Implementing Monte-Carlo Tree Search for the Computation of Multiple Sequence Alignment

Juliana De La Vega Fernández
MScAC Candidate
Computer Science Department
University of Toronto
jdelavegaf@cs.toronto.edu

Abstract

This project proposes a solution to the multiple sequence alignment (MSA) problem in computation biology. MSA focuses on finding the highest similarities between more than two sequences of DNA, RNA or protein. The proposed solution uses randomized Monte-Carlo Tree Search in order to find the best solution. Experiments performed on SABRE (SABmark 1.65), a popular protein MSA benchmark. Initial results evince further calibration of the algorithm is needed.

Introduction

MSA is a problem in computational biology where several sequences are aligned in order to obtain the similarities and differences within them (Schroedl 2005). In the case of DNA, the alphabet consists of $\Sigma = \{\texttt{C, G, A, T}\}$ which correspond to the four nucleotide bases. For proteins, the alphabet is made up of the 20 amino acids. Sequence comparisons allow the insertions of gaps, denoted by "-" in order to achieve a better alignment on the remaining characters. The alignment that has the fewest mismatches is usually the most biologically plausible one (Schroedl 2005).

In order to find the optimal alignment, the cost of alignment is calculated. In this case, the minimum cost sequence will be considered as optimal. The cost function that is utilized must take into account the computational efficiency and the biological meaning of the alignment reached. The most used cost function is the *sum-of-pairs* (Schroedl 2005). Considering k sequences of length n, each character in the sequence would have $O(2^k)$ matching possibilities, which would involve calculating $O(k^2)$ *sum-of-pairs* scores, for a total of $O(n^k)$ times. The total computational complexity of this approach is $O(n^k 2^k k^2)$. Using the *sum-of-pairs* cost function for MSA makes it an NP-complete problem.

A different approach is to build a progressive alignment. This will use a profile alignment to merge sequences according to a guide tree. This guide tree will determine which sequences are more closely related, and will merge them first. Progressive alignments have some draw backs, once a gap has been established between two sequences, it will exist for all other sequences aligned. If the gap was a mistake, it cannot be corrected. It is not guaranteed that progressive alignments will obtain the optimal alignment, however they have a better computational complexity. Each pairwise alignment will have a complexity of $O(n^2)$, and the complexity for the number of pairwise alignments that need to occur is $O(k^2)$. There will be $O(k)$ merge steps. This makes the total computational complexity of the method $O(k^2 n^2)$

Previous work has used Monte-Carlo tree search for the MSA problem, learning the gaps in the alignments using BeamNRPA, which resulted in a low memory solution (Edelkamp and Tang 2015). Parallel external-memory best-first search algorithm has also been used, outperforming the Iterative-Deepening Dynamic Programming approach (Hatem and Ruml 2013). Partial Expansion A* has been implemented, reducing the memory requirements of A*. This implementation could align up to seven sequences when infinite node expansion was allowed (Yoshizumi, Miura, and Ishida 2000). Other methods have used A* with quasi-natural gap costs, the common cost model used by biologists (Zhou and Hansen). Other researchers report using genetic algorithms (Kumar 2015), Dijkstra's algorithm, iterative deepening A*, best first search, amongst others (Schroedl 2005).

Problem Formulation

Sequence Alignment

For a set of n sequences, $S = s_1, ..., s_n$, where each s_i is made up of the alphabet Σ. The corresponding set of aligned sequences, $A = a_1, ..., a_n$ is made up of the alphabet Σ', where: $\Sigma' = \Sigma \cup \text{"} - \text{"}$. The new element, "-", corresponds to a gap. The aligned sequence a_i corresponds to the sequence s_i when the characters "-" are removed from the former. In the set A, the characters of the s_i sequences are shifted by adding gaps in order to make a k^{th} character of s_u match the j^{th} character of s_v ($s_u, s_v \in S$, and k and j are characters in s_u and s_v respectively). This will make $a_u[m] = a_v[m]$, where m is the new position of the k^{th} character of s_u and the j^{th} character of s_v in the A set (Edelkamp and Tang 2015).

Gaps

As stated before, a gap is a single or a sequence of characters "-". These correspond to the empty characters of $a_i \in A$.

For the ease of viewing, aligned sequences are displayed one on top of the other, each one in a different row, displaying matching characters by column. Gaps are used to shift the characters in a sequence so that more matches are achieved. A gap can can either represent a deletion or an insertion of a character on a sequence. Another type of modification that can be seen in alignments are substitutions. This happens when two different characters are aligned in the same column (Edelkamp and Tang 2015).

Scoring

For pairwise sequence alignment, when $n = 2$, the Levenshtein distance, or Edit distance, is used to compute the score achieved for the alignment. For the alignment $A = a_1, a_2$, where each a_i is made up of k characters, given any character j, the *sum of similarities* f for all columns is (Edelkamp and Tang 2015):

$$F(a_1, a_2) = \sum_j^k f(a_1[j], a_2[j])$$

Extending to MSA, for n alignments (Edelkamp and Tang 2015):

$$F(A) = F(a_1, ..., a_n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} F(a_i, a_j)$$

Multiple Sequence Alignment

Given S, find A such that A belongs to the set of all MSA that can be generated from S, and minimizes $F(A)$. This will make A be the optimal MSA for set S

This makes MSA a minimal-cost problem with the following features:

- The state space forms a lattice.
- the branching factor is very large O(2^n) where n is the number of sequences to be aligned.
- The distribution edge cost can take a large number of values.

Implementation

Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a method used for finding optimal decisions in a given domain by taking random samples from the decision space and building a search tree with the results obtained (Browne et al. 2012). Incrementally and asymmetrically, a tree is built. The *tree policy* is used to find the next node in the tree, while maintaining a balance between the exploration of unfamiliar nodes, and the exploitation of promising nodes. A *simulation* takes place from the selected node. The statistics obtained from the simulation may then update the node's ancestors according to the result (Browne et al. 2012).The moves in the simulation follow a *default policy* which could be a uniform random set of moves.

One of the great benefits the MCTS method has, is that given a constraint (computational budget in terms of time, memory, or iterations), the values that the MCTS reports correspond only to the terminal states at the end of each simulation. The intermediate steps do not have to be evaluated.

Algorithm 1 MCTS Pairwise Sequence Alignment

```
1:  function UCTSEARCH
2:      create a root node $v_0$ with state $s_0$
3:      while within computational budget time do
4:          $v_l \leftarrow$ TREEPOLICY($v_0$)
5:          $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
6:          BACKUP($v_l, \Delta$)
7:      return $s$(BESTCHILD($v_0, 0$))
8:
9:  function TREEPOLICY($v$)
10:     while $v$ is not terminal do
11:         if $v$ is not fully expanded then
12:             returnEXPAND($v$)
13:         else
14:             $v \leftarrow$ BESTCHILD($v, C_p$)
15:         return $v$
16:
17: function EXPAND($v$)
18:     Choose $a \in$ untried actions from $A(s(v))$
19:     add a new child $v'$ to $v$.
20:      with $s(v') = f(s(v), a)$
21:      and $a(v') = a$
22:     return $v'$
23:
24: function BESTCHILD($v, c$)
25:     return $\underset{v' \in \text{children of } v}{\arg\min} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
26:
27: function DEFAULTPOLICY(s)
28:     while $s$ is non-terminal do
29:         choose $a \in A(s)$ uniformly at random
30:         $s \leftarrow f(s, a)$
31:     return reward for state s
32:
33: function BACKUP($v, \Delta$)
34:     while $v$ is not null do
35:         $N(v) \leftarrow N(v) + 1$
36:         $Q(v) \leftarrow Q(v) + 1$
37:         $v \leftarrow$ parent of $v$
```

Pairwise Alignment Algorithm

Initially, a MCTS was implemented for performing pairwise alignments (see Alg. 1). In this algorithm the states will correspond to the alignment, the actions will be either matching or mismatching, insertion or deletion (gap). Given $S = s_1, s_2$ evaluated at characters $i$and j respectively, the actions will be described as:

- Matching: when $s_1[i] = s_2[j]$, obtaining $a_1[k] = s_1[i]$ and $a_2[k] = s_2[j]$
- Mismatching (substitution): when $s_1[i] \neq s_2[j]$, obtaining $a_1[k] = s_1[i]$ and $a_2[k] = s_2[j]$
- Insertion: on $s_2[j]$ with respect to $s_1[i]$, obtaining $a_1[k] =' -'$ and $a_2[k] = s_2[j]$
- Deletion: on $s_2[j]$ with respect to $s_1[i]$, obtaining $a_1[k] = s_1[i]$ and $a_2[k] =' -'$

Where the reward for making a match between two characters is 0, the reward for an insertion or a deletion is 2, and the reward for a mismatch is 3. These rewards were arbitrarily chosen so that matches are enforced first. In second place, a single gap is more likely to be aligned instead of a mismatch. And in third place, a mismatch would be more likely to be aligned than two consecutive gaps.

The MCTS class receives the two sequences that will be aligned, and the search is started when the function UCTSearch is called. This function creates the root node, which has an empty state, a Q value of 0, and 1 visit to its node (the current one). For the pairwise alignment, a computational budget of 30 seconds was used. within that budget, the tree policy begins.

The tree policy will perform the *while* loop as long as s_1 and s_2 have not been completely traversed. This means, as long as the node that is being evaluated does not contain a full alignment state. Inside this loop, each node will be expanded *if* it doesn't have all of its children. The function Expand selects one of the nodes available untried actions (match/mismatch, insertion or deletion), adding the child node to the node being evaluated, and returning the child node.

If the node has all of its children, then the function BestChild is called, selecting the best child out of the node's children and updating the reward on the evaluated node. After the Tree Policy, the Default Policy will perform rollouts that update the reward for the leaf nodes by performing random actions that lead to a terminal state. Only the rewards are updated, not the states. This is performed to select from the current nodes, the best one to continue exploring.

The Backup function would then proceed to update the statistics from the leaf nodes back to the root node. This ensures that the search will follow the optimal path of nodes that leads to the best reward. When the computational time allotted is up, the best child of the root node is selected. This child's state is returned by the UCTSearch function, containing the alignment of the two sequences.

MSA Algorithm

The previously described algorithm was extended to be capable of MSA (see Alg. 2). For this particular case, a modi-

Algorithm 2 Algorithm for MSA

```
1: function UPPERSEARCH(())
2:     create a root node $v_0$ with empty state $s_0$
3:     while within computational budget time do
4:         for $i$ in number of sequences $X$ do
5:             if in first iteration then
6:                 $x_1, x_2 \leftarrow$ RANDOMCOMBINATION($X$)
7:                 $s_l, r_l \leftarrow$ UCTSEARCH($x_1, x_2$)
8:                 $v_l \leftarrow$ UPDATEALIGNMENT($v_l, s_l$)
9:                 $x_* \leftarrow$ FLATTENALIGN($s_l$)
10:                $X = (X - \{x_1\}) - \{x_2\}$
11:            else
12:                $x_1 \leftarrow$ RANDOMSEQUENCE($X, x_*$)
13:                $s_l, r_l \leftarrow$ UCTSEARCH($x_*, x_1$)
14:                $v_l \leftarrow$ UPDATEALIGNMENT($v_l, s_l$)
15:                $x_* \leftarrow$ FLATTENALIGN($s_l$)
16:                $X = X - \{x_*\}$
17:        Back up best child node of $v_0$
18:    return $s(v_0$ best child)
19:
20: function RANDOMCOMBINATION($X$)
21:    $p =$ all pairs of$X$
22:    for all possible $i$ pairs do
23:        $trial_i \leftarrow$ UPPERDEFAULTPOLICY($p[i,0], p[i,1]$)
24:    return $p[\arg\min(trial)]$
25:
26: function RANDOMSEQUENCE($X, x_*$)
27:    for all $x_i$ in $X$ do
28:        $trial_i \leftarrow$ UPPERDEFAULTPOLICY($x_i, x_*$)
29:    return $x_{\arg\min(trial)}$
30:
31: function UPPERDEFAULTPOLICY($x_1, x_2$)
32:    while $s$ is non-terminal do
33:        choose $a \in A(s)$ uniformly at random
34:        $s \leftarrow f(s, a)$
35:    return reward for state s
36:
37: function FLATTENALIGN($s$)
38:    for all $i$ in length of $s$ do
39:        if $s[0]$ character $i$ is a gap then
40:            $s[0, i] = s[1, i]$
41:    return $s[0]$
42:
43: function UPDATEALIGNMENT($v, s$)
44:    $s_{l-1} = v.s$
45:    $s_l = s_{l-1} + s$
46:    create node $v_l$ with state $s_l$
47:    return $v_l$
```

fied version of the MCTS is performed to select the profiles to align in the progressive alignment. The MSA MCTS class contains, various functions. Upon being called, it will initiate function UpperSearch, where a root node with an empty state will be created. Within its computational budget, it will proceed to select all possible pairings between the sequences by calling RandomCombination. This function will perform rollouts for all of these combinations, and will return the combination that performs the best to UpperSearch. UCTSearch, (from alg. 1) is called, receiving the two sequences that aligned the best according to the rollouts, and will proceed to align them. The state of the alignment will be returned to UpperSearch.

The function UpdateAlignment is now called, it will set the new state to a new node, and make it the current node. The function FlattenAlign will make the new alignment into a 1-D array for future use. Finally, the set of X sequences is updated to represent the sequences that have not been aligned. In the for loop in UpperSearch, whenever the first two sequences are aligned, and any additional one will be aligned to them, the function RandomSequence is called. This function will perform rollouts with all the combinations of each element in the set of sequences X with the flattened alignments obtained before.

When the for loop in UpperSearch is completed, the full MSA is backed up to the root node. Finally, when the computational budget is reached, the best child is backed up to the root node. For experimentation purposes, all MSA's obtained by the algorithm were stored and analyzed using the *sum-of-pairs*-score and Total-Column-score (Edgar 2016).

Experimentation

Database

The algorithm was initially tested on **SABmark**, a publicly available data-set of MSA problems derived from the SCOP classification (Van Walle, Lasters, and Wyns 2005). This database covers the entire known fold space using protein sequences with very low to low and low to intermediate similarity. This means that these sequences have 0 to 50% identity. SABmark has two alignment sets: Twilight Zone and Superfamilies. The database is organized into Fasta format files, and contains 424 problems that range from 3 to 25 sequences each.

A second test of the algorithm is currently being performed on **BAliBase2.0**, which contains high quality, manually constructed MSAs with detailed annotations based on 3-D structural superposition (Bahr et al. 2001). Version 2.0 of the database has alignments of structural repeats, transmembrane sequences, and circular permutations (Bahr et al. 2001). This data-set was specifically designed as an evaluation resource for alignment. It consists of 167 reference alignments, made up of more than 2100 sequences in total (Bahr et al. 2001). The data-set is organized into RSF and MSF format files for the reference alignments, and Fasta format for the unaligned sequences.

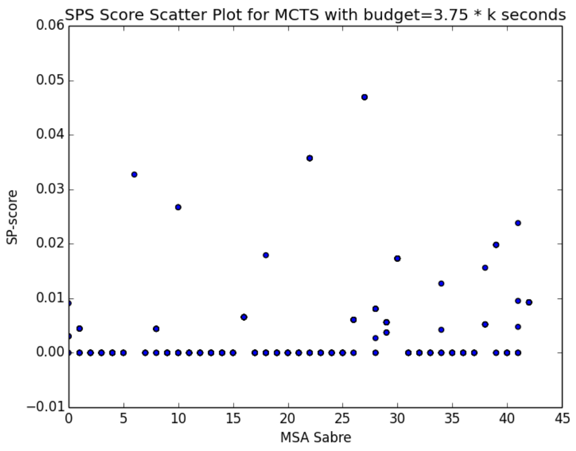

Figure 1: MSA MCTS results for 44 MSA problems in the SABmark database, using a budget time of 3.75 seconds per residue in the sequence.

Scoring

As mentioned before, the *sum-of-pairs* score and the Total-Column score were used to determine the quality of the alignment. The *sum-of-pairs* score is described as:

$$SP(a_1, a_2) = \sum_{m=1}^{k} S(a_i[m], a_j[m])$$

The *sum-of-pairs* score determines the programs success in aligning input sequences, and it corresponds to the ratio of the *sum-of-pairs* for all pairs of characters in every column of the alignment by the *sum-of-pairs* score for the reference alignment (Pais et al. 2014). The *sum-of-pairs* score must be equal to or less than the scores obtained from the reference pairwise alignments, and is extended to all $\binom{n}{2}$. This score evaluates the probability that evolutionary events occurred between two sequences. The score will be representative of the evolutionary distance between the pairs of sequences evaluated (Carrillo and Lipman 1988).

The Total-Columns score is a binary score function that is used to test the algorithms ability to align correctly all sequences (Carrillo and Lipman 1988). This score is calculated by considering the ratio of the *sum-of-pairs* score by the number of columns in the alignment. If all residues are correctly aligned, the Total-Columns score will be 1 (Carrillo and Lipman 1988).

A previous implementation of these scores, by Robert C. Edgar was used (Edgar 2016). This implementation was developed in C++ in 2004. It is publicly available for use, however it is deprecated. During the testing phase the file `qscore.h` was modified to update it for use. All the results from the algorithm were saved to Fasta files, and these were evaluated in C++ using a bash file. The results obtained were analyzed in python.

Results

The algorithm was first tested on aligning three short DNA sequences of 8 residues each in order to guarantee that it was

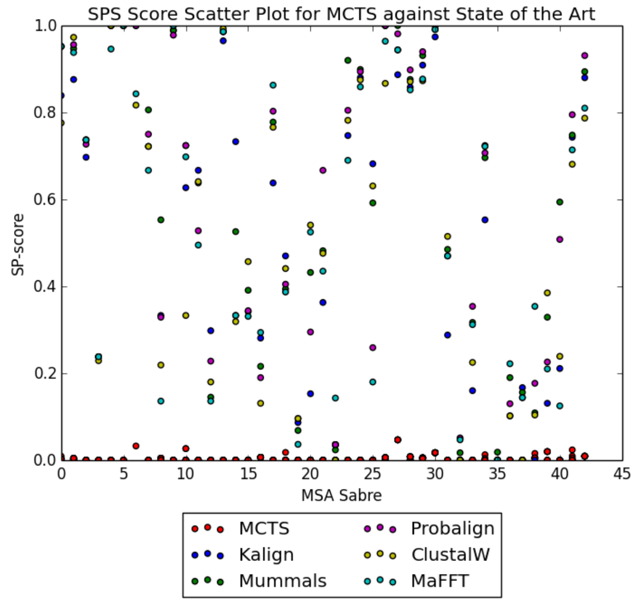

Figure 2: Comparisons of the MSA MCTS results to the benchmarks in the 44 SABmark MSA problems. The benchmarks evaluated were Clustal-W, Kalign v.2.03, Mummals, MaFFT 6.603einsi, and Probalign v1.0.

functional and not achieving merely random results. One example of a MSA of three short sequences that was used is:

```
U = 'ATCGAAT'
V = 'TTGGATT'
X = 'GTGGAAT'
```

Using the scoring, and saving all possible terminal node states, and with a budget of 15 minutes, the MSA MCTS achieved a *sum-of-pairs* score of 0.714, and a Total-Column score of 0.571, which is a better than random result for alignment. The aligned sequence that achieved this score was:

```
U:'AT-CGAAT-'
X:'GTG-GAAT-'
V:'TTG-GA-TT'
```

Said alignment was found in seven runs of the algorithm. On the other hand, in these runs other terminal but sub optimal state nodes reported a *sum-of-pairs* score of 0.238, and a Total-Column score of 0.000. These terminal nodes represent states that might have shown promising rewards up to a certain point of the alignment of the first sequences, but failed when aligning the final sequences. An example is shown below:

```
V:'TTGGATT------'
X:'GTGGA--AT----'
U:'----ATC--GAAT'
```

Sequences V and X displayed an alignment that was more plausible. However, when aligning sequence U the algorithm prioritized matching the first residue, rather than mismatching it, reducing its final score. Figure 1 portrays the results obtained after aligning 44 MSA problems from the SABmark data-set. It displays the *sum-of-pairs* score obtained in each of these. The Total-Column score is not displayed because it was 0 for all alignments. The *sum-of-pairs* score obtained in all sequences aligned was very low, it didn't go over 0.06.

Figure 2 displays the same results as figure 1, expanded to the results obtained by the benchmarks. The benchmark with the best *sum-of-pairs* score was Mummals, achieving an average score of 0.599. Probaling followed Mummals, with an average score of 0.578. With very similar scores, in descending order, MaFFT obtained an average score of 0.548, Kalign obtained 0.546, and ClustalW obtained a score of 0.545.

Critical Analysis

Brief Benchmark Description

Mummals is a program specifically built for multiple protein sequence analysis using probabilistic consistency (Pei and Grishin 2006). It improves the alignment quality using hidden Markov models with multiple math states describing local information (Pei and Grishin 2006). It is reported, and in this case experimentally proved that Mummals achieves statistically best accuracy among other leading algorithms such as MaFFT, ProbCons, and Muscle (Pei and Grishin 2006).

Probalign uses two techniques in its algorithm in order to attain better accuracy in its MSA. It uses pairwise posterior probabilities of residues to align sequences, and the partition function methodology in order to estimate said probabilities (Roshan and Livesay 2006). Its alignments are generally better than those obtained by the methods Muscle, MaFFT and Probcons. Probalign is an open source code project.

MaFFT, as its name suggests, uses the fast Fourier transform in order to identify homologous regions (Katoh et al. 2002). The transform converts amino acid sequences into volume and polarity values for each residue (Katoh et al. 2002). It has two heuristics, one progressive and one iterative. These heuristics improve the CPU time without sacrificing the quality of the alignment (Katoh et al. 2002).

Kalign is an algorithm that implements Wu-Manber string-matching algorithm improving the accuracy and speed of MSA (Lassmann and Sonnhammer 2005).It is reported to be 10 times as fast as ClustalW, and is as accurate as the best methods (Lassmann and Sonnhammer 2005). The algorithm calculates pairwise distances in order to build a guide tree, it calculates the distances using the mentioned algorithm, and uses dynamic programming in order to align the profiles (Lassmann and Sonnhammer 2005).

ClustalW is an algorithm built for multiple protein or nucleotide sequence alignment (ThompsonJD). It achieves the alignment in three steps. It performs a pairwise alignment, which it uses to create a guide tree, and finally perform a progressive alignment(ThompsonJD).

Strengths and Weaknesses of Approach

Within the strengths of the approach, the node class implemented allows for ease of accessibility to the complete state (alignment) at any point of the tree. I believe that the results would improve given a greater time to perform the search. For the SABmark test, each MSA problem was given 30 seconds for every 8 residues its longest sequence had in order to perform the pairwise alignment. This would imply, for a given MSA of 3 sequences, whose longest sequences had 200 residues only 37 minutes to build the alignment. It is still to be proven by the results in the BAliBase for which more time was allotted.

One of the weaknesses of the approach is that the Monte-Carlo Tree Search that was implemented is a basic one. Nevertheless, it has the strength that even the basic one is capable of performing a short MSA. This suggests that further fine tuning of the parameters in the Tree Search may lead to an efficient MSA solution. These parameters include: the C parameter that balances the exploitation-exploration dilemma, and the cost of each action. Hindsight optimization can be improved in the MCTS, this would provide a more formal basis to determinate the states given the complete sequences(Browne et al. 2012). Also, it could be possible to implement sample based planner to achieve a better performance of the MCTS, such as Forward Search Sparse Sampling, or Rapidly-exploring Random Trees.

When dealing with more than 5 sequences, and when the length of the sequences exceeds 200 residues, the approach is resource intensive in terms of CPU and memory use. This could be solved by implementing move pruning of the sub-optimal moves in the search tree. Likewise, the UpperSearch method could be improved by fully developing the Monte-Carlo Tree search in the UpperLevel, rather than only performing the DefaultPolicy. This could be done by performing a Nested Monte-Carlo Tree Search. Given a powerful computer, either leaf parallelization, or complete tree parallelization could prove useful to reduce running time of the algorithm and to broaden the search (Browne et al. 2012).

It would also be beneficial to try to implement a tree as described by Stefan Edelkamp (Edelkamp and Tang 2015), where only the positions of gaps are recorded in the state of each node. This would solve the problem of computational resources, as keeping track of the gaps requires less memory than keeping track of the complete alignment. In terms of memory, another improvement would be to implement the algorithm in C++ so that pointers could be used. This would reduce the memory used by the current python implementation.

Although the project proposal described the first test on BAliBase, the web page in which the download can be performed was being updated, and has been available for download since the 12th of December 2016. Before this date, since November, no downloads could be performed from it.

Comments on other algorithms

It was proposed to implement A* Search, Recursive Best-First Search and Simplified Memory Bounded A*. Starting by the latter, Memory Bounded A* would provide the advantage that it is a robust, optimal algorithm that uses limited amounts of memory, and given enough time it could solve problems in which A* runs out of memory (Russell et al. 2003). Simplified Memory Bounded A* proceeds like A*, expanding the best leaf until the memory is full, and then dropping the worst leaf node it is able to continue its search.

Recursive Best-First Search works in a similar way to Best-First Search, while using a linear space (Russell et al. 2003). It is more efficient than Iterative Deepening A*, but it suffers from excessive node regeneration. This would be troublesome in the MSA problem because of the large space that would be needed to handle. It would probably not arrive to an optimal solution.

Finally, A* search is an optimal algorithm to analyze a tree search. It uses two functions in order to estimate the cost of the cheapest solution, $g(n)$ which gives the cost from the root node to node n, and $h(n)$ which is the estimated cost of the cheapest path from n to the goal (Russell et al. 2003). A* is guaranteed to arrive to an optimal solution given the appropriate heuristic function. The problem here is the lack of that heuristic function for the particular problem of MSA. This is why MCTS was implemented instead of any of these three algorithms.

Future Work

When finished, testing on the BAliBase will be analyzed. Adjustments will be made to the UpperSearch Algorithm in order for it to perform a complete MCTS. The pairwise MCTS algorithm, will be updated in order to have an improved Q value function that better accommodates to the MSA problem. Other variants to the rollout policy will be studied. Implementation of the methods described by Stefan Edelkamp will also be studied (Edelkamp and Tang 2015). It could be useful to implement a MSA that rather than performing a pairwise matching of the sequences, matches all of them at the same time.

References

Bahr, A.; Thompson, J. D.; Thierry, J.-C.; and Poch, O. 2001. Balibase (benchmark alignment database): enhancements for repeats, transmembrane sequences and circular permutations. *Nucleic Acids Research* 29(1):323–326.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):1–43.

Carrillo, H., and Lipman, D. 1988. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics* 48(5):1073–1082.

Edelkamp, S., and Tang, Z. 2015. Monte-carlo tree search for the multiple sequence alignment problem. In *Eighth Annual Symposium on Combinatorial Search*.

Edgar, R. C. 2016. Qscore mutliple alignment scoring software.

Hatem, M., and Ruml, W. 2013. External memory best-first search for multiple sequence alignment. In *AAAI*.

Katoh, K.; Misawa, K.; Kuma, K.-i.; and Miyata, T. 2002. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic acids research* 30(14):3059–3066.

Kumar, M. 2015. An enhanced algorithm for multiple sequence alignment of protein sequences using genetic algorithm. *EXCLI Journal* 14:12321255.

Lassmann, T., and Sonnhammer, E. L. 2005. Kalign–an accurate and fast multiple sequence alignment algorithm. *BMC bioinformatics* 6(1):1.

Pais, F. S.-M.; de Cássia Ruy, P.; Oliveira, G.; and Coimbra, R. S. 2014. Assessing the efficiency of multiple sequence alignment programs. *Algorithms for Molecular Biology* 9(1):1.

Pei, J., and Grishin, N. V. 2006. Mummals: multiple sequence alignment improved by using hidden markov models with local structural information. *Nucleic Acids Research* 34(16):4364–4374.

Roshan, U., and Livesay, D. R. 2006. Probalign: multiple sequence alignment using partition function posterior probabilities. *Bioinformatics* 22(22):2715–2721.

Russell, S. J.; Norvig, P.; Canny, J. F.; Malik, J. M.; and Edwards, D. D. 2003. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Upper Saddle River.

Schroedl, S. 2005. An improved search algorithm for optimal multiple-sequence alignment. *Journal of Artificial Intelligence Research* 23:587623.

ThompsonJD, H. Gibsontj (1994): Clustalw: improving the sensitivity of progressive multiple sequence alignments through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res* 22:4673–80.

Van Walle, I.; Lasters, I.; and Wyns, L. 2005. Sabmarka benchmark for sequence alignment that covers the entire known fold space. *Bioinformatics* 21(7):1267–1268.

Yoshizumi, T.; Miura, T.; and Ishida, T. 2000. A* with partial expansion for large branching factor problems. In *AAAI/IAAI*, 923–929.

Zhou, R., and Hansen, E. K-group a* for multiple sequence alignment with quasi-natural gap costs. *16th IEEE International Conference on Tools with Artificial Intelligence*.