May 18, 2011

FastMRP

1 Methods

1.1 Algorithmic Ideas

To be able to compute MRP matrices for the largest datasets available today (tens of thousands of sequences), memory and running time need to be optimized simultaneously. To this end, we have used two main ideas in our algorithm.

On the larger datasets available today, an MRP matrix can contain hundreds of thousands of columns and tens of thousands of rows, and these numbers will likely grow dramatically in the future. Therefore, constructing the whole MRP matrix in memory and then writing it to a file is not feasible. Neither is keeping a larger number of question marks in memory an efficient approach. The simple insight that MRP matrices are typically very sparse can help reducing the memory footprint dramatically. A memory efficient approach can be devised to keep in memory two sets of more compact information: i) a list of bi-partitions induced by different trees, ii) the list of sequences included in each tree. These two sets of information are sufficient to create the MRP matrix. Furthermore, for each bi-partition, one can save only the sequences on one side of the partition, and the rest of sequences included in that tree should be on the other side. In addition, since writing to the final file happens row by row, it is more efficient to keep the above information on a per sequence basis. That is, we need a data-structure that, for each sequence, can efficiently return which trees the sequence belongs to, what bi-partitions are induced for each of those trees, and whether the sequence is on the 1 side or the 0 side of each of those bi-partitions. Given such a data structure, outputting the MRP matrix to a file is easy and efficient.

To create the aforementioned data structure, one can create the trees in memory first and then traverse them in some fashion to extract the list of bi-partitions. However, whenever creating the MRP matrix is the only goal of a program, a more efficient approach is to build the list of bi-partitions on the fly as the trees are being read from the input. If the trees are given in the Newick format, the bi-partitions can be induced by stacking the open brackets and sequences following them and then matching the open brackets on top of the stack against the close brackets. The details of how these two ideas can be implemented are given next.

1.2 Algorithmic Design

Let's assume we are given a set of trees $\mathcal{T} = \{T_0 \dots T_n\}$ in the Newick format on a disjoint but potentially overlapping set of taxa. For example, we can have

$\mathcal{T} = \{T_0 = (A, B, (C, D));, T_1 = (C, (A, (E, F)), D);\}$. The goal is to create an MRP matrix. For the given example, a potential MRP matrix is:

A 0 1 0
B 0 ? ?
C 1 0 0
D 1 0 0
E ? 1 1
F ? 1 1

Let's assume we have a function $m(s)$ that maps a sequence s to a sequence index. Creating such a function is trivial. In the above example, we can index the sequences as we encounter them so that our trees can be represented as $\{T_0 = [(0, 1, (2, 3));], T_1 = [(2, (0, (4, 5)), 3);]\}$.

Our algorithm makes use of the following data structures to keep the bi-partitions in the memory:

TreesPerSeq is an indexed list of sets of tree indices. Each element in *TreesPerSeq* is itself a set that corresponds to a particular sequence. Each set contains indices of trees that the sequence belong to. In the given example, $TreesPerSeq[0]$ is $\{0, 1\}$ to indicate that sequence 0 appears in T_0 and T_1, while $TreesPerSeq[4] = \{1\}$ to indicate that 4 appears only in T_1.

ColumnsPerSeq is an indexed list of lists of column indices. Each element in *ColumnsPerSeq* corresponds to a sequence and is itself a list of column indices in the MRP where that sequence should have a '1'. In the given example, $ColumnsPerSeq[0] = \{1\}$ to indicate that sequence 0 has a "1" only in the second column of the MRP matrix. Or, $ColumnsPerSeq[4] = \{1, 2\}$ to indicate sequence 4 has a "1" in the second and third columns.

EndIndexPerTree is a list of column indices. This list contains for each tree, the index of the last column in the MRP matrix that relates to a bi-partition in that tree. In our example, T_0 has only one bipartition, appearing in the column 0 of the MRP matrix, and therefore $EndIndexPerTree[0] = 0$. Tree T_1 has two bi-partitions and the last one appears in column 2. Therefore, $EndIndexPerTree[1] = 2$.

ColumnsPerSeq tells us where to put a "1" for each sequence. The two other lists provide the information necessary to decide between a "0" and a "?". From *EndIndexPerTree* we can find the tree corresponding to each column. If the tree corresponding to a column does not contain a sequence (based on *TreesPerSeq*) it has to be a "?". Otherwise, it is a "0". Our algorithm to create an MRP matrix given these three data-structures is outlined in Figure 1.

To create the data-structures outlined above, we use a stack data-structure:

S is an stack of lists of sequence indices (i.e, each element in the stack is itself a list). Each list, when completed, corresponds to a bipartition, which in turn corresponds to a column in the final MRP matrix.

The above data-structure is used as follows. Each tree T_i given in the Newick format is read from the input. The Newick representation of the tree can be trivially broken up into a list of tokens (using regular expressions for example).

```
for each sequence index s do
  output the name of the sequence in a new row
  Let c = 0
  Let oneColumns be the s^th element of ColumnsPerSeq
  Let nextOne be the first element of oneColumns.
  for each tree index t do
    let tEnd be t^th element of EndIndexPerTree
    if TreesPerSeq[s] contains t then
      while c ≤ tEnd do
        if nextOne is c then
          write a 1 to the output
          let nextOne be the next element of oneColumns.
        else
          write a 0 to the output
        end if
        Increment c
      end while
    else
      write ? to output tEnd − c + 1 times
      Increase c to tEnd + 1
    end if
  end for
end for
```

Figure 1: Outputting MRP

Such a list is obtained and the tokens are traversed in order. A new empty list is pushed to the stack whenever a "(" is encountered (the first open bracket is skipped). When sequence names are encountered, they are added to the list on top of the stack (if it exists). In addition, $TreesPerSeq$ is updated to indicate that the observed sequence is present in the current tree. When a ")" is reached, the elements in the list on top of the stack form a bipartition. The "top" element is popped out of the stack. Then, a new column in the matrix is created by updating the $ColumnsPerSeq$ of each sequence in the "top" list with an incrementing column index. If the stack is not empty, the sequences in the top list also belong to another bi-partition. They are therefore added to the list that is now on top of the stack. This process is repeated until we reach a semi-colon, indicating the end of the tree. At this point, $EndIndexPerTree$ is updated with the incrementing column index to mark where the tree ends. This process is described in the algorithm outlined in Figure 2.

As an example, consider the second tree in our example above, that is $T_1 = [(2, (0, (4, 5)), 3);$. Let's assume from the previous tree, we have kept a current column index (c), which is now at 0 (since the first tree has only one column). Initially S is empty. The first open bracket is skipped. When sequence 2 is read, it is not added to S since the stack is empty initially. However, the set at $TreePerSeq[2]$ is updated to include the index of current tree (i.e. 1). After reading the second "(", we add an empty list (let's call it l_1) to S. Then sequence 0 is read and is added to l_1 and $TreePerSeq[0]$ is updated with a 1. Then, we encounter another "(" and therefore add a second list to stack

```
Let c = −1
for each Newick tree T indexed i do
  for each token tok in T do
    if tok is a '(' then
      Push a new empty list to S
    end if
    if tok is a sequence name and S is not empty then
      Add m(tok) to the list on top of S
      Add i to the list on the m(tok)^th element of TreePerSeq
    end if
    if tok is a ')' and S is not empty then
      Pop top from S
      Increment c
      for each sequence index s is top do
        Add c to the list on the s^the position of ColumnsPerSeq
      end for
      if S is not empty then
        Add all elements of top to the list on top of S
      end if
    end if
    if tok is a ';' then
      Add c to EndIndexPerTree
    end if
  end for
end for
```

Figure 2: Reading Input and Creating the Data-structures

(call it l_2). Then 4 and 5 are read and are added to l_2, in addition to updating $TreePerSeq[4]$ and $TreePerSeq[5]$ with a 1. Then we encounter a ")". The contents of l_2 constitute a bipartition. We, therefore, increment c to 1 and then add $c = 1$ to $ColumnsPerSeq[4]$ and $ColumnsPerSeq[5]$ to indicate that sequences 4 and 5 have a character "1" in column 1 of their MRP. Next, l_2 is removed and all its elements are added to l_1 so that it contains $0, 4, 5$. Then another) is read, and the above process repeats. This time c is incremented to 2 and a 2 is added to $ColumnsPerSeq[0]$, $ColumnsPerSeq[4]$ and $ColumnsPerSeq[5]$. Since after this step the stack becomes empty, reading 3 only updates $TreePerSeq[3]$ and does not affect S or $ColumnsPerSeq$. After the semi-colon is read, $EndIndexPerTree[1]$ (where 1 is the tree index) is updated to current value of c, which is 2.