Optimizing PhosphoScore: An Open-Source Phosphorylation Site Assignment Tool

A project for CS167 at UCSB by Evan Senter

Abstract:

Traditional mass spectrometry techniques are capable of detecting the presence of phosphate groups, but have no means for determining which of the amino acids are phosphorylated. PhosphoScore is a program used to determine the most likely phosphorylation site for protein sequences generated from LC-MS/MS data. The objective of this project is to enhance the speed of PhosphoScore through both general code speedups and predictive tree pruning where possible, while not reducing the fidelity of the analysis. Throughout the remainder of this paper I will discuss what went well, what didn't work, and what the end result of this project has been.

Motivation / Background / Significance:

There is a central dogma in biology, that describes the flow of information in a cell - from DNA to RNA to proteins. From this dogma one can describe how proteins, the main actors on the cellular stage, get produced from their blueprints within DNA. Even after a protein's translation from mRNA and folding into a three dimensional structure, there are a number of post-translational modifications that it can undergo. These modifications (PTMs) give proteins a sense of state - for example sodium channels in cells are made up of integral membrane proteins that open/close in response to voltage, or the binding of a ligand (an effector). There are many such examples of how PTM of proteins plays an important role in dictating sub-cellular interaction - protein degradation, localization, signaling, pathways and many other core actions are driven by the state of a protein. While we won't explore them all, suffice it to say that there is more information about the function of a protein than what is coded in its primary sequence.

One such class of PTMs is phosphorylation, the addition of phosphate groups to the residues on amino acids within a protein. There are only three residues that can be phosphorylated - they are (listed in order of frequency) serine (S), threionine (T), and tyrosine (Y). Protein phosphorylation is a quickly growing field of study within proteomics, and is (rather expectedly) dubbed phosphoproteomics.

The purpose of PhosphoScore is to serve as a predictor of the most likely phosphorylation site for a protein, given its primary sequence and a mass spectra.

Mass spectrometry is a technique by which one can determine the primary sequence of a protein, by leveraging electrodynamics in an approach reminiscent of shotgun sequencing. Because two particles with the same mass to charge ratio follow the same trajectory when subjected to electromagnetic fields, one can fragment a protein into small oligopeptides and send them through a magnetic field, paying note of their displaced direction. From this a mass spectra can be constructed plotting mass to charge on one axis and amplitude of signal on the other. From this spectra it is possible to predict the sequence of the protein having just been run through the mass spectrometer. There is a catch however. Spectra have a hard time recognizing the location of PTMs on a protein, because when a primary sequence gets fragmented, it is also possible to rip off the PTM, in our case - phosphate. Mass spectra can, on the other hand, detect the total mass of a product, and if the sequence is known it is possible to quantify the number of phosphate groups on that protein.

So, the question becomes one of predicting which amino acids in the protein had phosphates attached to them when sequenced. Our task is made simpler by the fact that only three amino acids can have phosphate groups attached, and we know how many we are looking for (from the total mass of the product). This is the question PhosphoScore answers, providing a valuable tool to researchers in the field of proteomics interested in knowing the state of a protein in question.

Methods:

The method by which PhosphoScore works, in short, is to construct a tree of all possible solutions, and traverse the tree in such a way as to find the cheapest path. The nodes in the tree, from level to level, represent an amino acid that may have been phosphorylated. There are three possibilities, that it did or did not have the phosphate group, and neutral loss, which occurs when the protein is fragmented at the covalent bond attaching the phosphate to the amino acid and releasing water.

The tree grows at a rate of 3^n, which very quickly becomes a huge search space. In its original form, PhosphoScore performed a breadth-first traversal of the tree, using a cost function to compute a score for each node. The most likely phosphorylation sites can be found by selecting the lowest scoring leaf node, meaning that the spectra generated by having those amino acids phosphorylated minimizes the differences with the actual spectra. From the leaf node it is easy to backtrace up the tree to the root, and by determining what branches were taken find the phosphorylation sites for the entire sequence.

Performing a breadth-first search however is not ideal for an application such as this one though, because it forces a traversal of all non-leaf levels of the tree before any final scores are computed. This is where my two variations come in.

My initial proposal was to switch the breadth-first traversal to depth-first, because then it is possible to leverage some pruning techniques to help mitigate the searching. By traversing depth-first you have the advantage of knowing final scores in $\log_3$ time, and you can keep track of a global minimal path score. Because we are trying to minimize the final score (and the lowest score possible for a node is 0, meaning it is an exact match) anytime in the traversal that a node has a higher cumulative score (meaning the summed score from the root to that node) than the global minimum path score, we can prune the subtree with that node as its root. So, the pseudocode would look something like the following:

```
1 for each leaf node L that has not been pruned:
2   construct a path P from the root to L
3   for each node N in P:
4     if N has already been computed:
5       skip to the next node in P
6     else:
7       compute N's score and update N
8       prune N's subtree if N's score > best path's score
9       update best path score if N is a leaf and N's score is better
```

This is fairly similar to how I implemented the depth-first traversal - allow me to explain my reasoning. Depth-first traversal is an inherently recursive traversal, but the is encoded as an array. My main reason for not writing it recursively is that the pre-existing Build function (the method that constructs and computes the score for the tree) calls out to a number of helper methods whose interface is quite specific, and not each node is handled the same way. So I decided to do it iteratively, to give me more flexibility as to what is going on in the traversal. There is added overhead in doing it iteratively: In iterating over the leaf nodes (line 1) you must construct the path from the root to that node for each node that is a leaf. Secondly, you repeat many nodes this way, making it so you don't touch each node only once. But because the desire is to get as many full paths as quickly as possible (to allow the most pruning) I decided this was a worthwhile expense, and added the skipping if the node had already been computed. And finally, pruning became difficult to do.

The problem with pruning is this: you are iterating over the leaf nodes, but given an arbitrary node in the tree you need to flag all the leaves in the subtree with that node as root as 'pruned'. To do this I used a queue-like structure to add child nodes on a level by level basis until I got to the leaves, and then marked those all as pruned (in a hash data structure). By pruning you have the opportunity to save large amounts of time by skipping entire subtrees (and therefore many leaves, see line 1). Discussion of the results will be below, but first let me segway into the other optimization technique I tried.

Maintaining a breadth-first traversal, one can easily see a simple, but not very fruitful, optimization in the same spirit as above. It is quite simply to do the same global tracking of the best score, while still traversing level by level. I'm hesitant to call this pruning sense you only skip leaf nodes, but it turned out to be a very reliable technique that consistently gave similar results. In practice this can save you $(3^{\text{depth of tree - 1}}) / 2$ computations, which is only very useful at very large sizes.

Experimental Design and Results:

Datasets:

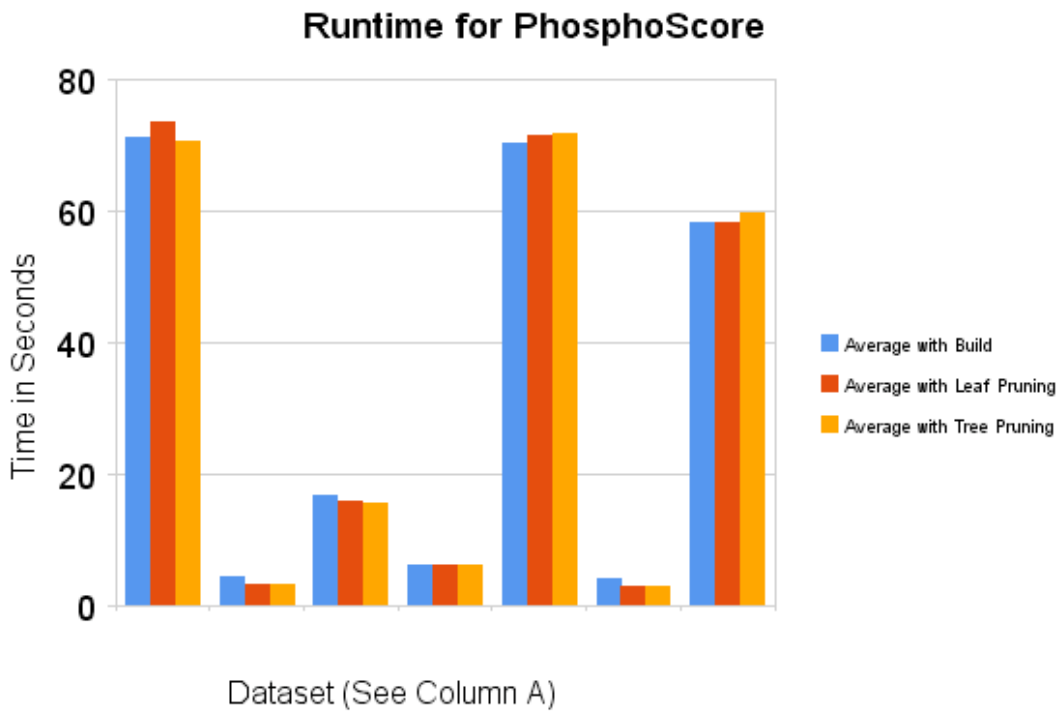
The data I used came from Brian Ruttenberg, the original author of PhosphoScore. I opted to use smaller sequence sizes in the interest of turn around time, so keep in mind when looking at the results that this is for small sequences, where pruning opportunities are not as plentiful or fruitful as large trees. The other reason I used these datasets is because I had a hard time mocking out data. One thing I experimented with was constructing my own sequences, but after spending some time on it and not getting it to work right with the Phosphoscore's parser, I decided to move on.

raw_data/ms2/051018_Jason_dDAVP_FT/data (39 files)
raw_data/ms2/051125_Jason_dDAVP_FT/data (69 files)
raw_data/ms2/GygiSyntheticData/data (763 files)
raw_data/ms2/PhosphoPIC/JH01_060512/data (109 files)
raw_data/ms3/051018_Jason_dDAVP_FT/data (38 files)
raw_data/ms3/051125_Jason_dDAVP_FT/data (62 files)
raw_data/ms4/data (19 files)

Experimental Setup:

There wasn't much experimental setup necessary, I used PhosphoScore's GUI - which now displays the runtime for a task once it is complete and ran my trials 3 times, and used the average. I used the original program as a control, and for my first test determined how much time was spent in the Build method, which came out to 36.2%. This number probably grows much larger as the primary sequence size increases, and another consideration is when running on the Gibbs setting to find the optimal parameters, setup and teardown times for each run compound over all the iterations to yield a higher cost overall. The two optimization techniques described above, depth-first with tree pruning and breadth-first with leaf pruning were run in the same fashion as the control.

Graphs/Tables:



This graph is adapted from a dataset available at the project webpage: (http://www.uweb.ucsb.edu/~evan_senter/), I didn't insert all the data in the interest of space.

Analysis:

Well, the first thing that's apparent from the graph is it sure doesn't look like things are going much faster. In fact, on runs using Gibbs (the 1st, 5th and 7th columns) the added cost of lookups to leverage pruning in the Build method outweigh the speedup that they yield, and compound in a similar fashion as the setup/teardown described above. But what I do think is optimistic is the Score runs, where each sequence only gets run one time. To see improvement on single runs for that small of a sequence tells me that the technique does in fact yield improvement. Overall for Gibbs, leaf pruning cost an additional 1.5% runtime and 1% for tree pruning. For Score runs, leaf pruning yielded a 14.1% speed increase, and 15% for tree pruning.

Discussion / Conclusion:

I did most of my discussion above, but there are a few more things I'd like to note. I am not confident in the fidelity of the tree pruning technique. I've seen both situations where it predicts the same sequence as the original program, and situations where it's predictions are not in line with the original code, which I have been using as a base line. Therefore I obviously conclude that this technique needs further revision before yielding accurate results. I spent much more time debugging it than I did writing it, and the issue simply stems from the fact that the code was originally written for a breadth-first traversal. To convert it to depth-first required me, in many places, to do conversions back to a breadth-oriented indexing in order to get helper methods to work properly, and restricted me from using a recursive (or stack based) approach.

I must say though, that I was quite pleased with the turnout of breadth-first leaf pruning. It was a very simple optimization I added in as an afterthought to tree pruning, and turned out to be completely consistent with the original data, with the following exception (Which applies to tree pruning as well). I quote from the PhosphoScore paper, "The Z-score is an expression of how many standard deviations the lowest cost path is from the mean of all possible paths. In PhosphoScore, a lower Z-score (i.e. more negative) is better since the lowest cost path is below the mean". This value necessarily changes when doing pruning, because you no longer are computing all possible paths. I wasn't sure how to properly handle this issue, and felt it was important to bring to attention in my paper.

All in all I'm glad I chose this topic for a project - it was small enough that it permitted me to work on it without a partner, and allowed me to learn about a field of cellular biology that I only had a passing knowledge of before. I am happy with the results of the project (it has come

leaps and bounds in the last two weeks) and although frustrating at times, was a rewarding experience in the end.

References:

PhosphoScore paper - PhosphoScore: An Open-Source Phosphorylation Site Assignment Tool for MSn Data (http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=2562657)