Fall 2023,CS 466 Project - Hirschberg Visualization

VIDYA KAMATH PAILODI (VIDYAK2)

ACM Reference Format:
Vidya Kamath Pailodi (vidyak2). 2023. Fall 2023,CS 466 Project - Hirschberg Visualization. 1, 1 (December 2023), 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Sequence alignment involves comparing biological sequences like DNA or proteins to find similarities and differences, which is important for understanding their functions and evolutionary relationships. Moreover, beyond understanding gene functions, sequence alignment serves as a vital tool in disease diagnosis, including cancer detection [9].

The Needleman-Wunsch algorithm [10], while effective in aligning sequences accurately, utilizes quadratic space complexity, making it resource-intensive for longer sequences. On the other hand Hirschberg's algorithm [8] offers a more efficient solution with linear space requirements, enabling alignment of lengthy sequences without excessive memory usage. In Hirschberg's technique, the whole problem is recursively divided into subproblems around the optimal alignment path, while using linear space. Within the recursion, Hirschberg's algorithm uses dynamic programming to find the alignment scores for the subproblem. However, understanding the recursive behavior of [8] can be confusing. These concepts are tricky and can be hard to grasp. This project aims to develop a tool to aid students by providing visualizations for the recursive part of the Hirschberg. In addition to the implementation, this project aims to create visualizations that illustrate the recursive steps within the alignment process by generating a recursive tree for each function call. With this approach the goal is to provide a clear demonstration of the method's operation with recursive calls, aiding in understanding the recursive behavior of Hirschberg. Additionally, the tool also provides the animation for recursive dynamic programming steps and the final reported path using the output from these recursive calls.

1.1 Motivation

There are many interactive tools available for visualizing the Needleman-Wunsch algorithm [1], [5]. However, in the case of Hirschberg's algorithm, a quick Google search leads only to some online content showcasing Hirschberg's visualization through YouTube videos [2] and pictures. Most visualizations only show the final result without detailing the step-by-step process, especially the recursive steps that are crucial for a deeper understanding. Also, a quick online search did not yield any tool offering a detailed, step-by-step view of how Hirschberg's algorithm fills its tables recursively. This gap in available resources motivates the need to create a tool that focuses on showing these recursive steps in a clear and accessible way, aiming to help students better grasp the internals of the algorithm.

Author's address: Vidya Kamath Pailodi (vidyak2).

XXXX-XXXX/2023/12-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 METHODOLOGY

The tool [4] is developed using the Python programming language because the primary objective is to create comprehensive visualizations, and Python has rich visualization libraries, such as matplotlib, making it a good choice for this project's needs. However, for the algorithmic implementation where efficiency is crucial, other languages like C++ might offer better performance and efficiency.

2.1 Datasets

To conduct space complexity analysis, I used randomly generated sequences using the base nucleotides A, T, C, G. The length of the sequences can be configured using input parameters while running the benchmark tests.

2.2 Scoring Function

A scoring function assigns numeric values to matches, mismatches, insertions, and deletions, which are used for evaluating the quality of alignments. Typically, a scoring matrix, such as the BLOSUM matrices for proteins, or substitution matrices for nucleotides in DNA/RNA, defines these values. In this project, I use the scoring matrix as shown in Table 1. A positive score is used for matching pairs similar while mismatches or gaps receive negative scores. The scoring functions are designed in a modular way so that it is easy to extend and use different scoring matrices in the future.

0	A	T	C	G
A	1	-1	-1	-1
T	-1	1	-1	-1
C	-1	-1	1	-1
G	-1	-1	-1	1

Table 1. Scoring matrix showing the cost for matches and mismatches. Also, indel=-1

2.3 Implementation

2.3.1 Needleman Wunsch Qudratic Space Alignment. The Implementation of the Needleman Wunsch algorithm is a slightly modified version of the implementation from the HW1 assignment. Modifications are done to add the visualizations. Below is a brief overview of the steps used in implementing the Needleman Wunsch algorithm for global alignment.

- **Global Alignment:** Find the alignment of the entire length of two sequences, considering matches, mismatches, insertions, and deletions. Dynamic programming is used to construct an alignment matrix where each cell represents the optimal alignment score up to that point. This matrix is filled iteratively using a recurrence relation. Also, a pointers matrix is filled to keep track of the path for traceback. The scoring function discussed in section 2.2 is used to assign scores to matches, mismatches, insertions, and deletions.
- **Global Alignment Score**: The alignment score is determined by the sum of individual scores along the aligned path, which is available in the cell corresponding to the last row and last column in the dynamic programming matrix.
- **Traceback:** After matrix filling, a traceback function is used to reconstruct the aligned sequences by tracing back from the bottom-right cell.

- **Time Complexity:** $O(m \times n)$, where n and m are the lengths of the sequences being aligned. Filling the $m \times n$ matrix requires $O(m \times n)$ operations.
- **Space Complexity:** $O(m \times n)$, as it requires storing the entire alignment matrix.

2.3.2 Hirschberg Linear Space Alignment. To implement the Hirschberg algorithm, an initial function is developed to compute the optimal alignment score with linear space, utilizing only two columns. The pseudo-code for this function is outlined in Algorithm 1. This function efficiently determines the optimal global alignment score. However, for identifying the precise alignment path, a recursive implementation, as taught in class, is implemented. This pseudo-code for recursive methodology is outlined in Algorithm 2. facilitating the comprehensive derivation of the optimal alignment.

The output of this recursive function calls are the cells that are in the optimal alignment path. But, for some of the sequences, the reported vertices will have some intermediate vertices missing. Hence we have to do a post-processing to get the intermediate vertices. To implement this functionality, the *reconstruct_path* function from [3] is reused.

- **Time Complexity:** $O(m \times n)$, where n and m are the lengths of the sequences being aligned. Filling the $m \times n$ matrix requires $O(m \times n)$ operations.
- **Space Complexity:** $O(m)$, as it requires storing only 2 columns of the alignment matrix to compute the score.

Algorithm 1: Compute prefix/suffix score in Linear Space

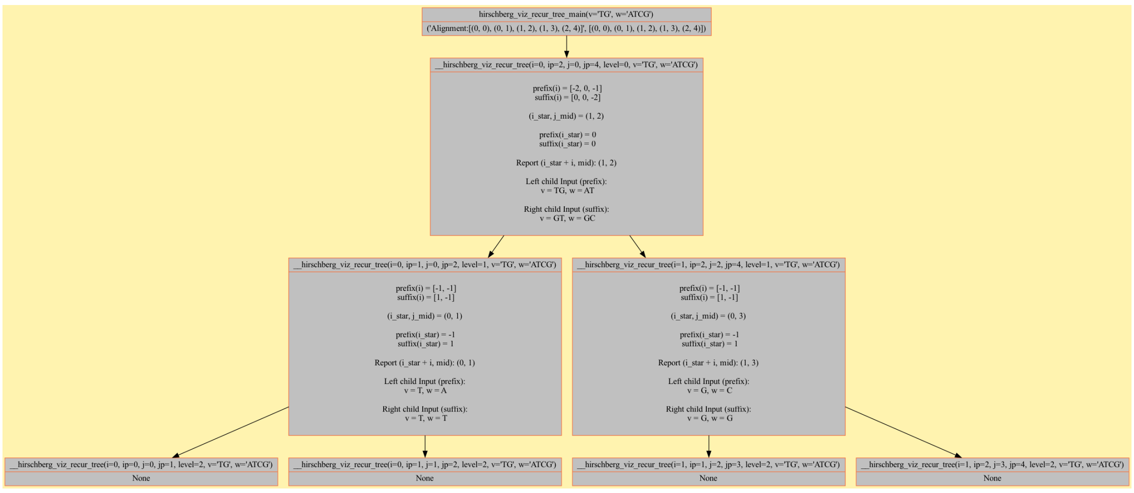
```
Input  : Strings $v$, $w$, Dictionary $\delta$
Output : Score $score$, prefix/suffix Column $lastColumn$
$m \leftarrow$ length of $v$;
$n \leftarrow$ length of $w$;
$M \leftarrow$ 2D array of size $(m+1) \times 2$ initialized with zeros;
$col \leftarrow 0$;
for $j \leftarrow 0$ to $n$ do
    for $i \leftarrow 0$ to $m$ do
        if $i == 0$ and $j == 0$ then
            $M[i][\text{col}] \leftarrow 0$;
        end
        else if $i == 0$ and $j > 0$ then
            $M[i][\text{col}] \leftarrow M[i][1-\text{col}] + \delta[\text{"}-\text{"}][w[j-1]]$;
        end
        else if $i > 0$ and $j == 0$ then
            $M[i][\text{col}] \leftarrow M[i-1][\text{col}] + \delta[v[i-1]][\text{"}-\text{"}]$;
        end
        else
            $matchOrMismatch \leftarrow M[i-1][1-\text{col}] + \delta[v[i-1]][w[j-1]]$;
            $deletion \leftarrow M[i-1][\text{col}] + \delta[v[i-1]][\text{"}-\text{"}]$;
            $insertion \leftarrow M[i][1-\text{col}] + \delta[\text{"}-\text{"}][w[j-1]]$;
            $M[i][\text{col}] \leftarrow \max(matchOrMismatch, deletion, insertion)$;
        end
    end
    $col \leftarrow 1 - \text{col}$;
end
$lastColumn \leftarrow$ list of elements from the last column of $M$;
$score \leftarrow M[m][1-\text{col}]$;
return $score, lastColumn$;
```

Algorithm 2: Hirschberg Algorithm Recursive Function

```
Input :Strings $v$, $w$, Integers $i$, $j$, $ip$, $jp$, List $res$, Function $score\_func$
1  Function __hirschberg($v, w, i, j, ip, jp, res, score\_func$):
       /* Base case                                                    */
2      if $jp - j == 1$ then
3          $res.append((i, j))$;
4          $res.append((ip, jp))$;
5          return;
6      end
7      if $jp - j < 1$ then
8          return;
9      end
       /* Middle column to split                                       */
10     $mid \leftarrow$ int$((j + jp)/2)$;
11     prefix_score, prefix
         $\leftarrow$ compute_score_lin_space($v[i : ip :], w[j : mid :], score\_func$) suffix_score, suffix $\leftarrow$
         compute_score_lin_space($v[i : ip :][:: -1], w[mid : jp :][:: -1], score\_func$)
12     $maxidx \leftarrow -1$;
13     $maxweight \leftarrow$ float("$-\infty$");
14     for $idx \leftarrow 0$ to len(prefix) do
15         $weight \leftarrow$ prefix$[idx]$ + suffix$[-idx - 1]$;
16         if $maxidx == -1$ or $maxweight \leq weight$ then
17             $maxidx \leftarrow$ idx;
18             $maxweight \leftarrow$ weight;
19         end
20     end
       /* Report the cell on the traceback path                        */
21     $res.append((maxidx + i, mid))$;
       /* Left subproblem                                              */
22     __hirschberg($v = v, w = w, i = i, j = j, ip = maxidx + i, jp = mid, res = res, score\_func =$
         $score\_func$);
       /* Right subproblem                                             */
23     __hirschberg($v = v, w = w, i = maxidx + i, j = mid, ip = ip, jp = jp, res = res, score\_func =$
         $score\_func$);
```

2.4 Visualization

To visualize the step-by-step execution of the Hirschberg Algorithm, two visualization methods were developed. Firstly, to generate an image file showcasing the recursion tree, illustrating variable values within each recursive call. These variables encompass the two strings, prefix and suffix scores, and the reported cells defining the optimal alignment path (as depicted in Figure 1). This visualization was facilitated using the recursion-visualizer [7] Python package, which was easily integrated by adding a decorator to the recursive function call. However, the challenge was in carefully implementing the recursive function to capture all intermediate variable values. To address this, a new **HirschbergViz** class was designed.



```
hirschberg_viz_recur_tree_main(v='TG', w='ATCG')
('Alignment:[(0, 0), (0, 1), (1, 2), (1, 3), (2, 4)]', [(0, 0), (0, 1), (1, 2), (1, 3), (2, 4)])

__hirschberg_viz_recur_tree(i=0, ip=2, j=0, jp=4, level=0, v='TG', w='ATCG')
prefix(i) = [-2, 0, -1]
suffix(i) = [0, 0, -2]
(i_star, j_mid) = (1, 2)
prefix(i_star) = 0
suffix(i_star) = 0
Report (i_star + i, mid): (1, 2)
Left child Input (prefix):
v = TG, w = AT
Right child Input (suffix):
v = GT, w = GC

__hirschberg_viz_recur_tree(i=0, ip=1, j=0, jp=2, level=1, v='TG', w='ATCG')
prefix(i) = [-1, -1]
suffix(i) = [1, -1]
(i_star, j_mid) = (0, 1)
prefix(i_star) = -1
suffix(i_star) = 1
Report (i_star + i, mid): (0, 1)
Left child Input (prefix):
v = T, w = A
Right child Input (suffix):
v = T, w = T

__hirschberg_viz_recur_tree(i=1, ip=2, j=2, jp=4, level=1, v='TG', w='ATCG')
prefix(i) = [-1, -1]
suffix(i) = [1, -1]
(i_star, j_mid) = (0, 3)
prefix(i_star) = -1
suffix(i_star) = 1
Report (i_star + i, mid): (1, 3)
Left child Input (prefix):
v = G, w = C
Right child Input (suffix):
v = G, w = G

__hirschberg_viz_recur_tree(i=0, ip=0, j=0, jp=1, level=2, v='TG', w='ATCG')
None

__hirschberg_viz_recur_tree(i=0, ip=1, j=1, jp=2, level=2, v='TG', w='ATCG')
None

__hirschberg_viz_recur_tree(i=1, ip=1, j=2, jp=3, level=2, v='TG', w='ATCG')
None

__hirschberg_viz_recur_tree(i=1, ip=2, j=3, jp=4, level=2, v='TG', w='ATCG')
None
```

Fig. 1. Recursion tree showing the recursive calls and the intermediate variables for aligning sequences $v = TG$, $w = ATCG$ using the Hirschberg Algorithm. (Input sequences are the same as from HW1 written assignment). Also, an animation showing the order of the recursive calls is generated and is available in the repository.

The second visualization involved creating a GIF (Graphical Image Format) showcasing the sequence of recursive function calls. The distinctive aspect of this project lies in this visualization, where the recursive steps of Hirschberg prune the Dynamic Programming table to identify cells along the optimal alignment path. A custom function utilizing *matplotlib* was implemented to generate the animation depicting this recursive process. More output artifacts can be found in the GitHub repository [4]. The different frames of the animation are shown in the figure 2. Figure 3 shows the final dynamic programming table generated by Needleman-Wunsch Algorithm.

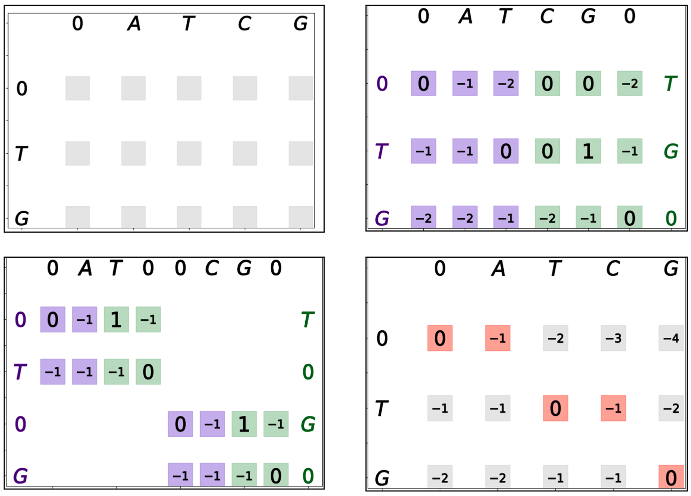
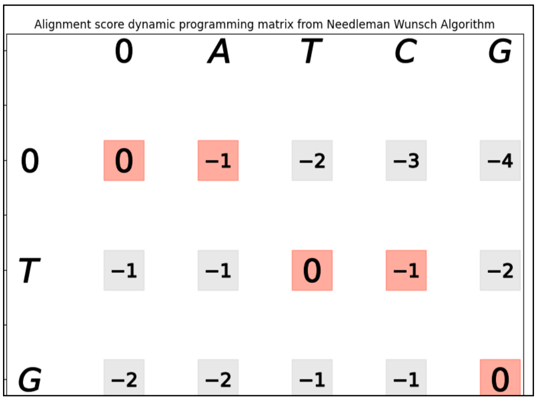


Fig. 2. Images (Clockwise from Left) generated at different stages of the recursive call of the Hirschberg Algorithm to compute the optimal alignment path. The same set of images are combined to form an animation and can be found in the repository.


Alignment score dynamic programming matrix from Needleman Wunsch Algorithm

$$\begin{array}{c|ccccc} & 0 & A & T & C & G \\ \hline 0 & 0 & -1 & -2 & -3 & -4 \\ T & -1 & -1 & 0 & -1 & -2 \\ G & -2 & -2 & -1 & -1 & 0 \end{array}$$

Fig. 3. Alignment score matrix generated by Needleman-Wunsch Algorithm. We can see that for sequences $v = TG$, and $w = ATCG$ the optimal alignment path and the score are the same.

3 EVALUATION

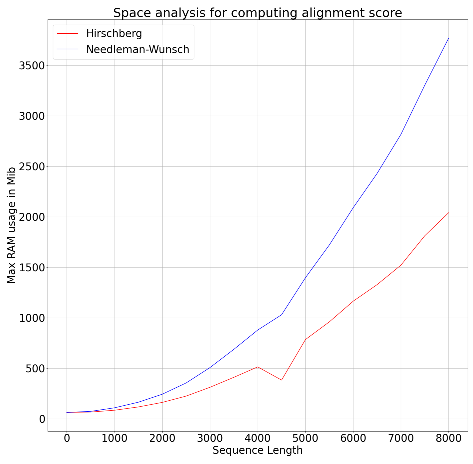
3.1 Space and Time complexity analysis

The benchmark experiments were conducted on a Linux system equipped with 8GB of RAM. The assessment centered on randomly generated DNA sequences spanning lengths from 0 to 8000. Space efficiency was gauged by measuring the maximum RAM consumption during the alignment function call, using the Python package $memory - profiler$ [6]. Additionally, wall clock time was recorded using the python's built-in $time$ module for computing the alignment score and constructing the actual alignment. For more insights into benchmarking scripts and results, please refer to the GitHub repository [4].

The analysis consisted of two scenarios. The first scenario involved determining the optimal alignment score using both Quadratic and linear space strategies. The second scenario involved the computation of the actual alignment. As seen in figure 4 we can see that across both evaluations, the Hirschberg algorithm consistently showcases reduced space usage compared to the Needleman-Wunsch algorithm. Moreover, from figure 5 we can see that the execution time for both algorithms is approximately the same thereby affirming their similar time complexities. The Hirschberg algorithm shows a slightly higher execution time for computing the alignment (second image in figure 5). This might be because of the post-processing needed for finding the missing intermediate cells on the optimal alignment path. These observations underline the space efficiency of Hirschberg while confirming the comparable time complexities between the two algorithms.

4 CORRECTNESS EVALUATION

To ensure the accuracy of the Hirschberg implementation, tests were conducted on shorter sequences ranging in length from 0 to 10. The scores generated by Hirschberg were compared with those produced by the Needleman-Wunsch algorithm. However, direct comparison of alignments wasn't feasible due to instances where multiple optimal alignments exist, leading to potential variations between the algorithms. Nonetheless, specific scenarios were manually verified, and alignments from both algorithms showed consistency. Although space constraints prevent the inclusion of these results in the report, they are available for reference in the Github repository [4].



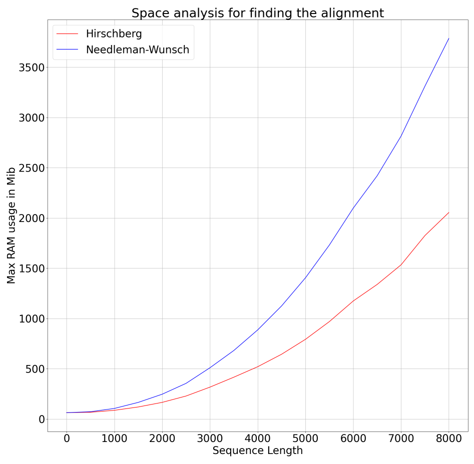
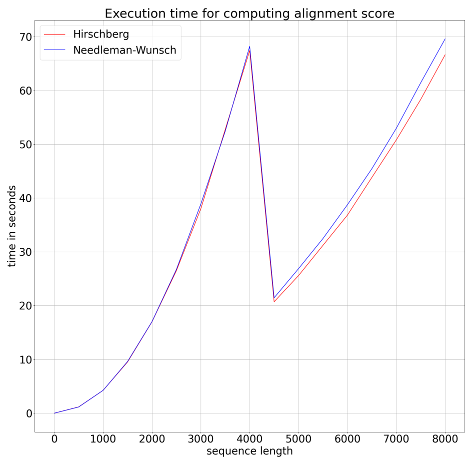


Fig. 4. Space analysis showing the maximum RAM usage on the y-axis as a function of input length sequence on the x-axis. The two plots represent the space usage for computing the optimal score and space usage for finding the alignment.



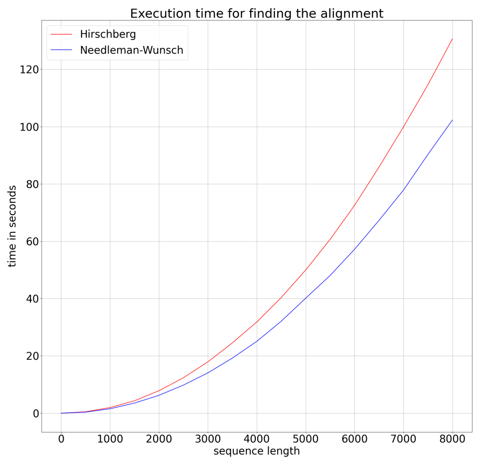


Fig. 5. Execution time showing the time in seconds on the y-axis as a function of input length sequence on the x-axis. The two plots represent the wall clock time for computing the optimal score and the wall clock time for finding the alignment.

5 CONCLUSION AND FUTURE WORK

This project implements the Hirschebrg Linear Space alignment algorithm and introduces an approach to visualize the recursive aspects of the Hirschberg algorithm through recursive trees and optimal path-finding visualizations. Operating as a command-line tool, it easily processes two sequences as inputs. As part of future work, I would like to expand this implementation to accommodate protein sequences and custom scoring functions. The required adaptations primarily involve the path reconstruction code only and the existing visualization code should work. Additionally, enhancing the visualization could involve animating the computation of prefix and suffix scores, highlighting the utilization of just two columns instead of directly showcasing filled tables. Another addition could be to support other types of alignments such as local, fitting alignment. To conclude, this work has made a significant contribution to generating recursive visualizations that can be easily enhanced with little modifications to support other types of input sequences and scoring functions.

REFERENCES

[1] [n. d.]. *Alignment Visualizer.* https://valiec.github.io/AlignmentVisualizer/index.html

[2] [n. d.]. *DPA for sequence alignment using Hirschberg's technique.* https://www.youtube.com/watch?v=cPQeJt-2Y1Q

[3] [n. d.]. *Hirschberg Path Reconstruction.* https://github.com/jialic2/CS466-Hirschberg-s-Algorithm

[4] [n. d.]. *Hirschberg Recursion Visiualizer.* https://github.com/VidyaKamath/HirschViz

[5] [n. d.]. *Interactive demo for Needleman-Wunsch algorithm.* http://experiments.mostafa.io/public/needleman-wunsch/

[6] [n. d.]. *Pyhton memory profiler.* https://stackify.com/top-5-python-memory-profilers/

[7] [n. d.]. *recursion-visualizer.* https://pypi.org/project/recursion-visualiser/r

[8] Daniel S. Hirschberg. 1975. A linear space algorithm for computing maximal common subsequences. *Commun. ACM* 18, 6 (1975), 341–343.

[9] Neil C. Jones and Pavel A. Pevzner. 2004. *An Introduction to Bioinformatics Algorithms.* MIT Press, Cambridge, MA.

[10] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4