Antibody Clustering

Algorithm description

pfr

Contents

1. Algorithmic improvements **2**
1.1. Partitioning . . . 2
1.2. Reduction to unmutated antibodies (essences) . . . 2
1.3. Megaclustering . . . 2
1.4. Final clustering . . . 3

2. Implementation improvements **4**
2.1. JSON parsing . . . 4
2.2. SIMD implementation of Levenshtein distance . . . 4
2.3. SIMD implementation of mutation lists . . . 4
2.4. Memory management . . . 5

A. JSON format **6**

Introduction

The contest scoring function has a loss proportional to the number of mistaken links in the clustering graph. This has several consequences: we can freely ignore very small clusters since they don't really affect the score, but on the other hand any mistake in a large cluster is severely penalized. Because cluster construction is a somewhat chaotic process, even the slightest approximation has a chance of resulting in a very different clustering. Such a clustering is not necessarily worse for practical purposes or even by any objective measure, but because it is very different from the reference output it will be greatly penalized in the current competition format.

As a result, our hand is forced: we must apply the clustering algorithm exactly. Our only hope for speedup is to apply it on a subgraph where edges that are unlikely to appear in the cluster graph have been removed (and conversely, edges that are certain to appear in the cluster graph have been contracted). This is what we do through partitioning and megaclustering (conversely, reduction to unmutated antibodies).

1. Algorithmic improvements

1.1. Partitioning

Property: Antibodies with different V gene families are never clustered together, even though the dissimilarity metric does not depend on that variable.

Therefore we can split the dataset into 7 partitions, one for each V gene family, and process each partition separately.

Note: While this nominally reduces the number of edges by more than $3\times$, this step is not critical to performance since megaclustering would have removed those edges anyway. The net performance gain is around 15%, whereas the other algorithmic improvements improve performance by orders of magnitude.

1.2. Reduction to unmutated antibodies (essences)

Property: Antibodies that only differ in mutation list are always clustered together (except for a single exception in the dataset).

This property is a consequence of the fact that in the absence of rounding errors, such antibodies have the smallest non-zero dissimilarity possible.

Such a subcluster is represented by the `Essence` class. Most of the rest of the clustering algorithm will manipulate whole *essences* rather than individual sequences. The only exception is the computation of the final distance matrix, which needs to precisely account for each mutation list individually.

1.3. Megaclustering

Property: All large clusters have at least one essence containing a large number of sequences. We call them *centers*. They generally represent a large portion of the cluster, so that whether an essence belongs to a cluster is mostly decided by distance to one of its centers.

Since centers are defined only by the number of sequences they contain, they are very easy to identify. We tentatively assign each essence to the cluster of the nearest center. This gives a coarse clustering represented by the `Megacluster` class. Because a cluster may have multiple centers, we also have to group centers that are close to each other to make them part of the same megacluster.

Note that in the process, we may group distinct clusters into the same megacluster. This is safe, because the final clustering will remove any extraneous links introduced by aggressive grouping. On the contrary, failure to group centers that are part of the same cluster cannot be recovered by final clustering, and would have a dramatic effect on efficiency. This is why we choose a center grouping

radius higher than the cluster cutoff threshold. The goal of megaclustering is to split the dataset by removing links that we know almost certainly will not be part of the ground truth linking matrix.

When computing dissimilarities, we need to pick a specific mutation list from the essence subcluster: we pick the "most representative" list of mutations (specifically, the median). Since this is only a single mutation list, dissimilarity computation is much faster than the precise computation of the average number of shared mutations that we will do in the final clustering step.

The nearest-neighbor search is done by first examining centers with the same length as the sequence, then centers with length differing by 1, and so on until the length difference imposes a lower bound on dissimilarity higher than the best dissimilarity found so far. This dramatically reduces the number of Levenshtein distances that need to be computed.

1.4. Final clustering

Each megacluster is clustered separately, based on an exhaustive distance matrix computation. Once the matrix is computed, we simply feed it to the reference sample implementation, which implements nearest-neighbor-chain clustering.

The essence dissimilarity matrix is computed in two steps: first, the Levenshtein distance matrix is computed. Then, the rest of the dissimilarity is computed. The most time-consuming part in that step is computing the average bonus, which is derived from the shared mutations. This would be easy if not for the fact that bonuses are clipped so that the dissimilarity is at least 0.001. This introduces a non-linearity that forces us to examine all pairs of mutation lists in both essences, the number of which grows quadratically with essence size. Introducing approximations such as randomly sampling from a few hundred or even thousand pairs is not good enough and will result in major errors with probability almost 50% in the provided dataset. Thankfully, representing each collection of mutation lists as a bag (weighted list) keeps the exact computation manageable, and large essence sizes no longer dominate the runtime.

2. Implementation improvements

2.1. JSON parsing

Our clustering algorithm is fast enough for parsing to become a significant bottleneck that must be optimized for.

Since the JSON data was generated by a Python 2 script, it follows a very predictable layout. We can completely ignore the content of certain lines, and for the lines containing useful data we know the exact column at which the data starts. The exact form in which the parser expects data is given in the appendix.

2.2. SIMD implementation of Levenshtein distance

It's straightforward to extend the dynamic programming Levenshtein distance to simultaneously compare one string s to 16 other strings $t_1, \ldots, t_{16}$, by simply replacing scalar operations with parallel operations. The number of operations is proportional to $|s| \max_i |t_i|$. To minimize the waste caused by the max operator, we process strings in order of increasing length, so that most strings in a given batch have length identical or close to the maximum length of the batch.

One drawback is that the calling code has to be reworked to issue calls in batches: this makes the SIMD implementation of `Partition::megacluster` a little bit more complex.

2.3. SIMD implementation of mutation lists

For speed, we want to encode mutation lists as bitmaps: each mutation is assigned to a fixed location so that they can be processed in a massively parallel fashion. However there are almost 4000 distinct mutations in the dataset, which would require almost 500 bytes. Since more data means more processing time, we want a more efficient representation.

To save space, we exploit the fact that only a single mutation can appear at a given location. Only 316 distinct locations appear in the dataset, so mutation lists are represented as words of length 384 on the alphabet $\{\bot, 0, 1, 2, \ldots, 15\}$, where $\bot$ is the absence of mutation and the other symbols represent the various kinds of mutations.

In order to maximize space efficiency, we store the 16 mutation symbols by encoding them in binary into 4 separate bitplanes $x_1, \ldots, x_4$, and we add a fifth plane x_5 to encode the $\bot$ symbol. Each mutation list uses $5 \times 384/8 = 240$ bytes.

x and y describe the same mutation if and only if the formula $x_5 \wedge y_5 \wedge \neg \bigvee_{i=1}^{4} (x_i \oplus y_i)$ holds. This is efficiently computed by SSE2 operations processing 128 mutation locations at a time. A series of 64-bit SSE4 popcount instructions wrap up the computation to return the number of shared mutations.

2.4. Memory management

In order to minimize time wasted on memory allocation, most temporary data is stored as part of large pre-allocated vectors. This was done partly because I mistakenly believed that static program initialization time wasn't included in the time measurement. However it's still a win in other cases: for example mutation lists have unknown size during parsing, so instead of growing a vector for each list (which would cause either reallocations or wasted memory), we append each mutation to a single global buffer, and represent a mutation list as a slice of that buffer (that is, a pointer and a length). This completely avoids allocations or memory copies.

While this approach is a performance win and thus makes sense in a contest, in many cases it introduces coupling between components that I wouldn't recommend for a production implementation.

Appendix A.

JSON format

The expected field order and indentation is:

```
[
  {
    "v_gene": {
      "all": ...,
      "gene": ...,
      "full": ...,
      "fam": ...
    },
    "seq_id": ...,
    "j_gene": {
      "all": ...,
      "full": ...,
      "gene": ...
    },
    "junc_aa": ...,
    "var_muts_nt": {
      "muts": [
        {
          "loc": ...,
          "mut": ...
        },
        ...
      ],
      "num": ...
    }
  },
  ...
]
```

When representing an empty mutation list, the opening and closing brackets must be collapsed down to a single line.