Searching Biological Sequence Databases

- 10.19.2006 -

Relevant Literature

- ***Using multiple alignments to improve seeded local alignment algorithms***
  Jason Flannick and Serafim Batzoglou

- ***Designing Multiple Simultaneous Seeds for DNA Similarity Search***
  Yanni Sun and Jeremy Buhler

Lecture Outline

I. Background

a. Sequence Alignment
b. Multiple Alignment Database
c. Phylogenetic Tree
d. Probabilistic Profile

II. Structuring the Problem

III. Typhon Components

a. Probabilistic Profile Construction
b. Region Decomposition
c. Seed Indexing

IV. Typhon Performance Evaluation

a. Sensitivity
b. Speed
c. Space

I. Background

Sequence Alignment

Sequence alignment is used to identify regions of similarity in genes and proteins. There are two types of sequence alignment: global and local. Global alignments try to produce the best match across the entire sequence length, while local alignments try to produce the best match within a limited region of the sequence. Sequence alignment is important because it allows us to compare genomes of different species, construct phylogenetic trees, and annotate genomes that have been recently sequenced.

Today, we will be exploring seeded local alignment algorithms. Let's first look at how a seed is defined. A ***seed*** is an ordered set of positions ($x_l < \ldots < x_w$) that you index in a MSA database. For a seed, its ***weight w*** = number of positions in seed and its ***span s*** = $x_w - x_l + 1$.

For example, if we had a seed P = {0, 1, 4, 5}, then w = 4 and s = 5 - 0 + 1 = 6. Within a sequence, indexing seed P at position X translates to recording the presence of the word (that seed P references to) at position X. As mentioned in the last lecture, the positions do not have to be contiguous. A budget for any seed indexing scheme simply refers to the average number of seeds that you index per position.

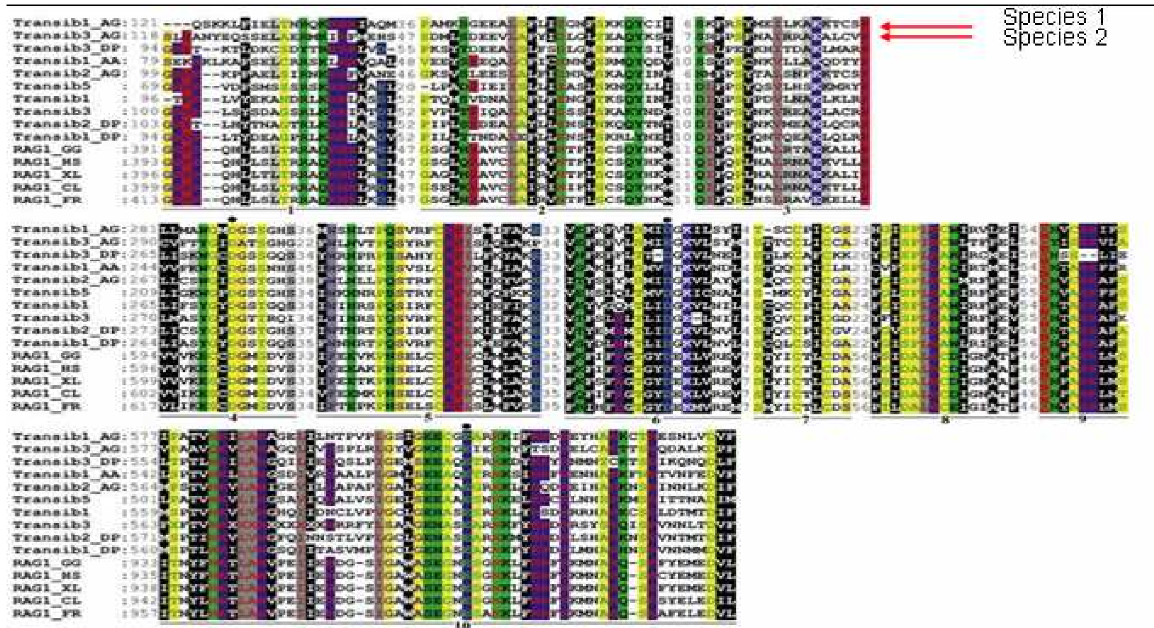
In the example below, Seed A{ 0,1,2,3 } is indexed at positions 0 and 1 in Sequence S.

There are many variants of seeded local alignment algorithms which try to come up with the optimal set of multiple seeds, given what you know about the sequence (ie: similarity level and length). Examples include BLAST, BLAT, BLASTZ, and Exonerate. The reoccurring theme is the use of indexing.

Multiple Alignment Database

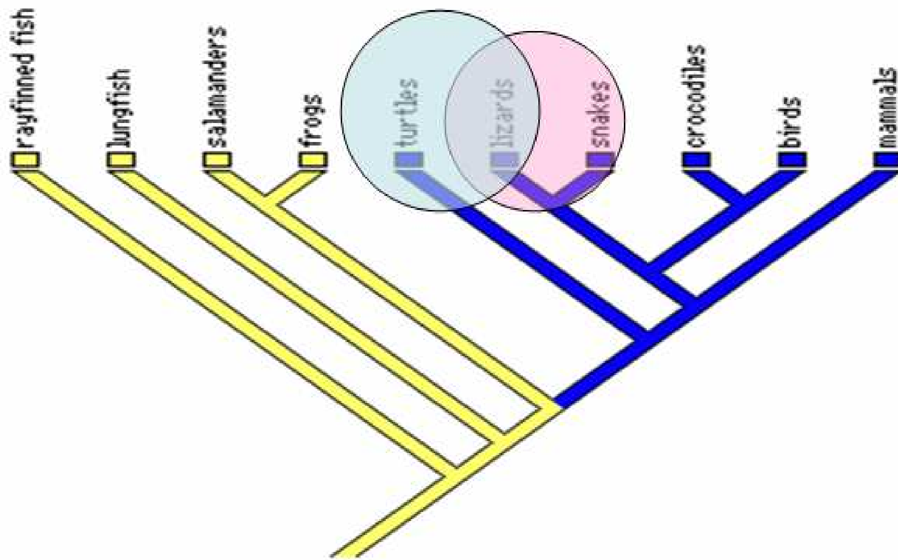
In a multiple sequence alignments (MSA), each row refers to a sequence. The underlying information of MSA's is instrumental to a number of tasks in computational biology. Using a MSA database provides higher search sensitivity than sequence

database alone. As sequencing technologies advance, the size of multiple alignments will increases with the addition of genomic data.



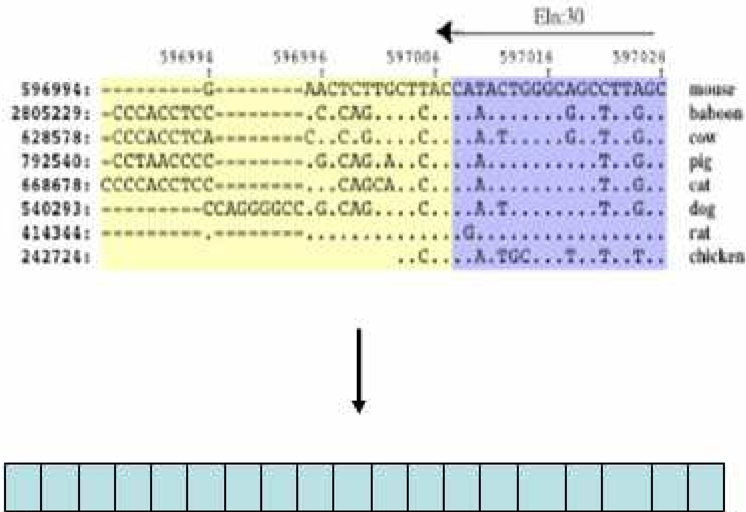
Phylogenetic Tree

A phylogenetic tree is an evolutionary tree that captures relationships between species which share a common ancestor. In the tree below, we can see that lizards and snakes are more closely related in evolution than turtles and lizards. The branching patterns in the tree reflect mutations/speciation events that occurred at some point in time. Every pair of species in the tree shares a common ancestor at some level.



Probabilistic Profile

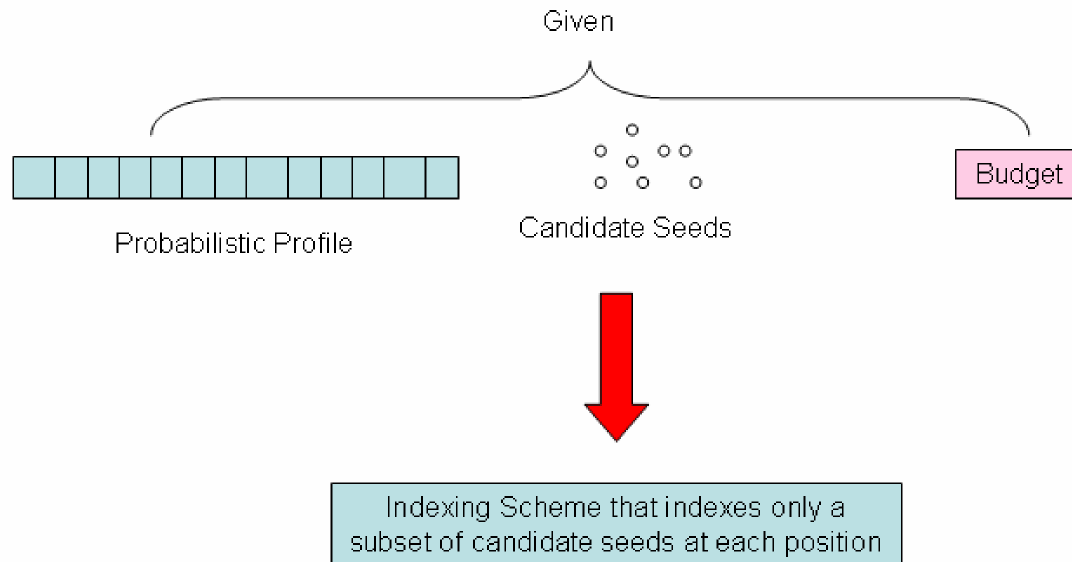
A probabilistic profile joins the information present in the phylogenetic tree and the MSA to produce a single representation of the MSA. In the profile below, dots represent consensus to the first (reference) sequence and dashes represent gaps. Each cell corresponds to a single column in the alignment and expresses the consensus letter of the MSA.



Within a probabilistic profile, region boundaries are determined by variations in conservation levels within the alignment. The computation of region boundaries will be discussed in detail later.

II. Structuring the Problem

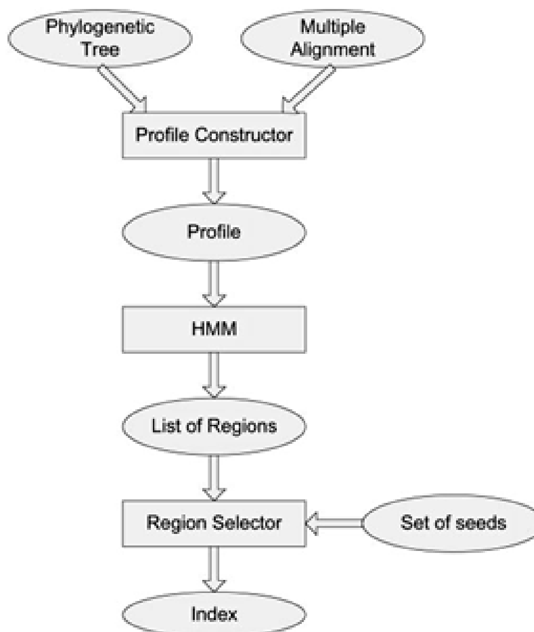
Now, we will structure the problem that Typhon (local alignment tool using a novel indexing algorithm) tries to solve. To start, we are faced with a MSA, query sequence, and a bunch of candidate seeds. The aim is to index seeds at each position such that the probability of homology detection between query and database is maximized. It would be ideal to index every position in the alignment with every seed, but the generation of such a large index is not practical.



The challenge of the problem is that we are constrained by a budget, which bounds the average number of seeds that you index per position. In previous methods of indexing, there lacks an intelligent assignment of candidate seeds to positions, thus causing the presence of a budget to decrease performance. How do we determine which subset of seeds to index at each position?

A useful source of information is our MSA database and phylogenetic tree. Using these, we can leverage the implicit information to come up with an intelligent method of seed assignment.

III. Typhon Components

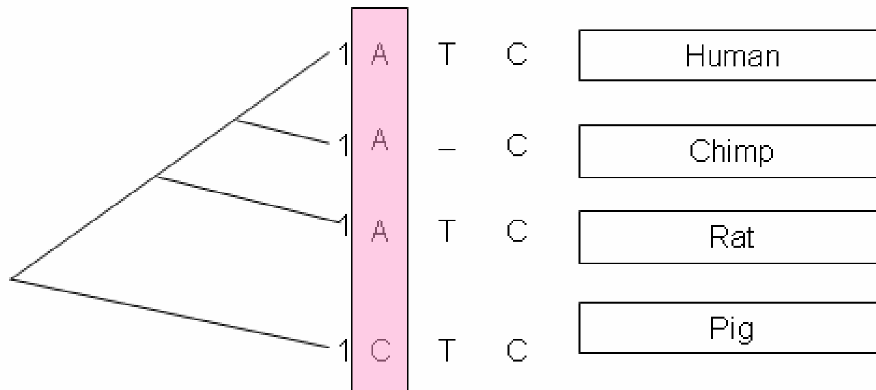


There are 3 main architectural components of Typhon: 1) Generating the probabilistic profile, 2) Identifying regions of the profile, and 3) Deciding which subsets of seeds to index for each region.

Probabilistic Profile Construction

Each position in the profile is composed of a tuple of 6 probabilties. $\boldsymbol{P_{present}}$ refers to the existence probability, or probability that a homolog to the position exists in the query sequence. This can be thought of as the likelihood that the position aligns to the query without a gap. There are 4 P_N values ($\boldsymbol{P_A}$, $\boldsymbol{P_C}$, $\boldsymbol{P_G}$, $\boldsymbol{P_T}$). These indicate the conditional probability that the homolog position contains an A, C, G, or T, given that a homolog position exists in the query. The nucleotide with the highest P_N value is the ***consensus character*** at this position. Finally, we have $\boldsymbol{P_{id}}$, which is the probability that the corresponding query position matches the consensus character.

Using the phylogenetic tree and MSA, we can derive the probabilities mentioned previously. Let's take a look at position 1 in the diagram below. The probability that a homolog exists is 1 for this position because no gaps exist. It then following that P_A = 0.75 (3 out of 4 sequences have an A) and P_C = 0.25 (1 out of 4 positions have a C). Now, let's look at position 2 in the diagram, where there is a gap in the Chimp sequence. Since $P_{present}$ = 0, all values of P_N are consequently set to 0.



Now that we have values for the lowest level species in the tree, we can propagate the values up the tree until we reach the root. According to Typhon, we can treat the calculations of $P_{present}$ and P_N independently.

To calculate $P_{present}$ for a node, you take the weighted average of its childrens' $P_{present}$ values. Here, weights are defined to be inversely proportional to the branch length between parent and child. We use the weighted average approach because there hasn't been any strong evolutionary evidence indicating how insertion/deletion events occur across the evolutionary tree. This is because insertion/deletion events are much harder to model mathematically, for several reasons. Gaps can 1) occur in batches (a single event that looks like multiple events) and 2) self-accumulate (gaps happen on top of gaps).

P_N values, on the other hand, are calculated using Felsentein's algorithm with Kimura matrix, which reflects substitution frequency. Kimura models evolution based on sequence mutations. Given an ancestral sequence, the evolution rate parameters specified by the model will determine what the sequence will mutate to after a certain time period. The calculated P_N values are then propagated up the tree such that when we arrive at the root, we set P_{id} to the maximum P_N value. Which P_N is selected reveals the consensus character, which is what we look for in the probabilistic profile when searching for matches.

Region Decomposition

The next step is region decomposition. Despite the MSA example provided below, columns with only gaps are not allowed in a typical alignment. In this MSA, the ideal decomposition is 3 states, or levels of conservation (1= very conserved 2 = somewhat conserved 3 = not conserved).

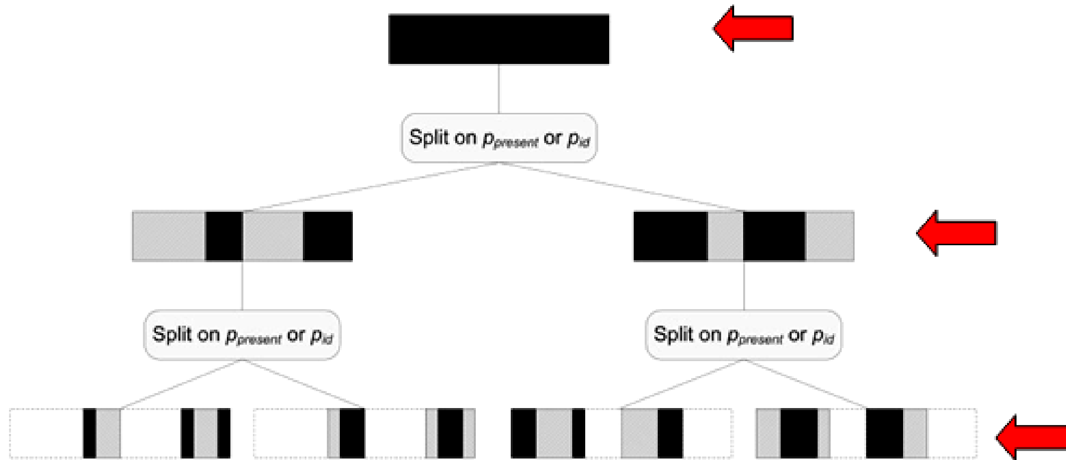
ATTGGAACCCAGGCCA----	AATT-GCGCC-----AA-TT	-----G----C----	ATGG-G---	-ATGCCCAAAAAAT
ATTGGAACTCAGGCCA----	AATT--CGCC-----AA-T-	-----G----C----	AT--G----	-ATGCCCATAAAAT
ATTGGAACCCAGGCCA----	AATT-CG--C-----A-TT-	-----G----T----	A-GGG----	-ATGCCCAAAAAAT
ATTGGAACCCAGGCCA----	A-TTGC-G-C-----AAT-T	-----G-----C---	ATGGGG---	-ATGCCCATAAAAT
1	2	3	2	1

The $P_{present}$ and P_{id} of a region class is defined to be the average over all positions in that region class. We use hidden markov models (HMM) in the algorithm to generate these distinct regions classes. Here, the states of an HMM refer to region classes, and state transitions refer to region boundaries. The emission probability is the probability that you emit a particular position (a particular P_{resent} or P_{id} value) in a given region class, and the transition probability is the probability that you move from/stay within a region class.

There are two ways to do region decomposition. In the simplistic method, a set of region classes is arbitrarily selected based on $P_{present}$ and P_{id}. Using this predetermined set of region classes, an HMM is constructed. Given an observation sequence, we produce an optimal set of regions that could have led to this observation sequence (Viterbi parse). Now, each position is assigned to a particular region class. The tradeoff between staying and transitioning influences the length of resulting regions, which is significant because we want regions to be larger than the span of a seed. The main issue with this approach is that because region classes are pre-fixed, we may end up with unbalanced region classes (some empty, some with lots of regions)

Typhon uses a more sophisticated, hierarchical approach to adaptively choose region classes. Here, a 2-stage HMM is constructed. The HMM is trained using the Baum-Welch algorithm to learn emission probabilities for each state. Now, we can examine the probabilistic profile as a whole and classify the positions into 2 classes, using the Viterbi algorithm. In each split, regions with high sequence identity will be separated from

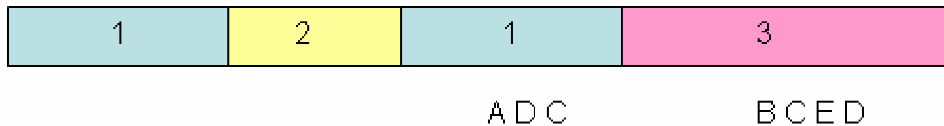
regions with low sequence identity. Each new region class resulting from the split does not have to be contiguous, as demonstrated in the following diagram.

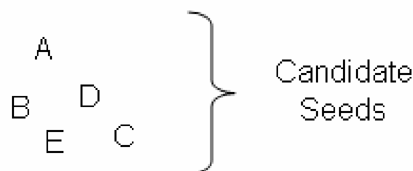


We partition recursively until a user-specified bound on the number of region classes is reached. Empirical evidence indicates that having 40 region classes is optimal. Also, we can preferentially select which regions to decompose further based on the current balance of region class sizes. Ideally, we want the sizes of region classes to be roughly equal so that flexibility in seed assignment is ensured. This approach is much more successful than the simplistic method at partitioning region classes evenly.

Seed Indexing

Now that regions are established in the probabilistic profile, we need to assign a subset of candidate seeds to each region to maximize the expected number of regions matched to a homolog.



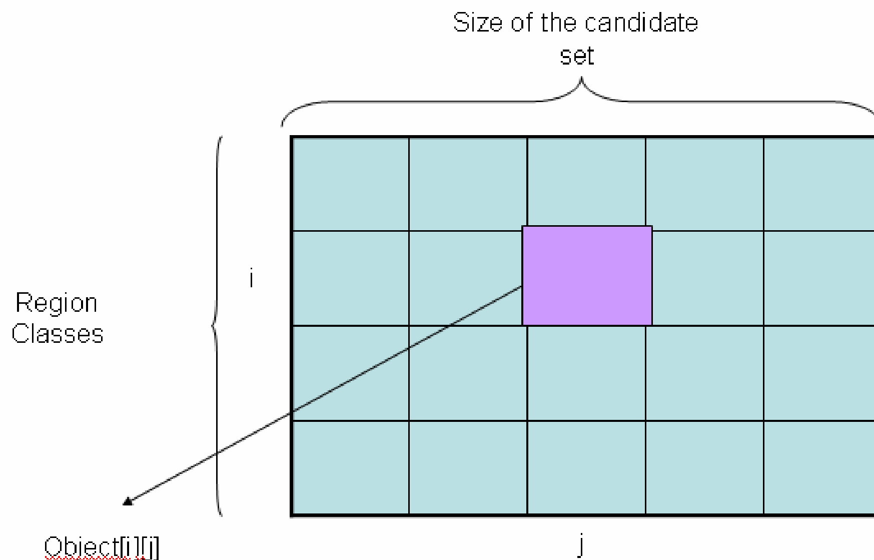


The intuition behind seed indexing on conservation classes is as follows: If we had infinite resources (time and memory), we would take lots of different candidate seeds and make multiple indices. With every index, similarities between the query sequence and MSA database would be examined.

However, faced with a budget of total indices, we need to allocate the budget effectively to maximize the total number of local alignments. If there is a local alignment in a highly conserved region between the query sequence and our MSA database, then we do not need to waste too much seed indexing here. For regions with low sequence similarity or lots of gaps, we also don't need to waste seed indexing here since there aren't alignments here to begin with. But in a region with moderate sequence similarity, we want to spend a significant proportion of our budget here because indexing more seeds actually pays off. The chance of finding good local alignments increases significantly with the number of seeds indexed.

As you can see, the indexing budget is rationed to each region class according to 1) how much effort is required to find good local alignments in each region class and 2) whether alignments even exist in that region class. There are two methods for seed indexing with a budget.

The first seed indexing method is rather simple. Consider a table where the x-axis represents region classes and the y-axis represents sizes of the different candidate subsets used. Each cell, ***object[i] [j]*** in the table, can be described by 2 dimensions: value and weight.



Value gives you the expected number of regions that would match a homolog when region class i is indexed with j seeds in the candidate set.

$$\text{object}[\,i\,][\,j\,] = P_{present} \times P_{hit} \times |C|$$

$P_{present}$, as defined earlier, is simply the probability that a region matches a homolog. P_{hit} is the conditional probability that seeds j would find a homolog in region i, given that the homolog exists. Finally, **|C|** refers to the total number of regions in region class i.

Weight can be thought of as the actual budget for a region class.

Weight [i] [j] =
(total length of all regions in a region class) x (# seeds indexed at each position)

Now, we can search the table and find the set of cells (1 in each row) that maximizes the total value given a fixed budget. In the case demonstrated above, our budget is 112, and the selected cells have total value = 30 + 20 + 32 + 25 = 107.

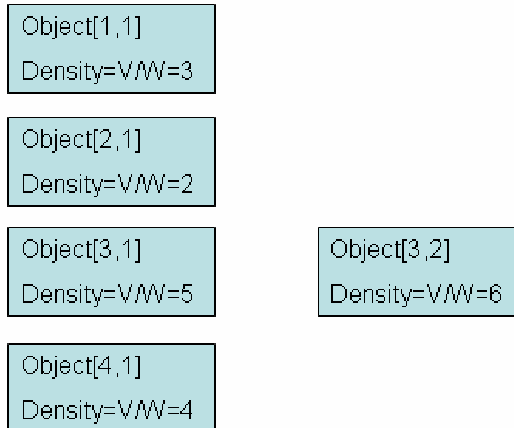
Number of Candidate Seeds

	1	2	3	4	5
Region Class 1	Weight, Value 10,5	Weight, Value 20,30	Weight, Value 30,31	Weight, Value 40,34	Weight, Value 50,40
Region Class 2	Weight, Value 15,8	Weight, Value 30,20	Weight, Value 45,22	Weight, Value 60,24	Weight, Value 75,30
Region Class 3	Weight, Value 12,7	Weight, Value 24,10	Weight, Value 36,32	Weight, Value 48,36	Weight, Value 60,40
Region Class 4	Weight, Value 9,9	Weight, Value 18,10	Weight, Value 27,25	Weight, Value 36,27	Weight, Value 5,30

This selection can be considered as a knapsack problem because you are trying to maximize your total value given a fixed constraint. Knapsack problems are typically solved by Dynamic Programming, which is inefficient (in terms of space) in this application because most region classes are indexed by a rather small subset of seeds. To fix this, Typhon uses a greedy approximation.

In the greedy approximation, ***density*** of each object is defined to be its value divided by its weight. High density objects are desirable, so we want to choose objects in decreasing density. Provided that only 1 object per row can be selected at a time, choose the object with the highest density in column 1 and remove it from the candidate set. This selected object is a keeper. Now, replace the removed object with its neighbor in the next column.

In the example below, object [3] [1] is selected and then replaced with object[3] [2] into the candidate set.



Having a new entrant causes value and weight to be re-calculated. Now, value is the number of additional regions that were matched due to the new entrant. Weight is the extra amount of budget that is consumed due to the new entrant. Re-compute density values for each object since values and weights have changed, and iterate the selection process until the budget limit is reached.

III. Typhon Performance Evaluation

To evaluate the results of Typhon (indexing a subset of seeds at each position), we can compare it to STANDARD (indexing all the seeds at all the positions). Let's look at the 3 aspects of algorithm performance: ***sensitivity***, ***speed*** and ***space***.

Sensitivity

Here, sensitivity is measured by how well you can detect hypothetical homologs. Typhon outperforms STANDARD. Typhon has the greatest sensitivity advantage when used on queries that are distant from species in the MSA (eg: fugu). For any query species, Typhon's sensitivity advantage is greatest for small budgets.

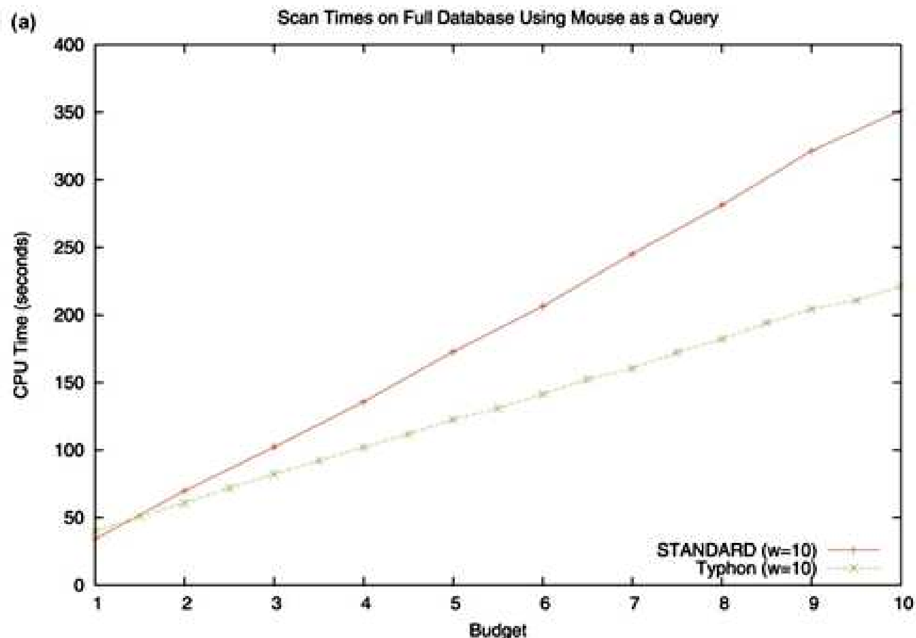
Budget	$w = 10$ Fugu HHAs		Exons		Mouse HHAs		Pig HHAs		$w = 11$ Fugu HHAs		Exons		Mouse HHAs		Pig HHAs	
	S	T	S	T	S	T	S	T	S	T	S	T	S	T	S	T
1	56	**68**	57	**74**	2020	**2137**	**6006**	5998	48	**64**	46	**69**	1856	**2060**	5930	**5949**
1.5	–	**69**	–	**75**	–	**2240**	–	**6049**	–	**65**	–	**71**	–	**2158**	–	**6027**
2	61	**69**	65	**76**	2190	**2259**	6074	**6079**	51	**65**	52	**72**	2089	**2208**	**6044**	6034
3	66	**69**	70	**77**	2258	**2295**	**6089**	6086	56	**67**	61	**74**	2186	**2248**	6067	**6079**
5	68	**69**	72	**77**	2302	**2325**	**6104**	6096	61	**67**	67	**75**	2256	**2286**	6083	**6084**
7	68	**69**	75	**77**	2324	**2339**	**6121**	6098	63	**67**	70	**75**	2283	**2312**	**6105**	6089
10	**69**	**69**	76	**77**	2338	**2346**	**6135**	6115	66	**67**	74	**75**	2301	**2320**	**6118**	6092

Also, it is shown that using information implicit from the MSA will increase search sensitivity (figure below), as does the variable allocation of seeds by region classes.

Budget	Fugu HHAs			Exons			Mouse HHAs			Pig HHAs			
	H	S	T	H	S	T	H	S	T	H	C	S	T
$w = 10$													
1	57	58	**61**	54	56	**65**	2771	3151	**3298**	7166	8651	**8709**	**8709**
1.5	–	–	**69**	–	–	**71**	–	–	**3473**	–	–	–	**8715**
2	64	63	**71**	64	64	**74**	3045	3450	**3528**	7361	8721	**8751**	8749
3	69	69	**72**	69	70	**75**	3140	3573	**3630**	7439	8748	8760	**8761**
5	71	71	**72**	70	72	**77**	3262	3661	**3691**	7499	8765	8771	**8772**
7	71	71	**72**	72	75	**77**	3303	3708	**3719**	7520	8771	**8778**	8775
10	**72**	**72**	**72**	74	76	**77**	3328	3727	**3735**	7554	8775	8789	**8790**
$w = 11$													
1	48	49	**56**	48	47	**58**	2522	2909	**3062**	6755	8492	**8665**	8656
1.5	–	–	**65**	–	–	**64**	–	–	**3297**	–	–	–	**8666**
2	52	52	**65**	54	52	**69**	2850	3286	**3379**	6976	8576	**8734**	8731
3	58	59	**68**	61	61	**73**	3013	3431	**3497**	7145	8646	**8750**	8749
5	64	64	**69**	66	67	**75**	3131	3545	**3591**	7355	8723	**8759**	**8759**
7	66	66	**70**	68	70	**75**	3215	3610	**3644**	7382	8735	**8775**	8771
10	69	69	**70**	71	74	**75**	3271	3665	**3688**	7405	8739	**8782**	8773

Speed

Speed can be decomposed into 2 factors: 1) time used building the index 2) time used scanning the index. Compared to STANDARD, Typhon spends more time building the index and less time scanning the index. Overall, the running times for both methods are highly data dependent and definitive conclusions are difficult to draw.



It appears that Typhon performs better in a full alignment database and STANDARD performs better using a smaller database. Results do indicate that Typhon does not cause a significant performance overhead compared to STANDARD. Further optimizations to parallelize multiple seed scanning should also make Typhon faster.

Space

Typhon and STANDARD are comparable except for an overhead requirement in Typhon. The overhead comes from a look up table for each candidate set pattern.