Project - MAGIC - Preprocessing Phase

Submitted by: Faddy Saad.
MAGIC's site: http://magicmapping.sourceforge.net

Introduction

MAGIC, (a rearrangement of an Integrative and Accurate method for Comparative Genome Mapping) is a tool for comparing two genomes (Swidan et. al., PLoS CB, 2006). It consists of two phases: A preprocessing phase for identifying maximal similar segments, and a mapping phase for clustering and classifying these segments. MAGIC's main novelty lies in its biologically intuitive clustering approach, which aims towards both calculating reorder-free segments and identifying orthologous segments. MAGIC also handles nuisance cross overlaps resulting from duplications that occurred before the speciation of the considered organisms from their most recent common ancestor. MAGIC is both robust and scalable: The former is asserted with respect to its initial input and with respect to its parameters' values. The latter enables applying MAGIC to distantly related organisms and to large genomes.
MAGICs improvements allow a comprehensive study of the diversity of genetic repertoires resulting from large-scale mutations, such as, indels and duplications, including explicitly transposable and phagic elements. MAGIC enables to conduct a comprehensive analysis of the different forces shaping genomes from different clades, and to quantify the importance of novel gene content introduced by horizontal gene transfer relative to gene duplication in genome evolution.
MAGIC's result can be used as well to investigate the breakpoint distribution in genomes.

In this document we describe a C/C++ implementation of the first phase of MAGIC, i.e., the preprocessing phase (PrePro). For more details on MAGIC we refer the reader to (Swidan et. al., PLoS CB, 2006).

Preprocessing Phase - Description

The input to the preprocessing phase constitutes two anchor files containing labeling of genomic segments of interest - which we refer to as KOs - in the two organisms, as well as two files containing the genomic sequence of the organisms (in the meanwhile MAGIC can handle only unichromosomal genomes). The output of the preprocessing phase is a comprehensive table of all maximal similar segments between the two genomes. Manipulating the input files to calculate the comprehensive table is done in six steps. We first give a verbal description of these steps, and then provide a detailed formal algorithmic description:

1. **Calculating Unique Common Anchors**: The input anchor files (named *.anchor) contain tab-delimited tables with the following columns "Orientation Start End KO". The "Orientation" column describes the strandedness of the segment (and equals "+" or "-"), the "Start" ("End") refers to the starting (ending) coordinate in the genomic sequence (and equals to a natural number or an integer), and the KO column contains the label of the entry (i.e., a general string). In this step, we seek to find the unique common anchors between these two files. The files, however, may

contain illegal entries. Hence, we first scan the files, remove these illegal entries, and output a new clean anchor file (named *.anchor.clean).
Examples of Illegal entries:

+	12312hhsh	12123333	KO327
A	12121	123121	KO2311
-	983738	9918281	OK122

To calculate the unique common anchors, we are assisted by the standard template library (STL) to use data structures preserving the uniqueness quality of their objects (in our case, anchors' labels).
The output of this part is saved by default in a file (Intersection.res).
The time complexity of this step is O(N*logN) (where N is the number of legal entries).

2. **Calculating KO Induced Segments (KISs)**: Based on the common unique anchors, one can calculate a coarse mapping between the two genomes by generating runs of successive anchors (with identical orientation). To do that, we search for maximal anchor runs (based on the result of Step 1). We refer to the resulting segments as KO induced segments (KISs).
By default, the program generates a file (KISs.res) containing the final result. This file has the following format:
KISi Start1 End1 Start2 End2 Orientation

KISi: A unique name for the KIS.
Start1: The starting coordinate of the entry in the first genome.
End1: The ending coordinate of the entry in the first genome.
Start2: The starting coordinate of the entry in the second genome.
End2: The ending coordinate of the entry in the second genome.
Orientation: The relative orientation of the KOs assembling the KIS between the two genomes (note that orientation here differs from the one given in the anchor files).
The time complexity of this step is O(N*logN) (N is the number of KOs).

3. **Global Alignment on KISs**: The KISs produced in Step 2 represent a coarse mapping. To verify the similarity of the segments in the KIS table, we globally align them. Since the genomic segments might be very long, exact global alignment methods cannot be used. Instead, we use MAFFT (Katoh et. al. , NAR, 2005; see also http://www.biophys.kyoto-u.ac.jp/~katoh/programs/align/mafft/) , a fast heuristic (with moderate memory requirements) for the global alignment.
MAFFT's time complexity:
MAFFT requires CPU time effectively proportional to the average sequence length *L* for amino acid or nucleotide sequence alignments consisting homologues of a single gene.
Both memory and time complexities of MAFFT are O(L) (Katoh et. al. , NAR, 2005).

4. **Extracting Unmatched Regions**: To refine the coarse KIS mapping, we search for unmatched regions in the results of the global alignments performed in Step 3. The unmatched regions are either segments that have undergone reordering (and thus disturb the collinear global alignment) or indels. To distinguish between the two cases, we search the other genome for hits based on these regions (in Step 5). If hits are found, the unmatched regions might result from reordering mutations (this is determined in the mapping phase), or otherwise they are most likely indels. The unmatched regions are

constructed by joining close gaps in each of the two genomes separately (< gapJoinLen parameter, see Table 1). Afterwards, sufficiently long unmatched regions (>gapExtractLen parameter, see Table 1) that overlap are merged together. Then, the merged unmatched regions are extracted, and the KIS is broken at the corresponding points. This step results in a new refined table, referred to as KIS*.
The time complexity of this step is: O(N*M) (where N is the number of KISs and M is the maximum number of gaps over all globally aligned KISs).

5. **Extracting Breakpoints and Local Alignments:** In Step 4 we have refined the KIS table by validating the similarity of existing KISs. In this step, on the other hand, we try to extend the existing table by adding new similarities to it. To do that, we extract all breakpoints from the KIS* table and locally align them against the other genome (e.g., breakpoint regions extracted from the first genome are locally aligned against the second genome and vice versa).
We have used YASS (Noe and Kucherov, NAR, 2005; see also http://bioinfo.lifl.fr/yass) for the local alignments.
Extracting breakpoints time complexity: O(N) (where N is the number of KISs in the table KIS*).

6. **Stitching Hits, Global Alignment, and Extracting Unmatched Regions**: To calculate new potential maximal similar segments, each set of local matches is scanned for hits that can be stitched together. Stitched hits need to have the same orientation. To determine which hits to stitch, three quantities are considered. The 1st is the difference between the distances of the two hits in the two organisms. Intuitively, the distance between two hits in a given organism is calculated by subtracting the end of one hit from the start of the other. If the segments between the two hits are similar, the distances between the two hits in the two organisms should be similar. Therefore, hits with an excessive difference in their distances (> stitchDifference parameter, see Table1) are not stitched. Second, we consider the distance between the two hits. If two segments in the two organisms are similar, one would expect to find many hits when locally aligning them. Thus, the distance between consecutive hits in similar segments should be short. Therefore, hits with too large a distance (> stitchDistance parameter, see Table1) are not stitched. Finally, after stitching hits that fulfill the above two requirements, we keep stitched segments that are long enough (> stitchMinLen parameter, see Table1).
In order to validate the similarity of the stitched hits, we globally align them (using MAFFT) and extract unmatched regions, and eventually add them to the KIS* table to get a new table of comprehensive set of maximal similar segments between the two given genomes.
(We use SIS - stitched induced segments - as a prefix to label entries corresponding to stitched hits in order to distinguish them from the KISs).
Time complexity: O(N^2) (where N is the number of hits).

PrePro Parameters

Parameter	***Value***	***Used For***	***In***
gapJoinLen	110bp	Joining close gaps	Extracting unmatched regions from global alignments (Step 4.)
gapExtractLen	200bp	Extracting long unmatched regions	
stitchDifference	2000bp	Finding hits having similar distances	Stitching hits resulting from local alignments and extracting long ones (Step 6.)
stitchDistance	15000bp	Finding close hits	
stitchMinLen	200bp	Removing short stitched hits	

Table 1

Project - MAGIC - Preprocessing Phase

Preprocessing Phase Formal Algorithmic Description

Step 1 (Calculating Unique Common Anchors)

Given two anchor tables, T1 and T2, PrePro scans both table entries and checks their correctness as follows:

1. T1 and T2 should contain four columns (as described above in step 1).

2. All values in the "Orientation" column should equal either the character '+' or the character '-'.

3. All values in the "Start" and the "End" columns should equal a natural number – i.e., an integer (no characters allowed).

4. All values in the "KO" column (which correspond to the labels of the segments) should have "KO" as their prefix, followed by a natural number (no spaces between the "KO" and the natural number are allowed).

5. Each KO should be unique in its file.

- "Cleaning" the tables:

The following procedure is applied to both T1 and T2:

```
Foreach entry i in T
        Set entry = extract( T( i ) )
        Check_it(entry)                 /*Checks the entry's correctness according to the above*/
end
```

Let T1* and T2* be the new "cleaned" tables.
Both of T1* and T2* are held in an instance of set<T> (C++ STL's data structure) where T is an instance of type SEQ (note that the set is sorted according to the KO labels).
For more information on SEQ see files seq.h and cleanfile.h.
See also "Software Design" section.

- Generating the intersection between T1* and T2*:

Finding the intersection between the two tables is done by applying the function "set_intersection" from the "algorithm" library in the C++ STL package.
Both T1* and T2* tables are restricted to the result of the intersection, and are sorted according to the "Start" field. We refer to the resulting tables as InterSec1 and InterSec2, respectively.

Step 2 (Finding KISs)

Let i be an entry in the table InterSec1, which is sorted according to the first organism's "Start" column. In order to find i's corresponding entry (i.e., the entry with the same KO label), say j, in the table InterSec2 (which is sorted according to the second organism's "Start" column), we map the entry's indices between the two tables using an array as follows:

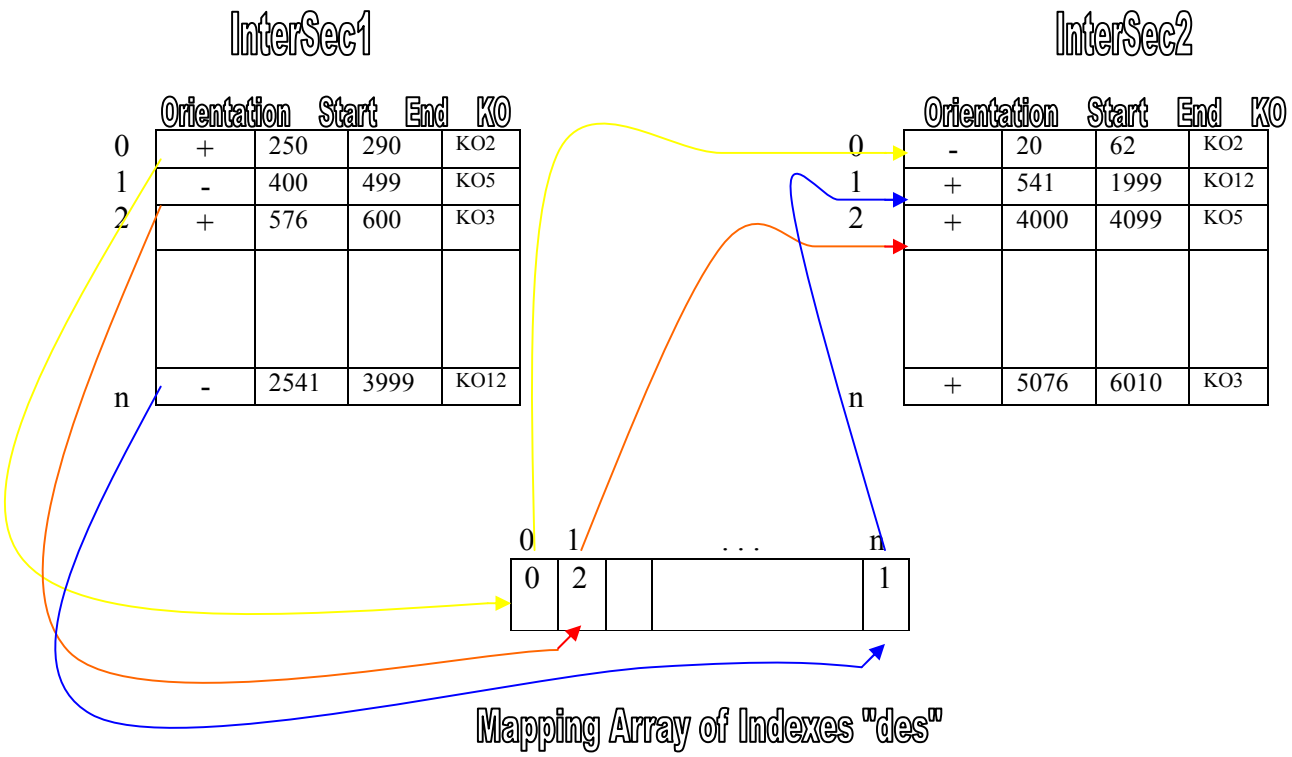

Figure 1

Denote by "des" the mapping array of the entry indices (see Fig. 1). Then, the corresponding index for i (i.e., the index of the same entry when sorting the table according to the second organism) in InterSec2 is given by des[i]. The "des" array is calculated as followed: For each index in the table "InterSec1", we search (using binary search) for its corresponding index in the table "InterSec2". The search is done based on the entry's label (recall that the labels are unique).
The binary search is done on a third auxiliary table, sorted by the KO field (Each entry of this table represents an entry of InterSec2 and its index in the table InterSec2).
The Time complexity of building the "des" array is O(NlogN) (where N is the number of entries in the table "InterSec1").

After calculating the "des" array, we are ready to identify the KISs. Finding the KISs is done according to the following procedure:

We say that two entries are successive in the first organism if they appear consecutively in table InterSec1. We say that two entries are successive in the second organism either if they appear in an incremental order in table InterSec2 and they share a similar orientation or they appear in a detrimental order in table InterSec2 and they share a dissimilar orientation. A KIS corresponds to a maximal (with respect to the containment order) group of entries that are successive in both the first and the second organisms. Such a group of entries is also referred to as a run. To identify the runs, we scan the 1st table (InterSec1) incrementally, while identifying for each given entry its corresponding entry in the 2nd table (InterSec2). When we identify two entries that are successive in both the first and the second table, we join them into a single modified entry (with segments in both organisms that span the ranges of the original two entries) and continue the scan with the newly generated one.
We refer to such entry as "KIS".
Generating a KIS table is done with the format shown in the previous section.

Step 3 (Global Alignment of KISs)

In this step we do the following:

1. Scan the KIS table (the result of step 2) and extract the genomic segments corresponding to the KISs.
2. If the KIS has a negative orientation, we reverse complement the **2nd** sequence (according to the standard Watson-Crick pairing: A<=>T , G<=>C) .
3. Call on MAFFT's global alignment function, and save the result of the alignment.
4. Go to 1 if there is still unaligned KISs.
5. Return the result of the global alignments of all KISs.

Step 4 (Extracting Unmatched Regions)

In order to extract unmatched regions, we search for proximal gaps in the global alignment results from step 3. In addition, we refine the KIS table according to the extracted unmatched regions.

Extracting unmatched regions includes 6 steps:

1. Finding gaps in both organisms in a global alignment result of a given KIS.
2. Joining proximal gaps in each organism separately.
3. Extracting sufficiently long gaps.
4. Joining intersecting gaps found according to the first and second organism.
5. Transforming the gaps coordinates.
6. Refining the KIS table.

In the following, we describe each step in more detail.
Consider KISi, one of the KISs in the table. Let res1 and res2 be the results of the global alignment on KISi in the first and second organisms, respectively.

1. To identify the gaps, we scan res1 and res2 and produce two tables - gaps1 and gaps2 - containing the coordinates of the gaps in res1 and res2, respectively. Note that these coordinates are relative to res1 and res2. Denote by n and m the number of rows in the tables gaps1 and gaps2, respectively.

2. Joining proximal gaps is done on the gaps1 and gaps2 tables separately (in the following "gaps" stands either for gaps1 or gaps2).

```
Set i = 1

while i < the number of rows  in "gaps"
  if the distance between "gaps"[i] and "gaps"[i-1] <= gapJoinLen  /*A predefined threshold*/
            start of i = start of i-1                              /*Joining*/
        remove "gaps"[i] from "gaps"                               /*Not relevant anymore*/
        endif
        Set i = i+1
end
```

3. Here we extract sufficiently long gaps from the modified gaps1 and gaps2 tables.

```
Set i = 0

while i < the number of rows  in "gaps"
        if length of "gaps"[i] < gapExtractLen          /*A predefined threshold*/
                remove "gaps"[i] from gaps
        endif
end
```

4. In order to find and join intersecting gaps inbetween the gaps1 and gaps2 tables, we first merge-sort the two tables in an ascending order according to their start coordinate. (Note that both tables are already sorted in an ascending order.) We refer to the merged table as "gaps".

Let n be the number of rows in "gaps".

```
Set i = 1
while i < n
        if end of "gaps" [i] <= end of "gaps" [i-1]
                start of "gaps" [i] = start of "gaps" [i-1]          /*Unite between gaps*/
                end of "gaps" [i] = end of "gaps" [i-1]
                remove "gaps" [i-1] from "gaps"
        else
                if start of "gap" [i] <= end of "gaps" [i-1]
                        start of "gaps" [i] = start of "gaps" [i-1]  /*Unite between gaps*/
                        remove "gaps" [i-1] from "gaps"
        endif
        Set i = i+1
end
```

5. The coordinates of each gap in the resulting "gap" table are relative to the results of the global alignment (res1 and res2). In order to get the coordinates (relative to the sequences with no dashes) we apply the following algorithm:

We use an auxiliary array of |res1| integers where each cell "i" is initialized with the number of dashes found in the prefix of res1 (the prefix's length is "i").
We scan the gap table of res1 and for each entry (including the coordinates of the gaps) we reduce the appropriate number of dashes.
The algorithm is demonstrated in the following procedure:
* Note that the procedure is invoked on both sequences gaps tables (res1 and res2, using res1 above is for illustration only).

```
Set i = 0
while i < n                                          /* n is the number of entries in the
                                                      gaps table*/
        start of "gaps" [i] = start of "gaps" [i] – "tmp" [i]   /*We ignore the dashes and this
                                                      way we get the absolute value of
                                                      the coordinate*/
        end of "gaps" [i] = end of "gaps" [i] – "tmp" [i]
        if start of "gaps" [i] == end of "gaps" [i]
                don't add this gap to gaps table
        endif
        add this gap to gaps table
        Set i = i+1
end
```

6. After step 5 we have two refined tables of gaps:
    1. gaps1
    2. gaps2

Note that both tables have the same number of gaps because they have both been created from the table gaps, which is the result of step 4.

Refining KISi is done as followed:

We scan the entries of both gaps tables and extract each gap of table "gaps1" out of the first sequence of KISi and each gap of table "gaps2" out of the second sequence of KISi. This way we divide KISi into new segments representing the KISi partitions.
Note that we take into consideration the orientation of KISi and divide it accordingly (if it was '+' then we divide both sequences of KISi as is, but if it's '-' then we take into account the reverse complement of the 2nd sequence of KISi).

* If there are no gaps then we don't divide KISi.
* After refining all the KISs in the KIS table we get a new table referred to as KIS*.
* In case that the orientation is '-' the refinement differs from the case when it's '+' and that's because of the transformation we've done on the 2nd sequence before performing global alignment (see step 3).
  Figure 2 demonstrates and clarifies this point.

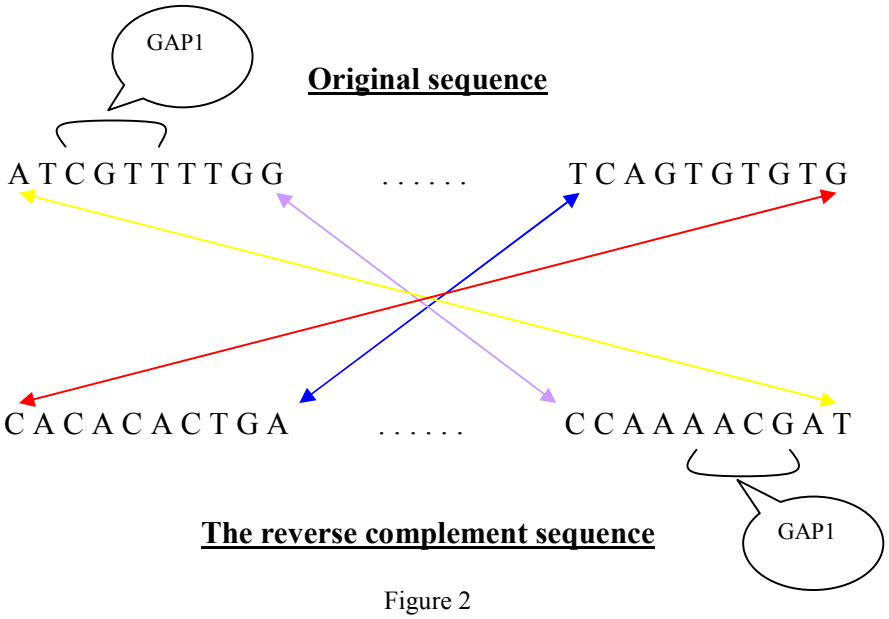

Figure 2

Step 5 (Extracting BreakPoints and Local Alignment)

As a result of performing the 4th step, we get a new and refined KIS table (KIS*).
In this step we locally align the breakpoints of KIS* against the whole genome taking into consideration the circularity of both genomes (Figure 3 illustrates the breakpoints of KIS*).

The flow chart of this step is as followed:

1. Sort the KIS* table in an ascending order according to the "Start1" field of the table.
2. Extract the breakpoints of the 1st sequence from the 1st genome.
3. Sort the KIS* table in an ascending order according to the "Start2" field of the table.
4. Extract the breakpoints of the 2nd sequence from the 2nd genome.
5. Locally align the breakpoints of the 1st sequence against the 2nd genome.
6. Locally align the breakpoints of the 2nd sequence against the 1st genome.

* Note that as mentioned before, the local alignments are done using YASS.
* The table we get after the local alignment includes 5 fields:
  Label Start1 End1 Start2 End2 Orientation

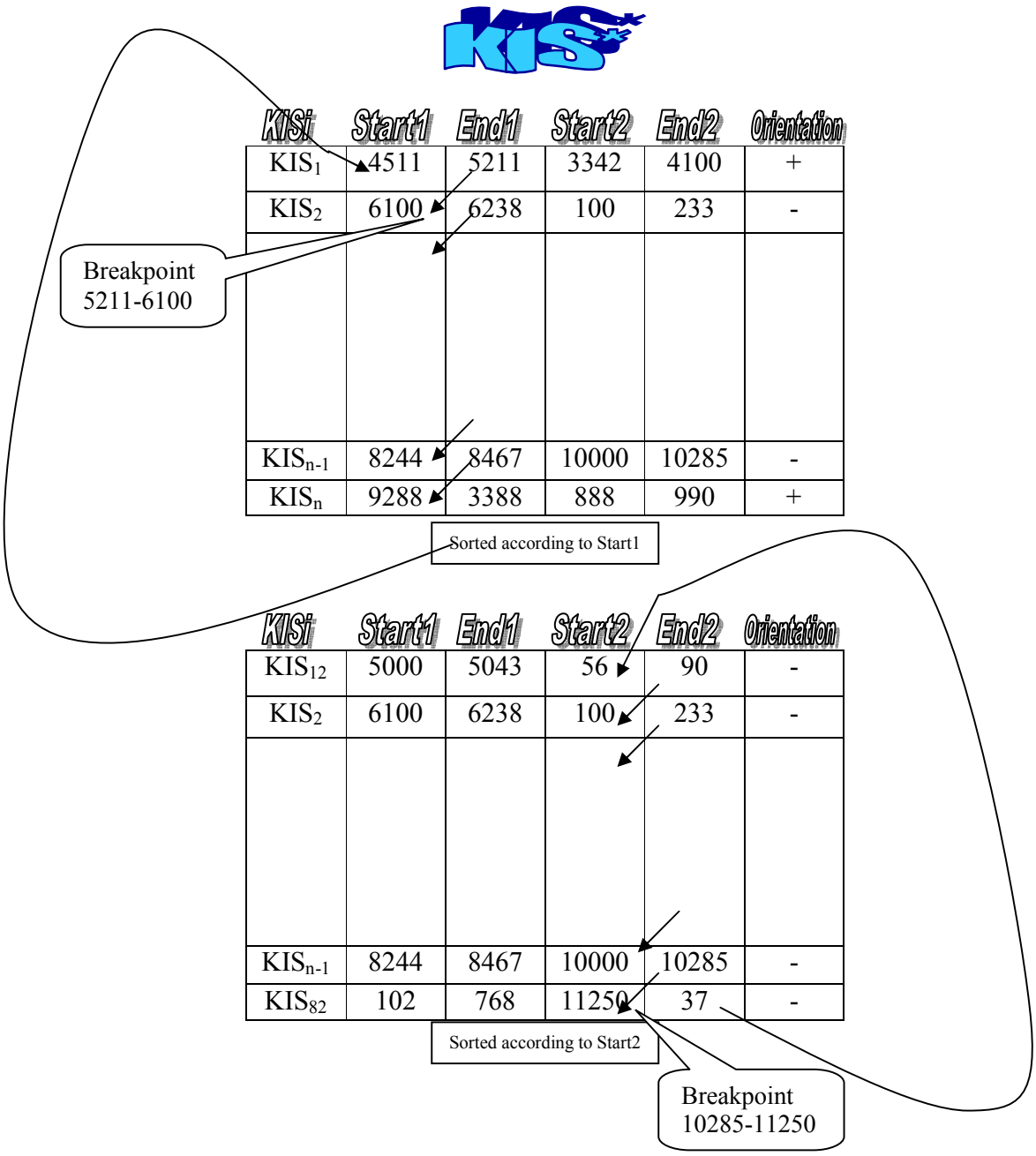
KISi Start1 End1 Start2 End2 Orientation
KIS1 4511 5211 3342 4100 +
KIS2 6100 6238 100 233 -
Breakpoint
5211-6100
KISn-1 8244 8467 10000 10285 -
KISn 9288 3388 888 990 +
Sorted according to Start1
KISi Start1 End1 Start2 End2 Orientation
KIS12 5000 5043 56 90 -
KIS2 6100 6238 100 233 -
KISn-1 8244 8467 10000 10285 -
KIS82 102 768 11250 37 -
Sorted according to Start2
Breakpoint
10285-11250

Figure 3

Project - MAGIC - Preprocessing Phase

Step 6 (Stitching Hits, Global Alignment and Extracting Unmatched Regions)

In this part we stitch local matches (the local alignment results from the previous step).
We scan the hits' table and for each pair of hits "i" and "j" where i<j, we perform the following:
If their orientation differs then we move on to the next pair.
Else we calculate the distance between the sequences of hits "i" and "j" belonging to the first genome (equals 0 if "j" overlaps "i") and refer to it as ℓ_1.
If ℓ_1 is greater than stitchDistance (see Preprocessing phase – Description, part 6) then we move on to the next pair.
Else we calculate the distance between the sequences of hit "i" and "j" belonging to the second genome (equals 0 if "j" overlaps "i") and refer to it as ℓ_2.
If the segments between the two hits are similar, the distances between the two hits in the two organisms should be similar. Therefore if "i" and "j" have an excessive difference in their distances ($|\ell_1 - \ell_2|$ > stitchDifference , see Table1) they won't be stitched and we move on to the next pair.
Else we stitch "i" and "j" by uniting their coordinates and move on to the next pair.
Finally, after stitching hits, we keep stitched segments that are long enough (> stitchMinLen parameter, see Table1).
In order to validate the similarity of the stitched hits, we globally align them (using MAFFT) and extract unmatched regions, and eventually add them to the KIS* table to get a new table of comprehensive set of maximal similar segments between the two given genomes.
(We use SIS - stitched induced segments - as a prefix to label entries corresponding to stitched hits in order to distinguish them from the KISs.)

Note:
The following steps were added in order to avoid dealing with circularity issues:

1. Before calculating ℓ_1 and ℓ_2, we check if both hits are circular, and in case they are we add the length of the appropriate genome to the ending coordinate of the suitable segment of the hit.
2. If the 2nd segment of hit "i" is circular and the distance between the 2nd segments of "i" and "j" is less than stitchDistance then we add the length of the 2nd genome to the 2nd segment of hit "j" in order to keep the same relation between the distance and the stitchDistance variable.
3. All previous steps are done taking into consideration the orientation of the hit.

Software description:

The application has two user interfaces:
1. command line.
2. MAGIC GUI .

- Following is a description of the parameters given as input to the application when running in the command line interface:

* pre-pro [options] path1/file1.anchor path2/file2.anchor path3/

Optional parameters:

1. -help : Displays a Help screen.

2. gJL <long> : Initializes gapJoinLen variable with <long> which is used for joining close gaps (Default 110).
3. gEL <long> : Initializes gapExtractLen variable with <long> which is used for extracting long unmatched regions (Default 200).
4. sDiff <long> : Initializes stitchDifference variable with <long> which is used for finding hits having similar distances (Default 2000).
5. sDis <long> : Initializes stitchDistance variable with <long> which is used for finding close hits (Default 15000).
6. sMin <long> : Initializes stitchMinLen variable with <long> which is used for removing short stitched hits (Default 200).
7. -circular <0/1><0/1> : -circular 00 says that both genomes are not circular. (10-first genome is circular,01-2nd genome is circular,11-both are circular, Default 00).
8. -eValue <real> : Determines the e-value threshold used for filtering local alignment hits. The parameter is passed as is to YASS (Default 0.01).

Note:

PRE-PRO assumes the existence of the files including the whole genomes (file1.genome and file2.genome) under path1 and path2 respectively.
The final result of PRE-PRO will be saved in the file path3/finalPREPRO.result .
Additional results will be also saved under path3 :

- KISs.res: The KIS* table.
- Intersection.res: The intersection result of KO's.
- file1.anchor.clean: The "clean" result of the file file1.anchor.
- file2.anchor.clean: The "clean" result of the file file2.anchor.

- When the application is activated from MAGIC's GUI, these parameters are set in the GUI (For more information please refer to the GUI documentation at http://magicmapping.sourceforge.net).

Software design:

Diagram 1 describes the relationships between the main classes in the software (class diagram).

Following is a description of the main classes:

SEQ:
The main purpose of this class is to hold all relevant information on a certain sequence.
Main fields in SEQ class are:

- The sequence's coordinates (start and end).
- The sequence's orientation.
- The sequence's KO.

KIS:
The main purpose of this class is to hold all relevant information on a certain KIS.
Main fields in KIS class are:

- The KIS's orientation.
- Two vectors of SEQ objects (the segments of both sequences of the KIS)

KISES
The main purpose of this class is to create the KIS table.
Main fields in KISES class are:

- The KIS table.

The KISES class is responsible on performing step 2 (see Preprocessing Phase - Description) .

lightKIS
The main purpose of this class is to hold all relevant information on a certain KIS or SIS.
Main fields in lightKIS class are:

- The coordinates of both sequences of the KIS/SIS.
- The orientation of the KIS/SIS.
- The name of the KIS/SIS.

CleanFile
The main purpose of this class is to legalize the given *.anchor files and provide new "clean" files with relevant entries only.
Main fields in CleanFile class are:

- The path and name of the file *.anchor.
- A set of all legal entries (each entry is preserved in a SEQ object).

The CleanFile class is responsible in performing step 1 (see Preprocessing Phase - Description)

PREPRO
PREPRO is the main class which includes all major methods.
Main fields in PREPRO class are:

- Both paths and file names of both *.anchor files.
- Both genomes of the files *.genome .

The PREPRO class is responsible on performing steps 3-6 (see Preprocessing Phase - Description).

Software Dependencies

PrePro running time has a direct dependency with the running time of both MAFFT (Global alignment method) and YASS (Local alignment method).
PrePro also depends on the functionality of the imported STL's libraries and data structures which were used in the implementation of the preprocessing phase.

Future Work

Developing and improving PrePro would be expressed by parallelizing the independent calls for MAFFT and YASS, this would result in a major improvement in the running time of PrePro.

Class Diagram

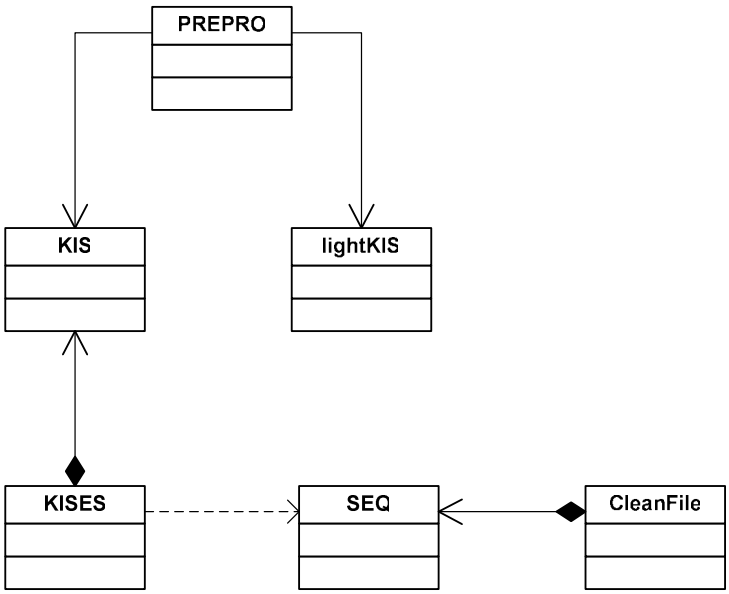

Diagram 1

Flow Chart of The PreProcessing Phase