Solve MSA problem by DP, A*, and GA

∗ Name:Renyang Guan ID:519021911058 E-mail: guanrenyang@sjtu.edu.cn

1 Problem Restatement

Multi-sequence alignment problem, also referred as MSA problem, is of vital significance to biological research. In this project, my task is to find the most *similar* sequence in the database to the given query.

The definition of *similar* is clarified in task instruction, which says the group of sequences with the smallest *cost* value is the most *similar* alignment. But what is the meaning of *cost*? For an alignment with 2 sequences, the cost matrix is revealed below: For an alignment with size greater

	MATCH	MISMATCH	GAP
Cost	0	3	2

than 2, the *cost* of it is the sum of all pairwise costs. Additionally, the **GAP-GAP** alignment is considered as **MATCH** with *cost* equalling to 0.

There are 5 queries for pairwise alignment and 2 queries for three-sequence alignment in this project. The goal of pairwise alignment is to find the most *similar* sequence to the given query, and that of three-sequence alignment is to find 2 sequences in the database, together with the given query yielding the smallest *cost*.

2 Algorithm

2.1 Dynamic Programming

Dynamic Programming algorithm for MSA problem is also called *Hirschberg's algorithm*. For each query we use this algorithm for each possible alignment to calculate a *cost* and find the most *similar* sequence by comparing *cost*s.

Pairwise Alignment

Suppose x and y are two sequences with the cost of them to be calculated. m and n are the lengths of x and y, respectively. Let x_i denote the ith character of x and y_j denote the jth character of y.

Then we define a matrix $Cost$ in size of $(m+1, n+1)$. $Cost(i, j)$ is the element in the ith row and jth column, denoting the **optimal** *cost* of sequences $(x_1, x_2, \cdots, x_i)$ and $(y_1, y_2, \cdots, y_j)$. Obviously, **the optimal *cost* of x and y is stored in $Cost(m, n)$.** , which is shown in Fig. **??**.

Since any optimal path passing through the current node $Cost(i, j)$ must include an optimal path passing through one of its predecessors, we only need to consider few predecessors of the current node, e.g. $Cost(i-1, j-1)$, $Cost(i-1, j)$, and $Cost(i, j-1)$, instead of all possible paths.

We could get the formula to calculate $Cost$ in a recursive way:

$$\begin{aligned} Cost(i,j) = \min[&Cost(i-1,j) + cost(x_i, \text{Gap}),\\ &Cost(i,j-1) + cost(\text{Gap}, y_j),\\ &Cost(i-1,j-1) + cost(x_i, y_j)] \end{aligned}$$

where $cost(p, q)$ denotes the cost of character p and q. When p xor q equals to GAP, $cost(p, q)$ equals to 2.

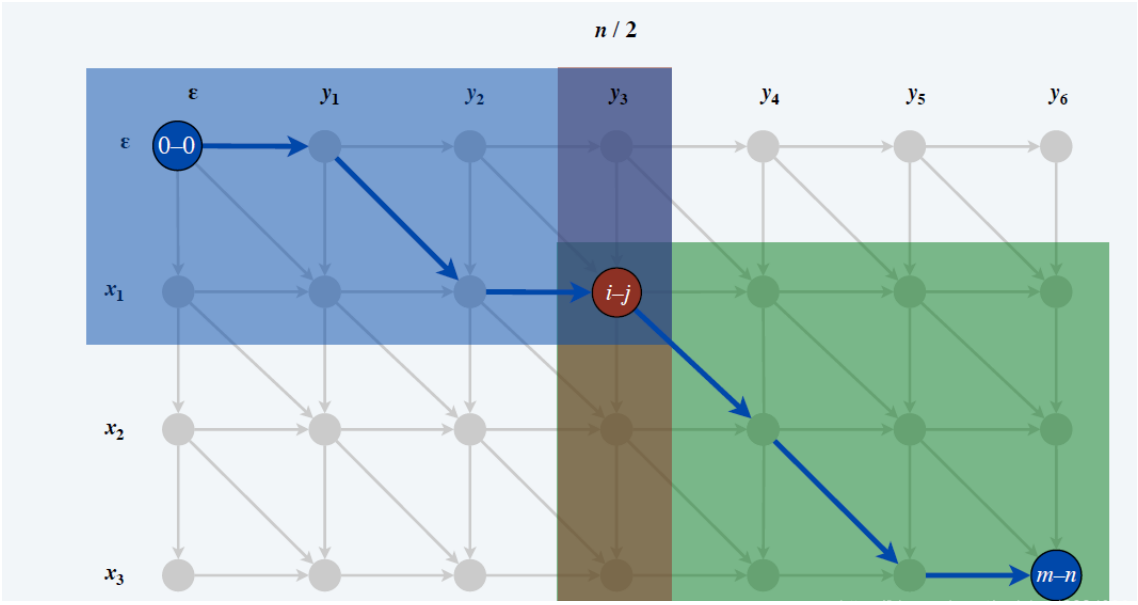

Figure 1: *Cost* Matrix

Certainly, we can use *recursion* to calculate $Cost$, because we only care about $Cost(m, n)$ but not the other entries. However, the space complexity of *recursion* function is unbearable since the stack space is limited.

As a result, we use a method called **Memoized Storage** instead, which means all the space (martix $Cost$) is allocated in advance and the intermediate results are saved. We refer to the intermediate results by looking up the table $Cost$, the complexity of which is constant.

Three-sequence Alignment

The algorithm for three-sequence alignment is essentially identical to that for pairwise alignment, except that the dimension of tensor $Cost$ becomes higher and the recursive formula is more complicated.

Suppose x, y, and z are three sequences with the cost to be calculated. Their lengths are m, n, and k respectively. The size of tensor $Cost$, which stores the intermediate optimal results, is $(m+1, n+1, k+1)$. The recurrence formula for calculating $Cost(i, j, k)$ is as follows:

$$\begin{aligned} Cost(i,j,k) = \min[&Cost(i-1,j,k) + 2 \times cost(x_i, \text{Gap}),\\ &Cost(i,j-1,k) + 2 \times cost(y_j, \text{Gap}),\\ &Cost(i,j,k-1) + 2 \times cost(z_k, \text{Gap}),\\ &Cost(i-1,j-1,k) + cost(x_i, y_j) + 2 \times cost(\cdot, \text{Gap}),\\ &Cost(i-1,j,k-1) + cost(x_i, z_k) + 2 \times cost(\cdot, \text{Gap}),\\ &Cost(i,j-1,k-1) + cost(y_j, z_k) + 2 \times cost(\cdot, \text{Gap}),\\ &Cost(i-1,j-1,k-1) + cost(x_i, y_j) + cost(x_i, z_k) + cost(y_j, z_k)] \end{aligned}$$

Completeness and Optimality

Dynamic Programming algorithm is **complete and optimal**, since the lengths of sequences are limited and the solution is calculated through a strictly proven recursive formula.

Time and Space Complexity

Suppose that k denotes the number of sequences in an alignment and l_i denotes the length of the kth sequence for $i \in [1, k]$.

Both of the time complexity and space complexity of *Dynamic Programming* algorithm with these preconditions are $O(\prod_{i=1}^{k}(l_i + 1))$, because we allocate a tensor of size $\prod_{i=1}^{k}(l_i + 1)$ and traverse each element in this tensor once. Each traversal is to calculate $Cost(i, j)$ according to the above formulas , the complexity of which is $O(1)$.

Let l denote the geometric mean of l_i for $i \in [1, k]$, **the time complexity and space complexity are both $O(l^k)$.**

2.2 A* Algorithm

A algorithm* is one kind of informed search algorithm. It inherits the idea of *Uni-cost search* algorithm and adds a *heuristic function* to the search cost.

Pairwise Alignment

Suppose x and y are two sequences and m and n are the lengths of them, respectively. Let $x[1 : i]$ denote the sequence $(x_1, x_2, \cdots, x_i)$ and $y[1 : j]$ denote the sequence $(y_1, y_2, \cdots, y_j)$. $x[1 : 0]$ and $y[1 : 0]$ means adding gap at the beginning of the sequences. Before explaining the algorithm, we need to formulate the basic attributes of the nodes on the search tree:

1. *States*: For any i and j, $i \in [1, m]$ and $j \in [1, n]$, each possible alignment of $x[1 : i]$ and $y[1 : j]$ is a state. For example, the states of `a` and `b` is {`('','')`,`('-','b')`,`('a','b')`,`('a','-')`}.

2. *Initial State*: The alignment of two sequences $x[1 : 0]$ and $y[1 : 0]$, which is `('','')`.

3. *Actions*: For pairwise alignment task, we define three *actions*: *gapX*, *gapY*, *gapNone*. For example, if the initial sequences are `ab` and `cd`, *gapX* means `('a','c')` → `('a-','cd')`, *gapY* means `('a','c')` → `('ab','c-')`, and *gapNone* means `('a','c')` → `('ab','cd')`.

4. *Transition model*: The *actions* have their expected effects, each of which generates a new *state*.

5. *Goal test*: This checks whether a state includes all the characters of the initial sequences.

6. *Path Cost*: *gapX* or *gapY* costs 2. If the added two characters are the same, *gapNone* costs 3, otherwise 0.

7. *Heuristic function*: $= cost(\cdot, \text{gap}) \times [(m - i) - (n - j)]$, According to Ref .[1], the heuristic function is *admissible*.

8. *Evaluation Cost = Path Cost + Heuristic function*

Since *Evaluation cost* is the estimated cost of the cheapest solution, *A* search* algorithm continuously transforms the current node to the node with lowest *evaluation cost*, until the goal is achieved.

Three-sequence Alignment

There are two main differences between three-sequence alignment algorithm and pairwise alignment algorithm, one is in *state* and the other is in *action*.

With respect to *state*, the number of entries is one more than before. We have to define more *action*s, e.g.*gapX*, *gapY*, *gapZ*, *gapXY*, *gapXZ*, *gapXZ*. *gapNone*. Meanwhile, the *path cost* and *heuristic function* also needs to be adjusted accordingly.

Completeness and Optimality

Because the heuristic function is *admissible* and we use the *tree-search version* of *A* search*, according to Ref. [2], *A* search* algorithm is **optimal**. Furthermore, *A* algorithm* is **complete** because the nodes with their costs less than or equal to the cost of optimal solution are finite.

2.3 Genetic Algorithm

Genetic algorithm is a type of heuristic algorithm, which draws lessons from the ideas of natural *selection*, *crossover*, and *mutation* in biology.

The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the *fitness* of every individual in the population is evaluated, multiple individuals are selected from the current population (based on their fitness), and modified to form a new population.

By the way, we formulate *genetic algorithm* in a general way, regardless of the number of sequences in an alignment. The basic elements of *genetic algorithm* is defined as follows. Suppose

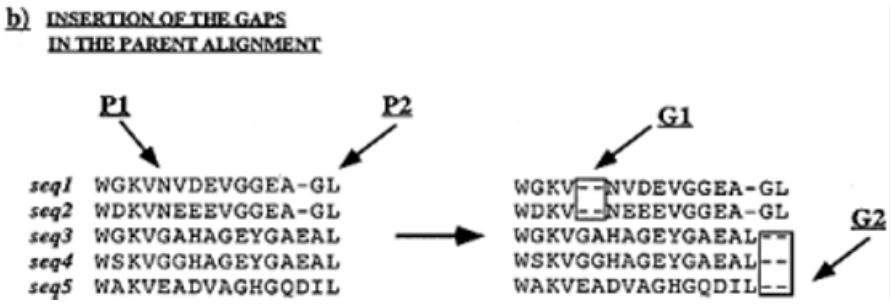

Figure 2: Mutation

that k denotes the number of sequences in an alignment and l_i denotes the length of the kth sequence for $i \in [1, k]$.

1. *Individual*: Individual is a possible alignment of all the sequences l_i, for $i \in [1, k]$.

2. *Fitness*: Let A denote the alignment and A_i denote the ith sequence of A. The *fitness* of an individual is definied as:

$$\text{Fitness}(A) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} Cost(A_i, A_j) \tag{1}$$

   where $Cost(A_i, A_j)$ is the pairwise *cost* of A_i and A_j, calculated by *DynamicProgramming* algorithm.

3. *Selection* We randomly drop a subset of the individuals based on their fitness, which means individuals with lower fitnesses are more likely to be selected.

4. *Crossover* For each pair of individuals of the population, we apply *crossover* to them with a certain probability. We use the *one-point crossover* of Ref. [3], which is shown is Fig. 3. The crossover combines two parent alignments through a single exchange. The first parent is cut straight at some randomly chosen position and the second one is tailored so that the right and the left pieces of each parent can be joined together while keeping the original sequence. Any void space that appears at the junction point is filled with *Gap* signs.

5. *Mutation* For each individual of the population, we apply *mutation* to them with a certain probability. We use the *gap insertion* operator of Ref. [3], which is shown in Fig. 2. This

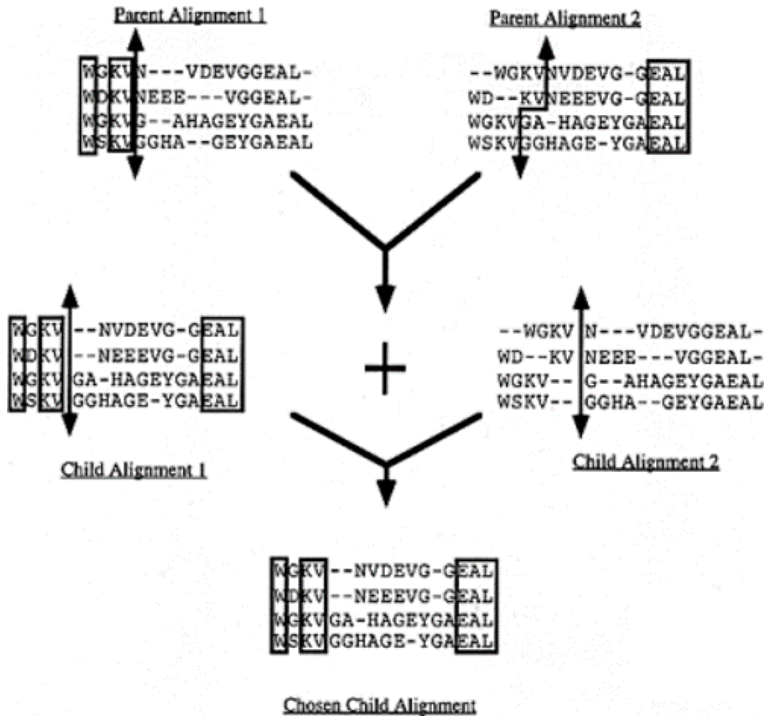

Figure 3: Crossover

operator extends alignments by inserting gaps. To keep the sequences aligned, each sequence will get a gap insertion of the same size. The position to be inserted into the gap is randomly selected.

6. *Goal test* The *genetic algorithm* terminates when the individuals of the population stay unchanged for several generations or the number of evolution reachs a given upper-bound.

In each evolution, we execute *fitness evaluation, selection, crossover, and mutation* in sequence until the condition required by *goal test* is reached.

Completeness and Optimality

The *genetic algorithm* is **not complete** without an upper bound to the number of iterations. Furthermore, the algorithm is **not optimal** but could yield an acceptable solution.

3 Implementation

The three algorithms, *Dynamic Programming algorithm* , *A* search algorithm*, and *Genetic algorithm*, were written in `PyThon` and were implemented in a `Windows 10` system. Some dependent libraries are needed to execute the programs. `copy` is necessary for *A* search* algorithm and `random` is necessary for *genetic algorithm*. `tqdm` and `time` are optional for all three programs, which are used for progress bar modification and execution time recording. Feel free to delete all the relating segments if these decorations make no sense to you.

3.1 Dynamic Programming

All the functions, constructor `__init__`, two functions for pairwise alignment task `pairwiseTask` and `__pairwiseAlign`, and two functions for three-sequence alignment task `triSeqTask` and `__triSeqAlign`, are integrated in a class `DynamicProgramming`.

Constructor

The constructor is used for *data reading and preprocessing*.

Searching Functions

`pairwiseTask` and `triSeqTask` are called *searching functions*, which are used to find the most suitable sequence in the database to match the given query.

For each sequence in the query file, we traverse the database once for pairwise alignment or twice for three-sequence alignment, to find the most *similar* sequence. In each traversal, we call the *alignment function*, which is explained in the next section, to compute the *cost* of the particular alignment.

Alignment Functions

`__pairwiseAlign` and `__triSeqAlign` are the *alignment functions*. They are used to **compute the cost of an alignment**. The input sequences are the strings in the alignment and the output is the *cost* value.

In this section, we explain `__pariwiseAlign` as an example.

Firstly, we allocate a space for matrix $Cost$, the concrete explanation of which is in Section. 2.1. Then we initialize the edge-elements of $Cost$, which means $Cost(\cdot, i) = 2i$ and $Cost(0, i) = 2j, i \in [0, m], j \in [0, n]]$, where m and n are the lengths of the sequences respectively. Finally, we compute the elements from the top left corner to the bottom right corner and we traverse each row from left to right.

3.2 A* Algorithm

The basic structure of the program is identical to that in Section. 3.1, where `AStar` class integrates five functions: `__init__`, `pairwiseTask`, `__pairwiseAlign`, `triSeqTask`, and `__triSeqAlign`. The only difference happens in *alignment functions* so We only explain these differences in this section.

Alignment Functions

The input and output of *alignment functions* are the same as that in Section. 3.1. We use the demo in *tutorial_1* to implement *AStar search* algorithm. The attributes in `A_Star_Node` class, function `expand_node`, `heuristic` and `goal_test` are modified to suit specific tasks according to the formulation in Section. 2.2.

3.3 Genetic Algorithm

The implementation of *genetic algorithm* includes two classes, `Individual` and `Genetic`.

Individual

In `Individual`, we define five *member variables*: `__generation`, `__fitness`, `__maxOffset`, `chromosome`, and `alignLength`. `__generation` and `__fitness` are used to store the generation and fitness of the individual. When initializing the individual, we randomly insert 0 to `__maxOffset` *gap signs* in each sequence header. `chromosome` is an array representing an alignment, each entry of which is a aligning scheme for each sequence. The example of `chromosome` is shown in Fig. . `alignLength` is the maximum alignment length, equalling to the length of the longest sequence.

The only meaningful *menber function* is `computeFitness`, the functionality of which is illustrated by its name. In this function, we traverse all the possible pair of sequences of `chromosome` and compute pairwise *Cost* by *dynamic programming* algorithm, which is explained in Section. 3.1. The other *member functions* are used to get the values of *member variables* for safety.

Genetic

This class is our core for implementing *genetic algorithm*. In the constructor function, we define some hyperparameters of the algorithm, including `population_size, mutation_rate, crossover_rate,` and `max_iteration`. Furthermore, `population` is a list of individuals.

The member functions in the *genetic* class are `init_population, evaluation, selection, cross_over, one_point_crossover, gap_insertion, Iteration`. The functionalities and implementations of them are shown below:

1. `Iteration`: It is the entrance of the class, in which we execute *fitness evaluation, selection, crossover,* and *mutation* in sequence for `max_iteration` times.
2. `init_population`: Initialize the population accroding the `population_size`. For each individual, we randomly add some *gap signs* in each sequence header in range $[0, \texttt{\_\_ maxOffset}]$.
3. `evaluation`: Fitness evaluation. For each individual in `population`, we calculate its fitness by calling `computeFitness`.
4. `selection`: We traverse each individual in `population` and select it with a probability of $1 - \frac{fitness}{sum\ of\ fitness}$. We keep on selecting until the number of selected individuals reaches `population_size`.
5. `cross_over`: The implementation is based on Section. 2.3.
6. `gap_insertion`: This function is equivalent to *mutation*. The implementation is based on Section. 2.3.

4 Result

Table 1: Pairwise Alignment Result

Query number	Solution index	Cost	Time/s
1	21	112	0.76
2	63	107	0.70
3	45	113	0.76
4	18	148	0.82
5	57	131	0.80

Table 2: Three-sequence Alignment Result

Query number	Solution		Cost	Time/h
	Index 1	Index 2		
1	10	70	290	1.85
2	2	46	122	2.07

Dynamic programming algorithm is relatively fast so that optimal solution can be solved in acceptable time. The **optimal solutions**, corresponding costs, and execution time by *Dynamic*

Programming algorithm are shown in Table. 1 and Table. 2 respectively. The execution time is limited in 1 second for pairwise alignment tasks, while that of three-alignment tasks is able 2 hours. *The raw data of time test is shown in Fig. 4 and Fig. 5.*

Figure 4: Pairwise Alignment Raw Test

Figure 5: Three-sequence Alignment Raw Test

The execution of the *A* search* algorithm has not been so smooth. The time complexity of *A* search* algorithm is too high to yield the optimal solution for a single query, not even the whole task. According to my valid estimation, which is done by `tqdm` toolkit, *A* algorithm* takes a few days to run. This is total impractical because my laptop can't spare so much time to execute the program. However, I think I have realized the essence of the this part of the project: *A* search algorithm is completely impractical for a slightly larger solution space.*

Table 3: Pairwise Alignment Result of GA

Query number	Solution index	Time/s
1	48	3414.53
2	28	3749.71
3	45	3995.2
4	**18**	**3949.6**
5	28	3732.8

Although I fail to use *A* search* algorithm to solve the original problem, I have verified the correctness of my implementation with smaller data. The query and database data for correctness validation are in file `Test_query.txt` and `Test_database.txt` respectively. For each query, I compute the cost of the the query and each sequence in the database. The validation program is implemented in function `correctnessValidation` of class `AStar`, and the result of it is stored in file `AStar_test_result.txt`. Meanwhile, I compute the correct costs by executing the program of

dynamic programming algorithm, the result of which is stored in file `DP_test_result.txt`. After comparison, I found that the results of the them are exactly the same, so the implementation

With respect to *genetic algorithm*, I set `crossover_rate = 0.5`, `mutation_rate = 0.4`, `population_size = 10`, and `max_iteration = 15`. These parameters are too small for the algorithm, but I had to set it this way to get the result of pairwise alignment. The result of *genetic algorithm* is shown in Table. 3, and Only the alignment of query 4 converges to the optimal solution while the others are only sub-optimal solutions. For three-sequence alignment, the estimated time for one query is about four days (100 hours), which is impractical for me. The raw data of *genetic algorithm* is saved in file `genetic_result.txt`.

5 Conclusion

Through this project, I realized *dynamic programming*, *A* search* algorithm and *genetic algorithm* to solve MSA problem with a large amount of data. It can be seen from the results that the time complexity of dynamic programming is relatively low, and it usually has strong practical significance. The state space of the *A* search* algorithm is at an exponential level and is impractical on larger data. The complexity of genetic algorithm depends on `population_size` and `max_iteration`, but usually the larger the two parameters, the better the effect. As a result, genetic algorithm has a trade-off between efficiency and effect.

References

[1] Comparing best-first search and dynamic programming for optimal multiple sequence alignment, Hohwald, Heath and Thayer, Ignacio and Korf, Richard E, IJCAI, 1239–1245, 2003.

[2] Artificial intelligence: a modern approach, Russell, Stuart and Norvig, Peter, 2002.

[3] SAGA: sequence alignment by genetic algorithm, Notredame, C'edric and Higgins, Desmond G, Nucleic acids research, 24, 8, 1515–1524, 1996, Oxford University Press

[4] External memory best-first search for multiple sequence alignment, Hatem, Matthew and Ruml, Wheeler, Proceedings of the AAAI Conference on Artificial Intelligence, 27, 1, 2013