CS 466 Final Project Report

Member: Roger Ho
NetID: pingche2
Github Link: https://github.com/TeemosCode/UIUC_CS466_Introduction_to_Bioinformatics_Final_project_Hirschberg_Visualization

Introduction

This Final project revolves around the previous final project proposal. In the previous Project Proposal, based on my very own personal and many other students' experience when learning about the Hirschberg algorithm in class, especially as we looked at the slides (Lecture5 Page 10 ~ Page 23) and HW1's Hirschberg questions. Even many were still confused of the actual internal workings of Hirschberg after writing the HW and discussing in class before the midterms as well as on piazza.

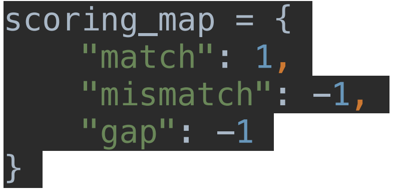
Going through the learning processes as well as discussion with many other students, I realized it is due to the high level nature that most students are unable to fully grasp hold of the idea, actual step by step process from the algorithm. Making matters worse, there isn't much resources online to help students know more about each working steps of this brilliant algorithm (A lot of students especially get confused about the calculations of the weight(i) of i* in the slides as well as its combination of prefix(i) + suffix(i), mostly are even more confused with the transposing of suffix matrix, its calculation and final back tracing steps in action).

As I really enjoyed the class (UIUC CS466 Introduction to Bioinformatics: Taught by Professor: Mohammed El-Kebir) and its contents, I hope that I could not only learn from the process of working on the final project, but also to give back to the bioinformatics community, especially the class if possible. Thus, my high level idea is to not only implement the Hirschberg algorithm, just to have others see the outputs of final alignments, but to really enable users/students/learners to see each step of the algorithm, and gain much more insight into the algorithm while learning.

The original proposal (and what I really hoped could exist online to assist those new to Hirschberg like myself) was to create something very very fancy that I believe would absolutely help those new comers to the field. A single web page application hosted online with inputs of two different DNA sequences, v & w (variable names like in the slides and HW1) as well as a scoring matrix/function, then after clicking a "run" button, there could be a series of pictures and blocks showing each Hirschberg DP recurrence and its sub trees, as well as all corresponding prefix(i)/suffix(i) values and list of their values (transposition/flipping/reverse, etc) in order. However, I soon realized that as a lone member on the project, studying the class notes and implementing the Algorithm itself was quite a challenge already. Furthermore, the fancy idea quickly seemed impossible to accomplish by myself in a few weeks after many of my original designs/

research (of tools) implementations and trials, where I originally tried to implement things in Unity and javascript where it could be easily integrated with web frameworks like express.js, D3.js and React.js/Redux.js to fully fulfill my ambition.

But after all the trials, planning, testing and implementations, I decided to tone it down a bit and focus on providing another kind of visualization through the terminal using the Python programming language. Where the users simply provide two inputs of DNA sequences, and a Python dictionary of scoring matrix/functions, such as the following Figure 1.



```
scoring_map = {
    "match": 1,
    "mismatch": -1,
    "gap": -1
}
```

Figure 1

Users can just simply run a method to see all intermediate calculations on the terminal of the Hirschberg algorithm. This is all used to simulate the original fancy idea of my implementation so that any future work, where I will discus in detail in the Future Work section, could be more easily transferred and implemented.

Design & Implementation

Since I another major goal of my final project is to also aim at making this as educational as possible, and hope it could be adopted by the professor for future teachings to CS/Bio students, I designed a Python Hirschberg class to implement its functionalities based solely on the lectures of the course and teaching of the professor.

In order to make this whole project relate to the course students and resonate with the course as much as possible, I implemented the algorithm closely based on the the slides (Lecture5) and HW1 (Question3). All naming conventions, layout of the recurrence and Dynamic Programming columns were all exactly/highly identical to those used in the HW and Slides. Furthermore, all knowledge to the project implementations were only based on what was taught and presented in this class, as I did not consult to any online resources (Google) in order to reflect the teachings of this course as closely as possible.

Furthermore, in order to have (I sincerely hope) this project easily updated, tested, refactored, further developed or transformed, I modularized every possible actions of the Hirschberg as much as possible by applying the Structured programming and Object Oriented Programming design patterns in software engineering. Actions of the algorithm were split into many methods of the Hirschberg class. Making it easy for development/modification and unit testing. As I hoped this is also educational, the variables are named as closely to the course materials as possible, the code is also filled with comments explaining logistics and usage in order to assist those that read the code.

The design of the implementation is to have one Python class named Hirschberg, where it has the following fields:

- ***v***: Takes user input string that would be on the row of the DP matrix with 'i' as its index (As the same as HW and slides)
- ***w***: Takes user input string that would be on the column of the DP matrix with 'j' as its index (As the same as HW and slides)

(Internally, the code logic would insert a character of "0" in front of both input string sequences to reflect the starting of the Hirschberg Algorithm with (0,0) as its starting point)

The following three *_score fields (The asterisk sign being a place holder with three values: match, mismatch, gap) are derived from user provided Python Dictionary map that serves as a scoring function for the algorithm. The Dictionary must be presented in the with three string data type keys: "match", "mismatch", "gap" where the values need to be integers.

- ***match_score***: Reads in user input of two character's matching score from the scoring matrix (Python Dictionary) provided by the user as specified previously in Figure 1.
- ***mismatch_score***: Reads in user input of two character's mismatching score from the scoring matrix (Python Dictionary) provided by the user as specified previously in Figure 1.
- ***gap_score***: Reads in user input of two character's gap score from the scoring matrix (Python Dictionary) provided by the user as specified previously in Figure 1.

- ***alignment_tuples_list***: A list that is used to save the coordinates presented in integer tuples of (i, j), ('i' indicates the index of the row (string v) and 'j' indicates the index of column (string w) as indicated in the slides) calculated by each Hirschberg recursion. The tuples would be sorted based on the second element of each tuple (the j index, meaning it is sorted based on the columns, where j = 0 ~ j = len(w))
- ***final_alignment_tuples_list***: A list that saves the tuples that presents the final global alignment with the maximum weight/score path for the two input sequences. The tuples are in the form of (i, j, s) where 'i' and 'j' mean the coordinate of the Dynamic Programming matrix as in the slides of v and w sequences, where 'i' is the index of the input v and 'j' is the index of input w. The third element in the tuple should hold the score/weight of the of the coordinate within the path. Current implementation does not need it, but as I thought about future developments and improvements could be made to even further visualize the weights of each point within the constructed path, I just left it here for future changes. It's current values are all filled in with 0. This would also include the tuples that would need to be post processed from the Hirschberg algorithm when two reported coordinates within neighboring gaps are not exactly one unit diagonally to each other, nor are they side by side, linked together on the path.

- ***final_alignment_strings_list***: List containing two global alignment strings, the first element is the v's alignment where the second element is input w's alignment. The strings are constructed based on the final alignment tuples.
- ***column_dp***: A list of two lists that is used to run each space efficient Dynamic Programming within the Hirschberg algorithm of just constantly keeping only two columns within the memory to calculate the prefix(i) and suffix(i) of the argmax weight. The first list element of the list represents all calculated row scores of the previous column, where the second list element of column_dp list is used to update the score of all rows of the current column. This is used throughout the calculations using dynamic programming in a linear-space manner.
- ***backtrace_steps_map***: This is currently just a place holder of a Python dictionary data type, intended for further future work where it could be assigned with step numbers of the Hirschberg as its keys and tuples of Hirschberg function names or step names for smarter rendering and visualization when this is transformed into the ideal web application.

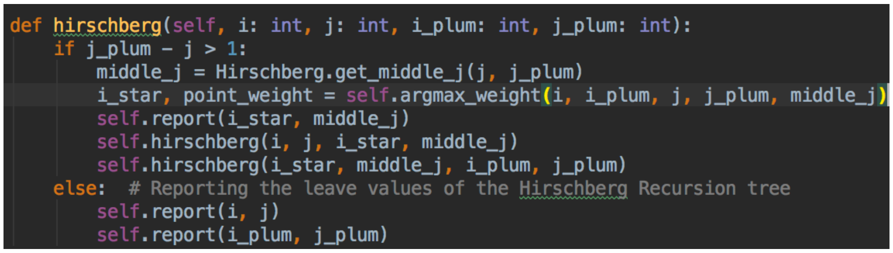
The class also have various fine-grained modularized methods, each working towards connecting the dots to perform the entire Hirschberg algorithm. The following briefly explains each method based on the order of the algorithm holistic operation stack:

- ***run()***: This method is simply called once users have created their Hirschberg object with the correct sequence inputs and Dictionary of the scoring matrix. This is the entry point to running the Hirschberg algorithm, where each various other methods are called and where each intermediate steps and final results would be printed on the terminal.
- ***hirschberg(i: int, j: int, i_plum: int, j_plum: int)***: The star of this program. It is constructed in the exact way as the pseudo code presented in the course slides of Lecture5, Page 17 as shown in Figure 2.
- ***get_middle_j(j, j_plum)***: The naming of the parameters are exactly as those in the pseudo of the lecture5 page 22. This is used to find the n/2, the middle column of the current Hirschberg problem to rum prefix and suffix calculations as well as dividing the problem into sub problems within the recurrence and recursion.
- ***argmax_weight(i, i_plum, j, j_plum, middle_j)***: The same function as in the pseudo code presented in the course slide in Figure 3. Where it is used to run the code of dynamic programming to calculate the prefix(i) and suffix(i) column scores in search of i* value after adding both prefix(i) and suffix(i) score together. It would return the i* value once it is calculated, being the index that generates maximum number of the two added column numbers. (The tie breaker I used here is to get the first value that is the largest. There could be many other tie breaking and could easily programming and refactored into this method if necessary).
- ***run_column_dp(row_index_start, row_index_end, column_index_start, column_index_end, middle_j, prefix_option=True)***: This is the method used to run all the logistics of the space efficient dynamic programming of Hirschberg. Where it would use the aforementioned field - column_dp to update each row of a column from its corresponding starting point to its ending point based on the passed in

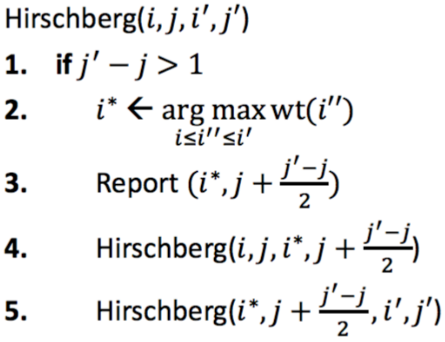
parameters. The starting point and ending points for rows, regardless of if this method is used to run for calculating prefix(i) or suffix(i) would be: row_index_start to row_index_end. The starting point and ending point for the columns would be different based on the last option - prefix_option, which by default is True, meaning unless specified, this method would be used to run and calculate prefix. If the prefix_option were given a False value, then the method would know to run dynamic programming to calculate the suffix. When calculating prefix, the starting and ending column for the method would be: column_index_start and middle_j. Whereas for calculating suffix, the starting point and ending points for the column of the dynamic programming would be: column_index_end to middle_j + 1 (Not overlapping the middle_j calculated value as specified and taught during class, before the midterm review and on piazza post: https://piazza.com/class/jzlp5lwaouu2ou?cid=78). In order to achieve the idea of transposing the suffix matrix, I did it by simply reversing both string slices needed to calculate the transposed matrix before running dynamic programming logistics on them. Since each Hirschberg run would be a sub problem within its recursion, and the starting point and ending points would change during each run, it makes it quite tricky to match the dynamic programming matrix (Visualization of slides: Lecture5, page:22) indexes to each sequence input's character for during different subproblem run. In order to make things easier and understanding to the users, I simply just sliced the strings based on the starting and ending points of each subproblem run provided in the arguments. Append a "0" character to the front of those strings to match the dynamic programming matrix strings for weight calculation in the examples of the course materials.

- ***theta_score(self, char1: str, char2: str, option="matching")***: This method is important and used to generate the scores for two different characters if they were to be aligned. The option parameter is used to determine if it should return a gap score. If it is provided as "matching" as by default, it would then compare both characters, char1, char2 and determine if they are the same or not to return a value of mismatch or match specified in the user provided score_map.
- ***report(self, i_optimal: int, j_optimal: int, point_weight=None)***: Used to save the current coordinate of the optimal 'i' and 'j' index for both sequence input of their global alignment into the alignment_tuples_list for further processing.
- ***post_process_construct_hirschberg_alignment(self, alignment_tuples_list: list)***: This method is called after all Hirschberg algorithm is finished to go through all coordinates generated by Hirschberg, and find out how one coordinate connects to the other one if both coordinates of column j and j - 1 are not directly diagonal to each other of side by side. It would go through all tuples (coordinates) within the alignment_tuples_list (Since in the lecture, we were taught that this post processing would be done from the very end of the coordinates, thus the alignment_tuple_list would be reversed based on the column index ('j'/second element within the tuple)), for each tuple, it looks at the tuple before it finds the best possible path to the tuple (coordinate) before it. All these new generated tuple (coordinate) points would be appended to the final_alignment_tuples_list including the initial coordinates in between, then it would finally be reversed for later usage of constructing the actual string representations of the final global alignment.

- ***reconstruct_alignment(self, current_tuple: tuple, prev_tuple: tuple)***: Called by the above method, its usage is to use another local defined function - dfs() to generate all possible walks of one coordinate (current_tuple) to its previous coordinate (prev_tuple), find the one that would link both coordinates generated by the Hirschberg algorithm and return them back as list.
- ***generate_final_alignment_strings_list()***: This method is used to go through the final_alignment_tuples_list once all intermediary coordinates are calculated, it would then regenerate the two optimal global alignment of the inputs in a list of string, The first element of the list being the alignment of v, where the second element would be the alignment of w.

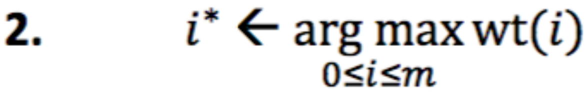


```
def hirschberg(self, i: int, j: int, i_plum: int, j_plum: int):
    if j_plum - j > 1:
        middle_j = Hirschberg.get_middle_j(j, j_plum)
        i_star, point_weight = self.argmax_weight(i, i_plum, j, j_plum, middle_j)
        self.report(i_star, middle_j)
        self.hirschberg(i, j, i_star, middle_j)
        self.hirschberg(i_star, middle_j, i_plum, j_plum)
    else:  # Reporting the leave values of the Hirschberg Recursion tree
        self.report(i, j)
        self.report(i_plum, j_plum)
```



```
Hirschberg$(i, j, i', j')$
1. if $j' - j > 1$
2.     $i^* \leftarrow \arg\max_{i \le i'' \le i'} \mathrm{wt}(i'')$
3.     Report $(i^*, j + \frac{j'-j}{2})$
4.     Hirschberg$(i, j, i^*, j + \frac{j'-j}{2})$
5.     Hirschberg$(i^*, j + \frac{j'-j}{2}, i', j')$
```

Figure 2

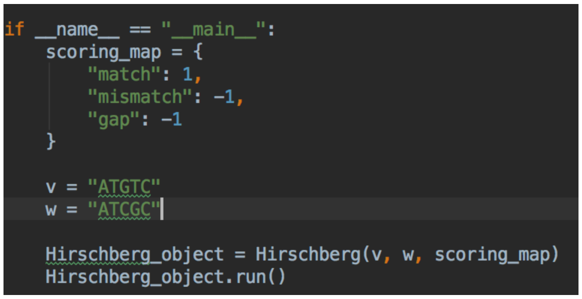


$$2. \qquad i^* \leftarrow \arg\max_{0 \le i \le m} \mathrm{wt}(i)$$

Figure 3

Running the Program & Course Materials Reflection Results

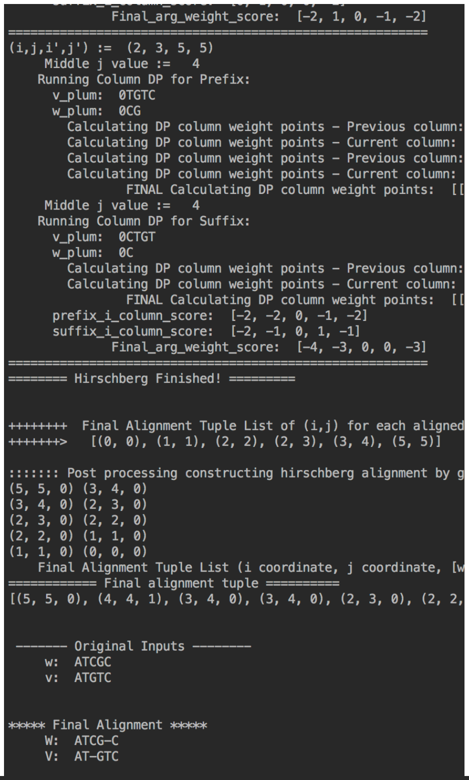
To run the program, a user just need to first specify a scoring map (Python dictionary with keys “match”, “mismatch” and “gap” with values as its scores as integers) and two input string sequences. Then the user then just initialize the Hirschberg object and evoke the run() method, as shown in Figure 4. Then all intermediary process and final values would then be shown on the terminal, as shown in Figure 5.



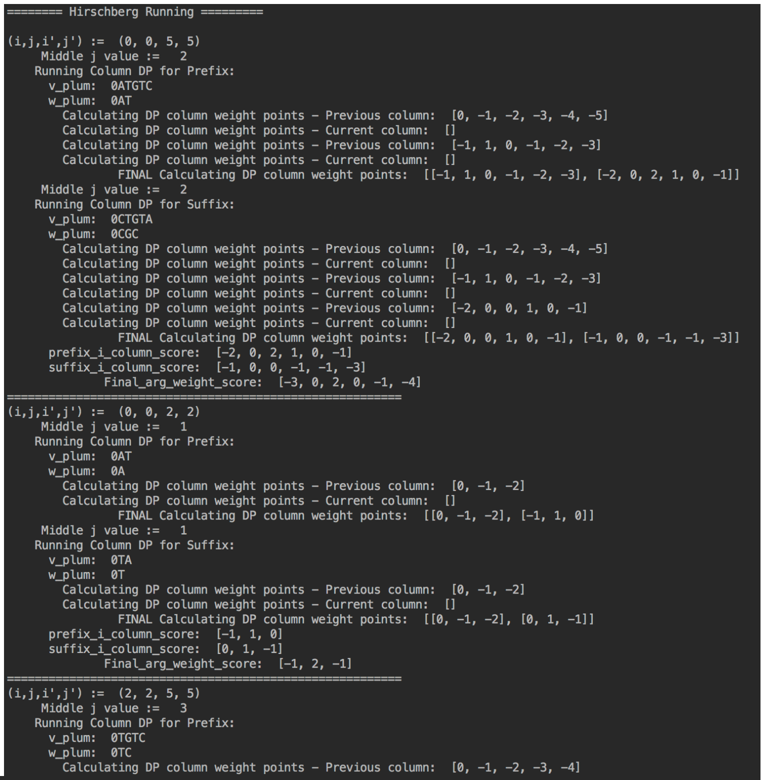
```
if __name__ == "__main__":
    scoring_map = {
        "match": 1,
        "mismatch": -1,
        "gap": -1
    }

    v = "ATGTC"
    w = "ATCGC"

    Hirschberg_object = Hirschberg(v, w, scoring_map)
    Hirschberg_object.run()
```

Figure 4

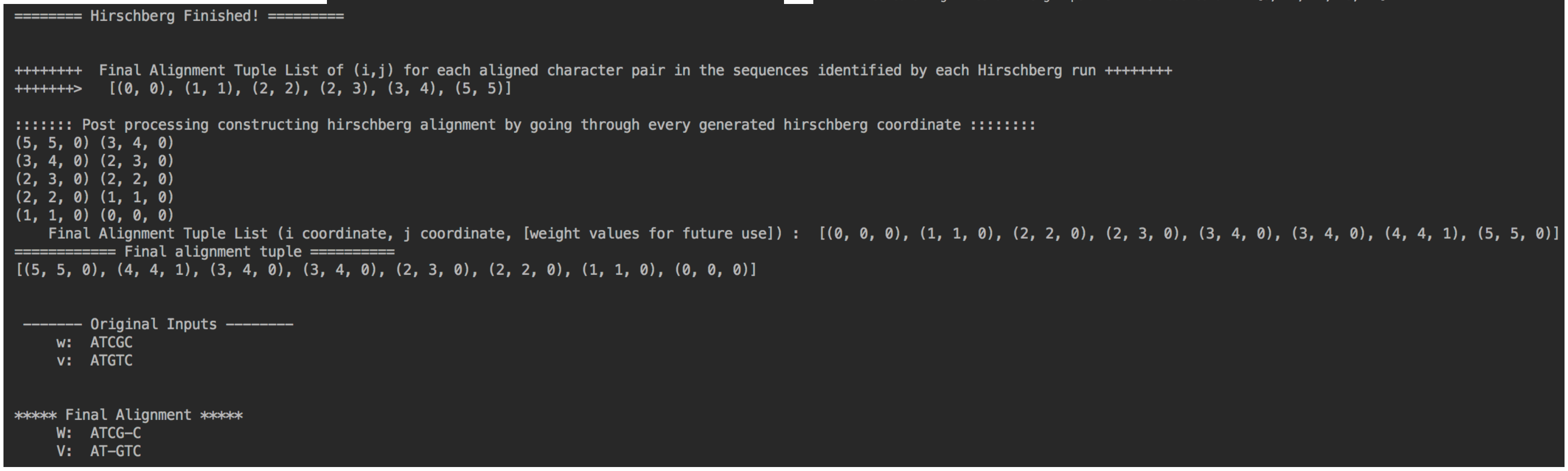
Figure 5



```
            Final_arg_weight_score:  [-2, 1, 0, -1, -2]
=========================================================
(i,j,i',j') :=  (2, 3, 5, 5)
     Middle j value :=   4
    Running Column DP for Prefix:
      v_plum:  0TGTC
      w_plum:  0CG
        Calculating DP column weight points - Previous column:
        Calculating DP column weight points - Current column:
        Calculating DP column weight points - Previous column:
        Calculating DP column weight points - Current column:
                FINAL Calculating DP column weight points:  [[
     Middle j value :=   4
    Running Column DP for Suffix:
      v_plum:  0CTGT
      w_plum:  0C
        Calculating DP column weight points - Previous column:
        Calculating DP column weight points - Current column:
                FINAL Calculating DP column weight points:  [[
      prefix_i_column_score:  [-2, -2, 0, -1, -2]
      suffix_i_column_score:  [-2, -1, 0, 1, -1]
            Final_arg_weight_score:  [-4, -3, 0, 0, -3]
=========================================================
======== Hirschberg Finished! =========


++++++++  Final Alignment Tuple List of (i,j) for each aligned
+++++++>   [(0, 0), (1, 1), (2, 2), (2, 3), (3, 4), (5, 5)]

::::::: Post processing constructing hirschberg alignment by g
(5, 5, 0) (3, 4, 0)
(3, 4, 0) (2, 3, 0)
(2, 3, 0) (2, 2, 0)
(2, 2, 0) (1, 1, 0)
(1, 1, 0) (0, 0, 0)
    Final Alignment Tuple List (i coordinate, j coordinate, [w
============ Final alignment tuple ==========
[(5, 5, 0), (4, 4, 1), (3, 4, 0), (3, 4, 0), (2, 3, 0), (2, 2,


 ------- Original Inputs --------
     w:  ATCGC
     v:  ATGTC


***** Final Alignment *****
     W:  ATCG-C
     V:  AT-GTC
```



```
======== Hirschberg Running =========

(i,j,i',j') :=  (0, 0, 5, 5)
     Middle j value :=   2
    Running Column DP for Prefix:
      v_plum:  0ATGTC
      w_plum:  0AT
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4, -5]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2, -3]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0, -1, -2, -3], [-2, 0, 2, 1, 0, -1]]
     Middle j value :=   2
    Running Column DP for Suffix:
      v_plum:  0CTGTA
      w_plum:  0CGC
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4, -5]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2, -3]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-2, 0, 0, 1, 0, -1]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-2, 0, 0, 1, 0, -1], [-1, 0, 0, -1, -1, -3]]
      prefix_i_column_score:  [-2, 0, 2, 1, 0, -1]
      suffix_i_column_score:  [-1, 0, 0, -1, -1, -3]
            Final_arg_weight_score:  [-3, 0, 2, 0, -1, -4]
=========================================================
(i,j,i',j') :=  (0, 0, 2, 2)
     Middle j value :=   1
    Running Column DP for Prefix:
      v_plum:  0AT
      w_plum:  0A
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2], [-1, 1, 0]]
     Middle j value :=   1
    Running Column DP for Suffix:
      v_plum:  0TA
      w_plum:  0T
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2], [0, 1, -1]]
      prefix_i_column_score:  [-1, 1, 0]
      suffix_i_column_score:  [0, 1, -1]
            Final_arg_weight_score:  [-1, 2, -1]
=========================================================
(i,j,i',j') :=  (2, 2, 5, 5)
     Middle j value :=   3
    Running Column DP for Prefix:
      v_plum:  0TGTC
      w_plum:  0TC
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4]
```



```
======== Hirschberg Finished! =========


++++++++  Final Alignment Tuple List of (i,j) for each aligned character pair in the sequences identified by each Hirschberg run ++++++++
+++++++>   [(0, 0), (1, 1), (2, 2), (2, 3), (3, 4), (5, 5)]

::::::: Post processing constructing hirschberg alignment by going through every generated hirschberg coordinate ::::::::
(5, 5, 0) (3, 4, 0)
(3, 4, 0) (2, 3, 0)
(2, 3, 0) (2, 2, 0)
(2, 2, 0) (1, 1, 0)
(1, 1, 0) (0, 0, 0)
    Final Alignment Tuple List (i coordinate, j coordinate, [weight values for future use]) :  [(0, 0, 0), (1, 1, 0), (2, 2, 0), (2, 3, 0), (3, 4, 0), (3, 4, 0), (4, 4, 1), (5, 5, 0)]
============ Final alignment tuple ==========
[(5, 5, 0), (4, 4, 1), (3, 4, 0), (3, 4, 0), (2, 3, 0), (2, 2, 0), (1, 1, 0), (0, 0, 0)]


 ------- Original Inputs --------
     w:  ATCGC
     v:  ATGTC


***** Final Alignment *****
     W:  ATCG-C
     V:  AT-GTC
```

As we can see in the top right corner of Figure 5, once the Hirschberg starts running, each process of its i, j, i', j' inputs would be printed on the screen in sections, where its middle_j value and its following prefix, suffix dynamic calculations would be printed out step by step. Then final the final value as well as the i* value would all then be shown for each block. Final when the algorithm finishes, it shows out the final alignment tuple (coordinate, the first element being index 'i' of v sequence, the second element being index 'j' of w sequence) list. Following would be showing the post processing construction of the alignment of going through all Hirschberg coordinates where the final total alignment coordinates would be printed. Finally the answers would be printed out on the terminal.

Since I aimed to have this project be as helpful and educational as possible to the course, I would present the algorithm running on the two main materials used within this course to teach Hirschberg. One would be the Slide of Lecture 5, page 22. The other would be the HW question 3 about Hirschberg, especially it is used to present all intermediary steps and the recursion tree. The results are shown in the Figure 6 at the end of the report.

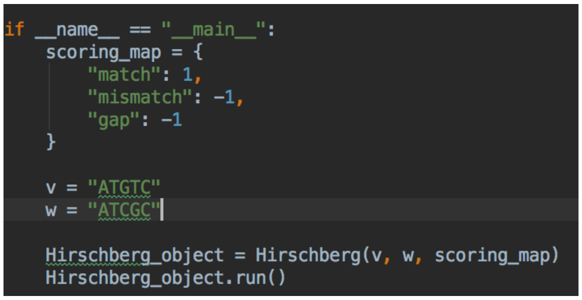
Conclusion & Future work

I have learned a lot about Hirschberg during the implementation. And since this project is built with helping others as the main goal in mind, it is relatively simply to add more “visualization” into the code by adding prints into the code. And as specified above in the course reflection, the code and visualization on the terminal are worked closely with reflecting the two materials, especially with the HW, in hopes that future students could find this insightful and educational as it can closely resonate with the course context. However, there are still some bugs that could be fixed, for example since I use a DFS algorithm to generate all possible paths between two Hirschberg coordinates in python during the post processing procedure, python would run into “RecursionError: maximum recursion depth exceeded in comparison” as the Cpython compiler isn’t very good at optimizing tail recursions. Thus would be better to improve it with iteratively in code.

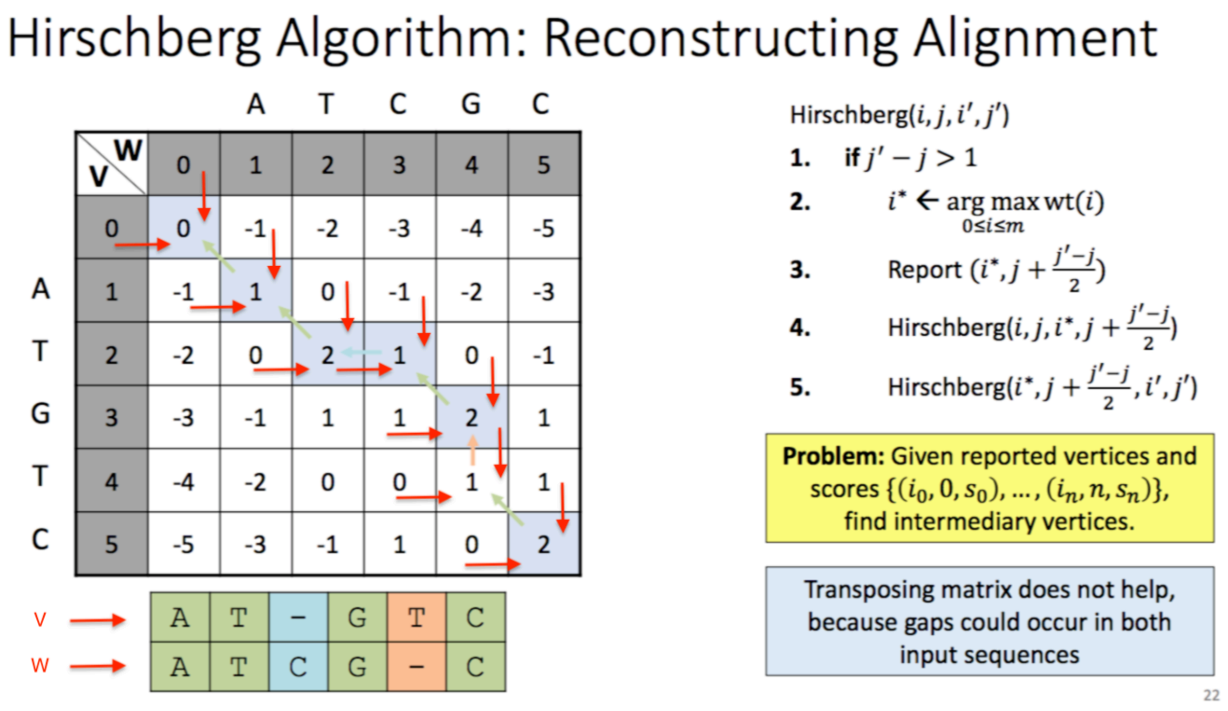
There are many steps to take to make this into my ideal version of a visualization and educational work. First would be to fix the minor bugs, then one could take advantage of the modularization to build it into an interactive program to learn about the Hirschberg algorithm. I have invested a lot of time in hoping this could be further enhanced in the future, thus there are parts within the program where comments are proceeded with 2 #, meaning they are alternative code snippets that the program could change and use to give itself more functionality. I would still like to work on this into the future and hope on day it could become an interactive web application that could be easily searched online by students like us when learning about this amazing algorithm. Maybe rewriting all the code in Javascript as another alternative or continue with using the Python programming language with other web frameworks and web technologies such as Flask, Django, HTML, CSS, etc.

A great thanks to Professor Mohammed El-Kebir and current TA, Ashwin Ramesh making this course enjoyable.

Figure 6 (More below)



```
if __name__ == "__main__":
    scoring_map = {
        "match": 1,
        "mismatch": -1,
        "gap": -1
    }

    v = "ATGTC"
    w = "ATCGC"

    Hirschberg_object = Hirschberg(v, w, scoring_map)
    Hirschberg_object.run()
```



Hirschberg Algorithm: Reconstructing Alignment

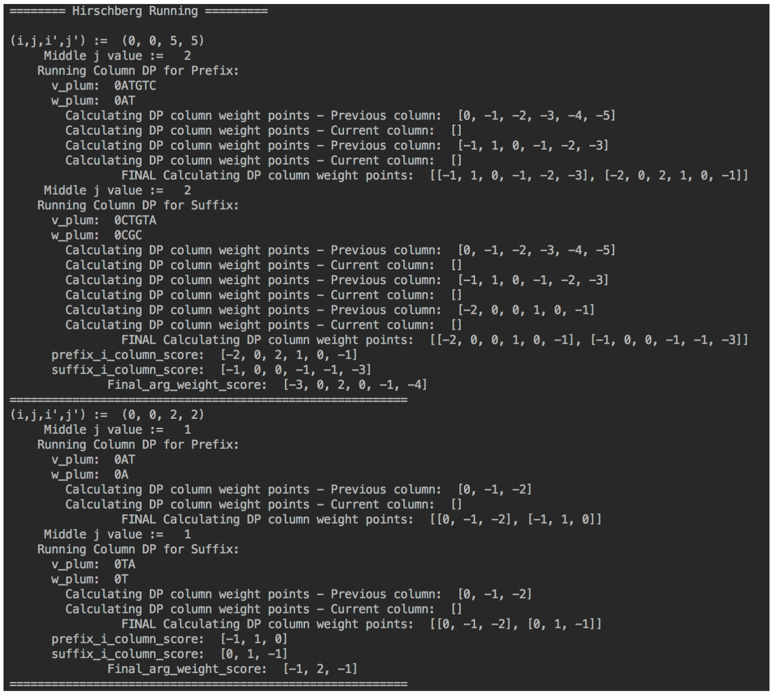
		A	T	C	G	C
	V\W	0	1	2	3	4	5
	0	0	-1	-2	-3	-4	-5
A	1	-1	1	0	-1	-2	-3
T	2	-2	0	2	1	0	-1
G	3	-3	-1	1	1	2	1
T	4	-4	-2	0	0	1	1
C	5	-5	-3	-1	1	0	2

V	A	T	–	G	T	C
W	A	T	C	G	–	C

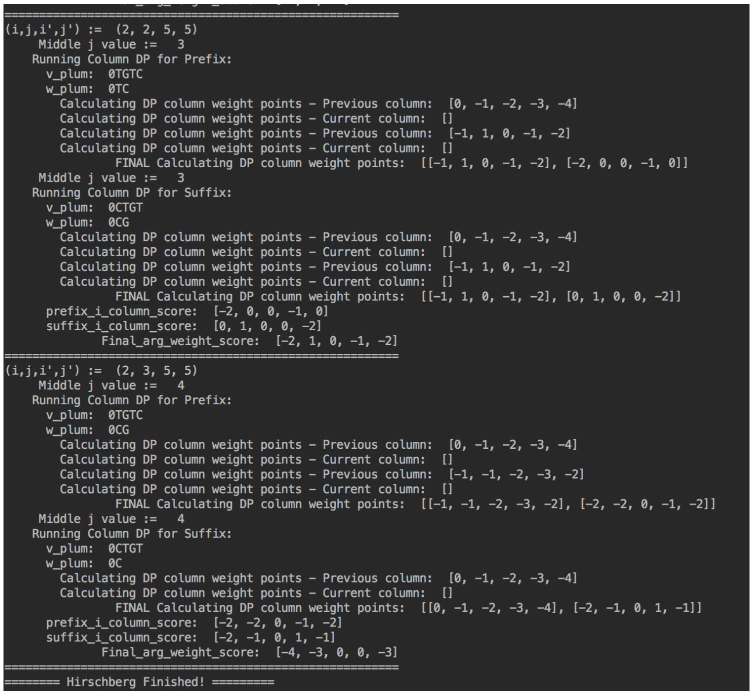
```
Hirschberg($i, j, i', j'$)
1.  if $j' - j > 1$
2.      $i^* \leftarrow \arg\max_{0 \le i \le m} \mathrm{wt}(i)$
3.      Report $(i^*, j + \frac{j'-j}{2})$
4.      Hirschberg($i, j, i^*, j + \frac{j'-j}{2}$)
5.      Hirschberg($i^*, j + \frac{j'-j}{2}, i', j'$)
```

Problem: Given reported vertices and scores $\{(i_0, 0, s_0), \ldots, (i_n, n, s_n)\}$, find intermediary vertices.

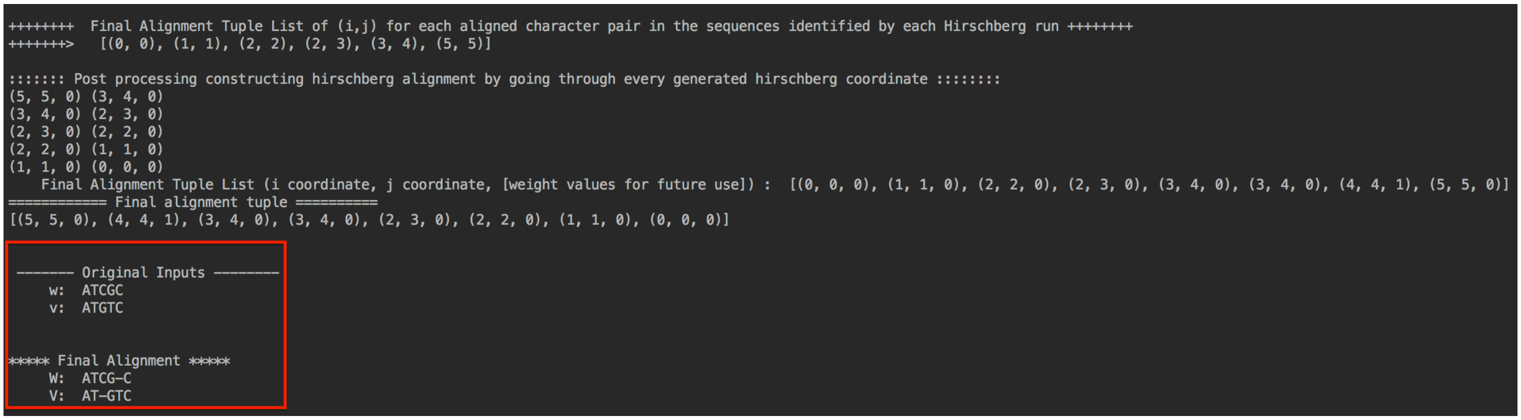
Transposing matrix does not help, because gaps could occur in both input sequences



```
======== Hirschberg Running =========

(i,j,i',j') :=  (0, 0, 5, 5)
     Middle j value :=   2
    Running Column DP for Prefix:
      v_plum:  0ATGTC
      w_plum:  0AT
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4, -5]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2, -3]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0, -1, -2, -3], [-2, 0, 2, 1, 0, -1]]
     Middle j value :=   2
    Running Column DP for Suffix:
      v_plum:  0CTGTA
      w_plum:  0CGC
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4, -5]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2, -3]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-2, 0, 0, 1, 0, -1]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-2, 0, 0, 1, 0, -1], [-1, 0, 0, -1, -1, -3]]
      prefix_i_column_score:  [-2, 0, 2, 1, 0, -1]
      suffix_i_column_score:  [-1, 0, 0, -1, -1, -3]
              Final_arg_weight_score:  [-3, 0, 2, 0, -1, -4]
==========================================================
(i,j,i',j') :=  (0, 0, 2, 2)
     Middle j value :=   1
    Running Column DP for Prefix:
      v_plum:  0AT
      w_plum:  0A
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2], [-1, 1, 0]]
     Middle j value :=   1
    Running Column DP for Suffix:
      v_plum:  0TA
      w_plum:  0T
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2], [0, 1, -1]]
      prefix_i_column_score:  [-1, 1, 0]
      suffix_i_column_score:  [0, 1, -1]
              Final_arg_weight_score:  [-1, 2, -1]
==========================================================
```



```
==========================================================
(i,j,i',j') :=  (2, 2, 5, 5)
     Middle j value :=   3
    Running Column DP for Prefix:
      v_plum:  0TGTC
      w_plum:  0TC
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0, -1, -2], [-2, 0, 0, -1, 0]]
     Middle j value :=   3
    Running Column DP for Suffix:
      v_plum:  0CTGT
      w_plum:  0CG
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0, -1, -2], [0, 1, 0, 0, -2]]
      prefix_i_column_score:  [-2, 0, 0, -1, 0]
      suffix_i_column_score:  [0, 1, 0, 0, -2]
              Final_arg_weight_score:  [-2, 1, 0, -1, -2]
==========================================================
(i,j,i',j') :=  (2, 3, 5, 5)
     Middle j value :=   4
    Running Column DP for Prefix:
      v_plum:  0TGTC
      w_plum:  0CG
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, -1, -2, -3, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, -1, -2, -3, -2], [-2, -2, 0, -1, -2]]
     Middle j value :=   4
    Running Column DP for Suffix:
      v_plum:  0CTGT
      w_plum:  0C
        Calculating DP column weight points - Previous column:  [0, -1, -2, -3, -4]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2, -3, -4], [-2, -1, 0, 1, -1]]
      prefix_i_column_score:  [-2, -2, 0, -1, -2]
      suffix_i_column_score:  [-2, -1, 0, 1, -1]
              Final_arg_weight_score:  [-4, -3, 0, 0, -3]
==========================================================
======== Hirschberg Finished! =========
```



```
++++++++  Final Alignment Tuple List of (i,j) for each aligned character pair in the sequences identified by each Hirschberg run ++++++++
+++++++>   [(0, 0), (1, 1), (2, 2), (2, 3), (3, 4), (5, 5)]

::::::: Post processing constructing hirschberg alignment by going through every generated hirschberg coordinate ::::::::
(5, 5, 0) (3, 4, 0)
(3, 4, 0) (2, 3, 0)
(2, 3, 0) (2, 2, 0)
(2, 2, 0) (1, 1, 0)
(1, 1, 0) (0, 0, 0)
    Final Alignment Tuple List (i coordinate, j coordinate, [weight values for future use]) :  [(0, 0, 0), (1, 1, 0), (2, 2, 0), (2, 3, 0), (3, 4, 0), (3, 4, 0), (4, 4, 1), (5, 5, 0)]
============ Final alignment tuple ==========
[(5, 5, 0), (4, 4, 1), (3, 4, 0), (3, 4, 0), (2, 3, 0), (2, 2, 0), (1, 1, 0), (0, 0, 0)]


 ------- Original Inputs --------
    w:  ATCGC
    v:  ATGTC


***** Final Alignment *****
    W:  ATCG-C
    V:  AT-GTC
```

3. **Linear Space Alignment** [10 points]

Consider two sequences $\mathbf{v}$ = CT and $\mathbf{w}$ = GCAT of length $m = |\mathbf{v}| = 2$ and $n = |\mathbf{w}| = 4$, respectively. In this exercise, we will compute an optimal global alignment of the two sequences using the Hirschberg algorithm. We will use a score of +1 for a match, -1 for a mismatch, and -1 for a insertion/deletion (i.e. a gap penalty of 1).

a. The initial call is HIRSCHBERG$(0, 0, m = 2, n = 4)$. We need to identify the middle vertex $(i^*, n/2 = 2)$. Fill out the following table for this initial call and indicate i^*. [2 points]

i	prefix(i)	suffix(i)	wt(i)
0	-2	0	-2
1	0	0	0
2	-1	-2	-3

Rubric:

- -1 for each wrong column (prefix or suffix only)

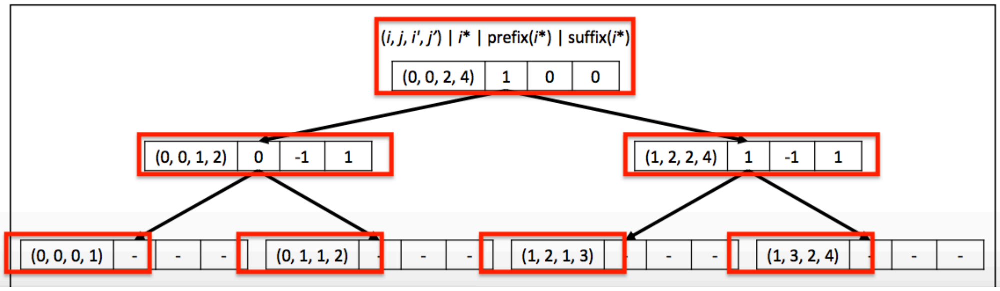
b. What are the two recursive calls that are made in this initial invocation HIRSCHBERG$(0, 0, m, n)$? [1 point]

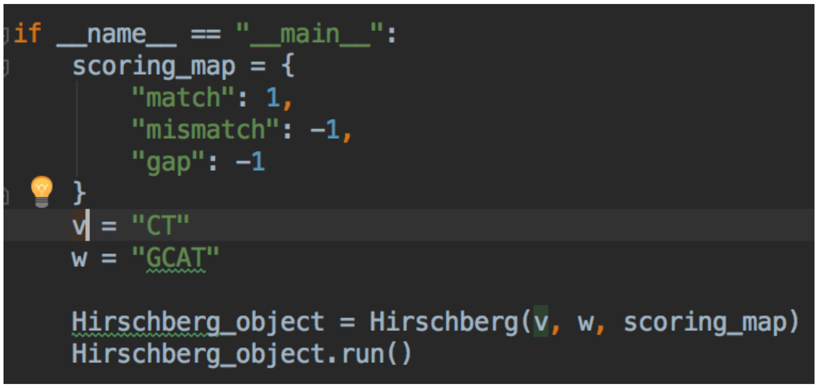
HIRSCHBERG$(0, 0, 1, 2)$ and HIRSCHBERG$(1, 2, 2, 4)$

Rubric:

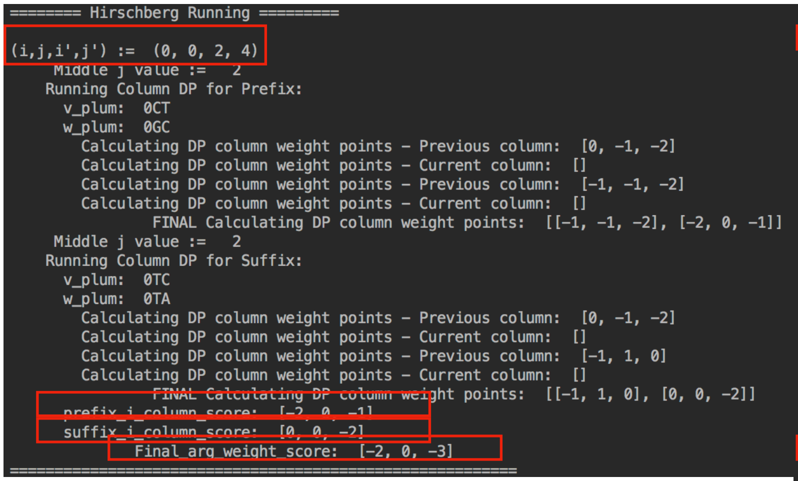
- -0.5 for each wrong call

c. Give the recursion tree, where each vertex corresponds to an invocation of HIRSCHBERG. See Lecture 1 for an example of a recursion tree. Label each vertex of this tree by the used arguments (i, j, i', j'). In addition, label each *internal* vertex by the value of i^*, prefix(i^*) and suffix(i^*). [5 points]

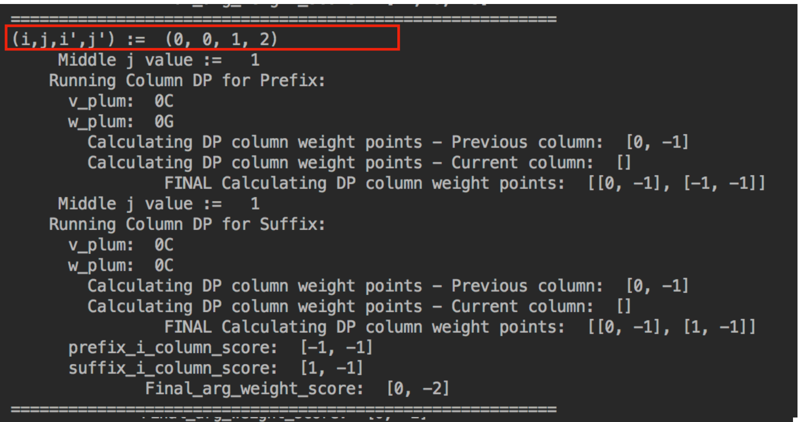




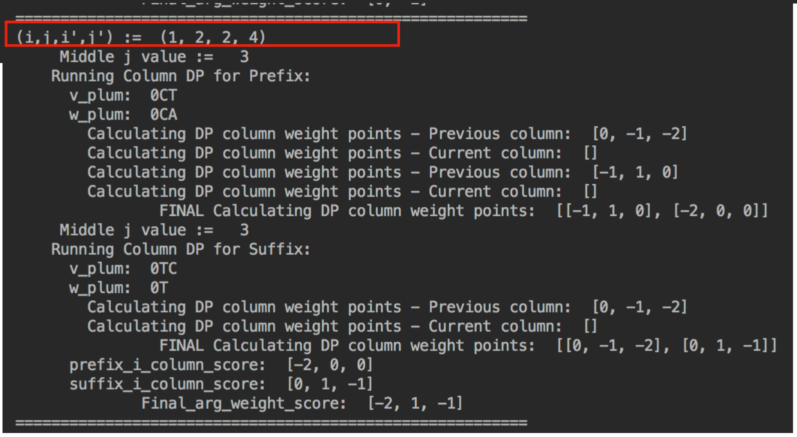
```python
if __name__ == "__main__":
    scoring_map = {
        "match": 1,
        "mismatch": -1,
        "gap": -1
    }
    v = "CT"
    w = "GCAT"

    Hirschberg_object = Hirschberg(v, w, scoring_map)
    Hirschberg_object.run()
```



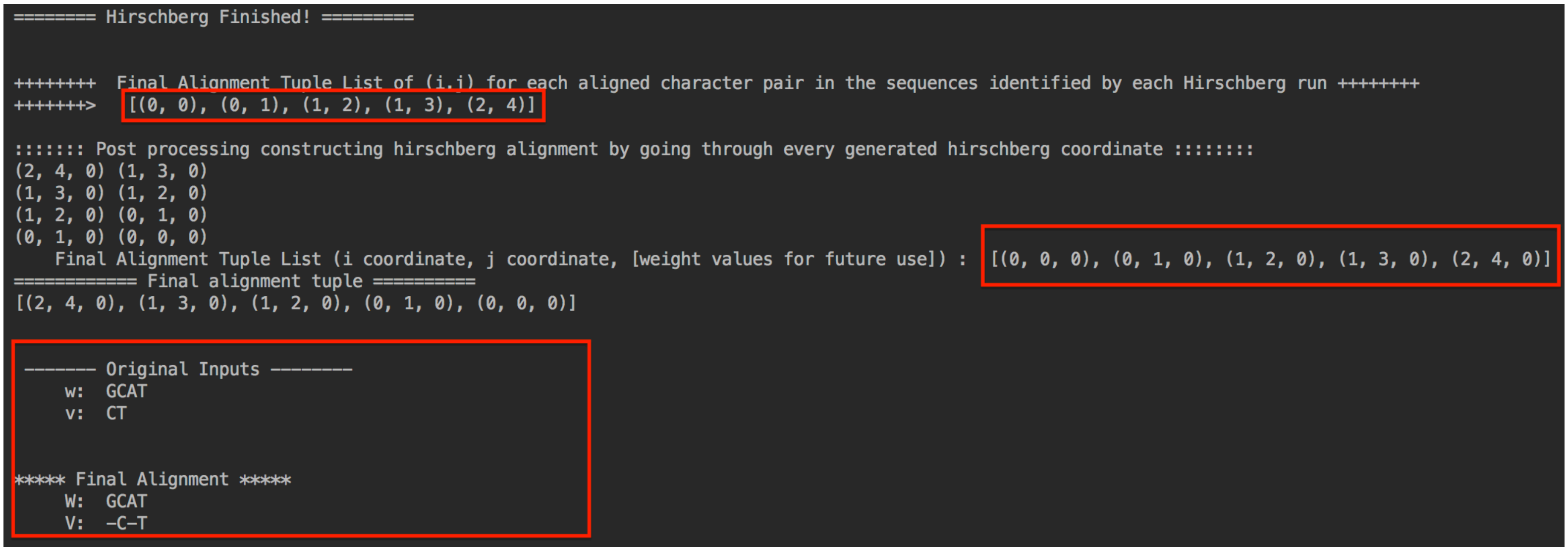
```
======== Hirschberg Running =========

(i,j,i',j') :=  (0, 0, 2, 4)
     Middle j value :=   2
    Running Column DP for Prefix:
      v_plum:  0CT
      w_plum:  0GC
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, -1, -2], [-2, 0, -1]]
     Middle j value :=   2
    Running Column DP for Suffix:
      v_plum:  0TC
      w_plum:  0TA
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0], [0, 0, -2]]
      prefix_i_column_score:  [-2, 0, -1]
      suffix_i_column_score:  [0, 0, -2]
              Final_arg_weight_score:  [-2, 0, -3]
==========================================================
```



```
==============================================================
(i,j,i',j') :=  (0, 0, 1, 2)
     Middle j value :=   1
    Running Column DP for Prefix:
      v_plum:  0C
      w_plum:  0G
        Calculating DP column weight points - Previous column:  [0, -1]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1], [-1, -1]]
     Middle j value :=   1
    Running Column DP for Suffix:
      v_plum:  0C
      w_plum:  0C
        Calculating DP column weight points - Previous column:  [0, -1]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1], [1, -1]]
      prefix_i_column_score:  [-1, -1]
      suffix_i_column_score:  [1, -1]
              Final_arg_weight_score:  [0, -2]
==============================================================
```



```
==========================================================
(i,j,i',j') :=  (1, 2, 2, 4)
     Middle j value :=   3
    Running Column DP for Prefix:
      v_plum:  0CT
      w_plum:  0CA
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
        Calculating DP column weight points - Previous column:  [-1, 1, 0]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[-1, 1, 0], [-2, 0, 0]]
     Middle j value :=   3
    Running Column DP for Suffix:
      v_plum:  0TC
      w_plum:  0T
        Calculating DP column weight points - Previous column:  [0, -1, -2]
        Calculating DP column weight points - Current column:  []
                FINAL Calculating DP column weight points:  [[0, -1, -2], [0, 1, -1]]
      prefix_i_column_score:  [-2, 0, 0]
      suffix_i_column_score:  [0, 1, -1]
              Final_arg_weight_score:  [-2, 1, -1]
==========================================================
```



```
======== Hirschberg Finished! =========


++++++++  Final Alignment Tuple List of (i,j) for each aligned character pair in the sequences identified by each Hirschberg run ++++++++
+++++++>  [(0, 0), (0, 1), (1, 2), (1, 3), (2, 4)]

::::::: Post processing constructing hirschberg alignment by going through every generated hirschberg coordinate :::::::
(2, 4, 0) (1, 3, 0)
(1, 3, 0) (1, 2, 0)
(1, 2, 0) (0, 1, 0)
(0, 1, 0) (0, 0, 0)
    Final Alignment Tuple List (i coordinate, j coordinate, [weight values for future use]) :  [(0, 0, 0), (0, 1, 0), (1, 2, 0), (1, 3, 0), (2, 4, 0)]
============ Final alignment tuple ==========
[(2, 4, 0), (1, 3, 0), (1, 2, 0), (0, 1, 0), (0, 0, 0)]


 ------- Original Inputs ---------
     w:  GCAT
     v:  CT


***** Final Alignment *****
     W:  GCAT
     V:  -C-T
```