Greylag Theory of Operation

> Simplicity is prerequisite for reliability --Edsger W. Dijkstra

Design Goals

- Use a high-level language where possible, with performance-critical portions implemented in a suitable low-level language.
- Create an implementation that is a tool for learning. The code should be easy to follow and easy to play around with and extend.
- Try to incorporate the best features from other existing search programs.
- Design for single-host and cluster parallelism.
- Aim for reasonable performance: Run time within a factor of two versus programs implemented entirely in C or C++, and memory profile good enough for current hardware.

Inside-Out Search

Greylag uses a novel (we believe) "inside-out" search strategy. Database search is essentially a depth-first search, but there are different ways that this search can be structured.

A straightforward search algorithm might work something like this

1. Choose a locus (database sequence) to digest.
2. Choose the N-terminal endpoint of a possible candidate peptide within that locus.
3. Choose the C-terminal endpoint, determining the candidate peptide.
4. Choose, one by one, the potential modifications for each candidate peptide residue position, including PCA modifications (if implemented), forming a set of modifications. Once this is done, the mass of the modified candidate peptide is fixed.
5. Choose, one by one, the candidate spectra (if any) that have a precursor mass close enough to the mass of the modified candidate peptide.
6. Score the chosen candidate spectrum against the modified candidate peptide, keeping track of the highest-scoring peptides for each spectrum.

We have not exhaustively investigated other programs, but we suspect that they all use algorithms similar to this.

Greylag's search algorithm looks like this

1. Choose the number of modifications, starting with zero and counting up to the limit (parameter `potential_mod_limit`).
2. Choose the mass regime pair (*e.g.*, MONO/MONO).
3. Choose a possible PCA modification, or no PCA modification.
4. Choose the modification conjunct. For example, the choice might be { `PO3H@STY phosphorylation`, `C2H2O@KST acetylation` }, which specifies that there will be at least one phosphorylation and at least one acetylation. The N- and C-terminal modifications, if any, are also similarly chosen.
5. Choose the "delta bag". This is a choice of the number of each kind of residue modification. For example, if we chose three modifications in step one, and two conjuncts in step 4's example, the possible delta bags would be `(1, 2)` (one phosphorylation and two acetylations), or `(2, 1)` (two phosphorylations and one acetylation).
6. Choose a locus.
7. Choose the N-terminal endpoint of a possible candidate peptide within that locus.
8. Choose the C-terminal endpoint, determining the candidate peptide. Once this is done, the mass of the modified candidate peptide is fixed, as we know the peptide residues and the number of each kind of modification.
9. Choose, one by one, the candidate spectra (if any) that have a precursor mass close enough to the mass of the modified candidate peptide.
10. Choose, one by one, the potential modifications for each candidate peptide residue position, subject to the choices made in previous steps.
11. Score the candidate spectra against the modified candidate peptide, keeping track of the highest-scoring peptides for each spectrum.

There are a number of advantages to this inside-out approach. Most importantly, it allows us to hoist a significant amount of the search out of the performance-critical loop, thus allowing quite a bit of the search to be implemented in a high-level language without loss of performance. This is a great benefit because high-level code is much easier to write, understand, make correct, debug, and modify.

For a typical search, in the traditional algorithm described above, step 2 and all subsequent steps will be executed millions of times, thus requiring a low-level implementation.

In the greylag approach, steps 1 through 5 will be executed a relatively small number of times, perhaps as much as few thousand times for complex searches, but far fewer for basic searches. This means that they are not particularly time-critical, and can be implemented in high-level code. Steps 6 through 11 must still be implemented in a low-level language, but the code required is smaller and simpler because part of the work has already been done in the high-level portion. In fact, the vast proportion of run time is spent in step 11 for most searches.

There are several other minor advantages to the inside-out approach. Because a number of common calculations are being lifted out of the inner loop, it's probably a little more efficient than the straightforward approach.

It also makes possible an incremental approach to modification search. Since zero modifications are searched first, then one, then two, *etc.*, one could potentially search for a fixed amount of time and then stop, knowing that the most likely combinations will have been searched first.

There is one potential disadvantage to the inside-out approach, which is that more total work will be done by the modification assignment performed in step 10, versus what happens in step 4 of the

straightforward approach. This seems not to be a problem in practice, probably because the cost of the final scoring in step 11 tends to dwarf the costs of the previous steps.

Validation Algorithm

The core algorithm is based on a step-up procedure at a specified FDR (False Discovery Rate), in the manner of Benjamini and Hochberg. In particular, the result scores are ordered, and the lowest threshold having the specified FDR (or better) is chosen. The number of false, non-decoy discoveries in the chosen set is assumed to be equal to the number of decoy discoveries, given that the decoy database is the same size as the original database.

Rather than just using scores, the greylag algorithm attempts to optimize the number of ids by adjusting both the score and delta. Roughly, the delta threshold is set to a range of different values, and at each value the score delta and number of ids is evaluated. The score and delta threshold pair having the most ids is chosen.

Outside of this core, `greylag-validate` allows some filtration of the input ids, according to several criteria. First, ids can be excluded based on trypticity--having an insufficient number of tryptic terminii. Ids can also be excluded if they do not satisfy a "minimum peptides per locus" criterion. Finally, a "maximum Sp rank criterion" can be applied (this supports SEQUEST output, but is not useful for greylag output).

After these filters are applied, some peptide ids are added back even if they do not satisfy the filter criteria or score/delta thresholds, in a method we call *saturation*. The basic idea is that if we have called an id for peptide PEPTIDE valid, we declare all other ids for that peptide (with possible modifications) valid as well, regardless of score/delta/etc. The rationale for this is that once we have identified a peptide as being present in the sample, it is much more likely that further identifications of that peptide are also valid, rather than just occurring by chance. Although this may introduce some false id's, the presumption is that there will still be a larger number of valid id's at the given FDR. If this is not the case, the saturation step is omitted. (The option `--no-saturation` will cause saturation to be skipped entirely.)

In addition to the saturation described above, we also perform *prefix saturation*, which is similar except that we add all identifications for peptides that have a *prefix* that is a peptide previously identified as valid. (We could also perform *suffix saturation*, but in testing this seems not to be helpful.)

Because these filters and saturation are applied *after* the score/delta thresholds are chosen, the resulting FDR may differ from that chosen by the user (usually the resulting FDR will be better). In order to minimize this difference, the core algorithm is run iteratively with differing goal FDRs, in order to try to generate a resulting FDR that is close to what the user has specified. (The option `--no-adjust-fdr` will omit this step.)

Specific Implementation Technologies and Tradeoffs

The implementation of greylag involved many technological choices and tradeoffs. Here is a discussion of several of the major choices and their rationales.

Mixed vs Low-Level Implementation

Peptide searching is very CPU-intensive, and somewhat memory-intensive as well, so the typical choice is a low-level language like C++ (or C) that can generate very efficient programs. An alternative approach is to write the non-critical portions of the program in a high-level language and to use a low-level language only for the performance critical sections. We chose the latter for the following reasons:

- Peptide search is still a research problem, requiring a lot of algorithm experimentation. High-level, interpreted languages are much more suited to this kind of prototyping situation.
- Low-level languages require a great deal of programmer expertise in order to write correct programs. A C++ programmer, for example, really needs to understand the content and implications of documents like the C++ FAQ and C++ FQA. One of our goals is that non-expert programmers (*e.g.*, biologists) be able to understand and modify greylag.
- Writing a program in a low-level language is very time-consuming, and our resources are limited.

Python as a High-Level Language

Python has the following salient advantages as the high-level language for greylag:

- It fails safely. If the program tries to write to a full disk or performs an illegal math operation, this will generate an error message, even if the programmer has forgotten to explicitly check for such errors. (Perl and C++, for example, do not have this property.)
- It's in broad use and its user community sees to be growing over time.
- It's cross-platform, so the same Python programs can potentially run across Linux, other Unixes, MacOS, and Windows.

Currently, the only other alternative with these properties is Ruby, which we might have chosen instead.

C++ as a Low-level Language

The only serious candidates for the low-level language are C++ and C (the latter being arguably now just a simpler subset of the former).

We chose C++ with the hope that using STL data structures and algorithms would help us to reduce the amount of code to be written, which seems to have worked.

However, using the STL also introduces a lot of mental overhead and opportunities for subtle bugs--one needs to understand the implications of *singular iterators*, for example.

Another problem with STL data structures--and C++ objects--is that they're noticeably slower than C-style data structures, and we found it necessary to drop vectors in favor of C arrays for several structures that are operated on in the most CPU-intensive core of the program.

In hindsight, for greylag C++ seems to be at best a wash versus C. We might switch to C in the future.

(Java was ruled out because it's somewhat slower, which is particularly important because we're only using the low-level language for performance-critical sections to begin with. Also, Java's VM makes it difficult to integrate under other languages, and there are lingering licensing issues.)

Methods for extending Python with C++

There are a number of ways to make C++ (or C) code callable from Python:

- The native C external function interface. This makes your code a regular Python module, and is nice because there are no external dependencies at all. A big minus is that Python reference counts have to be managed manually.
- Access to shared libraries via the Python ctypes module. All of the interface wrangling is in Python, which is nice.

- Pyrex. This is a translator from a C-like Python variant to C. It takes care of reference counts. The minuses are that it's one more language to know and an external dependency, and it's not yet clear whether Pyrex will have a long future.
- SWIG. Its big advantages are that it allows STL data structures like vectors to be accessed more-or-less normally as Python lists, etc. Likewise, C++ objects can be accessed as objects from Python as well. The downsides are that development of SWIG has slowed dramatically, it's very complex internally, and it generates huge hunks of fairly undebuggable wrapper code.

Greylag currently uses SWIG. Despite its problems, it keeps the code small and allows for functionality to be easily moved back and forth between Python and C++.

There are some other alternatives, too, like Boost.Python, SIP, Weave, etc. These don't appear to be viable for greylag.

Greylag versus other Search Programs

Generation of Peptide Modification Combinations and Limiting Search Depth

Greylag generates peptide modification combinations using depth first search (DFS), which means that memory usage is minimal (logarithmic). The only real limit is run time, which is necessarily exponential with respect to search depth. Greylag limits search depth by explicitly limiting the total number of modifications (modified residues) per peptide.

Greylag will not generate modified combinations for which there is no spectrum having a suitable precursor mass. So, for example, in considering a peptide 'AAAAAAAAAXXXXXXXDDDDDDDDD' with (say) mass 1200.00 and possible modifications A+10 and D+9, the modified mass would be 1286.00 for five +10 and four +9 modifications. If there are no actual spectra that have precursor mass close to 1286.00, greylag will not attempt to generate the specific modification combinations (*e.g.*, 'AA*AA*A*A*AAA*XXXXXXXD%DD%D%DDD thus saving some running time. Of the alternative programs, only SEQUEST appears to implement a similar optimization.

SEQUEST appears also to use DFS and limits search depth in a similar way, except that the depth of each kind (*e.g.*, oxidation) of modification is limited separately. The version of SEQUEST we have ("v.27 (rev.9)") can simultaneously search at most three kinds of modification.

X!Tandem also uses DFS, using a recursion-free "state engine" approach that also uses minimal memory. Rather than explicitly limiting search depth, it places a limit (of 4096) on the number of combinations searched for each peptide. In the simple case, this means that X!Tandem will consider at most 12 modifications per peptide. (It's not clear to us how terminal modifications and other advanced modifications such as point mutations figure in this.)

This limit is not a parameter of the program and appears to be undocumented, so users may not realize that larger modification sets are silently ignored. Also, since the limit cannot be adjusted (short of recompiling the program), X!Tandem cannot do more shallow--and therefore much faster--searches such as searching for peptides with at most two methionine oxidations. (Balancing this somewhat, X!Tandem does provide a unique "refinement" algorithm, but that is orthogonal to the considerations of this section.)

OMSSA appears to use breadth-first search on a per-peptide basis to generate modification combinations, and uses a combination limit similar to that of X!Tandem. This means that it will use an amount of memory linear in the number of combinations, which will be exponential in search depth. This would limit search to a depth of 25 or so on current hardware, but it's not clear whether this matters in practice.

MyriMatch uses DFS and a depth limit like that of greylag.

As a final detail, the implementations of X!Tandem, OMSSA, and MyriMatch appear to also limit the maximum number of modifications per peptide to the machine word size: 32 on a 32-bit machine and 64 on a 64-bit machine. (The corresponding limit for greylag is at least one billion.)

Shotgun Proteomics Principles