G}_{\mathrm{Pi}\cap \mathrm{sp}}} \mathrm{Sg}_{\mathrm{sp,g}}, \text{where } \mathrm{Sg}_{\mathrm{sp,g}} = \mathrm{k/n}$$

Here, *g* denotes a gene in the subnetwork *sp* that is highly expressed (top α%) in patient P_i; *k* is the number of patients of phenotype *d* who have gene *g* highly expressed; and *n* is the total number of patients of phenotype *d*.

Let $P_1 \ldots P_n$ be patients of phenotype *d*; and $P_{n+1} \ldots P_m$ be patients of other phenotypes d'. Two score vectors $\mathrm{Ssp}_{\mathrm{sp,d}}$ and $\mathrm{Ssp}_{\mathrm{sp,d'}}$ are assigned respectively for these two groups of patients, where

$$Ssp_{sp,d} =< SNet_{sp,1}, \ldots, SNet_{sp,n} >, Ssp_{sp,\neg d} =< SNet_{sp,n+1}, \ldots, SNet_{sp,m} >,$$

The t-statistics is now calculated between these two vectors, creating a final score for each subnetwork *sp*.

Step 3: Subnetwork Significance

The significance of the observed subnetworks is estimated by randomly permuting the phenotypes labels of two phenotypes *d* and *d'*, re-generating the subnetworks and re-computing their t-statistics scores. This generates a null distribution for the score of the subnetworks. The p-value of each subnetwork is then calculated relative to this null distribution. The detailed procedure is as follows:

A. Assign each patient a new phenotype label randomly with the probability distribution proportional to the original phenotype ratio. Re-generate the subnetworks and re-compute their t-statistic scores.
B. Repeat [A] for 1000 permutations. This creates a two dimensional histogram of the scores and sizes of the subnetworks.

C. Estimate the nominal p-value of each subnetwork by using the histogram created in point [B].

Finally, the subnetworks whose p-value was sufficiently small (<= 0.05) is considered to be significant. This would provide an independent set of significant subnetworks *SN* for each dataset. For the same disease, significant subnetworks *SN* are then compared across different datasets, thereby to show the consistency of SNet algorithm.

Various Alpha

This section describes the first modification on SNet algorithm. It was inspired by the thought that original α (=10%) was not guaranteed to be the most suitable coefficient. It would be more reasonable to adjust α to different values, and then decide which α to use by comparing the different result. One intuitive approach to this adjustment is to repeat the SNet algorithm with different α. However, the problem of this approach is that the subnetworks list generated by different α are different; in other words, it is not possible to map each subnetwork in the new list to any other subnetworks in the original list. Thereby changes made to specific subnetworks cannot be tracked. Hence, an alternative approach was used to carry out this modification, which is to evaluate the significance of different α by modifying the original subnetwork list.

This approach is illustrated in greater details as the following steps.

Step 1: Original Subnetwork List Generation

Define α_0 as original α, and use it to carry out SNet algorithm. This will provide a list of significant subnetworks *sp* on α_0. In this project a_0 is set to be 10%.

Step 2: Generate New Gene List GL1 with New α

Define another α called α_1, which is slightly larger than the original α_0. Repeat first part of Step 1 of SNet algorithm with $\alpha = \alpha_1$. This creates a list of genes GL_1 which turns up highly expressed across most of the patients of phenotype *d*.

Step 3: Subnetworks Modification

For each subnetwork SN in original subnetwork list *sp*, check each of its immediate neighbours *ni* (genes that are one edge away from SN). If *ni* belongs to GL_1, then add *ni* to this subnetwork. The detail of this process is shown in the Figure 2 below.

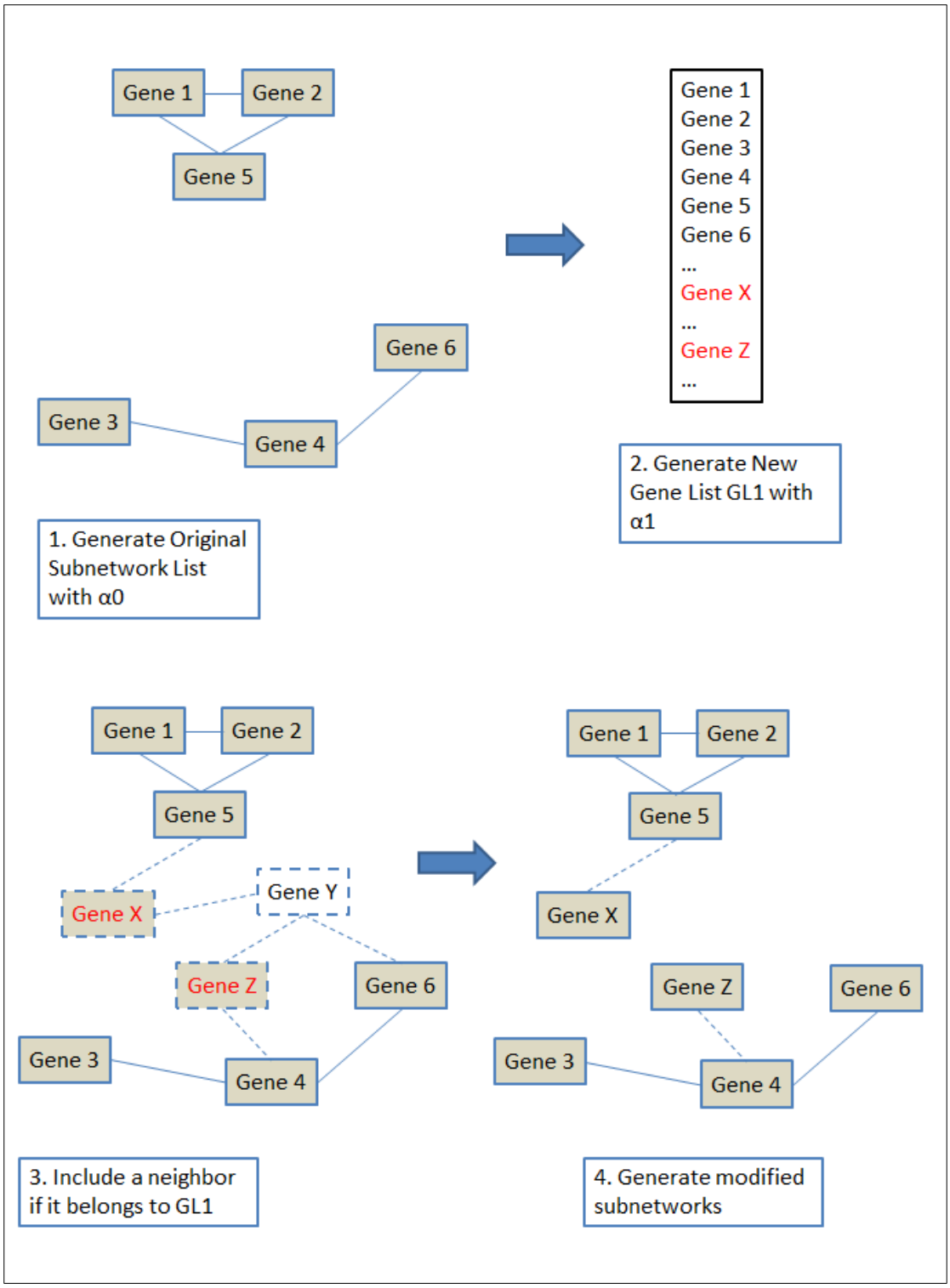

Figure 2 Various Alpha Algorithm Overview

Step 4: Subnetworks Significance

Use the null distribution generated in Step 3 of SNet to calculate the *p-value* of each modified subnetwork.

Merge Neighbouring Subnetworks

This is the second modification to SNet algorithm. This modification was based on the thought that parameter α was set to be too strict, there may exist such genes that they're not in the gene list GL, but due to their absence from the list, some significant subnetworks are separated into smaller subnetworks and hence no longer significant. The solution to this problem is to check subnetworks pairwise, if they are very close, then merge them as a new subnetwork and add the gene that joined them into new subnetwork.

This approach is illustrated in greater details as follows.

Step 1: Subnetworks Generation

Perform SNet on a dataset. This generates a list of subnetworks.

Step 2: Detect and Merge Neighbouring Subnetworks

Two subnetworks are defined as neighbouring if they can be connected by one gene. As shown in the Figure 3, subnetwork *one* and *three* are separated by only one gene *Gene W*, hence they're merged into a new subnetwork *one&three*. While Subnetwork *two* and *three* are separated by two genes *Gene X* and *Gene Y*, hence they are not merged.

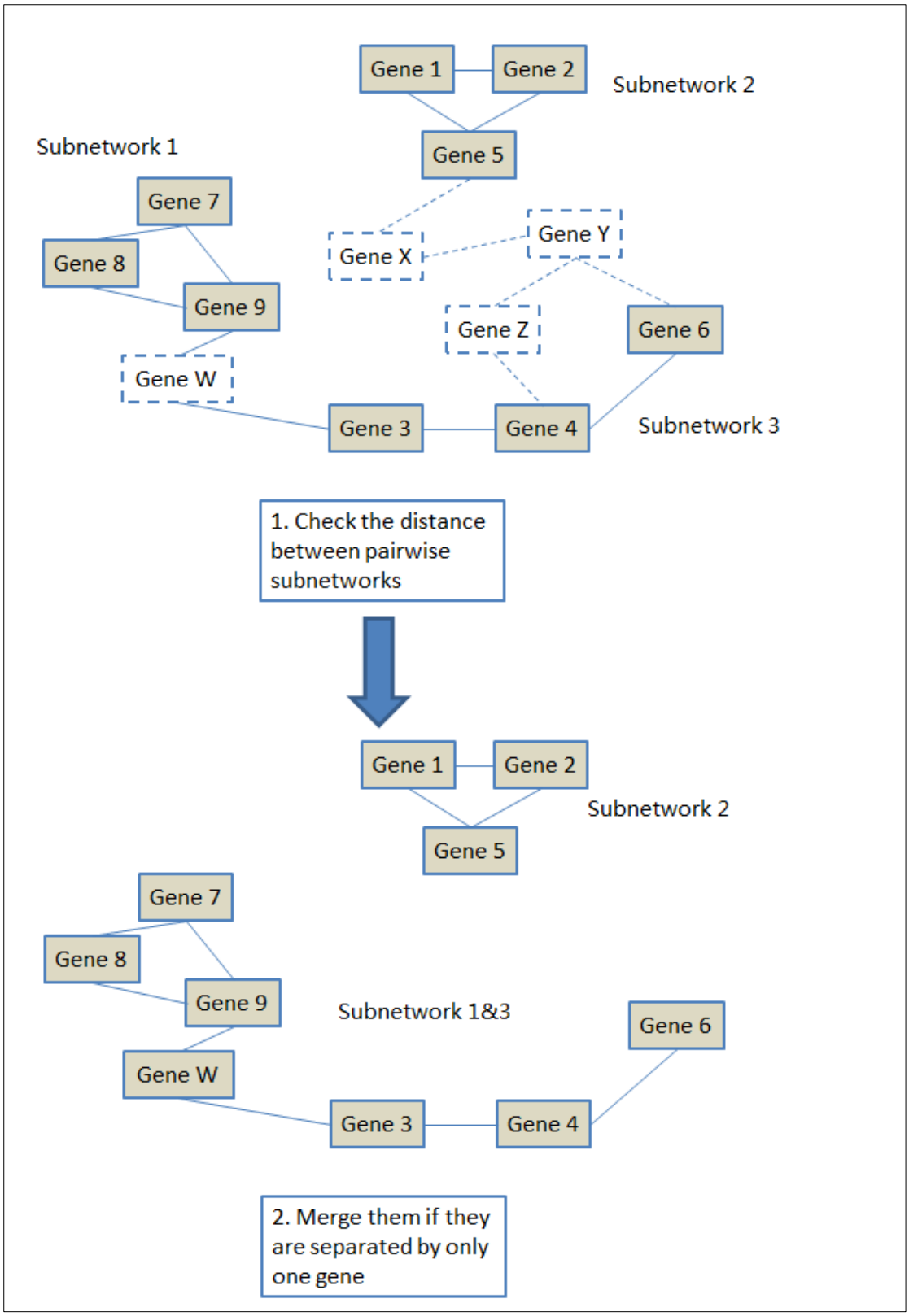

Figure 3 Neighbour Merge Algorithm Overview

Step 3: Subnetwork Significance

The null distribution generated in Step 3 of SNet is used to calculate the *p-value* for each modified subnetworks.

Overlap Checking

After generating a list of subnetworks for each of the two datasets, we then checked the overlap percentage of two subnetwork lists, thereby to evaluate whether the algorithm can produce consistent subnetworks among different datasets of same diseases. Since a subnetwork is a group of genes, thus it is not feasible to define two subnetworks as the same directly. Hence it is not feasible to calculate overlap of two subnetwork lists directly. Thereby three indirect methods were used to assess the overlap of subnetwork lists in different perspectives, which are Gene Overlap, Pathway Overlap and Subnetwork Overlap.

Gene Overlap

Gene overlap is to calculate the overlap of genes of two subnetwork lists. The process is done as follows:

Step 1: For each of the two subnetwork lists, retrieve all the genes of each subnetwork and add them into a list of genes called *GeneList(i)* (i=1,2);

Step 2: Remove duplications in the *GeneList(i)* (i=1,2);

Step 3: The overlap percentage will be calculated as overlap of two gene lists divided by the number of genes of the smaller gene list as follows:

$$\text{overlap} = \frac{(|\text{GeneList}(1)| \cap |\text{GeneList}(2)|)}{\min(|\text{GeneList}(1)|, |\text{GeneList}(2)|)}$$

Pathway Overlap

Pathway overlap is to calculate the overlap of pathways that each subnetwork list contained. It's calculated as follows:

1/ For each of the 2 subnetwork lists, generate a list of pathways which records all the pathway numbers that each subnetwork belongs to, call it as *PathwayList(i)* (i=1,2);

2/ Calculate the overlap of two pathway lists using the same method for calculating gene overlap, which is as follows:

$$\text{overlap} = \frac{(|\text{PathwayList}(1)| \cap |\text{PathwayList}(2)|)}{\min(|\text{PathwayList}(1)|, |\text{PathwayList}(2)|)}$$

Subnetwork Overlap

As mentioned in the beginning of this section, it is not feasible to define two subnetworks as the "same", since significant subnetworks are often composed of several genes, and a single gene difference will result two subnetworks as different. For example, both subnetwork A from *subnetwork list 1* and B from *subnetwork list 2* contains 7 genes, and they share 6 common genes. Although there is not an exact match between the two subnetworks, a 6/7 match indicates that they have a high similarity, as a single mismatch maybe caused by experimental errors or threshold choosing. Hence, such a strict approach is not a good way to check subnetwork overlap since it filtered quite a number of significant overlap. To improve it, a new approach for checking subnetwork overlap is introduced as below.

In this approach, two definitions were introduced, which are **perfect match** and **partial match**.

Two subnetworks are said to be **perfect match** if most of their genes are the same, and they contain roughly the same number of genes. In the case above, subnetwork A and B will be perfect match.

Sometimes due to missing some important genes, a big subnetwork is divided into two or more smaller subnetworks in the other subnetwork list; then this subnetwork cannot find a perfect match in the other subnetwork list. Partial match solves this problem. A subnetwork is **partial match** to a subnetwork list, if most of its genes can be found in the Gene List generated by this subnetwork list.

In the Figure 4 below, subnetwork A from *subnetwork list 1* and subnetwork C from *subnetwork list 2* are perfect match; while subnetwork B is only partial match to *subnetwork list 2* since *Gene 4* is missing in *subnetwork list 2*, and hence B is divided into two smaller subnetworks.

The standard definition of perfect match is as follows:

Subnetwork A and B are said to be the **perfect match** if:

$$\frac{|\text{A. genes} \cap \text{B. genes}|}{|\text{A. genes}|} > \gamma_1 \; and \; \frac{|\text{A. genes} \cap \text{B. genes}|}{|\text{B. genes}|} > \gamma_1$$

Subnetwork A is **partial match** to a subnetwork list C if:

$$\frac{|\text{A. genes} \cap \text{C. genes}|}{|\text{A. genes}|} > \gamma_2$$

In this project, $\gamma_1 = \gamma_2 = 75\%$.

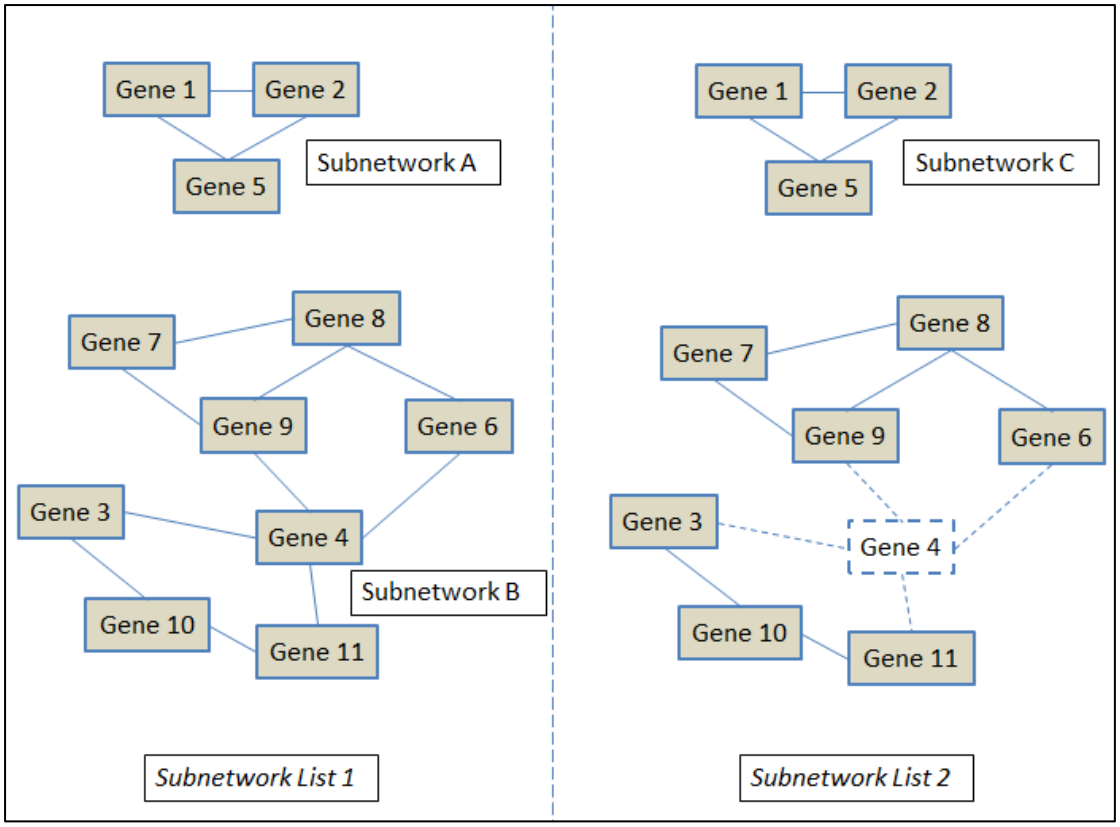

Figure 4 Perfect Match and Partial Match Overview

Through evaluation methods above, we can evaluate the similarity of two subnetwork lists, and thereby evaluate the consistency of our methods across different datasets.

Visualization

A visualization component was implemented to SNet, with the aid of Cytoscape software. It allows the users to visualize the subnetwork list been generated and to compare the subnetworks after modification with the original subnetworks.

The visualization is realized by setting subnetworks as node attributes and gene interactions as edge attribute respectively, and then apply the attributes on the pathway network.

Graphic User Interface

Graphic User Interface was employed to integrate all the functions mentioned above- the original SNet algorithm, Various Alpha algorithm, Neighbour Merge algorithm and visualization- into a software package, and allow future researchers to access these methods in an easier and more convenient way.

The program was implemented in C# using Microsoft Visual Studio 2010, with MySQL as database platform. The User Manual is attached in Appendix A and the source code is attached in the Appendix B. The software can be downloaded from the link provided in the User Manual.

Below is a snapshot for software user interface.

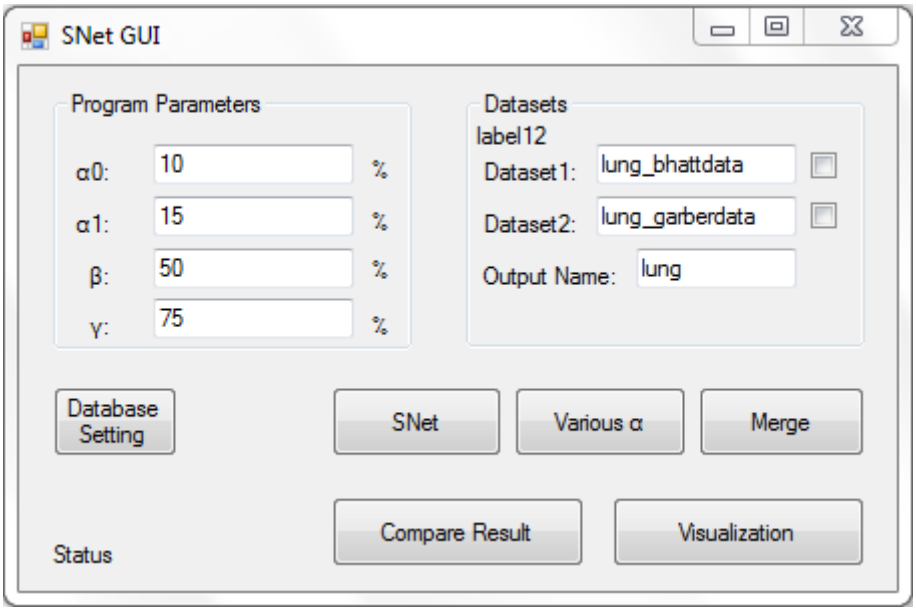

Figure 4 SNet GUI

Materials

Pathways sources were retrieved from PathwayAPI (Soh *et al.*, 2010) which offered the combined pathway information of KEGG, Ingenuity and Wikipathways along with a modern JSON-based application programming interface. PathwayAPI was dumped into the local MySQL database and then retrieved by C# using MySQL connector.

The testing datasets used were the same as original SNet project. Four disease types were used. Each disease contains two datasets generated by two different platforms. The selection of the two datasets for each disease is to check the overlap of results thereby to validity the algorithm. The details of each datasets are as follows:

- Duchenne Muscular Dystrophy (shorted as DMD): Comparison between patients suffering from DMD and normal patients. Haslett (Haslett *et al.*, 2002) uses the Affymetrix HG-U95Av2 GeneChip while Pescatori (Pescatori *et al.*, 2007) uses HGU133A GeneChip. Haslett contains 24 samples from 12 DMD patients and 12 unaffected controls and Pescatori consists of 36 samples from 22 DMD patients and 14 controls.
- Childhood Acute Lymphoblastic Leukaemia (shorted as ALL) Subtype: Comparison between two subtypes of childhood ALL leukaemia, E2A-PBX1 and BCR-ABL. Mary (Ross *et al.*, 2004) uses the Affymetrix HG-U95Av2 GeneChip with 15 BCR-ABL patients and 27 E2A-PBX1 patients. Yeoh *et al.* (2002) uses the U133A GeneChip with 15 BCR-ABL patients and 18 E2A-PBX1 patients.
- Lung Cancer (shorted as Lung): Comparison between patients suffering from squamous cell lung carcinomas and normal patients. For lung cancer, the

cDNA microarray data consisted of 13 samples with squamous cell lung carcinomas and five normal lung specimens (Garber *et al.,* 2001), while the data by Affymetrix human U95A oligonucleotide arrays consist of 21 squamous cell lung carcinomas and 17 normal lung specimens (Bhattacharjee *et al.,* 2001).

- Prostate Cancer (shorted as Prostate): Comparison of patients suffering from prostate cancer and normal patients. The cDNA microarray data consist of 62 tumors and 41 normal prostate samples (Lapointe et al., 2004), while the oligo microarray (Affymetrix U95Av2) data contain 52 tumor and 50 non-tumor samples (Singh et al., 2002).

The original data for each dataset is in *csv* format, and most of them only contain *probe_set_id* but lack of a universal gene name for each gene. Hence, a mapping table for each platform is downloaded from (HgU133Plus2, 2003), and a universal gene name was given to each gene before the datasets is dumped into database.

Data Pre-processing

For each disease, two datasets were processed by removing genes that do not appear in all the samples or don't have a gene name.

Results

In this section, the performance of SNet, Various Alpha and Neighbour Merge algorithm will be assessed in the following perspective: total number of significant subnetworks generated, significant subnetworks' size and overlap ratio of different datasets.

Number of Subnetwork Generated

Table 1 describes total number of significant subnetworks generated by SNet, Various Alpha and Neighbour Merge. Chart 1 presents it in chart format. From Chart 1, it is shown that in some cases Various Alpha produces more subnetworks than SNet, but in other cases it produces less; while for Neighbour Merge, it consistently produces fewer subnetworks than SNet in all datasets.

	SNet	Various Alpha	Neighbour Merged
DMD-has	469	403	395
DMD-pec	567	464	387
Lung-bha	873	894	740
Lung-gar	900	805	862
Prostate-lap	2168	2313	1977
Prostate-sin	1745	1923	1475
ALL-all	703	735	544
All-mar	507	616	473

Table 1 Total Number of Subnetworks Generated by SNet, Various Alpha and Neighbour Merge

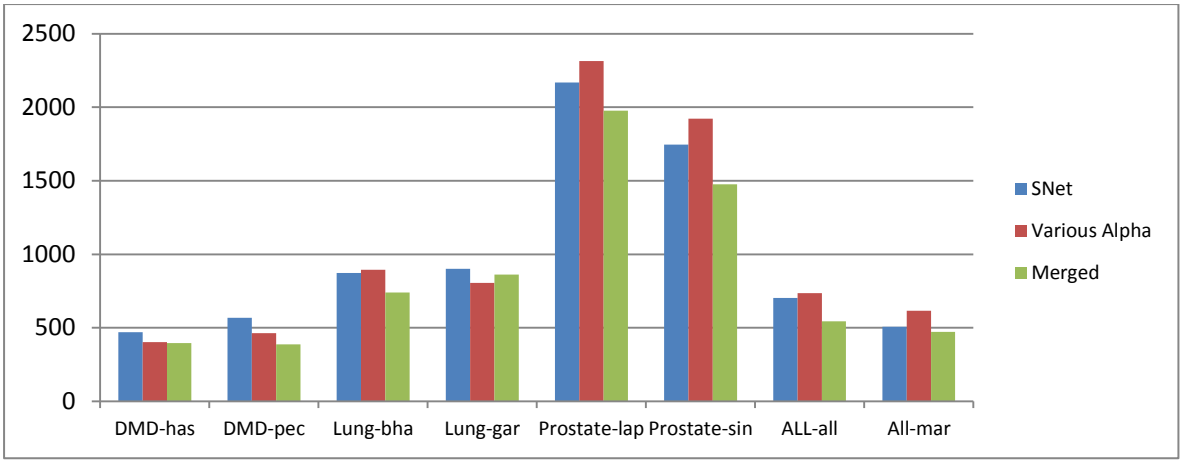

Chart 1 Total Number of Significant Subnetworks Generated by SNet, Various Alpha and Neighbour Merge

Subnetwork Size

The sizes of subnetworks generated by different algorithms are then compared. The sizes are segregated into 10 intervals according to the distribution of sizes: 1, 2-5, 6-10, 11-20, 21-30, 31-40, 41-50, 51-60, 61-70 and 70 onwards. Chart 2.1 – 2.8 depict size distribution of subnetworks generated from each dataset. Since the majority of subnetworks are of size 1, and listing them in the graph will make other interval unrecognizable, hence subnetworks of size 1 are neglected in the charts.

From these charts, it was shown that Neighbour Merge produces more subnetworks than SNet in most of intervals. Particularly it produces more extra-large subnetworks, whose sizes are more than 60. While for Various Alpha algorithm, it consistently produces more subnetworks in the small size range 2-5, but produces less large subnetworks.

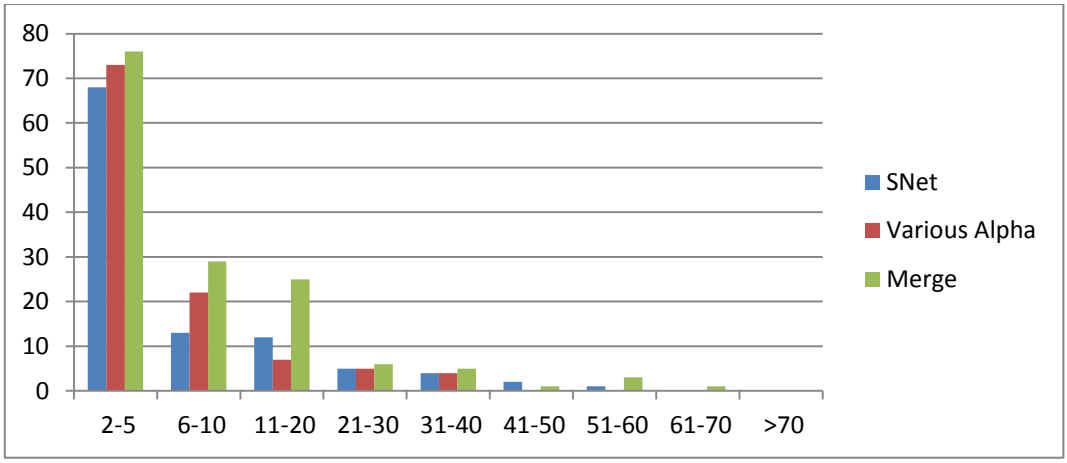

Chart 2.1 Subnetworks Size Distribution for DMD-Haslett

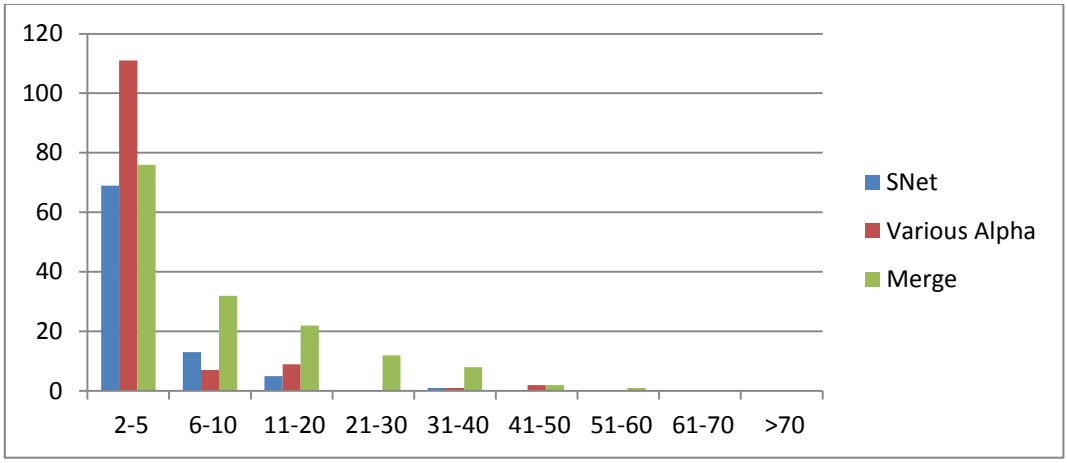

Chart 2.2 Subnetworks Size Distribution for DMD-Pescatori

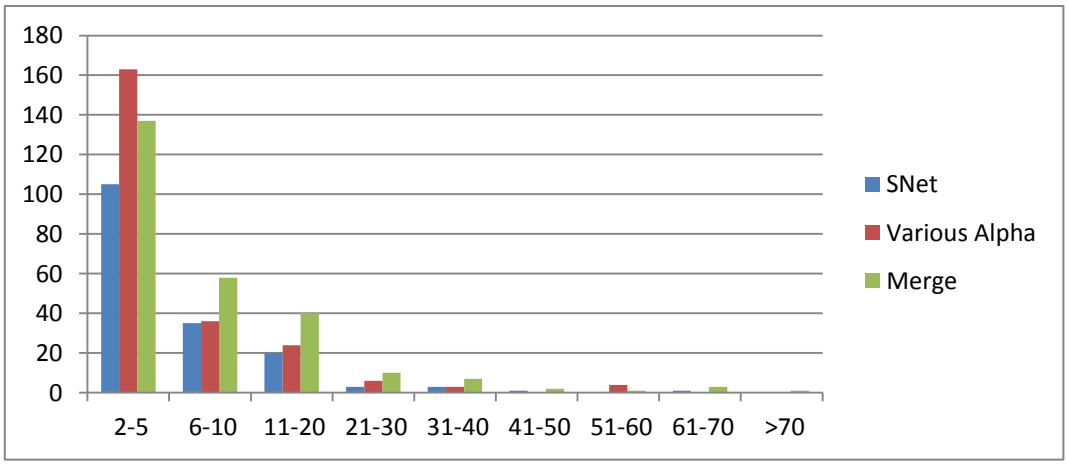

Chart 2.3 Subnetworks Size Distribution for Lung-Bhatt

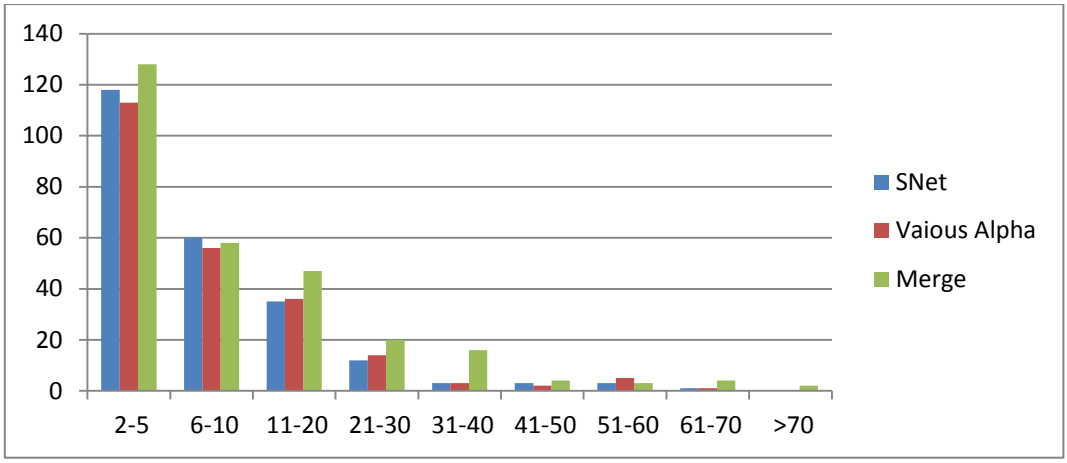

Chart 2.4 Subnetworks Size Distribution for Lung-Garber

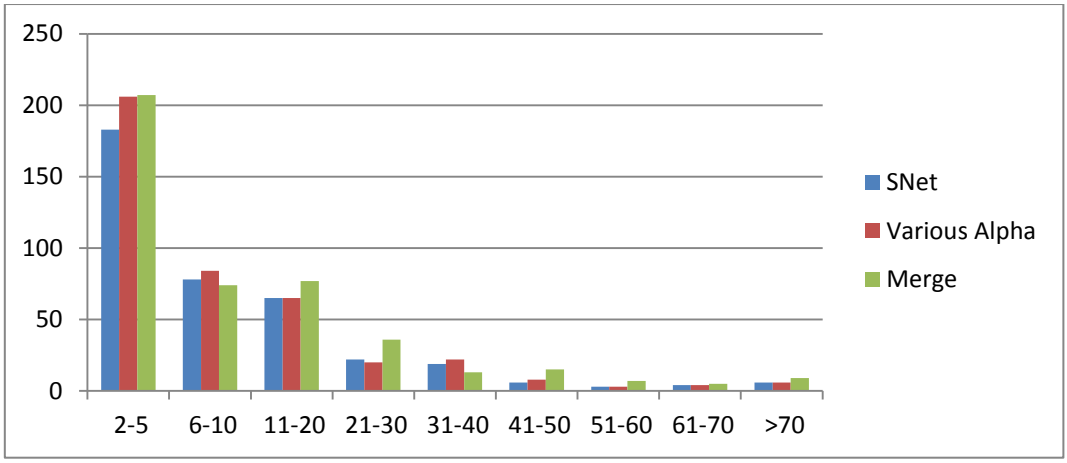

Chart 2.5 Subnetworks Size Distribution for Prostate-Lapointe

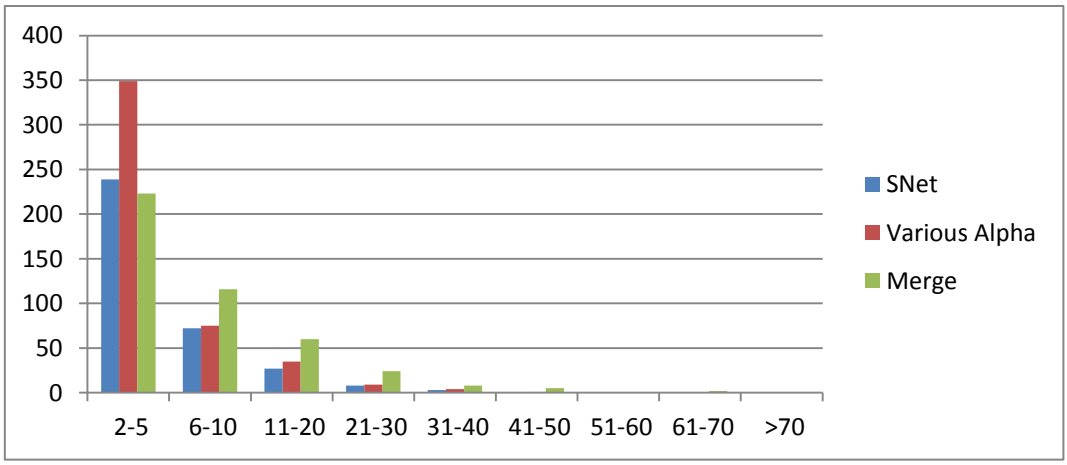

Chart 2.6 Subnetworks Size Distribution for Prostate-Singh

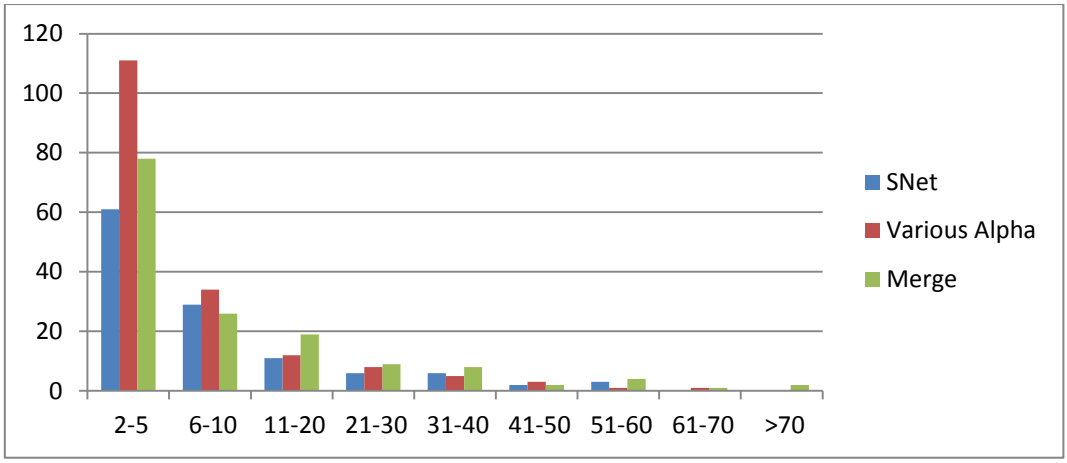

Chart 2.7 Subnetworks Size Distribution for All-Allen

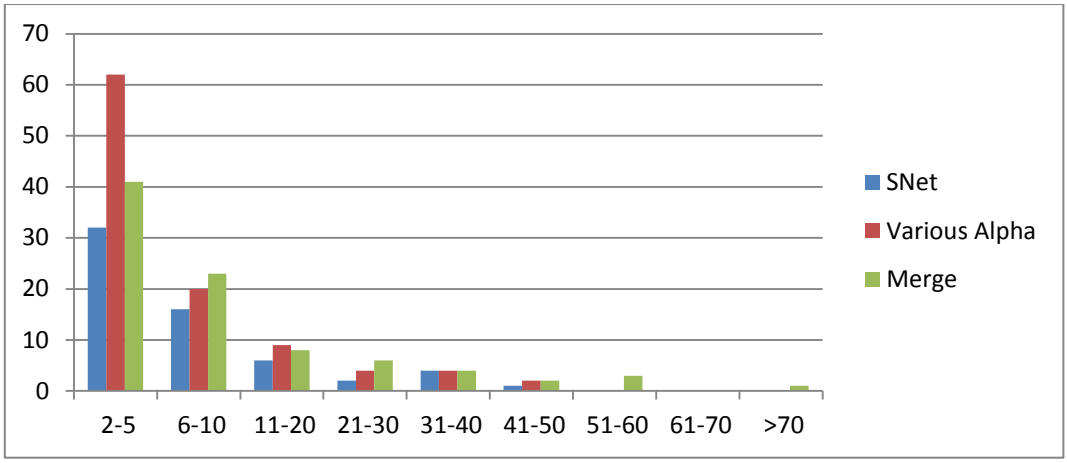

Chart 2.8 Subnetworks Size Distribution for All-Mary

Overlap between two datasets

Gene Overlap

The gene list overlap is shown in the Table 2 below. As shown in the table, Neighbour Merge consistently produces higher overlap than the original SNet algorithm, while the performance of Various Alpha varied vastly across different datasets.

	SNet	Various Alpha	Neighbour Merged
DMD	27%	20%	32%
Lung	25%	26%	29%
Prostate	42%	44%	43%
All	27%	29%	31%

Table 2 Gene Overlap of Subnetworks Produced by SNet, Various Alpha and Neighbour Merge

Pathway Overlap

The pathway overlap is shown in the Table 3 below. In general, the pathway overlaps produced by all three algorithms are much higher than the gene overlaps in all datasets.

As shown in the table, the average overlap of pathways produced by Neighbour Merge is slightly smaller than SNet. Particularly in ALL datasets, overlap produced by Neighbour Merge is 10% smaller than SNet.

For Various Alpha, the overlap percentage only slightly varied from SNet.

	SNet	Various Alpha	Merged
DMD	74%	71%	71%
Lung	83%	84%	83%
Prostate	96%	96%	95%
All	84%	80%	75%

Table 3 Pathway Overlap of Subnetworks Produced by SNet, Various Alpha and Neighbour Merge

Subnetwork Overlap

Table 4 and 5 depicts the number of Perfect Match and Partial Match found within subnetwork lists generated by different algorithms. Unlike gene overlap and pathway overlap, the Perfect Match and Partial Match are presented in the format of number of Matches found, instead of its percentage taken. This is because Perfect Match and Partial Match are not accurate estimating methods, and they can only provide a rough estimation of subnetwork overlaps. The detailed analysis of their accuracy will be discussed in the Discussion section.

As shown in Table 4, the Perfect Matches found in Various Alpha don't show a consistent pattern. Sometimes they are fewer than those found in SNet, but other times they are more.

However, Neighbour Merge consistently produces fewer Perfect Matches than SNet for all diseases tested. It seems that the subnetworks produced by Neighbour Merge are less consistent than SNet.

	SNet	Various Alpha	Neighbour Merged
DMD	89	18	38
Lung	155	170	85
Prostate	1671	1969	1210
All	132	147	90

Table 4 Perfect Match within Subnetworks Lists Produced by SNet, Various Alpha and Neighbour Merge

As shown in Table 5, in most cases (75%) Various Alpha produces more Partial Overlap than SNet. For Neighbour Merge it is the other way around, which in most cases (75%) produces less Partial Overlaps than SNet.

	SNet	Various Alpha	Neighbour Merged
DMD-Has	54	35	64
DMD-Pec	117	78	98
Lung-Bha	227	229	172
Lung-Gar	94	111	121
Prostate-Lap	493	593	474
Prostate-Sin	801	946	643
ALL-All	144	182	112
ALL-Mar	89	130	84

Table 5 Partial Match within Subnetworks Lists Produced by SNet, Various Alpha and Neighbour Merge

Significant Subnetworks Generated

For each disease, one of the most significant subnetworks generated by Neighbour Merge is selected. This subnetwork satisfies such criteria: 1. It appears in both datasets; 2. It has the biggest size in all the subnetworks that satisfy 1. These subnetworks can be further studied for their biological relevance to the respective disease.

Figure 5-9 below shows the most significant subnetworks from DMD, Lung, Prostate and ALL respectively.

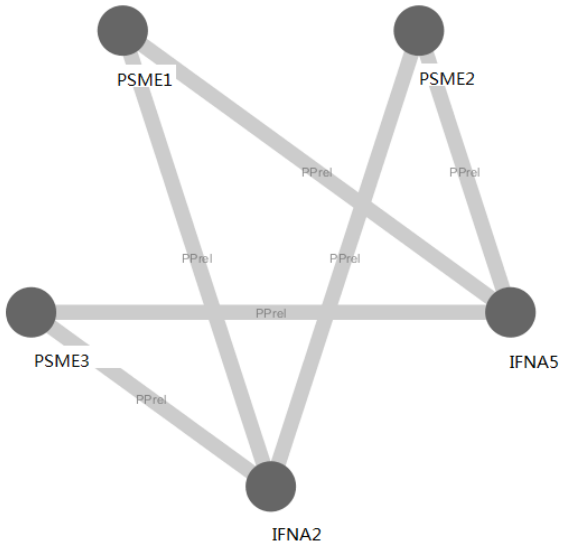

Figure 6 Most Significant Subnetwork Produced by Neighbour Merge for DMD

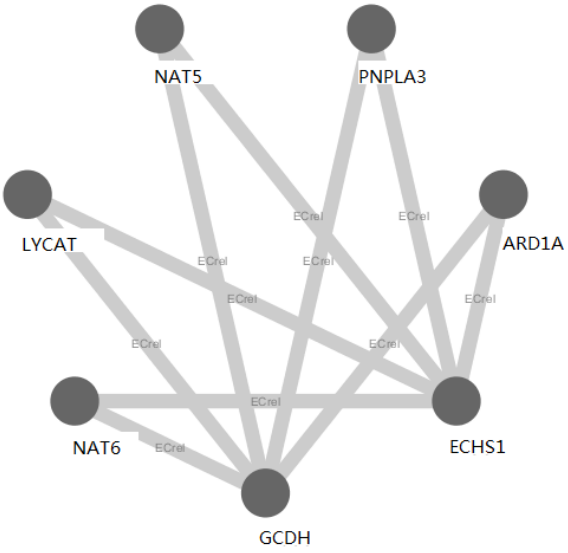

Figure 7 Most Significant Subnetwork Produced by Neighbour Merge for Lung

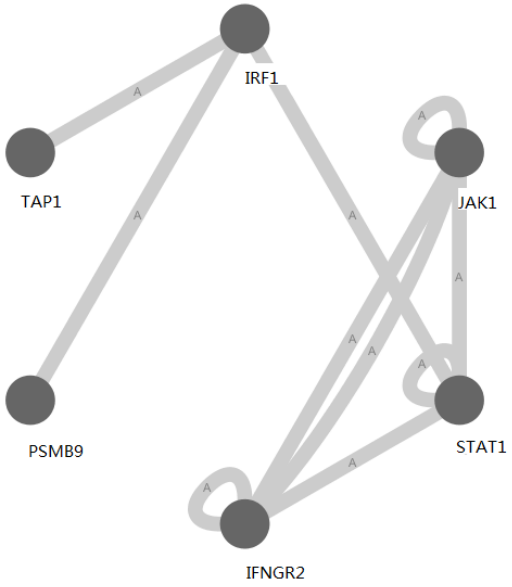

Figure 8 Most Significant Subnetwork Produced by Neighbour Merge for Prostate

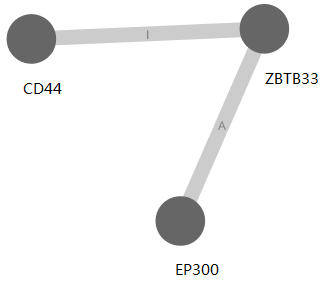

Figure 9 Most Significant Subnetwork Produced by Neighbour Merge for ALL

Discussion

In this section, a comparison was made between the results produced by the Soh's original SNet algorithm and the ones produced by the modified SNet algorithm. The effectiveness of Various Alpha and Neighbour Merge algorithm is then evaluated.

Inconsistency with Soh's Result

The first part of this project is a re-implementation of SNet. The algorithm used for SNet is the same as Soh proposed. However, the result produced by our SNet

algorithm is different from Soh's Result, even though the same datasets and the same SNet algorithm were used.

In Soh's research, the overlap of genes is between 51% and 93%. While in this project, the overlap of genes is between 25% and 42%. This is nearly half of Soh's result. In Soh's research, the overlap of pathways is between 48% and 91%. For our project, this is between 75% and 96%.

Several possible factors for causing this difference were proposed.

Different version of PathwayAPI used. The PathwayAPI used in this project was downloaded from PathwayAPI website (Soh, 2009), which is a different version from the one used in Soh's research. This should be the key factor that caused the results to be different. However, due to the unavailability of the PathwayAPI version used by Soh, this factor cannot be eliminated.

Problems in mapping Probe_set_id to Gene Name. The original datasets for each disease only contain *Probe_set_id* for each gene, and they were pre-processed by adding a *Gene_Name* to each gene with the aid of a mapping table. The mapping table used in this project may be different from the one used by Soh, thus, leading to different results.

Effectiveness of Various Alpha and Neighbour Merge

From the results, it is shown that with the use of Various Alpha method, there is not a consistent improvement on the number of overlaps and size of subnetworks. While, Neighbour Merge method has enhanced the results on SNet.

First of all, Neighbour Merge consistently produces subnetworks with larger sizes, as shown in Chart 1 and Chart 2.1 to 2.8. Particularly, Neighbour Merge produces more

extra-large subnetworks with size larger than 60. In addition, even though total overlaps produced by Neighbour Merge method is less than those produced by SNet, the size of overlapping subnetworks are significantly larger in Neighbour Merge method. Thus, Neighbour Merge method is more effective in improving the SNet algorithm, by increasing the subnetworks sizes, but not in the total number of subnetworks produced.

Checking Overlap Algorithm

As mentioned earlier, the algorithm used to check overlap is not accurate. As two subnetworks with the same genes may be labelled as different ones in SNet, the number of overlaps calculated by this algorithm is an overestimation.

From the results of Perfect Match and Partial Match, it is common that the same subnetwork may appear in more than one pair of overlaps. For example in the overlap result of Prostate datasets, the subnetwork 4339 from Lapointe's Data appeared 12 times, and subnetwork 4013 from Lapointe's Data appeared 14 times. The reason for such a situation is that the same gene may exist in more than one pathway, while when SNet is performed, this is not taken into consideration. When two subnetworks containing the same genes are generated from two different pathways, they are marked as different subnetworks in SNet algorithm. Hence, one subnetwork may correspond to more than one subnetwork in the other list. As a result, the number of overlaps generated by the overlap checking algorithm is an overestimation of the actual value.

Conclusion

In conclusion, this project has added two modifications to SNet algorithm, which are Various Alpha and Neighbour Merge. Various Alpha is shown to be not effective in improving the results of SNet, while Neighbour Merge method is more significant in the enhancement of SNet algorithm by increasing the size of subnetworks and consistency of results. Also, a visualization component was included for the users to better visualize and compare the subnetworks concerned. Finally, a software package which encompasses all the functions was implemented for a more convenient use of SNet.

Reference

Bhattacharjee A, Richards WG, Staunton J, Li C, Monti S, Vasa P, Ladd C, Beheshti J, Bueno R, Gillette M, Loda M, Weber G, Mark EJ, Lander ES, Wong W, Johnson BE, Golub TR, Sugarbaker DJ, Mayerson M. (2001). Classification of human lung carcinomas by mRNA expression profiling reveals distinct adenocarcinoma subclasses. *Proceedings of the National Academy of Sciences of the United States of America, 98*(24), 13790-13795.

Donny Soh, Difeng Dong, Yike Guo, Limsoon Wong. (2012). Finding Consistent disease subnetworks across datasets. *Genomics (accepted, not published)*.

Friedman N, Linial M, Nachman I, Pe'er D. (2000). Using Bayesinan networks to analyze expression data. *J Comput Biol, 7*(3-4), 601-620.

Garber ME, Troyanskaya OG, Schluens K, Petersen S, Thaesler Z, et al. (2011). Diversity of gene expression in adenocarcinoma of the lung. *Proceedings of the National Academy of Sciences of The United States of America, 98*(24), 13790-13795.

Haslett, J. N., Sanoudou, D., Kho, A. T., Bennett, R. R., Greenberg, S. A., Kohane, I. S., Beggs, A. H., and Kunkel, L. M. (2002). Gene expression comparison of biopsies from Duchenne muscular dystrophy (DMD) and normal skeletal muscle. *Proceedings of the National Academy of Sciences of the United States of America, 99*(23).

HgU133Plus2. (29 12, 2003). Retrieved 12 9, 2011, from Cardio Genomics: http://cardiogenomics.med.harvard.edu/groups/proj1/pages/hgu133plus2.html

Lahdesmaki H, Hautaniemi S, Shmulevich I, Yli-Harja O. (2006). Relationships between probabilistic Boolean networks and dynamic Bayesian networks as models of gene regulatory networks. *Signal Process, 86*(4), 814-834.

Lapointe J, Li C, Higgins JP, van de Rijin M, Bair E, Montgomery K, Ferrari M, Egevad L, Rayford W, Bergerheium U, Ekman P, DeMarzo AM, Tibshirani R, Botstein D, Brown PO, Brooks JD, Pollack JR. (2004). Gene expression profling identifies clinically relevant subtypes of prostate cancer. *Proceedings of the National Academy of Sciences of the United States of America, 101*(3), 811-816.

Pescatori, M., Broccolini, A., Minetti, C., Bertini, E., Bruno, C., Bernardini, C., Mirabella, M., Silvestri, G., Giglio, V., Modoni, A., Pedemonte, M., Tasca, G., Galluzzi, G., Mercuri, E., Tonali, P. A., and Ricci, E. (2007). Gene expression profiling in the early phases of DMD: a constant molecular signature characterizes DMD muscle from early postnatal life throughout disease progression. *FASEBJ, 21*(4).

Ross ME, Mahfouz R, Onciu M, Liu HC, Zhou X, Song G, Shurtleff SA, Pounds S, Cheng C, Ma J, Riberio RC, Rubnitz JE, Girtman K, Williams W, Raimondi SC, Liang DC, Shih LY, Pui CH, Downing JR. (2004). Gene expression profiling of pediatric acute myelogenous leukemia. *Blood, 104*(12), 3679-3687.

Singh D, Febbox PG, Ross K, Jackson DG, Manola J, Ladd C, Tamayo P, Renshaw AA, D'Amico AV, Richie JP, Lander ES, Loda M, Kantoff PW, Golub TR, Sellers WR.

(2002). Gene expression correlates of clinical prostate cancer behavior. *Cancer Cell, 1*(9), 203.

Soh D, Dong D, Guo Y, Wong L. (2010). Consistency, Comprehensiveness and Compatibility of Pathway Databases. *BMC Bioinformatics, 11*(449).

Soh, D. (2009). *Dump Database (sql)*. Retrieved 3 2, 2010, from Pathway API: http://www.pathwayapi.com/data/pathwaysql.zip

Tusher VG, Tibshirani R, Chu G. (2001). Significance analysis of microarrays applied to the ionizing radiation responces. *Proc Natl Acad Sci U S A, 98*(9), 5116-5121.

Yeoh EJ, Ross ME, Shurtle SA, Williams KW, Patel D, Mahfouz R, Behm FG, Raimondi SC, Relling MV, Patel A, Cheng. (n.d.). Classification, subtype discovery, and prediction of outcome in pediatric acute lymphoblastic leukemia by gene expression profiling. *Cancer Cell, 1*(2), 133-143.

Appendix A User Manual For SNet GUI

SNet GUI

USER'S MANUAL

Chai Haoqiang

Version 1 Deliverable

Download Link:
http://dl.dropbox.com/u/4168537/SNet%20GUI.zip

2011-10-23

Revision Sheet

Release No.	Date	Revision Description
1.0	2011-10-23	Initial Revision

1.0 GENERAL INFORMATION

1.1 System Overview

SNet is an algorithm developed by Soh *et al*. that produces connected gene groups from microarray datasets, with the aid of existing pathway information. The SNet GUI is a software developed to assist researchers perform SNet algorithm through a Graphic User Interface. It also incorporated three new functions to SNet, which are Neighbor Merge, Various Alpha and Visualization.

1.2 Authorized Use Permission

Usage of this software is open to everyone.

1.3 Points of Contact

For additional information, the developer can be contacted through his email: lucas.nus@gmail.com.

2.0 Pre-Requirement

2.1 System Requirement

The operation system should be Windows 2000/XP/2003/Vista/7 or higher versions, and is installed with MySQL 4.0 or higher versions.

2.2 MySQL Database Requirement

The MySQL Database should be pre-installed with PathwayAPI under schema called "pathwayapi". Otherwise, please download and install PathwayAPI from this site: http://www.pathwayapi.com/data/pathwaysql.zip.

2.3 Datasets Requirement

Datasets used should be pre-installed into MySQL database, under schema called "FYP". The table name for each datasets should follow the following format: "XXXX_normal", "XXXX_disease" where XXXX refers to dataset name.

3.0 GETTING STARTED

Extracting, Installing, and Running SNet GUI

3.1 Extracting from the .ZIP archive

In addition to user documentation, the SNet GUI.ZIP file contains installers for the Windows service aspect of the application. It must be executed before SNet GUI can be optimized and run.

3.1.1 Install Windows Service Component

Installs SNet GUI

3.1.2.1 Unzip the SNet GUi.zip file to a convenient location
3.1.2.2 Run setup.exe

3.2 Starting the Application

3.2.1 Start the 'SNet GUI" windows service

3.2.2 Configure the database
3.2.2.1 Click "Database Setting"
3.2.2.2 Input the "Service Host", "Username" and "Password" for Mysql Database
3.2.2.3 Input the "Directory" for output directory. E.g. "C:/"
3.2.2.4 Click "Done"

3.2.3 Configure Parameter
3.2.3.1 Input the parameters for current datasets
3.2.3.2 For alpha0 and alpha1, instead of 10% and 15%, they refers to the number of genes in the datasets times 10% and 15% respectively

3.2.4 Configure Datasets
3.2.4.1 Input the table name for datasets 1
3.2.4.2 Input the table name for datasets 2

3.2.5 Run SNet
3.2.5.1 Tick the datasets to be run (select either one or both datasets)
3.2.5.2 Click "SNet"
3.2.5.3 The label at left-bottom will show "processing". When the program finished execution, it will change to "Done."

3.2.6 Run Various Alpha

3.2.6.1 Tick the datasets to be run (select either one or both datasets)
3.2.6.2 Click “Various Alpha”
3.2.6.3 The label at left-bottom will show “processing”. When the program finished execution, it will change to “Done.”

3.2.7 Run Neighbor Merge
3.2.7.1 Tick the datasets to be run (select either one or both datasets)
3.2.7.2 Click “Neighbor Merge”
3.2.7.3 The label at left-bottom will show “processing”. When the program finished execution, it will change to “Done.”

3.2.8 Run Comparison
3.2.8.1 Precondition: Two subnetworks must be generated in the previous step
3.2.8.2 Click “Compare Result”

3.2.9 Output Format
3.2.9.1 Subnetworks file:
3.2.9.1.1 “disease”_“dataset”_ “disease”.csv
3.2.9.1.2 “disease”_“dataset”_ “disease”.rtf
3.2.9.1.3 “disease”_“dataset”_ “disease”_VariousAlpha.csv
3.2.9.1.4 “disease”_“dataset”_ “disease”_VariousAlpha.rtf
3.2.9.1.5 “disease”_“dataset”_ “disease”_merged.csv
3.2.9.1.6 “disease”_“dataset”_ “disease”_merged.rtf

3.2.9.2 Compare Result
3.2.9.2.1 compareResult_“disease”.txt

3.2.9.3 Compare Log
3.2.9.3.1 “disease”_ snOverlapRecord.csv
3.2.9.3.2 “disease”_ snOverlapRecord(VariousAlpha).csv
3.2.9.3.3 “disease”_ snOverlapRecord(Merged).csv
3.2.9.3.4 “disease”_ snPartialOverlapRecord.csv
3.2.9.3.5 “disease”_ snPartialOverlapRecord(VariousAlpha).csv
3.2.9.3.6 “disease”_ snPartialOverlapRecord(Merged).csv
3.2.9.3.7 “disease”_ snPartialOverlapRecord2.csv
3.2.9.3.8 “disease”_ snPartialOverlapRecord(VariousAlpha)2.csv
3.2.9.3.9 “disease”_ snPartialOverlapRecord(Merged)2.csv

Appendix B Source Code for SNet GUI

```
// Subnetwork Class

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SNet_V2._0.Module
{
   class Subnetwork
   {
      int _ID; // ID of this subnetwork
      string _pathwayID; // ID of the pathway that this subnetwork belongs
      List<Gene> _genes = new List<Gene>();
      List<GeneRelation> _geneRelations = new List<GeneRelation>();
      List<double>[] _scoreVector = new List<double>[2];
      List<double>[] _scoreVectorA1A2 = new List<double>[2];
      List<double>[] _scoreVectorMerged = new List<double>[2];
      double _tScore;
      double _pValue;
      double _tScoreA1A2;
      double _pValueA1A2;
      double _tScoreMerged;
      double _pValueMerged;

      public Subnetwork(int SNID)
      {
         _ID = SNID;
      }

      // Constructor for the subnetworks with single Gene
      public Subnetwork(List<string> gene, string pathwayID, int SNID)
      {
         _ID = SNID;
         _genes.Add(new Gene(gene));
         _pathwayID = pathwayID;
      }

      public void Add(List<string> geneRelation)
      {
         int flag1 = 0;
         int flag2 = 0;
         foreach (Gene g in _genes)
         {
            if (g.getGeneID() == geneRelation[1]) // Gene 1 already existed in subnetwork
               flag1 = 1;
            if (g.getGeneID() == geneRelation[2]) // Gene 2 already existed in subnetwork
               flag2 = 1;
         }
         if (flag1 == 0) // Means gene 1 is not in subnetwork yet, then add it in
         {
            Gene tempGene = new Gene(geneRelation[1], geneRelation[4], geneRelation[5]);
            _genes.Add(tempGene);
         }
         if (flag2 == 0)// Means gene 2 is not in subnetwork yet, then add it in
         {
            Gene tempGene = new Gene(geneRelation[2], geneRelation[6], geneRelation[7]);
            _genes.Add(tempGene);
         }
```

```
            GeneRelation gR = new GeneRelation(geneRelation[0], geneRelation[1], geneRelation[2],
geneRelation[3], geneRelation[4], geneRelation[6]);
            _geneRelations.Add(gR);
            updatePathwayID();
        }

        public void addRange(Subnetwork newSN)
        {
            _genes.AddRange(newSN.getGenes());
            _geneRelations.AddRange(newSN.getGeneRelations());
            _pathwayID = newSN.getPathwayID();
        }

        public void addScoreVector(List<double> scoreVector, int phenotype) { _scoreVector[phenotype] =
scoreVector; }
        public void addTScore(double tScore) { _tScore = tScore; }
        public void addPValue(double pValue) { _pValue = pValue; }
        public void addPValueA1A2(double pValue) { _pValueA1A2 = pValue; }
        public void addPValueMerged(double pValue) { _pValueMerged = pValue; }
        public List<Gene> getGenes() { return _genes; }
        public List<GeneRelation> getGeneRelations() { return _geneRelations; }
        public List<double>[] getScoreVector() { return _scoreVector; }
        public double getTScore() { return _tScore; }
        public double getTScoreA1A2() { return _tScoreA1A2; }
        public double getTScoreMerged() { return _tScoreMerged; }
        public double getPValue() { return _pValue; }
        public double getPValueA1A2() { return _pValueA1A2; }
        public double getPValueMerged() { return _pValueMerged; }
        public int getID() { return _ID; } // Returns the ID of this subnetwork
        public void updatePathwayID()
        {
            if (_geneRelations.Count > 0)
                _pathwayID = _geneRelations[0].getPathwayID();
        }
        public string getPathwayID() { return _pathwayID; }
        public void clear()
        {
            _genes = new List<Gene>();
        }
        public bool isEmpty()
        {
            if (_genes.Count == 0)
                return true;
            else return false;
        }
        public int overlap(Subnetwork sn)
        {
            int geneOverlap = 0;
            foreach (Gene g1 in _genes)
            {
                foreach (Gene g2 in sn.getGenes())
                {
                    if (g1.getGeneID() == g2.getGeneID())
                        geneOverlap++;
                }
            }
            return geneOverlap;
        }
    }

    class Gene : IEquatable<Gene>
    {
        string[] _gene = new string[3]; // 0 gene_id, 1 gene_name, 2 gScore
```

```
        public Gene() { }
        public Gene(List<string> gene)
        {
            for (int i = 0; i < 3; i++)
                _gene[i] = gene[i];
        }
        public Gene(string gene_id, string gene_name, string gScore)
        {
            _gene[0] = gene_id;
            _gene[1] = gene_name;
            _gene[2] = gScore;
        }
        public string[] getGene() { return _gene; }
        public string getGeneID() { return _gene[0]; }
        public string getGeneName() { return _gene[1]; }
        public string getGScore() { return _gene[2]; }

        public bool Equals(Gene other)
        {
            if (this._gene[0] == other._gene[0])
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }

    class GeneRelation
    {
        string _pathway_id;
        string _gene_id1;
        string _gene_id2;
        string _gene_name1;
        string _gene_name2;
        string _rs;

        public GeneRelation(string pathway_id, string gene_id1, string gene_id2, string rs, string gene_name1,
string gene_name2)
        {
            _pathway_id = pathway_id;
            _gene_id1 = gene_id1;
            _gene_id2 = gene_id2;
            _rs = rs;
            _gene_name1 = gene_name1;
            _gene_name2 = gene_name2;
        }

        public string[] getRelation()
        {
            string[] tempString = new string[4];
            tempString[0] = _pathway_id;
            tempString[1] = _gene_id1;
            tempString[2] = _gene_id2;
            tempString[3] = _rs;
            return tempString;
        }

        public string getRelationString()
        {
            string tempString = _pathway_id + " ";
```

```
            tempString += _gene_id1 + " ";
            tempString += _gene_id2 + " ";
            tempString += _rs;
            return tempString;
        }

        public string getPathwayID() { return _pathway_id; }
    }
}
```

```
// Logic Class
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using MySql.Data;
using MySql.Data.MySqlClient;
using System.IO;

namespace SNet_V2._0.Module
{
   class Logic
   {
      #region test
      public void testDatabase()
      {
         List<List<string>> test = retrieveTopAlphaPercentGene(10, 3, "genechip_normal");
         foreach (List<string> l in test)
         {
            foreach (string s in l)
            {
               Console.Write(s + " ");
            }
            Console.Write("\n");
         }
         Console.Read();
      }
      #endregion
      /* This region will connect to the SQL server and retrieve necessary data */

      //2011-10-12
      List<List<string>> _geneList;
      List<List<string>>[] _pathwayList = new List<List<string>>[400];
      public List<double> _totalTScore;
      #region Database Connection and Data Retrieving

      // Retrieve top 10% gene of each sample, and return in an list of list
      // SampleStart is in the genechip start from which column that samples started. (e.g. GeneName,
GeneID, Sample1, Sample2... Then it's 3)
      // alpha is a number, not a percent
      // Return Value: A List of list, each list is the top alpha geneID of that sample
      public List<List<string>> retrieveTopAlphaPercentGene(double alpha, int sampleStart, string
tableName)
      {
         List<List<string>> selectedGenes = new List<List<string>>();
         List<string> columnNames = new List<string>();

         #region routine connuction
         MySqlConnection connection = new MySqlConnection();
         MySqlDataAdapter data = new MySqlDataAdapter();
         connection.ConnectionString =
         "server=localhost;" // Server name (on my computer it's localhost)
         + "database=fyp;" // Database name (on my computer it's fyp)
         + "uid=root;" // User name
         + "password=root;"; // Password
         connection.Open();
         #endregion

         #region get column names
         MySqlCommand command_select = connection.CreateCommand();
         command_select.CommandText =
```

```
        "select column_name from INFORMATION_SCHEMA.COLUMNS where TABLE_NAME = '" +
tableName + "'"; // Select the column names from table

        data.SelectCommand = command_select;
        DataSet dataset = new DataSet();
        data.Fill(dataset, "sample_data");

        IDataReader dataReader = dataset.CreateDataReader();
        while (dataReader.Read())
        {
            columnNames.Add(dataReader.GetString(0));
        }

        for (int i = 0; i < sampleStart - 1; i++)
            columnNames.Remove(columnNames[0]);

        #endregion

        foreach (string name in columnNames)
        {
            List<string> singleColumnGenes = new List<string>();

            // This special code is for each column not to 0
            string columnNotZero = null;
            foreach (string colName in columnNames)
                columnNotZero = columnNotZero + " and " + colName + " <>0 ";

            command_select.CommandText =
            "select IDENTIFIER from " + tableName + " WHERE IDENTIFIER<>'0' " + columnNotZero + "
order by " + name + " limit " + alpha; ; // select Top alpha genes of each sample

            data.SelectCommand = command_select;
            DataSet tempDataSet = new DataSet();
            data.Fill(tempDataSet, "sample_data");

            dataReader = tempDataSet.CreateDataReader();
            while (dataReader.Read())
            {
                singleColumnGenes.Add(dataReader.GetString(0));
            }

            selectedGenes.Add(singleColumnGenes);
        }

        return selectedGenes;
    }

    public List<string>[] pathwayAPITableRetrieve(string query)
    {
        MySqlConnection connection = new MySqlConnection();
        MySqlDataAdapter data = new MySqlDataAdapter();
        connection.ConnectionString =
        "server=localhost;" // Server name (on my computer it's localhost)
        + "database=pathwayapi;" // Database name (on my computer it's pathwayapi)
        + "uid=root;" // User name
        + "password=root;"; // Password
        connection.Open();

        MySqlCommand command_select = connection.CreateCommand();
        command_select.CommandText = query;

        data.SelectCommand = command_select;
        DataSet dataset = new DataSet();
```

```
            data.Fill(dataset, "sample_data");

            int M = 100000; // Assume each pathway contains less than 10000 genes

            IDataReader dataReader = dataset.CreateDataReader();


            List<string>[] listArrayTemp = new List<string>[M];
            for (int i = 0; i < M; i++) listArrayTemp[i] = new List<string>();

            int rowNumber = 0; // The first row is for store of basic information, including "row count" and
"column count"
            while (dataReader.Read())
            {
               for (int i = 0; i < dataReader.FieldCount; i++)
               {
                  listArrayTemp[rowNumber].Add(Convert.ToString(dataReader.GetValue(i)));
               }
               rowNumber++;
            }

            List<string>[] listArray = new List<string>[rowNumber];
            for (int i = 0; i < rowNumber; i++)
            {
               listArray[i] = listArrayTemp[i];
            }

            return listArray;
         }
         #endregion
         #region core
         int nextSNID;
         UI ui;
         public Logic(UI _ui)
         {
            ui = _ui;
         }
         // Return all the genes that belong to SN list
         public List<Subnetwork> core(string phenotype1, string phenotype2, double alpha, double alpha2,
double beta, int sampleStart)
         {
            nextSNID = 0;
            for (int i = 0; i < 400; i++)
               _pathwayList[i] = new List<List<string>>();

            List<Subnetwork> subnetworks = subnetworkGeneration(phenotype1, alpha, beta, sampleStart); //
Generate Subnetwork

            Jump j = new Jump(Convert.ToString(subnetworks.Count));
            j.ShowDialog();


            subnetworks = scoring(subnetworks, phenotype1, phenotype2, alpha, sampleStart); // Score them


            subnetworks = validation(phenotype1, phenotype2, alpha, beta, sampleStart, subnetworks); //
Validate them
            subnetworks = sortSubnetworksBasedOnPValue(subnetworks); // Sort according to pValue

            #endregion

            return subnetworks;
         }
```

```
        #region subnetwork generation
        public List<Subnetwork> subnetworkGeneration(string phenotype, double alpha, double beta, int
sampleStart)
        {
            List<List<string>> topAlphaGenes = retrieveTopAlphaPercentGene(alpha, sampleStart, phenotype);
// Select List of Top alpha genes for each sample; 1000 is alpha, 3 is start position of sample,
"genechip_normal" is table name
            List<List<string>> topBetaGenes = geneListGeneration(topAlphaGenes, beta); // Select genes
belonging to the genelists of at least beta percent samples. For each element, 0 is geneName, 1 is G Score
            List<List<string>> geneList = addInGeneID(topBetaGenes); // Modify a bit. Now 0 is geneID, 1 is
geneName, 2 is GScore

            //2011-10-12
            _geneList = new List<List<string>>();
            foreach (List<string> ls in geneList)
                _geneList.Add(ls);

            List<Subnetwork> subnetworks = new List<Subnetwork>();

            // Go through each pathway and select genes that belong to both geneList and that pathway, also use
these genes to form subnetworks
            for (int i = 0; i < 398; i++)
            {
                List<List<string>> tempList = selectGenesOfThatPathway(geneList, i);
                subnetworks.AddRange(formSubnetwork(tempList, i)); // Add newly generated subnetworks in to
the big list
            }
            return subnetworks;
        }
        public List<List<string>> geneListGeneration(List<List<string>> topAlphaGenes, double beta)
        {
            List<List<string>> geneList = new List<List<string>>();
            List<string> recordList = new List<string>();
            foreach (List<string> lS in topAlphaGenes)
            {
                foreach (string s in lS)
                {
                    if (!recordList.Contains(s))
                    {
                        List<string> tempList = new List<string>();
                        int count = 1;
                        foreach (List<string> otherLS in topAlphaGenes)
                        {
                            if (otherLS != lS)
                            {
                                if (otherLS.Contains(s)) count++;
                            }
                        }
                        if (count > Convert.ToDouble(topAlphaGenes.Count) * beta)
                        {
                            recordList.Add(s);
                            tempList.Add(s); // Add in Gene Name
                            tempList.Add(Convert.ToString(Convert.ToDouble(count) /
Convert.ToDouble(topAlphaGenes.Count))); // Add in GScore
                            geneList.Add(tempList);
                        }
                    }
                }
            }
            return geneList;
        }
        public List<List<string>> addInGeneID(List<List<string>> geneList)
```

```
        {
            /// Input: A list of of List<string>, for each List<string>, 0 is GeneName, 1 is GScore
            /// Function: Add in GeneID
            /// OutPut: Another List of List<string>, for each List<string>, 0 is GeneID, 1 is GeneName, 2 is
GScore
            List<string>[] geneMapping;
            geneMapping = pathwayAPITableRetrieve("select * from gene_mapping;");
            var gene_id = from p in geneList
                          join q in geneMapping on p[0] equals q[1]
                          select new List<string> { q[0], p[0], p[1] }; // Now becomes 0 is GeneID, 1 is GeneName, 2
is G score
            geneList = new List<List<string>>();
            foreach (List<string> list in gene_id)
            {
                geneList.Add(list);
            }
            return geneList;
        }
        public List<List<string>> selectGenesOfThatPathway(List<List<string>> geneList, int
pathwayNumber)
        {
            /// Input: GeneList (0 is GeneID, 1 is GeneName, 2 is GScore), pathwayNumber
            /// Function: Filter GeneList, select those genes belong to given pathway
            /// OutPut: A smaller GeneList (0 is GeneID, 1 is GeneName, 2 is GScore), pathwayNumber
            List<string>[] pathwayGenes;
            List<List<string>> selectedGeneList = new List<List<string>>();

            pathwayGenes = pathwayAPITableRetrieve("select * from pathway_genes where pathway_id = " +
pathwayNumber + ";");

            var selected_gene_id = from gene in geneList
                                   join p in pathwayGenes on gene[0] equals p[1]
                                   select gene;

            foreach (List<string> a in selected_gene_id)
            {
                selectedGeneList.Add(a);
            }

            return selectedGeneList;
        }
        public List<Subnetwork> formSubnetwork(List<List<string>> geneList, int pathwayNumber)
        {
            /* Get the Subnetworks formed by selected genes in that pathway
             * Input: A List of List<string> Genes belonging to that pathway (0 is GeneID, 1 is GeneName, 2 is
GScore), pathwayNumber
             * Output is a list of Subnetworks (defined by Subnetwork class)
             */
            List<Subnetwork> SubnetworkList = new List<Subnetwork>();
            List<string>[] pathwayRS; // Gene relation in a certain pathway (retrieved from pathwayAPI)
            List<List<string>> pathwayRSList = new List<List<string>>();

            pathwayRS = pathwayAPITableRetrieve("select * from pathway_rs where pathway_id = " +
Convert.ToString(pathwayNumber) + ";");
            foreach (List<string> a in pathwayRS)
            {
                pathwayRSList.Add(a);

                //2011-10-12
                _pathwayList[pathwayNumber].Add(a);
            }

            //Remove relations where not both genes appear in the geneList
```

```
        foreach (List<string> RS in pathwayRSList.ToList())
        {
            int flag1 = 0;
            int flag2 = 0;
            foreach (List<string> GL in geneList)
            {
                if (GL[0] == RS[1])
                {
                    flag1 = 1;
                    RS.Add(GL[1]); // 4: GeneName
                    RS.Add(GL[2]); // 5: GScore
                }
            }
            foreach (List<string> GL in geneList)
            {
                if (GL[0] == RS[2])
                {
                    flag2 = 1;
                    RS.Add(GL[1]); // 6: GeneName
                    RS.Add(GL[2]); // 7: GScore
                }
            }
            if (flag1 == 0 || flag2 == 0)
            {
                pathwayRSList.Remove(RS);
            }
        }
        // By now, in the pathwayRSList, there should have a List of Relations
        // In each relation, 0 is pathwayID, 1 is geneID1, 2 is geneID2, 3 is relation, 4 geneName1, 5
GScore1, 6 geneName2, 7 GScore2
        // It must have information of both genes, otherwise it won't exist, because it would already be
deleted

        // Select Subnetworks having more than 1 genes using breadth first search
        while (pathwayRSList.Count() > 0)
        {
            Subnetwork tempSubnetwork = new Subnetwork(getNextSNID()); // Create a new subnetwork
            tempSubnetwork.Add(pathwayRSList[0]); // Add the first relation to it
            pathwayRSList.Remove(pathwayRSList[0]); // Remove the relation once its been added
            int count = pathwayRSList.Count();
            int smallCount = 0;
            do
            {
                // What's smallCount for?
                // Each round, I'll go through each unchecked relation and check if it is linked to the
tempSubnetwork
                // If in one round, one or more relation is added (smallCount becomes non zero), then I have to
redo the whole round again
                // in case there're relations that is linked to the newly added relation
                smallCount = 0;
                foreach (List<string> a in pathwayRSList.ToList())
                {
                    Gene tempGene1 = new Gene(a[1], a[4], a[5]);
                    Gene tempGene2 = new Gene(a[2], a[6], a[7]);
                    if (tempSubnetwork.getGenes().Contains(tempGene1) ||
tempSubnetwork.getGenes().Contains(tempGene2))
                    {
                        tempSubnetwork.Add(a);
                        pathwayRSList.Remove(a);
                        smallCount++;
                    }
                }
            } while (smallCount > 0);
```

```
            SubnetworkList.Add(tempSubnetwork);
        }

        // Assign Subnetworks with only one element
        foreach (List<string> g in geneList)
        {
            int flag = 0;
            Gene tempGene = new Gene(g);
            foreach (Subnetwork c in SubnetworkList) // Check all the subnetworks to see if this gene already
belong to one of them
            {
                if (c.getGenes().Contains(tempGene))
                    flag = 1;
            }
            if (flag == 0) // If this gene doesn't belong to any existing subnetwork, create a new subnetwork to
hold it
            {
                Subnetwork tempSubnetwork = new Subnetwork(g, Convert.ToString(pathwayNumber),
getNextSNID());
                SubnetworkList.Add(tempSubnetwork);
            }
        }

        return SubnetworkList;
        // This function can be improved in efficiency
        // When generating subnetworks in previous step, remove added genes from geneList
        // Thus when doing this step, the geneList will be much shorter
        // And we don't even need to check! 2011-06-05 20:51 Yes, agreed.
    }
    public int getNextSNID()
    {
        // returns the ID of "next" subnetwork
        nextSNID++;
        return nextSNID;
    }
    #endregion

    #region scoring
    public List<Subnetwork> scoring(List<Subnetwork> subnetworks, string phenotype1, string
phenotype2, double alpha, int sampleStart)
    {
        List<List<string>> topAlphaGenes1 = retrieveTopAlphaPercentGene(alpha, sampleStart,
phenotype1);
        List<List<string>> topAlphaGenes2 = retrieveTopAlphaPercentGene(alpha, sampleStart,
phenotype2);
        subnetworks = addScoreVector(subnetworks, topAlphaGenes1, 1);
        subnetworks = addScoreVector(subnetworks, topAlphaGenes2, 2);
        subnetworks = calculateTScore(subnetworks);

        return subnetworks;
    }
    public List<Subnetwork> addScoreVector(List<Subnetwork> subnetworks, List<List<string>>
topAlphaGenes, int whichPhenotype)
    {
        foreach (Subnetwork cluster in subnetworks)
        {
            List<double> SNVector = new List<double>();
            foreach (List<string> s in topAlphaGenes)
            {
                double SN = 0;
                foreach (Gene g in cluster.getGenes())
                {
```

```
                if (s.Contains(g.getGeneName()))
                    SN += Convert.ToDouble(g.getGScore());
                }
                SNVector.Add(SN);
            }
            cluster.addScoreVector(SNVector, whichPhenotype - 1);
        }
        return subnetworks;
    }
    public List<Subnetwork> calculateTScore(List<Subnetwork> subnetworks)
    {
        int k = 1;
        foreach (Subnetwork c in subnetworks)
        {
            double tScore;
            List<double>[] scoreVector = c.getScoreVector();
            double S;
            int n1 = scoreVector[0].Count();
            int n2 = scoreVector[1].Count();
            S = Math.Sqrt(((n1 - 1) * getVariance(scoreVector[0]) + (n2 - 1) * getVariance(scoreVector[1])) /
(n1 + n2 - 2));
            tScore = (scoreVector[0].Average() - scoreVector[1].Average()) / (S * Math.Sqrt(1 /
Convert.ToDouble(n1) + 1 / Convert.ToDouble(n2)));
            if ((getVariance(scoreVector[0]) + getVariance(scoreVector[1])) == 0)
            {
                if (scoreVector[0].Average() == scoreVector[1].Average())
                    tScore = 0;
                else tScore = 100;
                // Means they're very diffrent
                // Before this, every NaN is treated as infinity which is unfair
            }

            c.addTScore(tScore);

            k++;
        }
        return subnetworks;
    }
    public double getVariance(List<double> list)
    {
        // double avg = list.Average(); // I don't trust you
        double avg = 0;
        foreach (double d in list)
            avg += d;
        avg /= list.Count;

        double sumDeviation = 0;
        double variance;
        foreach (double d in list)
        {
            sumDeviation += (d - avg) * (d - avg);
        }
        variance = sumDeviation / (list.Count() - 1);
        return variance;
    }
    #endregion

    #region modification
    // For each subnetwork SN, find their immediate neighbour (IN), which is a gene that is only one edge
away
    // If this IN is in the geneList2, then add IN to the subnetwork SN
    public List<Subnetwork> alpha1alpha2(List<Subnetwork> subnetworks, string phenotype1, double
alpha2, double beta, int sampleStart)
```

```
    {
        List<List<string>> topAlphaGenes2 = retrieveTopAlphaPercentGene(alpha2, sampleStart,
phenotype1); // Select List of Top alpha genes for each sample; 1000 is alpha, 3 is start position of sample,
"genechip_normal" is table name

        List<List<string>> topBetaGenes2 = geneListGeneration(topAlphaGenes2, beta); // Select genes
belonging to the genelists of at least beta percent samples. For each element, 0 is geneName, 1 is G Score

        List<List<string>> geneListTemp2 = addInGeneID(topBetaGenes2); // Modify a bit. Now 0 is
geneID, 1 is geneName, 2 is GScore

        List<string> geneList2 = new List<string>(); // Modify a bit more. Now it only contains a list of
geneID.
        foreach (List<string> lS in geneListTemp2)
          geneList2.Add(lS[0]);

        int count = 1;

        foreach (Subnetwork sn in subnetworks)
        {
          int flag = 0; // flag becomes 1 when sn contains an IN that is in geneList2
          string pathwayNumber = sn.getPathwayID();

          List<string> selectedGenes = new List<string>(); // List of genes that are immediately connected
to sn
          List<List<string>> selectedRelations = new List<List<string>>(); // Relations that correspond to
selectedGenes
          List<string> geneIDs = new List<string>(); // IDs of Genes that are contained in sn
          foreach (Gene g in sn.getGenes())
            geneIDs.Add(g.getGeneID());

          List<string>[] pathwayRS; // Gene relation in a certain pathway (retrieved from pathwayAPI)
          List<List<string>> pathwayRSList = new List<List<string>>();
          pathwayRS = pathwayAPITableRetrieve("select * from pathway_rs where pathway_id = " +
pathwayNumber + ";");
          foreach (List<string> a in pathwayRS)
          {
            pathwayRSList.Add(a);
          }

          // Check each relation, if one of its node is contained in the sn but the other is not, then add the
other in the selectedGenes, and add the relation in the selectedRelations
          foreach (List<string> r in pathwayRSList)
          {
            if (geneIDs.Contains(r[1]) && !geneIDs.Contains(r[2]))
            {
              selectedGenes.Add(r[2]);
              selectedRelations.Add(r);
            }
            if (geneIDs.Contains(r[2]) && !geneIDs.Contains(r[1]))
            {
              selectedGenes.Add(r[1]);
              selectedRelations.Add(r);
            }
          }

          for (int i = 0; i < selectedGenes.Count; i++)
          {
            if (geneList2.Contains(selectedGenes[i])) // If this IN is contained in the geneList2, then add the
corresponded relation to the subnetwork
            {
              selectedRelations[i] = modifyRelation(selectedRelations[i], geneListTemp2); // Modifiy it
into a format that can be added to SN
```

```
                    sn.Add(selectedRelations[i]);
                    flag = 1;
                }
            }

            count++;
        }

        return subnetworks;
    }
    // Modify incoming relation List<string> by adding
    // 4, 5: GeneName and GScore of Gene1
    // 6, 7: GeneName and GScore of Gene2
    public List<string> modifyRelation(List<string> relation, List<List<string>> geneList)
    {
        foreach (List<string> GL in geneList)
        {
            if (GL[0] == relation[1])
            {
                relation.Add(GL[1]); // 4: GeneName
                relation.Add(GL[2]); // 5: GScore
            }
            if (GL[0] == relation[2])
            {
                relation.Add(GL[1]); // 6: GeneName
                relation.Add(GL[2]); // 7: GScore
            }
        }

        // Possible Problem
        if (relation.Count == 4)
        {
            relation.Add("Null"); // 4
            relation.Add("0.5"); // 5
            relation.Add("Null"); // 6
            relation.Add("0.5"); // 7
        }
        else if (relation.Count == 6)
        {
            relation.Add("Null"); // 6
            relation.Add("0.5"); // 7
        }

        return relation;
    }
    // Check subnetworks of the same pathway pairwisely
    // If they're separated by only one neighbour, merge them
    public List<Subnetwork> detectNearSubnetwork(List<Subnetwork> subnetworks, string phenotype1,
double alpha, double beta, int sampleStart, string directory)
    {
        int flag = 1;
        string recordingString = null;
        int round = 0;
        //List<List<int>> checkedPairs = new List<List<int>>(); // once SN1 and SN2 are checked to be "not
near", then put in this list, and don't check them next time
        int[,] snPairs = new int[subnetworks.Count + 1, subnetworks.Count + 1]; // used to record if one pair
is checked
        for (int i = 0; i <= subnetworks.Count; i++)
            for (int j = 0; j <= subnetworks.Count; j++) // i and j are the SNID of each SN
                snPairs[i, j] = 0; // once checked, change its value to 1

        while (flag == 1) // Indicating that new overlap was found
        {
```

```
            round++;
            // Console.WriteLine("Round: " + round); // Current round 2011-10-31
            // Console.WriteLine("SN Count: " + subnetworks.Count); // Current number of subnetworks
2011-10-31

            flag = 0; // Initialize it as 0, if nothing happened, it will remain as 0
            for (int i = 0; i < subnetworks.Count; i++)
            {
               for (int j = i + 1; j < subnetworks.Count; j++)
               {
                  if (subnetworks[i].getPathwayID() == subnetworks[j].getPathwayID()) // Only SNs in the
same pathway will be checked
                  {
                     int SNID1 = subnetworks[i].getID();
                     int SNID2 = subnetworks[j].getID();

                     if (snPairs[SNID1, SNID2] == 0) // Indicating that this pair is not checked yet
                     {
                        Subnetwork tempSN = checkOverlap(subnetworks[i], subnetworks[j], phenotype1, alpha,
beta, sampleStart);
                        if (tempSN.isEmpty() == false)
                        {
                           recordingString += tempSN.getID() + ": " + subnetworks[i].getID() + ", " +
subnetworks[j].getID() + "\n";
                           subnetworks[i] = tempSN;
                           subnetworks.Remove(subnetworks[j]);
                           j--;
                           flag = 1;

                           // Console.WriteLine("BreakHere"); 2011-10-31

                           break;
                        }
                        else
                        {
                           snPairs[SNID1, SNID2] = 1;
                           snPairs[SNID2, SNID1] = 1;
                        }
                     }
                  }
               }
               if (flag == 1)
                  break;
            }
         }

         printString(recordingString, directory, phenotype1 + "_mergedLog");

         return subnetworks;
      }

      // Check if two subnetworks sn1 and sn2 are one neighbor away
      // If yes, merge them and return a new subnetwork sn3
      // Otherwise, return an empty subnetwork sn3
      // IN: Immediate Neighbours of one subnetwork
      // relation: relations that are corresponding to IN
      public Subnetwork checkOverlap(Subnetwork sn1, Subnetwork sn2, string phenotype1, double alpha,
double beta, int sampleStart)
      //public Subnetwork checkOverlap(Subnetwork sn1, Subnetwork sn2, List<string> IN1, List<string>
IN2, List<List<string>> relation1, List<List<string>> relation2)
      {
         Subnetwork sn3 = new Subnetwork(sn1.getID());
         sn3.addRange(sn2);
```

```
        sn3.addRange(sn1);
        int flag = 0;

        // selecedGenes1 and selectedGenes2 contains IN of sn1 and sn2 respectively
        // selectedRelations1 and selectedRelations2 contains relations of sn1 and sn2 respectively
        // flag is 0 in the beginning, once one overlapped IN was found, flag became 1

        string pathwayNumber1 = sn1.getPathwayID();
        List<string> selectedGenes1 = new List<string>(); // List of genes that are immediately connected to
sn1
        List<List<string>> selectedRelations1 = new List<List<string>>(); // Relations that correspond to
selectedGenes1
        List<string> geneIDs1 = new List<string>(); // IDs of Genes that are contained in sn1
        foreach (Gene g in sn1.getGenes())
           geneIDs1.Add(g.getGeneID());

        //2011-10-12， ?上 ?面?的 ?comment也 ?是 ?今?天 ¬¨¬除y去 ¨£¤的 ?
        List<List<string>> pathwayRSList1 = _pathwayList[Convert.ToInt16(pathwayNumber1)];

        // Check each relation, if one of its node is contained in the sn2 but the other is not, then add the other
in the selectedGenes2, and add the relation in the selectedRelations2
        foreach (List<string> r in pathwayRSList1)
        {
           if (geneIDs1.Contains(r[1]) && !geneIDs1.Contains(r[2]))
           {
              selectedGenes1.Add(r[2]);
              selectedRelations1.Add(r);
           }
           if (geneIDs1.Contains(r[2]) && !geneIDs1.Contains(r[1]))
           {
              selectedGenes1.Add(r[1]);
              selectedRelations1.Add(r);
           }
        }

        string pathwayNumber2 = sn2.getPathwayID();
        List<string> selectedGenes2 = new List<string>(); // List of genes that are immediately connected to
sn2
        List<List<string>> selectedRelations2 = new List<List<string>>(); // Relations that correspond to
selectedGenes2
        List<string> geneIDs2 = new List<string>(); // IDs of Genes that are contained in sn2
        foreach (Gene g in sn2.getGenes())
           geneIDs2.Add(g.getGeneID());

        //2011-10-12
        List<List<string>> pathwayRSList2 = _pathwayList[Convert.ToInt16(pathwayNumber2)];

        // Check each relation, if one of its node is contained in the sn2 but the other is not, then add the other
in the selectedGenes2, and add the relation in the selectedRelations2
        foreach (List<string> r in pathwayRSList2)
        {
           if (geneIDs2.Contains(r[1]) && !geneIDs2.Contains(r[2]))
           {
              selectedGenes2.Add(r[2]);
              selectedRelations2.Add(r);
           }
           if (geneIDs2.Contains(r[2]) && !geneIDs2.Contains(r[1]))
           {
              selectedGenes2.Add(r[1]);
              selectedRelations2.Add(r);
           }
        }
```

```
            //2011-10-12
            List<List<string>> geneListTemp = _geneList;

            for (int i = 0; i < selectedGenes1.Count; i++)
            {
               for (int j = 0; j < selectedGenes2.Count; j++)
               {
                  if (selectedGenes1[i] == selectedGenes2[j])
                  {
                     List<string> relation1 = modifyRelation(selectedRelations1[i], geneListTemp);
                     List<string> relation2 = modifyRelation(selectedRelations2[j], geneListTemp);
                     sn3.Add(relation1);
                     sn3.Add(relation2);
                     flag = 1;
                  }
               }
            }

            if (flag == 0)
            {
               sn3.clear();
            }

            return sn3;
         }

         #endregion
         #region validation
         public List<Subnetwork> validation(string phenotype1, string phenotype2, double alpha, double beta,
int sampleStart, List<Subnetwork> comingSubnetworks)
         {
            // 1st, get 1000 randomly generated subnetworks, and calculate the tScore for each of them
            // Store these tScores in the totalTscore

            List<Subnetwork> bigSubnetworks = new List<Subnetwork>(); // A list of randomly generated
subnetworks
            List<double> totalTScore = new List<double>(); // A list of tScores
            int count = 0;

            // Repeat while loop, until more than 10000 subnetworks (tScores) are generated
            // Count is the termination restriction, make sure that the loop does not spend too much time
            Jump j;

            while (bigSubnetworks.Count < 5000 || count == 50)
            {
               j = new Jump(Convert.ToString(bigSubnetworks.Count));
               j.ShowDialog();

               // Genearte and merge topAlpha Genes of two phenotypes
               List<List<string>> topAlphaGenes1 = retrieveTopAlphaPercentGene(alpha, sampleStart,
phenotype1); // Select List of Top alpha genes for each sample; 1000 is alpha, 3 is start position of sample,
"genechip_normal" is table name
               List<List<string>> topAlphaGenes2 = retrieveTopAlphaPercentGene(alpha, sampleStart,
phenotype2); // Select List of Top alpha genes for each sample; 1000 is alpha, 3 is start position of sample,
"genechip_normal" is table name
               List<List<string>> topAlphaGenes = new List<List<string>>(); // Total List
               topAlphaGenes.AddRange(topAlphaGenes1);
               topAlphaGenes.AddRange(topAlphaGenes2);

               // Randomly relabel samples and generate clusters
               topAlphaGenes1 = randomSelect(topAlphaGenes, topAlphaGenes1.Count); // Randomly select
same number of samples of this phenotype from all
               topAlphaGenes2 = topAlphaGenes; // Remaining assigned to another phenotype
```

```
            List<List<string>> topBetaGenes = geneListGeneration(topAlphaGenes1, beta); // Select genes belong to Top beta of all sample's genelist. For each element, 0 is geneName, 1 is G value
            List<List<string>> geneList = addInGeneID(topBetaGenes); // Modify a bit. 0 is geneID, 1 is geneName, 2 is GScore

            List<Subnetwork> subnetworks = new List<Subnetwork>();

            // Go through each pathway, and select genes that both belong to geneList and that pathway, also use these genes form subnetworks
            for (int i = 0; i < 398; i++)
            {
               //Console.WriteLine("Pathway: " + i);
               List<List<string>> tempList = selectGenesOfThatPathway(geneList, i);
               subnetworks.AddRange(formSubnetwork(tempList, i)); // Add newly generated subnetworks in to the big list
            }

            // Scoring
            subnetworks = addScoreVector(subnetworks, topAlphaGenes1, 1);
            subnetworks = addScoreVector(subnetworks, topAlphaGenes2, 2);
            subnetworks = calculateTScore(subnetworks);

            bigSubnetworks.AddRange(subnetworks);

            count++;
         }

         // Move tScores from each subnetwork to the totalTScore
         foreach (Subnetwork s in bigSubnetworks)
            totalTScore.Add(s.getTScore());

         // Sort it from small to big
         totalTScore.Sort();

         // 2nd, Update the pValue of comingSubnetworks according to totalTscore from 1st
         foreach (Subnetwork s in comingSubnetworks)
         {
            double tScore = s.getTScore();
            s.addPValue(calPValue(tScore, totalTScore));
         }

         //2011-10-12
         _totalTScore = new List<double>();
         foreach (double d in totalTScore)
            _totalTScore.Add(d);
         return comingSubnetworks;
      }

      // tScore is the T-Score of one subnetwork
      // totalTScore is all the randomly generated T-Scores using random relabelling
      // The returned value is the pValue of this certain tScore, given the random distribution of totalTScore
      public double calPValue(double tScore, List<double> totalTScore)
      {
         double pValue;
         if (tScore >= 0)
         {
            int greaterThan = getGreaterThan(totalTScore, tScore);
            pValue = Convert.ToDouble(greaterThan) / Convert.ToDouble(totalTScore.Count);
         }
         else
         {
            int smallerThan = getSmallerThan(totalTScore, tScore);
```

```
        pValue = Convert.ToDouble(smallerThan) / Convert.ToDouble(totalTScore.Count);
    }
    return pValue;
}

// This is a regular binary search for positive tScore
// It's recursive, and returns the location of element that is  slightly smaller than or equal to b
// For an example of algorithm
// Please refer to Binary Search under Test Folder
int binarySearchPositive(List<double> a, double b, int start, int end)
{
    int middle = (start + end) / 2; // Middle point
    if (a[start] == b) return start; // Found in right boundary
    if (a[end] == b) return end; // Found in left boundary
    if (a[middle] == b) return middle; // Found in middle
    if (start == end) // Only 1 element left
        return start;
    if (middle == start) // Only 2 elements left
    {
        if (a[start] > b) // Even the left (smallest number) is larger than it
            return start; // Return location. This is good enough. Since = and > are the same
        if (a[end] < b) // Even the right boundary (largest number) is smaller than it
            return end + 1; // Return location + 1
        else
            return end;
    }
    if (a[middle] < b)
        return binarySearchPositive(a, b, middle, end);
    else
        return binarySearchPositive(a, b, start, middle);
}

// This is a regular binary search for Negative tScore
// It's recursive, and returns the location of element that is  slightly bigger than or equal to b
// For an example of algorithm
// Please refer to Binary Search under Test Folder
int binarySearchNegative(List<double> a, double b, int start, int end)
{
    int middle = (start + end) / 2; // Middle point
    if (a[start] == b) return start; // Found in right boundary
    if (a[end] == b) return end; // Found in left boundary
    if (a[middle] == b) return middle; // Found in middle
    if (start == end) // Only 1 element left
        return end;
    if (middle == start) // Only 2 elements left
    {
        if (a[start] > b) // Even the left (smallest number) is larger than it
            return start; // Return location. This is good enough. Since = and > are the same
        if (a[end] < b) // Even the right boundary (largest number) is smaller than it
            return end;
        else
            return end;
    }
    if (a[middle] < b)
        return binarySearchNegative(a, b, middle, end);
    else
        return binarySearchNegative(a, b, start, middle);
}

// Return number of elements in a, which are greater than or equal to b (a is sorted from small to big)
int getGreaterThan(List<double> a, double b)
{
    for (int i = 0; i < a.Count; i++)
```

```
        {
            if (a[i] >= b) return a.Count - i;
        }
        return a.Count - a.Count;
    }

    // Return number of elements in a, which are greater than or equal to b (a is sorted from small to big)
    int getSmallerThan(List<double> a, double b)
    {
        for (int i = a.Count - 1; i >= 0; i--)
        {
            if (a[i] <= b) return i + 1;
        }
        return 0;
    }

    // Randomly Select "numberToSelect" List<string> from topAlphaGenes, delete from it
    public List<List<string>> randomSelect(List<List<string>> topAlphaGenes, int numberToSelect)
    {
        List<List<string>> topAlphaGenes1 = new List<List<string>>();
        Random random = new Random();
        for (int i = 0; i < numberToSelect; i++)
        {
            int totalNumber = topAlphaGenes.Count();
            int select = Convert.ToInt16(Math.Floor(random.NextDouble() *
Convert.ToDouble(totalNumber)));
            topAlphaGenes1.Add(topAlphaGenes[select]);
            topAlphaGenes.Remove(topAlphaGenes[select]);
        }
        return topAlphaGenes1;
    }

    // Output: List of List<double>. Each List<double>, [0] left bound, [1] right bound, [2] count within the
range
    public List<List<double>> getDistribution(List<double> totalTScore)
    {
        List<List<double>> distribution = new List<List<double>>();
        List<double> totalTScore2 = new List<double>(); // And the following 4 lines, to avoid NaN
        foreach (double d in totalTScore)
        {
            if (d < 100000 && d > -100000)
                totalTScore2.Add(d);
        }
        double max = totalTScore2.Max();
        double min = totalTScore2.Min();
        int count = totalTScore.Count;
        double range = (max - min) / Convert.ToDouble(count / 10);

        Console.WriteLine(max + " " + min + " " + count + " " + range);
        Console.Read();

        // Set distribution range, initialize 0
        for (int i = 0; i < count / 10; i++)
        {
            List<double> list = new List<double>();
            list.Add(min);
            list.Add(min + range);
            list.Add(0);
            min = min + range;
            distribution.Add(list);
        }

        // Count number of points in each range
```

```
            foreach (double d in totalTScore)
            {
                foreach (List<double> list in distribution)
                {
                    if (d >= list[0] && d <= list[1])
                    {
                        list[2]++;
                        break;
                    }
                }
            }

            foreach (List<double> list in distribution)
                list[2] /= Convert.ToDouble(count); // Convert it to probability

            return distribution;
        }

        // Sort the subnetworks based on their pValue, from samll to big. If pValue is 0, then sort according to
geneCount
        public List<Subnetwork> sortSubnetworksBasedOnPValue(List<Subnetwork> subnetworks)
        {
            List<Subnetwork> newSubnetworks = new List<Subnetwork>();
            List<Subnetwork> newNewSubnetworks = new List<Subnetwork>();
            int count = subnetworks.Count;

            // Sort according to pValue
            for (int i = 0; i < count; i++) // Run so many times
            {
                double min = 10; // Since the biggest observed pValue is <1
                int selectedOne = 0;
                for (int j = 0; j < subnetworks.Count; j++)
                {
                    if (subnetworks[j].getPValue() <= min)
                    {
                        min = subnetworks[j].getPValue();
                        selectedOne = j;
                    }
                }
                newSubnetworks.Add(subnetworks[selectedOne]);
                subnetworks.Remove(subnetworks[selectedOne]);
            }

            count = newSubnetworks.Count;
            // Sort according to Number of Genes, big to small
            for (int i = 0; i < count; i++) // Run so many times
            {
                double max = 0;
                int selectedOne = 0;
                for (int j = 0; j < newSubnetworks.Count; j++)
                {
                    if (newSubnetworks[j].getPValue() == 0 && newSubnetworks[j].getGenes().Count > max)
                    {
                        max = newSubnetworks[j].getGenes().Count;
                        selectedOne = j;
                    }
                }
                newNewSubnetworks.Add(newSubnetworks[selectedOne]);
                newSubnetworks.Remove(newSubnetworks[selectedOne]);
            }

            // Add remaining subnetworks (pValue != 0)
            for (int i = 0; i < newSubnetworks.Count; i++)
```

```
        {
            newNewSubnetworks.Add(newSubnetworks[i]);
        }

        return newNewSubnetworks;
    }
    #endregion
    #region visualization
    public void drawSubnetworks(List<Subnetwork> SNs, string directory, string dataName, int
typeNumber)
    {
        List<List<List<string>>> selectedRelations = new List<List<List<string>>>();
        List<List<string>> selectedGenes = new List<List<string>>();
        for (int i = 0; i < 400; i++) // Assume there're less than 400 pathways
            selectedRelations.Add(new List<List<string>>());
        foreach (Subnetwork sn in SNs)
        {
            foreach (GeneRelation relation in sn.getGeneRelations())
            {
                int pathwayID = Convert.ToInt16(relation.getPathwayID());
                List<string> relationString = new List<string>();
                relationString.Add(relation.getRelation()[1]); // geneID of first gene
                relationString.Add(relation.getRelation()[2]); // geneID of second gene
                relationString.Add(relation.getRelation()[3]); // interaction type
                selectedRelations[pathwayID].Add(relationString); // add this relation to the pathway it belongs
to
            }
            foreach (Gene gene in sn.getGenes())
            {
                List<string> geneString = new List<string>();
                geneString.Add(gene.getGene()[0]); // geneID
                geneString.Add(gene.getGene()[1]); // geneName
                geneString.Add(gene.getGene()[2]); // GScore
                selectedGenes.Add(geneString); // Potential Problem: genes from different sn may dupilicate
(same gene different GScore)
            }
        }
        draw(selectedRelations, selectedGenes, directory, dataName, typeNumber);
    }

    // Type number: which column does the dataName corresponds to. 4 -> 1, 5 -> 2, etc.
    public void draw(List<List<List<string>>> selectedRelations, List<List<string>> selectedGenes, string
directory, string dataName, int typeNumber)
    {
        int pathwayID = 0;
        StreamWriter relations_sw = new StreamWriter(directory + dataName + "_edges" + ".eda");
        relations_sw.WriteLine("Interation_" + dataName + " " + "(class=string)");

        StreamWriter genes_sw = new StreamWriter(directory + dataName + "_nodes" + ".txt");
        genes_sw.WriteLine("GeneID" + "\t" + "GeneName" + "\t" + "GScore" + "\t" + "InTypeOne" + "\t"
+ "InTypeTwo");
        foreach (List<List<string>> pathway in selectedRelations)
        {
            if (pathway.Count > 0) // Means that it has at least one relation been selected
            {
                StreamWriter pthwy_sw = new StreamWriter(directory + "pthwy_" + pathwayID + ".txt");
                List<string>[] pathwayRS; // Gene relation in a certain pathway (retrieved from pathwayAPI)
                pathwayRS = pathwayAPITableRetrieve("select * from pathway_rs where pathway_id = " +
Convert.ToString(pathwayID) + ";");
                foreach (List<string> a in pathwayRS)
                {
                    pthwy_sw.WriteLine(a[1] + "\t" + a[2] + "\t" + a[3]);
                }
```

```
                pthwy_sw.Close();

                foreach (List<string> relation in pathway)
                {
                    relations_sw.WriteLine(relation[0] + " " + "(" + relation[2] + ") " + relation[1] + " = " +
dataName);
                }
            }
            pathwayID++;
        }
        foreach (List<string> gene in selectedGenes)
        {
            genes_sw.Write(gene[0] + "\t"); // geneID
            genes_sw.Write(gene[1] + "\t"); // geneName
            genes_sw.Write(gene[2] + "\t"); // GScore
            for (int i = 1; i < typeNumber; i++)
                genes_sw.Write("\t");
            genes_sw.Write(dataName);
            genes_sw.Write("\n"); // potential problem
        }

        relations_sw.Close();
        genes_sw.Close();
    }
    #endregion

    #region print
    public void printSubnetworks(List<Subnetwork> subnetworks, string directory, string fileName)
    {
        DateTime currentTime = new DateTime();
        currentTime = DateTime.Now;
        string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;

        List<int> pathwayID = new List<int>();
        StreamWriter sw = new StreamWriter(directory + timeString + " " + fileName + ".rtf");
        sw.WriteLine("Total Pathways: " + countPathway(subnetworks));
        foreach (Subnetwork s in subnetworks)
        {
            if (s.getPValue() <= 0.05)
            {
                sw.WriteLine("Subnetwork:" + s.getID());
                sw.WriteLine("PathwayID:" + s.getPathwayID());
                sw.WriteLine("TScore:" + s.getTScore() + " PValue: " + s.getPValue());
                sw.WriteLine("Genes: ");
                foreach (Gene g in s.getGenes())
                {
                    sw.Write(g.getGeneName() + "(" + g.getGeneID() + "),");
                }
                sw.Write("\nGene Relations: \n");
                foreach (GeneRelation gr in s.getGeneRelations())
                {
                    sw.WriteLine(gr.getRelationString());
                }
                sw.WriteLine("--------------------------------");
            }
        }
        sw.Close();
    }

    // Count how many non-redundent pathways are there in the subnetwork list
    public int countPathway(List<Subnetwork> subnetworks)
    {
```

```
        List<string> pathwayIDs = new List<string>();
        foreach (Subnetwork sn in subnetworks)
        {
            if (!pathwayIDs.Contains(sn.getPathwayID()))
                pathwayIDs.Add(sn.getPathwayID());
        }

        return pathwayIDs.Count;
    }

    // Get the bar chart distribution of SN count. 1, 2-5, 6-10, 11-20, 21-30, 31-40, 41-50, 51-60, 61-70, >70
    public List<int> getBarChart(List<Subnetwork> subnetworks)
    {
        List<int> barChart = new List<int>();
        for (int i = 0; i < 10; i++)
            barChart.Add(0);
        int geneCount = 0;
        foreach (Subnetwork sn in subnetworks)
        {
            if (sn.getPValue() <= 0.05)
            {
                geneCount = sn.getGenes().Count;
                if (geneCount == 1)
                    barChart[0]++;
                else if (geneCount > 1 && geneCount < 6)
                    barChart[1]++;
                else if (geneCount >= 6 && geneCount < 11)
                    barChart[2]++;
                for (int i = 0; i < 6; i++)
                {
                    if (geneCount >= (11 + i * 10) && geneCount < (21 + i * 10))
                        barChart[3 + i]++;
                }
                if (geneCount >= 71)
                    barChart[9]++;
            }
        }

        return barChart;
    }

    public void printSubnetworksCSV(List<Subnetwork> subnetworks, string directory, string fileName)
    {
        DateTime currentTime = new DateTime();
        currentTime = DateTime.Now;
        string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;

        StreamWriter sw = new StreamWriter(directory + timeString + " " + fileName + ".csv");
        sw.WriteLine("ID,PValue,TScore,Count");
        foreach (Subnetwork s in subnetworks)
        {
            if (s.getPValue() <= 0.05)
                sw.WriteLine(s.getID() + "," + s.getPValue() + "," + s.getTScore() + "," + s.getGenes().Count);
        }
        sw.Close();

        sw = new StreamWriter(directory + timeString + " " + "(BarChart)" + fileName + ".csv");
        sw.WriteLine("1, 2-5, 6-10, 11-20, 21-30, 31-40, 41-50, 51-60, 61-70, >70");
        foreach (int i in getBarChart(subnetworks))
            sw.Write(i + ",");
        sw.Close();
    }
```

```
        /// <summary>
        /// returns two list
        /// list1: all the genes in this subnetwork list
        /// list2: all the pathwayID in this subnetworks list
        /// </summary>
        /// <param name="subnetworks"></param>
        /// <returns></returns>
        public List<List<string>> getGeneList(List<Subnetwork> subnetworks)
        {
            List<string> geneList = new List<string>();
            List<string> pathwayList = new List<string>();

            foreach (Subnetwork sn in subnetworks)
            {
                if (sn.getPValue() <= 0.05)
                {
                    foreach (Gene g in sn.getGenes())
                    {
                        geneList.Add(g.getGeneName());
                    }
                    if (!pathwayList.Contains(sn.getPathwayID()))
                        pathwayList.Add(sn.getPathwayID());
                }
            }
            List<List<string>> strList = new List<List<string>>();
            strList.Add(geneList);
            strList.Add(pathwayList);

            return strList;
        }

        public List<string> printGenes(List<Subnetwork> subnetworks, string fileName)
        {
            DateTime currentTime = new DateTime();
            currentTime = DateTime.Now;
            string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;

            StreamWriter sw = new StreamWriter("F:/Temp/FYP/Test/TestData/temp/" + timeString + " " +
fileName + ".rtf");
            List<string> strList = new List<string>();

            foreach (Subnetwork sn in subnetworks)
            {
                if (sn.getPValue() <= 0.05)
                    foreach (Gene g in sn.getGenes())
                    {
                        sw.WriteLine(g.getGeneName());
                        strList.Add(g.getGeneName());
                    }
            }
            sw.Close();

            return strList;
        }

        public void printString(string file, string directory, string fileName)
        {
            DateTime currentTime = new DateTime();
            currentTime = DateTime.Now;
            string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;
```

```
            StreamWriter sw = new StreamWriter(directory + timeString + " " + fileName + ".rtf");
            sw.Write(file);
            sw.Close();
        }
        #endregion

        public double checkOverlap(List<string> list1, List<string> list2)
        {
            int count = 0;
            int minimum;
            list1 = removeDuplication(list1);
            list2 = removeDuplication(list2);
            if (list1.Count < list2.Count)
                minimum = list1.Count;
            else minimum = list2.Count;
            foreach (string str in list1)
            {
                if (list2.Contains(str))
                    count++;
            }
            return Convert.ToDouble(count) / Convert.ToDouble(minimum);
        }

        public List<string> removeDuplication(List<string> strList)
        {
            List<string> tempList = new List<string>();
            foreach (string str in strList)
            {
                if (!tempList.Contains(str))
                    tempList.Add(str);
            }
            // foreach (string str in tempList) 2011-10-31
            //    Console.Write(str + " "); 2011-10-31
            // Console.WriteLine(); 2011-10-31
            // Console.WriteLine(); 2011-10-31
            return tempList;
        }

        // each list is a pair of SN
        // snid1, snid2, overlap, overlap ratio, sn1.count, sn2.count, sn1.pvalue, sn2.pvalue, sn1.PathwayID,
sn2.PathwayID
        public List<List<string>> checkOverlap(List<Subnetwork> snList1, List<Subnetwork> snList2, double
gama)
        {
            List<List<string>> strList = new List<List<string>>();
            int overlap;
            foreach (Subnetwork sn1 in snList1)
            {
                if (sn1.getPValue() <= 0.05)
                    foreach (Subnetwork sn2 in snList2)
                    {
                        if (sn2.getPValue() <= 0.05)
                        {
                            overlap = sn1.overlap(sn2);
                            double overlapRatio = Convert.ToDouble(overlap) /
max(Convert.ToDouble(sn1.getGenes().Count), Convert.ToDouble(sn2.getGenes().Count));
                            List<string> tempList = new List<string>();
                            tempList.Add(Convert.ToString(sn1.getID()));
                            tempList.Add(Convert.ToString(sn2.getID()));
                            tempList.Add(Convert.ToString(overlapRatio));
                            tempList.Add(Convert.ToString(overlap));
                            tempList.Add(Convert.ToString(sn1.getGenes().Count));
```

```
                tempList.Add(Convert.ToString(sn2.getGenes().Count));
                tempList.Add(Convert.ToString(sn1.getPValue()));
                tempList.Add(Convert.ToString(sn2.getPValue()));
                tempList.Add(Convert.ToString(sn1.getPathwayID()));
                tempList.Add(Convert.ToString(sn2.getPathwayID()));

                if (overlapRatio > gama)
                    strList.Add(tempList);
            }
        }
    }

    return strList;
}

// Check a SN list against a gene list, snID, overlap, overlapRatio, count, p, pathwayID
public List<List<string>> checkOverlap(List<Subnetwork> snList, List<string> geneList, double gama)
{
    List<List<string>> strList = new List<List<string>>();
    double overlap;
    double overlapRatio;
    foreach (Subnetwork sn in snList)
    {
        if (sn.getPValue() <= 0.05)
        {
            overlap = checkOverlap(sn, geneList);
            overlapRatio = overlap / Convert.ToDouble(sn.getGenes().Count);
            List<string> tempList = new List<string>();
            tempList.Add(Convert.ToString(sn.getID()));
            tempList.Add(Convert.ToString(overlap));
            tempList.Add(Convert.ToString(overlapRatio));
            tempList.Add(Convert.ToString(sn.getGenes().Count));
            tempList.Add(Convert.ToString(sn.getPValue()));
            tempList.Add(Convert.ToString(sn.getPathwayID()));

            if (overlapRatio > gama)
                strList.Add(tempList);
        }
    }

    return strList;
}
// check how many genes in sn are contained in genelist
// return number of overlap
public double checkOverlap(Subnetwork sn, List<string> geneList)
{
    double overlap = 0;
    foreach (Gene g in sn.getGenes())
    {
        if (geneList.Contains(g.getGeneName()))
            overlap++;
    }

    return overlap;
}

// check if two subnetworks are the same
public List<string> checkSame(Subnetwork sn1, Subnetwork sn2)
{
    List<string> strList = new List<string>();
    List<string> str1 = new List<string>();
    List<string> str2 = new List<string>();
    foreach (Gene g in sn1.getGenes())
```

```
            {
                str1.Add(g.getGeneName());
            }
            foreach (Gene g in sn2.getGenes())
            {
                str2.Add(g.getGeneName());
            }
            double overlap = checkOverlap(str1, str2);
            int count = Convert.ToInt16(overlap * min(str1.Count, str2.Count));

            strList.Add(Convert.ToString(sn1.getID()));
            strList.Add(Convert.ToString(sn2.getID()));
            strList.Add(Convert.ToString(overlap));
            strList.Add(Convert.ToString(count));

            return strList;
        }

        private int min(int a, int b)
        {
            if (a < b) return a;
            return b;
        }

        private double min(double a, double b)
        {
            if (a < b) return a;
            return b;
        }

        private double max(double a, double b)
        {
            if (a > b) return a;
            return b;
        }
    }
}
```

// UI

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using MySql.Data;
using MySql.Data.MySqlClient;
using System.IO;

namespace SNet_V2._0.Module
{
    public partial class UI : Form
    {
        public double _alpha0, _alpha1, _beta, _gama;
        public string _dataset1, _dataset2, _host, _userName, _password, _directory, _outputName;
        List<Subnetwork> sn1 = null;
        List<Subnetwork> sn2 = null;
        string _currentSN = null;
        //private List<Subnetwork> subnetworks;
        Logic logic;
        public UI()
```

```
{
    InitializeComponent();
    _host = "localhost";
    _userName = "root";
    _password = "root";
    _directory = "F:/Temp/FYP/Data/Output/";
    _alpha0 = Convert.ToDouble(alpha0.Text);
    _alpha1 = Convert.ToDouble(alpha1.Text);
    _gama = 0.74;
    _dataset1 = dataset1.Text;
    _dataset2 = dataset2.Text;
    _outputName = outputName.Text;
    logic = new Logic(this);
}

private void button4_Click_1(object sender, EventArgs e)
{
    Database db = new Database(this);
    db.Show();
}

private void button1_Click_1(object sender, EventArgs e)
{
    shield();
    //generateSubentwrok();
}

private void button2_Click(object sender, EventArgs e)
{
    variousAlpha();
}

private void button3_Click(object sender, EventArgs e)
{
    mergeSubnetwork();
}

private void shield()
{
    List<List<Subnetwork>> snList1 = shield(1262, 1893, "dmd_haslettdata");
    StreamWriter sw = new StreamWriter(_directory + "1.1.txt");
    sw.Close();

    List<List<Subnetwork>> snList2 = shield(2228, 3342, "dmd_pescatoridata");
    string str1 = compareDifference("dmd", snList1, snList2);
    StreamWriter sw1 = new StreamWriter(_directory + "compareResult_dmd.txt");
    sw1.Write(str1);
    sw1.Close();

    snList1 = shield(1262, 1893, "lung_bhattdata");
    snList2 = shield(2416, 3625, "lung_garberdata");
    string str2 = compareDifference("lung", snList1, snList2);
    sw1 = new StreamWriter(_directory + "compareResult_lung.txt");
    sw1.Write(str2);
    sw1.Close();

    snList1 = shield(4300, 6451, "prostate_lapointedata");
    snList2 = shield(1262, 1893, "prostate_singhdata");
    string str3 = compareDifference("prostate", snList1, snList2);
    sw1 = new StreamWriter(_directory + "compareResult_prostate.txt");
    sw1.Write(str3);
    sw1.Close();
```

```
            snList1 = shield(2228, 3342, "subtype_allendata");
            snList2 = shield(1262, 1893, "subtype_marydata");
            string str4 = compareDifference("subtype", snList1, snList2);
            sw1 = new StreamWriter(_directory + "compareResult_subtype.txt");
            sw1.Write(str4);
            sw1.Close();
        }

        private string compareDifference(string dataset, List<List<Subnetwork>> snList1,
List<List<Subnetwork>> snList2)
        {
            List<List<List<string>>> strList1 = new List<List<List<string>>>();
            strList1.Add(logic.getGeneList(snList1[0]));
            strList1.Add(logic.getGeneList(snList1[1]));
            strList1.Add(logic.getGeneList(snList1[2]));

            List<List<List<string>>> strList2 = new List<List<List<string>>>();
            strList2.Add(logic.getGeneList(snList2[0]));
            strList2.Add(logic.getGeneList(snList2[1]));
            strList2.Add(logic.getGeneList(snList2[2]));

            string str = null;
            str += dataset + ":\n";
            double geneOverlap = logic.checkOverlap(strList1[0][0], strList2[0][0]);
            double pathwayOverlap = logic.checkOverlap(strList1[0][1], strList2[0][1]);
            double geneOverlapVariousAlpha = logic.checkOverlap(strList1[1][0], strList2[1][0]);
            double pathwayOverlapVariousAlpha = logic.checkOverlap(strList1[1][1], strList2[1][1]);
            double geneOverlapMerge = logic.checkOverlap(strList1[2][0], strList2[2][0]);
            double pathwayOverlapMerge = logic.checkOverlap(strList1[2][1], strList2[2][1]);

            List<List<List<string>>> snOverlap = new List<List<List<string>>>();
            snOverlap.Add(logic.checkOverlap(snList1[0], snList2[0], _gama));
            snOverlap.Add(logic.checkOverlap(snList1[1], snList2[1], _gama));
            snOverlap.Add(logic.checkOverlap(snList1[2], snList2[2], _gama));

            List<List<List<string>>> snPartialOverlap = new List<List<List<string>>>();
            snPartialOverlap.Add(logic.checkOverlap(snList1[0], logic.getGeneList(snList2[0])[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList1[1], logic.getGeneList(snList2[1])[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList1[2], logic.getGeneList(snList2[2])[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList2[0], logic.getGeneList(snList1[0])[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList2[1], logic.getGeneList(snList1[1])[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList2[2], logic.getGeneList(snList1[2])[0], _gama));

            str += "Gene Overlap: " + Convert.ToString(geneOverlap) + "\n";
            str += "Gene Overlap (Various Alpha): " + Convert.ToString(geneOverlapVariousAlpha) + "\n";
            str += "Gene Overlap (Merged): " + Convert.ToString(geneOverlapMerge) + "\n";
            str += "Pathway Overlap: " + Convert.ToString(pathwayOverlap) + "\n";
            str += "Pathway Overlap (Various Alpha): " + Convert.ToString(pathwayOverlapVariousAlpha) +
"\n";
            str += "Pathway Overlap (Merged): " + Convert.ToString(pathwayOverlapMerge) + "\n";
            str += "Subnetwork Overlap: " + Convert.ToString(snOverlap[0].Count) + "\n";
            str += "Subnetwork Overlap (Various Alpha): " + Convert.ToString(snOverlap[1].Count) + "\n";
            str += "Subnetwork Overlap (Merged): " + Convert.ToString(snOverlap[2].Count) + "\n";
            str += "Subnetwork Partial Overlap: " + Convert.ToString(snPartialOverlap[0].Count) + "\n";
            str += "Subnetwork Partial Overlap (Various Alpha): " +
Convert.ToString(snPartialOverlap[1].Count) + "\n";
            str += "Subnetwork Partial Overlap (Merged): " + Convert.ToString(snPartialOverlap[2].Count) +
"\n";
            str += "Subnetwork Partial Overlap2: " + Convert.ToString(snPartialOverlap[3].Count) + "\n";
            str += "Subnetwork Partial Overlap2 (Various Alpha)2: " +
Convert.ToString(snPartialOverlap[4].Count) + "\n";
            str += "Subnetwork Partial Overlap2 (Merged)2: " + Convert.ToString(snPartialOverlap[5].Count) +
"\n";
```

```
        DateTime currentTime = new DateTime();
        currentTime = DateTime.Now;
        string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;

        StreamWriter sw = new StreamWriter(_directory + timeString + dataset + "_snOverlapRecord.csv");
        sw.WriteLine("snid1, snid2, overlap, overlap ratio, sn1.count, sn2.count, sn1.pvalue, sn2.pvalue,
sn1.PathwayID, sn2.PathwayID");
        foreach (List<string> ls in snOverlap[0])
        {
           for (int i = 0; i < ls.Count; i++)
              sw.Write(ls[i] + ",");
           sw.Write("\n");
        }
        sw.Close();

        sw = new StreamWriter(_directory + timeString + dataset + "_snOverlapRecord(VariousAlpha).csv");
        sw.WriteLine("snid1, snid2, overlap, overlap ratio, sn1.count, sn2.count, sn1.pvalue, sn2.pvalue,
sn1.PathwayID, sn2.PathwayID");
        foreach (List<string> ls in snOverlap[1])
        {
           for (int i = 0; i < ls.Count; i++)
              sw.Write(ls[i] + ",");
           sw.Write("\n");
        }
        sw.Close();

        sw = new StreamWriter(_directory + timeString + dataset + "_snOverlapRecord(Merged).csv");
        sw.WriteLine("snid1, snid2, overlap, overlap ratio, sn1.count, sn2.count, sn1.pvalue, sn2.pvalue,
sn1.PathwayID, sn2.PathwayID");
        foreach (List<string> ls in snOverlap[2])
        {
           for (int i = 0; i < ls.Count; i++)
              sw.Write(ls[i] + ",");
           sw.Write("\n");
        }
        sw.Close();

        sw = new StreamWriter(_directory + timeString + dataset + "_snPartialOverlapRecord.csv");
        sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
        foreach (List<string> ls in snPartialOverlap[0])
        {
           for (int i = 0; i < ls.Count; i++)
              sw.Write(ls[i] + ",");
           sw.Write("\n");
        }
        sw.Close();

        sw = new StreamWriter(_directory + timeString + dataset +
"_snPartialOverlapRecord(VariousAlpha).csv");
        sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
        foreach (List<string> ls in snPartialOverlap[1])
        {
           for (int i = 0; i < ls.Count; i++)
              sw.Write(ls[i] + ",");
           sw.Write("\n");
        }
        sw.Close();

        sw = new StreamWriter(_directory + timeString + dataset +
"_snPartialOverlapRecord(Merged).csv");
        sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
```

```
            foreach (List<string> ls in snPartialOverlap[2])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();

            sw = new StreamWriter(_directory + timeString + dataset + "_snPartialOverlapRecord2.csv");
            sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
            foreach (List<string> ls in snPartialOverlap[3])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();

            sw = new StreamWriter(_directory + timeString + dataset +
"_snPartialOverlapRecord(VariousAlpha)2.csv");
            sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
            foreach (List<string> ls in snPartialOverlap[4])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();

            sw = new StreamWriter(_directory + timeString + dataset +
"_snPartialOverlapRecord(Merged)2.csv");
            sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
            foreach (List<string> ls in snPartialOverlap[5])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();
            return str;
        }

        private string compareDifference(string dataset, List<Subnetwork> snList1, List<Subnetwork> snList2)
        {
            List<List<string>> strList1 = new List<List<string>>();
            strList1 = logic.getGeneList(snList1);

            List<List<string>> strList2 = new List<List<string>>();
            strList2 = logic.getGeneList(snList2);

            string str = null;
            str += dataset + ":\n";
            double geneOverlap = logic.checkOverlap(strList1[0], strList2[0]);
            double pathwayOverlap = logic.checkOverlap(strList1[1], strList2[1]);

            List<List<List<string>>> snOverlap = new List<List<List<string>>>();
            snOverlap.Add(logic.checkOverlap(snList1, snList2, _gama));

            List<List<List<string>>> snPartialOverlap = new List<List<List<string>>>();
            snPartialOverlap.Add(logic.checkOverlap(snList1, logic.getGeneList(snList2)[0], _gama));
            snPartialOverlap.Add(logic.checkOverlap(snList2, logic.getGeneList(snList1)[0], _gama));

            str += "Gene Overlap: " + Convert.ToString(geneOverlap) + "\n";
```

```
            str += "Pathway Overlap: " + Convert.ToString(pathwayOverlap) + "\n";
            str += "Subnetwork Overlap: " + Convert.ToString(snOverlap[0].Count) + "\n";
            str += "Subnetwork Partial Overlap: " + Convert.ToString(snPartialOverlap[0].Count) + "\n";
            str += "Subnetwork Partial Overlap2: " + Convert.ToString(snPartialOverlap[3].Count) + "\n";

            DateTime currentTime = new DateTime();
            currentTime = DateTime.Now;
            string timeString = currentTime.Year + "-" + currentTime.Month + "-" + currentTime.Day + " " +
currentTime.Hour + "." + currentTime.Minute + "." + currentTime.Second + "." + currentTime.Millisecond;

            StreamWriter sw = new StreamWriter(_directory + timeString + dataset + "_snOverlapRecord" +
_currentSN + ".csv");
            sw.WriteLine("snid1, snid2, overlap, overlap ratio, sn1.count, sn2.count, sn1.pvalue, sn2.pvalue,
sn1.PathwayID, sn2.PathwayID");
            foreach (List<string> ls in snOverlap[0])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();

            sw = new StreamWriter(_directory + timeString + dataset + "_snPartialOverlapRecord" +
_currentSN + ".csv");
            sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
            foreach (List<string> ls in snPartialOverlap[0])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();

            sw = new StreamWriter(_directory + timeString + dataset + "_snPartialOverlapRecord2." +
_currentSN + ".csv");
            sw.WriteLine("snID, overlap, overlapRatio, count, p, pathwayID");
            foreach (List<string> ls in snPartialOverlap[3])
            {
                for (int i = 0; i < ls.Count; i++)
                    sw.Write(ls[i] + ",");
                sw.Write("\n");
            }
            sw.Close();


            StreamWriter sw1 = new StreamWriter(_directory + "compareResult_" + dataset + ".txt");
            sw1.Write(str);
            sw1.Close();

            return str;
        }
        /// <summary>
        /// strList: 三 ÿ个?list， €分 ¤别Àe为aoriginal， €variousAlpha， €mergedList； €每?个?list里 ¤?
又®?含?有®D两 ¢?个?list， €分 ¤别Àe为ageneList和 ¨þathwayidList
        /// </summary>
        /// <param name="alpha0"></param>
        /// <param name="alpha1"></param>
        /// <param name="dataset1"></param>
        /// <returns></returns>
        private List<List<Subnetwork>> shield(double alpha0, double alpha1, string dataset1)
        {
            _alpha0 = alpha0;
            _alpha1 = alpha1;
```

```
        _dataset1 = dataset1;

        List<List<Subnetwork>> snList = new List<List<Subnetwork>>();
        /*List<List<List<string>>> strList = new List<List<List<string>>>();

        strList.Add(generateSubentwrok());

        strList.Add(variousAlpha());

        strList.Add(mergeSubnetwork());*/

        snList.Add(generateSubentwrok());

        snList.Add(variousAlpha());

        snList.Add(mergeSubnetwork());

        return snList;
    }

    private List<Subnetwork> generateSubentwrok()
    {
        List<Subnetwork> sn = logic.core(_dataset1 + "_disease", _dataset1 + "_normal", _alpha0, _alpha1,
_beta, 3);
        //List<List<string>> strList = logic.getGeneList(subnetworks);
        logic.printSubnetworks(sn, _directory, _dataset1 + "_disease");
        logic.printSubnetworksCSV(sn, _directory, _dataset1 + "_disease");
        logic.drawSubnetworks(sn, _directory, _dataset1 + "_disease", 1);

        return sn;
    }

    private List<Subnetwork> variousAlpha()
    {
        List<Subnetwork> subnetworks = logic.core(_dataset1 + "_disease", _dataset1 + "_normal", _alpha0,
_alpha1, _beta, 3);

        subnetworks = logic.alpha1alpha2(subnetworks, _dataset1 + "_disease", _alpha1, _beta, 3);
        subnetworks = logic.scoring(subnetworks, _dataset1 + "_disease", _dataset1 + "_normal", _alpha0,
3);
        foreach (Subnetwork sn in subnetworks)
            sn.addPValue(logic.calPValue(sn.getTScore(), logic._totalTScore));
        subnetworks = logic.sortSubnetworksBasedOnPValue(subnetworks);
        List<List<string>> strList = logic.getGeneList(subnetworks);
        logic.printSubnetworks(subnetworks, _directory, _dataset1 + "_disease" + "_VariousAlpha");
        logic.printSubnetworksCSV(subnetworks, _directory, _dataset1 + "_disease" + "_VariousAlpha");
        logic.drawSubnetworks(subnetworks, _directory, _dataset1 + "_disease" + "_VariousAlpha", 2);

        return subnetworks;
    }

    private List<Subnetwork> mergeSubnetwork()
    {
        List<Subnetwork> subnetworks = logic.core(_dataset1 + "_disease", _dataset1 + "_normal", _alpha0,
_alpha1, _beta, 3);

        subnetworks = logic.detectNearSubnetwork(subnetworks, _dataset1 + "_disease", _alpha0, _beta, 3,
_directory);
        subnetworks = logic.scoring(subnetworks, _dataset1 + "_disease", _dataset1 + "_normal", _alpha0,
3);
        foreach (Subnetwork sn in subnetworks)
            sn.addPValue(logic.calPValue(sn.getTScore(), logic._totalTScore));
        subnetworks = logic.sortSubnetworksBasedOnPValue(subnetworks);
```

```
    List<List<string>> strList = logic.getGeneList(subnetworks);
    logic.printSubnetworks(subnetworks, _directory, _dataset1 + "_disease" + "_merged");
    logic.printSubnetworksCSV(subnetworks, _directory, _dataset1 + "_disease" + "_merged");
    logic.drawSubnetworks(subnetworks, _directory, _dataset1 + "_disease" + "_merged", 3);

    return subnetworks;
}

private void button4_Click(object sender, EventArgs e)
{
    Database db = new Database(this);
    db.Show();
}

private void button1_Click(object sender, EventArgs e)
{
    sn1 = null;
    sn2 = null;

    if (checkBox1.Checked == true)
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset1.Text;
        sn1 = generateSubentwrok();
        Status.Text = "done";
    }
    if (checkBox2.Checked == true)
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset2.Text;
        sn2 = generateSubentwrok();
        Status.Text = "done";
    }
    _currentSN = "SNet";
}

private void button2_Click_1(object sender, EventArgs e)
{
    sn1 = null;
    sn2 = null;
    if (checkBox1.Checked == true)
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset1.Text;
        sn1 = variousAlpha();
        Status.Text = "done";
    }
    if (checkBox2.Checked == true)
```

```
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset2.Text;
        sn2 = variousAlpha();
        Status.Text = "done";
    }
    _currentSN = "VariousAlpha";
}

private void button3_Click_1(object sender, EventArgs e)
{

    sn1 = null;
    sn2 = null;
    if (checkBox1.Checked == true)
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset1.Text;
        sn1 = mergeSubnetwork();
        Status.Text = "done";
    }
    if (checkBox2.Checked == true)
    {
        Status.Text = "processing...";
        _alpha0 = Convert.ToDouble(alpha0.Text);
        _alpha1 = Convert.ToDouble(alpha1.Text);
        _beta = Convert.ToDouble(beta.Text);
        _gama = Convert.ToDouble(gama.Text);
        _outputName = outputName.Text;
        _dataset1 = dataset2.Text;
        sn2 = mergeSubnetwork();
        Status.Text = "done";
    }
    _currentSN = "Merge";
}

private void button5_Click(object sender, EventArgs e)
{
    Jump j;

    if (sn1 == null)
    {
        j = new Jump("Subnetwork List 1 is empty!");
        j.ShowDialog();
    }

    else if (sn1 == null)
    {
        j = new Jump("Subnetwork List 2 is empty!");
        j.ShowDialog();
    }

    else
```

```
            {
                compareDifference(_dataset1, sn1, sn2);
            }
        }
    }
}


// Database UI

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SNet_V2._0.Module
{
    public partial class Database : Form
    {
        UI ui;
        public Database(UI _ui)
        {
            ui = _ui;
            InitializeComponent();
            host.Text = ui._host;
            username.Text = ui._userName;
            password.Text = ui._password;
            directory.Text = ui._directory;
        }

        private void button1_Click_1(object sender, EventArgs e)
        {
            ui._host = host.Text;
            ui._userName = username.Text;
            ui._password = password.Text;
            ui._directory = directory.Text;

            this.Close();
        }
    }
}
```

// Pop Window

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SNet_V2._0.Module
{
    public partial class Jump : Form
    {
        public Jump(String text)
        {
```

```
            InitializeComponent();
            label1.Text = text;
        }

        private void button1_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

Follow Up for FYP

2011-11-07

Chai Haoqiang

Modifications Made

In this follow up, two modifications to the algorithm were made: change p-value and remove singleton subnetworks.

Change P-Value: p-value generation was changed from two-sided to one-sided, which means for each tested subnetwork, only count number of random subnetworks whose t-score are bigger than its.

Remove Singletons: in the original output, many significant subnetworks were of size one. They were called singleton. This modification removes such singleton subnetworks from significant list.

Results

Table 1.1 and 2.1 were the original output for gene overlap and pathway overlap. Table 1.2 and 2.2 were the output after modifying p-value generation. Table 1.3 and 2.3 were the output after modifying p-value generation and removing singletons.

From comparison of Table 1.1 and 1.2, it is shown that there was a decrease of gene overlap when p-value generation was changed from two-sided to one-sided.

From comparison of Table 1.1 and 1.3, it is clearly shown that overlap of genes increased for most datasets when singletons were removed. The only exception was ALL, whose gene overlap ratio decreased vastly when singletons were removed. This suggested that in ALL, the overlapped subnetworks were mostly singletons.

For the pathway overlap, it is shown that both p-value modification and removing singletons has resulted a decrease on pathway overlap.

	SNet	Various Alpha	Neighbour Merged
DMD	27%	20%	32%
Lung	25%	26%	29%
Prostate	42%	44%	43%
All	27%	29%	31%

Table 1.1 Gene Overlap for Two Tailed P-value, Singleton Subnetworks Inclusive

	SNet	Various Alpha	Neighbour Merged
DMD	20%	24%	30%
Lung	19%	23%	26%
Prostate	35%	38%	34%
All	12%	12%	29%

Table 1.2 Gene Overlap for One Tailed P-value, Singleton Subnetworks Inclusive

	SNet	Various Alpha	Neighbour Merged
DMD	38%	26%	38%
Lung	29%	28%	33%
Prostate	46%	48%	45%
All	9%	13%	28%

Table 1.3 Gene Overlap for One Tailed P-value, Singleton Subnetworks Exclusive

	SNet	Various Alpha	Neighbour Merged
DMD	74%	71%	71%
Lung	83%	84%	83%
Prostate	96%	96%	95%
All	84%	80%	75%

Table 2.1 Pathway Overlap for Two Tailed P-value, Singleton Subnetworks Inclusive

	SNet	Various Alpha	Neighbour Merged
DMD	68%	67%	64%
Lung	82%	85%	82%
Prostate	95%	96%	92%
All	74%	79%	74%

Table 2.2 Pathway Overlap for One Tailed P-value, Singleton Subnetworks Inclusive

	SNet	Various Alpha	Neighbour Merged
DMD	48%	42%	68%
Lung	62%	60%	68%
Prostate	89%	90%	89%
All	42%	44%	52%

Table 2.3 Pathway Overlap for One Tailed P-value, Singleton Subnetworks Exclusive

Follow Up 2 for FYP

2011-11-18

Chai Haoqiang

Modifications Made

In this follow up, the singletons (subnetworks contain only one gene) were removed, and the subnetworks overlap were checked again for both Perfect Match and Partial Match.

The reason for doing this modification is because in the original output, singletons contribute to a major part of all subnetworks, and this introduces noises for overlap checking.

Results and Discussion

Table 1 shows the Perfect Match found for each datasets after removing singletons, and Table 2 shows the Partial Match found for each datasets after removing singletons. It was shown that Neighbour Merge produces more Perfect Matches and Partial Matches than SNet. Recall that when singletons were not removed, Neighbour Merge produces less subnetworks overlap than SNet in most cases; it can be concluded that the original high overlap for SNet was indeed resulted from large portion of singletons. On the other side, this result supported the hypothesis that Neighbour Merge and Various Alpha performs better than SNet in the way of producing higher overlaps for large subnetworks.

	SNet	Various Alpha	Neighbour Merged
DMD	3	2	4
Lung	13	15	7
Prostate	34	89	42
All	0	7	4

Table 1 Perfect Match within Subnetworks Lists Produced by SNet, Various Alpha and Neighbour Merge after Removing Singletons

	SNet	Various Alpha	Neighbour Merged
DMD-Has	2	2	10
DMD-Pec	9	22	30
Lung-Bha	27	37	32
Lung-Gar	6	11	6
Prostate-Lap	7	17	21
Prostate-Sin	106	157	138
ALL-All	12	41	18
ALL-Mar	2	14	11

Table 2 Partial Match within Subnetworks Lists Produced by SNet, Various Alpha and Neighbour Merge after Removing Singletons